PostgreSQL — одна из самых популярных в мире баз данных с открытым исходным кодом, которая поддерживает самые передовые возможности, включенные в стандарты SQL. Данная книга познакомит вас с последними новациями, появившимися в PostgreSQL 10.

Прочитав книгу, вы будете хорошо понимать основы PostgreSQL 10 и обладать навыками, необходимыми для разработки эффективных решений с применением этой базы данных. С той или иной степенью полноты книга охватывает практически все вопросы, с которыми встречается разработчик и администратор, начинающий профессионально работать с данной СУБД.

Издание рекомендовано ведущими разработчиками PostgreSQL в России, оно будет полезно как начинающим разработчикам, так и действующим администраторам.

Краткое содержание книги:

- основы реляционных баз данных, реляционной алгебры и моделирования данных;
- установка кластера PostgreSQL, создание базы данных и реализация модели данных;
- создание таблиц и представлений, построение индексов, определение триггеров, хранимых функций и других объектов схемы;
- манипулирование данными с помощью языка SQL;
- реализация бизнес-логики на стороне приложения с помощью триггеров и хранимых функций на языке PL/pgSQL;
- дополнительные типы данных, поддерживаемые PostgreSQL 10: массивы, JSONB и другие;
- разработка OLAP-решений с применением нововведений, появившихся в PostgreSQL 10;
- эффективное программирование базы данных на языке Python;
- тестирование кода, хранимого в базе данных, нахождение узких мест, повышение производительности и надежности приложений.

Изучаем PostgreSQL 10

Салахалдин Джуба, Андрей Волков

Изучаем PostgreSQL 10

Руководство для начинающих по созданию высокопроизводительных решений для базы данных PostgreSQL

Интернет -магазин:
www.dmkpress.com
Книга - почтой:
e-mail: orders@alians-kniga.ru
Оптовая продажа:
«Альянс-книга»
Тел/факс (499)782-3889
e-mail: books@alians-kniga.ru

Packt

ДМК издательство

ISBN 978-5-97060-643-8

Салахалдин Джуба, Андрей Волков

Изучаем PostgreSQL 10

Salahaldin Juba, Andrey Volkov

Learning PostgreSQL 10

Second Edition

A beginner's guide to building high-performance PostgreSQL database solutions

Packt
BIRMINGHAM - MUMBAI

Салахалдин Джуба, Андрей Волков

Изучаем PostgreSQL 10

Второе издание

Руководство для начинающих по разработке высокопроизводительных решений на основе СУБД PostgreSQL

Москва, 2019

УДК 004.655
ББК 32.973.26-018.2
Д40

Джуба С., Волков А.
Д40 Изучаем PostgreSQL 10 / пер. с анг. А. А. Слинкина. – М.: ДМК Пресс, 2019. – 400 с.: ил.

ISBN 978-5-97060-643-8

Прочитав книгу, вы будете хорошо понимать основы PostgreSQL 10 и обладать навыками, необходимыми для разработки эффективных решений с применением базы данных. Это хорошее пособие для близкого знакомства с PostgreSQL. С той или иной степенью полноты оно охватывает практически все вопросы, с которыми встречается разработчик и администратор, начинающий профессионально работать с этой СУБД.

Издание рекомендовано ведущими разработчиками PostgreSQL в России, оно будет полезно как начинающим разработчикам, так и действующим администраторам этой СУБД.

УДК 004.655
ББК 32.973.26-018.2

ISBN 978-1-78839-201-3 (анг.)
ISBN 978-5-97060-643-8 (рус.)

Благодарность от редакции

Издательство благодарит Ивана Евгеньевича Панченко за помощь в подготовке этого издания.

Эта книга – хорошее пособие для близкого знакомства с PostgreSQL. С той или иной степенью полноты она охватывает практически все вопросы, с которыми встречается разработчик и администратор, начинающий профессионально работать с этой СУБД. А то, что в книге не нашлось, есть в документации на русском языке: https://postgrespro.ru/docs.

Приятного погружения в Postgres!

Содержание

Об авторах

Салахалдин Джуба более десяти лет работает в промышленности и академических учреждениях, уделяя основное внимание разработкам приложений баз данных в крупномасштабных и корпоративных системах. Имеет степень магистра по управлению природоохранной деятельностью, а также степень бакалавра по проектированию компьютерных систем. Является сертифицированным разработчиком программных решений на основе продуктов Microsoft (MCSD).

Работал преимущественно с базами данных SQL Server, PostgreSQL и Greenplum. Разрабатывал для научных сообществ приложения, связанные с распределенной обработкой географической информации, во время работы в академических учреждениях участвовал во многих международных проектах и разработке стандартов, имеющих отношение к обработке изображений.

Работая программистом, занимался в основном определением ETL-процессов обработки данных, полученных извне, постановкой программных задач, популяризацией передовых практик работы с SQL, проектированием приложений OLTP и OLAP, исследованием и оценкой новых технологий, преподавательской деятельностью и консультированием.

Выражаю глубочайшую благодарность своему коллеге Андрею Волкову, без которого эта книга не состоялась бы. Также благодарю всех, кто оказывал поддержку, а особенно коллектив издательства Packt за советы, корректуру и замечания по оформлению книги. Хочу также поблагодарить свою семью, которая не оставляла меня заботами, хотя я был вынужден уделять много внимания книге в ущерб близким. И наконец, самые теплые слова и глубокая признательность моему покойному отцу, Икраему Джубе, который всегда поддерживал меня, помогал и наставлял в жизни.

Андрей Волков изучал банковские информационные системы. Свою карьеру начал финансовым аналитиком в коммерческом банке. Базы данных были его основным рабочим инструментом, и скоро он понял, что прямые запросы к базе данных и мастерское владение SQL гораздо эффективнее визуальных отчетов, когда нужно произвести ситуативный анализ. Он перешел в группу хранилищ данных и спустя некоторое время возглавил ее, заняв должность архитектора хранилищ данных.

Работал в основном с Oracle и занимался разработкой логических и физических моделей финансовых и бухгалтерских данных, реализацией их с помощью СУБД, разработкой ETL-процессов и аналитикой. Отвечал за обучение пользователей работе с хранилищем данных и инструментами бизнес-аналитики. Преподавание SQL также входило в его обязанности.

Проработав 10 лет в финансовом секторе, он сменил поле деятельности и теперь работает старшим разработчиком баз данных в телекоммуникационной компании. Здесь он имеет дело в основном с СУБД PostgreSQL и отвечает за моделирование данных и реализацию физических структур, разработку хранимых процедур, интеграцию баз данных с другими программными компонентами и разработку хранилища данных.

Имея большой опыт работы с Oracle и PostgreSQL – ведущей коммерческой РСУБД и одной из самых технически продвинутых РСУБД с открытым исходным кодом, – он может сравнивать их и понимает, в чем преимущества той и другой. Опыт реализации различных типов приложений баз данных, а также работы в роли бизнес-аналитика, использующего базы данных как инструмент, научил его понимать, как лучше применять средства СУБД в различных ситуациях. Накопленным опытом он с удовольствием делится в этой книге.

Я благодарен жене и сыну, которые поддерживали меня и давали возможность работать по вечерам и в выходные. Огромное спасибо редакторам за советы и помощь в организации материала. Но больше всего я благодарен основному автору, Салахалдину Джубе, который предложил мне принять участие в работе над книгой, включил в команду и, по сути дела, сделал большую часть работы.

О рецензентах

Д-р Изабель М. Д. Роза – обладатель стипендии имени Марии Склодовской-Кюри, с мая 2016 г. работает в Немецком исследовательском центре интегративного биоразнообразия (iDiv). Родилась в Лиссабоне, Португалия, в 1986 году. Получила степень бакалавра по лесотехнике (2007) и степень магистра по управлению природными ресурсами (2009) в Лиссабонском университете, а также степень доктора по вычислительной экологии (2013) в Имперском колледже Лондона. Является руководителем исследовательского проекта «Применение моделей изменения растительного покрова для решения важных вопросов охраны природы», финансируемого в рамках гранта H2020-MSCA-IF-2015. С 2013 года принимала участие в двух международных проектах, включая финансируемый ЕС проект с бюджетом 1,5 миллиона евро (Terragenesis, ERC-2011-StG_20101109). Автор 15 публикаций в рецензируемых научных журналах, в т. ч. *Nature Ecology and Evolution*, *Current Biology and Global Change Biology*, на которые имеется 252 ссылки (Google citations, октябрь 2017), индекс Хирша равен 8. За время работы в академических учреждениях приобрела ряд навыков, в т. ч. в области статистического анализа, программирования (на языках R, C++ и Python), работы с геоинформационными системами (ArcGIS и QGIS) и создания баз данных (PostgreSQL/PostGIS и SQLServer). Рецензировала книгу «Learning PostgreSQL», также выпущенную издательством Packt.

Шелдон Штраух – ветеран с 23-летним опытом консультирования таких компаний, как IBM, Sears, Ernst & Young, Kraft Foods. Имеет степень бакалавра по организации управления, применяет свои знания, чтобы помочь компаниям самоопределиться. В сферу его интересов входят сбор, управление и глубокий анализ данных, карты и их построение, бизнес-аналитика и применение анализа данных в целях непрерывного улучшения. В настоящее время занимается разработкой сквозного управления данными и добычей данных в компании Enova, оказывающей финансовые услуги и расположенной в Чикаго. В свободное время увлекается искусством, особенно музыкой, и путешествует со своей женой Мэрилин.

Предисловие

Выбор правильной системы управления базами данных – трудная задача из-за большого количества предложений на рынке. В зависимости от бизнес-модели можно выбрать коммерческую СУБД или базу данных с открытым исходным кодом и коммерческой поддержкой. Следует также принимать во внимание ряд технических и нетехнических факторов. Когда речь заходит о реляционной СУБД, PostgreSQL оказывается на вершине рейтинга по нескольким причинам. Лозунг PostgreSQL – *самая передовая база данных с открытым исходным кодом* – отражает развитость технических средств и уверенность сообщества.

PostgreSQL – объектно-реляционная система управления базами данных с открытым исходным кодом. К ее сильным сторонам относится расширяемость, она успешно конкурирует с основными реляционными СУБД: Oracle, SQL Server и MySQL. Благодаря разнообразию расширений и открытой лицензии PostgreSQL часто применяется в исследовательских проектах, но ее код также лежит в основе многих открытых и коммерческих СУБД, например Greenplum и Vertica. К тому же стартапы часто отдают предпочтение PostgreSQL в силу условий лицензии и изобилия компаний, оказывающих коммерческую поддержку.

Существуют версии PostgreSQL для большинства современных операционных систем, включая Windows, Mac и различные дистрибутивы Linux. Имеется также несколько расширений для доступа к данным, управления и мониторинга работы кластеров PostgreSQL, например pgAdmin, OmniDB и psql. Установка и настройки PostgreSQL достаточно просты и поддерживаются большинством диспетчеров пакетов, в т. ч. yum и apt. Разработчики баз данных не испытывают трудностей в освоении PostgreSQL, поскольку она совместима со стандартами **ANSI SQL**. Да и помимо стандартов существует масса ресурсов, которые помогут изучить PostgreSQL, – СУБД отлично документирована и может похвастаться очень активным и хорошо организованным сообществом.

PostgreSQL пригодна как для **OLTP**, так и для OLAP-приложений. Она совместима с **ACID**-транзакциями и для использования в **OLTP**-приложениях не нуждается ни в каких дополнениях. Что касается OLAP-приложений, то PostgreSQL поддерживает оконные функции, адаптеры внешних данных (**FDW** – Foreign Data Wrapper) и наследование таблиц; кроме того, существует немало внешних расширений для этой цели.

Несмотря на совместимость с ACID-транзакциями, PostgreSQL демонстрирует отличную производительность, поскольку в ней применены самые современные алгоритмы и методы. Например, в PostgreSQL используется архитектура **MVCC** (MultiVersion Concurrency Control – управление параллельным доступом с помощью многоверсионности) для обеспечения параллельного доступа к данным. Вдобавок PostgreSQL поддерживает как пессимистическую,

так и оптимистическую конкурентность, а поведение механизма блокировок можно изменять в зависимости от ситуации. Кроме того, в PostgreSQL имеется великолепный анализатор и такие передовые средства, как секционирование данных с помощью наследования таблиц и исключение в силу ограничений, позволяющие ускорить обработку очень больших объемов данных. PostgreSQL поддерживает несколько типов индексов, в т. ч. **B-Tree**, **GiN**, **GiST** и **BRIN**. А начиная с версии PostgreSQL 9.6 поддерживается параллельное выполнение запросов. Наконец, репликация позволяет балансировать нагрузку на различные узлы кластера.

PostgreSQL допускает масштабирование благодаря многочисленным представленным на рынке методам репликации, например Slony и pgpool-II. Дополнительно PostgreSQL изначально поддерживает синхронную и асинхронную потоковую репликацию, а также логическую репликацию. Это делает PostgreSQL чрезвычайно привлекательным решением, поскольку ее можно использовать для создания высокодоступных и высокопроизводительных систем.

Краткое содержание книги

Глава 1 «Реляционные базы данных» содержит введение в концепции реляционных систем управления базами данных, в т. ч. реляционную алгебру и моделирование данных. Здесь же описываются различные типы СУБД: графовые, документные, столбцовые, а также хранилища ключей и значений.

В главе 2 «PostgreSQL в действии» мы расскажем, как установить сервер и клиенты PostgreSQL на различных платформах. Также мы познакомимся с некоторыми возможностями PostgreSQL, в т. ч. встроенной поддержкой репликации и очень богатым набором типов данных.

В главе 3 «Основные строительные блоки PostgreSQL» описываются рекомендации по кодированию, в т. ч. принятые соглашения об идентификаторах. Здесь рассматриваются основные структурные элементы и взаимодействие между ними: шаблонные базы данных, пользовательские базы данных, табличные пространства, роли и настройки. Описываются также основные типы данных и таблицы.

Глава 4 «Дополнительные строительные блоки PostgreSQL» посвящена представлениям, индексам, функциям, пользовательским типам данных, триггерам и правилам. Рассматриваются различные применения этих элементов и сравниваются различные элементы, применимые для решения задачи, например правила и триггеры.

Глава 5 «Язык SQL» – введение в структурированный язык запросов (SQL), используемый для взаимодействия с базой данных: создания и обслуживания структур данных, а также для ввода данных в базу, их изменения, выборки и удаления. В SQL есть команды, относящиеся к трем подъязыкам: **языку определения данных** (**DDL**), **языку манипулирования данными** (**DML**) и **языку управления данными** (**DCL**). В этой главе описаны четыре команды

SQL, составляющие основу языка DML. Особое внимание уделено команде `SELECT`, на примере которой объясняются концепции группировки и фильтрации для демонстрации того, что такое выражения и условия SQL и как используются подзапросы. Здесь же рассматриваются некоторые вопросы реляционной алгебры в применении к соединению таблиц.

В главе 6 «Дополнительные сведения о написании запросов» описаны такие средства SQL, как общие табличные выражения и оконные функции. Они позволяют реализовать вещи, которые без них были бы невозможны, например рекурсивные запросы. Рассматриваются и другие конструкции, как то: фразы `DISTINCT ON` и `FILTER`, а также латеральные подзапросы. Без них, в принципе, можно обойтись, но с их помощью запросы получаются компактнее, проще и быстрее.

В главе 7 «Серверное программирование на PL/pgSQL» рассматриваются параметры функций, в т. ч. количество возвращенных строк и стоимость функции, которые нужны главным образом планировщику запросов. Здесь же представлены управляющие конструкции, например условные команды и циклы. Наконец, объясняется, что такое динамический SQL и как им лучше пользоваться.

В главе 8 «OLAP и хранилища данных» речь пойдет о применении реляционных баз данных для аналитической обработки. Обсуждаются различия между двумя типами рабочей нагрузки: OLTP и OLAP, а также вопросы моделирования в OLAP-приложениях. Кроме того, рассматриваются некоторые технические приемы выполнения **ETL** (extract, transform, load – извлечение, преобразование, загрузка), в т. ч. команда COPY. Также описываются средства PostgreSQL, предназначенные для повышения скорости выборки, в частности просмотр только индексов и секционирование таблиц.

Глава 9 «За пределами традиционных типов данных» посвящена некоторым нестандартным типам: массивам, хешам, JSON-документам и полнотекстовому поиску. Описываются операции и функции для каждого типа данных: инициализация, обновление, доступ и удаление. Наконец, продемонстрировано, как объединить PostgreSQL и Nginx для обслуживания REST-совместимых запросов чтения.

В главе 10 «Транзакции и управление параллельным доступом» подробно обсуждаются свойства ACID и их связь с управлением параллельным доступом. Здесь же рассмотрены уровни изоляции и их побочные эффекты – с демонстрацией этих эффектов на примерах SQL. Также уделено внимание различным методам блокировки, в т. ч. стратегиям пессимистической блокировки: блокировке на уровне строк и рекомендательным блокировкам.

Глава 11 «Безопасность в PostgreSQL» посвящена аутентификации и авторизации. Описываются методы аутентификации в PostgreSQL и объясняется структура конфигурационного файла аутентификации по адресу узла. Также обсуждаются разрешения на доступ к таким объектам базы данных, как схемы, таблицы, представления, индексы и столбцы. Наконец, описано, как защитить секретные данные, в т. ч. пароли, с помощью одностороннего и двустороннего шифрования.

В главе 12 «Каталоги PostgreSQL» приведено несколько рецептов обслуживания кластера базы данных, в т. ч. очистка данных, обслуживание пользовательских процессов, удаление индексов и неиспользуемых объектов базы данных, определение и добавление индексов по внешним ключам и т. д.

В главе 13 «Оптимизация производительности базы данных» обсуждается несколько подходов к оптимизации производительности. Описаны конфигурационные параметры кластера PostgreSQL, используемые для настройки производительности кластера в целом. Также разобраны типичные ошибки при написании запросов и предложено несколько способов повышения производительности, включая построение индексов, секционирование таблиц и исключение в силу ограничений.

В главе 14 «Тестирование» освещены некоторые аспекты процесса тестирования программного обеспечения и его особенности применительно к базам данных. Автономные тесты для баз данных можно писать в виде SQL-скриптов или хранимых функций. Существует несколько каркасов для написания автономных тестов и обработки результатов тестирования.

В главе 15 «PostgreSQL в приложениях на Python» обсуждаются различные продвинутые механизмы, в т. ч. организация пулов соединений, асинхронный доступ и **объектно-реляционное отображение** (**ORM**). Приведен пример, демонстрирующий подключение к базе данных, отправку запроса и выполнение обновления из программы на языке Python. Наконец, дается введение в различные технологии взаимодействия с PostgreSQL, чтобы разработчик имел полную картину современного состояния дел.

В главе 16 «Масштабируемость» рассматриваются проблема масштабируемости и теорема CAP. Здесь же обсуждается репликация в PostgreSQL – физическая и логическая. Описаны различные сценарии масштабирования и их реализация в PostgreSQL.

Что необходимо для чтения книги

Вообще говоря, для сервера и клиентов PostgreSQL не нужно какое-то особое оборудование. PostgreSQL устанавливается на все современные платформы, включая Linux, Windows и Mac. Если необходима некая конкретная библиотека, то мы приводим инструкции по ее установке.

Вам понадобится версия PostgreSQL 10; впрочем, большинство примеров будет работать и с предыдущими версиями. Для выполнения примеров кода и скриптов необходим какой-нибудь клиент PostgreSQL на вашей машине, предпочтительно psql, и доступ к удаленному серверу PostgreSQL.

В среде Windows команда cmd.exe не очень удобна, лучше бы поставить Cygwin (http://www.cygwin.com/) или еще какую-нибудь альтернативу, например PowerShell.

При чтении некоторых глав понадобится дополнительное ПО. Например, в главе 15 нужно будет установить Python и библиотеки, необходимые для работы с PostgreSQL. А в главе 16 удобнее всего будет работать с Docker.

И еще – в главах 9 и 11 рекомендуется использовать Linux, поскольку работать с некоторыми программами, например Nginx и GnuPG, в Windows не очень комфортно. Для запуска Linux на машине под управлением Windows можете воспользоваться программой Virtual Box (https://www.virtualbox.org/).

На кого рассчитана эта книга

Если вам интересно узнать о PostgreSQL – одной из самых популярных в мире реляционных баз данных, – то вы попали по адресу. Те, кого интересует создание добротных приложений для работы с базой данных или хранилищем данных, тоже найдут в книге много полезного. Предварительного опыта программирования или администрирования баз данных для чтения книги не требуется.

Графические выделения

В этой книге тип информации обозначается шрифтом. Ниже приведено несколько примеров с пояснениями.

Фрагменты кода внутри абзаца, имена таблиц базы данных, папок и файлов, данные, которые вводит пользователь, и адреса в Твиттере выделяются следующим образом: «В следующих строчках читается ссылка и результат передается функции `open`».

Кусок кода выглядит так:

```
fin = open('data/fake_weather_data.csv','r',newline='')
reader = csv.reader(fin)
for row in reader:
    myData.append(row)
```

Входная и выходная информация командных утилит выглядит так:

```
$ mongoimport --file fake_weather_data.csv
```

Новые термины и **важные фрагменты** выделяются полужирным шрифтом. Например, элементы графического интерфейса в меню или диалоговых окнах выглядят в книге так: «Чтобы загрузить новые модули, выберите пункт меню **Files** | **Settings** | **Project Name** | **Project Interpreter**».

Предупреждения и важные примечания выглядят так.

Советы и рекомендации выглядят так.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг — возможно, ошибку в тексте или в коде, — мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии данной книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Packt очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

Пожалуйста, свяжитесь с нами по адресу электронной почты dmkpress@gmail.com со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Глава 1

Реляционные базы данных

Эта и следующая главы содержат общий обзор вопросов разработки приложений баз данных, в т. ч. теоретические аспекты реляционных баз данных. Знакомство с теорией поможет не только создавать качественные проекты, но и овладеть тонкостями работы с реляционными базами.

Эта глава касается не только PostgreSQL, но реляционных баз вообще. Будут рассмотрены следующие вопросы:

- **системы управления базами данных:** понимание классификации баз данных дает разработчику возможность использовать лучшее из имеющегося для решения конкретной задачи;
- **реляционная алгебра:** знакомство с реляционной алгеброй позволит овладеть языком SQL, особенно техникой переписывания запросов;
- **моделирование данных:** применяя методы моделирования, вы сможете лучше донести свои идеи до коллег.

Системы управления базами данных

Разные системы управления базами данных (СУБД) поддерживают различные сценарии и требования. У СУБД долгая история. Сначала мы сделаем краткий обзор недавней истории, а затем расскажем о преобладающих на рынке категориях СУБД.

Историческая справка

Термином «база данных» обозначается много разных понятий. Более того, он наводит на мысли о других терминах: данные, информация, структура данных и управление. Базу данных можно определить как набор, или репозиторий данных, обладающий определенной структурой и управляемый **системой управления базами данных (СУБД)**. Данные могут быть структурированными таблицами, слабоструктурированными XML-документами или вовсе не иметь структуры, описываемой какой-то заранее определенной моделью.

На заре развития базы данных были ориентированы в основном на поддержку приложений в сфере бизнеса; это положило начало математически

точно определенной реляционной алгебре и реляционным системам баз данных. Кроме того, во многих отраслях бизнеса, а равно в научных приложениях используются массивы, изображения и пространственные данные, поэтому поддерживаются также новые модели, включая растровые изображения, карты и алгебраические операции с массивами. Графовые базы данных поддерживают запросы к графам, например о поиске кратчайшего пути между двумя вершинами. Они также обеспечивают простой обход графа.

С появлением веб-приложений, в частности социальных порталов, возникла необходимость поддерживать огромное количество запросов распределенным образом. Это привело к новой парадигме в области баз данных, получившей название **NoSQL** (**Not Only SQL**), с другими требованиями, например: примат производительности над отказоустойчивостью и возможность горизонтального масштабирования. В целом на эволюцию баз данных оказывали влияние разнообразные факторы, в том числе:

- **функциональные требования:** сама природа приложений, в которых используются базы данных, потребовала разработки расширений реляционных баз, таких как PostGIS (пространственные данные), или даже специализированных СУБД типа SciDB (для анализа научных данных);
- **нефункциональные требования:** успех языков объектно-ориентированного программирования породил новые тенденции, в частности объектно-ориентированные базы данных. Появились объектно-реляционные системы управления базами данных, перебрасывающие мост между реляционными базами и объектно-ориентированными языками. Взрывной рост объема данных и необходимость обрабатывать терабайты данных на стандартном оборудовании привели к появлению столбцовых баз данных, которые легко масштабируются по горизонтали.

Категории баз данных

Рождались и уходили в небытие различные модели баз данных, включая сетевую и иерархическую модель. Сейчас на рынке преобладают реляционные, объектно-реляционные и NoSQL базы данных. Не следует считать базы данных NoSQL и SQL соперниками – они дополняют друг друга. Применяя ту или иную систему баз данных, можно преодолеть ограничения технологий и выбрать оптимальный вариант.

Базы данных NoSQL

На базы данных NoSQL оказывает влияние теорема CAP, известная также как теорема Брюера. В 2002 году С. Джильберт и Н. Линч опубликовали ее формальное доказательство в статье «Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services». В 2009 году зародилось движение NoSQL. Сегодня насчитывается свыше 150 баз NoSQL (nosql-database.org).

Теорема CAP

Теорема CAP утверждает, что распределенная вычислительная система не может одновременно обеспечить все три следующих свойства:

- **согласованность** – все клиенты сразу же видят последние данные после каждого обновления;
- **доступность** – любой клиент может найти копию нужных данных даже в случае отказа узла. Иными словами, даже после выхода из строя части системы клиенты все равно могут обратиться к данным;
- **устойчивость к разделению** – система продолжает работать даже в случае утраты любого сообщения или отказа части системы.

Характер системы определяется тем, от какого свойства мы отказываемся. Например, пожертвовав согласованностью, мы получим простую, масштабируемую, высокопроизводительную систему управления базами данных. Зачастую основное различие между реляционной и NoSQL базой данных как раз и состоит в согласованности. В реляционной базе гарантируется выполнение свойств **ACID** (**атомарность**, **согласованность**, **изолированность**, **долговечность**). Напротив, во многих базах данных NoSQL принята модель **BASE** (basically available soft-state, eventual consistency – **базовая доступность, неустойчивое состояние, согласованность в конечном счете**).

Мотивация NoSQL

База данных NoSQL предоставляет средства для хранения нереляционных данных, их выборки и манипулирования ими. Такие базы являются распределенными, горизонтально масштабируемыми и, как правило, поставляются с открытым исходным кодом. NoSQL часто основана на модели BASE, отдающей предпочтение доступности над согласованностью. Эта модель дает неформальные гарантии того, что в отсутствие новых обновлений обращение к любому элементу данных рано или поздно вернет его последнюю версию. У такого подхода имеются следующие преимущества:

- простота проектирования;
- горизонтальная масштабируемость и простота репликации;
- отсутствие схемы;
- поддержка гигантских объемов данных.

Теперь рассмотрим некоторые типы баз данных NoSQL.

Хранилища ключей и значений

Хранилище ключей и значений – самый простой тип базы данных. Как следует из названия, хранилище основано на хеш-таблицах. Некоторые хранилища допускают хранение составных типов данных – списков и словарей. В некоторых ситуациях хранилища ключей и значений обеспечивают исключительную скорость, но им недостает поддержки сложных запросов и агрегирования. К числу наиболее известных баз данных такого типа относятся Riak, Redis, Memebase и MemcacheDB.

Столбцовые базы данных

Столбцовые базы данных организованы в виде совокупности столбцов, а не строк. Данные, принадлежащие одному столбцу, хранятся вместе.

В отличие от реляционных баз данных, добавление столбца обходится дешево и выполняется для каждой строки в отдельности. Количество столбцов в разных строках может различаться. Такая структура дает возможность исключить накладные расходы на хранение значений null. Эта модель ориентирована прежде всего на распределенные базы данных.

Одной из самых известных столбцовых баз данных является HBase, основанная на системе хранения Google Bigtable. Столбцовые базы данных проектировались в расчете на очень большие объемы данных, поэтому легко масштабируются. Если набор данных невелик, то использовать HBase не имеет смысла. Во-первых, в рекомендуемой топологии HBase должно быть не менее пяти узлов, а во-вторых, эта база сложна для изучения и администрирования.

Документные базы данных

Документная база данных предназначена для хранения слабоструктурированных данных. Основной единицей хранения в ней является документ. Данные в документах представлены в стандартных форматах, например: XML, JSON и BSON. Для документов не определена единая схема, они не обязаны иметь одинаковую структуру, поэтому обладают высокой степенью гибкости. В отличие от реляционных баз данных, изменить структуру документа легко, и эта процедура не мешает клиентам обращаться к данным.

Графовые базы данных

В основе графовых баз данных лежит теория графов. База данных содержит информацию о вершинах и ребрах графа. С вершинами и ребрами могут быть ассоциированы данные. Графовая база позволяет обходить вершины, следуя вдоль ребер. Поскольку граф – очень общая структура данных, такие базы можно использовать для представления самых разных данных. Самая известная реализация графовой базы данных с открытым исходным кодом и коммерческой поддержкой – Neo4j.

Реляционные и объектно-реляционные базы данных

Реляционные системы управления базами данных относятся к числу наиболее распространенных СУБД. Крайне маловероятно, что в наши дни найдется организация или персональный компьютер, где нет ни одной программы, так или иначе опирающейся на РСУБД.

Приложение может обращаться к реляционной базе, расположенной на выделенном сервере (или нескольких серверах) либо на том же компьютере, где оно работает; в последнем случае упрощенный движок РСУБД встроен в приложение в виде разделяемой библиотеки. Функциональность РСУБД зависит от

поставщика, но большинство совместимо со стандартами ANSI SQL. Формально реляционная база данных описывается реляционной алгеброй и основана на реляционной модели. **Объектно-реляционная база данных** похожа на реляционную и дополнительно поддерживает следующие объектно-ориентированные концепции:

- определенные пользователем и сложные типы данных;
- наследование.

Свойства ACID

В реляционной базе данных логическая операция называется транзакцией. Технически транзакция может состоять из нескольких операций с базой данных: **создания**, **чтения**, **обновления** и **удаления** (create, read, update, delete – **CRUD**). В качестве примера транзакции можно привести распределение фиксированного бюджета по нескольким проектам. Если увеличить сумму, выделенную одному проекту, то необходимо отобрать ту же сумму у какого-то другого проекта. В этом контексте свойства ACID описываются следующим образом:

- **атомарность:** всё или ничего, т. е. если какая-то часть транзакции завершается неудачно, то неудачно завершается и вся транзакция;
- **согласованность:** транзакция переводит базу данных из одного непротиворечивого состояния в другое, также непротиворечивое. Непротиворечивость базы данных обычно описывается с помощью ограничений и связей между данными. Представьте, к примеру, что вы захотели удалить свою учетную запись в интернет-магазине. Чтобы сохранить согласованность, нужно было бы удалить и детальную информацию, ассоциированную с учетной записью, например список адресов. Для этого служат ограничения внешнего ключа, о которых мы подробнее расскажем в следующей главе;
- **изолированность:** параллельное выполнение транзакций переводит систему в такое же состояние, как если бы они выполнялись последовательно;
- **долговечность:** результат зафиксированных, т. е. успешно выполненных, транзакций сохраняется даже в случае пропадания электропитания или аварии сервера. В PostgreSQL это обеспечивается **журналом предзаписи** (write-ahead log – WAL). В других базах данных, например в Oracle, аналогичный механизм называется журналом транзакций.

Язык SQL

Реляционные базы данных обычно дополняются **структурированным языком запросов** (structured query language – SQL). SQL – декларативный язык программирования баз данных, стандартизованный **Американским национальным институтом стандартов** (**ANSI**) и **Международной организацией по стандартизации** (**ISO**). Первый стандарт SQL был опубликован в 1986 го-

ду, за ним последовали стандарты SQL:1999, SQL:2003, SQL:2006, SQL:2008, SQL:2011 и SQL:2016.

Язык SQL состоит из нескольких частей:

- **язык определения данных** (**DDL**): позволяет определить и изменить структуру реляционной базы данных;
- **язык манипулирования данными** (**DML**): служит для выборки данных из отношений;
- **язык управления данными** (**DCL**): служит для управления правами доступа.

Понятия реляционной модели

Реляционная модель – это логика первого порядка, или исчисление предикатов, разработанная Эдгаром Ф. Коддом в 1970 году и изложенная в статье «A relational model of data for large shared data banks»[1]. База данных в ней представлена в виде набора отношений (relation). Состояние базы данных в целом определяется состоянием всех имеющихся в ней отношений. Из отношений можно извлекать различную информацию, применяя операции соединения, агрегирования и фильтрации. В этом разделе мы дадим обзор основных понятий реляционной модели и начнем с описания отношения, кортежа, атрибута и домена. Эти термины, используемые в формальной реляционной модели, соответствуют таблице, строке, столбцу и типу данных в языке SQL.

Отношение

Отношение можно представлять себе как таблицу с заголовком, столбцами и строками. Имя таблицы и ее заголовок необходимы для интерпретации данных, хранящихся в строках. Каждая строка представляет группу взаимосвязанных полей, описывающих некий объект.

Отношение состоит из множества кортежей. Каждый кортеж – это упорядоченное множество атрибутов, причем атрибуты и порядок их следования во всех кортежах одинаковы. Атрибут характеризуется доменом, т. е. типом и именем.

	customer_id	first_name	last_name	email
Кортеж →	1	thomas	sieh	thomas@example.com
Кортеж →	2	wang	kim	kim@example.com
	Атрибут ↑	Атрибут ↑	Атрибут ↑	Атрибут ↑

Схема отношения описывается его именем и атрибутами. Например, `customer (customer_id, first_name, last_name, email)` – схема отношения customer. **Состояние отношения** определяется множеством составляющих его кортежей, поэтому добавление, удаление и изменение кортежа переводят отношение в другое состояние.

---

[1] Реляционная модель данных для больших совместно используемых банков данных. http://citforum.ru/database/classics/codd/.

Место кортежа в отношении и их относительный порядок не играют роли. Кортежи отношения можно упорядочить по одному или по нескольким атрибутам. В отношении не может быть кортежей-дубликатов.

Отношение может представлять сущности реального мира, например клиента, или ассоциации между отношениями. Так, клиент может пользоваться несколькими службами, а каждая служба может быть предложена нескольким клиентам. Эту ситуацию можно смоделировать тремя отношениями: `customer`, `service` и `customer_service`, причем `customer_service` ассоциирует отношения `customer` и `service`. Разделение данных по отношениям – ключевая идея реляционного моделирования, у которой есть специальное название – нормализация. В процессе нормализации отношения и их столбцы организуются так, чтобы уменьшить избыточность. Предположим, к примеру, что служба хранится прямо в отношении `customer`. Если одной службой пользуется несколько клиентов, то это приведет к дублированию данных о клиентах. К тому же для обновления службы пришлось бы обновлять все ее копии в таблице клиентов.

Кортеж

Кортеж – это упорядоченный набор атрибутов. Атрибуты перечисляются в круглых скобках через запятую, например (john, smith, 1971). Элементы кортежа идентифицируются именем атрибута. Кортежи обладают следующими свойствами:

- $(a_1, a_2, a_3,\ldots, a_n) = (b_1, b_2, b_3,\ldots, b_n)$ тогда и только тогда, когда $a_1 = b_1$, $a_2 = b_2$, …, $a_n = b_n$;
- кортеж не является множеством, порядок элементов имеет значение, и дубликаты разрешены:
  - $(a_1, a_2) \neq (a_2, a_1)$;
  - $(a_1, a_1) \neq (a_1)$;
- количество атрибутов в кортеже конечно.

В строгой реляционной модели не допускаются многозначные и составные атрибуты. Это важно с точки зрения устранения избыточности и повышения степени согласованности данных. Но в современных реляционных базах данных это ограничение частично снято, чтобы можно было хранить данные таких составных типов, как JSON.

По поводу того, как применять нормализацию, ведутся ожесточенные споры. Мы рекомендуем нормализовать данные, если нет основательных причин этого не делать.

Значение NULL

Для исчисления предикатов в реляционных базах данных применяется **трехзначная логика** (**3VL**), в которой значений истинности три: истина (true), ложь (false) и неизвестно (unknown), или NULL. В реляционной базе данных третье значение, NULL, можно интерпретировать разными способами, например: данные неизвестны, данные отсутствуют, неприменимо или данные бу-

дут загружены позже. Трехзначная логика устраняет неоднозначность. Например, два значения NULL не равны друг другу.

Ниже приведены таблицы истинности для логических операторов OR и AND; отметим, что оба оператора коммутативны, т. е. *A AND B = B AND A*:

A	B	A AND B	A OR B
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	TRUE
NULL	TRUE	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
NULL	FALSE	FALSE	NULL
NULL	NULL	NULL	NULL

А вот таблица истинности для оператора NOT:

A	B
TRUE	FALSE
FALSE	TRUE
NULL	NULL

Атрибут

У каждого атрибута есть имя и домен, причем никакие два атрибута одного отношения не могут иметь одинаковые имена. Домен определяет множество допустимых значений атрибута. Один из способов задать домен – указать тип данных и ограничения на этот тип. Например, почасовая оплата должна быть положительным вещественным числом, большим пяти (если предположить, что минимальная почасовая оплата равна пяти долларам). Домен может быть непрерывным, как в случае зарплаты (положительное вещественное число), или дискретным, как в случае пола.

В строгой реляционной модели на домен налагается ограничение: значение должно быть атомарным, т. е. неделимым. Например, домен атрибута name (полное имя) не является атомарным, потому что имя можно разделить на две части: имя и фамилию. Приведем несколько примеров доменов:

- **телефон:** текст определенной длины, состоящий из цифр;
- **код страны:** двузначные и трехзначные коды стран определены в стандарте ISO 3166 (соответственно ISO alpha-2 и ISO alpha-3). Так, коды Германии равны DE и DEU.

В реальных приложениях лучше использовать международные стандарты в справочных таблицах, в т. ч. стран и валют. Это сделает ваши данные гораздо более понятными внешним программам и повысит качество данных.

Ограничения

В реляционной модели определено много ограничений для контроля целостности данных, избыточности и допустимости:

- **избыточность:** кортежи-дубликаты запрещены в отношении;
- **допустимость:** доменные ограничения контролируют допустимость данных;
- **целостность:** отношения в базе данных связаны друг с другом. Операция над одним отношением, например обновление или удаление кортежа, может оставить другое отношение в некорректном состоянии.

Ограничения в реляционной базе данных можно отнести к двум категориям:

- унаследованные от реляционной модели: ограничения доменной целостности, целостности сущностей и ссылочной целостности;
- семантические ограничения, бизнес-правила и ограничения, свойственные приложению, которые нельзя явно выразить средствами реляционной модели. Однако появление процедурных языков на основе SQL, в частности PL/pgsql в PostgreSQL, позволило моделировать такие ограничения и в реляционной базе данных.

Ограничение доменной целостности

Ограничение доменной целостности гарантирует допустимость данных. Прежде всего необходимо определиться с типом данных. Типом данных домена может быть integer, real, boolean, character, text, inet и т. д. Например, имя и адрес электронной почты имеют тип text. Зная тип данных, можно уже определить проверочное ограничение, например шаблон почтового адреса.

- **Проверочное ограничение:** такое ограничение применяется к одному атрибуту или к группе атрибутов. Рассмотрим отношение `customer_service`, в котором определены атрибуты `customer_id`, `service_id`, `start_date`, `end_date`, `order_date`. Мы можем определить для него ограничение `start_date<end_date`, проверяющее, что дата начала меньше даты конца.
- **Ограничение умолчания:** у атрибута может быть значение по умолчанию. Это может быть либо фиксированное значение, например стандартная почасовая оплата труда, скажем, 10 долларов. Или же динамически вычисляемое значение, например значение функции random, current time или date. Например, в отношении `customer_service` атрибут `order_date` может по умолчанию принимать текущую дату.
- **Ограничение уникальности:** гарантирует, что значение атрибута уникально среди всех кортежей отношения. Допускается также значение null. Пусть определено отношение `player` с атрибутами (`player_id`, `player_nickname`). У игрока есть уникальный идентификатор, и он может выбрать себе ник, который также должен идентифицировать его однозначно.
- **Ограничение not null:** по умолчанию атрибут может принимать значение null. Ограничение not null запрещает это. Например, у каждого человека в книге записей о рождении должно быть имя.

Ограничение сущностной целостности

В реляционной модели отношение определяется как множество кортежей. Это означает, что все кортежи в нем должны быть различны. Ограничение сущ-

ностной целостности означает наличие первичного ключа, представляющего собой один или несколько атрибутов со следующими характеристиками:

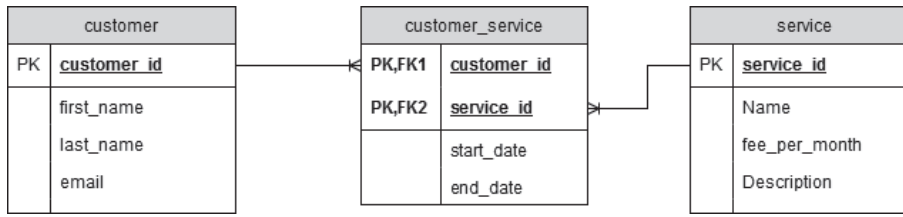
- ключ должен быть уникален;
- ни один атрибут, входящий в ключ, не должен принимать значение null.

В каждом отношении может быть только один первичный ключ, но много уникальных ключей. Ключ-кандидат – это минимальный набор атрибутов, с помощью которых можно однозначно идентифицировать кортеж. Любой уникальный, не принимающий значения null атрибут может быть ключом-кандидатом. Набор, состоящий из всех атрибутов, образует суперключ. На практике часто выбирают в качестве первичного ключа один атрибут, а не составной ключ (состоящий из двух или более атрибутов), чтобы было проще соединять отношения между собой.

Первичный ключ, генерируемый СУБД, называют **суррогатным**, или **синтетическим**. В противном случае ключ называется **натуральным**. В роли суррогатных ключей-кандидатов могут выступать порядковые номера или **универсальные уникальные идентификаторы** (**UUID**). У суррогатных ключей много преимуществ, в т. ч. высокая производительность, устойчивость к изменению требований, адаптируемость и совместимость с объектно-реляционными отображениями. Главный недостаток суррогатных ключей состоит в том, что они открывают возможность для появления кортежей-дубликатов, отличающихся только значением ключа.

Ограничения ссылочной целостности

Отношения связаны между собой общими атрибутами. Ограничения ссылочной целостности контролируют связи между двумя отношениями и гарантируют, что данные в кортежах согласованы. Если кортеж одного отношения ссылается на кортеж другого отношения, то этот второй кортеж должен существовать. Так, если с клиентом связана служба, то должны существовать и клиент, и служба. Например, в отношении `customer_service` не может быть кортежа со значениями (`5, 1,01-01-2014, NULL`), поскольку не существует клиента, для которого `customer_id` равно `5`.



Отсутствие ограничений ссылочной целостности может стать причиной многих проблем:

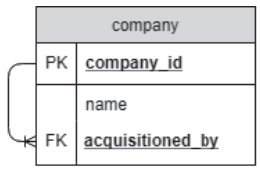
- некорректные данные в общих атрибутах;
- некорректная информация при соединении данных из разных отношений;

- снижение производительности из-за неоптимального плана выполнения, построенного планировщиком PostgreSQL или сторонней программой.

Наличие внешних ключей может повысить производительность чтения данных из нескольких таблиц. Планировщик выполнения запросов будет иметь более точную оценку количества подлежащих обработке строк. Отключение внешних ключей при выполнении массовой вставки резко ускоряет операцию.

Ограничения ссылочной целостности реализуются с помощью внешних ключей. **Внешним ключом** называется атрибут или группа атрибутов, однозначно идентифицирующих кортеж в отношении, на которое производится ссылка (внешнем отношении). В силу самой своей природы внешний ключ часто является первичным во внешнем отношении. Но, в отличие от первичного ключа, внешний может принимать значение null. Он также может ссылаться на уникальный атрибут внешнего отношения. Допуская значения null во внешнем ключе, мы получаем возможность моделировать различные ограничения кардинальности, т. е. ограничения на количество элементов по обе стороны связи. Например, если родитель может иметь более одного потомка, то говорят о связи один-ко-многим, поскольку с одним кортежем первичного (ссылающегося) отношения может быть ассоциировано несколько кортежей внешнего. Отношение может также ссылаться само на себя. Тогда внешний ключ называется **автореферентным**, или **рекурсивным**.

Например, одна компания может быть приобретена другой.



company_id	name	acquisitioned_by
1	Facebook	
2	WhatsApp	1

Для гарантии целостности данных можно определить несколько типов поведения в случае, когда кортеж во внешнем отношении обновляется или удаляется:

- **каскадом:** если кортеж во внешнем отношении удаляется или его внешний атрибут обновляется, то ссылающиеся на него кортежи первичного отношения также удаляются или обновляются;
- **запрет:** кортеж, на который ведет ссылка, не может быть удален, а его внешний атрибут не может быть изменен;
- **ничего не делать:** то же, что запрет, но решение откладывается до конца транзакции;
- **присваивание значения по умолчанию:** если кортеж во внешнем отношении удаляется или внешний атрибут обновляется, то внешнему ключу в первичном отношении присваивается значение по умолчанию;

- **присваивание null:** если кортеж во внешнем отношении удаляется, то внешнему ключу в первичном отношении присваивается значение null.

Семантические ограничения

Ограничения целостности или ограничения бизнес-логики касаются приложения в целом. Например, в любой момент у клиента может быть не более одной активной службы. Такие ограничения проверяются либо на уровне бизнес-логики прикладной программы, либо с помощью процедурных языков на основе SQL. Для этой цели можно также использовать триггеры и правила. Какой подход выбрать – процедурный язык на основе SQL или высокоуровневый язык программирования, – зависит от приложения; иногда комбинируют оба.

У процедурного языка на основе SQL имеются следующие преимущества:

- **производительность:** в РСУБД часто имеются сложные оптимизаторы для построения эффективных планов выполнения. А в некоторых случаях, например в приложениях добычи данных, объем обрабатываемых данных очень велик. Обработка на процедурном языке позволяет исключить передачу данных по сети. Кроме того, в некоторых процедурных языках применяются хитроумные алгоритмы кеширования;
- **срочные изменения:** процедурный язык на основе SQL позволяет исправлять ошибки, не останавливая службу.

У реализации бизнес-логики на уровне базы данных есть много плюсов и минусов, эта тема постоянно рождает споры. Отметим лишь некоторые недостатки: видимость внутреннего устройства базы данных, неэффективная работа программистов, вынужденных писать код без поддержки со стороны инструментов и IDE, трудности повторного использования кода.

Реляционная алгебра

Реляционная алгебра – это формальный язык реляционной модели. Она определяет набор замкнутых операций над отношениями, т. е. результатом любой операции является новое отношение. Реляционная алгебра наследует многие операторы от теории множеств. Операции реляционной алгебры можно отнести к двум группам:

- в первую группу входят теоретико-множественные операции: объединение, пересечение, разность множеств и декартово (или перекрестное) произведение;
- во вторую группу входят операции, специфичные для реляционной модели, например выборка и проекция. Операции реляционной алгебры бывают также унарными и бинарными.

Перечислим примитивные операторы:

- `select` (σ): унарная операция (выборка), записываемая в виде $\sigma\varphi_R$, где φ – предикат. Этот оператор выбирает такие кортежи *R*, для которых φ принимает значение true;

- `project` (π): унарная операция (проекция), которая отбирает часть атрибутов отношения и записывается в виде $\pi_{a1,a2,...,an}R()$, где $a1$, $a2$, ..., an – имена атрибутов;
- `cartesian product` (×): бинарная операция, которая порождает более сложное отношение, комбинируя кортежи двух операндов. Пусть R и S – два отношения, тогда $R \times S = (r_1, r_2, ..., r_n, s_1, s_2, ..., s_n)$, где $(r_1, r_2, ..., r_n) \in R$ и $(s_1, s_2, ..., s_n) \in S$;
- `union` (∪): теоретико-множественное объединение отношений. Отметим, что отношения должны быть совместимы по объединению, т. е. состоять из одинаковых атрибутов в одном и том же порядке. Формально $R \cup S = (r_1, r_2, ..., r_n) \cup (s_1, s_2, ..., s_n)$, где $(r_1, r_2, ..., r_n) \in R$ и $(s_1, s_2, ..., s_n) \in S$;
- `difference` (-): бинарная операция (разность), операнды которой должны быть совместимы по объединению. Создается новое отношение, содержащее те кортежи первого операнда, которые отсутствуют во втором. Формально $R - S = (r_1, r_2, ..., r_n)$, где $(r_1, r_2, ..., r_n) \in R$ и $(r_1, r_2, ..., r_n) \notin S$;
- `rename` (ρ): унарная операция (переименование), которая воздействует только на атрибуты. В основном этот оператор применяется, чтобы различить одноименные атрибуты в соединяемых отношениях или чтобы дать атрибуту более вразумительное имя для целей презентации. Записывается в виде $\rho_{a/b}R$, где a и b – имена атрибутов, причем b – один из атрибутов R.

Помимо примитивных операторов, имеются агрегатные функции, например: `sum`, `count`, `min`, `max` и `avg`. Примитивные операторы можно использовать для определения других реляционных операторов, в т. ч. левого соединения, правого соединения, эквисоединения и пересечения. Реляционная алгебра очень важна в силу своей выразительной мощи, помогающей оптимизировать и переписывать запросы. Например, операция выборки коммутативна, т. е. $\sigma_a\sigma_b R = \sigma_b\sigma_a R$. Другой пример – две последовательные выборки можно заменить одной с конъюнкцией предикатов: $\sigma_a\sigma_b R = \sigma_{a\ AND\ b} R$.

Операции выборки и проекции

Операция `SELECT` применяется, чтобы выбрать часть кортежей отношения. Она всегда возвращает множество кортежей без повторений, что следует из ограничения сущностной целостности. Например, запрос «*дай мне информацию о клиенте, для которого customer_id равно 2*» записывается следующим образом:

$\sigma_{customer\_id\ =2}$ customer.

Как уже было сказано, операция выборки коммутативна, т. е. запрос «*дай мне всех клиентов с именем kim, для которых известен адрес электронной почты*» можно записать тремя способами:

$\sigma_{email\ is\ not\ null}(\sigma_{first\_name=kim}$ customer);
$\sigma_{first\_name=kim}(\sigma_{email\ is\ not\ null}$ customer);
$\sigma_{first\_name=kim\ and\ email\ is\ not\ null}$ (customer).

Конечно же, предикаты выборки зависят от типа данных. Для числовых типов в качестве оператора сравнения можно указывать ≠, =, <, >, ⩾, ⩽. Предикативное выражение может также содержать скобки и функции. В SQL эквивалентом операции выборки является команда `SELECT *`, а предикат определяется во фразе `WHERE`.

Символ * означает «все атрибуты отношения». Отметим, что в производственных системах использовать * не рекомендуется, лучше перечислять необходимые атрибуты явно.

Следующее предложение `SELECT` эквивалентно выражению реляционной алгебры $\sigma_{\text{customer\_id}=2}$ customer:

```
SELECT * FROM customer WHERE customer_id = 2;
```

Оператор проекции можно наглядно представлять себе как вертикальный срез таблицы. Запрос «*дай мне имена клиентов*» в реляционной алгебре записывается так:

$\pi_{\text{first\_name, last\_name}}$ customer.

Ниже представлен результат проецирования:

first_name	last_name
thomas	sieh
wang	kim

В строгой реляционной модели кортежи-дубликаты не допускаются; количество кортежей, возвращенных оператором `PROJECT`, всегда меньше или равно количеству кортежей во всем отношении. Если в список атрибутов в операторе `PROJECT` входит первичный ключ, то в результирующем отношении будет столько же строк, сколько в исходном.

Оператор проекции тоже поддается оптимизации. Например, справедливо следующее тождество для каскадных проекций:

$\pi_a(\pi_a,\pi_b(R)) = \pi_a(R)$.

В SQL эквивалентом оператора `PROJECT` является команда `SELECT DISTINCT`. Ключевое слово `DISTINCT` означает, что нужно устранить дубликаты. Чтобы получить показанный выше результат, следует выполнить такую команду SQL:

```
SELECT DISTINCT first_name, last_name FROM customers;
```

В некоторых случаях операции `PROJECT` и `SELECT` можно выполнять в любом порядке. Запрос «*дай мне имя клиента с идентификатором customer_id = 2*» можно записать двумя способами:

$\sigma_{\text{customer\_id}=2}(\pi_{\text{first\_name, last\_name}}$ customer);
$\pi_{\text{first\_name, last\_name}}(\sigma_{\text{customer\_id}=2}$ customer).

Но бывает и так, что порядок операторов PROJECT и SELECT важен, иначе получается некорректное выражение. Запрос «*дай мне фамилии всех клиентов с именем kim*» можно записать только так:

$\pi_{(last\_name}(\sigma_{first\_name=kim}$ customer).

Операция переименования

Операция rename применяется, чтобы изменить имя атрибута результирующего отношения или присвоить конкретное имя результирующему отношению. Она позволяет:

- устранить неоднозначность в случае, когда в двух или более отношениях встречаются одноименные атрибуты;
- присвоить вразумительные имена атрибутам, особенно при формировании отчетов;
- изменить определение отношения, сохранив обратную совместимость.

Эквивалентом оператора rename в SQL является ключевое слово AS. Ниже создается отношение, содержащее один кортеж с одним атрибутом, которому присвоено имя PI:

```
SELECT 3.14::real AS PI;
```

Теоретико-множественные операции

К теоретико-множественным относятся операции union (объединение), intersection (пересечение) и minus (разность). Intersection не является примитивным оператором реляционной алгебры, потому что его можно выразить через объединение и разность:

$A \cap B = ((A \cup B) - (A - B)) - (B - A)$.

Операторы пересечения и объединения коммутативны:

$A \cap B = B \cap A$;
$A \cup B = B \cup A$.

Запрос «*дай мне идентификаторы всех клиентов, с которыми не ассоциирована ни одна служба*» можно записать так:

$\pi_{customer\_id}$ customer – $\pi_{customer\_id}$ customer_service.

Операция декартова произведения

Операция cartesian product позволяет построить отношение, содержащее комбинации кортежей двух других отношений. Пусть A и B – два отношения, и $C = A \times B$. Тогда:

количество атрибутов C равно количество атрибутов A + количество атрибутов B;

*количество кортежей C равно количество атрибутов A * количество кортежей B.*

На следующем рисунке показано декартово произведение отношений `customer` и `customer_service`.

customer_id	first_name	las_name
1	thomas	sieh
2	wang	kim

X

customer_service_id	customer_id	start_date	end_date
1	1	2017-11-01	2018-11-01
2	1	2017-11-01	2018-11-01

=

customer_id	first_name	las_name	customer_service_id	customer_id	start_date	end_date
1	thomas	sieh	1	1	2017-11-01	2018-11-01
2	wang	kim	1	1	2017-11-01	2018-11-01
1	thomas	sieh	2	1	2017-11-01	2018-11-01
2	wang	kim	2	1	2017-11-01	2018-11-01

В SQL эквивалентом декартова произведения является оператор перекрестного соединения `CROSS JOIN`. Запрос «*для клиента с идентификатором 1 выбрать идентификатор и имя клиента, а также идентификаторы ассоциированных с ним служб*» записывается так:

```
SELECT DISTINCT customer_id, first_name, last_name, service_id
FROM customer AS c CROSS JOIN customer_service AS cs
WHERE c.customer_id=cs.customer_id AND c.customer_id = 1;
```

На этом примере наглядно видна связь между реляционной алгеброй и языком SQL. Мы использовали операции `select`, `rename`, `project` и `cartesian product`. Покажем, как можно с помощью реляционной алгебры оптимизировать выполнение этого запроса. Его можно было бы выполнить несколькими способами.

План выполнения 1:

1. Выбрать клиента, для которого `customer_id = 1`.
2. Выбрать службу для клиента с идентификатором `customer_id = 1`.
3. Выполнить перекрестное соединение отношений, полученных на шагах 1 и 2.
4. Спроецировать отношение, полученное на шаге 3, на атрибуты `customer_id`, `first_name`, `last_name` и `service_id`.

План выполнения 2:

1. Выполнить перекрестное соединение отношений `customer` и `customer_service`.
2. Выбрать все кортежи, для которых

   ```
   customer_service.customer_id=customer.customer_id and
   customer.customer_id = 1.
   ```

3. Спроецировать отношение, полученное на шаге 2, на атрибуты `customer_id`, `first_name`, `last_name` и `service_id`.

Запрос SELECT написан именно таким образом, чтобы показать, как операции реляционной алгебры транслируются на язык SQL. Современный SQL позволяет проецировать на атрибуты без использования DISTINCT. Кроме того, следовало бы использовать внутреннее соединение, а не перекрестное.

У любого плана выполнения есть стоимость, выражаемая в терминах процессорного времени и количества операций доступа к памяти (запоминающему устройству с произвольной выборкой – ЗУПВ) и диску. РСУБД выбирает план с наименьшей стоимостью. В показанных выше планах операция `rename`, как и оператор `distinct`, для простоты была опущена.

Моделирование данных

Модель данных описывает сущности реального мира, например клиента, службу и продукты, а также связи между ними. Модель данных – это абстракция отношений в базе данных. С помощью модели разработчик представляет бизнес-требования в виде отношений. Модели также служат средством обмена информацией между разработчиками и заказчиками.

На предприятии модели данных играют очень важную роль – обеспечение согласованности данных между взаимодействующими системами. Так, если некоторая сущность не определена или определена неправильно, то это повлечет за собой несогласованность данных и разночтения. Например, если семантика сущности «клиент» определена расплывчато и разные подразделения называют одну и ту же сущность по-разному, скажем, «customer» и «client», то в отделе эксплуатации может возникнуть путаница.

Виды моделей данных

В стандарте ANSI определены следующие виды моделей данных:

- **концептуальная модель данных** – описывает семантику доменов и служит для перечисления основных бизнес-правил, действующих лиц и концепций. Содержит высокоуровневое описание бизнес-требований и часто называется моделью данных верхнего уровня;
- **логическая модель данных** – описывает семантику в контексте определенной технологии. Например, для объектно-ориентированных языков это могут быть UML-диаграммы классов;
- **физическая модель данных** – описывает, как хранятся данные на уровне оборудования. Оперирует такими понятиями, как сеть хранения данных, табличное пространство и т. д.

В соответствии со стандартом ANSI эта абстракция позволяет изменять некоторые из трех видов, не затрагивая остальных. Можно изменить логическую и физическую модель, оставив концептуальную неизменной. Например, какой алгоритм сортировки использовать – пузырьковый или быстрый, – для концептуальной модели данных несущественно. Как и структура отношений. Мы могли бы разделить одно отношение на несколько, применив правила

нормализации или воспользовавшись типом данных `enum` для моделирования справочных таблиц.

Модель сущность-связь

Модель **сущность-связь** (entity-relation – **ER**) относится к категории концептуальных моделей данных. В ней данные представлены в форме, понятной как заказчикам, так и разработчикам. Существуют специальные методы преобразования ER-модели в реляционную.

Концептуальное моделирование – часть **жизненного цикла разработки программного обеспечения**. Обычно этим занимаются после завершения сбора требований – к функциональности и к данным. В этот момент разработчик может нарисовать черновой вариант ER-диаграммы и описать функциональные требования с помощью диаграмм потоков данных, диаграмм последовательностей, пользовательских историй и многих других технических приемов.

На этапе проектирования разработчик базы данных должен уделять особое внимание качеству проекта, прогонять эталонные тесты для обеспечения требуемой производительности и проверять корректность пользовательских требований. Если моделируется простая система, то можно сразу приступать к кодированию. Но подходить к проектированию нужно тщательно, поскольку модель данных оказывает влияние не только на алгоритмы, но и на сами данные. Изменение модели в будущем может повлечь за собой серьезные трудности при организации переноса данных из одной структуры в другую.

При проектировании схемы базы часто бывает несколько альтернативных решений, и нужно выбирать. Перечислим некоторые распространенные ошибки.

- **Избыточные данные.** Неудачный проект базы данных грешит избыточностью. Наличие избыточных данных может стать причиной других проблем, в т. ч. несогласованности и снижения производительности. При обновлении кортежа, содержащего избыточные данные, необходимо обновить все кортежи, содержащие те же данные.
- **Переизбыток null.** В некоторых приложениях, например медицинских, данные по необходимости разрежены. Так, количество атрибутов в отношении `diagnostics` может исчисляться сотнями: жар, головная боль, насморк и т. д. Большинство из них для конкретного диагноза несущественно, но в общем случае все они нужны. Такую ситуацию можно смоделировать с помощью сложных типов данных, например JSON.
- **Сильная связанность.** В некоторых случаях сильная связанность ведет к запутанным структурам данных, которые трудно изменять. Бизнес-требования со временем меняются, и некоторые из них могут устареть. Моделирование обобщения и специализации (например, студент, обучающийся неполное время, является студентом) сильно связанным образом может стать причиной проблем.

Пример приложения

Для иллюстрации принципов модели сущность-связь рассмотрим веб-сайт для торговли автомобилями. Ниже приведены требования к этому приложению.

1. Сайт позволяет пользователю зарегистрироваться и предоставляет различные услуги в зависимости от категории пользователя.
2. Существуют продавцы и обычные пользователи. Продавец может создавать новые объявления о продаже автомобилей, прочие пользователи могут только производить поиск и просмотр.
3. В процессе регистрации любой пользователь должен указать свое полное имя и действительный адрес электронной почты. Этот адрес будет использоваться для входа в систему.
4. Продавец должен указать адрес.
5. Пользователь может оценить объявление и качество услуг продавца.
6. История поисков пользователя должна сохраняться для последующего использования.
7. Каждый продавец имеет ранг, который влияет на поиск его объявлений. Ранг определяется количеством опубликованных объявлений и рейтингами, проставленными пользователями.
8. У объявления о продаже имеется дата, а у автомобиля много атрибутов, в т. ч. цвет, количество дверей, количество предыдущих владельцев, регистрационный номер, фотографии и т. д.

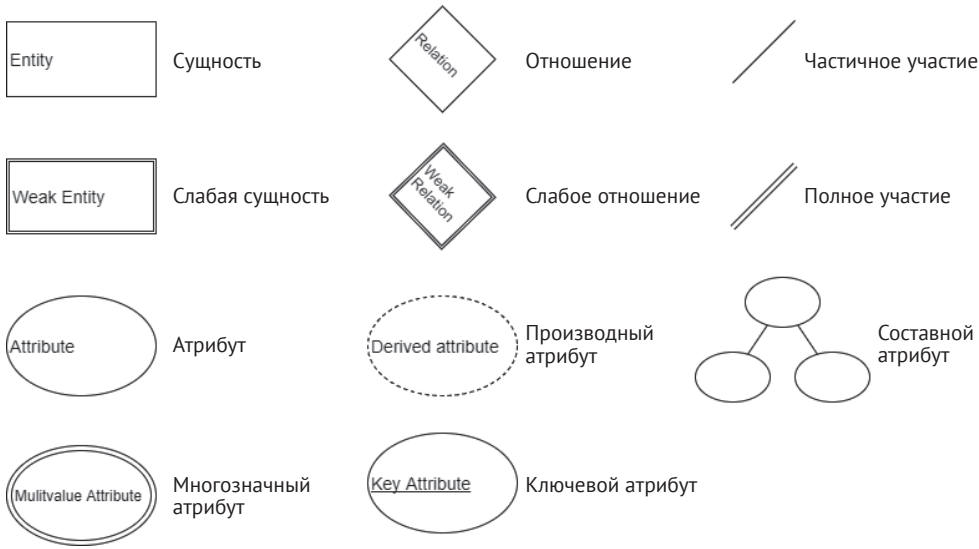
Сущности, атрибуты и ключи

На диаграмме сущность-связь представлены сущности, атрибуты и связи. Сущность соответствует объекту реального мира, например автомобилю или пользователю. Атрибут – это свойство объекта. Связь описывает ассоциацию между двумя или более сущностями.

Атрибуты могут быть простыми (атомарными) или составными. В составном атрибуте можно выделить отдельные части. Часть составного атрибута дает неполную информацию, которая в отрыве от всего остального бессмысленна. Например, адрес состоит из названия улицы, номера дома и почтового индекса. Любая его часть бесполезна без знания остальных.

Атрибуты бывают однозначными и многозначными. Цвет птицы – пример многозначного атрибута. Она может быть, например, черно-красной. У многозначного атрибута могут быть ограничения снизу и сверху на допустимое количество значений. Кроме того, одни атрибуты могут быть выведены из других, например возраст можно вывести из даты рождения. В нашем примере полный ранг продавца определяется количеством объявлений и рейтингами пользователей.

Для идентификации сущности служат ключевые атрибуты. Ключевой атрибут должен быть уникальным, но ему не обязательно соответствует первичный ключ в физической модели отношения.

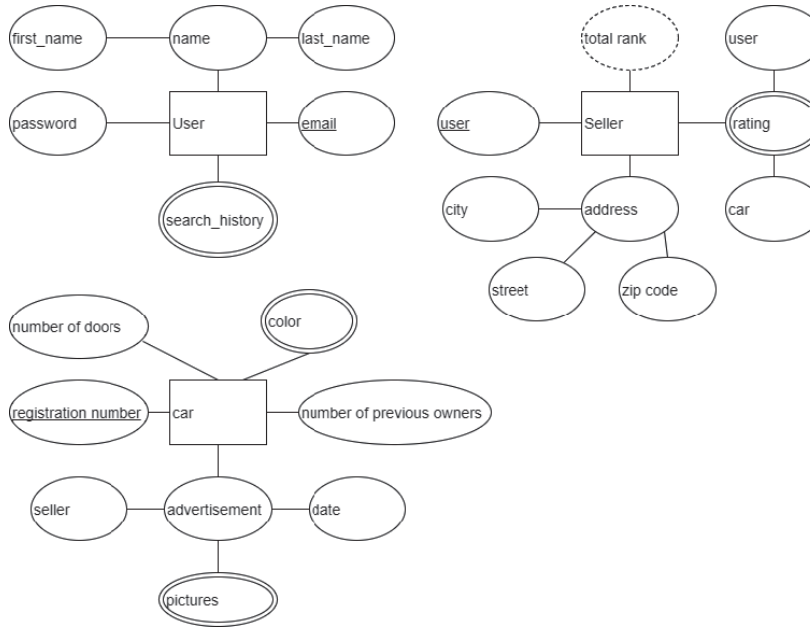


У сущности должно быть имя и набор атрибутов. Сущности бывают двух видов:

- **слабая сущность:** не имеет собственных ключевых атрибутов;
- **сильная**, или **регулярная**, **сущность:** имеет ключевой атрибут.

Слабая сущность обычно связана с какой-то сильной сущностью. Эта сильная сущность называется **идентифицирующей**. У слабых сущностей имеется частичный ключ, или дискриминатор, – атрибут, который в совокупности с идентифицирующей сущностью однозначно идентифицирует данную сущность. В нашем примере, если предположить, что при каждом поиске автомобиля пользователь задает новый ключ, ключ поиска будет частичным ключом. Символ слабой сущности – прямоугольник, ограниченный двойной линией.

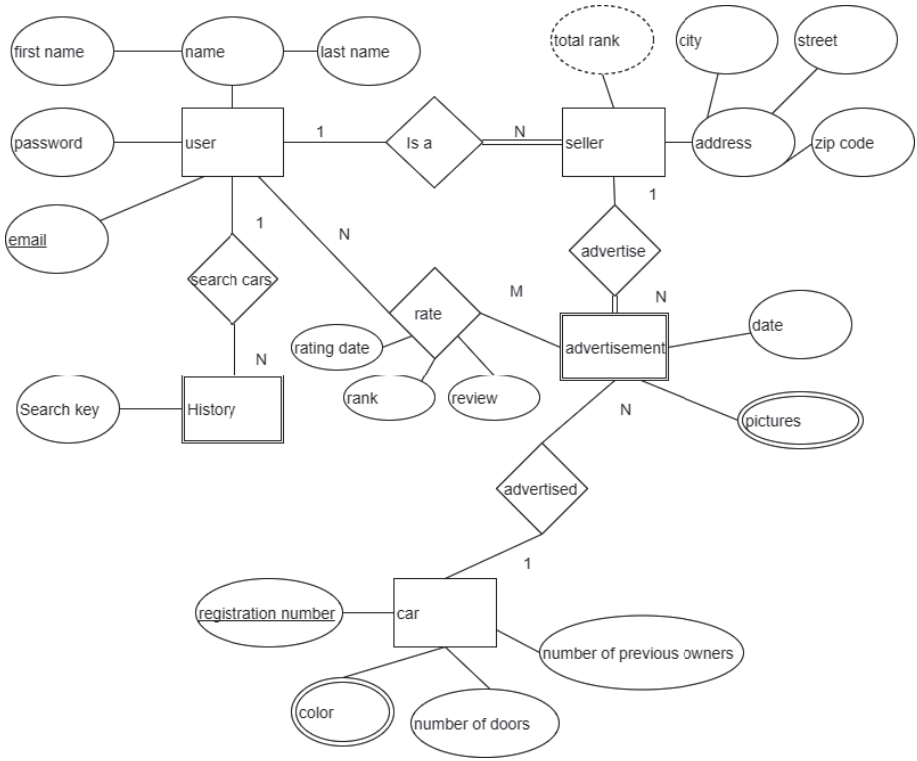
На следующем рисунке показан предварительный проект приложения. У сущности `user` несколько атрибутов. Атрибут `name` составной, а атрибут `email` ключевой. Сущность `seller` (продавец) – специализация сущности `user`. Атрибут `total rank` производный, он вычисляется путем объединения рейтингов, поставленных пользователями, и количества объявлений. Атрибут автомобиля `color` (цвет) многозначный. Пользователи могут выставлять рейтинги некоторым объявлениям продавца; это тернарное отношение, в котором участвуют три сущности: автомобиль, продавец и пользователь. Фотография автомобиля – многозначный атрибут сущности `advertisement` (объявление). Из рисунка видно, что объявления об одном автомобиле может дать несколько продавцов. Это реальная ситуация, потому что владелец может обратиться к нескольким посредникам с просьбой продать его автомобиль.



Если атрибут одной сущности ссылается на другую сущность, то образуется связь. В модели сущность-связь ссылки должны моделироваться не как атрибуты, а как связи или слабые сущности. Как и в случае сущностей, существует два вида связей: слабые и сильные. Слабая связь ассоциирует слабую сущность с другими сущностями. Связи, как и сущности, могут иметь атрибуты. В нашем примере объявление о продаже автомобиля дает продавец; дата объявления – свойство этой связи.

У связей имеется ограничение кардинальности, которое определяет, какие возможны комбинации участвующих в связи сущностей. Кардинальность связи объявления и продавца равна 1:N, т. е. каждое объявление дается одним продавцом, а каждый продавец может давать объявления о нескольких автомобилях. Связь между продавцом и пользователем называется полным участием (total participation) и обозначается двойной линией. Это означает, что продавец не может существовать сам по себе, он должен быть пользователем.

Ограничение кардинальности многие-ко-многим обозначается N:M, чтобы подчеркнуть, что количество участвующих сущностей по разные стороны связи может быть различным.



До сих пор мы обсуждали только базовые концепции диаграмм сущность-связь. Такие концепции, как нотация (min, max) в ограничениях кардинальности, тернарные (и n-арные) связи, обобщение, специализация и **расширенные диаграммы сущность-связь** (enhanced entity relation diagrams – **EER**), не обсуждались.

Отображение диаграмм сущность-связь на отношения

Правила отображения диаграммы сущность-связь на множество отношений (т. е. схему базы данных) почти прямолинейные, но не жесткие. Можно начать с моделирования сущности как атрибута, а затем уточнить связи. Атрибут, принадлежащий нескольким сущностям, можно выделить в независимую сущность. Ниже перечислены самые распространенные правила (список не полный).

- Отобразить регулярные сущности на отношения. Если у сущностей имеются составные атрибуты, то включить все части атрибутов. Выбрать один из ключевых атрибутов на роль первичного ключа.

- Отобразить слабые сущности на отношения, включить простые атрибуты и части составных атрибутов. Добавить внешний ключ для ссылки на идентифицирующую сущность. Первичный ключ обычно является комбинацией частичного и внешнего ключей.
- Если в отношении есть атрибут, связь с которым имеет кардинальность 1:1, то этот атрибут можно перенести в одну из участвующих сущностей.
- Если в отношении есть атрибут, связь с которым имеет кардинальность 1:N, то этот атрибут можно перенести в участвующую сущность со стороны, соответствующей N.
- Связям многие-ко-многим (N:M) поставить в соответствие новое отношение. Добавить внешние ключи для ссылки на участвующие сущности. Первичным ключом будет композиция внешних ключей.
- Многозначному атрибуту поставить в соответствие отношение. Добавить внешний ключ для ссылки на сущность, владеющую многозначным атрибутом. Первичным ключом будет композиция внешнего ключа и многозначного атрибута.

UML-диаграммы классов

Унифицированный язык моделирования (**UML**) – стандарт, разработанный организацией Object Management Group (OMG). UML-диаграммы широко применяются в программах моделирования. Разным видам моделирования соответствуют разные диаграммы, в том числе диаграммы классов, прецедентов, деятельности и реализации.

Диаграмма классов может описывать несколько типов ассоциаций, т. е. связей между классами. На ней можно изображать как атрибуты, так и методы. Диаграмму сущность-связь легко транслировать в UML-диаграмму классов. У диаграмм классов есть ряд достоинств:

- **обратное конструирование кода:** схему базы данных легко обратить и сгенерировать диаграмму классов;
- **моделирование объектов расширенной реляционной базы данных:** в современных реляционных базах есть ряд дополнительных типов объектов: последовательности, представления, индексы, функции и хранимые процедуры. UML-диаграммы классов способны представить такие типы.

Резюме

На проектирование системы управления базами данных оказывают влияние CAP-теоремы. Реляционные базы данных и базы данных NoSQL не соперничают, а дополняют друг друга. В одном приложении можно использовать базы данных разных типов. При этом хранилище типа «Ключ-значение» может быть использовано как механизм кеширования для повышения производительности реляционной базы данных.

На рынке преобладают реляционные и объектно-реляционные базы данных. В основе реляционных баз данных лежат понятие отношения и строгая математическая модель. Объектно-реляционные базы данных, в частности PostgreSQL, преодолевают ограничения реляционных баз данных за счет введения сложных типов данных, наследования и дополнительных расширений.

Фундаментальными для реляционных баз данных являются понятия отношения, кортежа и атрибута. Подобные базы гарантируют корректность и согласованность данных, применяя такие методы, как сущностная целостность, ограничения, ссылочная целостность и нормализация данных.

В следующей главе мы расскажем, как установить сервер и клиентские инструменты на различных платформах. Мы также познакомимся с такими средствами PostgreSQL, как встроенная поддержка репликации и развитая система типов.

Глава 2

PostgreSQL в действии

PostgreSQL, или просто **postgres**, – объектно-реляционная система управления базами данных с открытым исходным кодом. На первом месте в ней стоит расширяемость, техническое совершенство и совместимость. Она конкурирует с основными базами данными: Oracle, MySQL, SQL Server и другими. Используется в самых разных секторах, включая государственные учреждения, открытые и коммерческие продукты. Это кросс-платформенная СУБД, работающая в большинстве современных операционных систем, в т. ч. Windows, macOS и различных дистрибутивах Linux. Совместима со стандартами SQL и обладает всеми свойствами ACID.

В этой главе мы рассмотрим следующие темы:

- история PostgreSQL;
- некоторые продукты, берущие начало от PostgreSQL;
- компании, использующие PostgreSQL, и истории успеха;
- ситуации, в которых имеет смысл использовать PostgreSQL;
- архитектура, средства и возможности PostgreSQL;
- установка PostgreSQL в Linux и в Windows;
- клиентские инструменты PostgreSQL с упором на утилиту `psql`, которая чаще всего используется для повседневных работ.

Обзор PostgreSQL

PostgreSQL (http://www.PostgreSQL.org) обладает весьма развитой функциональностью и пригодна для работы в корпоративной среде, где требуются высокая производительность и масштабируемость. Вокруг PostgreSQL сложилось квалифицированное и радушное сообщество, а документация отличается высочайшим качеством.

История PostgreSQL

СУБД PostgreSQL начиналась как исследовательский проект в Калифорнийском университете в Беркли. С 1996 года она разрабатывалась сообществом, и до сих пор в разработке PostgreSQL принимают активное участие сообщества и университеты. Перечислим основные исторические вехи:

- **1977–1985, проект Ingres:** Майкл Стоунбрейкер (Michael Stonebraker) создал РСУБД на основе формальной реляционной модели;
- **1986–1994, postgres:** Майкл Стоунбрейкер (Michael Stonebraker) создал postgres, чтобы поддержать сложные типы данных и объектно-реляционную модель;
- **1995, Postgres95:** Эндрю Ю (Andrew Yu) и Джолли Чен (Jolly Chen) модифицировали язык запросов PostQUEL, являвшийся частью postgres, превратив его в расширенное подмножество SQL;
- **1996, PostgreSQL:** группа разработчиков потратила много времени и труда, чтобы стабилизировать postgres95. Первая версия с открытым исходным кодом была выпущена 29 января 1997 года. Поскольку проект стал открытым и в него были добавлены новые возможности и дополнения, название postgres95 изменили на PostgreSQL.

Первая версия PostgreSQL имела номер 6, что можно оправдать несколькими годами напряженных исследований и разработок. Будучи проектом с очень хорошей репутацией, PostgreSQL привлекла сотни разработчиков. В настоящее время PostgreSQL может похвастаться бесчисленными расширениями и очень активным сообществом.

Преимущества PostgreSQL

PostgreSQL обладает многими возможностями, которые привлекают разработчиков, администраторов, архитекторов и компании.

Преимущества PostgreSQL с точки зрения бизнеса

PostgreSQL – бесплатное ПО с открытым исходным кодом. Она выпускается на условиях весьма либеральной лицензии PostgreSQL. СУБД не может быть монополизирована и приобретена, что дает компаниям следующие преимущества:

- за лицензирование не нужно платить;
- количество развернутых экземпляров PostgreSQL не ограничено;
- более выгодная бизнес-модель;
- PostgreSQL совместима со стандартами SQL, поэтому нетрудно найти профессиональных разработчиков. PostgreSQL несложна для освоения, а перенос кода из другой СУБД обходится недорого. Кроме того, администрирование PostgreSQL легко автоматизировать, что позволяет существенно сэкономить на зарплате персоналу;
- PostgreSQL – кросс-платформенное ПО, к нему существуют интерфейсы из всех современных языков программирования, поэтому нет нужды изменять политику компании в части набора используемого ПО;
- PostgreSQL хорошо масштабируется и обеспечивает высокую производительность;
- PostgreSQL очень надежна, аварийное завершение случается редко. Кроме того, PostgreSQL в полной мере поддерживает свойства ACID, т. е.

устойчива к некоторым отказам оборудования. К тому же СУБД можно сконфигурировать и установить как кластер для обеспечения **высокой доступности**.

Преимущества PostgreSQL с точки зрения пользователя

PostgreSQL весьма привлекательна для разработчиков, администраторов и архитекторов. Развитая функциональность позволяет гибко решать различные задачи. Перечислим некоторые ее преимущества с точки зрения разработчика:

- новые версии выходят ежегодно; к настоящему моменту, начиная с версии 6.0, было выпущено 24 основные версии;
- наличие великолепной документации и активного сообщества позволяет быстро находить ответы на возникающие вопросы. Руководство по PostgreSQL насчитывает свыше 2500 страниц;
- богатый арсенал расширений позволяет разработчику сосредоточиться на логике приложения, а также оперативно реагировать на изменение требований;
- исходный код бесплатен. Его можно адаптировать под свои нужды без особого труда;
- развитая экосистема клиентов и административных средств позволяет легко и быстро выполнять такие рутинные задачи, как описание объектов базы данных, экспорт и импорт данных, резервное копирование и восстановление базы;
- решение административных задач не занимает много времени и поддается автоматизации;
- PostgreSQL легко интегрируется с другими СУБД, что открывает возможность для гибкой реализации программных проектов.

Применения PostgreSQL

PostgreSQL применяется в различных ситуациях. Основные области применения PostgreSQL можно отнести к двум категориям:

- обработка транзакций в реальном времени **(OLTP):** OLTP характеризуется большим количеством операций вставки, обновления и удаления, очень высокой скоростью работы и поддержанием целостности данных в многопользовательской среде. Производительность измеряется количеством транзакций в секунду;
- интерактивная аналитическая обработка **(OLAP):** OLAP характеризуется небольшим количеством сложных запросов, включающих агрегирование данных, получение гигантских объемов данных из разных источников в разных форматах, добычу данных и анализ исторических данных.

OLTP применяется для моделирования таких приложений, как **управление взаимоотношениями с клиентами** (**CRM**). Например, сайт торговли автомобилями из главы 1 – пример OLTP-приложения. OLAP находит применение в задачах бизнес-аналитики, поддержки принятия решений, генерации отчетов и планирования. Размер базы данных для OLTP сравнительно невелик

по сравнению с базой данных OLAP. При проектировании базы данных OLTP обычно придерживаются концепций реляционной модели, в т. ч. нормализации, тогда как базы данных OLAP меньше похожи на реляционные; схема часто имеет вид звезды или снежинки, а данные намеренно денормализованы.

В примере сайта торговли автомобилями можно было бы завести вторую базу для хранения исторических данных о продавцах и пользователях, чтобы анализировать предпочтения пользователей и активность продавцов. Это был бы пример OLAP-приложения.

В отличие от OLTP, основная операция OLAP – выборка и анализ данных. Данные для OLAP часто генерируются процессом **извлечения, преобразования и очистки** (**ETL**), который загружает данные в базу OLAP из разных источников в разных форматах. Для OLTP-приложений достаточно PostgreSQL в стандартной комплектации. А для поддержки OLAP имеется много расширений и инструментов, в т. ч. **адаптеры внешних данных** (foreign data wrappers – **FDW**), секционирование таблиц, а в последних версиях еще и параллельное выполнение запросов.

Истории успеха

PostgreSQL используется во многих отраслях, например телекоммуникации, медицина, географические приложения и электронная коммерция. Есть много компаний, оказывающих коммерческие консультационные услуги, например в переходе с коммерческой РСУБД на PostgreSQL с целью уменьшения лицензионных платежей. Такие компании часто оказывают влияние на развитие PostgreSQL, разрабатывая новые средства. Перечислим несколько компаний, применяющих PostgreSQL:

- в Skype PostgreSQL используется для хранения чатов и истории действий. Компания Skype разработала для PostgreSQL много инструментов под общим названием **SkyTools**;
- Instagram – социальная сеть для обмена фотографиями;
- Американское химическое общество (**ACS**) использует PostgreSQL для хранения архивов журнала объемом свыше терабайта.

Помимо этих компаний, PostgreSQL используется в HP, VMware и Heroku, а также многими научными обществами и организациями, в частности NASA.

Ответвления

Ответвлением называется независимый программный проект, основанный на каком-то другом проекте. От PostgreSQL существует более 20 ответвлений; расширяемый API этому весьма способствует. На протяжении многих лет различные группы создавали ответвления и затем включали результаты своей работы в PostgreSQL.

- **HadoopDB** – гибрид PostgreSQL с технологиями MapReduce, ориентированный на аналитику. Ниже перечислены наиболее популярные ответвления от PostgreSQL:

- в **Greenplum** используется архитектура без разделения ресурсов с **массово параллельной обработкой** (**MPP**). Применяется для создания хранилищ данных и аналитики. Greenplum начиналась как коммерческий проект, но в 2015 году ее код был раскрыт;
- **EnterpriseDB Advanced Server** – коммерческая СУБД, дополняющая PostgreSQL средствами Oracle;
- **Postgres-XC (eXtensible Cluster)** – кластер на основе PostgreSQL с несколькими ведущими узлами, построенный на базе архитектуры без разделения ресурсов. Упор сделан на масштабируемость записи, приложениям предоставляется тот же API, что и в PostgreSQL;
- **Vertica** – столбцовая база данных, разработка которой была начата Майклом Стоунбрейкером в 2005 году. В 2011 году проект был приобретен компанией HP. Vertica заимствовала у PostgreSQL синтаксический анализатор SQL, семантический анализатор и стандартные методы переписывания SQL-запросов;
- **Netezza** – популярное хранилище данных, начавшее жизнь как ответвление от PostgreSQL;
- **Amazon Redshift** – популярное хранилище данных, основанное на PostgreSQL 8.0.2. Предназначено в основном для приложений OLAP.

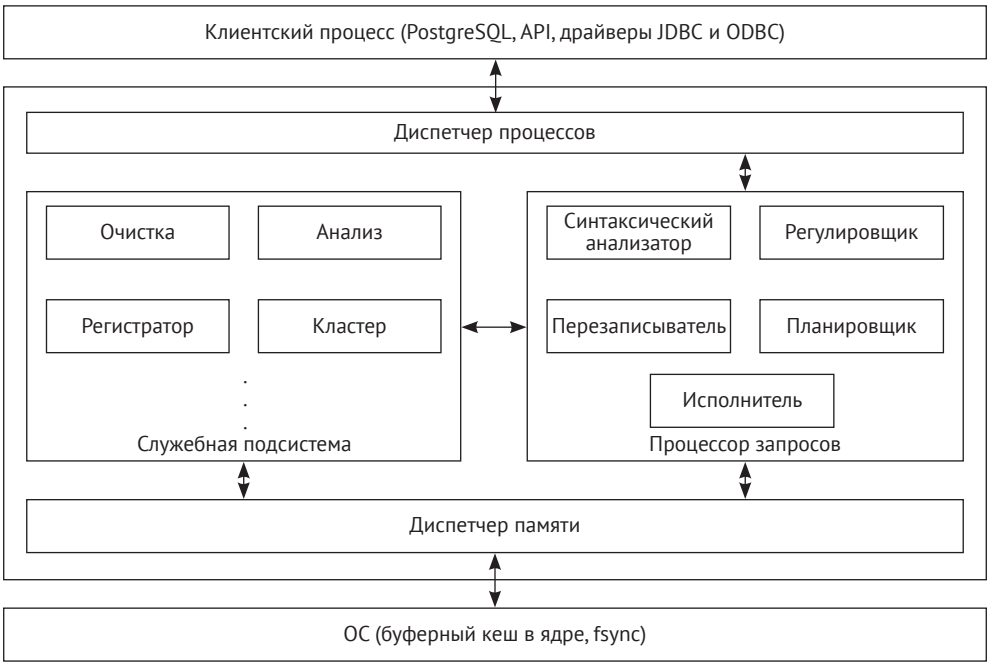
Архитектура PostgreSQL

В PostgreSQL используется клиент-серверная архитектура, когда клиент и сервер могут находиться в различных узлах. Обмен данными между клиентом и сервером обычно ведется по протоколу TCP/IP или через Linux-сокеты. Сервер PostgreSQL может одновременно обрабатывать несколько подключений клиентов. Типичная программа, работающая с PostgreSQL, состоит из следующих процессов операционной системы:

- **клиентский процесс**: фронтальная часть приложения отправляет запросы базе данных. В этой роли может выступать веб-сервер, которому нужно построить веб-страницу, или командная утилита для решения задач обслуживания. В состав PostgreSQL входят фронтальные команды `psql`, `createdb`, `dropdb`, `createuser` и другие;
- **серверный процесс**: управляет файлами базы данных, принимает запросы на подключение от клиентов и выполняет действия по их требованию. Серверный процесс называется postgres. PostgreSQL создает новый процесс для каждого подключения, поэтому процессы клиента и сервера общаются между собой, не прерывая главный серверный процесс. Серверные процессы существуют в течение ограниченного времени – до завершения соединения с клиентом.

Описанная концептуальная архитектура PostgreSQL дает представление о возможностях СУБД, а также о взаимодействии с клиентами и операционной системой. В сервере можно приблизительно выделить четыре подсистемы:

- **диспетчер процессов:** управляет клиентскими подключениями, в частности создает и завершает процессы;
- **процессор запросов:** запрос, отправленный клиентом PostgreSQL, разбирается синтаксическим анализатором, после чего подсистема регулировщика определяет тип запроса. Служебные запросы передаются служебной подсистеме. Запросы выборки, обновления и удаления модифицируются перезаписывателем, после чего планировщик генерирует план выполнения. Наконец, запрос выполняется, и результат возвращается клиенту;
- **служебная подсистема:** предоставляет средства обслуживания базы данных, например: затребование места в системе хранения, обновление статистики, экспорт и импорт данных в определенном формате, протоколирование путем обращения к регистратору;
- **диспетчер памяти:** отвечает за управление кешем и дисковыми буферами, выделяет память.



Почти все компоненты PostgreSQL, включая регистратор, планировщик, анализатор статистики и диспетчер памяти, допускают конфигурирование. Конфигурация PostgreSQL выбирается в соответствии с характером приложения: OLAP или OLTP.

Сообщество PostgreSQL

PostgreSQL располагает весьма активным и организованным сообществом, всегда готовым прийти на помощь. За последние 8 лет сообщество выпустило восемь основных версий. Разработчикам рассылаются объявления в еженедельном бюллетене.

Существуют десятки списков рассылки, сгруппированных по категориям, например: пользователь, разработчик, ассоциация. В качестве примеров упомянем списки pgsql-general, psql-doc и psql-bugs. Для начинающих особенно важен список pgsql-general. В нем обсуждаются любые вопросы, касающиеся установки, настройки, основ администрирования, возможностей PostgreSQL, ведутся дискуссии на общие темы – в общем, всё, кроме ошибок.

Сообщество PostgreSQL поддерживает службу агрегирования блогов **Planet PostgreSQL** (planet.PostgreSQL.org). Некоторые разработчики и компании с ее помощью делятся своими знаниями и опытом.

Возможности PostgreSQL

PostgreSQL обеспечивает сервис корпоративного уровня, гарантирующий непрерывность бизнеса. Например, поддерживается репликация, позволяющая обеспечить высокую доступность. PostgreSQL может получать данные из различных источников, а затем применять к ним процесс извлечения, преобразования и загрузки (**ETL**). Безопасность в PostgreSQL считается неотъемлемой чертой, а не функцией, которую хорошо бы иметь. Обновления безопасности поставляются в дополнительных (минорных) релизах. Наконец, в PostgreSQL реализовано бессчетное количество дополнений к SQL, крайне полезных расширений и интерактивных инструментов.

Репликация

Репликация позволяет организовать перенос данных с одного сервера на другой. Ее основные цели:

- **высокая доступность:** второй сервер может взять на себя обслуживание, если первый выйдет из строя;
- **балансировка нагрузки:** одни и те же запросы могут обслуживать несколько серверов;
- **ускорение работы:** для повышения производительности запрос выполняется сразу на нескольких машинах.

В PostgreSQL изначально встроена **потоковая репликация** типа ведущий-ведомый, основанная на трансляции файлов журналов. Это двоичная техника репликации, поскольку команды SQL не анализируются. Принцип заключается в том, чтобы сначала сделать снимок ведущего узла, а затем доставлять изменения – WAL-файлы – с ведущего узла на ведомый, где они воспроизводятся. На ведущем узле можно выполнять операции чтения и записи, а на ведомом – только операции чтения. Потоковую репликацию довольно просто настроить;

поддерживаются синхронная, асинхронная и каскадная репликации. В синхронном режиме операция модификации данных должна быть зафиксирована на всех серверах, и только тогда она считается успешной. В каскадном режиме ведомый узел является ведущим для другой реплики. Это позволяет горизонтально масштабировать PostgreSQL на операциях чтения.

В PostgreSQL 10 добавлена логическая репликация. В отличие от потоковой репликации, при которой двоичные данные копируются побитово, механизм логической репликации преобразует WAL-файлы в набор логических изменений. Это расширяет контроль над репликацией, например можно применять на этом этапе фильтры. Так, логическая репликация позволяет скопировать часть данных, тогда как в случае потоковой репликации ведомый узел является точной копией ведущего. Другой важный аспект логической репликации – возможность создания на ведомом сервере дополнительных индексов и временных таблиц. Наконец, мы можем реплицировать данные с нескольких серверов, объединив их на одном сервере.

Помимо встроенных в PostgreSQL методов репликации, существует ряд решений с открытым исходным кодом, ориентированных на различные рабочие нагрузки.

- **Slony-1:** система репликации с одним ведущим узлом и несколькими ведомыми. В отличие от встроенной в PostgreSQL, может использоваться с разными версиями сервера. Так, можно реплицировать данные из версии 9.6 в версию 10. Slony очень полезна для перехода на новую версию сервера без простоя.
- **pgpool-II:** это ПО промежуточного уровня, расположенное между PostgreSQL и клиентом. Помимо репликации, оно может использоваться для организации пула соединений и параллельного выполнения запросов.
- **Распределенное реплицированное блочное устройство** (Distributed Replicated Block Device – **DRBD**): общее решение для обеспечения высокой доступности. Можно рассматривать как сетевой аналог RAID-1.

Безопасность

PostgreSQL поддерживает несколько методов аутентификации, включая доверительные отношения, пароль, LDAP, GSSAPI, SSPI, Kerberos, по идентификатору в операционной системе, RADIUS, сертификат и PAM. Все уязвимости в базе данных перечислены на странице информации о безопасности PostgreSQL по адресу http://www.PostgreSQL.org/support/security/, где указаны версия СУБД, класс уязвимости и уязвимый компонент.

Обновления безопасности PostgreSQL распространяются в виде дополнительных версий и исправляются в следующей основной версии. Публикация в виде дополнительной версии упрощает администратору поддержание PostgreSQL в безопасном состоянии с минимальным временем простоя.

PostgreSQL управляет доступом к объектам базы данных на нескольких уровнях, в том числе база данных, таблица, представление, функция, после-

довательность и столбец. Это позволяет организовать детальный контроль авторизации.

PostgreSQL поддерживает аппаратное шифрование для защиты данных. Кроме того, определенные данные можно шифровать с помощью расширения pgcrypto.

Расширения

Механизм расширений позволяет определять новые типы данных. Для загрузки расширений в базу существует команда `CREATE EXTENSION`. Кроме того, имеется центральный репозиторий (**PGXN**) по адресу www.pgxn.org, позволяющий просматривать и загружать расширения. На этапе установки двоичного дистрибутива PostgreSQL предлагается пакет `postgresql-contib`, содержащий много полезных расширений, например `tablefunc`, который включает среди прочего сводные таблицы и расширение `pgcrypto` (общие сведения можно найти в файле `README` в установочном каталоге).

Именно благодаря поддержке расширений PostgreSQL располагает следующими возможностями:

- **типы данных PostgreSQL.** В PostgreSQL весьма богатая система типов данных. Изначально поддерживаются как примитивные, так и некоторые структурные типы, в частности массивы. Дополнительно поддерживаются следующие сложные типы данных:
  - **геометрические типы данных:** включая точку (point), отрезок (lseg), ломаную (path), многоугольник (polygon) и прямоугольник (box);
  - **сетевые адреса:** cidr, inet и macaddr;
  - **tsvector и tsquery:** отсортированный список лексем, лежащий в основе полнотекстового поиска;
  - **универсальный уникальный идентификатор (UUID):** UUID решает много проблем, связанных с базами данных, в т. ч. проблему автономной генерации данных;
  - **NoSQL:** поддерживается несколько типов данных из категории NoSQL, включая `XML`, `Hstore`, `JSONB`;
  - `Enum`. Типы диапазона и домена позволяют задавать различные ограничения: множество допустимых значений, ограничения на диапазон и проверочные ограничения;
  - составные типы данных, в которых атрибут состоит из нескольких частей;
- **поддерживаемые языки.** PostgreSQL позволяет писать функции на нескольких языках. Сообщество поддерживает языки SQL, C, Python, PL/pgSQL, Perl и Tcl. Кроме них, многие языки поддерживаются сторонними разработчиками, среди них Java, R, PHP, Ruby и язык оболочки UNIX.

В следующем примере показано, как создать новый составной тип `phone_number`. Два телефона считаются одинаковыми, если совпадают код региона и абонентский номер, для проверки этого написан оператор сравнения на равенство.

```
-- Создание составного типа данных
CREATE TYPE phone_number AS (
    area_code varchar(3),
    line_number varchar(7)
);
CREATE OR REPLACE FUNCTION phone_number_equal (phone_number,phone_number)
RETURNS boolean AS $$
BEGIN
    IF $1.area_code=$2.area_code AND $1.line_number=$2.line_number THEN
        RETURN TRUE ;
    ELSE
        RETURN FALSE;
    END IF;
END; $$ LANGUAGE plpgsql;
CREATE OPERATOR = (
    LEFTARG = phone_number,
    RIGHTARG = phone_number,
    PROCEDURE = phone_number_equal
);
-- Для тестирования
SELECT row('123','222244')::phone_number = row('1','222244')::phone_number;
```

Здесь создается новый тип данных `phone_number`, состоящий из двух атомарных частей: `area_code` и `line_number`. Перегруженный оператор `=` сравнивает значения этого типа на равенство. Чтобы определить поведение оператора `=`, мы должны описать его аргументы и функцию, которая их обрабатывает. В данном случае аргументы имеют тип `phone_number`, а функция называется `phone_number_equal`.

Возможности NoSQL

PostgreSQL – не просто реляционная база и язык SQL. Теперь в PostgreSQL нашли приют и типы данных, характерные для NoSQL. Богатый арсенал средств PostgreSQL и возможность создавать бессхемные хранилища данных открывают путь к разработке надежных и гибких приложений.

PostgreSQL поддерживает тип **JSON**, который нередко применяется для обмена данными между различными системами в современных REST-совместимых веб-приложениях. В версии PostgreSQL 9.4 появился также структурированный формат JSONB для хранения JSON-документов. Этот тип данных устраняет необходимость разбирать JSON-документ перед записью в базу данных. Иными словами, PostgreSQL может вставлять JSON-документы с такой же скоростью, как документные базы данных, не отказываясь при этом от свойств ACID. В версии PostgreSQL 9.5 добавлено несколько функций для еще большего упрощения работы с JSON-документами. В версии 10 поддержан полнотекстовый поиск в JSON- и JSONB-документах.

Расширение hstore поддерживает хранение пар ключ-значение. Это расширение предназначено для хранения слабоструктурированных данных и в некоторых ситуациях применяется, чтобы уменьшить количество редко используемых атрибутов, которые обычно содержат null.

Наконец, PostgreSQL поддерживает очень гибкий тип XML, который часто используется для определения формата документов. XML лежит в основе форматов RSS, Atom, SOAP и XHTML. PostgreSQL поддерживает несколько функций для создания XML-документов. Поддерживается также технология xpath для поиска в XML-документе.

Адаптеры внешних данных

В 2011 году была выпущена версия PostgreSQL 9.1 с поддержкой чтения по стандарту управления внешними данными в SQL (Management of External Data – MED) ISO/IEC 9075-9:2003. Стандарт SQL/MED определяет адаптеры внешних данных (foreign data wrappers – FDW), позволяющие реляционной базе управлять внешними данными. Благодаря адаптерам внешних данных можно обеспечить интеграцию в системе федеративных баз данных. PostgreSQL поддерживает РСУБД, NoSQL и адаптеры внешних данных для Oracle, Redis, Mongodb и файлов с разделителями.

Например, FDW могут быть полезны, когда имеется один сервер базы данных для целей аналитики, а результаты передаются на другой сервер, где играют роль кеша.

Кроме того, FDW можно использовать для тестирования изменения данных. Пусть имеются две базы данных, к одной из которых применено некое исправление. FDW позволяет оценить эффект исправления, сравнив данные в обеих базах. Это средство предназначено для совместного доступа к данным в разных базах PostgreSQL. Поддерживаются операции `SELECT`, `INSERT`, `UPDATE` и `DELETE` для внешних таблиц.

В примере ниже показано, как с помощью FDW читать файл в формате **CSV** (с полями, разделенными запятыми); например, это может пригодиться для разбора журналов. Допустим, требуется прочитать журналы базы данных, формируемые PostgreSQL. Это очень полезно в производственной среде для получения статистики выполнения запросов (структура файла документирована по адресу https://www.PostgreSQL.org/docs/current/static/runtime-configlogging.html). Чтобы включить протоколирование в формате CSV, необходимо изменить следующие значения в файле `PostgreSQL.conf`. Для простоты мы протоколируем все команды, но в производственной среде поступать так не рекомендуется.

```
log_destination = 'csvlog'
logging_collector = on
log_filename = 'PostgreSQL.log'
log_statement = 'all'
```

Чтобы изменения вступили в силу, необходимо перезапустить PostgreSQL:

```
$sudo service PostgreSQL restart
```

Чтобы установить FDW для файлов, выполните команду:

```
postgres=# CREATE EXTENSION file_fdw ;
CREATE EXTENSION
```

Для доступа к файлу нужно создать FDW-сервер:

```
postgres=# CREATE SERVER fileserver FOREIGN DATA WRAPPER file_fdw;
CREATE SERVER
```

Кроме того, нужно создать FDW-таблицу и связать ее с файлом журнала; в нашем случае он находится в папке кластерного каталога PostgreSQL:

```
postgres=# CREATE FOREIGN TABLE postgres_log
(
  log_time timestamp(3) with time zone,
  user_name text,
  database_name text,
  process_id integer,
  connection_from text,
  session_id text,
  session_line_num bigint,
  command_tag text,
  session_start_time timestamp with time zone,
  virtual_transaction_id text,
  transaction_id bigint,
  error_severity text,
  sql_state_code text,
  message text,
  detail text,
  hint text,
  internal_query text,
  internal_query_pos integer,
  context text,
  query text,
  query_pos integer,
  location text,
  application_name text
) SERVER fileserver OPTIONS ( filename
'/var/lib/PostgreSQL/10/main/log/PostgreSQL.csv', header 'true', format 'csv' );
    CREATE FOREIGN TABLE
```

Для тестирования примера подсчитаем количество строк в журнале:

```
postgres=# SELECT count(*) FROM postgres_log;
  count
-------
    62
(1 row)
```

Производительность

Высокая производительность PostgreSQL доказана на практике. Для повышения уровня параллелизма и масштабируемости применяется несколько технических приемов, в том числе:

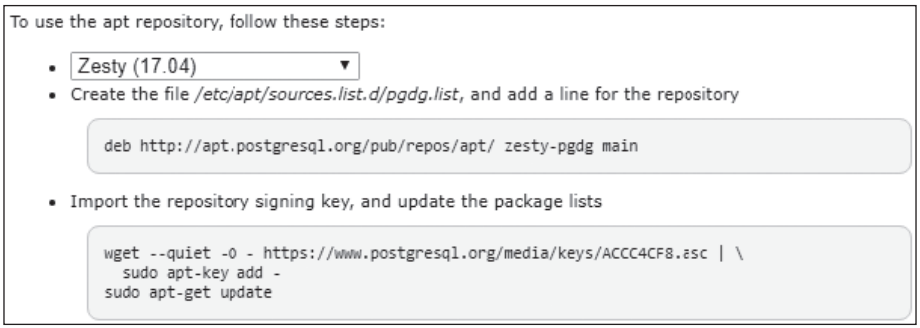
- **система блокировок:** PostgreSQL предоставляет несколько типов блокировок на уровне таблиц и строк. Для предотвращения длительных задержек и повышения уровня параллелизма используются мелкие блокировки;

- **индексы:** PostgreSQL предоставляет шесть типов индексов: B-дерево, хеш, **обобщенный инвертированный индекс** (**GIN**), **обобщенное дерево поиска** (**GiST**), SP-GiST и **индекс блочных диапазонов** (Block Range Index – **BRIN**). Индексы разных типов предназначены для решения разных задач. Например, B-дерево эффективно для сравнения на равенство и проверки попадания в диапазон. GIST-индексы используются для полнотекстового поиска и геопространственных данных. PostgreSQL поддерживает частичные, уникальные и многостолбцовые индексы, а также индексы по выражениям и классы операторов;
- **команды explain, analyze, vacuum и cluster:** в PostgreSQL имеется несколько команд для повышения производительности и обеспечения прозрачности. Команда `explain` показывает план выполнения SQL-команды. Можно изменить те или иные параметры, например настройки памяти, а затем сравнить планы до и после изменения. Команда `analyze` служит для сбора статистики по таблицам и столбцам. Команда `vacuum` применяется для сборки мусора и возврата неиспользуемого дискового пространства операционной системе. Команда `cluster` физически организует данные на диске. Все эти команды можно сконфигурировать в зависимости от характера нагрузки;
- **наследование таблиц и исключение с учетом ограничений:** наследование таблиц позволяет легко создавать таблицы с похожей структурой. Этот механизм дает возможность хранить подмножества данных по определенному критерию; иногда это резко ускоряет выборку информации, потому что для ответа на запрос нужно просматривать только подмножество данных;
- **развитые конструкции SQL:** PostgreSQL поддерживает передовые конструкции SQL: коррелированные и некоррелированные подзапросы, **общие табличные выражения** (common table expression – **CTE**), оконные функции и рекурсивные запросы. Освоив эти конструкции, разработчик сможет очень быстро писать лаконичный и изящный код на SQL и создавать сложные запросы с минимумом усилий. Сообщество продолжает развивать SQL в каждой новой версии; в версии 9.6 добавлены три фразы: `GROUPING SETS`, `CUBE` и `ROLLUP`.

Установка PostgreSQL

PostgreSQL устанавливается практически на все современные операционные системы: все сравнительно недавние дистрибутивы Linux, Windows 2000 SP4 и старше, FreeBSD, OpenBSD, macOS, AIX и Solaris. Кроме того, PostgreSQL работает на процессорах разной архитектуры, включая x86, x86_64, IA64 и другие. На странице http://buildfarm.PostgreSQL.org можно проверить, поддерживается ли конкретная платформа (операционная система + архитектура ЦП). Можно скачать и установить двоичный дистрибутив или самостоятельно откомпилировать исходный код PostgreSQL.

В целях автоматизации установки PostgreSQL и уменьшения нагрузки на администратора сервера рекомендуется использовать двоичный комплект поставки, который может установить менеджер пакетов операционной системы. У этого подхода один недостаток: версия будет не самой последней. Впрочем, на официальном сайте https://www.PostgreSQL.org/download/ выложены дистрибутивы для наиболее распространенных платформ, включая BSD, Linux, macOS, Solaris и Windows, и инструкции по установке. На рисунке ниже показано, как добавить репозиторий **APT** в систему Ubuntu Zesty. Отметим, что инструкции по установке генерируются для конкретного имени ОС.



Установка PostgreSQL с помощью менеджера пакетов APT

Программа **Advanced Package Tool** (**APT**) применяется для установки/удаления программ в дистрибутиве Debian Linux и основанных на нем, в т. ч. Ubuntu.

Как уже отмечалось, последние двоичные версии PostgreSQL могут отсутствовать в официальных репозиториях Debian и Ubuntu. Чтобы настроить apt-репозиторий PostgreSQL, выполните следующие команды:

```
$sudo sh -c 'echo "deb http://apt.PostgreSQL.org/pub/repos/apt/
$(lsb_release -cs)-pgdg main" > /etc/apt/sources.list.d/pgdg.list'
$wget --quiet -O - https://www.PostgreSQL.org/media/keys/ACCC4CF8.asc |
sudo apt-key add -
$sudo apt-get update
```

После добавления нового репозитория рекомендуется обновить систему:

```
$sudo apt-get upgrade
```

Установка клиентов

Если сервер PostgreSQL уже установлен и вы хотите взаимодействовать с ним, то понадобится установить пакет `postgresql-client`. Для этого откройте терминал и выполните команду:

```
sudo apt-get install PostgreSQL-client-10
```

В результате на машину будет установлено несколько клиентских программ, включая интерактивный терминал PostgreSQL (`psql`). Чтобы узнать, какие про-

граммы были установлены, зайдите в установочный каталог. Отметим, что путь к этому каталогу зависит от версии PostgreSQL и операционной системы.

```
ls /usr/lib/PostgreSQL/10/bin/
clusterdb createdb createuser dropdb dropuser pg_basebackup pg_dump
pg_dumpall pg_isready pg_receiveWAL pg_recvlogical pg_restore psql
reindexdb vacuumdb
```

Чтобы подключиться к работающему серверу PostgreSQL с помощью `psql`, необходимо задать строку соединения, включающую имя узла, имя базы данных, номер порта и имя пользователя.

Есть еще один мощный клиент с графическим интерфейсом – `pgadmin4`, который используется для администрирования и разработки. Начинающие предпочитают pgAdmin, а `psql` применяется в скриптах оболочки.

У `pgadmin4` весьма развитая функциональность. Он кросс-платформенный, работает в браузере, имеет серверный и настольный режимы и т. д. Чтобы установить pgAdmin в настольном режиме, выполните такие команды:

```
# установить виртуальную среду и pip
sudo apt-get install pip
sudo pip install --upgrade pip
sudo pip install virtualenvwrapper

# создать и активировать виртуальную среду
virtualenv pgadmin && cd pgadmin && source bin/activate

# получить и установить последнюю версию pgadmin
wget
https://ftp.PostgreSQL.org/pub/pgadmin/pgadmin4/v2.0/pip/pgadmin4-2.0-py2.p
y3-none-any.whl
pip install pgadmin4-2.0-py2.py3-none-any.whl

# создать и выполнить конфигурационный файл на основе config.py
cp ./lib/python2.7/site-packages/pgadmin4/config.py ./lib/python2.7/sitepackages/
pgadmin4/config-local.py
python ./lib/python2.7/site-packages/pgadmin4/pgAdmin4.py
```

Для взаимодействия с PostgreSQL можно использовать также программу DBeaver (https://dbeaver.jkiss.org). Это универсальный SQL-клиент, работающий со многими базами данных, в т. ч. и с PostgreSQL.

Установка сервера

Для установки сервера выполните следующую команду:

```
$sudo apt-get install PostgreSQL-10
```

Процедура установки дает информацию о местонахождении конфигурационных файлов и данных PostgreSQL, локали, номере порта и состоянии сервера:

```
Creating config file /etc/PostgreSQL-common/createcluster.conf with new version
Setting up PostgreSQL-10 (10.0-1.pgdg80+1) ...
Creating new PostgreSQL cluster 10/main ...
/usr/lib/PostgreSQL/10/bin/initdb -D /var/lib/PostgreSQL/10/main --authlocal
peer --auth-host md5
```

```
The files belonging to this database system will be owned by user "postgres".
This user must also own the server process.

The database cluster will be initialized with locale "en_US.UTF-8".
The default database encoding has accordingly been set to "UTF8".
The default text search configuration will be set to "english".

Data page checksums are disabled.

fixing permissions on existing directory /var/lib/PostgreSQL/10/main ... ok
creating subdirectories ... ok
selecting default max_connections ... 100
selecting default shared_buffers ... 128MB
selecting dynamic shared memory implementation ... posix
creating configuration files ... ok
running bootstrap script ... ok
performing post-bootstrap initialization ... ok
syncing data to disk ... ok

Success. You can now start the database server using:

    /usr/lib/PostgreSQL/10/bin/pg_ctl -D /var/lib/PostgreSQL/10/main -l logfile start

Ver Cluster Port Status Owner Data directory Log file
10 main 5432 down postgres /var/lib/PostgreSQL/10/main
/var/log/PostgreSQL/PostgreSQL-10-main.log
update-alternatives: using
/usr/share/PostgreSQL/10/man/man1/postmaster.1.gz to provide
/usr/share/man/man1/postmaster.1.gz (postmaster.1.gz) in auto mode
Processing triggers for libc-bin (2.19-18+deb8u10) ...
Processing triggers for systemd (215-17+deb8u7) ...
```

PostgreSQL инициализирует на диске область, называемую кластером баз данных. Кластер состоит из набора баз данных, управляемых одним экземпляром сервера. Следовательно, если создать несколько кластеров, то можно будет запустить на одном физическом компьютере несколько экземпляров PostgreSQL, одной и той же или разных версий.

По умолчанию с кластером баз данных ассоциируется локаль, установленная в операционной системе, но можно явно задать другую локаль в момент создания кластера.

Еще один важный пакет `postgresql-contrib` содержит расширения, одобренные сообществом. Часто этот пакет устанавливается отдельно. Но в apt-репозитории PostgreSQL 10 APT он объединен с пакетом сервера.

Для проверки корректности установки поищите с помощью утилиты pgrep процессы `postgres`:

```
$pgrep -a postgres
3807 /usr/lib/PostgreSQL/10/bin/postgres -D /var/lib/PostgreSQL/10/main -c
config_file=/etc/PostgreSQL/10/main/PostgreSQL.conf
3809 postgres: 10/main: checkpointer process
3810 postgres: 10/main: writer process
3811 postgres: 10/main: WAL writer process
```

```
3812 postgres: 10/main: autovacuum launcher process
3813 postgres: 10/main: stats collector process
3814 postgres: 10/main: bgworker: logical replication launcher
```

Здесь мы видим главный серверный процесс с двумя параметрами: `-D` задает кластер баз данных, а `-c` – конфигурационный файл. Кроме того, показано несколько служебных процессов, в т. ч. `autovacuum` и сборщик статистики.

Наконец, можно установить сервер и клиент одной командой:

```
sudo apt-get install PostgreSQL-10 PostgreSQL-client-10
```

Базовая конфигурация сервера

Для доступа к серверу необходимо понимать механизм аутентификации в PostgreSQL. В системах на базе Linux к серверу можно подключиться через UNIX-сокет или по протоколу TCP/IP. Кроме того, PostgreSQL поддерживает много методов аутентификации.

В процессе установки сервера создается новый пользователь операционной системы и пользователь базы данных с именем `postgres`. Этот пользователь может подключаться к серверу, применяя одноранговую аутентификацию, когда для доступа к базам данных используется имя пользователя в операционной системе. Одноранговая аутентификация поддерживается только для локальных соединений – через UNIX-сокет – и только в Linux.

Аутентификация клиента управляется конфигурационным файлом `pg_hba.conf`, где `pg` – аббревиатура PostgreSQL, `hba` означает **host-based authentication** (аутентификация по узлам). Чтобы найти в нем строки, относящиеся к одноранговой аутентификации, выполните команду:

```
grep -v '^#' /etc/PostgreSQL/10/main/pg_hba.conf|grep 'peer'
local all postgres peer
local replication all peer
```

Первая строка результата означает, что пользователь postgres может подключаться ко всем базам данных через Unix-сокет, применяя одноранговую аутентификацию.

Чтобы подключиться к серверу базы данных от имени пользователя `postgres`, мы должны сначала войти в операционную систему как `postgres`, а затем вызвать `psql`. В Linux это делается так:

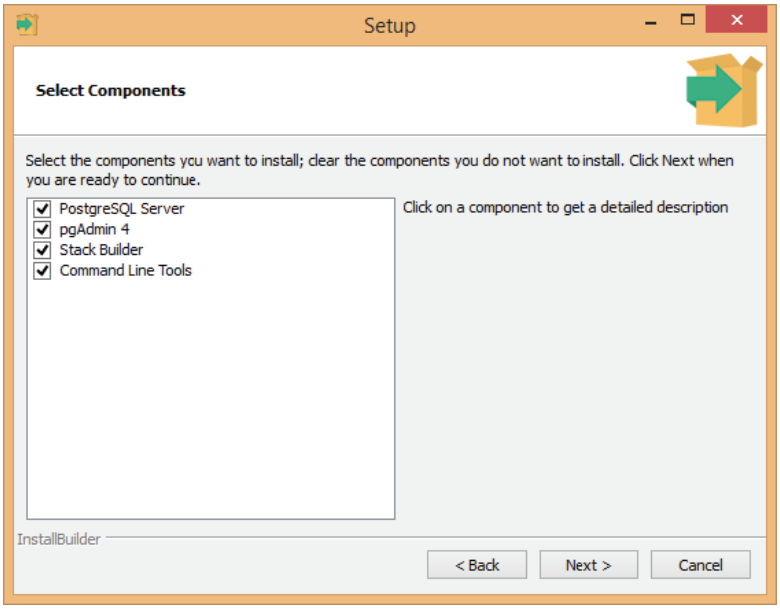
```
$sudo -u postgres psql
psql (10.0)
Type "help" for help.

postgres=# SELECT version();
version

------------------------------------------------------------------------
-----------------------------------
 PostgreSQL 10.0 on x86_64-pc-linux-gnu, compiled by gcc (Ubuntu
5.4.0-6ubuntu1~16.04.4) 5.4.0 20160609, 64-bit
(1 row)
```

Здесь мы интерактивно выполнили команду `SELECT version();` и получили в ответ сведения о версии PostgreSQL. Как видим, установлена версия PostgreSQL 10.0.

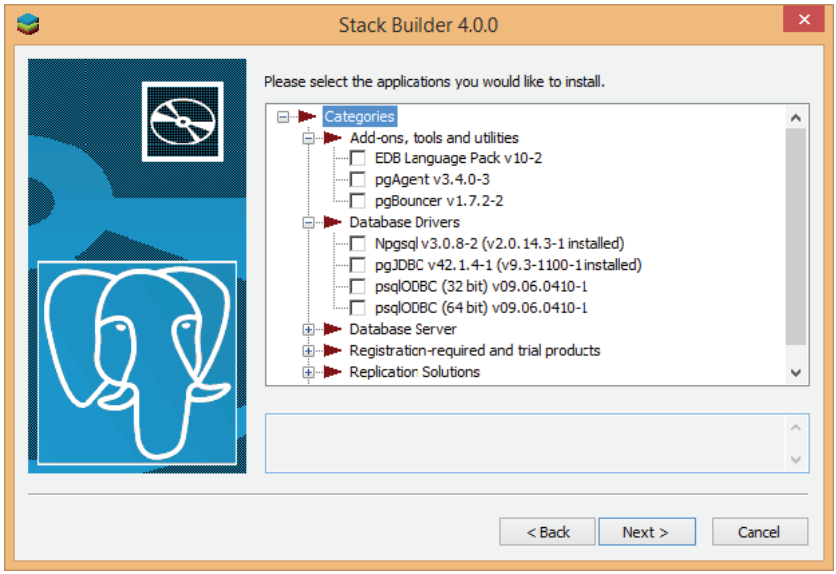
До выхода PostgreSQL 10 номер версии состоял из трех цифр. Новые основные версии выходят примерно раз в год и обычно сопровождаются сменой внутреннего формата данных. Это означает, что обратная совместимость хранимых данных между основными версиями не гарантируется. Номер следующей основной версии получается из номера предыдущей увеличением первой или второй цифры на единицу, например 9.5 и 9.6. При переходе к новой дополнительной версии изменяется третья цифра номера, например 9.6.1 и 9.6.2. В дополнительных версиях только исправляются ошибки. Начиная с PostgreSQL 10 схема нумерации версий изменилась: основные версии определяются только первым числом, т. е. следующая основная версия будет иметь номер 11. Номера дополнительных версий определяются вторым числом, например 10.0 и 10.1.

Установка PostgreSQL в Windows

Установить PostgreSQL в Windows для начинающих проще, чем в Linux. Можно скачать двоичный дистрибутив PostgreSQL с сайта EnterpriseDB (https://www.enterprisedb.com/downloads/postgres-PostgreSQL-downloads#windows). Мастер установки проведет по всем шагам процедуры, он попросит задать местоположение двоичных файлов, кластера баз данных, номер порта, пароль пользователя `postgres` и локаль.



Установщик запускает мастер построения комплекта, который позволяет выбрать необходимые драйверы PostgreSQL и многие другие утилиты:



В последних версиях Windows использовать клиент `psql` не очень удобно из-за отсутствия таких возможностей, как копирование, вставка и изменение размера в команде `CMD.exe`. Часто в Windows работают с клиентом `pgAdmin`. Альтернатива – взять другую командную оболочку, например `PowerShell` или какой-нибудь эмулятор Linux типа `Cygwin` и `MobaXterm`.

PostgreSQL в Windows устанавливается как служба. Чтобы проверить корректность установки, перейдите на вкладку Службы в Диспетчере задач.

Клиенты PostgreSQL

Помимо PgAdmin III и `psql`, инструменты для PostgreSQL выпускают многочисленные компании. Эти инструменты решают разные задачи: администрирование базы данных, моделирование, разработка, генерация отчетов, ETL и обратное конструирование. В состав самой PostgreSQL, помимо `psql`, входит еще несколько клиентских средств:

- **обертки** стандартных SQL-команд типа `CREATE USER`. Они упрощают автоматизацию часто решаемых задач;
- **резервное копирование и репликация** – PostgreSQL поддерживает физическое и логическое резервное копирование. Физическая резервная копия – это мгновенный снимок файлов базы данных. Физическое резервное копирование можно объединить с трансляцией WAL-файлов, реализовав тем самым потоковую репликацию или горячий резерв.

Логическая резервная копия – это выгрузка базы данных в виде SQL-команд. В отличие от физического копирования, таким способом можно выгрузить и восстановить одну базу данных, а не все сразу;

- **утилиты** – в комплект поставки PostgreSQL входит несколько утилит, облегчающих извлечение информации и диагностику проблем.

Сообщество PostgreSQL стремится к унификации интерфейса клиентов, поскольку так их проще изучать и использовать. Например, параметры соединения во всех клиентских утилитах задаются одинаково. Ниже перечислены параметры соединения для `psql` и всех остальных клиентов PostgreSQL:

- `-d`: имя базы данных;
- `-h`: имя или IP-адрес узла;
- `-u`: им пользователя;
- `-p`: номер порта.

Кроме того, большинство клиентов знает о переменных окружения, поддерживаемых библиотекой `libpq`, в т. ч. `PGHOST`, `PGDATABASE` и `PGUSER`. Эти переменные можно использовать для задания параметров соединения по умолчанию.

Клиент psql

Клиент `psql` поддерживается сообществом и входит в двоичный дистрибутив PostgreSQL. У него много впечатляющих возможностей, в том числе:

- **psql допускает конфигурирование**. Это и поведение на уровне сеанса, например `commit` и `rollback`, и приглашение, и файлы истории, и даже сокращения команд SQL;
- **интеграция с редактором и программой листания.** Результат запроса можно перенаправить в программу листания, например less или more. Сам `psql` не содержит встроенного редактора, но может работать с внешними. Ниже показано, как использовать редактор `nano` для редактирования функции:

```
postgres=# \setenv PSQL_EDITOR /bin/nano
postgres=# \ef
```

- **автозавершение и подсказка по синтаксису SQL.** `psql` поддерживает автозавершение имен объектов базы данных. Можно также задавать формат вывода результатов запроса, например HTML или latex. Например, при использовании метакоманды `\ef` генерируется такой шаблон:

```
CREATE FUNCTION ( )
    RETURNS
    LANGUAGE
    -- common options: IMMUTABLE STABLE STRICT SECURITY DEFINER
AS $function$

$function$
```

- клиент `psql` очень удобен в скриптах оболочки, для выборки данных и изучения PostgreSQL. Ниже перечислено несколько особенно часто используемых метакоманд `psql`:

- \d+ [pattern]: выводит всю информацию об отношении. В PostgreSQL термином «отношение» обозначаются таблица, представление, последовательность или индекс;
- \df+ [pattern]: описывает функцию;
- \z [pattern]: выводит права доступа к отношению.

Для программирования на языке оболочки в psql есть ряд удобных параметров:

- -A: не выравнивать вывод, по умолчанию вывод выровнен;
- -q (quiet): не выводить приветственное и информационные сообщения;
- -t: выводить только сами кортежи, без заголовков;
- -X: игнорировать конфигурационный файл ~/.psqlrc;
- -o: выводить результат запроса в указанный файл;
- -F: задает разделитель полей. Полезен для генерации CSV-файла, который затем можно импортировать в Excel;
- PGOPTIONS: позволяет задать параметры, передаваемые серверу на этапе выполнения и определяющие его поведение, например просмотр одних лишь индексов или тайм-аут команды.

Ниже показан bash-скрипт, возвращающий количество открытых подключений:

```
#!/bin/bash
connection_number=`PGOPTIONS='--statement_timeout=0' psql -AqXt -c"SELECT
count(*) FROM pg_stat_activity"`
```

Результат команды psql -AqXt -d postgres -c "SELECT count(*) FROM pg_stat_activity" присваивается переменной bash. Рассмотренные выше параметры -AqXt означают, что результат следует выводить без какой-либо дополнительной информации в виде:

```
$psql -AqXt -c "SELECT count(*) FROM pg_stat_activity"
 1
```

Дополнительные настройки psql

Клиент psql допускает персонализацию. В файле .psqlrc можно сохранить предпочтения пользователя, определяющие следующие аспекты:

- внешний вид;
- поведение;
- сокращения команд.

Можно изменить приглашение psql, показав информацию из строки соединения, в т. ч. имя сервера, имя базы данных, имя пользователя и номер порта. Для задания пользовательских предпочтений служат переменные psql PROMPT1, PROMPT2 и PROMPT3. Первая выводится как приглашение к вводу новой команды, вторая – когда требуется ввести продолжение команды. В примере ниже демонстрируется модификация приглашения; по умолчанию при подключении к базе показывается только имя этой базы.

Для присваивания значения переменной psql служит метакоманда \set. В данном случае переменной PROMPT1 присвоено значение (%n@%M:%>) [%/]%R%#%x >. Знак % – подстановочный маркер. Ниже показаны результаты подстановки:

```
postgres=# \set PROMPT1 '(%n@%M:%>) [%/]%R%#%x > '
(postgres@[local]:5432) [postgres]=# > BEGIN;
BEGIN
(postgres@[local]:5432) [postgres]=#* > SELECT 1;
 ?column?
----------
        1
(1 row)

 (postgres@[local]:5432) [postgres]=#* > SELECT 1/0;
ERROR: division by zero
 (postgres@[local]:5432) [postgres]=#! > ROLLBACK;
ROLLBACK
 (postgres@[local]:5432) [postgres]=# > SELECT
postgres-# 1;
```

В этом примере использованы следующие спецификаторы:

- %M: полное имя узла. В примере отображается [local], потому что мы подключались через Linux-сокет;
- %>: номер порта PostgreSQL;
- %n: имя текущего пользователя базы данных;
- %/: имя текущей базы данных;
- %R: обычно заменяется знаком =, но если по какой-то причине сеанс прерван, то подставляется знак !;
- %#: позволяет отличить суперпользователя от обычного. Знак # обозначает суперпользователя, знак > – обычного;
- %x: состояние транзакции. Знак * означает, что команда находится внутри блока транзакции, а знак ! – что в транзакции произошла ошибка.

Обратите внимание, что команда SELECT 1 записана в двух строчках и в начале второй напечатано приглашение PROMPT2. И еще обратите внимание на знак *, обозначающий блок транзакции.

В psql можно создавать сокращение частых запросов, например для команды, которая выводит информацию о выполняемых в данный момент командах. Для этого тоже используется метакоманда \set, а знак : означает подстановку.

```
postgres=# \set activity 'SELECT pid, query, backend_type, state FROM
pg_stat_activity';
postgres=# :activity;
 pid   | query | backend_type | state
-------+-------------------------------------------------------------------+---
------------------+--------
  3814 | | background worker |
  3812 | | autovacuum launcher |
 22827 | SELECT pid, query, backend_type, state FROM pg_stat_activity; |
```

```
client backend | active
  3810 | | background writer |
  3809 | | checkpointer |
  3811 | | WALwriter |
(6 rows)
```

psql позволяет настроить поведение транзакций. Для этого предусмотрены три переменные: ON_ERROR_ROLLBACK , ON_ERROR_STOP и AUTOCOMMIT.

- ON_ERROR_STOP: по умолчанию psql продолжает выполнять команды даже после возникновения ошибки. Иногда это полезно, например при выгрузке и восстановлении всей базы данных, когда некоторые ошибки, скажем, отсутствие расширений, можно игнорировать. Но при разработке приложений игнорировать ошибки нельзя, поэтому рекомендуется присвоить этой переменной значение on. Эта переменная полезна в сочетании с параметрами -f, \i, \ir.

```
$ echo -e 'SELECT 1/0;\nSELECT 1;'>/tmp/test_rollback.sql
$ psql
psql (10.0)
Type "help" for help.

postgres=# \i /tmp/test_rollback.sql
psql:/tmp/test_rollback.sql:1: ERROR: division by zero
 ?column?
----------
        1
(1 row)

postgres=# \set ON_ERROR_STOP on
postgres=# \i /tmp/test_rollback.sql
psql:/tmp/test_rollback.sql:1: ERROR: division by zero
```

- ON_ERROR_ROLLBACK: если в блоке транзакции возникает ошибка, то в зависимости от значения этой переменной может происходить одно из трех. Если переменная равна off, то вся транзакция откатывается – это поведение по умолчанию. Если переменная равна on, то ошибка игнорируется и транзакция продолжается. Наконец, если она равна interactive, то ошибки игнорируются в интерактивных сеансах, но не тогда, когда скрипт читается из файла.
- AUTOCOMMIT: SQL-команды, не включенные явно в транзакционный блок, фиксируются автоматически. Чтобы снизить вероятность ошибки человека, рекомендуется не включать этот режим.

Выключение режима AUTOCOMMIT полезно, потому что оставляет возможность выполнить rollback и отменить нежелательные изменения. При развертывании или модификации базы данных на живой системе рекомендуется производить все изменения в транзакционном блоке и дополнительно подготовить скрипт отката.

Наконец, метакоманда \timing показывает время выполнения и часто используется для грубой оценки производительности. А метакоманда \pset управляет форматированием вывода.

Утилиты PostgreSQL

В комплект поставки PostgreSQL входит несколько утилит, являющихся обертками SQL-команд. Они позволяют создавать и удалять базы данных, пользователей и языки. Так, утилиты `dropdb` и `createdb` обертывают команды `DROP DATABASE [ IF EXISTS ]` и `CREATE DATABASE` соответственно.

Кроме того, PostgreSQL предоставляет средства для обслуживания системных объектов, в т. ч. `clusterdb` и `reindexdb`. Утилита `clusterdb` обертывает команду `CLUSTER`, которая физически переупорядочивает таблицу, сообразуясь с информацией в некотором индексе. Так можно повысить производительность чтения благодаря пространственной локальности ссылок. Кластеризация таблицы ускоряет выборку данных из соседних блоков и потому уменьшает накладные расходы на доступ к диску.

Утилита `reindexdb` обертывает SQL-команду `reindex`. Для перестраивания индекса есть несколько причин, в т. ч. повреждение (на практике случается редко) и разрастание вследствие многочисленных операций вставки и удаления.

Помимо всего вышеперечисленного, PostgreSQL предоставляет следующие средства:

- **физическое резервное копирование.** Используется для копирования файлов базы данных путем создания мгновенного снимка диска. Это очень быстрый способ, но созданную таким образом резервную копию можно восстановить только в совместимую версию PostgreSQL. Для этой цели служит программа `pg_basebackup`. Она часто применяется для первоначальной настройки потоковой репликации, поскольку ведомый узел должен быть копией ведущего;
- **логическое резервное копирование.** Используется для копирования объектов базы данных в виде SQL-команд: `CREATE TABLE`, `CREATE VIEW`, `COPY` и т. п. Такую копию можно восстановить в другую версию PostgreSQL, но это медленный способ. Для выгрузки одной базы или всего кластера служат утилиты `pg_dump` и `pg_dumpall` соответственно. Утилиту `pg_dump` можно также использовать для выгрузки одного или нескольких отношений, возможно, со схемой. У нее есть много параметров, например выгружать только схему или только данные. Утилита `pg_dumpall` вызывает `pg_dump`, чтобы выгрузить все базы данных в кластере. Наконец, `pg_restore` восстанавливает данные из файлов, созданных с помощью `pg_dump` или `pg_dumpall`.

`pg_dump` не включает в файл команду `CREATE DATABASE`. Это значит, что можно выгрузить базу `customer` и загрузить данные в базу `client`. Но если у вас были особые привилегии при работе со старой базой, например `CONNECT`, то их нужно будет перенести и на новую.

Резюме

PostgreSQL – объектно-ориентированная реляционная система управления базами данных с открытым исходным кодом. Она совместима со стандартом ANSI SQL и поддерживает много дополнительных возможностей. Лозунг Post-

greSQL – *самая передовая база данных с открытым исходным кодом* – отражает ее техническое совершенство. Это результат многих лет исследований, проводимых совместно академическим сообществом и промышленностью. Стартапы часто выбирают PostgreSQL за либеральные условия лицензирования и выгодную бизнес-модель.

Многие разработчики также отдают предпочтение PostgreSQL, ценя ее возможности и преимущества. PostgreSQL пригодна как для OLTP, так и для OLAP-приложений. Она полностью поддерживает свойства ACID, поэтому для работы в OLTP-приложениях ничего больше не требуется. Что касается OLAP-приложений, PostgreSQL поддерживает оконные функции, адаптеры внешних данных и наследование таблиц. Существуют также многочисленные сторонние расширения. PostgreSQL лежит в основе нескольких коммерческих СУБД. Кроме того, существует несколько ответвлений с открытым исходным кодом, в которые добавлены новые возможности и технологии, например MPP и MapReduce.

PostgreSQL располагает хорошо организованным и активным сообществом пользователей, разработчиков, компаний и ассоциаций. Сообщество обогащает PostgreSQL ежедневно; многие компании вносят свой вклад в PostgreSQL, публикуя статьи, делясь передовым опытом, направляя разработчикам запросы на новую функциональность, предлагая свои наработки и разрабатывая новые инструменты и приложения.

Первый опыт работы с PostgreSQL протекает легко и безболезненно, для установки достаточно нескольких минут. В комплект поставки PostgreSQL входит много клиентских инструментов для взаимодействия с пользователем. Утилита `psql` предлагает удобные метакоманды `\h`, `\?`, `\timing</kbd>`, `\pset` и механизм автозавершения, что помогает работать быстрее.

В следующей главе мы познакомимся с основными строительными блоками PostgreSQL. Мы также создадим первую базу данных и поработаем с такими командами языка определения данных, как `CREATE TABLE` и `CREATE VIEW`. Там же будут даны рекомендации о стиле кодирования на SQL и приведен общий обзор взаимодействия компонентов.

Глава 3

Основные строительные блоки PostgreSQL

В этой главе мы создадим базу данных и познакомимся с основными строительными блоками PostgreSQL. Концептуальная модель сайта торговли автомобилями, представленная в главе 1, будет преобразована в физическую модель. Кроме того, мы кратко обсудим некоторые приемы моделирования данных, в частности суррогатные ключи, и дадим ряд рекомендаций по кодированию.

Также мы рассмотрим иерархию объектов базы данных в PostgreSQL. Это поможет нам лучше понять, как конфигурировать и оптимизировать кластер баз данных. Мы подробно расскажем о шаблонных и пользовательских базах данных, полях, табличных пространствах, схемах, конфигурационных параметрах и таблицах. Вот перечень тем этой главы:

- кодирование базы данных;
- иерархия объектов в PostgreSQL;
- компоненты базы данных PostgreSQL;
- база данных для сайта торговли автомобилями.

Кодирование базы данных

К кодированию базы данных применимы все принципы программной инженерии. Важно, чтобы код был чистым, а циклы разработки – быстрыми. В этом разделе мы рассмотрим соглашение об именовании, документирование и управление версиями. Если все это вам знакомо, то можете пропустить данный раздел.

Соглашение об именовании объектов базы данных

Соглашение об именовании говорит о том, как должны выбираться имена. В хороших именах прослеживается закономерность, помогающая разработчику предсказывать, как называется тот или иной объект. В организации должен быть принят стандарт именования. О том, как называть объекты базы данных, много спорят. Например, кто-то предпочитает суффиксы или префиксы, чтобы

по имени можно было сделать вывод о типе объекта. Так, для таблиц можно выбрать суффикс `tbl`, а для представлений – суффикс `vw`.

При выборе имен для объектов базы данных следует отдавать предпочтение описательным именам, по возможности избегая акронимов и аббревиатур. Кроме того, лучше, чтобы имя было в единственном числе, поскольку таблица часто отображается на сущность в языке программирования высокого уровня, так что, выбирая имя в единственном числе, мы унифицируем имена на уровне базы данных и на уровне бизнес-логики. К тому же единственное число упрощает задание кардинальности в связях между таблицами.

В мире баз данных в составных именах для разделения слов обычно употребляется подчеркивание, а не верблюжья нотация (CamelCase), что согласуется со стандартом ANSI SQL в части закавычивания идентификаторов и чувствительности к регистру – в ANSI SQL не заключенные в кавычки идентификаторы нечувствительны к регистру.

Вообще говоря, разработчик сам решает, как выбирать имена применительно к ситуации. Но если вы работаете с уже существующим проектом, не придумывайте новых соглашений, не обсудив их с членами команды. В этой книге мы будем пользоваться следующими соглашениями:

- имена таблиц и представлений не имеют суффиксов;
- имена объектов базы данных уникальны в пределах одной базы;
- идентификаторы записываются в единственном числе, это относится к именам таблиц, представлений и столбцов;
- слова в составных именах разделяются знаками подчеркивания;
- имя первичного ключа образуется из имени таблицы и суффикса `id`;
- внешний ключ называется так же, как соответствующий первичный ключ в связанной таблице;
- при выборе имен первичных ключей, внешних ключей и последовательностей не используются ключевые слова SQL. Список ключевых слов можно найти в документации PostgreSQL на странице https://www.postgresql.org/docs/current/static/sql-keywords-appendix.html.

Идентификаторы в PostgreSQL

Длина имени объекта в PostgreSQL не должна превышать 63 символов; PostgreSQL следует правилам ANSI SQL, касающимся чувствительности к регистру. Если вам больше нравится верблюжья нотация, заключайте имена объектов в двойные кавычки. На имена идентификаторов в PostgreSQL налагаются следующие ограничения:

1) имя идентификатора должно начинаться знаком подчеркивания или буквой. Допускаются буквы латиницы и других алфавитов;
2) имя идентификатора может содержать буквы, цифры, знак подчеркивания и знак доллара. В целях совместимости знак доллара использовать не рекомендуется;
3) минимальная длина идентификатора – 1 символ, максимальная – 63 символа.

Кроме того, не рекомендуется использовать ключевые слова SQL в качестве имен таблиц.

Документация

Документация важна как разработчикам, так и владельцам предприятий, чтобы понимать картину в целом. Документировать следует схему, объекты базы данных и код. Полезны также диаграммы сущность-связь и диаграммы классов. Программ для создания диаграмм – UML и сущность-связь – полно. Можно, например, использовать такие графические редакторы, как yEd, или онлайновый инструмент draw.io. Существует также много коммерческих программ UML-моделирования, поддерживающих реверс-инжиниринг.

Документация кода помогает разобраться в сложных SQL-командах. В PostgreSQL однострочные комментарии начинаются двумя дефисами (--), а многострочные заключаются между /* и */. Однострочный комментарий начинается с -- и продолжается до конца строки, поэтому его можно использовать в одной строке с кодом. Наконец, описание объекта базы данных можно сохранить вместе с ним в базе с помощью команды `COMMENT ON`.

Система управления версиями

Код имеет смысл хранить в системе управления версиями типа Git или SVN. SQL-код лучше оформлять в виде скрипта и исполнять в контексте одной транзакции. При таком подходе будет проще восстановить данные в случае ошибки.

У объектов базы данных разные свойства: одни являются частью схемы, другие управляют доступом. Ниже мы предлагаем, как лучше организовать код базы данных, чтобы обеспечить **разделение обязанностей**.

Для каждой базы данных в кластере PostgreSQL заведите DDL-скрипт для объектов, являющихся частью схемы, и DML-скрипт, который заполняет таблицы статическими данными. Состояние объекта схемы определяется его структурой и хранящимися в нем данными; чтобы заново создать объект, его нужно сначала удалить. Впрочем, структура объекта схемы изменяется нечасто. Однако если все же требуется рефакторинг таких объектов, как таблицы, то возникает необходимость в переносе данных. Короче говоря, для изменения схемы нужно предварительное планирование.

DDL-скрипты для создания объектов, не являющихся частью физической схемы, например представлений и функций, храните отдельно. При этом определения представлений и функций храните вместе, чтобы было проще производить рефакторинг.

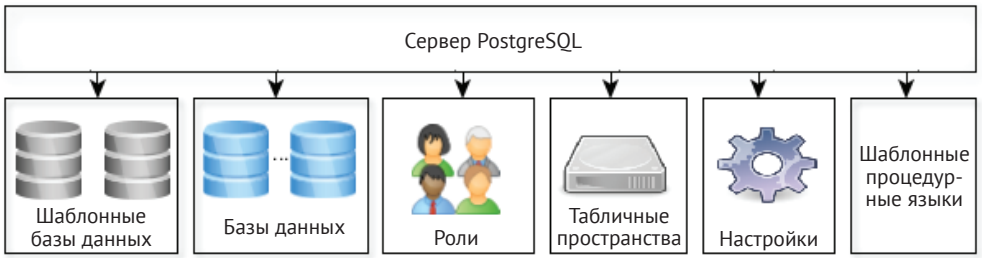
DCL-скрипт стоит выделить. Это позволит разделить безопасность и функциональные требования к базе данных, так что разработчики и администраторы смогут работать, не мешая друг другу.

Средство миграции базы данных

Существует возможность интегрировать средство миграции базы данных, например Flywaydb (https://flywaydb.org), с Git. Это позволяет настроить непрерывную интеграцию, а кроме того, администратор будет видеть, что происходит с базой данных.

Иерархия объектов в PostgreSQL

Для понимания отношений и взаимодействий между объектами нужно разобраться, как организованы логические объекты в базе данных PostgreSQL. Базы данных, роли, табличные пространства, настройки и процедурные языки находятся на одном уровне иерархии, как показано на рисунке ниже.



Шаблонные базы данных

По умолчанию новая база данных создается как клон шаблонной базы данных `template1`. В шаблонной базе находятся таблицы, представления и функции, необходимые для моделирования отношений между пользовательскими объектами базы данных. Все они являются частью системной схемы `pg_catalog`.

По своей сути схема близка к понятию пространства имен в объектно-ориентированных языках. Она используется для организации объектов базы данных, функциональности, прав доступа, а также для устранения конфликтов имен.

В PostgreSQL есть две шаблонные базы данных:

- `template1`: база данных, клонируемая по умолчанию. Ее можно модифицировать, тогда изменения будут отражены во всех вновь создаваемых базах. Например, если вы собираетесь использовать во всех базах некоторое расширение, то установите его в базу `template1`. Разумеется, это расширение не появится в уже существующих базах, но будет включено во все базы, созданные впоследствии.
- `template0`: дополнительная база данных, у которой несколько задач:
  - если `template1` повреждена, то `template0` можно использовать для ее ремонта;

– она также полезна при восстановлении из резервной копии. Выгруженная база данных содержит, в частности, все расширения. Если расширение установлено в `template1`, то при попытке восстановить копию в базу, созданную по этому шаблону, возникнет конфликт. А если перед восстановлением создать базу по шаблону `template0`, то конфликтов не будет. Кроме того, `template0`, в отличие от `template1`, не содержит данных, зависящих от кодировки или локали.

Можно также создать базу, указав в качестве шаблона пользовательскую базу данных. Это удобно для тестирования, рефакторинга, планирования развертывания и других целей.

Пользовательские базы данных

В одном кластере баз данных можно создать сколько угодно баз. При подключении к серверу PostgreSQL открывается только одна база, указанная в строке соединения. Поэтому обратиться к данным из другой базы нельзя, если только не используется адаптер внешних данных или расширение `dblink`.

У каждой базы данных в кластере имеются владелец и ассоциированные с ней разрешения для контроля действий, разрешенных определенной роли. Разрешения для доступа к объектам PostgreSQL – базам данных, таблицам, представлениям и последовательностям – в `psql` представляются следующим образом:

```
<user>=<privileges>/granted by
```

Если часть `user` отсутствует, значит, разрешения предоставлены специальной роли `PUBLIC`.

В `psql` для перечисления всех баз данных в кластере вместе с атрибутами служит метакоманда `\l`:

```
postgres=# \l
                                  List of databases
  Name      | Owner          | Encoding | Collate     | Ctype       | Access privileges
------------+----------------+----------+-------------+-------------+----------------------
 car_portal | car_portal_app | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
 postgres   | postgres       | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
 template0  | postgres       | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/postgres +
               | | | | | postgres=CTc/postgres
 template1  | postgres       | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/postgres +
               | | | | | postgres=CTc/postgres
```

Для базы данных определены следующие разрешения:

- **создание** (`-C`): позволяет создавать новые схемы в базе данных;
- **подключение** (`-c`): проверяется при попытке подключения к базе;
- **временные** (`-T`): позволяют создавать временные таблицы. Они похожи на обычные, но уничтожаются после завершения сеанса работы с сервером.

В предыдущем примере базе данных не были назначены разрешения явно. Отметим, что роли `public` по умолчанию разрешено подключаться к базе данных `template1`.

PostgreSQL поддерживает широкий набор кодировок текста – как однобайтовых (SQL_ASCII), так и многобайтовых (UTF-8). Полный перечень всех кодировок имеется на странице http://www.postgresql.org/docs/current/static/multibyte.html.

Имеется также ряд других атрибутов для разных целей.

- **Обслуживание:** атрибут `datfrozenxid` позволяет узнать, пора ли уже очищать базу данных.
- **Управление хранением:** атрибут `dattablespace` определяет табличное пространство базы данных.
- **Параллелизм:** атрибут `datconnlimit` задает число допустимых подключений (-1 означает, что ограничения нет).
- **Защита:** атрибут `datallowconn` запрещает подключение к базе данных. В основном он используется, чтобы защитить базу `template0` от изменения.

Метакоманда `psql \c` открывает новое соединение с базой данных и закрывает текущее. Она принимает строку соединения, которая может содержать имя и пароль пользователя.

```
postgres=# \c template0
FATAL: database "template0" is not currently accepting connections Previous
connection kept
```

В каталоге `pg_catalog` находятся обычные таблицы, для работы с ними можно использовать команды `SELECT`, `UPDATE` и `DELETE`. Однако делать это не рекомендуется, а если все же делаете, то будьте предельно внимательны и осторожны.

Каталожные таблицы очень полезны для автоматизации некоторых задач; каталогу `pg_catalog` посвящена глава 12. В следующем примере показано, как изменить максимальное количество подключений к базе данных с -1 на 1 командой `ALTER DATABASE`.

```
postgres=# SELECT datconnlimit FROM pg_database WHERE datname='postgres';
 datconnlimit
--------------
            -1
(1 row)

postgres=# ALTER DATABASE postgres CONNECTION LIMIT 1;
ALTER DATABASE
postgres=# SELECT datconnlimit FROM pg_database WHERE datname='postgres';
 datconnlimit
 --------------
 1
 (1 row)
```

Роли

Роли принадлежат кластеру баз данных, а не конкретной базе. В качестве роли может выступать пользователь или группа. Концепция роли заменяет концепции групп и пользователей в старых версиях PostgreSQL. Ради совместимости

в версии PostgreSQL 8.1 и более поздних все еще поддерживаются команды CREATE USER и CREATE GROUP.

У ролей имеются следующие атрибуты.

- **Superuser:** эта роль обходит все проверки, кроме атрибута login.
- **Login:** роль с атрибутом login позволяет подключаться к базе данных.
- **Createdb:** роль с таким атрибутом позволяет создавать базы данных.
- **Createrole:** роль с этим атрибутом дает возможность создавать, изменять и удалять другие роли.
- **Replication:** роль с данным атрибутом позволяет выполнять потоковую репликацию.
- **Password:** пароль роли используется при аутентификации методом md5. Его можно зашифровать и задать для него срок действия. Отметим, что этот пароль отличается от пароля в операционной системе.
- **Connection limit:** задает максимальное число одновременных подключений. Каждое подключение потребляет аппаратные ресурсы, поэтому рекомендуется использовать средства организации пула подключений, например **pgpool-II** или **PgBouncer**, либо такие API, как Apache **DBCP** или **c3p0**.
- **Inherit:** роль с таким атрибутом наследует разрешения от ролей, членом которых является. Атрибут Inherit подразумевается по умолчанию.
- **Bypassrls:** роль с этим атрибутом обходит механизм безопасности на уровне строк (**RLS**).

Роль postgres с атрибутом superuser создается в процессе установки PostgreSQL. Команда CREATE USER эквивалентна CREATE ROLE с атрибутом LOGIN, а CREATE GROUP эквивалентна CREATE ROLE с атрибутом NOLOGIN.

Роль может быть членом другой роли – это упрощает управление разрешениями в базе данных. Например, мы можем создать роль без атрибута login (иными словами, группу) и дать ей разрешения на доступ к объектам базы данных. Если новой роли понадобится доступ к тем же объектам с такими же разрешениями, как у группы, то ее можно будет сделать членом существующей группы. Для этой цели предназначены команды GRANT и REVOKE (см. главу 11).

Роли в кластере необязательно должны иметь разрешения для доступа к каждой базе в этом кластере.

Табличное пространство

Табличное пространство – это область для хранения базы данных или ее объектов. С помощью табличных пространств администратор решает следующие задачи:

- **обслуживание:** если в разделе жесткого диска, в котором создан кластер баз данных, кончается место и его нельзя расширить, то можно создать табличное пространство в другом разделе и переместить туда данные;

- **оптимизация:** данные, к которым часто производится доступ, можно разместить на быстром носителе, например SSD-диске. А не столь критичные для производительности таблицы оставить на медленном диске.

Табличное пространство создается командой `CREATE TABLESPACE`.

Шаблонные процедурные языки

Шаблонные процедурные языки предназначены для удобства регистрации языков программирования. Существует два способа это сделать. Первый – указать только название языка. В таком случае PostgreSQL заглянет в шаблон языка и определит параметры. Второй – указать имя и параметры. Для установки языка служит команда `CREATE LANGUAGE`.

Начиная с версии PostgreSQL 9.1 для установки языка программирования можно использовать команду `CREATE EXTENSION`. Сведения о шаблонных процедурных языках хранятся в таблице `pg_pltemplate`. Ее можно было бы убрать и хранить информацию о языке в его установочном скрипте.

Параметры

Параметры PostgreSQL управляют различными аспектами работы сервера, в т. ч. репликацией, журналами предзаписи, потреблением ресурсов, планированием запросов, протоколированием, аутентификацией, сбором статистики, сборкой мусора, клиентскими подключениями, блокировками, обработкой ошибок и отладкой.

Следующая SQL-команда показывает, сколько всего есть параметров. Отметим, что результат зависит от конкретной установки и конфигурации:

```
postgres=# SELECT count(*) FROM pg_settings;
 count
-------
   269
(1 row)
```

Настройки могут быть разных типов:

- **булевы:** 0, 1, true, false, on, off в любом регистре. В эту категорию попадает параметр `ENABLE_SEQSCAN`;
- **целые:** целочисленные значения имеют параметры, относящиеся к памяти и времени; для каждого из них существует подразумеваемая единица измерения, например минута или секунда. Во избежание путаницы PostgreSQL позволяет задавать единицы измерения явно. Например, параметр `shared_buffers` мог бы иметь значение 128 MB;
- **перечисления:** предопределенные значения, например `ERROR` или `WARNING`;
- **с плавающей точкой:** к таковым относится параметр `cpu_operator_cost`, предназначенный для оптимизации планов выполнения;
- **строки:** строка может задавать положение файла на диске.

Контекст параметра определяет, как его изменять и когда изменение вступает в силу. Существуют следующие контексты:

- **внутренний:** параметр невозможно изменить непосредственно. Требуется либо перекомпилировать исходный код, либо заново инициализировать кластер баз данных. Например, максимальная длина идентификатора в PostgreSQL составляет 63 символа;
- **серверный:** после изменения параметра необходимо перезагрузить сервер. Такие параметры обычно находятся в конфигурационном файле `postgresql.conf`;
- **Sighup:** перезапускать сервер не нужно. Чтобы параметр вступил в силу, нужно изменить файл `postgresql.conf`, а затем послать сигнал SIGHUP серверному процессу;
- **заднего плана:** перезапускать сервер не нужно. Можно установить для одного сеанса;
- **суперпользовательский:** такой параметр может изменять только суперпользователь. Задается в файле `postgresql.conf` или командой `SET`;
- **пользовательский:** аналогичен суперпользовательскому, обычно задается на уровне сеанса.

Для изменения и просмотра значения параметра предназначены команды `SET` и `SHOW`. Они используются в контексте пользователя или суперпользователя. Обычно задание значения в файле `postgresql.conf` имеет глобальный эффект.

Параметры могут иметь также локальное действие и применяться в других контекстах, например сеанса или таблицы. Допустим, к примеру, что мы хотим, чтобы некоторые клиенты могли выполнять только операции чтения; это полезно при настройке таких инструментов, как Confluence (Atlassian). Достичь нужного эффекта можно, установив параметр `default_transaction_read_only`:

```
postgres=# SET default_transaction_read_only to on;
SET
postgres=# CREATE TABLE test_readonly AS SELECT 1;
ERROR: cannot execute CREATE TABLE AS in a read-only transaction
```

Здесь создание таблицы завершилось ошибкой в уже открытом сеансе. Но если открыть новый сеанс и попытаться выполнить в нем команду `CREATE TABLE`, то все получится, потому что по умолчанию параметр `default_transaction_read_only` равен `off`. С другой стороны, как было сказано выше, задание параметра `default_transaction_read_only` в файле `postgresql.conf` будет иметь глобальный эффект.

PostgreSQL предоставляет также функцию `pg_reload_conf()`, эквивалентную отправке сигнала `SIGHUP` процессу postgres.

Вообще говоря, функция `pg_reload_conf()` или перезагрузка конфигурационных параметров с помощью скрипта init безопаснее отправки сигнала `SIGHUP`, т. к. у человека меньше возможностей совершить ошибку.

Чтобы перевести базу данных в режим чтения в Debian Linux, можно выполнить следующие действия:

1) открыть файл `postgresql.conf` и изменить значение параметра `default_transaction_read_only`. В Ubuntu это делается такими командами:

```
$sudo su postgres
$CONF=/etc/postgresql/10/main/postgresql.conf
$sed -i "s/#default_transaction_read_only =
off/default_transaction_read_only = on/" $CONF
```

2) перезагрузить конфигурацию, выполнив функцию `pg_reload_conf()`:

```
$psql -U postgres -c "SELECT pg_reload_conf()"
```

Следует тщательно планировать изменения параметров, после которых придется перезагружать сервер. Если изменение некритическое, то можно модифицировать файл `postgresql.conf` и оставить до следующей перезагрузки сервера в связи с обновлением безопасности. Если же изменение срочное, то нужно выработать определенную процедуру, например запланировать выключение и проинформировать о нем пользователей. Для разработчиков наибольший интерес представляют параметры из двух категорий:

- **параметры подключений по умолчанию:** управляют поведением команд, локалью и форматированием;
- **планирование запросов:** управляют конфигурацией планировщика и подсказывают разработчику, как можно было бы перезаписать SQL-запрос.

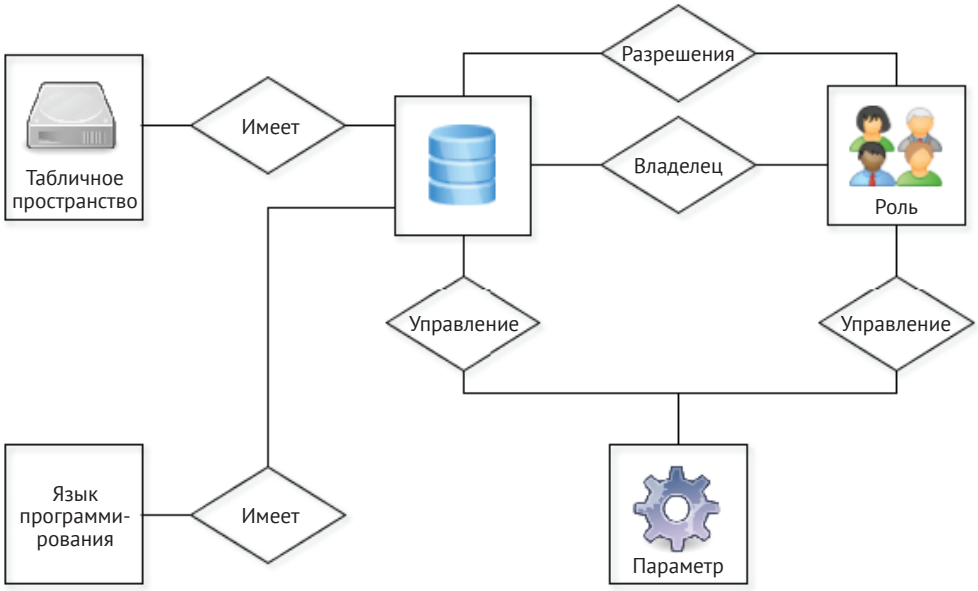
Взаимодействия с объектами PostgreSQL верхнего уровня

Подведем итоги. Сервер PostgreSQL может содержать несколько баз данных, языков программирования, ролей и табличных пространств. У каждой базы данных есть владелец и табличное пространство по умолчанию. Роли может быть дано разрешение на доступ к нескольким базам данным, роль также может быть владельцем баз данных. Параметры PostgreSQL можно использовать для управления поведением сервера на нескольких уровнях, например на уровне базы данных и отдельного сеанса. Наконец, в базу данных можно установить несколько языков программирования.

Для создания базы данных необходимо задать владельца и кодировку. Если кодировка `template1` не совпадает с желаемой, то следует явно использовать шаблон `template0`.

Предположим, что владельцем базы на сайте торговли автомобилями является роль `car_portal_role` и используется кодировка `UTF-8`. Чтобы создать такую базу, нужно выполнить команды:

```
CREATE ROLE car_portal_app LOGIN;
CREATE DATABASE car_portal ENCODING 'UTF-8' LC_COLLATE 'en_US.UTF-8'
LC_CTYPE 'en_US.UTF-8' TEMPLATE template0 OWNER car_portal_app;
```

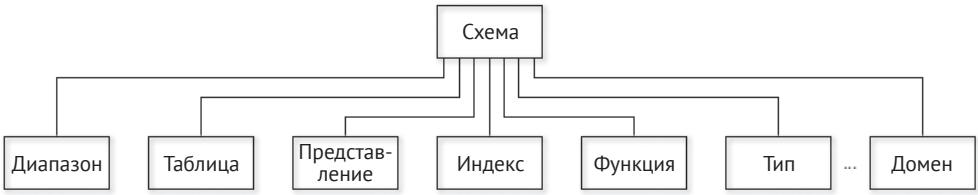


Компоненты базы данных PostgreSQL

Базу данных PostgreSQL можно рассматривать как контейнер схем, в базе должна быть, по меньшей мере, одна схема. Схема базы данных служит для организации объектов базы данных – по аналогии с пространствами имен в языках программирования.

Схема

Схема содержит все именованные объекты базы данных: таблицы, представления, функции, агрегаты, индексы, последовательности, триггеры, типы данных, домены и диапазоны. Одно и то же имя объекта может встречаться в разных схемах.



В шаблонных базах данных имеется схема *public*, а это означает, что во всех вновь созданных базах будет схема с таким именем. По умолчанию любой пользователь неявно имеет доступ к этой схеме, и это тоже наследуется от шаблонных баз данных. Наличие такого доступа позволяет считать, что никаких схем нет вообще. Это полезно в небольших компаниях, где не требуется сложная организация безопасности, а кроме того, позволяет постепенно переходить на PostgreSQL с СУБД, не имеющими схем.

В многопользовательской среде с несколькими базами данных не забудьте отозвать у всех пользователей разрешение создавать объекты в схеме public. Для этого нужно выполнить следующую команду во вновь созданной базе или в базе `template1`: `REVOKE CREATE ON SCHEMA public FROM PUBLIC;`.

Для обращения к объекту следует указать имя схемы и имя объекта, разделив их точкой. Например, чтобы выбрать все строки таблицы `pg_database` в схеме `pg_catalog`, нужно выполнить команду:

```
SELECT * FROM pg_catalog.pg_database;
-- Можно также использовать такую команду:
TABLE pg_catalog.pg_database;
```

Полные имена объектов писать долго, поэтому многие разработчики предпочитают использовать простые имена без указания схемы. В PostgreSQL имеется параметр `search_path`, аналогичный директиве `using` в языке C++. В пути поиска перечисляются схемы, в которых сервер ищет имена объектов. По умолчанию путь поиска такой: `$user, public`:

```
postgres=# SHOW search_path;
   search_path
-----------------
 "$user", public
(1 row)
```

Если существует схема с таким же именем, как у текущего пользователя, то при поиске объектов она будет просматриваться первой, и в ней же будут создаваться новые объекты. Если объект не найден в схемах, перечисленных в `search_path`, то возникает ошибка:

Применение схем

У схем есть несколько применений:

- **управление авторизацией** – в многопользовательской среде схемы можно использовать для группировки объектов по ролям;
- **организация объектов базы данных** – можно сгруппировать объекты в соответствии с бизнес-логикой. Например, выделить в отдельную группу исторические данные и данные аудита и завести для них специальную схему;
- **хранение стороннего SQL-кода** – расширения, входящие в состав сторонних пакетов, могут использоваться в нескольких приложениях.

Хранение их в отдельных схемах упрощает повторное использование и обновление.

Чтобы создать схему `car_portal_app`, принадлежащую роли `car_portal_app`, нужно выполнить команду:

```
CREATE SCHEMA car_portal_app AUTHORIZATION car_portal_app;
 -- Если имя схемы не указано, то оно совпадает с именем владельца
 CREATE SCHEMA AUTHORIZATION car_portal_app;
```

Для получения дополнительных сведений о команде `CREATE SCHEMA` воспользуйтесь метакомандой `psql \h`, которая отображает встроенную в `psql` справку, или загляните в страницу руководства по PostgreSQL http://www.postgresql.org/docs/current/static/sqlcreateschema.html.

Таблица

Команда `CREATE TABLE` выполняет разнообразные функции. Например, ее можно использовать для клонирования таблицы, что удобно при проведении рефакторинга базы данных, когда нужно создать скрипт для отката изменений. Она же позволяет материализовать результаты `SELECT` для повышения производительности или сохранения данных для последующего использования.

В PostgreSQL таблицы используются для моделирования представлений и последовательностей. Существует несколько типов таблиц:

- **постоянная** – возникает в момент создания и пропадает после удаления;
- **временная** – существует в течение сеанса. Часто такие таблицы используют в процедурных языках для моделирования бизнес-логики;
- **нежурналируемая** – операции с нежурналируемыми таблицами выполняются гораздо быстрее, чем с постоянными, т. к. данные не записываются в WAL-файлы. Но такие таблицы неустойчивы к сбоям. Кроме того, поскольку потоковая репликация основана на доставке журналов, нежурналируемую таблицу нельзя реплицировать на ведомый узел;
- **дочерняя** – такая таблица наследует одну или несколько таблиц. Часто наследование применяется совместно с исключением в силу ограничений, чтобы физически разделить данные на жестком диске и повысить производительность за счет выборки подмножества данных с определенным значением.

Синтаксис команды `CREATE TABLE` довольно обширный, полное описание можно найти на странице http://www.postgresql.org/docs/current/static/sql-createtable.html. Обычно указывается следующая информация:

- имя создаваемой таблицы;
- тип таблицы;
- параметры хранения, которые используются для управления выделением места и некоторых других административных задач;
- столбцы таблицы, включая тип данных, значения по умолчанию и ограничения;
- имя клонированной таблицы и параметры клонирования.

Встроенные типы данных

При проектировании таблицы базы данных следует тщательно продумывать типы столбцов. После передачи базы данных в эксплуатацию изменение типа данных может оказаться непростой операцией, особенно для высоконагруженных таблиц. Проблема возникает из-за блокировки таблицы, а иногда таблицу даже приходится перезаписывать. Принимая решение о типе данных, следует учитывать следующие факторы:

- **расширяемость:** можно ли увеличить или уменьшить максимальную длину столбца, не прибегая к полной перезаписи и полному просмотру таблицы?
- **размер типа данных:** перестраховка, например выбор тип `bigint` вместо `int` влечет повышенное расходование места на диске.

PostgreSQL предлагает обширный набор типов данных. Вот некоторые категории встроенных типов:

- числовые типы;
- символьные типы;
- дата и время.

Эти типы встречаются практически во всех реляционных базах данных, и зачастую их достаточно для моделирования традиционных приложений.

Числовые типы

Имя типа	Примечания	Размер	Диапазон
smallint	Эквивалент в SQL: Int2	2 байта	От –32 768 до +32 768
int	Эквивалент в SQL: Int4. Псевдоним – integer	4 байта	От –2 147 483 648 до +2 147 483 647
bigint	Эквивалент в SQL: Int8.	8 байтов	От –9 223 372 036 854 775 808 до +9 223 372 036 854 775 807
numeric или decimal	В PostgreSQL не различаются	Переменный	До 131 072 знаков перед запятой, до 16 383 знаков после запятой
real	Специальные значения: -Infinity, Infinity, NaN	4 байта	Платформенно-зависимый, точность не ниже 6 знаков. Чаще всего диапазон от 1E-37 до 1E+37

PostgreSQL поддерживает математические операторы и функции, в т. ч. тригонометрические функции и поразрядные операции. Тип `smallint` позволяет сэкономить место на диске, а `bigint` используется, когда диапазона типа `integer` не хватает.

Порядковые типы `smallserial`, `serial` и `bigserial` – надстройки над `smallint`, `int` и `bigint` соответственно. Они часто используются в качестве суррогатных ключей и по умолчанию не могут принимать значения `null`. За кулисами порядковый тип основан на последовательностях – объектах базы данных, которые генерируют арифметическую прогрессию с заданными минимумом, максимумом и величиной инкремента. Например, следующая команда создает таблицу `customer` со столбцом `customer_id`:

```
CREATE TABLE customer (
  customer_id SERIAL
);
```

За кулисами генерируется такой код:

```
CREATE SEQUENCE custome_customer_id_seq;
CREATE TABLE customer (
  customer_id integer NOT NULL DEFAULT nextval('customer_customer_id_seq')
);
ALTER SEQUENCE customer_customer_id_seq OWNED BY customer.Customer_id;
```

При создании столбца типа `serial` имейте в виду следующее:

- будет создана последовательность с именем `tableName_columnName_seq`. В примере выше последовательность будет называться `customer_customer_id_seq`;
- на столбец будет наложено ограничение `Not Null`;
- со столбцом будет ассоциировано значение по умолчанию, возвращаемое функцией `nextval()`;
- последовательность будет принадлежать столбцу, т. е. автоматически удаляться при удалении столбца.

Предыдущий пример показывает, что PostgreSQL присваивает объекту имя, если оно не задано явно. Имена объектов имеют вид `{tablename}_{columnname(s)}_{suffix}`, где суффиксы `pkey`, `key`, `excl`, `idx`, `fkey` и `check` соответствуют ограничению первичного ключа, уникальному ограничению, ограничению исключения, индексу, ограничению внешнего ключа и проверочному ограничению. При работе с порядковыми типами часто забывают выдать необходимые разрешения на доступ к последовательности – это ошибка.

Как и в языке C, результат целого выражения – целое число. Поэтому математические операции `3/2` и `1/3` дают `1` и `0` соответственно, т. е. дробная часть отбрасывается. Но, в отличие от C, в PostgreSQL используется алгоритм округления к ближайшему четному в случае приведения типа `double` к `int`:

```
postgres=# SELECT CAST (5.9 AS INT) AS rounded_up, CAST(5.1 AS INTEGER) AS
rounded_down, CAST (-23.5 AS INT) as round_negative , 5.5::INT AS
another_syntax;
 rounded_up | rounded_down | round_negative | another_syntax
------------+--------------+----------------+----------------
          6 |            5 |            -24 |              6
(1 row)
postgres=# SELECT 2/3 AS "2/3", 1/3 AS "1/3", 3/2 AS "3/2";
 2/3 | 1/3 | 3/2
-----+-----+-----
   0 |   0 |   1
(1 row)
```

Типы `numeric` (или `decimal`) рекомендуется использовать для хранения денежных сумм и других величин, для которых требуется абсолютная точность. Значение типа numeric можно определить тремя способами:

- numeric (точность, масштаб);
- numeric (точность);
- numeric.

Точность – это общее количество цифр, а масштаб – количество цифр в дробной части. Например, у числа 12.344 точность равна 5, а масштаб – 3. Если тип `numeric` используется без указания точности и масштаба, то в столбце может храниться значение с любой точностью или масштабом.

Если абсолютная точность не нужна, не используйте типы `numeric` и `decimal`, т. к. операции с ними выполняются медленнее, чем с числами типа `float` и `double`.

Значения с одинарной и двойной точностью неточны, т. е. в некоторых случаях не могут быть точно представлены во внутреннем двоичном формате, и тогда хранятся в приближенном виде. Полную документацию по числовым типам можно найти на странице https://www.postgresql.org/docs/10/static/datatype-numeric.html.

Символьные типы

Имя типа	Примечания	Концевые пробелы	Максимальная длина
char	Эквивалентно char(1)	Семантически не значимы	1
name	Эквивалентно varchar(64). Используется PostgreSQL для имен объектов	Семантически значимы	64
char(n)	Псевдоним: character(n). Последовательность символов фиксированной длины. Внутреннее название – bpchar (blank padded character)	Семантически не значимы	10485760
varchar(n)	Псевдоним: character varying(n). Последовательность символов переменной длины, но не более n	Семантически значимы	10485760
text	Последовательность символов переменной длины	Семантически значимы	Не ограничена

В PostgreSQL имеется два символьных типа: `char(n)` и `varchar(n)`, где `n` – максимально допустимое число символов. В случае типа `char` значение дополняется справа пробелами до указанной длины. При выполнении операций над значениями типа `char` концевые пробелы игнорируются. Рассмотрим пример:

```
postgres=# SELECT 'a'::CHAR(2) = 'a '::CHAR(2) ,length('a '::CHAR(10));
 ?column? | length
----------+--------
t         | 1
(1 row)
```

Выполнять бинарные операции над значениями типа varchar или text и строками символов не рекомендуется из-за концевых пробелов.

Для обоих типов, `char` и `varchar`, попытка записать в столбец строку, длина которой превосходит максимально допустимую, командой `INSERT` или `UPDATE`

приводит к ошибке, если только лишние символы не являются пробелами, а в этом случае строка усекается. Приведение типа автоматически приводит к усечению – ошибка не возникает. В следующем примере показаны проблемы, возникающие из-за смешения данных разных типов:

```
postgres=# SELECT 'a '::VARCHAR(2)='a '::text, 'a '::CHAR(2)='a '::text, 'a
'::CHAR(2)='a '::VARCHAR(2);
 ?column? | ?column? | ?column?
----------+----------+----------
 t | f | t
(1 row)

postgres=# SELECT length ('a '::CHAR(2)), length ('a '::VARCHAR(2));
 length | length
--------+--------
      1 |      2
(1 row)
```

Здесь `'a '::CHAR(2)` равно `'a '::VARCHAR(2)`, но длины строк различны, что противоречит логике. Также мы видим, что `'a'::CHAR(2)` не равно `'a '::text`. И наконец, `'a '::VARCHAR(2)` равно `'a'::text`. Возникает противоречие, потому что, согласно правилам математики, если `a` равно `b` и `b` равно `c`, то `a` должно быть равно `c`.

В PostgreSQL размер области, выделенной для хранения текста, зависит от длины текста, а также его кодировки и сжатия. Тип данных `text` можно интерпретировать как тип `varchar()` без ограничения длины. Максимально возможный размер текста равен 1 ГБ, что совпадает с максимальным размером столбца.

В случае строк фиксированной длины типы `character` и `character varying` занимают одинаковое место на диске. Для строк переменной длины тип `character varying` занимает меньше места, потому что к значению типа `character` справа дописываются пробелы. В примере ниже показано, сколько места занимают строки фиксированной и переменной длины. Мы просто создаем две таблицы и заполняем их случайными данными.

```
CREATE TABLE char_size_test (
  size CHAR(10)
);
CREATE TABLE varchar_size_test(
  size varchar(10)
);
WITH test_data AS (
  SELECT substring(md5(random()::text), 1, 5) FROM generate_series (1, 1000000)
),
char_data_insert AS (
  INSERT INTO char_size_test SELECT * FROM test_data
)INSERT INTO varchar_size_test SELECT * FROM test_data;
```

Теперь получим размеры таблиц:

```
postgres=# \dt+ varchar_size_test
                          List of relations
 Schema | Name              | Type  | Owner    | Size  | Description
--------+-------------------+-------+----------+-------+-------------
```

```
 public | varchar_size_test | table | postgres | 35 MB |
(1 row)

postgres=# \dt+ char_size_test
                            List of relations
 Schema | Name           | Type  | Owner    | Size  | Description
--------+----------------+-------+----------+-------+-------------
 public | char_size_test | table | postgres | 42 MB |
(1 row)
```

Тип varchar можно эмулировать с помощью типа text, дополненного ограничением на длину текста. Так, следующие два фрагмента кода эквивалентны:

```
CREATE TABLE emulate_varchar(
    test VARCHAR(4)
);
--семантически эквивалентно
CREATE TABLE emulate_varchar (
    test TEXT,
    CONSTRAINT test_length CHECK (length(test) <= 4)
);
```

В PostgreSQL производительность всех символьных типов данных одинакова, поэтому рекомендуется использовать тип text, поскольку так разработчику будет проще реагировать на изменение требований. Часто требуется изменить длину текста, например номера квитанции с 6 до 8 символов. Если использовать тип text, то для реализации нового требования достаточно будет модифицировать проверочное ограничение, не изменяя структуру таблицы. Полная документация по символьным типам данных приведена на странице https://www.postgresql.org/docs/10/static/datatype-character.html.

Дата и время

Типы даты и времени используются при описании времени возникновения событий, например даты рождения. PostgreSQL поддерживает следующие типы:

Имя типа	Размер в байтах	Описание	Нижняя граница	Верхняя граница
timestamp without timezone	8	Дата и время без часового пояса, эквивалентно timestamp	4713 до н. э.	294276 н. э.
timestamp with timezone	8	Дата и время с часовым поясом, эквивалентно timestamptz	4713 до н. э.	294276 н. э.
date	4	Только дата	4713 до н. э.	294276 н. э.
time without timezone	8	Время суток	00:00:00	24:00:00
time with timezone	12	Время суток с часовым поясом	00:00:00+1459	24:00:00-1459
interval	16	Временной интервал	–178 000 000 лет	+178 000 000 лет

В PostgreSQL временные метки с часовым поясом и без хранятся в **универсальном координированном времени** (**UTC**), и без часового пояса может

храниться только время. Это объясняет, почему размер типов `timestamp with time zone` и `timestamp without time zone` одинаков.

Правильно использовать временные метки можно двумя способами. Первый – задавать всюду тип `timestamp without time zone`, возложив на клиента учет часовых поясов. Это полезно, когда разрабатывается приложение для внутреннего пользования в одном часовом поясе, а также когда клиенты знают о различии часовых поясов.

Второй подход – всюду использовать тип `timestamp with time zone`. Ниже приведены советы, как избежать подводных камней при работе с типом `timestamptz`.

- Обязательно устанавливайте часовой пояс по умолчанию для всех подключений. Для этого нужно настроить часовой пояс в файле postgresql.conf. Поскольку PostgreSQL хранит временную метку с часовым поясом в формате UTC, рекомендуется задавать в качестве часового пояса по умолчанию тоже UTC. Заодно UTC позволит преодолеть потенциальные проблемы с **летним временем** (Daylight Saving Time – **DST**).
- Часовой пояс следует задавать в каждой операции создания, чтения, обновления и удаления.
- Не выполняйте операции, в которых участвуют типы с часовым поясом и без, поскольку обычно результат оказывается неверным из-за внутренних преобразований.
- Не изобретайте собственных преобразований, пусть преобразованием времени из одного часового пояса в другой занимается сервер.
- Изучите типы данных в языке программирования, чтобы понять, какие типы следует использовать совместно с PostgreSQL, чтобы избежать лишней обработки.

В PostgreSQL есть два важных параметра: `time zone` и `datestyle`. Параметр `datestyle` играет двоякую роль:

- **задание формата отображения:** `datestyle` определяет, как форматировать значения типа `timestamp` и `timestamptz`;
- **интерпретация неоднозначных данных:** `datestyle` определяет, как интерпретировать значения типа `timestamp` и `timestamptz`.

Представления `pg_timezone_names` и `pg_timezone_abbrevs` дают полный список полных и сокращенных названий часовых поясов соответственно, а также информацию о смещении от UTC и о том, применяется ли в данном часовом поясе летнее время. В следующем фрагменте устанавливается часовой пояс Иерусалима, после чего выводится локальная дата и время в Иерусалиме:

```
postgres=# SET timezone TO 'Asia/jerusalem';
SET
postgres=# SELECT now();
              now
-------------------------------
 2017-11-02 18:06:29.336462+02
(1 row)
```

Команда AT TIME ZONE преобразует временную метку с часовым поясом или без в указанный часовый пояс. Ее поведение зависит от преобразуемого типа.

```
postgres=# SHOW timezone;
    TimeZone
---------------
 Asia/Jerusalem
(1 row)

postgres=# SELECT now(), now()::timestamp, now() AT TIME ZONE 'CST',
now()::timestamp AT TIME ZONE 'CST';
                          now |                      now | timezone   | timezone
------------------------------+--------------------------+------------
--------+----------------------------
 2017-11-02 18:07:33.8087+02 | 2017-11-02 18:07:33.8087 | 2017-11-02
10:07:33.8087 | 2017-11-03 02:07:33.8087+02
(1 row)
```

Функция now() возвращает текущую временную метку в часовом поясе Asia/Jerusalem. Обратите внимание, что смещение равно +02. В результате приведения типа timestamptz к timestamp в конструкции now()::timestamp смещение часового пояса отбрасывается.

Выражение now() AT TIME ZONE 'CST' преобразует временную метку в часовом поясе Jerusalem во временную метку в часовом поясе CST. Поскольку центральное стандартное время смещено на -6 часов от UTC, а иерусалимское время – на +2, то вычитается 8 часов.

Последнее выражение now()::timestamp AT TIME ZONE 'CST' интерпретирует текущее время так, будто оно находится в часовом поясе CST, и преобразует его в часовой пояс, установленный для подключения, Asia/jerusalem. Следовательно, это выражение эквивалентно такому:

```
postgres=# SELECT ('2017-11-02 18:07:33.8087'::timestamp AT time zone 'CST'
AT TIME ZONE 'Asia/jerusalem')::timestamptz;
          timezone
----------------------------
 2017-11-03 02:07:33.8087+02
(1 row)
```

Преобразования между временным метками с часовыми поясами и без можно резюмировать следующим образом:

- выражение 'y'::TIMESTAMP WITHOUT TIMEZONE AT TIME ZONE 'x' означает, что значение y типа TIMESTAMP будет преобразовано из часового пояса x в часовой пояс сеанса;
- выражение 'y'::TIMESTAMP WITH TIMEZONE AT TIME ZONE 'x' преобразует значение y типа TIMESTAMPTZ в значение типа TIMESTAMP в указанном часовом поясе x.

PostgreSQL понимает, как интерпретировать временные метки с часовым поясом. В следующем примере показано, как PostgreSQL обрабатывает летнее время:

```
postgres=# SET timezone TO 'Europe/Berlin';
SET
postgres=# SELECT '2017-03-26 2:00:00'::timestamptz;
      timestamptz
------------------------
 2017-03-26 03:00:00+02
(1 row)
```

Тип `date` рекомендуется использовать, когда время не важно, например для задания даты рождения, праздников и дней отсутствия на работе. Для хранения времени с часовым поясом требуется 12 байтов: 8 для времени и 4 для часового пояса. Для хранения времени без часового пояса требуется только 8 байтов. Для преобразования между часовыми поясами можно использовать конструкцию `AT TIME ZONE`.

Наконец, тип `interval` представляет период времени, например предположительное время работы над задачей. Он очень важен для операций с временными метками.

Результат простых арифметических операций, например + и –, над значениями типа `timestamp` и `timestamptz` имеет тип `interval`. Результат тех же операций над значениями типа `date` имеет тип `integer`. В следующем примере показано вычитание значений типа `timestamptz` и `date`. Обратите внимание на формат задания интервалов:

```
postgres=# SELECT '2014-10-11'::date - '2014-10-10'::date = 1,
   '2014-09-01 23:30:00'::timestamptz -'2014-09-01 22:00:00'::timestamptz =
    Interval '1 hour, 30 minutes';
 ?column? | ?column?
----------+----------
        t |        t
(1 row)
```

База данных сайта торговли автомобилями

Сейчас мы уже можем преобразовать логическую модель базы сайта торговли автомобилями из главы 1 в физическую. При создании таблиц сверяйтесь со следующим минимальным контрольным списком.

- Что будет первичным ключом?
- Каковы значения по умолчанию у каждого столбца?
- Каков тип каждого столбца?
- Каковы ограничения на столбцы или группы столбцов?
- Правильно ли заданы разрешения для таблиц, последовательностей и схем?
- Заданы ли внешние ключи с соответствующими действиями?
- Каков жизненный цикл данных?
- Какие операции над данными разрешены?

При создании схемы базы данных мы не будем строго следовать формальной реляционной модели, а вместо натуральных ключей будем использовать суррогатные. У суррогатных ключей есть ряд преимуществ:

- натуральные ключи могут изменяться: никто не мешает поменять адрес электронной почты. С одной стороны, можно усомниться, а является ли вообще *натуральным ключом* то, что может изменяться. Но, с другой стороны, можно возразить, что даже после изменения он однозначно идентифицирует человека. Так или иначе, суррогатный ключ гарантирует, что если одна строка ссылается на другую, то эта ссылка не станет недействительной из-за изменения ключа;
- некорректные предположения о натуральных ключах. Возьмем, к примеру, адрес электронной почты. Предполагается, что он уникально идентифицирует человека. Но это неправда – некоторые поставщики почтовых служб устанавливают политику истечения срока действия адреса при неактивности владельца. У частных компаний могут быть коллективные адреса вида `contact@...`, `support@...` и т. д. То же относится к номерам телефонов, фиксированных и мобильных;
- суррогатные ключи можно использовать для прослеживания времени в реляционной базе данных. Так, в некоторых компаниях очень строгие требования к безопасности, согласно которым после каждой операции создается новая версия данных. Для суррогатных ключей используются компактные типы данных, например integer. Поэтому производительность оказывается выше, чем при работе с натуральными ключами;
- в PostgreSQL суррогатные ключи можно использовать для устранения эффекта искажения статистики. По умолчанию сервер собирает статистику для каждого столбца в отдельности. Иногда это некорректно, потому что столбцы коррелируют между собой. В таком случае PostgreSQL передает неправильную оценку планировщику, что приводит к генерации неоптимальных планов. Чтобы преодолеть это ограничение, разработчик может сообщить серверу о межстолбцовой корреляции;
- средства объектно-реляционного отображения, например Hibernate, поддерживают суррогатные ключи лучше, чем натуральные.

Но у суррогатных ключей есть и недостатки:

- суррогатный ключ генерируется автоматически, поэтому могут получаться разные значения. Например, мы вставляем вроде бы одни и те же данные в тестовую и промежуточную базу, а потом оказывается, что записи различаются;
- суррогатный ключ бессодержателен. Информативнее ссылаться на человека по имени, а не по автоматически сгенерированному числу;
- суррогатные ключи могут приводить к избыточности данных и без должной осторожности порождать кортежи-дубликаты.

На диаграмме сущность-связь для сайта торговли автомобилями есть сущность `user`. Поскольку `user` – зарезервированное слово, мы назовем соответствующую таблицу `account`. Попутно отметим, что если очень хочется использовать для именования объекта базы данных зарезервированное слово, то нужно заключить его в кавычки. Ниже показано, как создать таблицу с именем `user`:

```
postgres=# \set VERBOSITY 'verbose'
postgres=# CREATE TABLE user AS SELECT 1;
ERROR: 42601: syntax error at or near "user"
LINE 1: CREATE TABLE user AS SELECT 1;
                     ^
LOCATION: scanner_yyerror, scan.l:1086
postgres=# CREATE TABLE "user" AS SELECT 1;
SELECT 1
```

В предыдущем примере установлен параметр `psql VERBOSITY`, поэтому показываются коды и подробные сообщения об ошибках. Коды ошибок полезны, когда нужно перехватывать исключения.

Для создания таблицы `account` выполните следующую команду:

```
CREATE TABLE account (
  account_id SERIAL PRIMARY KEY,
  first_name TEXT NOT NULL,
  last_name TEXT NOT NULL,
  email TEXT NOT NULL UNIQUE,
  password TEXT NOT NULL,
  CHECK (email ~* '^\w+@\w+[.]\w+$'),
  CHECK (char_length(password)>=8)
);
```

Сделаем несколько замечаний:

- атрибут `account_id` типа `serial` выбран в качестве первичного ключа. Он по определению уникален и не может быть равен null;
- атрибуты `first_name`, `last_name`, `email` и `password` не могут принимать значение null;
- пароль `password` должен содержать не менее восьми символов. На практике длина пароля контролируется на уровне бизнес-логики, потому что пароль не должен храниться в базе данных в открытом виде. Дополнительные сведения о безопасности см. в главе 11;
- атрибут `email` сопоставляется с регулярным выражением (отметим, что это регулярное выражение сильно упрощено, адреса электронной почты могут быть сложнее).

Неявно создаются следующие объекты:

- последовательность для поддержки типа `serial`;
- два индекса, оба уникальных. Первый – индекс по первичному ключу `account_id`. Второй нужен для проверки адреса электронной почты.

Для создания таблицы `seller_account` выполните следующую команду:

```
CREATE TABLE seller_account (
  seller_account_id SERIAL PRIMARY KEY,
  account_id INT UNIQUE NOT NULL REFERENCES
  account(account_id),
  number_of_advertizement INT DEFAULT 0,
  user_ranking float,
  total_rank float
);
```

Как легко видеть, между таблицами `seller_account` и `account` имеется связь один-к-одному. Это гарантируется ограничениями `NOT NULL` и `UNIQUE` на атрибут `account_id`. В данном случае для моделирования учетной записи продавца можно было бы также сделать `account_id` первичным ключом:

```
CREATE TABLE seller_account (
  account_id INT PRIMARY KEY REFERENCES account(account_id)
  ...
);
```

Но первый подход более гибкий. Требования могут измениться, и тогда связь между учетной записью пользователя и учетной записью продавца может стать типа один-ко-многим, а не один-к-одному. Например, понятие пользователя можно обобщить на компании и разрешить компании иметь несколько учетных записей продавца.

Для создания таблицы `car` нам прежде всего нужна модель автомобиля. Это важно, поскольку даст пользователю приложения – в основном продавцу – возможность выбирать модель, а не вводить ее название. К тому же у продавца может и не быть полной информации о модели. Да и вообще, если разрешить вводить модель вручную, то могут возникнуть несогласованности из-за ошибок пользователя. В реальных приложениях справочную информацию – названия валют, стран и моделей автомобилей – можно запросить у специализированных поставщиков. Так, сведения о странах предоставляются ISO и публикуются на странице https://www.iso.org/iso-3166-country-codes.html.

Резюме

В этой главе мы рассмотрели основные строительные блоки PostgreSQL. В кластере баз данных имеется ряд объектов, общих для всех баз. Это роли, табличные пространства, шаблонные базы данных, шаблонные процедурные языки, а также некоторые параметры. Табличное пространство – это область в системе хранения, которая обычно используется администратором баз данных для оптимизации и обслуживания.

База данных `template1` клонируется при каждом создании новой базы. В нее можно загрузить расширения, которые должны присутствовать во всех базах данных. База данных `template0` – запасная копия на случай повреждения `template1`. Кроме того, ее можно использовать, если в `template1` определена не та локаль, что нужно.

У роли имеется несколько атрибутов, например `login`, `superuser` и `createdb`. В старых версиях PostgreSQL роль называлась пользователем, если ей был разрешен вход в базу данных, и группой в противном случае. Роль может быть членом другой роли – это упрощает управление разрешениями в базе данных.

В PostgreSQL более двухсот параметров, управляющих поведением базы данных. Они действуют в разных контекстах, а именно: внутренние, сервер-

ные, заднего плана, пользовательские, суперпользовательские и SIGHUP. Просмотреть их можно с помощью представления `pg_settings`.

Пользовательская база данных – это контейнер для схем, таблиц, представлений, функций, диапазонов, доменов, последовательностей и индексов. Существует три вида разрешений: для создания новых объектов, для создания временных таблиц и для подключения. Ряд аспектов поведения базы данных можно контролировать с помощью команды `ALTER DATABASE`. В каталожной таблице `pg_database` хранятся описания всех баз данных в кластере.

PostgreSQL располагает обширным набором типов данных, включая числовые, символьные и дату и время. Правильный выбор типа данных – компромисс между расширяемостью, потреблением памяти и производительностью. Следует с осторожностью подходить к выполнению операций с данными разных типов, поскольку при этом производятся неявные преобразования. Например, следует хорошо знать, как ведет себя система при сравнении данных типа `text` и `varchar`. Это относится также к типам даты и времени.

Таблицы – основные структурные элементы PostgreSQL; сервер использует их также для реализации представлений и последовательностей. Таблицы бывают временными и постоянными. В процессе потоковой репликации нежурналируемые (`unlogged`) таблицы не реплицируются на ведомые узлы.

В следующей главе мы рассмотрим дополнительные структурные элементы: индексы и представления. Прочитав эти главы, вы будете понимать, как спроектировать и реализовать физическую базу данных для приложения.

Глава 4

Дополнительные строительные блоки PostgreSQL

В этой главе мы рассмотрим остальные строительные блоки PostgreSQL: представления, индексы, функции, триггеры и правила. Мы также расскажем о DDL-командах CREATE и ALTER. Поскольку лексическую структуру и DML-команды мы еще не обсуждали, постараемся ограничиться самыми простыми DML-командами. Итак, в этой главе будут рассмотрены следующие вопросы:

- **представления:** это важная часть моделирования базы данных, поскольку играет роль интерфейса, или уровня абстракции. Мы обсудим синтаксис определения представлений, порядок их использования и приведем пример обновляемого представления;
- **индексы:** это секретный соус, помогающий обеспечивать согласованность и высокую производительность. Мы обсудим типы индексов;
- **функции:** применяются для реализации сложной логики внутри самой базы данных, но могут использоваться так же, как представления. В этой главе функции обсуждаются кратко, а более полное рассмотрение отложено до главы 7;
- **пользовательские типы данных:** важное преимущество PostgreSQL – возможность определять новые типы данных. В этом разделе мы рассмотрим несколько ситуаций, когда пользовательские типы данных помогают решить задачу;
- **триггеры и правила:** позволяют разработчику обрабатывать события, генерируемые командами INSERT, UPDATE, DELETE и другими. Триггеры используются для моделирования сложных бизнес-требований, которые трудно реализовать средствами одного лишь SQL.

Представления

Представление можно считать именованным запросом или оберткой вокруг команды SELECT. Представления – существенный строительный блок реляци-

онных баз данных с точки зрения UML-моделирования; его можно интерпретировать как метод UML-класса. Представления используются для следующих целей:

- чтобы упростить сложные запросы и повысить степень модульности кода;
- для повышения производительности посредством кеширования результатов и использования их в будущем;
- чтобы уменьшить объем SQL-кода;
- чтобы перебросить мост между реляционными базами данных и объектно-ориентированными языками (особенно в этом смысле полезны обновляемые представления);
- чтобы реализовать авторизацию на уровне строк – не давать доступа к строкам, не удовлетворяющим заданному условию;
- для реализации интерфейсов и уровня абстракции, расположенного между языками высокого уровня и реляционными базами;
- для реализации срочных изменений.

Представление должно отвечать текущим, а не предполагаемым будущим потребностям бизнеса. Его следует проектировать с учетом представления конкретной функциональности. Отметим, что чем больше в представлении атрибутов, тем больше усилий придется приложить для его рефакторинга. Кроме того, если представление агрегирует данные из многих таблиц и используется в качестве интерфейса, то возможно снижение производительности. Причин тому много, например неоптимальный план выполнения из-за устаревшей статистики некоторых таблиц и т. д.

Если сложная бизнес-логика реализуется в базе данных с помощью представлений и хранимых процедур, то рефакторинг базы, а в особенности базовых таблиц, может превратиться в кошмар. Чтобы избежать этого, подумайте о переносе бизнес-логики на уровень приложения.

У некоторых систем, например средств объектно-реляционного отображения, могут быть специальные требования, допустим наличие уникального ключа. Это ограничивает применимость в них представлений, однако в какой-то мере проблему можно сгладить, подменив первичные ключи оконными функциями, например `row_number`.

В PostgreSQL представление за кулисами моделируется как таблица с правилом `_RETURN`. То есть теоретически можно создать таблицу и преобразовать ее в представление, но делать так не рекомендуется. Дерево зависимостей представления строго контролируется, т. е. невозможно удалить или структурно изменить представление, от которого зависят другие представления:

```
postgres=# CREATE VIEW test AS SELECT 1 as v;
CREATE VIEW
postgres=# CREATE VIEW test2 AS SELECT v FROM test;
CREATE VIEW
postgres=# CREATE OR REPLACE VIEW test AS SELECT 1 as val;
ERROR: cannot change name of view column "v" to "val"
```

Синтаксис определения представления

Показанная ниже команда CREATE VIEW позволяет создать представление, а при наличии ключевого слова REPLACE заменить уже существующее представление. Имена атрибутов представления можно задать явно, в противном случае они наследуются от команды SELECT:

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] [ RECURSIVE ] VIEW name [ (
column_name [, ...] ) ]
    [ WITH ( view_option_name [= view_option_value] [, ... ] ) ]
    AS query
    [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

Синтаксис создания материализованных представлений отличается от показанного выше. Он описан ниже в разделе «Материализованные представления».

В примере ниже показано, как создать представление, которое выводит всю информацию о пользователе, кроме пароля. Это может быть полезно, чтобы ограничить доступ приложения к паролю. Отметим, что имена столбцов наследуются от столбцов в команде SELECT, как показывает метакоманда \d для объекта account_information:

```
car_portal=> CREATE VIEW account_information AS SELECT account_id,
first_name, last_name, email FROM account;
CREATE VIEW
car_portal=> \d account_information
        View "car_portal_app.account_information"
   Column   |   Type  | Collation | Nullable | Default
------------+---------+-----------+----------+---------
account_id  | integer |           |          |
first_name  | text    |           |          |
last_name   | text    |           |          |
email       | text    |           |          |
```

Имена столбцов представления можно задать и явно, как показано в примере ниже:

```
CREATE OR REPLACE VIEW account_information
(account_id,first_name,last_name,email) AS SELECT account_id, first_name,
last_name, email FROM account;
```

Если определение представления изменяется с помощью ключевого слова REPLACE, то списки столбцов в старом и новом представлениях должны быть одинаковы, включая имя, тип и порядок следования. В следующем примере показано, что произойдет при попытке изменить порядок столбцов:

```
car_portal=> CREATE OR REPLACE VIEW account_information AS SELECT
account_id, last_name, first_name, email FROM account;
ERROR: cannot change name of view column "first_name" to "last_name"
```

Категории представлений

В PostgreSQL имеется несколько категорий представлений.

- **Временные представления.** Такое представление автоматически удаляется в конце сеанса. Если ключевое слово `TEMPORARY` или `TEMP` отсутствует, то жизненный цикл представления начинается в момент создания и заканчивается в момент явного удаления.
- **Рекурсивные представления.** Рекурсивное представление напоминает рекурсивную функцию в языках программирования. Список столбцов в этом случае обязателен. Рекурсия, в частности рекурсивные представления и рекурсивные **общие табличные выражения** (CTE), позволяет строить очень сложные запросы, особенно для иерархически организованных данных.
- **Обновляемые представления.** Обновляемые представления позволяют обращаться с представлениями, как с таблицами, т. е. выполнять команды `INSERT`, `UPDATE` и `DELETE`. Обновляемые представления в какой-то мере можно рассматривать как мост между реляционной и объектной моделями, они дают некий аналог полиморфизма.
- **Материализованные представления.** Это, по сути дела, таблица, содержимое которой периодически обновляется заранее заданным запросом. Материализованные представления повышают производительность запросов, которые выполняются долго, и часто применяются к статическим данным. Можно считать их вариантом кеширования. Рекурсия будет рассмотрена в последующих главах, а здесь мы сосредоточимся на обновляемых и материализованных представлениях.

Материализованные представления

Синтаксис определения материализованных и обычных представлений несколько различается. Материализованные представления – расширение PostgreSQL, но их поддерживают и другие СУБД, например Oracle. Как показано ниже, материализованное представление можно создать в определенном табличном пространстве, и для него можно задать параметром хранения `storage_parameter`, что вполне логично, т. к. материализованные представления – физические объекты:

```
CREATE MATERIALIZED VIEW [ IF NOT EXISTS ] table_name
    [ (column_name [, ...] ) ]
    [ WITH ( storage_parameter [= value] [, ... ] ) ]
    [ TABLESPACE tablespace_name ]
    AS query
    [ WITH [ NO ] DATA ]
```

В момент создания материализованного представления его можно заполнить или оставить пустым. Для заполнения пустого материализованного представления служит команда `REFRESH MATERIALIZED VIEW` с таким синтаксисом:

```
REFRESH MATERIALIZED VIEW [ CONCURRENTLY ] name [ WITH [ NO ] DATA ]
```

Попытка выбрать данные из незаполненного представления заканчивается ошибкой, показанной в примере ниже:

```
car_portal=> CREATE MATERIALIZED VIEW test_mat AS SELECT 1 WITH NO DATA;
CREATE MATERIALIZED VIEW
car_portal=> TABLE test_mat;
ERROR: materialized view "test_mat" has not been populated
HINT: Use the REFRESH MATERIALIZED VIEW command.
```

Обновить представление, как предлагается в подсказке, можно следующим образом:

```
car_portal=> REFRESH MATERIALIZED VIEW test_mat;
REFRESH MATERIALIZED VIEW
car_portal=> TABLE test_mat;
 ?column?
----------
        1
(1 row)
```


Обновление материализованного представления – блокирующая команда, т. е. одновременно выполняемые команды SELECT будут приостановлены на время, пока производится обновление. Эту проблему можно решить с помощью параллельного обновления, но тогда над представлением должен быть построен уникальный индекс.

Материализованные представления часто используются совместно с хранилищами данных. В этом случае выполняется ряд запросов в интересах бизнес-аналитики и поддержки принятия решений. Данные в таких приложениях изменяются редко, но вычисления и агрегирование занимают много времени. В общем случае материализованные представления применяются для следующих целей:

- формирование сводных отчетов;
- кеширование результатов повторяющихся запросов;
- повышение производительности благодаря однократной обработке данных.

Поскольку материализованные представления – это таблицы, их можно индексировать, что резко ускоряет работу с ними.

Обновляемые представления

По умолчанию простые представления в PostgreSQL являются автообновляемыми, т. е. к ним можно применять команды DELETE, INSERT и UPDATE, которые воздействуют на данные в базовой таблице. Если представление не является обновляемым (а значит, и простым) из-за нарушения одного из перечисленных ниже ограничений, то его все-таки можно сделать таковым с помощью триггеров и правил. Представление является автоматически обновляемым, если выполнены следующие условия:

- представление должно быть построено только над одной таблицей или одним обновляемым представлением;
- определение представления не содержит на верхнем уровне следующих фраз и теоретико-множественных операторов: DISTINCT, WITH, GROUP BY, OFFSET, HAVING, LIMIT, UNION, EXCEPT, INTERSECT;
- в списке select должны быть только сами столбцы базовой таблицы, использование функций и выражений не допускается. Кроме того, столбцы не должны повторяться;
- не должно быть установлено свойство security_barrier.

Для сайта торговли автомобилями можно определить обновляемое представление, показывающее только учетные записи, не принадлежащие продавцам:

```
CREATE VIEW user_account AS
SELECT account_id, first_name, last_name, email, password
FROM account WHERE account_id NOT IN (SELECT account_id FROM seller_account);
```

Для проверки попробуем вставить в него строку:

```
car_portal=> INSERT INTO user_account VALUES
(default,'first_name1','last_name1','test@email.com','password');
INSERT 0 1
```

Для автообновляемого представления нельзя изменить строку, которую представление не возвращает. Попробуем вставить учетную запись вместе с записью продавца, а затем удалить ее.

```
car_portal=> WITH account_info AS ( INSERT INTO user_account VALUES
(default,'first_name2','last_name2','test2@email.com','password') RETURNING
account_id)
INSERT INTO seller_account (account_id, street_name, street_number,
zip_code, city) SELECT account_id, 'street1', '555', '555', 'test_city'
FROM account_info;
INSERT 0 1
```

Обратите внимание, что вставка в представление user_account прошла успешно. Но тем не менее удалить эту запись с помощью обновляемого представления не получится – помешает проверочное ограничение:

```
car_portal=> DELETE FROM user_account WHERE first_name = 'first_name2';
DELETE 0
car_portal=> SELECT * FROM account where first_name like 'first_name%';
 account_id |  first_name |  last_name |           email | password
------------+-------------+------------+-----------------+----------
        482 | first_name1 | last_name1 | test@email.com  | password
        484 | first_name2 | last_name2 | test2@email.com | password
(2 rows)
```

Для управления поведением автоматически обновляемых представлений служит фраза WITH CHECK OPTION. Если она отсутствует, то команды UPDATE и INSERT успешно выполняются, даже если строка не видна в представлении, что риско-

ванно с точки зрения безопасности. Эта возможность демонстрируется в примере ниже. Сначала создадим таблицу:

```
CREATE TABLE a (val INT);
CREATE VIEW test_check_option AS SELECT * FROM a WHERE val > 0
WITH CHECK OPTION;
```

Для тестирования `CHECK OPTION` попробуем вставить строку, не удовлетворяющую проверочному условию:

```
car_portal=> INSERT INTO test_check_option VALUES (-1);
ERROR: new row violates check option for view "test_check_option"
DETAIL: Failing row contains (-1).
```

Дополнительные сведения о представлениях см. на странице https://www.postgresql.org/docs/current/static/sql-createview.html. Если вы не уверены, является представление автоматически обновляемым или нет, то можете узнать, проверив флаг `is_insertable_into` в таблице `information_schema`:

```
car_portal=# SELECT table_name, is_insertable_into FROM
information_schema.tables WHERE table_name = 'user_account';
   table_name | is_insertable_into
--------------+--------------------
 user_account | YES
(1 row)
```

Индексы

Индекс – это физический объект базы данных, построенный над одним или несколькими столбцами таблицы. В PostgreSQL есть несколько типов индексов и, соответственно, способов их использования. Индексы применяются для решения следующих задач:

- **оптимизация производительности** – индекс позволяет эффективно выбирать из таблицы небольшое число строк. Что такое «небольшое» число, определяется общим количеством строк в таблице и параметрами плана выполнения;
- **контроль ограничений** – индекс позволяет проверять заданные для строк ограничения. Например, для проверки ограничения `UNIQUE` автоматически создается индекс по соответствующему столбцу.

В следующем примере показано, как использовать GIST-индекс для запрета перекрывающихся диапазонов дат. Дополнительные сведения см. на странице https://www.postgresql.org/docs/current/static/rangetypes.html.

```
CREATE TABLE no_date_overlap (
    date_range daterange,
    EXCLUDE USING GIST (date_range WITH &&)
);
```

Проверим, попытавшись создать перекрывающиеся диапазоны:

```
car_portal=# INSERT INTO no_date_overlap values('[2010-01-01, 2020-01-01)');
INSERT 0 1
```

```
car_portal=# INSERT INTO no_date_overlap values('[2010-01-01, 2017-01-01)');
ERROR: conflicting key value violates exclusion constraint
"no_date_overlap_date_range_excl"
DETAIL: Key (date_range)=([2010-01-01,2017-01-01)) conflicts with existing
key (date_range)=([2010-01-01,2020-01-01)).
```

Синтаксис создания индекса

Индексы создаются командой `CREATE INDEX`. Поскольку индекс – физический объект базы данных, мы можем указать табличное пространство и параметр хранения `storage_parameter`. Индекс можно строить по столбцам или по выражениям. Элементы индекса можно сортировать в порядке возрастания (`ASC`) или убывания (`DESC`). Кроме того, можно задать порядок сортировки для значений `NULL` – в начале или в конце индекса. Если индекс создается по текстовым полям, то можно также задать порядок сравнения (collation). Ниже приведен полный синтаксис команды.

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ [ IF NOT EXISTS ] name ] ON
table_name [ USING method ]
  ( { column_name | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC
| DESC ] [ NULLS { FIRST | LAST } ] [, ...] )
  [ WITH ( storage_parameter = value [, ... ] ) ]
  [ TABLESPACE tablespace_name ]
  [ WHERE predicate ]
```


Для первичных и уникальных ключей индексы создаются автоматически.

Избирательность индекса

Рассмотрим таблицу `account_history` в базе данных для сайта торговли автомобилями. У ограничения уникальности `UNIQUE (account_id, search_key, search_date)` есть две цели. Первая – проверка того, что ни один ключ поиска не вставляется дважды для одной и той же даты, даже если пользователь искал по нему несколько раз. Вторая – быстрая выборка данных. Пусть требуется вывести последние 10 поисков данного пользователя. Запрос может выглядеть так:

```
SELECT search_key FROM account_history WHERE account_id = <account>
GROUP BY search_key ORDER BY max(search_date) limit 10;
```

Этот запрос возвращает 10 записей, содержащих ключи поиска `search_key`, упорядоченные по дате поиска `search_date`. Если таблица `account_history` содержит миллионы строк, то чтение всех данных займет очень много времени. Но не в этом случае, поскольку наличие уникального индекса позволяет читать данные только для одного указанного пользователя.

Если таблица мала, то индексы над ней не используются. В этом случае планировщик PostgreSQL предпочитает просматривать таблицу целиком. Чтобы убедиться в этом, заполним таблицу `account_history` совсем небольшим набором данных:

```
WITH test_account AS (
  INSERT INTO account VALUES (1000, 'test_first_name', 'test_last_name',
                              'test@email.com', 'password')
  RETURNING account_id
),car AS ( SELECT i as car_model FROM (VALUES('brand=BMW'), ('brand=WV'))
AS foo(i)
),manufacturing_date AS ( SELECT 'year='|| i as date FROM generate_series
(2015, 2014, -1) as foo(i))
INSERT INTO account_history (account_id, search_key, search_date)
SELECT account_id, car.car_model||'&'||manufacturing_date.date, current_date
FROM test_account, car, manufacturing_date;
VACUUM ANALYZE;
```

Чтобы узнать, используется ли индекс, выполним такой запрос:

```
car_portal=> SELECT search_key FROM account_history WHERE account_id = 1000
GROUP BY search_key ORDER BY max(search_date) limit 10;
     search_key
--------------------
 brand=WV&year=2014
 brand=BMW&year=2014
 brand=WV&year=2015
 brand=BMW&year=2015
(4 rows)

car_portal=> EXPLAIN SELECT search_key FROM account_history WHERE
account_id = 1000 GROUP BY search_key ORDER BY max(search_date) limit 10;
QUERY PLAN
------------------------------------------------------------------------------
 Limit (cost=1.17..1.18 rows=3 width=23)
   -> Sort (cost=1.17..1.18 rows=3 width=23)
         Sort Key: (max(search_date))
         -> HashAggregate (cost=1.12..1.15 rows=3 width=23)
              Group Key: search_key
              -> Seq Scan on account_history (cost=0.00..1.10 rows=4 width=23)
                    Filter: (account_id = 1000)
```

В этом примере индекс не используется. Планировщик решает, применять индекс или нет, сообразуясь со стоимостью плана выполнения. Для одного и того же запроса, но с разными параметрами планировщик может выбрать разные планы, ориентируясь на гистограмму распределения данных. Даже если набор данных велик, но предикат, заданный в условии, отфильтровывает мало данных, индекс использоваться не будет. Чтобы продемонстрировать такую ситуацию, добавим строки еще для одной учетной записи и выполним запрос снова:

```
WITH test_account AS (
 INSERT INTO account VALUES (2000, 'test_first_name',
  'test_last_name','test2@email.com', 'password') RETURNING account_id
),car AS ( SELECT i as car_model FROM (VALUES('brand=BMW'), ('brand=WV'),
('brand=Audi'), ('brand=MB')) AS foo(i)
),manufacturing_date AS ( SELECT 'year='|| i as date FROM generate_series
(2017, 1900, -1) as foo(i))
INSERT INTO account_history (account_id, search_key, search_date) SELECT
```

```
account_id, car.car_model||'&'||manufacturing_date.date, current_date
FROM test_account, car, manufacturing_date;
VACUUM ANALYZE;
```

Выполняем запрос для второй учетной записи:

```
car_portal=> EXPLAIN SELECT search_key FROM account_history WHERE
account_id = 2000 GROUP BY search_key ORDER BY max(search_date) limit 10;
 QUERY PLAN
------------------------------------------------------------------------------------
 Limit (cost=27.10..27.13 rows=10 width=23)
 -> Sort (cost=27.10..28.27 rows=468 width=23)
 Sort Key: (max(search_date))
 -> HashAggregate (cost=12.31..16.99 rows=468 width=23)
 Group Key: search_key
 -> Seq Scan on account_history (cost=0.00..9.95 rows=472 width=23)
 Filter: (account_id = 2000)
(7 rows)
```

Индекс и на этот раз не используется, потому что выгоднее прочитать всю таблицу. Избирательность индекса очень низкая, т. к. для учетной записи `2000` имеется 427 строк, а для учетной записи `1000` – только четыре строки:

```
car_portal=> SELECT count(*), account_id FROM account_history group by
account_id;
 count | account_id
-------+------------
   472 |       2000
     4 |       1000
(2 rows)
```

Наконец, выполним тот же запрос для учетной записи `1000`. В этом случае избирательность высокая, поэтому индекс используется:

```
EXPLAIN SELECT search_key FROM account_history WHERE account_id = 1000
GROUP BY search_key ORDER BY max(search_date) limit 10;
QUERY PLAN
------------------------------------------------------------------------------
------------------------------------------------------------------
Limit (cost=8.71..8.72 rows=4 width=23)
  -> Sort (cost=8.71..8.72 rows=4 width=23)
       Sort Key: (max(search_date))
       -> GroupAggregate (cost=8.60..8.67 rows=4 width=23)
            Group Key: search_key
            -> Sort (cost=8.60..8.61 rows=4 width=23)
                  Sort Key: search_key
                  -> Bitmap Heap Scan on account_history
(cost=4.30..8.56 rows=4 width=23)
                        Recheck Cond: (account_id = 1000)
                        -> Bitmap Index Scan on
account_history_account_id_search_key_search_date_key (cost=0.00..4.30
rows=4 width=0)
                              Index Cond: (account_id = 1000)
(11 rows)
```

Типы индексов

PostgreSQL поддерживает индексы разных типов, каждый из них применяется в определенных ситуациях.

- **B-дерево (B-tree).** Это тип индекса по умолчанию, он выбирается, когда в команде `CREATE INDEX` тип не указан. Буква **B** означает balanced (**сбалансированное**), т. е. по разные стороны от каждого промежуточного узла дерева находится примерно одинаковое количество узлов. B-деревья эффективны для поиска по условию равенства, принадлежности диапазону и совпадения с null. Индекс типа B-tree можно строить для любых типов данных PostgreSQL.
- **Хеш-индекс.** До версии PostgreSQL 10 поддержка хеш-индексов была неполной. Не гарантировалась транзакционная безопасность, и не поддерживалась потоковая репликация на ведомые узлы. В PostgreSQL 10 эти ограничения сняты. Хеш-индексы полезны для поиска по условию равенства.
- **Обобщенный обратный индекс (GIN).** GIN-индекс полезен, когда несколько значений нужно отобразить на одну строку. Он используется со сложными структурами данных, например массивами, и для полнотекстового поиска.
- **Обобщенное дерево поиска (GiST).** GiST-индексы позволяют строить обобщенные сбалансированные древовидные структуры. Они полезны для индексации геометрических типов данных, а также для полнотекстового поиска.
- **GiST с двоичным разбиением пространства (SP-GiST).** Аналогичны GIST-индексам и поддерживают деревья поиска по двоичному разбиению пространства. Вообще говоря, индексы типа GIST, GIN и SP-GIST предназначены для работы со сложными пользовательскими типами данных.
- **Блочно-диапазонный индекс (BRIN).** Этот тип появился в версии PostgreSQL 9.5. BRIN-индекс полезен для очень больших таблиц, когда место на диске ограничено. Он медленнее B-деревьев, но занимает меньше места.

Категории индексов

Индексы можно классифицировать следующим образом:

- **частичный индекс** – индексируется только подмножество таблицы, удовлетворяющее заданному предикату; в определении индекса присутствует фраза `WHERE`. Идея в том, чтобы уменьшить размер индекса, сделав его более быстрым и удобным для обслуживания;
- **уникальный индекс** – гарантирует, что каждое значение встречается только один раз. В таблице `account` со столбцом `email` ассоциировано ограничение уникальности. Оно реализуется уникальным индексом, о чем свидетельствует метакоманда `\d`:

```
\d account
                    Table "car_portal_app.account"
 Column     |  Type   | Collation | Nullable | Default
------------+---------+-----------+----------+--------------------
-----------------------
account_id |  integer |           | not null |
nextval('account_account_id_seq'::regclass)
first_name |     text |           | not null |
last_name  |     text |           | not null |
email      |     text |           | not null |
password   |     text |           | not null |
Indexes:
"account_pkey" PRIMARY KEY, btree (account_id)
"account_email_key" UNIQUE CONSTRAINT, btree (email)
```

- **индекс по нескольким столбцам** применяется для поддержки запросов определенного вида. Рассмотрим запрос `SELECT * FROM table WHERE column`$_1$ `= constant`$_1$ `and column`$_2$ `= constant`$_2$ `AND ... column`$_n$ `= constant`$_n$; в этом случае можно создать индекс по столбцам `column`$_1$, `column`$_2$, ..., `column`$_n$, если n меньше или равно 32;
- **индекс по выражению** можно строить не только по нескольким столбцам, но и по выражениям, включающим вызовы функций.

Индекс может относиться сразу к нескольким категориям, например возможен уникальный частичный индекс. Это позволяет гибко приспосабливаться к бизнес-требованиям или добиваться максимального быстродействия. Так, в следующих главах мы воспользуемся индексом по результатам функции `lower()`или `upper()`:

```
car_portal=> CREATE index on account(lower(first_name));
CREATE INDEX
```

Такой индекс позволяет искать учетную запись по имени владельца без учета регистра:

```
SELECT * FROM account WHERE lower(first_name) = lower('foo');
```

Индекс по выражению используется, только если выражение во фразе `WHERE` В ТОЧНОСТИ совпадает с выражением, по которому строился индекс.

Еще одно применение индекса по выражению – фильтрация строк после приведения к другому типу данных. Например, время вылета можно хранить в виде `timestamp`, но ищем мы часто по дате, а не по времени.

Как уже было сказано, индекс может быть одновременно уникальным и частичным. Предположим, что имеется таблица `employee`, в которой у каждого работника, за исключением главы компании, имеется начальник. Это можно смоделировать с помощью самоссылающейся таблицы:

```
CREATE TABLE employee (employee_id INT PRIMARY KEY, supervisor_id INT);
ALTER TABLE employee ADD CONSTRAINT supervisor_id_fkey FOREIGN KEY
(supervisor_id) REFERENCES employee(employee_id);
```

Для гарантии того, что существует только одна строка без начальника, можно добавить такой уникальный индекс:

```
CREATE UNIQUE INDEX ON employee ((1)) WHERE supervisor_id IS NULL;
```

Уникальный индекс по константному выражению (1) допускает только одну строку со значением null. При вставке первой такой строки будет построен индекс с ключом 1. Вторая попытка вставить строку со значением null приведет к ошибке, потому что ключ 1 уже есть:

```
car_portal=> INSERT INTO employee VALUES (1, NULL);
INSERT 0 1
car_portal=> INSERT INTO employee VALUES (2, 1);
INSERT 0 1
car_portal=> INSERT INTO employee VALUES (3, NULL);
ERROR: duplicate key value violates unique constraint "employee_expr_idx"
DETAIL: Key ((1))=(1) already exists.
```

В настоящее время по нескольким столбцам можно строить только индексы типов `B-tree`, `GIN`, `GIST` и `BRIN`. При создании многостолбцового индекса порядок столбцов важен. Поскольку многостолбцовый индекс обычно большой, планировщик может предпочесть последовательный просмотр таблицы, а не поиск по индексу.

Рекомендации по работе с индексами

Часто бывает полезно индексировать столбцы, встречающиеся в предикатах и внешних ключах. Это дает PostgreSQL возможность не просматривать таблицу последовательно, а искать по индексу. Индексы дают выигрыш не только при выполнении `SELECT`, но также `DELETE` и `UPDATE`. Есть несколько случаев, когда индекс не используется; чаще всего это происходит, когда таблица мала. Для больших таблиц надо тщательно планировать емкость системы хранения, потому что размеры индексов могут быть очень велики. Отметим также, что наличие индексов снижает скорость вставки, потому что требуется обновлять индексы, а на это тоже уходит время.

Есть несколько каталожных таблиц и функций, полезных для обслуживания индексов, например таблица `pg_stat_all_indexes`, в которой хранится статистика использования индексов. Подробнее об обслуживании индексов см. главу 12.

При создании индексов проверяйте, что индекса по тем же столбцам (или выражениям) еще не существует, иначе могут появиться дублирующие индексы. PostgreSQL в этом случае не выдает предупреждений:

```
car_portal=# CREATE index on car_portal_app.account(first_name);
CREATE INDEX
car_portal=# CREATE index on car_portal_app.account(first_name);
CREATE INDEX
```

Редко, но бывает, что индекс чрезмерно разрастается. Для перестраивания индекса PostgreSQL предоставляет команду REINDEX. Отметим, что REINDEX – блокирующая команда. Чтобы обойти эту проблему, можно параллельно (без блокировки) создать индекс, идентичный исходному, а затем удалить исходный. Параллельное создание индекса – предпочтительное решение в активных системах, но оно требует больше ресурсов, чем обычное индексирование.

К тому же при параллельном индексировании можно столкнуться с подводными камнями; иногда создание индексов завершается неудачно, и тогда остается некорректный индекс. Его можно удалить или перестроить, соответственно, командами DROP и REINDEX. Ниже приведена команда (блокирующая) перестраивания индекса account_history_account_id_search_key_search_date_key:

```
car_portal=# REINDEX index
car_portal_app.account_history_account_id_search_key_search_date_key;
REINDEX
```

Другой, неблокирующий способ параллельного создания индекса выглядит так:

```
car_portal=# CREATE UNIQUE INDEX CONCURRENTLY ON
car_portal_app.account_history(account_id, search_key, search_date);
CREATE INDEX
```

Наконец, нужно удалить старый индекс. В данном случае мы не можем воспользоваться командой DROP INDEX, потому что индекс был создан неявно для поддержки ограничения уникальности. Чтобы избавиться от него, нужно удалить ограничение:

```
car_portal=# ALTER TABLE car_portal_app.account_history DROP CONSTRAINT
account_history_account_id_search_key_search_date_key;
ALTER TABLE
```

А затем снова добавить его:

```
car_portal=> ALTER TABLE account_history ADD CONSTRAINT
account_history_account_id_search_key_search_date_key UNIQUE USING INDEX
account_history_account_id_search_key_search_date_idx;
NOTICE: ALTER TABLE / ADD CONSTRAINT USING INDEX will rename index
"account_history_account_id_search_key_search_date_idx" to
"account_history_account_id_search_key_search_date_key"
ALTER TABLE
```

Функции

Функции в PostgreSQL решают конкретную задачу и обычно состоят из объявлений, выражений и команд. PostgreSQL предлагает широчайший спектр встроенных функций почти для всех существующих типов. В этой главе мы займемся пользовательскими функциями, а подробно о синтаксисе и параметрах функций поговорим в последующих главах.

Встроенные языки программирования PostgreSQL

PostgreSQL безо всяких расширений поддерживает написание пользовательских функций на языках C, SQL и PL/pgSQL. Еще три процедурных языка – PL/Tcl, PL/Python и PL/Perl – входят в стандартный дистрибутив, но их надо явно добавить командой `CREATE EXTENSION` или с помощью утилиты `createlang`. Чтобы добавить язык и сделать его доступным во всех базах данных, проще всего создать его в базе данных `template1` сразу после установки кластера. Для C, SQL и PL/pgSQL этого делать не надо.

Начинающим проще всего пользоваться языками SQL и PL/pgSQL, поскольку они поддерживаются непосредственно. К тому же они хорошо переносимы и не нуждаются в специальном внимании в процессе перевода кластера на новую версию. Написать функцию на C не так просто, как на SQL или PL/pgSQL, но поскольку C – универсальный язык программирования, на нем можно создавать очень сложные функции для работы со сложными типами данных, например изображениями.

Создание функции на языке C

В примере ниже на C написана функция факториал. Этот пример можно взять за образец при создании более сложных функций. Процедура создания функции на C состоит из четырех шагов:

1) установить библиотеку `postgresql-server-development`;
2) написать функцию, создать make-файл и откомпилировать ее в виде разделяемой библиотеки (с расширением .so);
3) задать местоположение разделяемой библиотеки. Проще всего указать абсолютный путь при создании функции или скопировать файл библиотеки в каталог библиотек PostgreSQL;
4) создать функцию в базе данных командой `CREATE FUNCTION`.

Для установки библиотеки разработки PostgreSQL можно воспользоваться программой `apt`:

```
sudo apt-get install postgresql-server-dev-10
```

Для компиляции кода на C обычно используется программа `make`. Ниже приведен простой файл `makefile` для компиляции функции факториал. Для получения информации об установленной версии PostgreSQL используется утилита `pg_config`:

```
MODULES = fact
PG_CONFIG = pg_config
PGXS = $(shell $(PG_CONFIG) --pgxs)
INCLUDEDIR = $(shell $(PG_CONFIG) --includedir-server)
include $(PGXS)

fact.so: fact.o
    cc -shared -o fact.so fact.o

fact.o: fact.c
    cc -o fact.o -c fact.c $(CFLAGS) -I$(INCLUDEDIR)
```

Исходный код функции fact приведен ниже:

```
#include "postgres.h"
#include "fmgr.h"
#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

Datum fact(PG_FUNCTION_ARGS);
PG_FUNCTION_INFO_V1(fact);

Datum
fact(PG_FUNCTION_ARGS) {
int32 fact = PG_GETARG_INT32(0);
int32 count = 1, result = 1;
for (count = 1; count <= fact; count++)
    result = result * count;
PG_RETURN_INT32(result);
}
```

Осталось откомпилировать код, скопировать библиотеку в каталог PostgreSQL и создать функцию от имени postgres или обычного пользователя. Компиляция производится так:

```
$ make -f makefile
cc -o fact.o -c fact.c -Wall -Wmissing-prototypes -Wpointer-arith
-Wdeclaration-after-statement -Wendif-labels -Wmissing-format-attribute
-Wformat-security -fno-strict-aliasing -fwrapv -fexcess-precision=standard
-g -g -O2 -fstack-protector-strong -Wformat -Werror=format-security –fPIC
-pie -fno-omit-frame-pointer -fPIC -I/usr/include/postgresql/10/server
cc -shared -o fact.so fact.o
```

Для копирования файла нужны права суперпользователя:

```
$sudo cp fact.so $(pg_config --pkglibdir)/
```

Наконец, от имени пользователя postgres создадим функцию в базе данных template1 и протестируем ее:

```
$ psql -d template1 -c "CREATE FUNCTION fact(INTEGER) RETURNS INTEGER AS
'fact', 'fact' LANGUAGE C STRICT;"
CREATE FUNCTION
$ psql -d template1 -c "SELECT fact(5);"
 fact
------
  120
(1 row)
```

Написание функций на C выглядит очень сложно, по сравнению с написанием на SQL и PL/pgSQL. Кроме того, при плохо поставленном обслуживании могут возникнуть проблемы при переходе на новую версию СУБД.

```
CREATE OR REPLACE FUNCTION is_updatable_view (text) RETURNS BOOLEAN AS
$$
    SELECT is_insertable_into='YES' FROM information_schema.tables
```

```
WHERE table_type = 'VIEW' AND table_name = $1
$$ LANGUAGE SQL;
```

Тело SQL-функции может состоять из нескольких SQL-команд; результат последней команды определяет тип возвращаемого значения. SQL-функция не годится для конструирования динамических SQL-команд, поскольку вместо ее аргумента могут подставляться только данные, но не идентификаторы. Следующий фрагмент не является допустимой SQL-функцией:

```
CREATE FUNCTION drop_table (text) RETURNS VOID AS
$$
    DROP TABLE $1;
$$ LANGUAGE SQL;
```

Язык PL/pgSQL обладает более развитыми возможностями, и в повседневной работе лучше пользоваться им. Он может содержать объявления переменных, условные команды и циклы, перехват исключений и т. д. Следующая функция возвращает факториал целого числа:

```
CREATE OR REPLACE FUNCTION fact(fact INT) RETURNS INT AS
$$
DECLARE
    count INT = 1;
    result INT = 1;
BEGIN
    FOR count IN 1..fact LOOP
        result = result* count;
    END LOOP;
    RETURN result;
END;
$$ LANGUAGE plpgsql;
```

Применение функций

В PostgreSQL функции применяются для разных целей. Некоторые разработчики используют их как абстрактный интерфейс с языками программирования, чтобы скрыть модель данных. Но есть и другие применения:

- реализация сложной логики, для которой средств SQL недостаточно;
- осуществление действий до или после выполнения SQL-команды с помощью триггеров;
- вычищение SQL-кода путем перемещения часто используемых частей в модули;
- автоматизация типичных задач с помощью динамического SQL.

Зависимости между функциями

Зависимости между функциями не очень хорошо отслеживаются в системном каталоге PostgreSQL, поэтому при работе с ними нужно быть настороже, чтобы не образовались висячие функции. Следующий пример демонстрирует, как такое может случиться:

```
CREATE OR REPLACE FUNCTION test_dep (INT) RETURNS INT AS $$
BEGIN
        RETURN $1;
END;
$$
LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION test_dep_2(INT) RETURNS INT AS
$$
BEGIN
    RETURN test_dep($1);
END;
$$
LANGUAGE plpgsql;
DROP FUNCTION test_dep(int);
```

Теперь функция `test_dep_2` висячая. При попытке ее вызвать возникнет ошибка:

```
SELECT test_dep_2 (5);
ERROR: function test_dep(integer) does not exist
LINE 1: SELECT test_dep($1)
               ^
HINT: No function matches the given name and argument types. You might
need to add explicit type casts.
QUERY: SELECT test_dep($1)
CONTEXT: PL/pgSQL function test_dep_2(integer) line 3 at RETURN
```

Категории функций в PostgreSQL

Новая функция по умолчанию помечается волатильной, если явно не указано противное. Если функция не волатильная, то лучше пометить ее как стабильную или неизменяемую, поскольку это помогает планировщику генерировать оптимальные планы выполнения. В PostgreSQL есть три категории волатильности:

- **volatile:** волатильная функция может возвращать различные результаты, даже если ее аргументы не изменяются, либо изменять данные в базе. Примером может служить функция `random()`;
- **stable и immutable:** такие функции не изменяют базу данных и гарантированно возвращают один и тот же результат при вызовах с одинаковыми аргументами. Стабильная функция, помеченная ключевым словом `stable`, дает такую гарантию в контексте команды, а неизменяемая функция с ключевым словом `immutable` – глобально, вне всякого контекста.

Например, функция `random()` волатильная, т. к. дает разные результаты при каждом вызове. Функция `round()` неизменяемая, потому что при неизменном аргументе всегда возвращает неизменный результат. Функция `now()` стабильная, т. к. в контексте команды или транзакции возвращает один и тот же результат, как видно из примера ниже:

```
postgres=# BEGIN;
BEGIN
postgres=# SELECT now();
              now
-------------------------------
 2017-10-30 22:00:04.723596+01
(1 row)

postgres=# SELECT 'Some time has passed', now();
 ?column?             | now
----------------------+-------------------------------
 Some time has passed | 2017-10-30 22:00:04.723596+01
(1 row)
```

Анонимные функции в PostgreSQL

В PostgreSQL имеется команда DO, позволяющая выполнять анонимные блоки кода. Она уменьшает потребность в написании скриптов оболочки для административных задач. Отметим, однако, что функции в PostgreSQL транзакционные, поэтому если, например, в функции нужно создать индекс, то скрипт оболочки предпочтительнее. Предположим, что в базе для сайта торговли автомобилями понадобился пользователь, которому разрешено выполнять только команды SELECT. Создадим роль:

```
CREATE user select_only;
```

Теперь нужно выдать этой роли разрешение SELECT для каждой таблицы. Это можно сделать, воспользовавшись динамическим SQL:

```
DO $$
    DECLARE r record;
BEGIN
    FOR r IN
        SELECT table_schema, table_name
        FROM information_schema.tables
        WHERE table_schema = 'car_portal_app'
    LOOP
        EXECUTE 'GRANT SELECT ON ' || quote_ident(r.table_schema) || '.'||
quote_ident(r.table_name) || ' TO select_only';
    END LOOP;
END$$;
```

Пользовательские типы данных

В PostgreSQL есть два способа определить пользовательский тип данных:

- **команда CREATE DOMAIN:** позволяет создавать пользовательские типы данных с ограничениями с целью сделать исходный код более модульным;
- **команда CREATE TYPE:** часто используется для создания составного типа, что полезно в процедурных языках, где такие типы служат для воз-

врата значений. Эта же команда может создавать тип ENUM, позволяющий уменьшить количество соединений со справочными таблицами.

Разработчики часто используют плоские таблицы вместо пользовательских типов из-за отсутствия поддержки в драйверах, в т. ч. JDBC и ODBC. Впрочем, в JDBC составные типы данных можно получить в виде объектов Java и разобрать вручную.

Объекты доменов, как и другие объекты базы данных, должны иметь уникальные имена в пределах схемы. Основное применение доменов – в качестве образцов. Рассмотрим, к примеру, текстовый тип, который не может принимать значение null и содержать пробелы. Это образец. В базе для сайта торговли автомобилями такими свойствами обладают столбцы first_name и last_name таблицы account, определенные следующим образом:

```
first_name TEXT NOT NULL,
last_name TEXT NOT NULL,
CHECK(first_name !~ '\s' AND last_name !~ '\s'),
```

Вместо ограничений мы можем создать домен

```
CREATE DOMAIN text_without_space_and_null AS TEXT NOT NULL CHECK (value !~ '\s');
```

и использовать его в определении столбцов first_name и last_name.

Чтобы протестировать домен text_without_space_and_null, включим его в определение таблицы:

```
CREATE TABLE test_domain (
test_att text_without_space_and_null
);
```

а затем выполним несколько команд INSERT

```
postgres=# INSERT INTO test_domain values ('hello');
INSERT 0 1
postgres=# INSERT INTO test_domain values ('hello world');
ERROR: value for domain text_without_space_and_null violates check
constraint "text_without_space_and_null_check"
postgres=# INSERT INTO test_domain values (null);
ERROR: domain text_without_space_and_null does not allow null values
```

Еще одно применение домены находят для создания идентификаторов, уникальных в нескольких таблицах, в угоду людям, которые предпочитают запрашивать информацию, указывая не имя, а число. Для этого создадим последовательность и обернем ее доменом:

```
CREATE SEQUENCE global_id_seq;
CREATE DOMAIN global_serial INT DEFAULT NEXTVAL('global_id_seq') NOT NULL;
```

Наконец, мы можем изменить домен командой ALTER DOMAIN. Если в домен добавляется новое ограничение, то все атрибуты, определенные с помощью этого домена, будут проверены на соблюдение этого ограничения. При желании проверку старых значений можно подавить, а затем почистить таблицы

вручную. Допустим, что мы хотим добавить в домен text_without_space_and_null ограничение на длину текста:

```
ALTER DOMAIN text_without_space_and_null ADD CONSTRAINT
text_without_space_and_null_length_chk check (length(value)<=15);
```

Эта команда завершится неудачно, если какой-нибудь атрибут, принадлежащий этому домену, длиннее 15 символов. Если мы хотим все же применять ограничение к новым данным, а старые пока оставить как есть, то это можно сделать:

```
ALTER DOMAIN text_without_space_and_null ADD CONSTRAINT text_without_
space_and_null_length_chk check (length(value)<=15) NOT VALID;
```

После вычистки старых данных можно будет проверить для них ограничение, выполнив команду ALTER DOMAIN ... VALIDATE CONSTRAINT. Наконец, метакоманда psql \dD+ выводит описание домена.

Составные типы данных очень полезны для создания функций, особенно когда возвращаемый тип представляет собой строку из нескольких значений. Допустим, нам нужна функция, возвращающая seller_id, seller_name, количество объявлений и полный ранг продавца. Первым делом создадим новый тип:

```
CREATE TYPE seller_information AS (seller_id INT, seller_name TEXT,
number_of_advertisements BIGINT, total_rank float);
```

Затем воспользуемся новым типом для возврата значения из функции:

```
CREATE OR REPLACE FUNCTION seller_information (account_id INT )
RETURNS seller_information AS
$$
SELECT seller_account.seller_account_id, first_name || last_name as
seller_name, count(*), sum(rank)::float/count(*)
FROM account
INNER JOIN
     seller_account ON account.account_id = seller_account.account_id
LEFT JOIN
     advertisement ON advertisement.seller_account_id =
     seller_account.seller_account_id
LEFT JOIN
     advertisement_rating ON advertisement.advertisement_id =
     advertisement_rating.advertisement_id
WHERE account.account_id = $1
GROUP BY seller_account.seller_account_id, first_name, last_name
$$
LANGUAGE SQL;
```

Команду CREATE TYPE можно использовать также для определения перечислений enum; атрибут такого типа может принимать только одно из заранее заданных значений. Благодаря перечислениям удается уменьшить количество соединений в некоторых запросах, так что SQL-код становится компактнее и проще для понимания. В таблице advertisement_rating имеется столбец rank, определенный следующим образом:

```
-- Это часть определения таблицы advertisement_rating.
rank INT NOT NULL,
CHECK (rank IN (1,2,3,4,5)),
```

Смысл этого кода не очевиден. Одни могут счесть, что наивысший ранг равен 1, другие – что 5. Для устранения разночтений можно создать справочную таблицу:

```
CREATE TABLE rank (
    rank_id SERIAL PRIMARY KEY,
    rank_name TEXT NOT NULL
);
INSERT INTO rank VALUES (1, 'poor') , (2, 'fair'), (3, 'good') ,
                        (4,'very good') , ( 5, 'excellent');
```

При таком подходе смысл рангов сразу понятен. Можно даже изменять таблицу в соответствии с новыми бизнес-требованиями, например ввести 10-балльную шкалу. Кроме того, изменение записей в таблице рангов не потребует блокировать таблицу `advertisement_rating`, потому что нам не понадобится менять ограничение `CHECK (rank IN (1, 2, 3, 4, 5))` командой `ALTER TABLE`. Но у этого подхода есть и недостаток – для получения информации о семантике `rank_id` придется соединять таблицы `advertisement_rating` и `rank`. Чем больше подобных справочных таблиц, чем длиннее запрос.

Альтернативный подход к моделированию ранга состоит в использовании типа перечисления:

```
CREATE TYPE rank AS ENUM ('poor', 'fair', 'good', 'very good', 'excellent');
```

Метакоманда `psql \dT` выводит описание типа перечисления. Можно также воспользоваться функцией `enum_range`:

```
postgres=# SELECT enum_range(null::rank);
 enum_range
---------------------------------------
 {poor,fair,good,"very good",excellent}
```

Порядок значений типа `enum` определяет порядок их перечисления в момент создания.

```
postgres=# SELECT unnest(enum_range(null::rank)) order by 1 desc;
 unnest
-----------
 excellent
 very good
 good
 fair
 poor
(5 rows)
```

Перечисления в PostgreSQL типов безопасны, и, в частности, значения различных типов `enum` нельзя сравнивать между собой. Перечисления можно изменять, добавляя новые значения.

Триггеры и правила

PostgreSQL предоставляет триггеры и правила, позволяющие автоматически выполнять некоторые действия при возникновении событий, например выполнении команд INSERT, UPDATE или DELETE. Для команды SELECT триггеры и правила не определяются, за исключением правила _RETURN, которое используется во внутренней реализации представлений.

С точки зрения функциональности триггеры обладают большей общностью, чем правила, и позволяют проще реализовать сложные действия. Но в ряде случаев необходимую функциональность можно получить с помощью как триггеров, так и правил. По производительности правила быстрее, зато триггеры проще и совместимы с другими РСУБД (правила – специфическое расширение PostgreSQL).

Правила

Создание правила означает либо модификацию правила по умолчанию, либо создание нового правила для определенного действия с определенной таблицей или представлением. Иными словами, правило для действия при вставке может либо изменить поведение вставки, либо создать для нее новое действие. Следует иметь в виду, что вся система правил основана на макросах языка C. Поэтому можно получить странные результаты при использовании правил совместно с волатильными функциями типа random() и функциями последовательностей типа nextval(). В следующем примере демонстрируются подвохи правил. Допустим, требуется организовать аудит действий с таблицей car. Для этого создадим новую таблицу car_log, в которой будут храниться все действия с таблицей car: обновления, удаления и вставки. Сделать это можно с помощью правил:

```
CREATE TABLE car_log (LIKE car);
ALTER TABLE car_log
   ADD COLUMN car_log_action varchar (1) NOT NULL,
   ADD COLUMN car_log_time TIMESTAMP WITH TIME ZONE NOT NULL;

CREATE RULE car_log AS ON INSERT TO car DO ALSO
   INSERT INTO car_log (car_id, car_model_id, number_of_owners,
        registration_number, number_of_doors, manufacture_year,car_log_action,
        car_log_time)
   VALUES (new.car_id, new.car_model_id,new.number_of_owners,
        new.registration_number, new.number_of_doors, new.manufacture_year,'I',
        now());
```

Здесь мы создаем таблицу car_log, структурно аналогичную таблице car, но имеющую два дополнительных атрибута для протоколирования действий: тип операции (в примере 'I' означает вставку) и время действия. Мы также создаем правило для таблицы car, сводящееся к вставке в car_log копии записи, вставленной в car. Для тестирования вставим одну запись:

```
INSERT INTO car (car_id, car_model_id, number_of_owners,
  registration_number, number_of_doors, manufacture_year) VALUES (100000, 2,
  2, 'x', 3, 2017);
```

Проверим содержимое таблицы car_log:

```
car_portal=> SELECT to_json(car) FROM car where registration_number ='x';
to_json
------------------------------------------------------------------------
------------------------------------------------------------------------
{"car_id":100000,"number_of_owners":2,"registration_number":"x","manufactur
e_year":2017,"number_of_doors":3,"car_model_id":2,"mileage":null}
(1 row)
car_portal=> SELECT to_json(car_log) FROM car_log where registration_number
='x';
to_json
------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------
{"car_id":100000,"number_of_owners":2,"registration_number":"x","manufactur
e_year":2017,"number_of_doors":3,"car_model_id":2,"mileage":null,"car_log_a
ction":"I","car_log_time":"2017-11-20T20:08:50.107895+01:00"}
(1 row)
```

Как видим, все идет по плану. В таблицу car вставлена одна запись, и точно такая же запись вставлена в таблицу car_log. Но, как уже отмечалось, система правил основана на макросах, и это может приводить к некоторым проблемам. Одна из них возникает, если мы захотим использовать значение по умолчанию:

```
INSERT INTO car (car_id, car_model_id, number_of_owners,
registration_number, number_of_doors, manufacture_year)
VALUES (default, 2, 2, 'y', 3, 2017);
```

Посмотрим, как изменилось содержимое обеих таблиц:

```
car_portal=> SELECT to_json(car) FROM car where registration_number ='y';
to_json
------------------------------------------------------------------------
------------------------------------------------------------
{"car_id":230,"number_of_owners":2,"registration_number":"y","manufacture_y
ear":2017,"number_of_doors":3,"car_model_id":2,"mileage":null}
(1 row)

car_portal=> SELECT to_json(car_log) FROM car_log where registration_number
='y';
to_json
------------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------
{"car_id":231,"number_of_owners":2,"registration_number":"y","manufacture_y
ear":2017,"number_of_doors":3,"car_model_id":2,"mileage":null,"car_log_acti
on":"I","car_log_time":"2017-11-20T20:14:49.820841+01:00"}
(1 row)
```

Обратите внимание, что значение столбца car_id в таблице car на единицу меньше, чем в car_log. Ключевое слово DEFAULT означает, что car_id нужно присвоить значение по умолчанию, которое в данном случае равно nextval('car_car_id_seq'::regclass).

Правила могут быть условными, т. е. действие переопределяется при выполнении некоторого условия. Однако невозможно задать условные правила для команд INSERT, UPDATE и DELETE, применяемых к представлениям, если отсутствует безусловное правило. Для решения этой проблемы можно создать фиктивное безусловное правило. Правила над представлениями – один из способов реализовать обновляемые представления.

Триггеры

PostgreSQL запускает триггерную функцию, когда возникает некоторое событие в таблице, представлении или внешней таблице. Триггер выполняется, когда пользователь пытается модифицировать данные с помощью любой команды языка манипулирования данными (DML), а именно INSERT, UPDATE, DELETE или TRUNCATE. Ниже приведен синтаксис команды создания триггера:

```
CREATE [ CONSTRAINT ] TRIGGER name { BEFORE | AFTER | INSTEAD OF } { event
[ OR ... ] }
    ON table_name
    [ FROM referenced_table_name ]
    [ NOT DEFERRABLE | [ DEFERRABLE ] [ INITIALLY IMMEDIATE | INITIALLY DEFERRED ] ]
    [ REFERENCING { { OLD | NEW } TABLE [ AS ] transition_relation_name } [ ... ] ]
    [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( condition ) ]
    EXECUTE PROCEDURE function_name ( arguments )
```

где event – одно из событий:

```
    INSERT
    UPDATE [ OF column_name [, ... ] ]
    DELETE
    TRUNCATE
```

Существует три временных контекста триггера.

- BEFORE: применяется только к таблицам. Триггер запускается перед проверкой ограничений и выполнением операции. Полезно для проверки ограничений, охватывающих несколько таблиц, которые невозможно смоделировать как ограничения ссылочной целостности.
- AFTER: также применяется только к таблицам. Запускается после выполнения операции. Полезно для каскадного применения изменений к другим таблицам. Примером может служить аудит данных.
- INSTEAD OF: применяется к представлениям, чтобы сделать их обновляемыми.

Если в команде создания триггера присутствует фраза FOR EACH ROW, то триггерная функция вызывается для каждой строки, затронутой операцией. Триггер с фразой FOR EACH STATEMENT выполняется один раз для операции. Если присут-

ствует условие WHEN, то триггер применяется только к строкам, удовлетворяющим условию.

Наконец, триггер может быть помечен ключевым словом CONSTRAINT, чтобы указать, в какой момент он выполняется: в конце команды или в конце транзакции. Триггер-ограничение может быть только типа AFTER или FOR EACH ROW, а время его срабатывания определяется следующими ключевыми словами:

- DEFERRABLE: срабатывание триггера откладывается до конца транзакции;
- INITIALLY DEFERRED: определяет время выполнения триггера и означает, что триггер выполняется в конце транзакции. Триггер должен быть определен как DEFERRABLE;
- NOT DEFERRABLE: это поведение по умолчанию – триггер срабатывает после каждой команды;
- INITIALLY IMMEDIATE: определяет время выполнения триггера и означает, что триггер выполняется после каждой команды. Триггер должен быть определен как DEFERRABLE.

Имя триггера обычно отражает временной контекст его выполнения.

Опции времени срабатывания – DEFERRABLE, INITIALLY DEFERRED, NOT DEFERRABLE, INITIALLY IMMEDIATE – применимы и к триггерам-ограничениям. Они очень полезны в случаях, когда PostgreSQL взаимодействует с внешними системами, например memcached. Предположим, к примеру, что над таблицей определен триггер, и эта таблица кешируется: при каждом обновлении таблицы обновляется также и кеш. Поскольку система кеширования не транзакционная, мы можем отложить обновление до момента завершения транзакции, когда будет гарантирована согласованность данных.

Чтобы лучше понять, как работают триггеры, применим их для ведения таблицы car_log. Сразу отметим, что триггер должен иметь тип AFTER, потому что перед тем как вставлять данные в новую таблицу, их нужно проверить на соблюдение ограничений. Прежде чем создавать триггер, нужно создать функцию:

```
CREATE OR REPLACE FUNCTION car_log_trg () RETURNS TRIGGER AS
$$
BEGIN
IF TG_OP = 'INSERT' THEN
    INSERT INTO car_log SELECT NEW.*, 'I', NOW();
ELSIF TG_OP = 'UPDATE' THEN
    INSERT INTO car_log SELECT NEW.*, 'U', NOW();
ELSIF TG_OP = 'DELETE' THEN
    INSERT INTO car_log SELECT OLD.*, 'D', NOW();
END IF;
RETURN NULL; -- игнорируется, потому что это триггер типа AFTER
END;
$$
LANGUAGE plpgsql;
```

Создадим триггер командой:

```
CREATE TRIGGER car_log AFTER INSERT OR UPDATE OR DELETE ON car FOR EACH ROW
EXECUTE PROCEDURE car_log_trg ();
```

Триггерная функция должна удовлетворять следующим требованиям:

- **тип возвращаемого значения:** функция должна возвращать псевдотип TRIGGER;
- **возвращаемое значение:** функция должна возвращать значение. Для триггеров типа AFTER ... EACH ROW оно часто равно NULL, а для триггеров уровня команды – строка, содержащая точную структуру таблицы, для которой сработал триггер;
- **отсутствие аргументов:** в объявлении триггерной функции не должно быть аргументов. Аргументы, если они необходимы, передаются с помощью переменной TG_ARG. При вызове триггерной функции автоматически создается несколько переменных, в т. ч. TG_ARG и NEW. Все переменные перечислены в таблице ниже.

Триггерная переменная	Тип данных	Описание
NEW	RECORD	Вставленная или обновленная строка. В случае триггера уровня команды NULL
OLD	RECORD	Строка до обновления или удаления. В случае триггера уровня команды NULL
TG_NAME	NAME	Имя триггера
TG_OP	NAME	Операция: INSERT, UPDATE, DELETE, TRUNCATE
TG_WHEN	NAME	Момент срабатывания триггера: AFTER или BEFORE
TG_RELID	OID	OID отношения. Чтобы получить имя отношения, нужно привести к типу regclass::text
TG_TABLE_NAME	NAME	Имя таблицы, над которой определен триггер
TG_TABLE_SCHEMA	NAME	Имя схемы таблицы, над которой определен триггер
TG_ARG[]	Массив типа TEXT	Аргумент триггерной функции. Индексирование начинается с нуля, при указании недопустимого индекса возвращается NULL
TG_NARG	INTEGER	Количество аргументов, переданных триггерной функции

Если триггер уровня строки, срабатывающий перед операцией, возвращает значение null, то операция отменяется. Это означает, что следующий триггер не сработает, а соответствующая строка не будет удалена, обновлена или вставлена. Для триггеров, срабатывающих после операции, и триггеров уровня команды возвращенное значение игнорируется, однако операция отменяется, если триггер возбуждает исключение, – это следствие транзакционной природы базы данных.

Если в примере с аудитом таблицы car изменить определение этой таблицы, например добавить или удалить столбец, то триггерная функция над таблицей car завершится ошибкой, поэтому попытка вставки или обновления строки игнорируется. Проблему можно решить, перехватив исключение внутри функции.

Триггеры с аргументами

В следующем примере мы рассмотрим еще одну общую технику аудита. Она применима к нескольким таблицам, причем необязательно аудировать все столбцы. Нам понадобится расширение `hstore`, в котором определен тип хеша, а также функции и операторы для работы с ним. Старую и новую строки таблицы мы будем хранить в виде хешей. Прежде всего создадим расширение `hstore` и таблицу для хранения данных аудита:

```
SET search_path to car_portal_app;
CREATE extension hstore;
CREATE TABLE car_portal_app.log
(
    schema_name text NOT NULL,
    table_name text NOT NULL,
    old_row hstore,
    new_row hstore,
    action TEXT check (action IN ('I','U','D')) NOT NULL,
    created_by text NOT NULL,
    created_on timestamp without time zone NOT NULL
);
```

Далее определим триггерную функцию:

```
CREATE OR REPLACE FUNCTION car_portal_app.log_audit() RETURNS trigger AS
$$
DECLARE
    log_row log;
    excluded_columns text[] = NULL;
BEGIN
    log_row = ROW (TG_TABLE_SCHEMA::text,
        TG_TABLE_NAME::text,NULL,NULL,NULL,current_user::TEXT,
        current_timestamp);
    IF TG_ARGV[0] IS NOT NULL THEN excluded_columns = TG_ARGV[0]::text[]; END IF;
    IF (TG_OP = 'INSERT') THEN
        log_row.new_row = hstore(NEW.*) - excluded_columns;
        log_row.action ='I';
    ELSIF (TG_OP = 'UPDATE' AND (hstore(OLD.*) - excluded_columns !=
           hstore(NEW.*)-excluded_columns)) THEN
        log_row.old_row = hstor(OLD.*) - excluded_columns;
        log_row.new_row = hstore(NEW.* )- excluded_columns;
        log_row.action ='U';
    ELSIF (TG_OP = 'DELETE') THEN
        log_row.old_row = hstore (OLD.*) - excluded_columns;
        log_row.action ='D';
    ELSE
        RETURN NULL; -- обновить исключенные столбцы
    END IF;
    INSERT INTO log SELECT log_row.*;
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

В этой функции определена переменная log_row типа log. С помощью конструктора строки ROW в нее записываются имя триггерной таблицы, имя схемы триггерной таблицы, текущий пользователь и текущая временная метка. Далее функция извлекает из переменной имена столбцов, которые не нужно аудировать. Обратите внимание, что имена столбцов передаются в виде массива элементов типа text. И напоследок триггерная функция заполняет поля log.action, log.old_row и log.new_row в зависимости от значения переменной TG_OP.

Чтобы применить эту триггерную функцию к таблице car в предположении, что атрибут number_of_doors аудировать не нужно, создадим триггер следующим образом:

```
CREATE TRIGGER car_log_trg AFTER INSERT OR UPDATE OR DELETE ON car_portal_app.car
FOR EACH ROW EXECUTE PROCEDURE log_audit('{number_of_doors}');
```

Литеральный массив {number_of_doors} передается функции log_audit, где к нему можно обратиться через переменную TG_ARG. Внутри функции выражение hstore(NEW.*) - excluded_columns используется для того, чтобы преобразовать переменную NEW в значение типа hstore и удалить из полученного хеша ключи, определенные в массиве excluded_columns. Ниже продемонстрировано поведение триггера при выполнении команды INSERT:

```
car_portal=# INSERT INTO car (car_id, car_model_id, number_of_owners,
registration_number, number_of_doors, manufacture_year)
VALUES (default, 2, 2, 'z', 3, 2017);
INSERT 0 1
```

Чтобы показать результат, выберем содержимое таблицы log в формате JSON:

```
car_portal=# SELECT jsonb_pretty((to_json(log))::jsonb) FROM
car_portal_app.log WHERE action = 'I' and
new_row->'registration_number'='z';
                    jsonb_pretty
-------------------------------------------------
 { +
       "action": "I", +
       "new_row": { +
           "car_id": "235", +
           "mileage": null, +
           "car_model_id": "2", +
           "manufacture_year": "2017", +
           "number_of_owners": "2", +
           "registration_number": "z" +
       }, +
       "old_row": null, +
       "created_by": "postgres", +
       "created_on": "2017-11-20T20:42:41.132194",+
       "table_name": "car", +
       "schema_name": "car_portal_app" +
}
(1 row)
```

Триггеры и обновляемые представления

Представления, которые не являются автоматически обновляемыми, можно сделать обновляемыми с помощью триггеров. Рассмотрим представление seller_account_information, дающее информацию об учетной записи продавца:

```
CREATE OR REPLACE VIEW seller_account_info AS SELECT account.account_id,
first_name, last_name, email, password, seller_account_id, total_rank,
number_of_advertisement, street_name, street_number, zip_code , city
FROM account INNER JOIN
seller_account ON (account.account_id = seller_account.account_id);
```

Убедимся, что оно не автообновляемое:

```
car_portal=# SELECT is_insertable_into FROM information_schema.tables WHERE
table_name = 'seller_account_info';
 is_insertable_into
--------------------
 NO
(1 row)
```

В показанной ниже триггерной функции предполагается, что столбцы account_id и seller_account_id генерируются автоматически с помощью последовательностей, поскольку имеют тип serial. Такой подход позволяет не проверять ограничение уникальности при вставке новых строк, и к тому же во множестве значений первичных ключей не будет больших разрывов. По той же причине предполагается, что первичные ключи не изменяются. Изменение первичного ключа может вызывать проблемы, если только не используются внешние ключи с каскадным удалением и обновлением.

Наконец, отметим, что триггерная функция возвращает NEW для операций INSERT и UPDATE, OLD – для операции DELETE и NULL – в случае исключения. Важно возвращать правильное значение, чтобы сервер мог подсчитать количество строк, затронутых операцией. Также очень важно, какое значение возвращается в случае вставки новой строки. Благодаря ключевому слову RETURNING мы знаем, какое значение получили идентификаторы NEW.account_id и NEW.seller_account_id после вставки. Если возвращаются неправильные идентификаторы, то могут возникнуть трудные для отладки ошибки в средствах объектно-реляционного отображения типа Hibernate.

```
CREATE OR REPLACE FUNCTION seller_account_info_update () RETURNS TRIGGER AS
$$
DECLARE
    acc_id INT;
    seller_acc_id INT;
BEGIN
    IF (TG_OP = 'INSERT') THEN
        WITH inserted_account AS (
          INSERT INTO car_portal_app.account (account_id, first_name,
              last_name, password, email) VALUES (DEFAULT, NEW.first_name,
              NEW.last_name, NEW.password, NEW.email)
          RETURNING account_id
```

```
        ),
        inserted_seller_account AS (
          INSERT INTO car_portal_app.seller_account(seller_account_id,
              account_id, total_rank, number_of_advertisement, street_name,
              street_number, zip_code, city)
          SELECT
            nextval('car_portal_app.seller_account_seller_account_id_seq'::regclass),
            account_id, NEW.total_rank, NEW.number_of_advertisement, NEW.street_name,
            NEW.street_number, NEW.zip_code, NEW.city
          FROM inserted_account
          RETURNING account_id, seller_account_id
        )
        SELECT account_id, seller_account_id INTO acc_id, seller_acc_id
        FROM inserted_seller_account;
        NEW.account_id = acc_id;
        NEW.seller_account_id = seller_acc_id;
        RETURN NEW;
    ELSIF (TG_OP = 'UPDATE' AND OLD.account_id = NEW.account_id AND
            OLD.seller_account_id = NEW.seller_account_id) THEN
        UPDATE car_portal_app.account SET first_name = new.first_name,
            last_name = new.last_name, password= new.password, email = new.email
        WHERE account_id = new.account_id;
        UPDATE car_portal_app.seller_account SET total_rank = NEW.total_rank,
            number_of_advertisement= NEW.number_of_advertisement, street_name=
            NEW.street_name, street_number = NEW.street_number, zip_code =
            NEW.zip_code, city = NEW.city
        WHERE seller_account_id = NEW.seller_account_id;
        RETURN NEW;
    ELSIF (TG_OP = 'DELETE') THEN
        DELETE FROM car_portal_app.seller_account
        WHERE seller_account_id = OLD.seller_account_id;
        DELETE FROM car_portal_app.account
        WHERE account_id = OLD.account_id;
        RETURN OLD;
    ELSE
        RAISE EXCEPTION 'An error occurred for % operation', TG_OP;
        RETURN NULL;
    END IF;
END;
$$ LANGUAGE plpgsql;
```

Чтобы протестировать триггерную функцию, сначала создадим триггер:

```
CREATE TRIGGER seller_account_info_trg INSTEAD OF INSERT OR UPDATE OR
DELETE ON car_portal_app.seller_account_info
FOR EACH ROW EXECUTE PROCEDURE seller_account_info_update ();
```

Для тестирования вставки в представление выполним следующий код:

```
car_portal=# INSERT INTO car_portal_app.seller_account_info
(first_name,last_name, password, email, total_rank,
number_of_advertisement, street_name, street_number, zip_code, city) VALUES
('test_first_name', 'test_last_name', 'test_password',
```

```
'test_email@test.com', NULL, 0, 'test_street_name', 'test_street_number',
'test_zip_code','test_city') RETURNING account_id, seller_account_id;
 account_id | seller_account_id
------------+-------------------
        482 |               147
(1 row)
```

Обратите внимание на возвращенные значения – пользователю правильно возвращены частичные ключи. Для тестирования DELETE и UPDATE достаточно выполнить такой код:

```
car_portal=# UPDATE car_portal_app.seller_account_info set email =
'test2@test.com' WHERE seller_account_id=147 RETURNING seller_account_id;
 seller_account_id
-------------------
               147
(1 row)

UPDATE 1
car_portal=# DELETE FROM car_portal_app.seller_account_info WHERE
seller_account_id=147;
DELETE 1
```

Наконец, если мы попробуем удалить все учетные записи продавцов, то возникнет ошибка из-за нарушения ограничения ссылочной целостности:

```
car_portal=# DELETE FROM car_portal_app.seller_account_info;
ERROR: update or delete on table "seller_account" violates foreign key
constraint "advertisement_seller_account_id_fkey" on table "advertisement"
DETAIL: Key (seller_account_id)=(57) is still referenced from table
"advertisement".
CONTEXT: SQL statement "DELETE FROM car_portal_app.seller_account WHERE
seller_account_id = OLD.seller_account_id"
PL/pgSQL function seller_account_info_update() line 21 at SQL statement
car_portal=#
```

Резюме

В этой главе мы обсудили индексы, представления, функции, пользовательские типы данных, а также правила и триггеры. Представление – это именованный запрос, или обертка вокруг команды SELECT. Их можно использовать в качестве уровня доступа к данным, как уровень абстракции или для управления разрешениями.

В PostgreSQL выделяются временные, материализованные, обновляемые и рекурсивные представления. Простые представления автоматически являются обновляемыми. Но и сложное представление можно сделать обновляемым, воспользовавшись правилами и триггерами. Индексы – это физические объекты базы данных, они строятся по одному или нескольким столбцам таблицы либо по выражениям. Индексы применяются для повышения производительности и для контроля данных. Существует несколько типов индексов,

в т. ч. B-дерево, хеш, GIN, GIST и BRIN. По умолчанию строится индекс типа B-дерева. GIN и GIST-индексы полезны для индексирования сложных типов данных и для полнотекстового поиска. Есть несколько категорий индексов, позволяющих решать различные задачи. Например, частичные индексы строятся только по подмножеству данных, удовлетворяющих некоторому условию. Уникальные индексы часто применяются для контроля уникальности первичных ключей. Наконец, в некоторых ситуациях для выборки данных полезны многостолбцовые индексы.

Сведения о статистике использования индексов можно получить из таблицы `pg_stat_all_indexes`, они бывают полезны для обслуживания базы данных. Если индекс по какой-то причине разрастается, его можно перестроить параллельно, при этом таблица не блокируется.

Функции PostgreSQL используются для разных целей, в т. ч. сходных с целями представлений. Для написания функций на языках C, SQL и PL/pgsql не нужны дополнительные расширения. Одно из важных применений функций – помощь в обслуживании базы данных. Анонимные функции позволяют сделать многое без написания внешних скриптов, например на языке оболочки bash.

Важно указывать категорию функции – стабильная, неизменяемая или волатильная, – поскольку это помогает сгенерировать оптимальный план выполнения. К сожалению, зависимости между функциями не отслеживаются в каталоге базы данных, поэтому при реализации сложной логики с помощью функций нужно проявлять осторожность. Команды `CREATE DOMAIN` и `CREATE TYPE` служат для создания пользовательских типов данных. Тип `ENUM` уменьшает количество соединений таблиц, а значит, делает SQL-код понятнее и эффективнее. Триггеры и правила применяются в PostgreSQL, когда нужно выполнить действие при возникновении некоторого события. В некоторых ситуациях они взаимозаменяемы. Но при использовании правил совместно с волатильными функциями возможны побочные эффекты.

Глава 5

Язык SQL

Структурированный язык запросов (**SQL**) используется для описания структуры базы данных, манипулирования данными в базе и предъявления запросов. Эта глава посвящена **языку манипулирования данными** (**DML**).

Прочитав данную главу, вы будете понимать концепцию SQL, логику SQL-команд. Вы сможете сами писать запросы и манипулировать данными. Полное справочное руководство по SQL имеется в официальной документации PostgreSQL на странице http://www.postgresql.org/docs/current/static/sql.html.

В этой главе мы рассмотрим следующие темы:

- основы SQL;
- лексическая структура;
- команда SELECT;
- команда UPDATE;
- команда DELETE.

Примеры кода относятся к базе данных для сайта продажи автомобилей, описанной в предыдущих главах. Скрипты создания и заполнения базы демонстрационными данными имеются на сайте, сопровождающем книгу. Он называется `schema.sql` и `data.sql`. Все примеры кода из этой главы находятся в файле `examples.sql`.

О работе с утилитой `psql` см. главу 2.

Основы SQL

Язык SQL служит для манипулирования данными в базе и запросов к ней. Он также используется для определения и изменения структуры базы, т. е. для реализации модели данных. Все это вы уже знаете из предыдущих глав.

SQL состоит из трех частей:

- язык определения данных (DDL);
- язык манипулирования данными (DML);
- язык управления данными (DCL).

Язык DDL нужен для создания и управления структурой данных, язык DML – для управления самими данными, а язык DCL – для управления доступом к данным. Обычно структура данных определяется только один раз и изменя-

ется сравнительно редко. Но вставка, изменение и выборка данных производятся постоянно. Поэтому DML используется чаще, чем DDL.

В отличие от многих других языков, SQL – не императивный язык программирования. Точнее, на нем невозможно подробно описать алгоритм обработки данных. Поэтому может сложиться впечатление, будто мы не способны управлять данными. На императивном языке программист описывает работу с данными на очень детальном уровне: откуда взять данные и как это сделать, как обойти массив записей, когда и как обработать их. Если необходимо обработать данные из нескольких источников, то программист должен реализовать связи между ними на уровне приложения, а не в самой базе.

Но SQL – декларативный язык. Иными словами, чтобы получить тот же результат на других языках, нужно написать целую историю. А на SQL разработчик пишет только предложение, выражающее суть, а детали оставляет базе данных. В SQL-команде определяем формат, в котором мы хотим получить данные, указываем, в каких таблицах данные хранятся, и формулируем правила их обработки. Все необходимые операции, их точный порядок и алгоритмы обработки данных определяет база, а разработчику до этого не должно быть дела.

Однако не стоит считать, что этот черный ящик – обязательно зло. Во-первых, не такой уж он и черный: есть способы узнать, как база обрабатывает данные, и даже повлиять на это. Во-вторых, логика обработки SQL-команды вполне детерминирована. Даже если вы не понимаете, как база данных выполняет запрос на низком уровне, логика обработки и ее результат полностью определены командой.

Этим определяется и размер команды (минимального автономного элемента выполнения). Например, в Java любая операция, в частности присваивание значения переменной, логически обрабатывается как отдельный шаг алгоритма. Напротив, в SQL весь алгоритм выполняется как единое целое – одна команда. Нет никакой возможности получить данные на промежуточном этапе выполнения запроса. Но это ни в коей мере не ограничивает сложность запроса. В одной команде можно реализовать весьма изощренный алгоритм. Как правило, реализация сложной логики на SQL занимает меньше времени, чем на языке более низкого уровня. Разработчик имеет дело с логическими реляционными структурами данных, и ему не нужно изобретать собственные алгоритмы их обработки на физическом уровне. Именно поэтому SQL – такой мощный язык.

Еще одна отличительная особенность SQL – наличие стандарта языка. Это означает, что все современные реляционные базы данных поддерживают SQL. Конечно, в каждой базе есть собственный диалект SQL с дополнительными возможностями, но ядро языка везде одно и то же. И в PostgreSQL реализован собственный диалект SQL, по ходу дела мы будем указывать на отличия от других РСУБД. Кстати, в самом начале пути база postgres не поддерживала SQL. Он был добавлен лишь в 1994 году, и, чтобы отметить этот факт, база была переименована в PostgreSQL.

Лексическая структура SQL

Минимальная единица выполнения в SQL называется **командой**, или запросом. Так, каждое из следующих предложений является командой:

```
SELECT car_id, number_of_doors FROM car_portal_app.car;
DELETE FROM car_portal_app.a;
SELECT now();
```

Команды SQL завершаются точкой с запятой.

Конец ввода также завершает команду, но что это такое, зависит от используемого инструмента. Например, `psql` не станет выполнять команду после нажатия *Enter*, если в конце нет точки с запятой; он просто начнет новую строку. Но если `psql` выполняет SQL-скрипт, прочитанный из файла, то последняя команда всегда выполняется, даже если она не заканчивается точкой с запятой.

Перечислим элементы лексической структуры SQL:

- **ключевые слова** определяют, что должна сделать база данных;
- **идентификаторы** ссылаются на объекты базы данных – таблицы, их столбцы, функции и т. д.;
- **константы** (или **литералы**) – части выражения, значения которых определены непосредственно в коде;
- **операторы** определяют, как следует обрабатывать данные в выражениях;
- специальные символы – круглые и квадратные скобки, запятые и т. д. – имеют особое значение, но операторами не являются;
- **пропуски (**пробелы, знаки табуляции и т. п.) разделяют слова;
- **комментарии** описывают смысл частей кода.

Примерами ключевых слов могут служить `SELECT` и `UPDATE`. Для SQL они наделены специальным смыслом – они определяют назначение команды или ее частей. Полный перечень ключевых слов имеется на странице http://www.postgresql.org/docs/current/static/sql-keywordsappendix.html.

Идентификаторы – это имена объектов базы данных. **Полное имя объекта**, например таблицы или представления, состоит из имени схемы (см. главу 3) и имени самого объекта, разделенных точкой. Если имя схемы входит в параметр `search_path` или объект принадлежит схеме текущего пользователя, то указывать имя схемы необязательно. В таком случае говорят о коротком, или неквалифицированном, имени объекта. Имена столбцов таблицы образуются так же: имя таблицы, точка, имя столбца. Если в запросе фигурирует только одна таблица, то указывать имя таблицы необязательно; в других случаях можно использовать псевдоним таблицы.

Язык SQL нечувствителен к регистру. И ключевые слова, и идентификаторы могут включать буквы (a–z), цифры (0–9), знак подчеркивания (_) и знак доллара ($), но не могут начинаться знаком подчеркивания или доллара. Поэтому, не зная языка, трудно сказать, что является ключевым словом, а что идентификатором. Обычно ключевые слова записывают заглавными буквами.

Идентификаторы могут содержать и символы, отличные от вышеперечисленных, но тогда их нужно заключать в двойные кавычки. В принципе, имя объекта может совпадать с ключевым словом, но поступать так не рекомендуется.

Константы в SQL называют еще литералами. PostgreSQL поддерживает три вида неявно типизированных констант: числа, строки и битовые строки. Если нужна константа другого типа, необходимо выполнить явное или неявное преобразование.

Числовые константы могут содержать цифры и необязательные десятичную точку и показатель степени. Вот несколько примеров допустимых числовых констант:

```
SELECT 1, 1.2, 0.3, .5, 1e15, 12.65e-6;
```

Строковые константы заключаются в кавычки. В PostgreSQL допускается два вида кавычек: одиночные и специфичные для PostgreSQL долларовые (последнее – нестандартная специфика PostgreSQL). Если перед строковой константой поставить букву E, то внутри кавычек можно употреблять экранированные специальные символы, как в языке C: `\n` означает переход на новую строку, а `\t` – табуляцию. Одиночная кавычка внутри литерала должна быть повторена дважды (`''`) или экранирована (`\'`). Наличие буквы U с амперсандом перед строкой позволяет задавать внутри кавычек символы Юникода в виде кода, предваряемого обратной косой чертой.

Ниже приведены примеры строковых констант:

```
car_portal=> SELECT 'a', 'aa''aa', E'aa\naa', $$aa'aa$$,
U&'\041C\0418\0420';
 ?column? | ?column? | ?column? | ?column? | ?column?
----------+----------+----------+----------+----------
 a        | aa'aa    | aa      +| aa'aa    | МИР
          |          | aa       |          |
```

Первый пример простой – одна буква `a`. Во втором имеется одиночная кавычка в середине строки. В третьем внутри строки находится знак новой строки в нотации C: `\n`. Следующая строка заключена в долларовые кавычки. И последняя строка состоит из символов Юникода.

Строковые константы в долларовых кавычках интерпретируются точно так, как написаны. В них не распознаются ни управляющие последовательности, ни какие-либо другие кавычки, кроме долларовых, написанных точно так же, как в начале строки. Между знаками доллара можно задать имя долларовой кавычки, это позволяет вкладывать одну закавыченную строку в другую, например:

```
car_portal=> SELECT $str1$SELECT $$dollar-quoted string$$;$str1$;
 ?column?
---------------------------------
SELECT $$dollar-quoted string$$;
```

Здесь последовательности символов `$str1$` играют роль кавычек, а заключенный внутри них долларовый литерал не завершает строку. Поэтому в долларовые кавычки очень часто заключается тело функции, которое передается серверу PostgreSQL в виде строкового литерала.

Или же можно поставить в начале букву X и включать любые цифры, а также буквы A–F – в таком случае строка записывается в шестнадцатеричной системе счисления. Часто битовые строки преобразуются к числовому виду:

```
car_portal=> SELECT B'01010101'::int, X'AB21'::int;
 int4 |  int4
------+-------
   85 | 43809
```

Операторы играют важную роль в обработке данных и используются в выражениях. Оператор принимает один или два аргумента (операнда) и возвращает значение. PostgreSQL поддерживает широкий спектр операторов для всех типов данных. В командах операторы выглядят как последовательности символов из следующего списка: + - * / < > = ~ ! @ # % ^ & | ` ?.

Если в одном выражении используется несколько операторов, то они выполняются в определенном порядке. У одних операторов, например умножения (*) и деления (/), приоритет выше, а у других, например логических операторов и операторов сравнения, ниже. Операторы с одинаковым приоритетом выполняются слева направо. Краткий перечень операторов с указанием приоритетов приведен на странице https://www.postgresql.org/docs/current/static/sql-syntax-lexical.html#SQLPRECEDENCE.

Далее перечислим специальные символы:

- **круглые скобки** (**()**) используются для задания порядка операций и для группировки выражений. В некоторых SQL-командах имеют специальный смысл. Также употребляются в заголовке функции;
- **квадратные скобки** (**[]**) используются для выбора элементов массива;
- **двоеточие** (:) используется для доступа к части массива;
- **двойное двоеточие** (**::**) используется для приведения типа;
- **запятая** (**,**) используется для разделения элементов списка;
- **точка** (.) используется для разделения имени схемы, имени таблицы и имени столбца;
- **точка с запятой** (**;**) завершает команду;
- **звездочка** (*****) обозначает все столбцы таблицы или все элементы составного значения.

Слова отделяются друг от друга пропусками. В SQL пропуском считается произвольное число пробелов, знаков новой строки и знаков табуляции.

Комментарии могут встречаться в любой части SQL-кода. Сервер игнорирует комментарии, трактуя их как пропуски. Комментарий расположен между парами символов /* и */. Кроме того, комментарием считается часть строки от -- до конца строки.

В языке DML имеется всего четыре команды:

- INSERT – вставка новых данных в базу;
- UPDATE – изменение данных;
- DELETE – удаление данных;
- SELECT – выборка данных.

Команды записываются в понятном человеку виде, структура всех команд строго фиксирована. Все синтаксические диаграммы можно найти в документации PostgreSQL на странице http://www.postgresql.org/docs/current/static/sql-commands.html. Ниже в этой главе мы опишем основные элементы команд.

Первым делом мы рассмотрим команду SELECT, потому что она используется чаще всего и входит составной частью в другие команды. SQL допускает вложенные команды, когда результат одной команды является исходными данными для другой. Такие вложенные запросы называются **подзапросами**.

Запрос данных командой SELECT

Команды, или запросы SELECT (или просто запросы), применяются для выборки данных из базы. Источниками данных для запроса могут быть таблицы, представления, функции или фраза VALUES. Все они являются или могут рассматриваться как отношения. Результатом SELECT также является отношение, которое в общем случае может иметь несколько столбцов и много строк. Поскольку в SQL источник и результат запроса имеют одинаковую природу, результат запроса может выступать в роли источника для другой команды. В таком случае оба запроса рассматриваются как часть большего запроса. Источник данных, формат вывода, фильтрация, группировка, упорядочение и необходимые преобразования данных – всё это задается в коде запроса.

Вообще говоря, запросы SELECT не изменяют данных в базе и могут рассматриваться как операция чтения, но есть одно исключение. Если в запросе встречается волатильная функция, то она может изменить данные.

Структура запроса SELECT

Начнем с простого примера. Мы будем использовать базу данных для сайта торговли автомобилями, описанную в предыдущих главах.

Для подключения к базе выполним команду:

```
> psql -h localhost car_portal
```

В базе имеется таблица car, содержащая сведения о зарегистрированных в системе автомобилях. Пусть требуется получить из базы информацию об автомобилях с тремя дверями, отсортировав ее по идентификатору автомобиля. Результат должен содержать не более 5 строк, поскольку пользовательский интерфейс выводит данные страницами по пять записей. Запрос имеет вид:

```
SELECT car_id, registration_number, manufacture_year
  FROM car_portal_app.car
```

```
  WHERE number_of_doors=3
  ORDER BY car_id
  LIMIT 5;
```

И вот его результат:

```
 car_id | registration_number | manufacture_year
--------+---------------------+-----------------
      2 | VSVW4565            | 2014
      5 | BXGK6290            | 2009
      6 | ORIU9886            | 2007
      7 | TGVF4726            | 2009
      8 | JISW6779            | 2013
(5 rows)
```

Запрос начинается ключевым словом SELECT, которое определяет тип команды. За ним идет список извлекаемых из базы данных полей через запятую. Вместо списка можно поставить знак * – это означает, что нужно выбрать все поля.

Имя таблицы указывается после ключевого слова FROM. Можно выбирать данные сразу из нескольких таблиц. После ключевого слова WHERE указан критерий фильтрации – предикат. После ORDER BY задается порядок сортировки. А ключевое слово LIMIT сообщает базе, что нужно вернуть не более пяти строк.

Эти части запроса – ключевые слова и следующие за ними выражения – называются фразами: фраза FROM, фраза WHERE и т. д. У каждой фразы свое назначение и логика. Они должны следовать в определенном порядке. Ни одна фраза не является обязательной. Упрощенная синтаксическая диаграмма команды SELECT выглядит следующим образом:

```
SELECT [DISTINCT | ALL] <expression>[[AS] <output_name>][, …]
[FROM <table>[, <table>… | <JOIN clause>…]
[WHERE <condition>]
[GROUP BY <expression>|<output_name>|<output_number> [,…]]
[HAVING <condition>]
[ORDER BY <expression>|<output_name>|<output_number> [ASC | DESC] [NULLS
FIRST | LAST] [,…]]
[OFFSET <expression>]
[LIMIT <expression>];
```

Мы опустили некоторые элементы, например фразы WINDOW, WITH и FOR UPDATE. Полную синтаксическую диаграмму см. на странице http://www.postgresql.org/docs/current/static/sql-select.html.

Некоторые опущенные элементы будут описаны в следующих главах.

Все части команды SELECT необязательны. Например, запрос будет проще, если фильтрация или сортировка не нужна:

```
SELECT * FROM car_portal_app.car;
```

Даже фраза FROM необязательна. Если для вычисления выражения не нужны данные из базы, то запрос имеет вид:

```
car_portal=> SELECT 1;
 ?column?
----------
 1
```

Можно считать это аналогом программы «Hello world» в SQL.

Фраза FROM необязательна в PostgreSQL, но в других РСУБД, например в Oracle, она должна присутствовать.

Логическая последовательность операций, выполняемых командой SELECT, такова:

1) выбрать все записи из всех исходных таблиц. Если во фразе FROM есть подзапросы, то они вычисляются первыми;
2) образовать все возможные комбинации этих записей и отбросить те из них, для которых не удовлетворяются условия JOIN, а в случае внешних соединений установить некоторые поля в таких комбинациях в NULL;
3) отфильтровать комбинации, не удовлетворяющие предикату во фразе WHERE;
4) построить группы в соответствии со значениями выражений в списке GROUP BY;
5) оставить только группы, удовлетворяющие условиям HAVING;
6) вычислить выражения в списке выборки;
7) исключить строки-дубликаты, если присутствует ключевое слово DISTINCT;
8) применить теоретико-множественные операции UNION, EXCEPT и INTERSECT;
9) отсортировать строки в соответствии с фразой ORDER BY;
10) оставить только записи, отвечающие условиям во фразах OFFSET и LIMIT.

На самом деле PostgreSQL оптимизирует этот алгоритм, выполняя шаги в другом порядке или даже одновременно. Например, если задана фраза LIMIT 1, то не имеет смысла выбирать все строки из исходных таблиц, достаточно только одной, удовлетворяющей условию WHERE.

Список выборки

После ключевого слова SELECT задается список столбцов или выражений, выбираемых из базы данных. Он называется **списком выборки** и определяет структуру результата: количества, имена и типы выбранных значений.

Каждому выражению в списке выборки сопоставляется имя. Если имя не задано пользователем, то сервер выбирает его автоматически, и в большинстве случаев оно отражает источник данных: имя столбца таблицы или имя функции. В остальных случаях имя имеет вид ?column?. Допустимо, а иногда даже желательно назвать выбранное выражение по-другому. Для этого предназначено ключевое слово AS, например:

```
SELECT car_id AS identifier_of_the_car ...
```

Теперь результат выборки из столбца car_id таблицы car будет назван identifier_of_the_car. Слово AS необязательно. При выборе имени выходного столбца

следует соблюдать правила именования идентификаторов в SQL. Несколько столбцов результата могут иметь одинаковые имена. Можно использовать имена в двойных кавычках, например если результат запроса играет роль отчета и дальнейшей обработке не подвергается. В таком случае имеет смысл выбирать имена колонок, понятные человеку:

```
SELECT car_id "Идентификатор автомобиля" ...
```

Часто вместо списка выборки употребляют звездочку *, которая обозначает все столбцы всех таблиц, упомянутых во фразе FROM. Можно также использовать звездочку для каждой таблицы в отдельности:

```
SELECT car.*, car_model.make ...
```

В данном случае выбираются все столбцы из таблицы car и только один столбец – make – из таблицы car_model.

Считается дурным тоном использовать * в случаях, когда запрос встречается в другом коде: в приложениях, хранимых процедурах, определениях представлений и т. д. Это не рекомендуется, потому что получается, что формат результата зависит не только от кода запроса, но и от структуры данных. Если структура данных изменится, то изменится и формат результата, и приложение может перестать работать. Если же все столбцы перечислены явно, то добавление нового столбца во входную таблицу не отразится на результатах запроса и работоспособности приложения.

Так, в нашем примере вместо SELECT * ... было бы безопаснее написать:

```
SELECT car_id, number_of_owners, registration_number, number_of_doors,
car_model_id, mileage ...
```

SQL-выражения

Выражения в списке выборки называются **однозначными**, или **скалярными выражениями**, поскольку каждому выражению соответствует только одно значение (хотя оно может быть и массивом).

Иногда скалярные выражения называют SQL-выражениями или просто выражениями. У любого SQL-выражения есть тип данных, определяемый типом (или типами) входных данных. Во многих случаях тип данных можно явно изменить. Каждый элемент списка выборки становится столбцом выходного набора данных, имеющим тип соответствующего выражения.

SQL-выражения могут содержать:

- имена столбцов (в большинстве случаев);
- константы;
- операторы;
- скобки, задающие порядок вычислений;
- вызовы функций;
- агрегатные выражения (см. ниже);
- скалярные подзапросы;
- приведения типов;
- условные выражения.

Этот список неполон. Существуют и другие виды SQL-выражений, которые в этой главе не рассматриваются.

Имена столбцов могут быть **полными** (**квалифицированными**) и **короткими** (**неквалифицированными**). Полное имя содержит, помимо собственно имени столбца, имя таблицы и, возможно, имя схемы, разделенные точками. **Короткое** имя не содержит имени таблицы. Полные имена обязательны, если в нескольких таблицах, упомянутых во фразе `FROM`, встречаются одноименные столбцы. Использование в этом случае коротких имен приведет к ошибке «column reference is ambiguous» (неоднозначная ссылка на столбец). Это означает, что база данных не понимает, какой столбец имеется в виду. Вместо имени таблицы можно употреблять ее псевдоним, а в случае подзапросов или функции использование псевдонима обязательно.

Ниже приведен пример употребления полных имен в списке выборки:

```
SELECT car.car_id, car.number_of_owners FROM car_portal_app.car;
```

SQL поддерживает все операторы, встречающиеся в большинстве других языков программирования: логические, арифметические, над строками, бинарные, над датой и временем и т. д. Ниже мы обсудим логические операторы в связи с условиями в SQL. А вот пример арифметических операторов в выражениях:

```
car_portal=> SELECT 1+1 AS two, 13%4 AS one, -5 AS minus_five, 5! AS
factorial, |/25 AS square_root;
 two | one | minus_five | factorial | square_root
-----+-----+------------+-----------+-------------
   2 |   1 |         -5 |       120 |           5
```

В PostgreSQL разрешено также создавать пользовательские операторы.

В состав SQL-выражения могут также входить вызовы функций. Для вызова функции нужно указать ее имя и в скобках аргументы:

```
car_portal=> SELECT substring('this is a string constant',11,6);
 substring
-----------
 string
```

Здесь вызывается встроенная функция `substring` с тремя аргументами: строка и два целых числа. Она извлекает часть переданной строки, начиная с символа, номер которого задан вторым аргументом, и длиной, заданной третьим аргументом. По умолчанию PostgreSQL назначает выходному столбцу такое же имя, как у функции.

Даже если у функции нет аргументов, скобки все равно необходимы – они показывают, что это именно функция, а не имя столбца и не ключевое слово.

Еще одна вещь, способствующая гибкости и мощи SQL, – скалярные подзапросы, которые могут быть частью однозначного выражения. Это позволяет объединять результаты нескольких запросов. Подзапрос или запрос называет-

ся **скалярным**, если он возвращает ровно один столбец и нуль либо одну строку. Они не отличаются от обычных запросов каким-то особым синтаксисом.

Рассмотрим пример:

```
car_portal=> SELECT (SELECT 1) + (SELECT 2) AS three;
 three
-------
     3
```

Здесь один скалярный подзапрос возвращает значение 1, другой – значение 2. Эти результаты складываются, так что значение всего выражения равно 3.

Под **приведением типа** понимается изменение типа значения. Синтаксически приведение типа может записываться по-разному, но смысл всегда один и тот же:

- `CAST ( <value> AS <type>);`
- `<value>::<type>;`
- `<type> '<value>';`
- `<type> (<value>).`

Первая форма – стандартный синтаксис SQL, который поддерживается большинством баз данных. Вторая – специфика PostgreSQL. Третья применима только к строковым константам и обычно используется, когда нужно определить константу не строкового и не числового типа. Последняя форма похожа на функцию и применима только к типам, имена которых совпадают с именами существующих функций, что не очень удобно. Поэтому этот вариант синтаксиса употребляется редко.

Во многих случаях PostgreSQL выполняет приведение типа неявно. Например, оператор конкатенации || принимает два операнда типа string. При попытке конкатенировать строку с числом PostgreSQL автоматически преобразует число в строку:

```
car_portal=> SELECT 'One plus one equals ' || (1+1) AS str;
str
----------------------
One plus one equals 2
```

Условное выражение возвращает различные результаты в зависимости от условия. Оно похоже на предложение `IF - THEN - ELSE` в других языках программирования. Вот его синтаксис:

```
CASE WHEN <condition1> THEN <expression1> [WHEN <condition2> THEN
<expression2> ...] [ELSE <expression n>] END
```

Поведение понятно: если выполняется первое условие, то возвращается результат первого выражения; если второе условие – то результат второго выражения и т. д. Если ни одно условие не выполнено, то вычисляется выражение, указанное в ветви `ELSE`. Каждое условие `<condition>` само является выражением, возвращающим булево значение (true или false). Все выражения после клю-

чевого слова THEN должны возвращать значения одного типа или, по крайней мере, совместимых типов.

Пар условие-выражение должно быть не менее одной. Ветвь ELSE необязательна, и если она отсутствует и ни одно условие не равно true, то все выражение CASE возвращает NULL.

Выражения CASE можно использовать всюду, где допустимы SQL-выражения. Выражения CASE могут быть вложенными, т. е. одно выражение может находиться на месте части condition или expression в другом выражении. Порядок вычисления условий в точности такой, как указано. Это означает, что если некоторое условие в выражении CASE вообще вычисляется, значит, все предшествующие ему условия равны false. Если некоторое условие равно true, то все последующие не вычисляются вовсе.

Существует упрощенный синтаксис выражений CASE. Если все условия сводятся к сравнению одного и того же выражения на равенство с несколькими значениями, то можно воспользоваться такой формой:

```
CASE <checked_expression> WHEN <value1> THEN <result1> [WHEN <value2> THEN
<result2> ...] [ELSE <result_n>] END
```

Это означает, что когда значение выражения checked_expression равно value1, возвращается result1, и т. д.

Ниже приведен пример выражения CASE:

```
car_portal=> SELECT CASE WHEN now() > date_trunc('day', now()) + interval '12 hours'
THEN 'PM' ELSE 'AM' END;
 case
------
 PM
```

Здесь текущее время сравнивается с полуднем (из текущего времени выделяется день, что дает полночь, а затем прибавляется интервал, равный 12 часам). Если текущее время позже полудня (оператор >), то результатом выражения будет PM, иначе AM.

В одном SQL-выражении может быть много операторов, вызовов функций приведений типов и т. д. В спецификации языка на длину выражения не налагается ограничений. Список выборки – не единственное место, где допускаются SQL-выражения. На самом деле их можно использовать почти в любой части SQL-команды. Например, результаты запроса можно сортировать по некоторому SQL-выражению. SQL-выражения, возвращающие булевы значения, часто используются в качестве условий во фразе WHERE.

PostgreSQL поддерживает «закорачивание» вычисления выражений и в некоторых случаях не вычисляет части выражения, если они не могут повлиять на результат. Например, при вычислении выражения false AND z() функция z() не вызывается, потому что результат оператора AND определен первым операндом, константой false, и будет равен false, какое бы значение ни вернула z().

Ключевое слово DISTINCT

К списку выборки также имеют непосредственное отношение ключевые слова DISTINCT и ALL, которые могут находиться сразу после слова SELECT. Если присутствует DISTINCT, то из выходного набора данных удаляются дубликаты. Если же указано слово ALL (это режим по умолчанию), то возвращаются все строки.

Рассмотрим два примера:

```
car_portal=> SELECT ALL make FROM car_portal_app.car_model;
 make
---------------
Audi
Audi
Audi
Audi
BMW
BMW
...
(99 rows)
```

и

```
car_portal=> SELECT DISTINCT make FROM car_portal_app.car_model;
 make
---------------
Ferrari
GMC
Citroen
UAZ
Audi
Volvo
...
(25 rows)
```

В обоих случаях мы производим выборку из таблицы моделей автомобилей. Но первый запрос вернул 99 записей, а второй – только 25. Это объясняется тем, что в первом случае возвращены все строки входной таблицы, а во втором – только уникальные строки. Ключевое слово DISTINCT удаляет дубликаты, ориентируясь на список выборки, а не на данные в исходной таблице. Так, если бы мы возвращали только первую букву модели, то результат оказался бы еще короче, потому что некоторые названия начинаются с одной и той же буквы:

```
SELECT DISTINCT substring(make, 1, 1) FROM car_portal_app.car_model;
substring
-----------
H
S
C
J
L
I
...
(21 rows)
```

Слово DISTINCT применимо и тогда, когда выбирается несколько столбцов. В этом случае удаляются повторяющиеся комбинации значений столбцов.

Фраза FROM

Источники строк указываются после ключевого слова FROM. Эта часть запроса называется фразой FROM. Выбирать данные можно из нуля, одной и более таблиц. Если источник строк отсутствует, то фразы FROM быть не должно. Возможны следующие источники строк:

- таблица;
- представление;
- функция;
- подзапрос;
- фраза VALUES.

Несколько источников перечисляется через запятую или включается во фразы JOIN.

Для таблиц во фразе FROM можно задавать псевдонимы с помощью ключевого слова AS:

```
car_portal=> SELECT a.car_id, a.number_of_doors FROM car_portal_app.car AS a;
 car_id | number_of_doors
--------+-----------------
      1 |               5
      2 |               3
      3 |               5
 ...
```

Здесь для таблицы car_portal_app.car указан псевдоним a, который добавлен к именам столбцов в списке выборки. Если для таблицы или представления указан псевдоним во фразе FROM, то ни в списке выборки, ни где-либо еще нельзя ссылаться на таблицу по имени. Если во фразе FROM встречается подзапрос, то задавать для него псевдоним обязательно. Псевдонимы часто используются в случае самосоединения, когда одна таблица встречается во фразе FROM несколько раз.

Выборка из нескольких таблиц

Разрешается выбирать данные сразу из нескольких источников. Пусть имеется две таблицы по три строки в каждой:

```
car_portal=> SELECT * FROM car_portal_app.a;
 a_int | a_text
-------+--------
     1 | one
     2 | two
     3 | three
(3 rows)

car_portal=> SELECT * FROM car_portal_app.b;
 b_int | b_text
```

```
-------+--------
     2 | two
     3 | three
     4 | four
(3 rows)
```

Если выбирать записи из обеих, то будут возвращены все возможные комбинации строк:

```
car_portal=> SELECT * FROM car_portal_app.a, car_portal_app.b;
 a_int | a_text | b_int | b_text
-------+--------+-------+--------
     1 | one    | 2     | two
     1 | one    | 3     | three
     1 | one    | 4     | four
     2 | two    | 2     | two
     2 | two    | 3     | three
     2 | two    | 4     | four
     3 | three  | 2     | two
     3 | three  | 3     | three
     3 | three  | 4     | four
(9 rows)
```

Множество всех возможных комбинаций строк, взятых по одной из каждой таблицы, называется **декартовым произведением** таблиц и редко бывает полезно. Как правило, пользователя интересуют только некоторые комбинации строк: такие, в которых строка из одной таблицы соответствует строке из другой таблицы согласно некоторому критерию. Например, часто необходимо выбрать только такие комбинации, в которых соответственные целочисленные поля в двух таблицах равны. Для этого мы должны изменить запрос:

```
car_portal=> SELECT * FROM car_portal_app.a, car_portal_app.b WHERE a_int=b_int;
a_ int | a_text | b_int | b_text
-------+--------+-------+--------
     2 | two    | 2     | two
     3 | three  | 3     | three
(2 rows)
```

Таблицы соединяются по условию `a_int=b_int`. Условия соединения можно задавать во фразе `WHERE`, но обычно лучше делать это во фразе `FROM`, чтобы было ясно, что они предназначены именно для соединения, а не для фильтрации результата соединения, хотя формально разницы никакой нет.

Для добавления условий соединения во фразу `FROM` служит ключевое слово `JOIN`. Следующий запрос логически эквивалентен предыдущему и дает такие же результаты:

```
SELECT * FROM car_portal_app.a JOIN car_portal_app.b ON a_int=b_int;
```

Условие `JOIN` можно задать одним из трех способов: с помощью ключевых слов `ON`, `USING` и `NATURAL`.

```
<first table> JOIN <second table> ON <condition>
```

В роли условия может выступать любое SQL-выражение, возвращающее булево значение. Необязательно даже включать столбцы соединяемых таблиц.

```
<first table> JOIN <second table> USING (<field list>)
```

Здесь предполагается соединение по равенству всех полей, перечисленных через запятую в списке `<field list>`. Поля должны существовать в обеих таблицах и называться одинаково, поэтому такой синтаксис недостаточно гибкий.

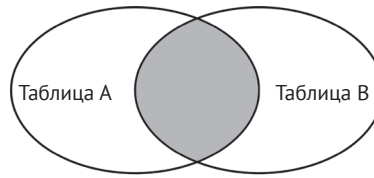
```
<first table> NATURAL JOIN <second table>
```

Здесь предполагается соединение по равенству всех одноименных полей в обеих таблицах.

Синтаксис с применением `USING` и `NATURAL JOIN` обладает тем же недостатком, что и использование * в списке выборки. Если изменить структуру таблицы, например добавить новый столбец или переименовать старый, то может получиться, что запрос останется корректным, но его семантика изменится. Такие ошибки трудно искать.

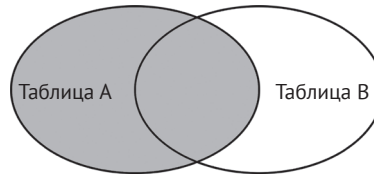
Что, если не для всех строк первой таблицы существует соответствие во второй? В нашем примере только значения 2 и 3 присутствуют в обеих таблицах. Поэтому если соединять по условию `a_int=b_int`, то только эти две строки и будут отобраны из каждой таблицы. Такое соединение называется **внутренним**.

Его можно наглядно представить заштрихованной областью на следующем рисунке:



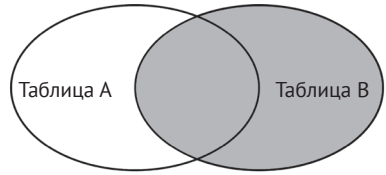
Внутреннее соединение

Если выбираются все записи из одной таблицы вне зависимости от существования парной записи в другой, то получается **внешнее соединение**. Существует три типа внешних соединений, все они представлены на рисунках ниже.



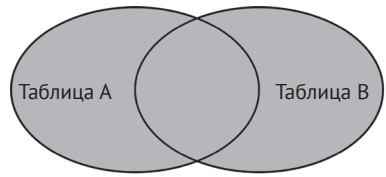
Левое внешнее соединение

Если из первой таблицы выбираются все записи, а из второй записи – удовлетворяющие условию соединения, то говорят о **левом внешнем соединении**.



Правое внешнее соединение

Если из второй таблицы выбираются все записи, а из первой – записи, удовлетворяющие условию соединения, то говорят о **правом внешнем соединении**.



Полное внешнее соединение

Если из обеих таблиц выбираются все записи, то мы имеем **полное внешнее соединение**. Ключевые слова `INNER` и `OUTER` необязательны. Рассмотрим несколько примеров:

```
car_portal=> SELECT * FROM car_portal_app.a JOIN car_portal_app.b
ON a_int=b_int;
 a_int | a_text | b_int | b_text
-------+--------+-------+--------
     2 | two    |     2 | two
     3 | three  |     3 | three
(2 rows)

car_portal=> SELECT * FROM car_portal_app.a LEFT JOIN car_portal_app.b
ON a_int=b_int;
 a_int | a_text | b_int | b_text
-------+--------+-------+--------
     1 | one    |       |
     2 | two    |     2 | two
     3 | three  |     3 | three
(3 rows)

car_portal=> SELECT * FROM car_portal_app.a RIGHT JOIN car_portal_app.b
ON a_int=b_int;
 a_int | a_text | b_int | b_text
-------+--------+-------+--------
     2 | two    |     2 | two
```

```
     3 | three  |     3 | three
       |        |     4 | four
(3 rows)

car_portal=> SELECT * FROM car_portal_app.a FULL JOIN car_portal_app.b
ON a_int=b_int;
 a_int | a_text | b_int | b_text
-------+--------+-------+--------
     1 | one    |       |
     2 | two    |     2 | two
     3 | three  |     3 | three
       |        |     4 | four
(4 rows)
```

Отметим, что декартово произведение и полное внешнее соединение – не одно и то же. В декартово произведение входят все возможные комбинации записей таблиц без учета каких-либо правил сопоставления. А в случае полного внешнего соединения пара записей возвращается, если удовлетворяются условия соединения. Те записи, для которых не нашлось пары в другой таблице, возвращаются по отдельности. В столбцах внешнего соединения, взятых из той таблицы, где не оказалось пары, будут находиться значения NULL.

Опрашивать можно не только таблицы, но также представления, функции и подзапросы, а значит, их также можно соединять, применяя такой же синтаксис:

```
car_portal=> SELECT *
 FROM car_portal_app.a
 INNER JOIN (SELECT * FROM car_portal_app.b WHERE b_text = 'two') subq ON
a.a_int=subq.b_int;
 a_int | a_text | b_int | b_text
-------+--------+-------+--------
     2 | two    |     2 | two
```

В этом примере подзапросу сопоставлен псевдоним subq, который используется в условии соединения.

Соединять можно и больше двух таблиц. На самом деле всякий оператор JOIN соединяет все таблицы слева от ключевого слова JOIN с таблицей, указанной непосредственно справа от него. Например, следующий запрос правилен:

```
SELECT *
  FROM table_a
    JOIN table_b ON table_a.field1=table_b.field1
    JOIN table_c ON table_a.field2=table_c.field2 AND
table_b.field3=table_c.field3;
```

В момент соединения с таблицей table_c таблица table_a уже встречалась во фразе FROM, поэтому на нее можно сослаться в условии соединения.

Однако такой запрос недопустим:

```
SELECT *
  FROM table_a
```

```
    JOIN table_b ON table_b.field3=table_c.field3
    JOIN table_c ON table_a.field2=table_c.field2
```

Ошибка состоит в том, что в момент JOIN table_b таблица table_c еще не встречалась, поэтому на нее нельзя ссылаться.

Декартово произведение тоже можно реализовать с помощью соединения. Для этого предназначена конструкция CROSS JOIN:

```
SELECT * FROM car_portal_app.a CROSS JOIN car_portal_app.b;
```

Этот код эквивалентен такому:

```
SELECT * FROM car_portal_app.a, car_portal_app.b;
```

Условие соединения в операторе INNER JOIN логически эквивалентно условию фильтрации строк во фразе WHERE, так что следующие два запроса, по существу, не отличаются:

```
SELECT * FROM car_portal_app.a INNER JOIN car_portal_app.b ON a.a_int=b.b_int;
```

```
SELECT * FROM car_portal_app.a, car_portal_app.b WHERE a.a_int=b.b_int;
```

Но для внешних соединений это уже не так. В PostgreSQL невозможно реализовать внешнее соединение с помощью фразы WHERE, хотя в некоторых других СУБД такая возможность есть.

Самосоединения

Можно соединять таблицу саму с собой, это называется **самосоединением**. Никакого специального синтаксиса для самосоединения не предусмотрено. На самом деле все источники данных в запросе независимы, пусть даже физически они совпадают. Пусть, например, для каждой записи таблицы a требуется узнать, сколько в ней существует записей, для которых значение в поле a_int больше, чем в том же поле данной записи. Задачу решает такой запрос:

```
car_portal=> SELECT t1.a_int AS current, t2.a_int AS bigger
  FROM car_portal_app.a t1
  INNER JOIN car_portal_app.a t2 ON t2.a_int > t1.a_int;
current | bigger
--------+--------
      1 |      2
      1 |      3
      2 |      3
(3 rows)
```

Таблица a соединяется с собой. С точки зрения логики запроса не важно, соединяются две разные таблицы или два раза одна и та же. Чтобы можно было ссылаться на поля и различать экземпляры таблиц, мы используем псевдонимы. Первому экземпляру сопоставлен псевдоним t1, второму – t2. Результат показывает, что для значения 1 существует два больших значения: 2 и 3, а для значения 2 – только одно большее значение, 3. Столбец первой таблицы, содержащий исследуемое значение, назван current, а столбец второй таблицы, содержащий большие значения, – bigger.

Значение 3 не представлено в левом столбце результата, потому что в таблице не существует значений, больших 3. Но если бы мы захотели отразить этот факт явно, то могли бы воспользоваться левым соединением:

```
car_portal=> SELECT t1.a_int AS current, t2.a_int AS bigger
  FROM car_portal_app.a t1
  LEFT JOIN car_portal_app.a t2 ON t2.a_int > t1.a_int;
current | bigger
--------+--------
      1 |      2
      1 |      3
      2 |      3
      3 |
(4 rows)
```

Фраза WHERE

Во многих случаях отобранные из входных таблиц строки необходимо отфильтровать. Для этого предназначена фраза `WHERE`. Условие фильтрации задается после ключевого слова `WHERE` и представляет собой SQL-выражение, возвращающее булево значение. Таким образом, синтаксис условия `WHERE` такой же, как у выражений в списке выборки. Это особенность PostgreSQL, в других СУБД булевы значения в списке выборки не допускаются, поэтому условия и выражения SQL – не одно и то же. В PostgreSQL единственная разница между ними – тип возвращаемого значения.

Приведем простые примеры условий `WHERE`:

```
SELECT * FROM car_portal_app.car_model WHERE make='Peugeot';
SELECT * FROM car_portal_app.car WHERE mileage < 25000;
SELECT * FROM car_portal_app.car WHERE number_of_doors > 3 AND number_of_owners <= 2;
```

Хотя в PostgreSQL SQL-выражения и SQL-условия формально не различаются, обычно выражения, возвращающие булево значение, все-таки называются условными выражениями или просто условиями. Чаще всего они употребляются во фразе `WHERE`, в выражении `CASE` и в операторе `JOIN`.

В условных выражениях нередко используются логические операторы `AND`, `OR` и `NOT`. Они принимают булевы операнды и возвращают булевы значения. Логические операторы вычисляются в порядке `NOT`, `AND`, `OR`, но все они имеют более низкий приоритет, чем другие операторы. PostgreSQL старается оптимизировать вычисление логических выражений. Например, если при вычислении оператора `OR` известно, что первый операнд равен true, то второй операнд не вычисляется вовсе, потому что результат `OR` уже известен. Поэтому PostgreSQL может изменять порядок вычисления выражений, чтобы быстрее получить результат.

Иногда это приводит к проблемам. Например, деление на нуль запрещено, и попытка отфильтровать строки, основываясь на результате такого деления, некорректна:

```
car_portal=> SELECT * FROM t WHERE b/a>0.5 and a<>0;
ERROR: division by zero
```

Не гарантируется, что PostgreSQL вычислит условие `a<>0` раньше, чем `b/a>0.5`, а значит, в случае, когда `a` равно 0, произойдет ошибка. Чтобы застраховаться, следовало бы использовать выражение `CASE`, потому в нем условия вычисляются в том порядке, в котором записаны:

```
car_portal=> SELECT * FROM t WHERE CASE WHEN a=0 THEN false ELSE b/a>0.5 END;
 a | b
---+---
(0 rows)
```

В условных выражениях используются и другие операторы или выражения, возвращающие булевы значения:

- операторы сравнения;
- операторы сопоставления с образцом;
- оператор `OVERLAPS`;
- конструкции сравнения строк и массивов;
- выражения подзапросов;
- любая функция, возвращающая булево или приводимое к булеву значение.

Как и в случае списка выборки, функции можно использовать во фразе `WHERE`, а также в любом месте выражения. Пусть требуется найти автомобили, для которых название модели состоит из четырех символов. Это можно сделать с помощью функции length:

```
car_portal=> SELECT * FROM car_portal_app.car_model WHERE length(model)=4;
 car_model_id | make       | model
--------------+------------+-------
           47 | KIA        | Seed
           57 | Nissan     | GT-R
           70 | Renault    | Clio
...
```

Операторы сравнения

К операторам сравнения относятся < (меньше), > (больше), <= (меньше или равно), >= (больше или равно), = (равно) <> или != (то и другое означает не равно). Они используются для сравнения не только чисел, но и любых сравнимых величин, например дат и строк. Существует также оператор `BETWEEN`, который употребляется следующим образом:

```
x BETWEEN a AND b
```

и эквивалентен выражению

```
x>=a AND a<=b
```

Оператор `OVERLAPS` проверяет, перекрываются ли два диапазона дат, например:

```
car_portal=> SELECT 1 WHERE (date '2017-10-15', date '2017-10-31')
OVERLAPS (date '2017-10-25', date '2017-11-15');
 ?column?
----------
        1
```

Формально наибольший приоритет имеют операторы сравнения >= и <=. За ними следует BETWEEN, затем OVERLAPS, затем < и >. Наименьший приоритет у оператора =. Однако довольно трудно привести практический пример употребления нескольких операторов сравнения в одном выражении без скобок.

Сопоставление с образцом

Говоря о сопоставлении с образцом, всегда имеют в виду строки. Есть два похожих оператора: LIKE и ILIKE. Они сравнивают строку с образцом, который может содержать только два метасимвола: _ (соответствует одному любому символу) и % (соответствует произвольной последовательности любых символов, в т. ч. пустой строке). LIKE и ILIKE отличаются только тем, что первый чувствителен к регистру, а второй – нет.

Например, чтобы получить модели автомобилей, названия которых начинаются буквой s (в любом регистре) и содержат ровно четыре символа, можно было бы написать:

```
car_portal=> SELECT * FROM car_portal_app.car_model WHERE model ILIKE 's___';
 car_model_id | make       | model
--------------+------------+-------
           47 | KIA        | Seed
```

Существуют еще два оператора сопоставления с образцом: SIMILAR и ~ (тильда). Они производят сравнение с регулярным выражением. Разница в том, что SIMILAR применяет синтаксис регулярных выражений, определенный в стандарте SQL, а ~ – синтаксис из стандарта **POSIX** (Portable Operating System Interface – переносимый интерфейс операционных систем).

В примере ниже выбираются модели, названия которых состоят ровно из двух слов:

```
car_portal=> SELECT * FROM car_portal_app.car_model WHERE model ~ '^\w+\W+\w+$';
 car_model_id | make          | model
--------------+---------------+-------------
           21 | Citroen       | C4 Picasso
           33 | Ford          | C-Max
           34 | Ford          | S-Max
...
```

Конструкции сравнения строк и массивов

Эти конструкции служат для сравнения нескольких значений, сравнения с группой значений и сравнения с массивом.

Оператор IN проверяет, совпадает ли значение хотя бы с одним значением из списка:

```
a IN (1, 2, 3)
```

Этот код возвращает true, если a равно 1, 2 или 3. Это краткий и более чистый эквивалент выражения:

```
(a = 1 OR a = 2 OR a = 3)
```

В SQL имеется тип массива, позволяющий рассматривать несколько элементов как одно значение. Подробнее о массивах речь пойдет в главе 9. Массивы обогащают арсенал способов сравнения. Например, вот как можно проверить, что a больше какого-нибудь из значений x, y, z:

```
a > ANY (ARRAY[x, y, z])
```

Этот код эквивалентен следующему:

```
(a > x OR a > y OR a > z)
```

А этот код проверяет, что a больше каждого из x, y, z:

```
a > ALL (ARRAY[x, y, z])
```

и эквивалентен такому:

```
(a > x AND a > y AND a > z )
```

Ключевые слова IN, ALL и ANY (и его синоним SOME) применимы и к подзапросам, где они имеют такую же семантику. Результат подзапроса можно использовать всюду, где допустимо множество значений или массив. Например, можно выбрать такие записи из одной таблицы, для которых некоторое значение встречается в другой таблице.

Следующий запрос отбирает модели машин, если хотя бы один автомобиль такой модели выставлен на продажу:

```
car_portal=> SELECT * FROM car_portal_app.car_model
WHERE car_model_id IN (SELECT car_model_id FROM car_portal_app.car);
 car_model_id | make          | model
--------------+---------------+--------------
            2 | Audi          | A2
            3 | Audi          | A3
            4 | Audi          | A4
...
(86 rows)
```

Иногда (но не всегда) выражение IN можно заменить внутренним соединением. Рассмотрим пример:

```
car_portal=> SELECT car_model.*
FROM car_portal_app.car_model INNER JOIN car_portal_app.car USING (car_model_id);
 car_model_id | make          | model
--------------+---------------+--------------
            2 | Audi          | A2
            2 | Audi          | A2
            2 | Audi          | A2
            3 | Audi          | A3
            3 | Audi          | A3
            4 | Audi          | A4
            4 | Audi          | A4
...
(229 rows)
```

Опрашивается одна и та же таблица, возвращаются одни и те же столбцы, но во втором случае строк возвращено больше. Дело в том, что на продажу выставлено много автомобилей одной модели, и для них модель выбирается несколько раз.

Конструкция `NOT IN` вместе с подзапросом иногда работает очень медленно, потому что проверять, что значение не существует, дороже, чем обратное.

Группировка и агрегирование

В предыдущих примерах количество строк, возвращенных запросом `SELECT`, совпадало с количеством строк во входной таблице (или таблицах), оставшей(их)ся после фильтрации. Иными словами, каждая строка исходной таблицы (или соединенных таблиц) превращается ровно в одну строку результата. Строки обрабатываются одна за другой.

SQL предлагает также способ агрегировать результаты обработки нескольких записей и возвращать одну результирующую строку. Самый простой пример – подсчет количества строк в таблице. На входе мы имеем все строки, на выходе – всего одну. Для таких задач предназначены операции группировки и агрегирования.

Фраза GROUP BY

Фраза `GROUP BY` служит для группировки. Под **группировкой** понимается разбиение всего множества входных записей на несколько групп, имея в виду получение одной результирующей строки для каждой группы. Группировка производится по списку выражений. Все строки с одной и той же комбинацией значений группирующих выражений объединяются в одну группу. Таким образом, группы определяются значениями выражений, указанных во фразе `GROUP BY`. Обычно имеет смысл включать эти выражения в список выборки, чтобы было видно, к какой группе относится каждая результирующая строка.

Сгруппируем данные по марке и модели автомобиля и выберем группы:

```
car_portal=> SELECT a.make, a.model
FROM car_portal_app.car_model a
INNER JOIN car_portal_app.car b ON a.car_model_id=b.car_model_id
GROUP BY a.make, a.model;
 make          | model
---------------+-------------
 Opel          | Corsa
 Ferrari       | 458 Italia
 Peugeot       | 308
 Opel          | Insignia
...
(86 rows)
```

Здесь мы видим список всех моделей, представленных в таблице `car`. Каждая результирующая запись представляет группу записей исходных таблиц, относящихся к одной и той же модели. На самом деле этот запрос дает такой же ре-

зультат, как SELECT DISTINCT make, model... без GROUP BY, но логика их выполнения различна. DISTINCT удаляет дубликаты, а GROUP BY объединяет дубликаты в одну группу.

От простой группировки строк толку мало. Обычно требуется произвести некоторые вычисления над группами. Так, было бы интересно знать, сколько автомобилей каждой модели имеется в системе. Это делается с помощью **агрегирования**, т. е. такого вычисления над группами записей, в результате которого для каждой группы возвращается одно значение. Для этой цели имеются специальные агрегатные функции, указываемые в списке выборки. Чтобы получить количество автомобилей, воспользуемся функцией count:

```
car_portal=> SELECT a.make, a.model, count(*)
FROM car_portal_app.car_model a
INNER JOIN car_portal_app.car b ON a.car_model_id=b.car_model_id
GROUP BY a.make, a.model;
 make          | model        | count
---------------+--------------+-------
 Opel          | Corsa        | 6
 Ferrari       | 458 Italia   | 4
 Peugeot       | 308          | 3
 Opel          | Insignia     | 4
...
(86 rows)
```

В PostgreSQL имеется несколько агрегатных функций. Чаще всего используются count, sum, max, min и avg, которые вычисляют, соответственно, количество записей в группе, сумму числовых выражений по всем записям, наибольшее, наименьшее и среднее значения выражения. Существуют и другие агрегатные функции, например corr вычисляет коэффициент корреляции двух аргументов, stddev – стандартное отклонение, string_agg – конкатенацию строковых значений и т. д.

Для агрегирования записи группируются, т. е. несколько записей отображается в одну. Поэтому в списке выборки не может быть ничего, кроме агрегатных функций и выражений, вошедших в список GROUP BY, в противном случае возникнет ошибка:

```
car_portal=> SELECT a_int, a_text FROM car_portal_app.a GROUP BY a_int;
ERROR: column "a.a_text" must appear in the GROUP BY clause or be used in
an aggregate function
```

Разрешается создавать новые выражения на основе тех, что вошли в список GROUP BY. Например, если имеется фраза GROUP BY a, b, то можно написать запрос, начинающийся с SELECT a+b.

А что, если требуется объединить в одну группу все записи таблицы безотносительно к отдельным полям? Тогда нужно включить в список выборки агрегатные функции (и только их!) и опустить фразу GROUP BY:

```
car_portal=> SELECT count(*) FROM car_portal_app.car;
 count
```

```
-------
   229
(1 row)
```

Если фраза GROUP BY отсутствует, но предполагается группировка, то создается ровно одна группа. Отметим, что SQL-запросы с агрегатными функциями в списке выборки, но без фразы GROUP BY, всегда возвращают ровно одну строку, даже если во входных таблицах нет записей или все они отфильтрованы:

```
car_portal=> SELECT count(*) FROM car_portal_app.car WHERE number_of_doors = 15;
 count
-------
     0
(1 row)
```

Не бывает машин с 15 дверьми. Если производится выборка с таким условием WHERE, то не будет возвращено ни одной строки. Однако если в списке выборки находится агрегатная функция count(*), то запрос вернет одну строку со значением 0.

Чтобы подсчитать количество уникальных значений выражения, воспользуемся функцией count(DISTINCT <expression>):

```
car_portal=> SELECT count(*), count(DISTINCT car_model_id) FROM car_portal_app.car;
 count | count
-------+-------
   229 | 86
(1 row)
```

Здесь первый столбец результата содержит общее количество автомобилей, а второй – количество различных моделей, выставленных на продажу. Поскольку для некоторых моделей выставлено по нескольку автомобилей, второе число меньше.

Фраза HAVING

Агрегатные функции не могут встречаться во фразе WHERE, однако мы можем отфильтровать группы, удовлетворяющие некоторому условию. Это не то же самое, что фильтрация во фразе WHERE, потому что WHERE фильтрует входные строки, а группы вычисляются уже после этого.

Для фильтрации групп предназначена фраза HAVING. Она похожа на фразу WHERE, но в ней разрешены агрегатные функции. Фраза HAVING следует после GROUP BY. Пусть требуется узнать, для каких моделей на продажу выставлено более пяти автомобилей. Это можно сделать с помощью такого подзапроса:

```
car_portal=> SELECT make, model FROM
(
  SELECT a.make, a.model, count(*) c
    FROM car_portal_app.car_model a
      INNER JOIN car_portal_app.car b ON a.car_model_id=b.car_model_id
    GROUP BY a.make, a.model
) subq
```

```
WHERE c > 5;
 make    | model
---------+-------
 Opel    | Corsa
 Peugeot | 208
(2 rows)
```

Но есть и более простой способ, основанный на фразе HAVING:

```
car_portal=> SELECT a.make, a.model
  FROM car_portal_app.car_model a
    INNER JOIN car_portal_app.car b ON a.car_model_id=b.car_model_id
  GROUP BY a.make, a.model
  HAVING count(*)>5;
 make    | model
---------+-------
 Opel    | Corsa
 Peugeot | 208
```

Упорядочение и ограничение количества результатов

По умолчанию результаты запроса не упорядочены. Порядок строк не определен и может зависеть от их физического расположения на диске, от алгоритма соединения и других факторов. Часто требуется отсортировать результирующий набор. Для этого предназначена фраза ORDER BY, где задаются выражения, по которым производится сортировка. Сначала записи сортируются по первому выражению. Если в каких-то записях значения первого выражения совпадают, то они сортируются по второму выражению и т. д.

После каждого элемента списка ORDER BY можно указать порядок сортировки: по возрастанию (ASC) или по убыванию (DESC). По умолчанию производится сортировка в порядке возрастания. По умолчанию считается, что значение NULL больше любого другого, но это можно указать и явно: NULLS FIRST означает, что значения NULL должны располагаться в начале, а NULLS LAST – что в конце.

Необязательно, чтобы во фразе ORDER BY были те же выражения, что в списке выборки, но обычно это так. Для удобства разрешается использовать в списке ORDER BY псевдонимы, сопоставленные выходным столбцам, а не только полные выражения. Кроме того, можно указывать не имена, а номера столбцов. Таким образом, следующие запросы эквивалентны:

```
SELECT number_of_owners, manufacture_year, trunc(mileage/1000) as kmiles
FROM car_portal_app.car
ORDER BY number_of_owners, manufacture_year, trunc(mileage/1000) DESC;

SELECT number_of_owners, manufacture_year, trunc(mileage/1000) as kmiles
FROM car_portal_app.car
ORDER BY number_of_owners, manufacture_year, kmiles DESC;

SELECT number_of_owners, manufacture_year, trunc(mileage/1000) as kmiles
FROM car_portal_app.car
ORDER BY 1, 2, 3 DESC;
```

Иногда требуется ограничить количество строк результата, отбросив остальные. Для этого предназначено ключевое слово LIMIT:

```
car_portal=> SELECT * FROM car_portal_app.car_model LIMIT 5;
 car_model_id | make  | model
--------------+-------+-------
            1 | Audi  | A1
            2 | Audi  | A2
            3 | Audi  | A3
            4 | Audi  | A4
            5 | Audi  | A5
(5 rows)
```

Здесь возвращается только пять строк вне зависимости от того, сколько их на самом деле в таблице. Иногда это используется в скалярных подзапросах, которые не должны возвращать более одной строки.

Еще одна похожая задача – пропустить несколько начальных строк. Для этого служит ключевое слово OFFSET. Часто OFFSET и LIMIT употребляются вместе:

```
car_portal=> SELECT * FROM car_portal_app.car_model OFFSET 5 LIMIT 5;
 car_model_id |  make | model
--------------+-------+-------
            6 | Audi  | A6
            7 | Audi  | A8
            8 | BMW   | 1er
            9 | BMW   | 3er
           10 | BMW   | 5er
(5 rows)
```

Типичный пример употребления OFFSET и LIMIT – разбиение результата на страницы в веб-приложениях. Так, если результат отображается страницами по десять строк, то на третьей странице должны быть показаны строки с 21 по 30. Чтобы этого добиться, можно воспользоваться конструкцией OFFSET 20 LIMIT 10. Как правило, при использовании OFFSET и LIMIT строки должны быть упорядочены, иначе не определено, какие строки будут показаны. В таком случае эти ключевые слова указываются в запросе после фразы ORDER BY.

Подзапросы

Подзапросы – чрезвычайно мощное средство SQL. Они могут находиться чуть ли не в любом месте запроса. Наиболее очевидный способ использования подзапросов – во фразе FROM в качестве источника данных для основного запроса:

```
car_portal=> SELECT * FROM
  (SELECT car_model_id, count(*) c FROM car_portal_app.car
   GROUP BY car_model_id) subq
  WHERE c = 1;
 car_model_id | c
--------------+---
            8 | 1
           80 | 1
...
(14 rows)
```

Когда подзапрос используется во фразе FROM, ему должен быть сопоставлен псевдоним. В примере выше подзапросу присвоено имя subq.

Часто подзапросы используются в SQL-условиях в составе выражения IN:

```
car_portal=> SELECT car_id, registration_number
FROM car_portal_app.car
WHERE car_model_id IN
 (SELECT car_model_id FROM car_portal_app.car_model WHERE make='Peugeot');
 car_id | registration_number
--------+---------------------
      1 | MUWH4675
     14 | MTZC8798
     18 | VFZF9207
...
(18 rows)
```

Скалярные подзапросы можно использовать в любом месте выражения – в списке выборки, во фразе WHERE, во фразе GROUP BY и т. д. Даже в LIMIT:

```
car_portal=> SELECT (SELECT count(*) FROM car_portal_app.car_model)
FROM car_portal_app.car
LIMIT (SELECT MIN(car_id)+2 FROM car_portal_app.car);
 count
-------
    99
    99
    99
(3 rows)
```

Это особенность PostgreSQL. Не каждая РСУБД поддерживает подзапросы всюду, где допустимы выражения.

Невозможно сослаться извне на внутренние элементы подзапроса. Однако подзапрос может ссылаться на элементы главного запроса. Например, пусть требуется подсчитать количество выставленных на продажу автомобилей каждой модели и выбрать пять самых популярных моделей. С помощью подзапроса это можно сделать следующим образом:

```
car_portal=> SELECT make, model,
    (SELECT count(*) FROM car_portal_app.car WHERE car_model_id = main.car_model_id)
FROM car_portal_app.car_model AS main
ORDER BY 3 DESC
LIMIT 5;
 Make    | model    | count
---------+----------+-------
 Peugeot | 208      |     7
    Opel | Corsa    |     6
    Jeep | Wrangler |     5
 Renault | Laguna   |     5
 Peugeot | 407      |     5
(5 rows)
```

Здесь в списке выборки подзапроса имеется ссылка на таблицу из главного запроса по ее псевдониму main. Подзапрос выполняется для каждой записи, вы-

бранной из таблицы `main`, и в его условии `WHERE` фигурирует значение столбца `car_model_id`.

Подзапросы могут быть вложенными, т. е. один подзапрос может находиться внутри другого.

Теоретико-множественные операции – UNION, EXCEPT, INTERSECT

Теоретико-множественные операции используются для *объединения* результатов нескольких запросов. Это не то же самое, что соединение, хотя конечный результат часто можно получить и с помощью соединения. Проще говоря, соединение означает, что записи, взятые из двух таблиц, помещаются рядом друг с другом горизонтально. Количество столбцов в соединении равно сумме количества столбцов в исходных таблицах, а количество строк зависит от условий соединения.

Напротив, объединение означает, что результат одного запроса располагается под результатом другого. Количество столбцов остается тем же самым, а количество строк суммируется.

Существует три теоретико-множественные операции:

- **UNION**: добавляет результат одного запроса к результату другого;
- **INTERSECT**: возвращает записи, присутствующие в результатах обоих запросов;
- **EXCEPT**: возвращает записи, присутствующие в результатах первого запроса, но отсутствующие в результатах второго, – разность.

Синтаксически теоретико-множественные операции записываются следующим образом:

```
<query1> UNION <query2>;

<query1> INTERSECT <query2>;

<query1> EXCEPT <query2>;
```

Разрешается использовать несколько теоретико-множественных операций в одной команде:

```
SELECT a, b FROM t1
UNION
SELECT c, d FROM t2
INTERSECT
SELECT e, f FROM t3;
```

У всех теоретико-множественных операций одинаковые приоритеты, т. е. они выполняются в том порядке, в каком записаны. Однако записи могут возвращаться в произвольном порядке, если отсутствует фраза `ORDER BY`. Если же эта фраза присутствует, то должна располагаться после всех теоретико-множественных операций. Поэтому не имеет смысла включать ее в подзапросы.

По умолчанию все теоретико-множественные операции удаляют дубликаты, как если бы было написано SELECT DISTINCT. Если требуется вернуть все записи, следует добавить ключевое слово ALL после названия операции:

```
<query1> UNION ALL <query2>.
```

Теоретико-множественные операции позволяют найти симметрическую разность двух таблиц:

```
car_portal=> SELECT 'a', * FROM
(
  SELECT * FROM car_portal_app.a
  EXCEPT ALL
  SELECT * FROM car_portal_app.b
) v1
UNION ALL
SELECT 'b', * FROM
(
  SELECT * FROM car_portal_app.b
  EXCEPT ALL
  SELECT * FROM car_portal_app.a
) v2;
?column? | a_int | a_text
----------+-------+--------
a | 1 | one
b | 4 | four
(2 rows)
```

Как видим, строка 1 существует в таблице a, но отсутствует в таблице b, а строка 4 существует в таблице b, но отсутствует в a.

Добавить один набор записей к другому можно только в том случае, когда количество столбцов в обоих наборах одинаково и типы соответственных столбцов совпадают или совместимы. Имена столбцов могут различаться, они всегда берутся из первого набора.

В других РСУБД теоретико-множественные операции могут называться по-другому. Например, в Oracle операция EXCEPT называется MINUS.

Значения NULL

NULL – специальное значение, которое может принимать любой столбец или выражение, если только это явно не запрещено. NULL означает отсутствие всякого значения. В некоторых случаях NULL можно интерпретировать как неизвестное значение. NULL не равен ни true, ни false. Работать со значениями NULL неприятно, потому что почти все операторы, получив NULL в качестве аргумента, возвращают также NULL. При попытке сравнить значения, одно из которых равно NULL, получится NULL, а вовсе не true и не false.

Рассмотрим, к примеру, следующее условие:

```
WHERE a > b
```

Если a или b принимает значение NULL, то это условие вернет NULL. Это еще более-менее ожидаемо, чего не скажешь о следующем условии:

```
WHERE a = b
```

Здесь если a и b принимают значение NULL, то и результат будет NULL. Оператор сравнения на равенство = возвращает NULL, если хотя бы один его аргумент равен NULL. В частности, следующее условие вернет NULL, даже если a принимает значение NULL:

```
WHERE a = NULL
```

Для проверки выражения на NULL имеется специальный предикат: IS NULL.

Так, в приведенных выше примерах, чтобы найти записи, для которых a = b или a и b одновременно принимают значение NULL, условие нужно изменить следующим образом:

```
WHERE a = b OR (a IS NULL AND b IS NULL)
```

Существует также специальная конструкция, позволяющая проверить эквивалентность выражений с учетом NULL: IS NOT DISTINCT FROM. С ее помощью этот пример можно переписать в виде:

```
WHERE a IS NOT DISTINCT FROM b
```

С логическими операторами дело обстоит по-другому. Иногда они возвращают не NULL, даже если аргументом является NULL. В логических операторах NULL означает неизвестное значение.

Оператор AND возвращает false, если хотя бы один операнд равен false, даже если второй при этом имеет значение NULL. Оператор OR возвращает true, если хотя бы один операнд равен true. Во всех остальных случаях результат неизвестен, т. е. принимает значение NULL:

```
car_portal=> SELECT true AND NULL, false AND NULL, true OR NULL, false OR
NULL, NOT NULL;
 ?column? | ?column? | ?column? | ?column? | ?column?
----------+----------+----------+----------+----------
          | f        | t        |          |
```

В выражении IN значения NULL трактуются неочевидным образом:

```
car_portal=> SELECT 1 IN (1, NULL) as in;
in
----
t

car_portal=> SELECT 2 IN (1, NULL) as in;
in
----
(1 row)
```

Если при вычислении выражения IN проверяемое значение не входит в список после слова IN (или в результат подзапроса), но в этом списке существует

значение NULL, то результатом будет NULL, а не false. Это можно понять, если считать NULL неизвестным значением, как в логических операторах. В первом примере ясно, что 1 входит в список, указанный в выражении IN, поэтому результат равен true. Во втором случае 2, безусловно, не равно 1, но насчет NULL нам ничего *неизвестно*. Потому-то и результат неизвестен.

Функции могут интерпретировать значения NULL по-разному – в зависимости от кода. Большинство встроенных функций возвращает NULL, если хотя бы один аргумент имеет значение NULL.

Агрегатные функции работают с NULL по-другому. Они обрабатывают много строк, а стало быть, много значений. В общем случае они игнорируют значения NULL. Функция sum вычисляет сумму всех значений, отличных от NULL, и игнорирует NULL. Если все слагаемые принимают значение NULL, то sum возвращает NULL. Точно так же ведут себя функции avg, max и min. Но не count. Функция count возвращает количество значений, отличных от NULL. Поэтому, если все значения принимают значение NULL, то count возвращает 0.

В отличие от некоторых других баз данных, в PostgreSQL пустая строка не совпадает с NULL.

Рассмотрим пример:

```
car_portal=> SELECT a IS NULL, b IS NULL, a = b FROM (SELECT ''::text a,
NULL::text b) v;
 ?column? | ?column? | ?column?
----------+----------+----------
 f        | t        |
```

Есть еще две функции, специально предназначенные для работы с NULL: COALESCE и NULLIF.

Функция COALESCE принимает произвольное число аргументов одного и того же или совместимых типов и возвращает значение первого аргумента, отличного от NULL:

```
COALESCE(a, b, c)
```

Этот код эквивалентен такому:

```
CASE WHEN a IS NOT NULL THEN a WHEN b IS NOT NULL THEN b ELSE c END
```

Функция NULLIF принимает два аргумента и возвращает NULL, если они равны. В противном случае возвращается значение первого аргумента. В некотором смысле она противоположна COALESCE.

```
NULLIF (a, b)
```

Этот код эквивалентен такому:

```
CASE WHEN a = b THEN NULL ELSE a END
```

Еще одна особенность значений NULL состоит в том, что они игнорируются ограничениями уникальности. То есть даже если некоторый столбец таблицы объявлен уникальным, все равно может существовать несколько строк, в ко-

торых он принимает значение NULL. Кроме того, индексы типа B-дерева – наиболее распространенные – не индексируют значений NULL. Рассмотрим запрос:

```
SELECT * FROM t WHERE a IS NULL
```

При его выполнении индекс, построенный по столбцу a, не используется.

Изменение данных в базе

Данные можно вставлять в таблицы, обновлять и удалять. Соответственные команды называются INSERT, UPDATE и DELETE.

Команда INSERT

Команда INSERT вставляет новые данные в таблицу. Строки всегда вставляются только в одну таблицу. Команда имеет следующий синтаксис:

```
INSERT INTO <table_name> [(<field_list>)]
{VALUES (<expression_list>)[,...]}|{DEFAULT VALUES}|<SELECT query>;
```

Имя таблицы, в которую вставляются записи, указывается после ключевых слов INSERT INTO. Существует два синтаксически различных варианта команды INSERT: для вставки одной или нескольких отдельных записей и для вставки целого набора записей.

Для вставки одной или нескольких записей используется ключевое слово VALUES, за которым следует список значений. Элементы списка соответствуют полям таблицы в порядке их следования. Если заполняются не все поля, то имена заполняемых полей должны быть указаны в скобках после имени таблицы. Те поля, что не указаны, получат значения по умолчанию, если таковые определены, или будут установлены в NULL.

Количество элементов в списке VALUES должно быть таким же, как количество полей после имени таблицы:

```
car_portal=> INSERT INTO car_portal_app.a (a_int) VALUES (6);
INSERT 0 1
```

Результатом успешно выполненной команды INSERT является слово INSERT, за которым следует OID вставленной строки (если вставлена только одна строка и для таблицы включены OID, в противном случае 0) и количество вставленных строк. Дополнительные сведения об OID см. в главе 12.

Другой способ записать в поле значение по умолчанию – воспользоваться ключевым словом DEFAULT в списке VALUES. Если для поля не определено значение по умолчанию, то будет записано значение NULL:

```
INSERT INTO car_portal_app.a (a_text) VALUES (default);
```

Есть также возможность записать во все поля значения по умолчанию, для этого нужно воспользоваться ключевыми словами DEFAULT VALUES:

```
INSERT INTO car_portal_app.a DEFAULT VALUES;
```

Синтаксическая форма с VALUES позволяет вставить сразу несколько записей:

```
INSERT INTO car_portal_app.a (a_int, a_text) VALUES (7, 'seven'), (8,'eight');
```

Это специфика PostgreSQL. Некоторые другие базы данных дают возможность вставлять только по одной записи.

На самом деле в PostgreSQL VALUES – это отдельная SQL-команда. Поэтому ее можно использовать в качестве подзапроса в любом запросе SELECT:

```
car_portal=> SELECT * FROM (VALUES (7, 'seven'), (8, 'eight')) v;
 column1 | column2
---------+---------
       7 | seven
       8 | eight
(2 rows)
```

Если вставляемые записи берутся из другой таблицы или представления, то используется форма с подзапросом SELECT, а не с фразой VALUES:

```
INSERT INTO car_portal_app.a SELECT * FROM car_portal_app.b;
```

Результат запроса должен соответствовать структуре таблицы: иметь такое же количество столбцов совместимых типов в таком же порядке.

В запросе SELECT допустимо ссылаться на таблицу, в которую вставляются записи. Например, чтобы продублировать все записи таблицы, можно выполнить такую команду:

```
INSERT INTO car_portal_app.a SELECT * FROM car_portal_app.a;
```

По умолчанию команда INSERT возвращает количество вставленных записей. Но можно также вернуть сами вставленные записи или некоторые их поля (в том же формате, что в случае запроса SELECT). Для этого используется ключевое слово RETURNING, за которым следует список подлежащих возврату полей:

```
car_portal=> INSERT INTO car_portal_app.a SELECT * FROM car_portal_app.b
RETURNING a_int;
 a_int
-------
     2
     3
     4
(3 rows)

INSERT 0 3
```

Если для таблицы, в которую производится вставка, определено ограничение уникальности, то при попытке вставить конфликтующие записи произойдет ошибка. Однако можно сказать, чтобы INSERT игнорировала такие записи или обновляла их вместо вставки.

В предположении, что для таблицы b существует ограничение уникальности, рассмотрим вставку в поле b_int:

```
car_portal=> INSERT INTO b VALUES (2, 'new_two');
ERROR: duplicate key value violates unique constraint "b_b_int_key"
DETAIL: Key (b_int)=(2) already exists.
```

Но что, если мы хотим изменить запись, если она уже существует? Тогда поступим так:

```
car_portal=> INSERT INTO b VALUES (2, 'new_two')
ON CONFLICT (b_int) DO UPDATE SET b_text = excluded.b_text
RETURNING *;
 b_int | b_text
-------+---------
     2 | new_two
(1 row)
INSERT 0 1
```

Здесь фраза `ON CONFLICT` говорит, что нужно сделать в случае, когда уже существует запись с таким же значением в поле `b_int`. Псевдоним `excluded` относится к вставляемым значениям. Чтобы сослаться на значения, уже существующие в таблице, нужно было бы указать имя таблицы.

В стандарте SQL определена специальная команда `MERGE`, реализующая аналогичную функциональность. Но в PostgreSQL фраза `ON CONFLICT` является частью синтаксиса `INSERT`. В других РСУБД это может быть реализовано по-другому.

Команда UPDATE

Команда `UPDATE` применяется для изменения данных в строках таблицы без изменения их количества. Вот ее синтаксис:

```
UPDATE <table_name>
SET <field_name> = <expression>[, ...]
[FROM <table_name> [JOIN clause]]
[WHERE <condition>];
```

Использовать команду `UPDATE` можно двумя способами. Первый похож на простую команду `SELECT` и называется **подвыборкой**. Второй служит для обновления данных в одной таблице, исходя из данных в других таблицах, и похож на команду `SELECT` из нескольких таблиц. В большинстве случаев нужного результата можно добиться любым из этих способов.

В PostgreSQL за одну операцию можно обновить только одну таблицу. В других базах данных при некоторых условиях допускается одновременное обновление нескольких таблиц.

UPDATE с подвыборкой

Выражение для задания нового значения является обычным SQL-выражением. В этом выражении допускается ссылка на обновляемое поле, и вместо него подставляется старое значение:

```
UPDATE t SET f = f+1 WHERE a = 5;
```

Часто в командах UPDATE употребляются подзапросы. Чтобы сослаться из подзапроса на обновляемую таблицу, ей необходимо сопоставить псевдоним:

```
car_portal=> UPDATE car_portal_app.a updated SET a_text =
  (SELECT b_text FROM car_portal_app.b WHERE b_int = updated.a_int);
UPDATE 7
```

Если подзапрос не возвращает результата, то в поле будет записано NULL.

Обратите внимание, что результатом команды UPDATE является слово UPDATE, за которым следует количество обновленных записей.

Фраза WHERE аналогична используемой в команде SELECT. Если она не задана, то обновляются все записи.

UPDATE с дополнительными таблицами

Другой способ обновить строки таблицы – воспользоваться фразой FROM так же, как это делается в команде SELECT:

```
UPDATE car_portal_app.a SET a_int = b_int FROM car_portal_app.b
WHERE a.a_text=b.b_text;
```

Обновляются все строки a, для которых в b существуют строки с таким же значением в текстовом поле. Новое значение числового поля берется из таблицы b. Технически это не что иное, как внутреннее соединение двух таблиц. Однако синтаксис отличается. Поскольку таблица a не является частью фразы FROM, то обычный синтаксис соединения здесь неприменим, и таблицы соединяются по условию во фразе WHERE. Если бы использовалась еще одна таблица, то ее можно было бы соединить с b, применив обычный синтаксис соединения, внутреннего или внешнего.

Во многих случаях синтаксис команды UPDATE с FROM выглядит более понятно. Например, следующая команда вносит в таблицу те же самые изменения, что предыдущая, но не так очевидна:

```
UPDATE car_portal_app.a
SET a_int = (SELECT b_int FROM car_portal_app.b WHERE a.a_text=b.b_text)
WHERE a_text IN (SELECT b_text FROM car_portal_app.b);
```

Еще одно преимущество формы с FROM состоит в том, что зачастую она работает гораздо быстрее. С другой стороны, можно получить непредсказуемые результаты, если одной записи обновляемой таблицы соответствует несколько записей из таблиц, указанных во фразе FROM:

```
UPDATE car_portal_app.a SET a_int = b_int FROM car_portal_app.b;
```

Этот запрос синтаксически корректен. Однако известно, что в таблице b несколько записей. Какая из них будет выбрана для обновления каждой строки, не определено, потому что условие WHERE отсутствует. Та же проблема имеет место, если фраза WHERE не определяет взаимно-однозначное соответствие.

```
car_portal=> UPDATE car_portal_app.a SET a_int = b_int FROM
car_portal_app.b WHERE b_int>=a_int;
UPDATE 6
```

Для каждой записи таблицы a существует более одной записи в таблице b, для которой b_int больше или равно a_int. Поэтому результат такого обновления не определен. Однако же PostgreSQL не запрещает такие команды, так что следует проявлять осторожность.

Команда обновления может вернуть обновленные записи, если присутствует фраза RETURNING, как в INSERT:

```
car_portal=> UPDATE car_portal_app.a SET a_int = 0 RETURNING *;
 a_int | a_text
-------+--------
     0 | one
     0 | two
...
```

Команда DELETE

Команда DELETE служит для удаления записей из базы. Как и в случае UPDATE, есть два способа удаления: с помощью подвыборки и с использованием других таблиц, одной или нескольких. Форма с подвыборкой имеет такой синтаксис:

```
DELETE FROM <table_name> [WHERE <condition>];
```

Записи, удовлетворяющие условию condition, будут удалены из таблицы. Если фраза WHERE опущена, то удаляются все записи.

Форма DELETE с использованием другой таблицы аналогична форме UPDATE с фразой FROM. Только вместо FROM следует использовать ключевое слово USING, потому что FROM уже занято в другой части команды DELETE:

```
car_portal=> DELETE FROM car_portal_app.a USING car_portal_app.b
WHERE a.a_int=b.b_int;
DELETE 6
```

Эта команда удаляет из таблицы a все записи, для которых в таблице b существует запись с таким же значением числового поля. В результате выполнения команды сообщается, сколько записей было удалено.

Показанная выше команда эквивалентна такой:

```
DELETE FROM car_portal_app.a WHERE a_int IN
  (SELECT b_int FROM car_portal_app.b);
```

Как и команды UPDATE и INSERT, команда DELETE возвращает удаленные строки, если присутствует ключевое слово RETURNING:

```
car_portal=> DELETE FROM car_portal_app.a RETURNING *;
 a_int | a_text
-------+--------
     0 | one
     0 | two
...
```

Команда TRUNCATE

Существует еще одна команда, которая изменяет данные, оставляя неизменной структуру таблицы, – TRUNCATE. Она очищает таблицу целиком и почти мгновенно. Эффект точно такой же, как при использовании команды DELETE без фразы WHERE. Особенно полезна она для больших таблиц.

```
car_portal=> TRUNCATE TABLE car_portal_app.a;
TRUNCATE TABLE
```

Резюме

Язык SQL используется для взаимодействия с базой данных: создания и обслуживания структур данных, добавления данных, их изменения, удаления и выборки. В SQL имеется три подъязыка: DDL (язык определения данных), DML (язык манипулирования данными) и DCL (язык управления данными). В этой главе описаны четыре команды, составляющих язык DML: SELECT, INSERT, UPDATE и DELETE.

На примере команды SELECT мы детально рассмотрели такие концепции SQL, как группировка и фильтрация с использованием SQL-выражений и условий, а также объяснили, как используются подзапросы. Кроме того, в разделе, посвященном соединению таблиц, мы затронули ряд вопросов реляционной алгебры.

В следующих главах будут затронуты более сложные темы: ряд дополнительных возможностей и приемов работы с SQL, а также язык программирования PL/pgSQL, на котором можно писать функции.

Глава 6

Дополнительные сведения о написании запросов

В этой главе мы обсудим некоторые продвинутые средства SQL, поддерживаемые PostgreSQL, а также ряд приемов написания запросов. Будут рассмотрены следующие темы:

- общие табличные выражения:
- оконные функции;
- продвинутые методы работы с SQL.

Мы будем использовать ту же демонстрационную базу `car_portal`, что и в предыдущих главах. Рекомендуется пересоздать базу, чтобы результаты не отличались от показанных в тексте книги. Скрипты для создания и заполнения базы (`schema.sql` и `data.sql`) находятся в коде, сопровождающем эту главу. Все примеры кода можно найти в файле `examples.sql`.

Общие табличные выражения

Хотя SQL – декларативный язык, в нем есть средства для последовательного выполнения и для повторного использования кода.

Общие табличные выражения (common table expressions – **CTE**) позволяют один раз определить подзапрос и присвоить ему имя, а затем использовать его в нескольких местах главного запроса.

Ниже приведена упрощенная синтаксическая диаграмма CTE:

```
WITH <subquery name> AS (<subquery code>) [, ...]
SELECT <Select list> FROM <subquery name>;
```

Здесь `subquery code` – запрос, результаты которого будут впоследствии использованы в главном запросе, как будто это физическая таблица. Подзапрос в скобках после ключевого слова `AS` и есть общее табличное выражение. Его можно также назвать подкомандой или вспомогательной командой. После блока `WITH` находится главный запрос. Команда в целом называется `WITH`-запросом.

В CTE можно использовать не только команды `SELECT`, но также `INSERT`, `UPDATE` и `DELETE`. В одном `WITH`-запросе может быть несколько CTE. Каждому CTE сопо-

ставляется имя, расположенное перед ключевым словом AS. Главный запрос может ссылаться на CTE по этому имени. Одно CTE может ссылаться на другое по имени. Однако разрешается ссылаться только на CTE, определенные раньше.

Ссылки на CTE в главном запросе можно трактовать как имена таблиц. На самом деле PostgreSQL выполняет CTE только один раз, кеширует результаты, а затем повторно использует, вместо того чтобы выполнять подзапросы всякий раз, как они встречаются в главном запросе. Таким образом, CTE действительно становятся похожи на таблицы.

CTE помогают организовать SQL-код. Пусть требуется найти в базе модели, выпущенные после 2010 года, и вычислить для них минимальное количество прежних владельцев. Рассмотрим такой код:

```
car_portal=> WITH pre_select AS
(
  SELECT car_id, number_of_owners, car_model_id
    FROM car_portal_app.car WHERE manufacture_year >= 2010
),
joined_data AS
(
  SELECT car_id, make, model, number_of_owners
    FROM pre_select
      INNER JOIN car_portal_app.car_model ON pre_select.car_model_id=
  car_model.car_model_id
),
minimal_owners AS (SELECT min(number_of_owners) AS min_number_of_owners
FROM pre_select)
SELECT car_id, make, model, number_of_owners
  FROM joined_data INNER JOIN minimal_owners
    ON joined_data.number_of_owners = minimal_owners.min_number_of_owners;
 car_id | make          | model      | number_of_owners
--------+---------------+------------+------------------
      2 | Opel          | Corsa      | 1
      3 | Citroen       | C3         | 1
     11 | Nissan        | GT-R       | 1
     36 | KIA           | Magentis   | 1
…
(25 rows)
```

Здесь логическая часть запроса представлена в виде последовательности действий: фильтрация в pre_select и соединение в joined_data. Вторая часть – вычисление количества владельцев – выполняется в отдельном подзапросе, minimal_owners. Таким образом, реализация логики запроса оказывается похожа на императивный язык программирования.

Использование CTE в этом примере не ускоряет выполнения запроса. Однако бывают случаи, когда CTE повышают производительность. Более того, иногда без CTE написать запрос вообще невозможно. В следующих разделах мы рассмотрим эти ситуации подробнее.

Порядок выполнения CTE не определен. Цель PostgreSQL – выполнить главный запрос. Если в нем есть ссылки на CTE, то PostgreSQL выполняет их сначала. Если на CTE типа SELECT нет ни прямых, ни косвенных ссылок из главного запроса, то оно не выполняется вовсе. Подкоманды, изменяющие данные, выполняются всегда.

CTE как средство повторного использования SQL-кода

Если выполнение подзапроса занимает много времени и этот подзапрос встречается в команде несколько раз, то имеет смысл оформить его в виде CTE и использовать результаты повторно. Это ускоряет выполнение запроса, потому что PostgreSQL исполняет CTE только один раз, кеширует результаты в памяти или на диске – в зависимости от размера, а затем использует повторно.

Пусть требуется найти относительно новые модели. Для этого нужно вычислить средний возраст автомобилей каждой модели, а затем выбрать те модели, для которых средний возраст меньше, чем средний возраст по всем моделям.

Это можно сделать так:

```
car_portal=> SELECT make, model, avg_age FROM
(
  SELECT car_model_id, avg(EXTRACT(YEAR FROM now())-manufacture_year) AS avg_age
    FROM car_portal_app.car
    GROUP BY car_model_id
) age_subq1
INNER JOIN car_portal_app.car_model ON car_model.car_model_id =
age_subq1.car_model_id
WHERE avg_age < (SELECT avg(avg_age) FROM
(
  SELECT avg(EXTRACT(YEAR FROM now()) - manufacture_year) avg_age
  FROM car_portal_app.car
  GROUP BY car_model_id
) age_subq2
);
 make          | model        | avg_age
---------------+--------------+-------------
 BMW           | 1er          |           1
 BMW           | X6           |         2.5
 Mercedes Benz | A klasse     |         1.5
…
(41 rows)
```

Функция EXTRACT возвращает целочисленное значение указанной части даты. В этом запросе она используется, чтобы выделить год из текущей даты. Разность между текущим годом и годом выпуска manufacture_year и есть возраст машины.

В этом запросе есть два похожих подзапроса: опрашивается одна и та же таблица и производятся одинаковые группировка и агрегирования. Поэтому мы можем получить выигрыш от использования CTE:

```
car_portal=> WITH age_subq AS
(
  SELECT car_model_id, avg(EXTRACT(YEAR FROM now())-manufacture_year) AS avg_age
    FROM car_portal_app.car
    GROUP BY car_model_id
)
SELECT make, model, avg_age
  FROM age_subq
    INNER JOIN car_portal_app.car_model ON car_model.car_model_id =
age_subq.car_model_id
  WHERE avg_age < (SELECT avg(avg_age) FROM age_subq);
 make          | model        | avg_age
---------------+--------------+-------------
 BMW           | 1er          |           1
 BMW           | X6           |         2.5
 Mercedes Benz | A klasse     |         1.5
...
(41 rows)
```

Оба запроса дают одинаковый результат. Но первый запрос выполнялся 1.9 миллисекунды, а второй – 1.0 миллисекунды. Конечно, в абсолютных цифрах разница ничтожна, но в относительных – `WITH`-запрос почти в два раза быстрее. Если бы мы обрабатывали миллионы записей, то и абсолютная разница оказалась бы существенной.

Еще одно преимущество CTE в данном случае – более компактный и простой для понимания код. И это тоже аргумент в пользу `WITH` – длинные и сложные запросы можно переписать в виде CTE ради чистоты и понятности кода, пусть даже это не влияет на производительность.

Но иногда CTE лучше не использовать. Например, может возникнуть мысль заранее выбрать из таблицы некоторые столбцы в надежде, что это ускорит выполнение запроса, т. к. базе придется обрабатывать меньше информации. В таком случае запрос мог бы выглядеть следующим образом:

```
WITH car_subquery AS
  (SELECT number_of_owners, manufacture_year, number_of_doors
   FROM car_portal_app.car)
SELECT number_of_owners, number_of_doors FROM car_subquery
WHERE manufacture_year = 2008;
```

Эффект получается противоположным ожидаемому. PostgreSQL не применяет фразу `WHERE` из главного запроса к подкоманде. База данных выберет из таблицы все вообще записи, возьмет из каждой по три столбца и сохранит их во временном наборе данных в памяти. Затем этот временный набор будет профильтрован по предикату `manufacture_year = 2008`. Если бы по столбцу `manufacture_year` был построен индекс, то сервер не стал бы его использовать, потому что опрашивается временная, а не настоящая таблица.

Поэтому следующий запрос выполняется в пять раз быстрее предыдущего, хотя они вроде бы почти ничем не отличаются:

```
SELECT number_of_owners, manufacture_year, number_of_doors
FROM car_portal_app.car
WHERE manufacture_year = 2008;
```

Рекурсивные и иерархические запросы

CTE может ссылаться на себя же. Такие команды называются рекурсивными запросами. У этих запросов своеобразная структура, сообщающая базе данных об их характере:

```
WITH RECURSIVE <subquery_name> (<field list>) AS
(
    <non-recursive term>
    UNION [ALL|DISTINCT]
    <recursive term>
)
[,...]
<main query>
```

Обе части – нерекурсивная и рекурсивная – подзапросы, которые должны возвращать одинаковое число полей одинаковых типов. Имена полей задаются в объявлении рекурсивного запроса в целом, поэтому не важно, как они названы в подзапросах.

Нерекурсивную часть называют также якорным подзапросом, а рекурсивную – итеративным подзапросом.

Якорный подзапрос – отправная точка для выполнения рекурсивного запроса. Он не может ссылаться на имя рекурсивного запроса и выполняется только один раз. Результаты якорного подзапроса передаются назад тому же самому CTE, после чего выполняется рекурсивная часть. Такие итерации повторяются до тех пор, пока результат рекурсивного подзапроса не окажется пустым. Результатом всего запроса является объединение множества строк, возвращенных нерекурсивной частью и всеми итерациями рекурсивной части. Если присутствует фраза `UNION ALL`, то возвращаются все эти строки, а если `UNION DISTINCT` или просто `UNION`, то дубликаты удаляются.

Отметим, что алгоритм выполнения рекурсивного запроса подразумевает итерацию, а не рекурсию. Однако в стандарте SQL такие запросы называются рекурсивными. В других базах данных та же логика реализуется аналогичным способом, но синтаксис может немного отличаться.

Ниже показано, как можно применить рекурсивный запрос для вычисления факториала:

```
car_portal=> WITH RECURSIVE subq (n, factorial) AS
(
  SELECT 1, 1
  UNION ALL
  SELECT n + 1, factorial * (n + 1) from subq WHERE n < 5
)
SELECT * FROM subq;
 n | factorial
---+-----------
```

```
 1 |          1
 2 |          2
 3 |          6
 4 |         24
 5 |        120
(5 rows)
```

Здесь `SELECT 1, 1` – якорный подзапрос. Он возвращает одну строку (оба поля `n` и `factorial` равны 1), которая передается следующей итерации. На первой итерации к значению `n` прибавляется единица, и значение `factorial` умножается на `(n+1)`, так что получается 2 и 2. Эта строка передается следующей итерации, на которой вычисляются значения 3 и 6. И так далее. На последней итерации возвращается строка, в которой поле `n` равно 5.

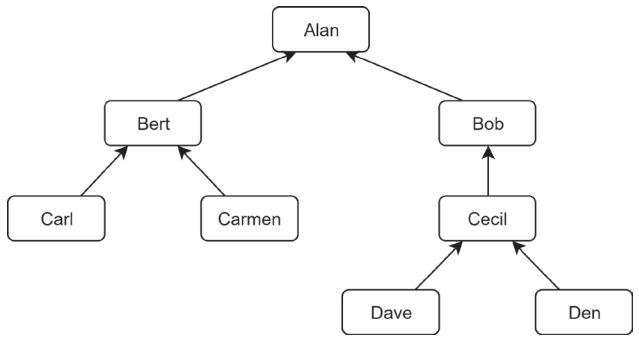
Эта строка отфильтровывается фразой `WHERE` итеративного подзапроса, поэтому на следующей итерации не возвращается ничего, и выполнение останавливается. Таким образом, весь рекурсивный запрос возвращает список из пяти чисел от 1 до 5 и соответствующие им факториалы.

Если бы фразы `WHERE` не было, то выполнение никогда не закончилось бы, что рано или поздно привело бы к ошибке переполнения.

Этот пример нетрудно реализовать без всяких рекурсивных запросов. В PostgreSQL есть способ генерировать последовательности чисел и использовать их в подзапросах. Но существуют задачи, которые без рекурсивных запросов не решаются, – это запросы к иерархии объектов.

В базе данных для сайта торговли автомобилями иерархических данных нет, поэтому для иллюстрации нам понадобятся тестовые данные. Типичная иерархия подразумевает наличие связи родитель-потомок между объектами, когда один и тот же объект может одновременно быть родителем и потомком.

Рассмотрим семью. У Алана два ребенка: Берт и Боб. У Берта тоже два ребенка: Карл и Кармен. У Боба один ребенок, Сесиль, у которой два ребенка: Дэйв и Дэн. Генеалогическое древо показано на рисунке ниже. Стрелки направлены от ребенка к родителю.



Иерархическая связь

В базе данных сведения об иерархии можно хранить в простой таблице с двумя столбцами: person и parent. Первый столбец будет первичным ключом, второй – внешним ключом, ссылающимся на ту же таблицу:

```
car_portal=>CREATE TABLE family (person text PRIMARY KEY, parent text
REFERENCES family);
CREATE TABLE
```

```
car_portal=> INSERT INTO family VALUES ('Alan', NULL),
  ('Bert', 'Alan'), ('Bob', 'Alan'), ('Carl', 'Bert'), ('Carmen', 'Bert'),
  ('Cecil', 'Bob'),
  ('Dave', 'Cecil'), ('Den', 'Cecil');
INSERT 0 8
```

В первой вставленной записи поле parent равно NULL, это означает, что о родителе Алана у нас нет информации.

Пусть требуется построить полные родословные всех детей. Сделать это путем соединения таблиц невозможно, потому что каждое соединение описывает только один уровень иерархии, а в общем случае количество уровней неизвестно.

Задачу решает следующий рекурсивный запрос:

```
car_portal=> WITH RECURSIVE genealogy (bloodline, person, level) AS
(
  SELECT person, person, 0 FROM family WHERE parent IS NULL
  UNION ALL
  SELECT g.bloodline || ' -> ' || f.person, f.person, g.level + 1
    FROM family f, genealogy g WHERE f.parent = g.person
)
SELECT bloodline, level FROM genealogy;
           bloodline           | level
-------------------------------+-------
 Alan                          |     0
 Alan -> Bert                  |     1
 Alan -> Bob                   |     1
 Alan -> Bert -> Carl          |     2
 Alan -> Bert -> Carmen        |     2
 Alan -> Bob -> Cecil          |     2
 Alan -> Bob -> Cecil -> Dave  |     3
 Alan -> Bob -> Cecil -> Den   |     3
(8 rows)
```

В нерекурсивной части выбирается начало иерархии, т. е. человек, не имеющий родителя. Его имя будет стоять в начале каждой родословной. На первой итерации рекурсивного подзапроса выбираются его дети (те, в которых в поле parent находится значение поля person, выбранное нерекурсивным подзапросом). Их имена добавляются в конец родословной через разделитель ->. На второй итерации выбираются дети детей и т. д. Если детей не найдено, выполнение останавливается. Значение в поле level (уровень) увеличивается на 1 на каждой итерации, чтобы можно было показать номер итерации в результатах.

При выполнении таких иерархических запросов может возникнуть проблема. Если в данных имеются циклы, то рекурсивный запрос, написанный, как показано выше, никогда не остановится. Давайте, к примеру, изменим одну запись в таблице `family`:

```
UPDATE family SET parent = 'Bert' WHERE person = 'Alan';
```

Теперь в данных имеется цикл: Алан является ребенком собственного ребенка. Если снова запустить запрос для построения родословной, то он будет крутиться в цикле, пока в конечном итоге не завершится ошибкой[1]. Чтобы воспользоваться этим запросом, нужно как-то заставить его остановиться. Для этого можно проверять, был ли человек, обрабатываемый рекурсивным подзапросом, уже включен в родословную:

```
car_portal=> WITH RECURSIVE genealogy (bloodline, person, level, processed)
AS
(
  SELECT person, person, 0, ARRAY[person] FROM family WHERE person = 'Alan'
  UNION ALL
  SELECT g.bloodline || ' -> ' || f.person, f.person, g.level + 1,
         processed || f.person
  FROM family f, genealogy g
  WHERE f.parent = g.person AND NOT f.person = ANY(processed)
)
SELECT bloodline, level FROM genealogy;
           bloodline          | level
------------------------------+-------
 Alan                         |     0
 Alan -> Bert                 |     1
 Alan -> Bob                  |     1
 Alan -> Bert -> Carl         |     2
 Alan -> Bert -> Carmen       |     2
 Alan -> Bob -> Cecil         |     2
 Alan -> Bob -> Cecil -> Dave |     3
 Alan -> Bob -> Cecil -> Den  |     3
(8 rows)
```

Результат получился такой же, как в предыдущем примере. Поле `processed` используется в CTE, но не выбирается в главном запросе. На самом деле в нем хранятся те же данные, что в поле `bloodline`, но в такой форме, чтобы их было удобно использовать во фразе `WHERE`. На каждой итерации в массив добавляется имя только что обработанного человека. Кроме того, в условии `WHERE` в рекурсивном подзапросе дополнительно проверяется, что обрабатываемый человек отсутствует в массиве `processed`.

---

[1] На самом деле запрос сразу же завершится, потому что нерекурсивная часть возвращает пустое множество, а значит, рекурсивная не выполняется. Чтобы зациклить запрос, нужно изменить условие отбора в нерекурсивной части на person = 'Alan'. – *Прим. перев.*

На рекурсивные запросы накладываются кое-какие ограничения. Например, в рекурсивной части не должно быть агрегирования, а на имя рекурсивного запроса разрешается ссылаться только в рекурсивной части.

Изменение данных сразу в нескольких таблицах

Еще одно полезное применение CTE – выполнение сразу нескольких команд изменения данных. Для этого нужно включить команды `INSERT`, `UPDATE` и `DELETE` в CTE. Результаты этих команд можно передать последующим CTE или главному запросу, если добавить фразу `RETURNING`. Как и в случае команд `SELECT`, количество общих табличных выражений, изменяющих данные, не ограничено.

Допустим, требуется добавить в базу данных новый автомобиль, но соответствующей модели нет в таблице `car_model`. Тогда нужно добавить новую запись в `car_model`, взять идентификатор этой записи и вставить этот идентификатор в запись, добавляемую в таблицу `car`:

```
car_portal=> INSERT INTO car_portal_app.car_model (make, model) VALUES
('Ford','Mustang') RETURNING car_model_id;
 car_model_id
--------------
          100
(1 row)
INSERT 0 1

car_portal=> INSERT INTO car_portal_app.car (number_of_owners,
registration_number, manufacture_year, number_of_doors, car_model_id,
mileage)
VALUES (1, 'GTR1231', 2014, 4, 100, 10423);
INSERT 0 1
```

Но иногда неудобно выполнять две команды и где-то сохранять промежуточный идентификатор. `WITH`-запрос позволяет внести изменения в обе таблицы за одну операцию:

```
car_portal=# WITH car_model_insert AS
(
  INSERT INTO car_portal_app.car_model (make, model) VALUES ('Ford','Mustang')
  RETURNING car_model_id
)
INSERT INTO car_portal_app.car
(number_of_owners, registration_number, manufacture_year,
number_of_doors, car_model_id, mileage)
SELECT 1, 'GTR1231', 2014, 4, car_model_id, 10423 FROM car_model_insert;
INSERT 0 1
```

Как уже было сказано, CTE, изменяющие данные, выполняются всегда – не важно, упоминаются они в главном запросе или нет. Однако порядок их выполнения не определен. Но на него можно повлиять, если сделать одно CTE зависящим от другого.

Что, если несколько CTE изменяют одну и ту же таблицу или одно использует результаты другого? Имеется несколько принципов изоляции и взаимодействия.

- Для подкоманд:
  - все подкоманды работают с теми данными, которые существовали на момент начала всего WITH-запроса;
  - они не видят результатов друг друга. Например, подкоманда DELETE не может удалить строку, вставленную какой-то подкомандой INSERT;
  - единственный способ передать информацию об обработанных записях из одного CTE в другое – воспользоваться фразой RETURNING.
- Для триггеров, определенных над изменяемыми таблицами:
  - для триггеров BEFORE. Триггеры уровня команды выполняются сразу после выполнения всех подкоманд. Триггеры уровня строки выполняются непосредственно перед изменением каждой записи. Это означает, что триггер уровня строки для одной подкоманды может быть выполнен раньше триггера уровня команды для другой подкоманды, даже если изменяется одна и та же таблица;
  - для триггеров AFTER. Триггеры уровня команды и уровня строки выполняются после всего WITH-запроса. Они выполняются группами для каждой подкоманды: сначала триггеры уровня строки, потом уровня команды. Это означает, что триггер уровня команды для одной подкоманды может быть выполнен раньше триггера уровня строки для другой подкоманды, даже если изменяется одна и та же таблица;
  - команды внутри кода триггеров видят изменения данных, произведенные другими подкомандами.
- Для ограничений, определенных над изменяемыми таблицами в предположении, что в них отсутствует слово DEFERRED:
  - ограничения PRIMARY KEY и UNIQUE проверяются для каждой записи в момент вставки или обновления. При этом учитываются изменения, произведенные другими подкомандами;
  - ограничения CHECK для каждой записи для администраторов в момент вставки или обновления. При этом изменения, произведенные другими подкомандами, не учитываются;
  - ограничения FOREIGN KEY проверяются в конце выполнения всего WITH-запроса.

Приведем простой пример зависимости и взаимодействия между CTE:

```
car_portal=> CREATE TABLE t (f int UNIQUE);
CREATE TABLE

car_portal=> INSERT INTO t VALUES (1);
INSERT 0 1

car_portal=> WITH del_query AS (DELETE FROM t) INSERT INTO t VALUES (1);
ERROR: duplicate key value violates unique constraint "t_f_key"
```

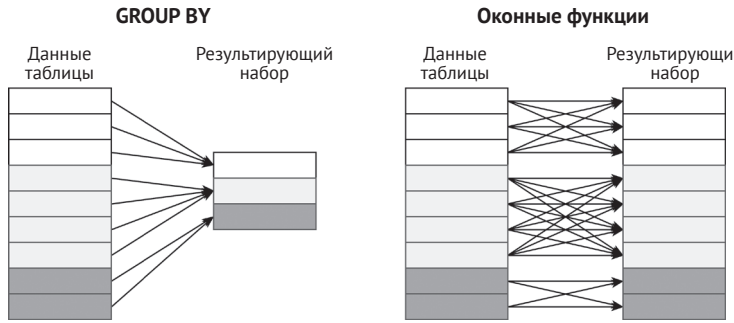
Последний запрос завершился неудачно, потому что PostgreSQL попыталась выполнить главный запрос раньше CTE. Если создать зависимость, которая заставит сначала выполнить CTE, то старая запись будет удалена, а новая вставлена. Тогда ограничение не будет нарушено.

```
car_portal=> WITH del_query AS (DELETE FROM t RETURNING f)
INSERT INTO t SELECT 1 WHERE (SELECT count(*) FROM del_query) IS NOT NULL;
INSERT 0 1
```

В этом коде условие `WHERE` в главном запросе не имеет практического значения, потому что результат `count` никогда не равен `NULL`. Однако коль скоро CTE упоминается в запросе, оно выполняется раньше главного запроса.

Оконные функции

Помимо группировки и агрегирования, PostgreSQL предлагает еще один способ вычислений, затрагивающих несколько записей, – **оконные функции**. В случае группировки и агрегирования выводится одна запись для каждой группы входных записей. Оконные функции делают нечто похожее, но выполняются для каждой записи, а количество записей на входе и на выходе одинаково.



GROUP BY и оконные функции

На этом рисунке прямоугольниками представлены записи таблицы. Будем считать, что цвет прямоугольника обозначает значение поля, по которому производится группировка записей. При использовании `GROUP BY` для каждого нового значения создается группа, которой соответствует одна запись в результате запроса. Мы объясняли это в главе 5. Оконные функции дают доступ ко всем записям группы (которая в этом случае называется разделом), а количество записей на выходе такое же, как на входе. При использовании оконных функций группировка не обязательна, хотя и возможна.

Оконные функции вычисляются после группировки и агрегирования. Поэтому в запросе `SELECT` они могут находиться только в списке выборки и во фразе `ORDER BY`.

Определение окна

Синтаксис оконных функций выглядит так:

```
<function_name> (<function_arguments>)
OVER(
  [PARTITION BY <expression_list>]
  [ORDER BY <order_by_list>]
  [{ROWS | RANGE} <frame_start> |
  {ROWS | RANGE} BETWEEN <frame_start> AND <frame_end>])
```

Конструкция в скобках после слова OVER называется **определением окна**. Последняя его часть, начинающаяся словом ROWS, называется **определением фрейма**. Синтаксис частей frame_start и frame_end будет описан позже.

Вообще говоря, оконные функции похожи на агрегатные. Они обрабатывают наборы записей. Эти наборы строятся отдельно для каждой обрабатываемой записи. Потому-то оконные функции, в отличие от агрегатных, вычисляются для каждой строки. Набор записей, обрабатываемых оконной функцией, строится следующим образом.

Вначале обрабатывается фраза PARTITION BY. Отбираются все записи, для которых выражения, перечисленные в списке expression_list, имеют такие же значения, как в текущей записи. Это множество строк называется **разделом** (partition). Текущая строка также включается в раздел. В общем-то, фраза PARTITION BY логически и синтаксически идентична фразе GROUP BY, только в ней запрещается ссылаться на выходные столбцы по именам или по номерам.

Иными словами, при обработке каждой записи оконная функция просматривает все остальные записи, чтобы понять, какие из них попадают в один раздел с текущей. Если фраза PARTITION BY отсутствует, то на данном шаге создается один раздел, содержащий все записи.

Далее раздел сортируется в соответствии с фразой ORDER BY, логически и синтаксически идентичной той же фразе в команде SELECT. И снова ссылки на выходные столбцы по именам или по номерам не допускаются. Если фраза ORDER BY опущена, то считается, что позиция каждой записи множества одинакова.

Наконец, обрабатывается определение фрейма[1]. Это означает, что мы берем подмножество раздела, которое следует передать оконной функции. Оно и называется **оконным фреймом**. У фрейма есть начало и конец. Начало фрейма, которое в синтаксической диаграмме названо frame_start, может принимать следующие значения:

- UNBOUNDED PRECEDING: самая первая запись раздела;
- <N> PRECEDING: запись, предшествующая текущей в упорядоченном разделе и отстоящая от нее на N записей. Здесь <N> – целочисленное выражение,

[1] Здесь слово «фрейм» употребляется в том же смысле, что тег <FRAME> в языке HTML, и обозначает секцию окна (если следовать метафоре физического окна, разделенного деревянными или металлическими элементами на прямоугольные секции). Чтобы не плодить сущности, употребляется тот же перевод. – *Прим. перев.*

которое не может быть отрицательным и не может содержать агрегатные или другие оконные функции. 0 PRECEDING указывает на текущую запись;

- CURRENT ROW: текущая строка;
- <N> FOLLOWING: запись, следующая за текущей в упорядоченном разделе и отстоящая от нее на N записей.

Конец фрейма, frame_end, может принимать следующие значения:

- <N> PRECEDING;
- CURRENT ROW;
- <N> FOLLOWING;
- UNBOUNDED FOLLOWING: последняя запись раздела.

Начало должно предшествовать концу. Поэтому спецификация ROWS BETWEEN CURRENT ROW AND 1 PRECEDING некорректна.

Оконный фрейм можно определить в режиме ROWS или RANGE. Режим влияет на семантику CURRENT ROW. В режиме ROWS CURRENT ROW указывает на саму текущую запись, а в режиме RANGE – на первую или последнюю запись с такой же позицией, что у текущей, в смысле сортировки, заданной фразой ORDER BY.

В режиме RANGE в определении фрейма можно использовать варианты UNBOUNDED ... или CURRENT ROW, но не <N> PRECEDING.

Если часть frame_end опущена, предполагается CURRENT ROW.

Если опущено все определение фрейма, то предполагается определение RANGE UNBOUNDED PRECEDING.

```
OVER (PARTITION BY a ORDER BY b ROWS BETWEEN UNBOUNDED PRECEDING AND 5 FOLLOWING)
```

Оно означает, что для каждой строки раздел образуют все записи с одинаковым значением в поле a. Затем раздел сортируется в порядке возрастания поля b, а фрейм содержит все записи от первой до пятой после текущей строки.

Фраза WINDOW

Определения окон могут быть довольно длинными, и часто использовать их в списке выборки неудобно. PostgreSQL предлагает способ определять и именовать окна, которые затем используются во фразе OVER оконной функции. Это делается с помощью фразы WINDOW команды SELECT, которая может находиться после фразы HAVING, например:

```
SELECT count() OVER w, sum(b) OVER w,
    avg(b) OVER (w ORDER BY c ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
FROM table1
WINDOW w AS (PARTITION BY a)
```

Определенное таким образом окно можно использовать как есть. В примере выше функции count и sum так и делают. Но его можно и уточнить, как сделано в функции avg. Синтаксически разница состоит в следующем: чтобы повторно использовать определение окна, имя этого окна следует указать после ключевого слова OVER без скобок. Если же мы хотим расширить определение окна фразой ORDER BY или определением фрейма, то имя окна следует указать внутри скобок.

Если одно и то же определение окна используется несколько раз, то PostgreSQL оптимизирует выполнение запроса: строит разделы только один раз и использует результаты повторно.

Использование оконных функций

Все агрегатные функции, в т. ч. определенные пользователем, можно использовать как оконные, за исключением агрегатов по упорядоченным наборам и по гипотетическим наборам. На то, что функция выступает в роли оконной, указывает наличие фразы `OVER`.

Если агрегатная функция используется в качестве оконной, то она агрегирует строки, принадлежащие оконному фрейму текущей строки. Чаще всего оконные функции применяются для вычисления различных статистических показателей. В базе данных для сайта торговли автомобилями есть таблица `advertisement`, содержащая информацию о рекламных объявлениях, созданных пользователями. Пусть требуется проанализировать, сколько объявлений создано за период. Такой отчет генерируется следующим запросом:

```
car_portal=> WITH monthly_data AS (
  SELECT date_trunc('month', advertisement_date) AS month, count(*) as cnt
    FROM car_portal_app.advertisement
    GROUP BY date_trunc('month', advertisement_date)
)
  SELECT to_char(month,'YYYY-MM') as month, cnt,
    sum(cnt) OVER (w ORDER BY month) AS cnt_year,
    round(avg(cnt) OVER (ORDER BY month ROWS BETWEEN 2 PRECEDING AND 2 FOLLOWING), 1)
      AS mov_avg,
    round(cnt / sum(cnt) OVER w * 100,1) AS ratio_year
  FROM monthly_data WINDOW w AS (PARTITION BY date_trunc('year',month));
   month | cnt | cnt_year | mov_avg | ratio_year
---------+-----+----------+---------+------------
 2014-01 |  42 |       42 |    40.3 |        5.8
 2014-02 |  49 |       91 |    44.5 |        6.7
 2014-03 |  30 |      121 |    56.8 |        4.1
 2014-04 |  57 |      178 |    69.0 |        7.8
 2014-05 | 106 |      284 |    73.0 |       14.6
 2014-06 | 103 |      387 |    81.0 |       14.2
 2014-07 |  69 |      456 |    86.0 |        9.5
 2014-08 |  70 |      526 |    74.0 |        9.6
 2014-09 |  82 |      608 |    60.6 |       11.3
 2014-10 |  46 |      654 |    54.2 |        6.3
 2014-11 |  36 |      690 |    49.8 |        5.0
 2014-12 |  37 |      727 |    35.2 |        5.1
 2015-01 |  48 |       48 |    32.5 |       84.2
 2015-02 |   9 |       57 |    31.3 |       15.8

(14 rows)
```

Во фразе `WITH` данные агрегируются по месяцам. В главном запросе определяется окно `w`, предполагающее разбиение по годам. Это означает, что любая

оконная функция, работающая с окном w, будет видеть записи за тот же год, что и текущая запись.

Первая оконная функция, sum, пользуется окном w. Поскольку присутствует фраза ORDER BY, у каждой записи есть определенная позиция в разделе. Определение фрейма опущено, поэтому предполагается фрейм RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW. Следовательно, функция вычисляет сумму значений по всем записям, начиная с начала года и до текущего месяца включительно, т. е. накопительный итог за год.

Вторая функция, avg, для каждой записи вычисляет скользящее среднее по пяти записям – две предшествующие, текущая и две последующие. Ранее определенное окно не используется, потому что при вычислении скользящего среднего год не принимается во внимание, важен только порядок записей.

Третья оконная функция, sum, пользуется тем же окном w. Вычисленная ею сумма значений за год является знаменателем в выражении, которое дает вклад текущего месяца в сумму.

Существует ряд оконных функций, не являющихся агрегатными. Они служат для получения значений других записей в разделе, для вычисления ранга текущей записи относительно остальных и для генерации номеров строк.

Изменим предыдущий отчет, будем считать, что требуется вычислить две разности: между количеством объявлений в текущем месяце и предыдущем месяце того же года и в том же месяце прошлого года, а также ранг текущего месяца. Вот как выглядит такой запрос:

```
car_portal=> WITH monthly_data AS (
  SELECT date_trunc('month', advertisement_date) AS month, count(*) as cnt
    FROM car_portal_app.advertisement
    GROUP BY date_trunc('month', advertisement_date)
)
SELECT to_char(month,'YYYY-MM') as month, cnt,
    cnt - lag(cnt) OVER (ORDER BY month) as prev_m,
    cnt - lag(cnt, 12) OVER (ORDER BY month) as prev_y,
    rank() OVER (w ORDER BY cnt DESC) as rank
FROM monthly_data
WINDOW w AS (PARTITION BY date_trunc('year',month))
ORDER BY month DESC;
   month | cnt | prev_m | prev_y | rank
---------+-----+--------+--------+------
 2015-02 |   9 |    -39 |    -40 |    2
 2015-01 |  48 |     11 |      6 |    1
 2014-12 |  37 |      1 |        |   10
 2014-11 |  36 |    -10 |        |   11
 2014-10 |  46 |    -36 |        |    8
 2014-09 |  82 |     12 |        |    3
 2014-08 |  70 |      1 |        |    4
 2014-07 |  69 |    -34 |        |    5
 2014-06 | 103 |     -3 |        |    2
 2014-05 | 106 |     49 |        |    1
 2014-04 |  57 |     27 |        |    6
```

```
 2014-03 |  30 |    -19 |        |   12
 2014-02 |  49 |      7 |        |    7
 2014-01 |  42 |        |        |    9

(14 rows)
```

Функция `lag` возвращает значение указанного выражения для записи, отстоящей на заданное число записей (по умолчанию 1) назад от текущей. При первом вызове эта функция возвращает значение поля `cnt` из предыдущей записи, т. е. количество объявлений в предыдущем месяце. Как видим, в феврале 2015 года опубликовано 9 объявлений – на 39 меньше, чем в январе 2015-го.

При втором вызове `lag` возвращает значение `cnt` для записи, отстоящей на 12 записей, т. е. на год назад от предыдущей. Число объявлений в феврале 2015 года на 42 меньше, чем в феврале 2014-го.

Функция `rank` возвращает ранг текущей строки внутри раздела. Имеется в виду ранг с промежутками, т. е. в случае, когда две записи занимают одинаковую позицию в порядке, заданном фразой `ORDER BY`, обе получают одинаковый ранг, а следующая получает ранг на две единицы больше. То есть у нас будет две первые записи и одна третья.

Перечислим другие оконные функции:

- `lead`: аналогична `lag`, но возвращает значение выражения, вычисленное для записи, отстоящей от текущей на указанное количество записей вперед;
- `first_value`, `last_value`, `nth_value`: возвращают значения выражения, вычисленные соответственно для первой, последней и *n*-й записи фрейма;
- `row_number`: возвращает номер текущей строки в разделе;
- `dense_rank`: возвращает ранг текущей строки без промежутков (плотный ранг);
- `percent_rank` и `cume_dist`: возвращает относительный ранг текущей строки. Разница между функциями состоит в том, что в первой числителем дроби является ранг, а во второй – номер строки;
- `ntile`: делит раздел на заданное количество равных частей и возвращает номер части, в которую попала текущая строка.

Подробное описание этих функций см. на странице http://www.postgresql.org/docs/current/static/functions-window.html.

Оконные функции с группировкой и агрегированием

Поскольку оконные функции вычисляются после группировки, агрегатные функции могут быть аргументами оконных, но не наоборот. Код `sum(count(*)) OVER()` правилен, как и код `sum(a) OVER(ORDER BY count(*))`. Однако код `sum(count(*) OVER())` недопустим.

Например, чтобы вычислить ранги учетных записей продавцов по числу данных ими объявлений, можно выполнить такой запрос:

```
car_portal=> SELECT seller_account_id, dense_rank() OVER(ORDER BY count(*) DESC)
FROM car_portal_app.advertisement
GROUP BY seller_account_id;
 seller_account_id | dense_rank
-------------------+------------
                26 |          1
               128 |          2
                28 |          2
               126 |          2
...
```

Продвинутые методы работы с SQL

В этом разделе мы рассмотрим следующие методы работы с SQL:

- фраза DISTINCT ON, позволяющая находить первые записи в группах;
- извлечение выборочных данных из большой таблицы;
- функции, возвращающие множества;
- латеральные соединения, позволяющие ссылаться из одного подзапроса на другой;
- специальные методы группировки, полезные для генерации отчетов;
- специальные агрегатные функции.

Выборка первых записей

Часто бывает необходимо найти первые записи относительно какого-то критерия. Допустим, к примеру, что в базе данных car_portal нужно найти первое объявление для каждого идентификатора car_id в таблице advertisement.

В этом случае нам может помочь группировка. Для реализации логики необходим подзапрос:

```
SELECT advertisement_id, advertisement_date, adv.car_id, seller_account_id
  FROM car_portal_app.advertisement adv
  INNER JOIN
  (SELECT car_id, min(advertisement_date) min_date
   FROM car_portal_app.advertisement
   GROUP BY car_id) first
  ON adv.car_id=first.car_id AND adv.advertisement_date = first.min_date;
```

Но если логика упорядочения настолько сложна, что для ее реализации функции min недостаточно, то такой подход работать не будет. Проблему могут решить оконные функции, но не всегда они удобны:

```
SELECT DISTINCT first_value(advertisement_id) OVER w AS advertisement_id,
  min(advertisement_date) OVER w AS advertisement_date,
  car_id, first_value(seller_account_id) OVER w AS seller_account_id
FROM car_portal_app.advertisement
WINDOW w AS (PARTITION BY car_id ORDER BY advertisement_date);
```

Здесь слово DISTINCT удаляет дубликаты, которые попали в одну группу.

PostgreSQL предлагает явный способ выбрать первую запись из каждой группы – ключевое слово DISTINCT ON. Синтаксически это выглядит так:

```
SELECT DISTINCT ON (<expression_list>) <Select-List>
...
ORDER BY <order_by_list>
```

Здесь для каждой уникальной комбинации значений из списка выражений expression_list команда SELECT возвращает только первую запись. Для определения того, что такое «первая», служит фраза ORDER BY. Все выражения из списка expression_list должны присутствовать в списке order_by_list.

К рассматриваемой задаче эту идею можно применить следующим образом:

```
SELECT DISTINCT ON (car_id) advertisement_id, advertisement_date, car_id,
seller_account_id
FROM car_portal_app.advertisement
ORDER BY car_id, advertisement_date;
```

Этот код гораздо чище, проще для понимания и в большинстве случаев быстрее работает.

Извлечение выборочных данных

Допустим, что база данных car_portal очень велика, а нам нужно собрать статистику по хранящимся в ней данным. Абсолютной точности не требуется, будет достаточно и оценки. Но важно сделать это быстро. В PostgreSQL имеется способ опрашивать не всю таблицу, а случайную выборку из нее:

```
SELECT ... FROM <table> TABLESAMPLE <sampling_method> ( <argument> [, ...] )
[ REPEATABLE ( <seed> ) ]
```

Метод формирования выборки, sampling_method, может принимать значения BERNOULLI или SYSTEM. Оба эти метода входят в стандартный комплект поставки PostgreSQL, другие можно добавить в качестве расширений. Оба метода принимают один аргумент – вероятность включения строки в выборку в процентах. Различие в том, что метод BERNOULLI предполагает полный просмотр таблицы, а алгоритм для каждой строки решает, включать ее в выборку или нет. Метод SYSTEM делает то же самое, но на уровне блоков строк. Это значительно быстрее, но результаты не столь точны, потому что количество строк в разных блоках неодинаково. Ключевое слово REPEATABLE задает начальное значение для случайных функций, используемых в алгоритме формирования выборки. Запросы с одинаковым начальным значением возвращают одинаковые результаты при условии, что данные в таблице не изменялись.

Отбор строк происходит на этапе чтения данных из таблицы, еще до фильтрации, группировки и всех прочих операций.

Для иллюстрации увеличим количество записей в таблице advertisement, выполнив следующий запрос:

```
car_portal=> INSERT INTO advertisement (advertisement_date, car_id, seller_account_id)
  SELECT advertisement_date, car_id, seller_account_id from advertisement;
INSERT 0 784
```

Мы попросту дублируем все записи в таблице, удваивая их число. Если выполнить этот запрос много раз, то таблица окажется достаточно большой. Мы проделали это 12 раз. После этого даже простой подсчет количества записей занимает заметное время:

```
car_portal=> \timing
Timing is on.
car_portal=> SELECT count(*) FROM advertisement;
  count
---------
 6422528
(1 row)
Time: 394.577 ms
```

Теперь попробуем сделать то же самое для выборочных данных:

```
car_portal=> SELECT count(*) * 100 FROM advertisement TABLESAMPLE SYSTEM
(1);
 ?column?
----------
  6248600
(1 row)
Time: 8.344 ms
```

Второй запрос гораздо быстрее, но результат не вполне точный. Мы умножили значение функции `count` на 100, потому что она возвращает процент строк. При многократном выполнении запроса результат всякий раз будет разным, и иногда он будет сильно отличаться от истинного количества строк в таблице. Чтобы повысить качество, увеличим долю строк в выборке:

```
car_portal=> SELECT count(*) * 10 from advertisement TABLESAMPLE SYSTEM
(10);
 ?column?
----------
  6427580
(1 row)
Time: 92.793 ms
```

Время работы увеличилось пропорционально.

Функции, возвращающие множества

Функции в PostgreSQL могут возвращать не только одиночные значения, но и отношения. Они называются **функциями, возвращающими множество**.

Вот типичная задача, встающая перед всяким разработчиком на SQL: сгенерировать последовательность целых чисел, по одному в записи. У такой последовательности, или отношения, много применений. Предположим, к примеру, что в базе `car_portal` нужно подсчитать количество автомобилей, выпущенных

в каждом году от 2010-го до 2015-го, показав нули, если автомобилей за какой-то год в базе нет. Простой SELECT из одной лишь таблицы car задачу не решит. Невозможно показать данные, которых нет в таблице.

Поэтому было бы полезно иметь таблицу, содержащую числа от 2010 до 2015. Тогда мы смогли бы выполнить внешнее соединение между ней и таблицей результатов.

Можно было бы подготовить такую таблицу заранее, но это не очень хорошо, потому что сколько понадобится записей, заранее неизвестно, а создавать слишком большую таблицу – значит впустую транжирить место на диске.

Но существует функция generate_series, специально предназначенная для этой цели. Она возвращает множество целых чисел между двумя заданными. Для решения нашей задачи надо было бы написать такой запрос:

```
car_portal=> SELECT years.manufacture_year, count(car_id)
  FROM generate_series(2010, 2015) as years (manufacture_year)
    LEFT JOIN car_portal_app.car ON car.manufacture_year = years.manufacture_year
  GROUP BY years.manufacture_year
  ORDER BY 1;
 manufacture_year | count
------------------+-------
             2010 |    11
             2011 |    12
             2012 |    12
             2013 |    21
             2014 |    16
             2015 |     0
(6 rows)
```

Здесь функция generate_series возвращает шесть целых чисел от 2010 до 2015. Этому множеству сопоставлен псевдоним years. Таблица car соединяется слева с этим множеством, и теперь мы видим в результирующем наборе все годы.

При вызове generate_series можно задать шаг:

```
car_portal=> SELECT * FROM generate_series(5, 11, 3);
 generate_series
-----------------
               5
               8
              11
(3 rows)
```

Функция generate_series может также возвращать множество значений типа timestamp:

```
car_portal=> SELECT * FROM generate_series('2015-01-01'::date,
'2015-01-31'::date, interval '7 days');
 generate_series
-----------------------
 2015-01-01 00:00:00-05
```

```
 2015-01-08 00:00:00-05
 2015-01-15 00:00:00-05
 2015-01-22 00:00:00-05
 2015-01-29 00:00:00-05
(5 rows)
```

Есть еще две функции, возвращающие множества, предназначенные для работы с массивами:

- `generate_subscripts`: генерирует набор индексов для указанной размерности заданного массива. Полезна для перечисления элементов массива в команде SQL;
- `unnest`: преобразует заданный массив в набор строк, каждая запись которого соответствует элементу массива.

Функции, возвращающие множества, называют еще **табличными функциями**.

Табличная функция может возвращать множество строк предопределенного типа, например `generate_series` возвращает значение типа `setof int` или `setof bigint` (в зависимости от типа входного аргумента). А может возвращать множество записей абстрактного типа. Это позволяет возвращать таблицы с разным числом столбцов, зависящим от аргументов. SQL требует, чтобы структура строк всех входных отношений была определена таким образом, чтобы структура результирующего множества также была определена. Именно поэтому использование табличных функций в запросе может сопровождаться определением структуры строки возвращаемого множества:

```
function_name (<arguments>) [AS] alias (column_name column_type [, ...])
```

Результаты нескольких табличных функций можно объединять, как если бы они были соединены по позиции строки. Для этого служит конструкция `ROWS FROM`:

```
ROWS FROM (function_call [,…]) [[AS] alias (column_name [,...])]
```

Эта конструкция возвращает отношение, поэтому ее можно использовать во фразе `FROM`, как любое другое отношение. Количество строк равно размеру самого большого результата перечисленных функций. Столбцы соответствуют функциям во фразе `ROWS FROM`. Если какая-то функция возвращает меньше строк, чем другие, то отсутствующие значения будут равны `NULL`:

```
car_portal=> SELECT foo.a, foo.b FROM
ROWS FROM (generate_series(1,3), generate_series(1,7,2)) AS foo(a, b);
 a | b
---+---
 1 | 1
 2 | 3
 3 | 5
   | 7
(4 rows)
```

Латеральные подзапросы

Подзапросы рассматривались в предыдущей главе. Но стоит подробнее остановиться на одном конкретном способе их использования.

Очень удобно использовать подзапросы в списке выборки. Например, в командах SELECT с их помощью можно создавать вычисляемые атрибуты при опросе таблицы. Снова обратимся к базе данных car_portal. Пусть требуется для каждого автомобиля в таблице car оценить его возраст, сравнив с возрастом других автомобилей той же модели. И кроме того, мы хотим получить общее количество автомобилей той же модели.

Оба дополнительных поля можно сгенерировать скалярными подзапросами:

```
car_portal=> SELECT car_id, manufacture_year,
  CASE WHEN manufacture_year <= (
    SELECT avg(manufacture_year) FROM car_portal_app.car WHERE car_model_id
= c.car_model_id)
    THEN 'old' ELSE 'new' END as age,
  (SELECT count(*) FROM car_portal_app.car WHERE car_model_id = c.car_model_id)
    AS same_model_count
  FROM car_portal_app.car c;
 car_id | manufacture_year | age | same_model_count
--------+------------------+-----+-----------------
      1 |             2008 | old |               3
      2 |             2014 | new |               6
      3 |             2014 | new |               2
...
(229 rows)
```

Эти подзапросы интересны тем, что могут ссылаться на главную таблицу в своей фразе WHERE clause. С помощью подобных подзапросов легко добавить в запрос дополнительные столбцы. Но есть и проблема – производительность. Таблица car просматривается сервером один раз в главном запросе, а затем еще два раза для каждой выбранной строки, т. е. для столбцов age and same_model_count.

Конечно, можно независимо вычислить эти агрегаты по одному разу для каждой модели, а затем соединить результаты с таблицей car:

```
car_portal=> SELECT car_id, manufacture_year,
    CASE WHEN manufacture_year <= avg_year THEN 'old' ELSE 'new' END as age,
        same_model_count
    FROM car_portal_app.car
      INNER JOIN (
        SELECT car_model_id, avg(manufacture_year) avg_year, count(*) same_model_count
        FROM car_portal_app.car
        GROUP BY car_model_id) subq
        USING (car_model_id);
 car_id | manufacture_year | age | same_model_count
--------+------------------+-----+-----------------
      1 |             2008 | old |               3
      2 |             2014 | new |               6
```

```
      3 |             2014 | new |                2
...
(229 rows)
```

Результат тот же самый, а запрос выполняется в 20 раз быстрее. Однако этот запрос хорош только тогда, когда нужно выбрать из базы данных много строк. Если нужно получить информацию только об одном автомобиле, то первый запрос будет быстрее.

Легко видеть, что первый запрос мог бы работать лучше, если бы мы включили в список выборки подзапроса два столбца, но, увы, это невозможно. Скалярные запросы могут возвращать только один столбец.

Существует еще один способ использования подзапросов. Он сочетает в себе преимущества подзапросов в списке выборки, способных обращаться к главной таблице из фразы WHERE, с подзапросами во фразе FROM, которые могут возвращать несколько столбцов. Если поместить ключевое слово LATERAL перед подзапросом во фразе FROM, то он сможет ссылаться на любой элемент, предшествующий ему во фразе FROM.

Запрос выглядит следующим образом:

```
car_portal=> SELECT car_id, manufacture_year,
    CASE WHEN manufacture_year <= avg_year THEN 'old' ELSE 'new' END as age,
    same_model_count
FROM car_portal_app.car c,
  LATERAL (
    SELECT avg(manufacture_year) avg_year,count(*) same_model_count
    FROM car_portal_app.car
    WHERE car_model_id = c.car_model_id) subq;
 car_id | manufacture_year | age | same_model_count
--------+------------------+-----+-----------------
      1 |             2008 | old |                3
      2 |             2014 | new |                6
      3 |             2014 | new |                2
...
(229 rows)
```

Этот запрос работает приблизительно в два раза быстрее первого, он оптимален, когда нужно выбрать из таблицы car всего одну строку.

Синтаксис JOIN также допускается в латеральных подзапросах, хотя в большинстве случаев условие соединения каким-то образом включается во фразу WHERE подзапросов.

Употреблять LATERAL с функциями, возвращающими множества, нет необходимости. Все функции, упомянутые во фразе FROM, и так могут использовать результаты любых предшествующих функций или подзапросов:

```
car_portal=> SELECT a, b
FROM generate_series(1,3) AS a, generate_series(a, a+2) AS b;
 a | b
---+---
 1 | 1
```

```
 1 | 2
 1 | 3
 2 | 2
 2 | 3
 2 | 4
 3 | 3
 3 | 4
 3 | 5
(9 rows)
```

Здесь первая функция, с псевдонимом a, возвращает три строки. Для каждой из них вызывается вторая функция, возвращающая еще три строки.

Дополнительные средства группировки

Базы данных часто служат источником данных для различных отчетов. А в отчетах принято в одной и той же таблице показывать подытоги, итоги и общие итоги, подразумевающие группировку и агрегирование. Рассмотрим отчет о количестве объявлений в разрезе марок и кварталов, в котором требуется также показать итоги по каждому кварталу (суммарно по всем маркам) и общий итог. Вот как отбираются данные для такого отчета:

```
car_portal=> SELECT to_char(advertisement_date, 'YYYY-Q') as quarter, make,
count(*)
  FROM advertisement a
    INNER JOIN car c ON a.car_id = c.car_id
    INNER JOIN car_model m ON m.car_model_id = c.car_model_id
  GROUP BY quarter, make;
 quarter | make          | count
---------+---------------+-------
  2014-4 | Peugeot       |    12
  2014-2 | Daewoo        |     8
  2014-4 | Skoda         |     5
...
```

Чтобы вычислить итоги, понадобятся дополнительные запросы:

```
SELECT to_char(advertisement_date, 'YYYY-Q') as quarter, count(*)
  FROM advertisement a
    INNER JOIN car c ON a.car_id = c.car_id
    INNER JOIN car_model m ON m.car_model_id = c.car_model_id
  GROUP BY quarter;

SELECT count(*)
  FROM advertisement a
    INNER JOIN car c ON a.car_id = c.car_id
    INNER JOIN car_model m ON m.car_model_id = c.car_model_id;
```

Соединения в этих запросах излишни, они не влияют на результат, но мы их оставили, чтобы показать, что запросы фактически такие же, как первый, отличается только группировка.

PostgreSQL позволяет объединить все три запроса в один с помощью специальной конструкции GROUP BY GROUPING SETS:

```
SELECT to_char(advertisement_date, 'YYYY-Q') as quarter, make, count(*)
  FROM advertisement a
    INNER JOIN car c ON a.car_id = c.car_id
    INNER JOIN car_model m ON m.car_model_id = c.car_model_id
  GROUP BY GROUPING SETS ((quarter, make), (quarter), ())
  ORDER BY quarter NULLS LAST, make NULLS LAST;
 quarter | make          | count
---------+---------------+-------
  2014-1 | Alfa Romeo    |     2
  2014-1 | Audi          |     5
...
  2014-1 | Volvo         |     9
  2014-1 |               |   121 <- Это квартальный подытог
 2014-1
  2014-2 | Audi          |    18
  2014-2 | Citroen       |    40
...
  2014-2 | Volvo         |    12
  2014-2 |               |   266 <- Это квартальный подытог
2014-2
  2014-3 | Audi          |    11
...
  2015-1 | Volvo         |     4
  2015-1 |               |    57 <- Это квартальный подытог
2015-1
         |               |   784 <- Это общий итог
```

Благодаря конструкции GROUPING SETS ((quarter, make), (quarter), ()) запрос работает как UNION ALL одинаковых запросов с разными фразами GROUP BY:

- GROUP BY quarter, make;
- GROUP BY quarter – здесь поле make будет иметь значение NULL;
- все строки помещаются в одну группу, оба поля – quarter и make – будут иметь значение NULL.

Вообще говоря, фраза GROUP BY принимает не только выражения, но и группирующие элементы, которые могут быть выражениями или конструкциями типа GROUPING SETS.

Два других группирующих элемента – ROLLUP и CUBE.

- ROLLUP (a, b, c) эквивалентно GROUPING SETS ((a, b, c), (a, b), (c), ()).
- CUBE (a, b, c) эквивалентно GROUPING SETS ((a, b, c), (a, b), (a, c), (b, c), (a), (b), (c), ()) – все возможные комбинации аргументов.

Разрешено также использовать CUBE и ROLLUP внутри GROUPING SETS, а равно комбинировать различные группирующие элементы в одной фразе GROUP BY, например: GROUP BY a, CUBE (b, c). Это эквивалентно GROUP BY GROUPING SETS ((a, b, c), (a, b), (a,c), (a)) – по сути дела, комбинация группирующих элементов.

Хотя GROUPING SETS логически эквивалентно UNION ALL результатов запросов с разными фразами GROUP BY, реализована эта конструкция не так, как теорети-

ко-множественная операция. PostgreSQL просматривает таблицы и выполняет соединения и фильтрацию только один раз. Следовательно, применение таких методов группировки может повысить производительность запросов.

Документация по конструкции `GROUPING SETS` находится на странице https://www.postgresql.org/docs/current/static/queries-table-expressions.html#QUERIES-GROUPING-SETS.

Дополнительные виды агрегирования

Есть несколько агрегатных функций, которые выполняются специальным образом.

Первая группа таких функций называется **агрегатами по упорядоченному множеству** (ordered-set aggregates). Они учитывают не только значения выражений, переданных в качестве аргументов, но и их порядок. Такие агрегаты имеют отношение к статистике и вычислению процентилей.

Процентиль – это значение группы, в которой процентное значение общих значений равно этому значению или меньше его. Например, если некоторое значение равно 95-му процентилю, значит, оно больше 95 процентов остальных значений. PostgreSQL позволяет вычислить непрерывный или дискретный процентиль с помощью функций `percentile_cont` и `percentile_disc` соответственно. Дискретный процентиль совпадает с одним из значений группы, а непрерывный – результат интерполяции двух существующих значений. Есть возможность вычислить один процентиль для заданной процентной доли или сразу несколько процентилей для заданного массива долей.

Вот, например, запрос о распределении количества объявлений по автомобилям:

```
car_portal=> SELECT percentile_disc(ARRAY[0.25, 0.5, 0.75]) WITHIN GROUP
(ORDER BY cnt)
  FROM (SELECT count(*) cnt FROM car_portal_app.advertisement
  GROUP BY car_id) subq;
 percentile_disc
-----------------
 {2,3,5}
(1 row)
```

Результат означает, что для 25 процентов автомобилей существует не более двух объявлений, для 50 процентов – три объявления и для 75 процентов – пять объявлений.

Синтаксис агрегатных функций по упорядоченному множеству отличается от обычных агрегатных функций, в нем используется специальная конструкция `WITHIN GROUP (ORDER BY expression)`. Здесь выражение `expression` – фактически аргумент функции. На результат функции влияет не только порядок строк, но и значения этих выражений. В противоположность фразе `ORDER BY` в команде `SELECT` допускается только одно выражение, и ссылок на номера результирующих столбцов быть не должно.

Еще одна агрегатная функция по упорядоченному множеству называется mode. Она возвращает самое часто встречающееся значение в группе. Если два значения встречаются одинаковое число раз, то возвращается первое из них.

Например, следующий запрос возвращает идентификатор самой частой встречающейся модели автомобиля в базе данных:

```
car_portal=> SELECT mode() WITHIN GROUP (ORDER BY car_model_id)
FROM car_portal_app.car;
 mode
------
   64
(1 row)
```

Чтобы получить тот же результат без этой функции, понадобилось бы самосоединение или сортировка и ограничение количества результатов. То и другое обошлось бы дороже.

Другая группа агрегатов с таким же синтаксисом – агрегатные функции по гипотетическому множеству: rank, dense_rank, percent_rank и cume_dist. Существуют оконные функции с такими же именами. Но оконные функции не принимают аргумента и возвращают результат для текущей строки. Для агрегатных функций понятия текущей строки не существует, потому что они вычисляются для группы. Зато они принимают аргумент: значение гипотетической текущей строки.

Например, агрегатная функция rank возвращает ранг заданного значения в упорядоченном множестве, если бы оно в этом множестве присутствовало:

```
car_portal=> SELECT rank(2) WITHIN GROUP (ORDER BY a)
FROM generate_series(1,10,3) a;
 rank
------
    2
(1 row)
```

В данном случае значения 2 нет в результате функции generate_series (она возвращает множество 1..4..7..10). Но если бы оно там присутствовало, то занимало бы вторую позицию.

Еще один достойный упоминания аспект агрегатных функций – фраза FILTER. Она по заданному условию отфильтровывает строки, передаваемые агрегатной функции. Пусть, например, требуется подсчитать количество автомобилей для каждой модели и каждого значения количества дверей. Если бы мы сгруппировали данные по этим двум полям, то результат получился бы правильным, но не очень удобным для отчетности:

```
car_portal=> SELECT car_model_id, number_of_doors, count(*)
FROM car_portal_app.car
GROUP BY car_model_id, number_of_doors;
 car_model_id | number_of_doors | count
--------------+-----------------+-------
           47 |               4 |     1
```

```
          42 |              3 |     2
          76 |              5 |     1
          52 |              5 |     2
...
```

Благодаря фразе FILTER результат становится гораздо понятнее:

```
car_portal=> SELECT car_model_id,
    count(*) FILTER (WHERE number_of_doors = 2) doors2,
    count(*) FILTER (WHERE number_of_doors = 3) doors3,
    count(*) FILTER (WHERE number_of_doors = 4) doors4,
    count(*) FILTER (WHERE number_of_doors = 5) doors5
    FROM car_portal_app.car GROUP BY car_model_id;
 car_model_id | doors2 | doors3 | doors4 | doors5
--------------+--------+--------+--------+--------
          43 |      0 |      0 |      0 |      2
           8 |      0 |      0 |      1 |      0
          11 |      0 |      2 |      1 |      0
          80 |      0 |      1 |      0 |      0
...
```

Обратите внимание, что этот запрос не считает автомобили с количеством дверей меньше двух и больше пяти.

Того же результата можно достичь и таким образом:

```
count(CASE WHEN number_of_doors = 2 THEN 1 END) doors2
```

Но вариант с фразой FILTER короче и проще.

Резюме

В этой главе были рассмотрены дополнительные средства SQL, в т. ч. общие табличные выражения и оконные функции. С их помощью можно реализовать вещи, которые никак иначе не сделаешь, например рекурсивные запросы.

Другие рассмотренные конструкции, например фраза DISTINCT ON, группирующие элементы, фраза FILTER и побочные подзапросы, не являются столь же незаменимыми, но помогают сделать запрос меньше, проще и быстрее.

SQL можно использовать для реализации очень сложной логики. Но в особо трудных ситуациях запросы могут стать перегруженными и трудными для сопровождения. А иногда решить задачу на чистом SQL вообще невозможно. В таких случаях на помощь приходит процедурный язык, и в следующей главе мы рассмотрим один из них: PL/pgSQL.

Глава 7

Серверное программирование на PL/pgSQL

Возможность писать функции в PostgreSQL – вещь изумительная. В контексте сервера базы данных можно решить любую задачу. Функция может относиться непосредственно к манипулированию данными, например агрегированию и аудиту, или к таким разнообразным операциям, как сбор статистики, мониторинг, доступ к системной информации и планирование работ. В этой главе мы займемся PL/pgSQL, полноценным процедурным языком программирования для PostgreSQL, который устанавливается по умолчанию.

На PL/pgSQL оказал больше влияние язык PL/SQL, используемый для написания хранимых процедур в Oracle. PL/pgSQL обладает развитыми структурами управления и прекрасно интегрирован с триггерами, индексами, правилами, пользовательскими типами данных и операторами PostgreSQL. Перечислим несколько преимуществ PL/pgSQL:

- простота изучения и применения;
- очень хорошая поддержка и документация;
- гибкие типы выходных данных, поддержка полиморфизма;
- возможность возвращать скалярные значения и множества.

В этой главе мы рассмотрим следующие темы:

- сравнение языков SQL и PL/pgSQL;
- параметры функций в PostgreSQL;
- команды управления в PL/pgSQL;
- предопределенные переменные в функциях;
- обработка исключений;
- динамический SQL.

Сравнение языков SQL и PL/pgSQL

Как было показано в главе 4, функции можно писать на языках C, SQL и PL/pgSQL. У каждого подхода есть свои плюсы и минусы. Можно считать, что функция

на SQL – это обертка вокруг параметризованной команды SELECT. SQL-функции можно вызывать прямо из подзапроса, чтобы повысить производительность. Кроме того, поскольку план выполнения SQL-функции не кешируется, как в случае PL/pgSQL, она зачастую ведет себя лучше, чем функция на PL/pgSQL. Кеширование в PL/ pgSQL иногда имеет нежелательные побочные эффекты, например кеширование важных значений типа timestamp. Об этом можно прочитать в документации на странице http://www.postgresql.org/docs/current/interactive/plpgsqlimplementation.html. Наконец, с появлением общих табличных выражений, в т. ч. рекурсивных, оконных функций и латеральных соединений стало возможно реализовывать сложную логику одними лишь средствами SQL. Если операцию можно выполнить на SQL, то не стоит прибегать к PL/pgSQL.

План выполнения PL/pgSQL-функции кешируется, что позволяет уменьшить время выполнения, но может и нанести вред, если план не является оптимальным для конкретных параметров функции. С точки зрения функциональности, PL/pgSQL – значительно более мощный язык, чем SQL. PL/pgSQL поддерживает ряд возможностей, отсутствующих в SQL, в том числе:

- возможность возбуждать исключения, а также генерировать сообщения разных уровней, например информационные и отладочные;
- поддержка конструирования динамических SQL-команд с помощью команды EXECUTE;
- обработка исключений EXCEPTION;
- полный набор команд присваивания, управления и циклов;
- поддержка курсоров;
- полная интеграция с триггерами PostgreSQL. SQL-функции нельзя использовать совместно с триггерами.

Параметры функций в PostgreSQL

В главе 4 мы упоминали о категориях функций: неизменяемые, стабильные и волатильные. В этом разделе мы продолжим рассмотрение, обратив внимание на другие стороны, которые не являются спецификой только PL/pgSQL.

Параметры функций, относящиеся к авторизации

Сначала рассмотрим параметры, относящиеся к авторизации. Любая вызванная функция выполняется в некотором контексте безопасности, определяющем ее привилегии. Контекст безопасности контролируется следующими параметрами:

- SECURITY DEFINER;
- SECURITY INVOKER.

По умолчанию подразумевается режим SECURITY INVOKER, означающий, что функция выполняется с привилегиями вызывающей стороны. В режиме SECURITY DEFINER функция выполняется с привилегиями создавшего ее пользователя. Для функций в режиме SECURITY INVOKER у пользователя должны быть

разрешения на выполнение тех операций CRUD, которые встречаются в функции, в противном случае возникнет ошибка. Функции в режиме SECURITY DEFINER очень полезны при определении триггеров или для временного повышения привилегий пользователя – только на время выполнения задачи, для которой написана функция.

Для ознакомления с этими параметрами безопасности создадим две простенькие функции от имени пользователя postgres:

```
CREATE FUNCTION test_security_definer () RETURNS TEXT AS $$
  SELECT format ('current_user:%s session_user:%s', current_user, session_user);
$$ LANGUAGE SQL SECURITY DEFINER;

CREATE FUNCTION test_security_invoker () RETURNS TEXT AS $$
  SELECT format ('current_user:%s session_user:%s', current_user, session_user);
$$ LANGUAGE SQL SECURITY INVOKER;
```

Выполним их в сеансе пользователя postgres:

```
$ psql -U postgres car_portal
psql (10.0)
Type "help" for help.
car_portal=# SELECT test_security_definer() , test_security_invoker();
                        test_security_definer | test_security_invoker
----------------------------------------------+----------------------------
---------------
 current_user:postgres session_user:postgres | current_user:postgres
session_user:postgres
(1 row)
```

Теперь откроем другой сеанс от имени пользователя car_portal_app:

```
psql -U car_portal_app car_portal
psql (10.0)
Type "help" for help.
car_portal=> SELECT test_security_definer() , test_security_invoker();
                                test_security_definer | test_security_invoker
------------------------------------------------------+--------------------
-------------------------------------
current_user:postgres session_user:car_portal_app |
current_user:car_portal_app session_user:car_portal_app
(1 row)
```

Функции test_security_definer и test_security_invoker не отличаются ничем, кроме параметра безопасности. При выполнении от имени пользователя postgres они дают одинаковые результаты, потому что их создавал и вызывал один и тот же пользователь.

Но если эти функции выполняет пользователь car_portal_app, то test_security_definer возвращает current_user:postgres session_user:car_portal_app. В этом случае session_user совпадает с car_portal_app, поскольку именно этот пользователь запустил сеанс psql. Однако же current_user, выполняющий команду SELECT, – это postgres.

Параметры функции, относящиеся к планировщику

Параметры этой категории нужны для передачи планировщику информации о стоимости выполнения функции, что позволяет генерировать хорошие планы выполнения. Следующие три параметра помогают планировщику определить стоимость выполнения функции, ожидаемое количество возвращенных строк и возможность перемещения вызова функции в конец при вычислении предикатов.

- `LEAKPROOF`: означает, что у функции нет побочных эффектов. Она не раскрывает никакой информации о своих аргументах и, в частности, не генерирует сообщений об ошибке, относящихся к аргументу. Этот параметр влияет на представления с параметром `security_barrier`.
- `COST`: объявляет стоимость выполнения в расчете на строку. По умолчанию для функции на C она равна 1, а для функции на PL/pgSQL – 100. Эта стоимость используется планировщиком для построения оптимального плана выполнения.
- `ROWS`: оценка количества строк, возвращаемых функцией (для функций, возвращающих множества). По умолчанию 1000.

Чтобы понять, на что влияет параметр `rows`, рассмотрим пример:

```
CREATE OR REPLACE FUNCTION a() RETURNS SETOF INTEGER AS $$
    SELECT 1;
$$ LANGUAGE SQL;
```

Выполним следующий запрос:

```
car_portal=> EXPLAIN SELECT * FROM a() CROSS JOIN (Values(1),(2),(3)) as
foo;
QUERY PLAN
------------------------------------------------------------------------
 Nested Loop (cost=0.25..47.80 rows=3000 width=8)
    -> Function Scan on a (cost=0.25..10.25 rows=1000 width=4)
    -> Materialize (cost=0.00..0.05 rows=3 width=4)
        -> Values Scan on "*VALUES*" (cost=0.00..0.04 rows=3 width=4)
(4 rows)
```

Значение, возвращаемое SQL-функцией, имеет тип `SETOF INTEGER`, а значит, планировщик ожидает, что функция вернет более одной строки. Поскольку параметр `ROWS` не задан, планировщик берет значение по умолчанию, 1000. И тогда перекрестное соединение `CROSS JOIN` возвращает 3 * 1000 = 3000 строк.

В предыдущем примере неточная оценка некритична. Но в реальных задачах, где имеется несколько соединений, ошибка в оценке числа строк распространяется и усиливается, что ведет к плохим планам выполнения.

Параметр `COST` определяет, когда будет выполняться функция, в частности:

- порядок выполнения функции;
- можно ли перемещать вызов функции в конец.

В следующем примере показано, как параметр `COST` влияет на порядок выполнения функции. Рассмотрим такие две функции:

```
CREATE OR REPLACE FUNCTION slow_function (anyelement) RETURNS BOOLEAN AS $$
BEGIN
  RAISE NOTICE 'Slow function %', $1;
  RETURN TRUE;
END; $$ LANGUAGE PLPGSQL COST 10000;

CREATE OR REPLACE FUNCTION fast_function (anyelement) RETURNS BOOLEAN AS $$
BEGIN
  RAISE NOTICE 'Fast function %', $1;
  RETURN TRUE;
END; $$ LANGUAGE PLPGSQL COST 0.0001;
```

Функции `fast_function` и `slow_function` не отличаются ничем, кроме параметра `cost`:

```
car_portal=> EXPLAIN SELECT * FROM pg_language WHERE fast_function(lanname)
AND slow_function(lanname) AND lanname ILIKE '%sql%';
QUERY PLAN
------------------------------------------------------------------------------
--------------------
 Seq Scan on pg_language (cost=0.00..101.05 rows=1 width=114)
  Filter: (fast_function(lanname) AND (lanname ~~* '%sql%'::text) AND
slow_function(lanname))
(2 rows)

car_portal=# EXPLAIN SELECT * FROM pg_language WHERE slow_function(lanname)
AND fast_function(lanname) AND lanname ILIKE '%sql%';
QUERY PLAN
------------------------------------------------------------------------------
--------------------
 Seq Scan on pg_language (cost=0.00..101.05 rows=1 width=114)
  Filter: (fast_function(lanname) AND (lanname ~~* '%sql%'::text) AND
slow_function(lanname))
(2 rows)
```

Команды отличаются только порядком следования предикатов во фразе `WHERE`. План выполнения обеих команд одинаков. Обратите внимание, в каком порядке вычисляются предикаты в узле Filter плана выполнения. Сначала вычисляется `fast_function`, затем оператор `ILIKE`, и в самом конце `slow_function`. При выполнении этих команд мы получим такой результат:

```
car_portal=> SELECT lanname FROM pg_language WHERE lanname ILIKE '%sql%'
AND slow_function(lanname)AND fast_function(lanname);
NOTICE: Fast function internal
NOTICE: Fast function c
NOTICE: Fast function sql
NOTICE: Slow function sql
NOTICE: Fast function plpgsql
NOTICE: Slow function plpgsql
 lanname
---------
sql
plpgsql
(2 rows)
```

Отметим, что функция fast_function была выполнена четыре раза, а slow_function – только два. Это называется **закорачиванием** вычисления. Функция slow_function выполняется только тогда, когда fast_function и оператор ILIKE возвращают true.

В PostgreSQL оператор ILIKE эквивалентен оператору ~~*, а оператор LIKE – оператору ~~.

В главе 4 мы говорили, что представления можно использовать для реализации авторизации и для сокрытия данных от некоторых пользователей. В ранних версиях postgres параметр COST можно было использовать для взлома представлений, но ситуация улучшилась с появлением параметров LEAKPROOF и SECURITY_BARRIER.

Чтобы параметр COST можно было использовать для получения данных из представления, должно выполняться несколько условий, в том числе:

- параметр COST должен сообщать, что функция очень медленная;
- функция должна быть помечена как LEAKPROOF. Помечать функции таким образом разрешено только суперпользователям;
- для представления не должен быть установлен флаг security_barrier;
- функция должна выполняться, а не игнорироваться из-за закорачивания вычислений.

Удовлетворить все эти условия весьма трудно.

Ниже приведен гипотетический пример взлома представлений. Сначала создадим представление и изменим функцию fast_function, добавив параметр LEAKPROOF.

```
CREATE OR REPLACE VIEW pg_sql_pl AS SELECT lanname FROM pg_language
WHERE lanname ILIKE '%sql%';
ALTER FUNCTION fast_function(anyelement) LEAKPROOF;
Для демонстрации эксплойта выполним запрос:
car_portal=# SELECT * FROM pg_sql_pl WHERE fast_function(lanname);
NOTICE: Fast function internal
NOTICE: Fast function c
NOTICE: Fast function sql
NOTICE: Fast function plpgsql
 lanname
---------
sql
plpgsql
(2 rows)
```

В этом примере само представление не должно показывать функции типа c и internal. Но благодаря параметру COST функция была выполнена до применения фильтра lanname ILIKE '%sql%' и вывела информацию, которую нельзя было бы увидеть через представление.

Поскольку только суперпользователям разрешено помечать функцию флагом LEAKPROOF, эксплойты на основе параметра COST в новых версиях PostgreSQL невозможны.

Параметры функции, относящиеся к конфигурации

Конфигурационные параметры PostgreSQL можно задавать как на уровне сеанса, так и в области видимости функции. Это особенно полезно в тех случаях, когда требуется переопределить параметры сеанса. С помощью параметров можно затребовать ресурсы, например объем памяти, необходимой для выполнения операции (work_mem), или определять поведение, скажем, запрет последовательного просмотра или соединения методом вложенных циклов. Разрешается воздействовать только на параметры, определяемые в контексте пользователя, т. е. те, которые задаются для сеанса работы с пользователем.

В команде SET задается новое значение, принимаемое параметром при входе в функцию. По выходе из функции параметр возвращается к прежнему значению. Конфигурационный параметр можно явно задать для всей функции или локально переопределить внутри функции. Можно также унаследовать значение параметра от сеанса, воспользовавшись ключевым словом CURRENT.

Конфигурационные параметры часто используются для настройки производительности функции в условиях ограниченных ресурсов, при работе с унаследованным или плохо спроектированным кодом, когда статистика дает неверную оценку и т. д. Предположим, к примеру, что функция плохо ведет себя из-за нормализации базы данных. В таком случае перепроектирование базы может оказаться дорогостоящей операцией. Для решения этой проблемы можно было бы изменить план выполнения, включив или выключив некоторые параметры. Допустим, что разработчику нужно быстро исправить следующую команду, которая пользуется дисковой сортировкой слиянием, не изменяя параметра work_mem для сеанса:

```
car_portal=# EXPLAIN (analyze, buffers) SELECT md5(random()::text)
FROM generate_series(1, 1000000) ORDER BY 1;
QUERY PLAN
------------------------------------------------------------------------------
-------------------------------------------------------
 Sort (cost=69.83..72.33 rows=1000 width=32) (actual
time=7324.824..9141.209 rows=1000000 loops=1)
    Sort Key: (md5((random())::text))
    Sort Method: external merge Disk: 42056kB
    Buffers: temp read=10114 written=10113
    -> Function Scan on generate_series (cost=0.00..20.00 rows=1000
width=32) (actual time=193.730..1774.369 rows=1000000 loops=1)
    Buffers: temp read=1710 written=1709
 Planning time: 0.055 ms
 Execution time: 9192.393 ms
(8 rows)
```

В данном случае мы можем обернуть команду SELECT функцией и внутри функции переопределить work_mem:

```
CREATE OR REPLACE FUNCTION configuration_test () RETURNS TABLE(md5 text) AS
$$
    SELECT md5(random()::text) FROM generate_series(1, 1000000) order by 1;
```

```
$$ LANGUAGE SQL
SET enable_seqscan FROM CURRENT
SET work_mem = '100MB';
```

Теперь выполним функции и посмотрим, что дало изменение:

```
car_portal=# EXPLAIN (ANALYZE ,BUFFERS) SELECT * FROM configuration_test();
QUERY PLAN
------------------------------------------------------------------------------
------------------------------------------------------
 Function Scan on configuration_test (cost=0.25..10.25 rows=1000 width=32)
(actual time=7377.196..7405.635 rows=1000000 loops=1)
 Planning time: 0.028 ms
 Execution time: 7444.426 ms
(3 rows)

Time: 7444.984 ms
```

При входе в функцию параметру work_mem присвоено значение 100 Mib, это повлияло на план выполнения, и теперь сортировка выполняется в памяти. Функция выполняется быстрее, чем запрос. Чтобы подтвердить результат, изменим work_mem в сеансе и сравним:

```
car_portal=# set work_mem to '100MB';
SET
Time: 0.343 ms
car_portal=# EXPLAIN (analyze, buffers) SELECT md5(random()::text) FROM
generate_series(1, 1000000) order by 1;
QUERY PLAN
------------------------------------------------------------------------------
------------------------------------------------------
 Sort (cost=69.83..72.33 rows=1000 width=32) (actual
time=7432.831..7615.666 rows=1000000 loops=1)
    Sort Key: (md5((random())::text))
    Sort Method: quicksort Memory: 101756kB
    -> Function Scan on generate_series (cost=0.00..20.00 rows=1000
width=32) (actual time=104.305..1626.839 rows=1000000 loops=1)
 Planning time: 0.050 ms
 Execution time: 7662.650 ms
(6 rows)

Time: 7663.269 ms
```

Команды управления в PL/pgSQL

Структуры управления – важная часть языка PL/pgSQL; благодаря им разработчики могут реализовать очень сложную бизнес-логику внутри PostgreSQL.

Объявления

Синтаксис объявления переменной имеет вид:

```
name [ CONSTANT ] type [ COLLATE collation_name ] [ NOT NULL ]
     [ { DEFAULT | := | = } expression ];
```

- `name`: имя, составленное по правилам, изложенным в главе 3. В частности, имя не может начинаться цифрой.
- `CONSTANT`: такой переменной нельзя присваивать значение после инициализации. Это удобно для определения констант, например PI.
- `type`: переменная может иметь простой тип, скажем `integer`, пользовательский тип, псевдотип, тип `record` и т. д. Поскольку при создании любой таблицы неявно создается тип, мы можем объявить переменную такого типа.

В PostgreSQL следующие два объявления эквивалентны, однако второе лучше совместимо с Oracle. Кроме того, оно яснее выражает намерение разработчика, т. к. его нельзя перепутать со ссылкой на фактическую таблицу. Это предпочтительный способ объявления типа.

- `myrow tablename;`
- `myrow tablename%ROWTYPE;`

- `NOT NULL`: вызывает ошибку при попытке записать в переменную значение null. Переменные, объявленные как `NOT NULL`, должны быть инициализированы.
- `DEFAULT`: откладывает инициализацию переменной до момента входа в блок. Это полезно, когда нужно, чтобы переменная типа `timestamp` получала значение в момент вызова функции, а не в момент ее предварительной компиляции.
- Выражение `expression` – комбинация одного или нескольких явно заданных значений, операторов и функций, вычисление которой дает новое значение.

Тело PL/pgSQL-функции состоит из вложенных блоков с факультативной секцией объявлений и меткой.

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
END [ label ];
```

Ключевые слова `BEGIN` и `END` в этом контексте используются не для управления транзакционным поведением, а для группировки. Секция объявлений предназначена для объявления переменных, а метка `label` – для именования блока и образования полных имен переменных. Наконец, в любой PL/pgSQL-функции имеется скрытый блок, помеченный именем функции, в котором собраны предопределенные переменные, например `FOUND`. Чтобы разобраться в блоке функции, рассмотрим код рекурсивной функции `factorial`:

```
CREATE OR REPLACE FUNCTION factorial(INTEGER ) RETURNS INTEGER AS $$
BEGIN
  IF $1 IS NULL OR $1 < 0 THEN RAISE NOTICE 'Invalid Number';
```

```
    RETURN NULL;
  ELSIF $1 = 1 THEN
    RETURN 1;
  ELSE
    RETURN factorial($1 - 1) * $1;
END IF;
END;
$$ LANGUAGE 'plpgsql';
```

Блок определяет область видимости переменных. В нашем примере областью видимости аргумента $1 является вся функция. Секции объявлений в этой функции нет. Чтобы лучше понять, что такое область видимости различных блоков, перепишем функцию `factorial` немного по-другому:

```
CREATE OR REPLACE FUNCTION factorial(INTEGER ) RETURNS INTEGER AS $$
  DECLARE
    fact ALIAS FOR $1;
  BEGIN
    IF fact IS NULL OR fact < 0 THEN
      RAISE NOTICE 'Invalid Number';
      RETURN NULL;
    ELSIF fact = 1 THEN
      RETURN 1;
    END IF;
    DECLARE
      result INT;
    BEGIN
      result = factorial(fact - 1) * fact;
      RETURN result;
    END;
  END;
$$ LANGUAGE 'plpgsql';
```

Эта функция состоит из двух блоков, переменная `fact` – псевдоним первого аргумента. Во внутреннем блоке объявлена переменная `result` типа `integer`. Поскольку переменная `fact` объявлена в объемлющем блоке, ее можно использовать и во внутреннем. Переменную же `result` можно использовать только во внутреннем блоке.

Присваивание

Операторы := и = служат для присваивания переменной значения выражения:

```
variable { := | = } expression;
```

Имена переменных не должны конфликтовать с именами столбцов. Это важно при написании параметризованных команд SQL. Оператор = поддерживается не только в версии PostgreSQL 10, но и в предыдущих. К сожалению, об этом ничего не сказано в документации по предыдущим версиям. Впрочем, поскольку оператор = используется в SQL для сравнения, то во избежание недоразумений лучше для присваивания использовать оператор :=.

В некоторых контекстах допускается только один оператор присваивания:

- когда присваивается значение по умолчанию, обязательно использовать оператор =, о чем написано в документации по адресу http://www.postgresql.org/docs/current/interactive/sql-createfunction.html;
- для обозначения именованных параметров при вызове функции используется только оператор :=.

В примере ниже показан случай, когда = и := не являются синонимами.

```
CREATE OR REPLACE FUNCTION cast_numeric_to_int (numeric_value numeric,
round boolean = TRUE /* допустимо только "=". Использование ":=" приведет к синтаксической
ошибке */)
RETURNS INT AS
$$
BEGIN
    RETURN (CASE WHEN round = TRUE THEN CAST (numeric_value AS INTEGER)
      WHEN numeric_value>= 0 THEN CAST (numeric_value -.5 AS INTEGER)
      WHEN numeric_value< 0 THEN CAST (numeric_value +.5 AS INTEGER)
      ELSE NULL
    END);
END;
$$ LANGUAGE plpgsql;
```

Протестируем присваивание:

```
car_portal=# SELECT cast_numeric_to_int(2.3, round:= true);
 cast_numeric_to_int
---------------------
                   2
(1 row)

car_portal=# SELECT cast_numeric_to_int(2.3, round= true);
ERROR: column "round" does not exist
LINE 1: SELECT cast_numeric_to_int(2.3, round= true);
```

Выражение присваивания может быть одним атомарным значением, например `pi = 3.14`, или строкой:

```
DO $$
  DECLARE
    test record;
  BEGIN
    test = ROW (1,'hello', 3.14);
    RAISE notice '%', test;
  END;
$$ LANGUAGE plpgsql;

DO $$
DECLARE
  number_of_accounts INT:=0;
BEGIN
  number_of_accounts:= (SELECT COUNT(*) FROM car_portal_app.account)::INT;
  RAISE NOTICE 'number_of accounts: %', number_of_accounts;
END;$$
LANGUAGE plpgsql;
```

Существуют и другие способы присваивания значений переменным по результатам запроса, возвращающего одну строку:

```
SELECT select_expressions INTO [STRICT] targets FROM ...;
INSERT ... RETURNING expressions INTO [STRICT] targets;
UPDATE ... RETURNING expressions INTO [STRICT] targets;
DELETE ... RETURNING expressions INTO [STRICT] targets;
```

Часто в роли выражений выступают имена столбцов, а целями (`targets`) являются имена переменных. В случае команды `SELECT INTO` цель может иметь тип `record`. Команда `INSERT ... RETURNING` нередко используется для того, чтобы вернуть значение некоторого столбца по умолчанию, например когда первичный ключ имеет тип `SERIAL` или `BIGSERIAL`:

```
CREATE TABLE test (
  id SERIAL PRIMARY KEY,
  name TEXT NOT NULL
);
```

Для проверки присваивания выполним следующий код:

```
DO $$
  DECLARE
    auto_generated_id INT;
  BEGIN
    INSERT INTO test(name) VALUES ('Hello World') RETURNING id
    INTO auto_generated_id;
    RAISE NOTICE 'The primary key is: %', auto_generated_id;
  END
$$;
NOTICE: The primary key is: 1
DO
```

Значение по умолчанию можно было бы получить и после вставки строки на чистом SQL, воспользовавшись CTE:

```
WITH get_id AS (
 INSERT INTO test(name) VALUES ('Hello World')
 RETURNING id
) SELECT * FROM get_id;
```

Наконец, для выполнения присваивания можно было бы использовать полные имена; в триггерных функциях для манипулирования значениями до и после выполнения операции употребляются имена таблиц `OLD` и `NEW`.

Условные команды

PostgreSQL поддерживает команды `IF` и `CASE`, что позволяет выполнять команды в зависимости от условия. Конструкция IF выглядит следующим образом:

- `IF ... THEN ... END IF`
- `IF ... THEN ... ELSE ... END IF`
- `IF ... THEN ... ELSIF ... THEN ... ELSE ... END IF`

У команды CASE есть две формы:

- CASE ... WHEN ... THEN ... ELSE ... END CASE
- CASE WHEN ... THEN ... ELSE ... END CASE

Чтобы лучше освоить команду IF, рассмотрим задачу о текстовом представлении ранга объявления:

```
CREATE OR REPLACE FUNCTION cast_rank_to_text (rank int) RETURNS TEXT AS $$
DECLARE
  rank ALIAS FOR $1;
  rank_result TEXT;
BEGIN
  IF rank = 5 THEN rank_result = 'Excellent';
  ELSIF rank = 4 THEN rank_result = 'Very Good';
  ELSIF rank = 3 THEN rank_result = 'Good';
  ELSIF rank = 2 THEN rank_result ='Fair';
  ELSIF rank = 1 THEN rank_result ='Poor';
  ELSE rank_result ='No such rank';
  END IF;
  RETURN rank_result;
END;
$$ Language plpgsql;
--- для тестирования функции
SELECT n,cast_rank_to_text(n) FROM generate_series(1,6) as foo(n);
```

После выполнения любой ветви IF управление передается команде, следующей за END IF, если только внутри ветви нет команды RETURN. Если не выполнена ни одна из ветвей IF и ELSIF, то выполняется ветвь ELSE. Отметим также, что все структуры управления могут быть вложенными, т. е. одна команда IF может находиться внутри другой.

В следующем фрагменте та же логика реализована с помощью команды CASE:

```
CREATE OR REPLACE FUNCTION cast_rank_to_text (rank int) RETURNS TEXT AS
$$
DECLARE
  rank ALIAS FOR $1;
  rank_result TEXT;
BEGIN
  CASE rank
    WHEN 5 THEN rank_result = 'Excellent';
    WHEN 4 THEN rank_result = 'Very Good';
    WHEN 3 THEN rank_result = 'Good';
    WHEN 2 THEN rank_result ='Fair';
    WHEN 1 THEN rank_result ='Poor';
    ELSE rank_result ='No such rank';
  END CASE;
  RETURN rank_result;
END;$$ Language plpgsql;
```

После того как некоторая ветвь CASE выполнена, управление сразу передается команде, следующей за CASE. Отметим также, что в этой форме CASE запрещено сопоставление со значением NULL, потому что результатом сравнения с NULL

является NULL. Чтобы обойти это ограничение, можно воспользоваться второй формой и задать условие сопоставления явно:

```
CREATE OR REPLACE FUNCTION cast_rank_to_text (rank int) RETURNS TEXT AS $$
DECLARE
  rank ALIAS FOR $1;
  rank_result TEXT;
BEGIN
  CASE
    WHEN rank=5 THEN rank_result = 'Excellent';
    WHEN rank=4 THEN rank_result = 'Very Good';
    WHEN rank=3 THEN rank_result = 'Good';
    WHEN rank=2 THEN rank_result ='Fair';
    WHEN rank=1 THEN rank_result ='Poor';
    WHEN rank IS NULL THEN RAISE EXCEPTION 'Rank should be not NULL';
    ELSE rank_result ='No such rank';
  END CASE;
  RETURN rank_result;
END;
$$ Language plpgsql;
--- тестируем
SELECT cast_rank_to_text(null);
```

Наконец, если ни одна ветвь CASE не подошла и ветвь ELSE отсутствует, то возбуждается исключение:

```
DO $$
DECLARE
  i int := 0;
BEGIN
  CASE WHEN i=1 then
    RAISE NOTICE 'i is one';
  END CASE;
END;
$$ LANGUAGE plpgsql;
ERROR: case not found
HINT: CASE statement is missing ELSE part.
CONTEXT: PL/pgSQL function inline_code_block line 5 at CASE
```

Итерирование

Итерирование позволяет повторять блок команд. При этом часто требуется определить начальную точку и условие окончания. В PostgreSQL есть несколько команд для обхода результатов запроса и организации циклов: LOOP, CONTINUE, EXIT, FOR, WHILE и FOR EACH.

Команда LOOP

Самая простая команда LOOP выглядит так:

```
[ <<label>> ]
LOOP
    statements
END LOOP [ label ];
```

Для демонстрации перепишем функцию factorial:

```
DROP FUNCTION IF EXISTS factorial (int);
CREATE OR REPLACE FUNCTION factorial (fact int) RETURNS BIGINT AS $$
DECLARE
  result bigint = 1;
BEGIN
  IF fact = 1 THEN RETURN 1;
  ELSIF fact IS NULL or fact < 1 THEN RAISE EXCEPTION 'Provide a positive integer';
  ELSE
    LOOP
      result = result*fact;
      fact = fact-1;
      EXIT WHEN fact = 1;
    END LOOP;
  END IF;
  RETURN result;
END; $$ LANGUAGE plpgsql;
```

Здесь условная команда EXIT предотвращает зацикливание. После выполнения EXIT управление передается команде, следующей за LOOP. Для управления выполнением команд внутри цикла LOOP PL/pgSQL предоставляет также команду CONTINUE, которая похожа на EXIT, только вместо выхода из цикла передает управление на его начало, пропуская код, оставшийся до конца цикла.

Рекомендуется избегать команд CONTINUE и EXIT, особенно в середине блока, потому что они нарушают порядок выполнения, что затрудняет чтение кода.

Команда WHILE

Команда WHILE продолжает выполнять блок команд, пока истинно заданное условие. Вот ее синтаксис:

```
[ <<label>> ]
WHILE boolean-expression LOOP
    statements
END LOOP [ label ];
```

Ниже мы печатаем дни текущего месяца в цикле while:

```
DO $$
DECLARE
  first_day_in_month date := date_trunc('month', current_date)::date;
  last_day_in_month date := (date_trunc('month', current_date)+
      INTERVAL '1 MONTH - 1 day')::date;
  counter date = first_day_in_month;
BEGIN
  WHILE (counter <= last_day_in_month) LOOP
    RAISE notice '%', counter;
    counter := counter + interval '1 day';
  END LOOP;
END;
$$ LANGUAGE plpgsql;
```

Команда FOR

В PL/pgSQL есть две формы команды FOR, предназначенные для:

- обхода строк, возвращенных SQL-запросом;
- обхода диапазона целых чисел.

Синтаксис команды FOR представлен ниже:

```
[ <<label>> ]
FOR name IN [ REVERSE ] expression1 .. expression2 [ BY expression ] LOOP
    statements
END LOOP [ label ];
```

Здесь name – имя локальной переменной типа integer. Областью видимости этой переменной является цикл FOR. Команды внутри цикла могут ее читать, но не могут изменять. Но это поведение можно изменить, определив переменную в секции объявлений объемлющего блока. Выражения expression1 и expression2 должны иметь целые значения. Если expression1 равно expression2, то цикл FOR выполняется только один раз.

Ключевое слово REVERSE необязательно; оно определяет порядок, в котором генерируется диапазон (возрастающий или убывающий). Если слово REVERSE опущено, то expression1 должно быть меньше expression2, иначе цикл не выполнится ни разу. Наконец, ключевое слово BY задает шаг между двумя последовательными числами; следующее за ним выражение expression должно быть положительным целым числом. В показанном ниже цикле FOR производится обход диапазона отрицательных чисел в обратном порядке и печатаются значения -1, -3, …, -9.

```
DO $$
BEGIN
  FOR j IN REVERSE -1 .. -10 BY 2 LOOP
    RAISE NOTICE '%', j;
  END LOOP;
END; $$
LANGUAGE plpgsql;
```

Синтаксис цикла для обхода результатов запроса несколько иной:

```
[ <<label>> ]
FOR target IN query LOOP
    statements
END LOOP [ label ];
```

Локальная переменная target объявлена в объемлющем блоке. Она необязательно должна быть примитивного типа, скажем, integer или text, допускается также составной тип и тип RECORD. В PL/pgSQL обходить можно результат выполнения курсора или команды SELECT. Курсор – это специальный объект, инкапсулирующий запрос SELECT и позволяющий читать результаты по нескольку строк за раз. В следующем примере печатаются имена всех баз данных:

```
DO $$
DECLARE
```

```
    database RECORD;
BEGIN
    FOR database IN SELECT * FROM pg_database LOOP
      RAISE notice '%', database.datname;
    END LOOP;
END; $$;
----- выводится
NOTICE: postgres
NOTICE: template1
NOTICE: template0
....
DO
```

Возврат из функции

Команда возврата завершает функцию и передает управление вызывающей стороне. Существует несколько разновидностей команды возврата: `RETURN`, `RETURN NEXT`, `RETURN QUERY`, `RETURN QUERY EXECUTE` и т. д. Команда `RETURN` может возвращать одно значение или множество, как будет показано ниже. Рассмотрим с этой точки зрения анонимную функцию:

```
DO $$
BEGIN
    RETURN;
    RAISE NOTICE 'This statement will not be executed';
END
$$
LANGUAGE plpgsql;
--- выводится
DO
```

Как видим, из-за `RETURN` эта функция завершается еще до выполнения команды `RAISE`.

Возврат void

Тип `void` используется, когда смысл функции заключается в побочных эффектах, например записи в журнал. Встроенная функция `pg_sleep` задерживает выполнение серверного процесса на заданное число секунд:

```
postgres=# \df pg_sleep
List of functions
 Schema     | Name     | Result data type | Argument data types | Type
------------+----------+------------------+---------------------+--------
 pg_catalog | pg_sleep | void             | double precision    | normal
(1 row)
```

Возврат одной строки

Функция PL/pgSQL может возвращать одну строку, примером служит функция `factorial`.

Некоторые разработчики называют скалярными функции на PL/pgSQL и на SQL, которые возвращают одну строку, содержащую один столбец со скалярным значением.

Тип возвращаемого значения может быть базовым, составным, доменным или псевдотипом. Показанная ниже функция возвращает представление учетной записи в формате JSON:

```
-- на sql
CREATE OR REPLACE FUNCTION car_portal_app.get_account_in_json (account_id
INT) RETURNS JSON AS $$
  SELECT row_to_json(account) FROM car_portal_app.account WHERE account_id = $1;
$$ LANGUAGE SQL;

--- на plpgsql
CREATE OR REPLACE FUNCTION car_portal_app.get_account_in_json1 (acc_id INT)
RETURNS JSON AS $$
BEGIN
  RETURN (SELECT row_to_json(account) FROM car_portal_app.account
      WHERE account_id = acc_id);
END;
$$ LANGUAGE plpgsql;
```

Возврат нескольких строк

Для возврата нескольких строк используются функции, возвращающие множества. Строка может иметь базовый тип (например, integer), составной, табличный, доменный или псевдотип. Функция, возвращающая множество, помечается ключевым словом SETOF:

```
-- На SQL
CREATE OR REPLACE FUNCTION car_portal_app.car_model(model_name TEXT)
RETURNS SETOF car_portal_app.car_model AS $$
    SELECT car_model_id, make, model FROM car_portal_app.car_model
    WHERE model = model_name;
$$ LANGUAGE SQL;

-- На plpgSQL
CREATE OR REPLACE FUNCTION car_portal_app.car_model1(model_name TEXT)
RETURNS SETOF car_portal_app.car_model AS $$
BEGIN
  RETURN QUERY SELECT car_model_id, make, model FROM car_portal_app.car_model
  WHERE model = model_name;
END;
$$ LANGUAGE plpgsql;
```

Выполним эти функции. Обратите внимание, как кеширование, осуществляемое в PL/pgSQL, влияет на производительность:

```
car_portal=> \timing
Timing is on.
car_portal=> SELECT * FROM car_portal_app.car_model('A1');
 car_model_id | make | model
--------------+------+-------
            1 | Audi | A1
(1 row)

Time: 1,026 ms

car_portal=> SELECT * FROM car_portal_app.car_model1('A1');
```

```
 car_model_id | make | model
--------------+------+-------
            1 | Audi | A1
(1 row)

Time: 0,546 ms
```

Если для возвращаемого значения еще нет типа, то можно:

- определить и использовать новый тип данных;
- использовать тип TABLE;
- использовать выходные параметры и тип данных record.

Пусть требуется вернуть только поля car_model_id и make, и соответствующий тип данных не определен. Тогда предыдущую функцию можно переписать в виде:

```
-- SQL
CREATE OR REPLACE FUNCTION car_portal_app.car_model2(model_name TEXT)
RETURNS TABLE (car_model_id INT , make TEXT) AS $$
    SELECT car_model_id, make FROM car_portal_app.car_model
    WHERE model = model_name;
$$ LANGUAGE SQL;

-- plpgSQL
CREATE OR REPLACE FUNCTION car_portal_app.car_model3(model_name TEXT)
RETURNS TABLE (car_model_id INT , make TEXT) AS $$
BEGIN
    RETURN QUERY SELECT car_model_id, make FROM car_portal_app.car_model
    WHERE model = model_name;
END;
$$ LANGUAGE plpgsql;
```

Протестируем эту функцию:

```
car_portal=> SELECT * FROM car_portal_app.car_model2('A1');
 car_model_id | make
--------------+------
            1 | Audi
(1 row)

Time: 0,797 ms

car_portal=> SELECT * FROM car_portal_app.car_model3('A1');
ERROR: column reference "car_model_id" is ambiguous
LINE 1: SELECT car_model_id, make FROM car_portal_app.car_model WHE...
               ^
DETAIL: It could refer to either a PL/pgSQL variable or a table column.
QUERY: SELECT car_model_id, make FROM car_portal_app.car_model WHERE model
= model_name
CONTEXT: PL/pgSQL function car_model3(text) line 3 at RETURN QUERY
Time: 0,926 ms
```

Ошибка возникла, потому что PL/pgSQL спутал имя столбца с определением таблицы. Дело в том, что возврат таблицы – сокращенная запись выходных параметров. Чтобы исправить ошибку, нужно переименовать атрибуты:

```
CREATE OR REPLACE FUNCTION car_portal_app.car_model3(model_name TEXT)
RETURNS TABLE (car_model_id INT , make TEXT) AS $$
BEGIN
    RETURN QUERY SELECT a.car_model_id, a.make FROM car_portal_app.car_model a
    WHERE model = model_name;
END;
$$ LANGUAGE plpgsql;
car_portal=> SELECT * FROM car_portal_app.car_model3('A1');
 car_model_id | make
--------------+------
            1 | Audi
(1 row)
```

Эту функцию можно также записать с помощью выходных параметров с ключевым словом OUT; именно так на самом деле и реализован возврат таблицы:

```
CREATE OR REPLACE FUNCTION car_portal_app.car_model4(model_name TEXT, OUT
car_model_id INT, OUT make TEXT ) RETURNS SETOF RECORD AS $$
BEGIN
  RETURN QUERY SELECT a.car_model_id, a.make FROM car_portal_app.car_model a
  WHERE model = model_name;
END;
$$ LANGUAGE plpgsql;

car_portal=> SELECT * FROM car_portal_app.car_model4('A1'::text);
 car_model_id | make
--------------+------
            1 | Audi
(1 row)
```

Предопределенные переменные в функциях

В PL/pgSQL-функциях имеется несколько специальных переменных, которые автоматически создаются в блоке верхнего уровня. Например, в триггерной функции создаются переменные NEW, OLD и TG_OP.

Помимо специальных триггерных переменных, существует булева переменная FOUND. Команды SELECT, INSERT, UPDATE, DELETE и PERFORM устанавливают ее в true, если, соответственно, выбрана, вставлена, обновлена или удалена хотя бы одна строка.

Команда PERFORM похожа на SELECT, но отбрасывает результат запроса. Отметим также, что команда EXECUTE не изменяет значения FOUND. В следующих примерах показано, как отражаются на FOUND действия, выполненные командами INSERT и PERFORM:

```
DO $$
BEGIN
  CREATE TABLE t1(f1 int);

  INSERT INTO t1 VALUES (1);
  RAISE NOTICE '%', FOUND;
```

```
  PERFORM * FROM t1 WHERE f1 = 0;
  RAISE NOTICE '%', FOUND;
  DROP TABLE t1;
END;
$$LANGUAGE plpgsql;
--- выводится
NOTICE: t
NOTICE: f
```

В предыдущем запросе можно было бы также получить OID (идентификатор объекта) последней вставленной строки, а также количество строк, затронутых командами INSERT, UPDATE и DELETE. Для этого служит команда

```
GET DIAGNOSTICS variable = item;
```

В предположении, что объявлена переменная i типа integer, количество затронутых строк можно получить следующим образом:

```
GET DIAGNOSTICS i = ROW_COUNT;
```

Обработка исключений

Для перехвата и возбуждения исключений в PostgreSQL предназначены команды EXCEPTION и RAISE. Ошибки возникают при нарушении ограничений целостности данных или при выполнении недопустимых операций, например: присваивание текста целому числу, деление на нуль, присваивание переменной значения вне допустимого диапазона и т. д. По умолчанию любая ошибка в PL/pgSQL-функции приводит к прерыванию выполнения и откату изменений. Чтобы восстановиться, можно перехватить ошибку с помощью фразы EXCEPTION, синтаксически очень похожей на блоки в PL/pgSQL. Кроме того, ошибки можно возбуждать в команде RAISE. Чтобы разобраться во всем этом, рассмотрим следующую вспомогательную функцию:

```
CREATE OR REPLACE FUNCTION check_not_null (value anyelement ) RETURNS VOID
AS
$$
BEGIN
    IF (value IS NULL) THEN
      RAISE EXCEPTION USING ERRCODE = 'check_violation';
    END IF;
END;
$$ LANGUAGE plpgsql;
```

Полиморфная функция check_not_null просто возбуждает ошибку, в которой состояние SQLSTATE содержит строку check_violation. Если вызвать эту функцию, передав ей NULL в качестве аргумента, то возникнет ошибка:

```
car_portal=> SELECT check_not_null(null::text);
ERROR: check_violation
CONTEXT: PL/pgSQL function check_not_null(anyelement) line 3 at RAISE
Time: 0,775 ms
```

Чтобы можно было точно узнать, когда и почему возникло исключение, в PostgreSQL определено несколько категорий кодов ошибок. Полный перечень приведен на странице http://www.postgresql.org/docs/current/interactive/errcodes-appendix.html. Например, если пользователь возбудил исключение, не указав ERRCODE, то SQLSTATE будет содержать строку P001, а если нарушено ограничение уникальности, то строку 23505.

Для определения причины ошибки в команде EXCEPTION можно сравнивать SQLSTATE или название условия:

```
WHEN unique_violation THEN ...
WHEN SQLSTATE '23505' THEN ...
```

Наконец, при возбуждении ошибки можно задать сообщение об ошибке и SQLSTATE, памятуя о том, что код ошибки ERRCODE должен содержать 5 цифр и (или) букв в кодировке ASCII, но не должен быть равен 00000, например:

```
DO $$
BEGIN
  RAISE EXCEPTION USING ERRCODE = '1234X', MESSAGE = 'test customized SQLSTATE:';
  EXCEPTION WHEN SQLSTATE '1234X' THEN
    RAISE NOTICE '% %', SQLERRM, SQLSTATE;
END;
$$ LANGUAGE plpgsql;
```

В результате выполнения этой анонимной функции печатается:

```
NOTICE: test customized SQLSTATE: 1234X
DO
Time: 0,943 ms
```

Для демонстрации перехвата исключений перепишем функцию factorial так, чтобы она возвращала null, если передан аргумент null:

```
DROP FUNCTION IF EXISTS factorial( INTEGER );
CREATE OR REPLACE FUNCTION factorial(INTEGER ) RETURNS BIGINT AS $$
DECLARE
  fact ALIAS FOR $1;
BEGIN
  PERFORM check_not_null(fact);
  IF fact > 1 THEN RETURN factorial(fact - 1) * fact;
  ELSIF fact IN (0,1) THEN RETURN 1;
  ELSE RETURN NULL;
  END IF;

  EXCEPTION
    WHEN check_violation THEN RETURN NULL;
    WHEN OTHERS THEN RAISE NOTICE '% %', SQLERRM, SQLSTATE;
  END;
$$ LANGUAGE 'plpgsql';
```

Протестируем эту функцию:

```
car_portal=> \pset null 'null'
Null display is "null".
```

```
car_portal=> SELECT * FROM factorial(null::int);
 factorial
-----------
 null
(1 row)
Time: 1,018 ms
```

Функция factorial не возбудила ошибку, поскольку она была перехвачена в команде EXCEPTION и вместо нее возвращено NULL. Обратите внимание, что сравнивается не SQLSTATE, а имя условия. Специальное имя условия OTHERS соответствует любой ошибке, оно часто используется, чтобы перехватить неожиданные ошибки.

Если SQLERRM и SQLSTATE не позволяют достаточно точно определить природу исключения, то можно получить дополнительную информацию с помощью команды GET STACKED DIAGNOSTICS:

```
GET STACKED DIAGNOSTICS variable { = | := } item [ , ... ];
```

Здесь item – ключевое слово, идентифицирующее некоторый аспект исключения. Например, ключевые слова COLUMN_NAME, TABLE_NAME и SCHEMA_NAME обозначают, соответственно, имя столбца, таблицы и схемы.

Динамический SQL

Динамический SQL служит для построения и выполнения запросов «на лету». В отличие от статической команды SQL, полный текст динамической команды заранее неизвестен и может меняться от выполнения к выполнению. Допускаются команды подъязыков DDL, DCL и DML. Динамический SQL позволяет сократить объем однотипных задач. Например, таким образом можно без особых усилий ежедневно создавать секционирование некоторой таблицы, добавлять отсутствующие индексы по внешним ключам или средства аудита данных. Еще одно важное применение динамического SQL – устранение побочных эффектов кеширования в PL/pgSQL, поскольку запросы, выполняемые с помощью команды EXECUTE, не кешируются.

Для реализации динамического SQL предназначена команда EXECUTE. Она принимает и интерпретирует строку. Синтаксис этой команды показан ниже:

```
EXECUTE command-string [ INTO [STRICT] target ] [ USING expression [, ...] ];
```

Динамическое выполнение команд DDL

Иногда бывает необходимо выполнить операции на уровне объектов базы данных: таблиц, индексов, столбцов, ролей и т. д. Например, после развертывания таблицы с целью обновления статистики часто производят ее очистку и анализ. Так, для анализа таблиц из схемы car_portal_app можно было бы написать такой скрипт:

```
DO $$
DECLARE
  table_name text;
BEGIN
  FOR table_name IN SELECT tablename FROM pg_tables
  WHERE schemaname ='car_portal_app'
  LOOP
      RAISE NOTICE 'Analyzing %', table_name;
      EXECUTE 'ANALYZE car_portal_app.' || table_name;
  END LOOP;
END;
$$;
```

Динамическое выполнение команд DML

Некоторые приложения работают с данными интерактивно. Например, ежемесячно могут генерироваться данные для выставления счетов. Бывает и так, что приложение фильтрует данные на основе критерия, заданного пользователем. Для таких задач очень удобен динамический SQL. На сайте торговли автомобилями необходима возможность динамически искать учетные записи по заданному предикату:

```
CREATE OR REPLACE FUNCTION car_portal_app.get_account (predicate TEXT)
RETURNS SETOF car_portal_app.account AS
$$
BEGIN
    RETURN QUERY EXECUTE 'SELECT * FROM car_portal_app.account WHERE ' ||
predicate;
END;
$$ LANGUAGE plpgsql;
```

Протестируем эту функцию:

```
car_portal=> SELECT * FROM car_portal_app.get_account ('true') limit 1;
 account_id | first_name | last_name | email           | password
------------+------------+-----------+-----------------+------------------
---------------
          1 | James      | Butt      | jbutt@gmail.com |
1b9ef408e82e38346e6ebebf2dcc5ece
(1 row)

car_portal=> SELECT * FROM car_portal_app.get_account
(E'first_name=\'James\'');
 account_id | first_name | last_name | email           | password
------------+------------+-----------+-----------------+------------------
---------------
          1 | James      | Butt      | jbutt@gmail.com |
1b9ef408e82e38346e6ebebf2dcc5ece
(1 row)
```

Динамический SQL и кеширование

Как уже было сказано, PL/pgSQL кеширует планы выполнения. Это хорошо, если сгенерированный план статический. Например, для следующей команды ожидается, что всегда будет использоваться просмотр индекса благодаря его избирательности. В таком случае кеширование плана экономит время и повышает производительность.

```
SELECT * FROM account WHERE account_id =<INT>
```

Но бывает и по-другому. Предположим, что построен индекс по столбцу `advertisement_date`, и мы хотим получить количество объявлений, начиная с некоторой даты:

```
SELECT count (*) FROM car_portal_app.advertisement
WHERE advertisement_date >= <certain_date>;
```

Для чтения записей из таблицы `advertisement` можно применить как просмотр индекса, так и последовательный просмотр. Всё зависит от избирательности индекса, которая определяется значением `certain_date`. Кеширование плана выполнения такого запроса может создать серьезные проблемы, поэтому показанный ниже вариант функции следует признать неудачным:

```
CREATE OR REPLACE FUNCTION car_portal_app.get_advertisement_count
(some_date timestamptz ) RETURNS BIGINT AS $$
BEGIN
  RETURN (SELECT count (*) FROM car_portal_app.advertisement
  WHERE advertisement_date >=some_date)::bigint;
END;
$$ LANGUAGE plpgsql;
```

Для решения проблемы мы можем переписать функцию, указав в качестве языка SQL, или оставить PL/pgSQL, но воспользоваться командой `EXECUTE`:

```
CREATE OR REPLACE FUNCTION car_portal_app.get_advertisement_count
(some_date timestamptz ) RETURNS BIGINT AS $$
DECLARE
  count BIGINT;
BEGIN
  EXECUTE 'SELECT count (*) FROM car_portal_app.advertisement
  WHERE advertisement_date >= $1' USING some_date INTO count;
  RETURN count;
END;
$$ LANGUAGE plpgsql;
```

Рекомендации по использованию динамического SQL

Динамический SQL может стать причиной серьезных проблем, если пользоваться им бездумно, поскольку уязвим для атак внедрением SQL. Цель подобных атак – выполнить команду, которая выдаст секретную информацию или даже уничтожит данные в базе. Вот очень простой пример PL/pgSQL-функции, уязвимой к внедрению SQL:

```
CREATE OR REPLACE FUNCTION car_portal_app.can_login (email text, pass text)
RETURNS BOOLEAN AS $$
DECLARE
  stmt TEXT;
  result bool;
BEGIN
  stmt = E'SELECT COALESCE (count(*)=1, false) FROM car_portal_app.account
WHERE email = \''|| $1 || E'\' and password = \''||$2||E'\'';
  RAISE NOTICE '%' , stmt;
  EXECUTE stmt INTO result;
  RETURN result;
END;
$$ LANGUAGE plpgsql;
```

Эта функция возвращает `true`, если адрес электронной почты совпадает с паролем. Для ее проверки вставим строку и внедрим какой-нибудь код:

```
car_portal=> SELECT car_portal_app.can_login('jbutt@gmail.com',
md5('jbutt@gmail.com'));
NOTICE: SELECT COALESCE (count(*)=1, false) FROM account WHERE email =
'jbutt@gmail.com' and password = '1b9ef408e82e38346e6ebebf2dcc5ece'
 can_login
-----------
 t
(1 row)

car_portal=> SELECT car_portal_app.can_login('jbutt@gmail.com',
md5('jbutt@yahoo.com'));
NOTICE: SELECT COALESCE (count(*)=1, false) FROM account WHERE email =
'jbutt@gmail.com' and password = '37eb43e4d439589d274b6f921b1e4a0d'
 can_login
-----------
 f
(1 row)

car_portal=> SELECT car_portal_app.can_login(E'jbutt@gmail.com\'--', 'Do
not know password');
NOTICE: SELECT COALESCE (count(*)=1, false) FROM account WHERE email =
'jbutt@gmail.com'--' and password = 'Do not know password'
 can_login
-----------
 t
(1 row)
```

Отметим, что функция возвращает `true`, даже если пароль не совпадает с паролем, хранящимся в таблице, – просто потому, что предикат закомментирован, как видно из напечатанного сообщения:

```
SELECT COALESCE (count(*)=1, false) FROM account WHERE email =
'jbutt@gmail.com'--' and password = 'Do not know password'
```

Чтобы защитить код от таких атак, соблюдайте следующие рекомендации:

- для параметризованных динамических SQL-команд используйте фразу `USING`;
- используйте для построения запросов функцию `format`. Отметим, что спецификатор `%I` экранирует аргумент как идентификатор, а `%L` – как литерал;

- для правильного форматирования идентификаторов и литералов используйте функции `quote_ident()`, `quote_literal()` и `quote_nullable()`.

Приведем один из способов безопасного написания показанной выше функции:

```
CREATE OR REPLACE FUNCTION car_portal_app.can_login (email text, pass text)
RETURNS BOOLEAN AS
$$
DECLARE
  stmt TEXT;
  result bool;
BEGIN
  stmt = format('SELECT COALESCE (count(*)=1, false) FROM
car_portal_app.account WHERE email = %Land password = %L', $1,$2);
  RAISE NOTICE '%' , stmt;
  EXECUTE stmt INTO result;
  RETURN result;
END;
$$ LANGUAGE plpgsql;
```

Резюме

PostgreSQL предоставляет полнофункциональный язык программирования PL/pgSQL, тесно интегрированный с системой триггеров. Языки PL/pgSQL и SQL можно использовать для реализации очень сложной логики. По возможности следует пользоваться функциями на SQL. С появлением таких продвинутых средств, как оконные функции и латеральные соединения, стало возможно решать весьма сложные задачи, не выходя за пределы стандартного SQL.

В PostgreSQL существует несколько параметров для управления поведением функций, они в равной мере относятся к функциям на PL/pgSQL и SQL. Например, параметры `SECURITY DEFINER` и `SECURITY INVOKER` определяют контекст безопасности и привилегии. Параметры `COST`, `LEAKPROOF` и `ROWS` помогают планировщику генерировать планы выполнения. К функциям также применимы некоторые конфигурационные параметры.

В языке PL/pgSQL имеется полный набор средств для присваивания, условного выполнения, организации циклов и обработки исключений. Функции могут возвращать псевдотип `VOID`, скалярные значения, записи и т. д. Кроме того, поддерживаются параметры вида `IN`, `OUT` и `INOUT`.

Динамический SQL позволяет конструировать SQL-команды во время выполнения. Можно создавать гибкие функции общего назначения, потому что полный текст SQL-команды неизвестен до этапа компиляции. Но с динамическим SQL нужно обращаться осторожно, поскольку он уязвим для атак внедрением SQL.

В следующей главе мы обсудим стратегии моделирования для OLAP-приложений, а также опишем некоторые средства PostgreSQL, которые делают ее подходящей СУБД для таких приложений. К ним, в частности, относится SQL-команда `COPY`, а также стратегии секционирования данных посредством наследования.

Глава 8

OLAP и хранилища данных

База данных обычно играет роль хранилища в комплексном программном решении. В зависимости от типа решения и решаемой задачи конфигурация базы и структура данных могут различаться. Обычно база данных настраивается для работы в одном из двух режимов: **оперативной транзакционной обработки** (online transaction processing – **OLTP**) и **оперативного анализа данных** (online analytical processing – **OLAP**).

Если база данных работает в качестве серверной части приложения, то требуется OLTP-решение. Это означает, что база регулярно будет выполнять много мелких транзакций. Сайт торговли автомобилями, с которым мы имели дело в предыдущих главах, является типичным примером структуры данных для OLTP. Приложение, работающее с такой базой данных, выполняет транзакцию всякий раз, как пользователь что-то делает: создает учетную запись, изменяет пароль, вводит в систему новый автомобиль, создает или изменяет объявление и т. д. Любое действие такого рода подразумевает транзакцию в базе, в результате которой создается, обновляется или удаляется одна или несколько строк в одной или нескольких таблицах. Чем больше пользователей работает с системой, тем чаще выполняются транзакции. База данных должна справляться с нагрузкой, а ее производительность измеряется количеством транзакций в секунду. Объем данных обычно не является первостепенным фактором.

Перечислим ключевые характеристики базы данных, работающей в режиме OLTP:

- нормализованная структура;
- относительно небольшой объем данных;
- относительно большое количество транзакций;
- каждая транзакция невелика, затрагивает одну или несколько записей;
- пользователи обычно выполняют всевозможные операции с данными: выборку, вставку, удаление и обновление.

Если база данных играет роль источника данных для отчетов и для анализа, то требуется OLAP-решение. Это прямая противоположность OLTP: данных много, но изменяются они редко. Количество запросов сравнительно мало, но сами запросы большие и сложные, они требуют чтения и агрегирования огромного количества данных. Производительность измеряется временем выполнения запросов. OLAP-решения часто называют **хранилищами**, или **складами данных**.

База данных, работающая в режиме OLAP, обладает следующими характеристиками:

- денормализованная структура;
- относительно большой объем данных;
- относительно небольшое количество транзакций;
- каждая транзакция велика, затрагивает миллионы записей;
- пользователи обычно выполняют только выборку.

В этой главе мы обсудим некоторые свойства PostgreSQL, помогающие в реализации хранилищ данных, а именно:

- онлайновая аналитическая обработка: что это такое и что в ней особенного;
- секционирование: способ физической организации очень больших таблиц и управления ими;
- параллельные запросы: метод ускорения выполнения запроса;
- просмотр только индексов: способ повысить производительность путем правильного построения индексов.

Мы расширим базу данных `car_portal` – добавим в нее еще одну схему, `dwh`, и создадим в ней несколько таблиц. Саму схему и демонстрационные данные можно найти в коде, приложенном к этой главе: в файлах `schema.sql` и `data.sql`. Все примеры кода находятся в файле `examples.sql`.

Оперативная аналитическая обработка

Компания, управляющая сайтом торговли автомобилями, могла бы хранить журналы HTTP-доступа в таблице базы данных и использовать ее для анализа действий пользователей, например чтобы измерить производительность приложения, выявить закономерности в поведении пользователей или просто собрать статистику о том, какие модели вызывают наибольший интерес. Эти данные можно было бы вставлять в таблицу, а потом не модифицировать и не удалять, кроме разве что очень старых данных. Объем, правда, получается гораздо больше, чем имеется предметных данных во всей базе, но к этим данным обращаются сравнительно редко и только сотрудники компании – для анализа и формирования отчетов.

Никто не ожидает, что эти пользователи будут выполнять много запросов в секунду, скорее наоборот, но зато эти запросы будут большими и сложными, поэтому время выполнения имеет значение.

Еще одна особенность баз данных для OLAP – тот факт, что они не всегда актуальны. Поскольку анализ производится раз в неделю или в месяц (например, сравнение количества запросов в текущем и в предыдущем месяце), не имеет смысла лезть из кожи вон, чтобы данные в хранилище отражали картину на текущий момент времени. Данные можно загружать в хранилище периодически, например один или несколько раз в день.

Извлечение, преобразование и загрузка

Рассмотрим задачу о загрузке журналов HTTP-доступа в базу данных и подготовки их для анализа. Подобные задачи называются **извлечением, преобразованием и загрузкой** (extract, transform, load – ETL).

Предположим, что HTTP-сервер, на котором работает сайт торговли автомобилями, пишет журналы доступа в файлы, которые пересоздаются ежедневно. Если в качестве сервера используется популярная программа `nginx`, то строка журнала по умолчанию выглядит следующим образом:

```
94.31.90.168 - - [01/Jul/2017:01:00:22 +0000] "GET / HTTP/1.1" 200 227 "-"
"Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/55.0.2883.87 Safari/537.36"
```

В ней выделяются следующие поля, разделенные пробелами: удаленный адрес, удаленный пользователь, временная метка, метод доступа и URL ресурса, код состояния, размер тела, количество байтов, URL источника запроса (HTTP referrer) и пользовательский агент.

Процесс настройки HTTP-сервера выходит за рамки этой книги. Но отметим, что загрузка в базу данных PostgreSQL станет проще, если изменить формат журнала. Чтобы в качестве разделителей использовались точки с запятой, нужно переопределить директиву `log_format` в конфигурационном файле `nginx.conf`:

```
log_format main '$time_local;$remote_addr;$remote_user;'
                '"$request";$status;$body_bytes_sent;'
                '"$http_referer";"$http_user_agent"'
```

Теперь журнал порождается в формате CSV:

```
01/Jul/2017:01:00:22 +0000;94.31.90.168;-;"GET / HTTP/1.1";200;227;"-
";"Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/61.0.3163.79 Safari/537.36"
```

Состав полей не изменился, только временная метка переместилась на первое место. Поля разделены точкой с запятой. В базе данных имеется таблица `dwh.access_log` с такой структурой:

```
CREATE TABLE dwh.access_log
(
  ts timestamp with time zone,
  remote_address text,
  remote_user text,
  url text,
  status_code int,
  body_size int,
  http_referer text,
  http_user_agent text
);
```

PostgreSQL позволяет быстро загрузить в таблицу данные – не построчно, как в команде INSERT, а сразу много строк. Это делается командой COPY. Данные можно брать из файла, физически находящегося на сервере базы данных в любом месте, доступном процессу PostgreSQL, или из потока. По умолчанию данные должны быть представлены в виде списка полей, разделенных знаками табуляции, а записи должны быть разделены знаками новой строки. Но всё это настраивается. Команда COPY не включена в стандарт SQL. Ниже представлена упрощенная синтаксическая диаграмма команды:

```
COPY <table name> [(column [, ...])] FROM { <file name> | STDIN } [[WITH]
(<options>)]
```

Здесь options задает формат данных: разделители, виды кавычек, символы экранирования и некоторые другие параметры. Если вместо имени файла указано STDIN, данные читаются не из файла, а из стандартного ввода. Если в файле присутствуют данные не для всех столбцов таблицы или они расположены в другом порядке, то можно указать список столбцов.

Чтобы загрузить данные в базу, необходимо как-то скопировать файл на сервер, а затем выполнить команду:

```
COPY dwh.access_log FROM '/tmp/access.log';
```

В результате в таблицу dwh.access_log будет загружено содержимое файла /tmp/access.log. Однако это не слишком удачная мысль по нескольким причинам:

- только суперпользователю разрешено читать данные из файла командой COPY;
- поскольку обращаться к файлу будет сервер PostgreSQL, файл необходимо поместить туда, где хранятся прочие файлы базы данных, либо дать процессу postgres доступ к другим каталогам. То и другое плохо с точки зрения безопасности;
- копирование файла на сервер само по себе может оказаться проблемой, поскольку необходимо разрешить какому-то пользователю (или приложению) доступ к файловой системе сервера либо смонтировать общие папки, доступные серверу.

Чтобы избежать всех этих проблем, следует использовать команду COPY для загрузки данных из потока, а точнее из стандартного ввода. Для этого понадобится приложение, которое подключается к базе данных, запускает команду COPY, а затем передает на сервер поток данных. По счастью, всё это умеет делать psql.

Попробуем загрузить демонстрационный файл access.log, имеющийся в прилагаемом исходном коде.

Допустим, что файл находится в текущем каталоге. База данных работает на локальной машине, а пользователю car_portal_app разрешено к ней подключаться. Следующая команда, запущенная в системе Linux, загружает данные из файла в таблицу dwh.access_log:

```
user@host:~$ cat access.log | psql -h localhost -U car_portal_app -d
car_portal -c "COPY dwh.access_log FROM STDIN WITH (format csv, delimiter ';')"
COPY 15456
```

Команда `cat` печатает содержимое файла в стандартный вывод. Он перенаправляется в команду `psql`, которая выполняет команду `COPY`, загружающую данные из стандартного ввода. После завершения печатается слово `COPY` и количество скопированных строк.

В Windows то же самое делается так:

```
c:\Users\user> psql -h localhost -U car_portal_app -d car_portal -c "COPY
dwh.access_log FROM STDIN WITH (format csv, delimiter ';')" < access.log
COPY 15456
```

Эту операцию можно также выполнить интерактивно из оболочки `psql`. Для этого в `psql` имеется команда `\copy`, синтаксис которой похож на синтаксис SQL-команды `COPY`: `\copy dwh.access_log FROM 'access.log' WITH csv delimiter ';'`. За кулисами она делает то же самое. На сервере выполняется команда `COPY ... FROM STDIN`, а `psql` читает файл и передает его содержимое серверу.

Применив этот подход, очень легко организовать простую процедуру ETL и запускать ее один раз в день. Надо только настроить ротацию журналов на HTTP-сервере, чтобы одни и те же данные не загружались дважды.

Команда `COPY` может не только загружать данные в таблицу, но и выгружать данные из таблицы в файл или на стандартный вывод. Эту возможность тоже можно использовать в процессах ETL, реализованных в виде простых bash-скриптов, без применения сложного ПО, которое обращается к базе данных через интерфейс JDBC или другие API.

Вот пример копирования таблицы с одного сервера на другой (в предположении, что структура таблицы в обеих базах данных одинакова):

```
$ psql -h server1 -d database1 -c "COPY table1 TO STDOUT"
| psql -h server2 -d database2 -c "COPY table2 FROM
STDIN"
```

Здесь первая команда `psql` читает таблицу и выводит ее на стандартный вывод. Этот поток перенаправляется на стандартный вход второй команды `psql`, которая загружает данные в другую таблицу.

ETL может также включать обогащение исходных данных дополнительными атрибутами или их предварительную обработку с целью упрощения последующих запросов. Например, можно отобразить записи файла на строки базы данных `car_portal`. Предположим, что обращение по URL `/api/cars/30` можно отобразить на строку таблицы `car` с идентификатором `car_id = 30`. Такую предобработку можно произвести в базе данных следующим образом:

```
car_portal=> ALTER TABLE dwh.access_log ADD car_id int;
ALTER TABLE
```

```
car_portal=> UPDATE dwh.access_log
  SET car_id = (SELECT regexp_matches(url, '/api/cars/(\d+)\W'))[1]::int
  WHERE url like '%/api/cars/%';
UPDATE 2297
```

Аналогично в таблицу access_log можно добавить атрибуты для соединения с другими таблицами.

На практике процесс ETL обычно устроен сложнее. Как правило, проверяется, существуют ли уже данные в конечной таблице, и производится их очистка перед загрузкой. Таким образом, процесс оказывается идемпотентным. Кроме того, процесс может решить, какие данные загружать, и найти входные данные. Рассмотрим, к примеру, задание ETL, которое каждый день загружает один и тот же набор данных. Если по какой-то причине случился сбой, то на следующий день процесс должен понять, что произошло, и загрузить два набора данных вместо одного. Оповещение нужных систем или людей о сбое также может быть возложено на процесс ETL.

На рынке имеется много различных инструментов ETL, в том числе с открытым исходным кодом.

Моделирование данных для OLAP

Большая таблица в хранилище данных, содержащая субъект анализа, обычно называется **таблицей фактов**. Журнал доступа по HTTP, который мы обсуждали в предыдущем разделе, как раз и играет роль таблицы фактов.

Не имеет смысла выполнять какие-либо аналитические запросы к таблице фактов, включающие группировку по car_id, не понимая, что стоит за значениями поля car_id. Следовательно, таблица car тоже должна присутствовать в хранилище. Разница между этой таблицей и таблицей фактов состоит в том, что в таблицу фактов данные постоянно загружаются, а таблица car в основном статична. Количество записей в таблице car во много раз меньше. Такие таблицы, служащие для преобразования идентификаторов в имена, называются **справочными таблицами**, или **таблицами измерений**. В таблице car имеется внешний ключ car_model_id, указывающий на запись в таблице car_model, именно он и используется для преобразования идентификатора в модель и марку автомобиля.

Обычно таблица фактов используется так: большие запросы SELECT выполняются не слишком часто, но читают огромное количество записей, исчисляемое миллионами или десятками миллионов. В таком случае любые дополнительные операции сервера обходятся очень дорого. Это касается и соединения таблицы фактов с таблицами измерений.

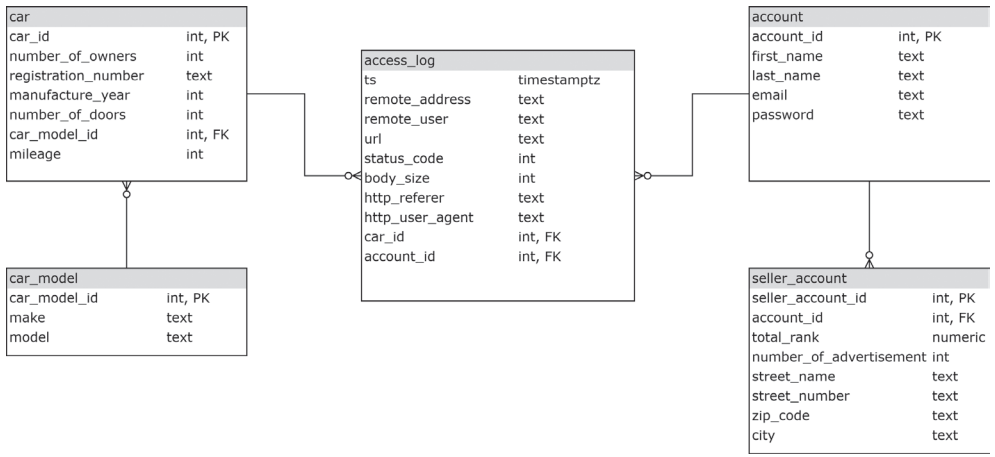
Поэтому данные нередко подвергают денормализации. На практике это означает, что соединение таблиц производится заранее и результат сохраня-

ется в новой таблице. Например, если бы записи журнала доступа были отображены на `car_id`, как в предыдущем разделе, а аналитическая задача состояла бы в вычислении статистических показателей о марках машин, то нам пришлось бы выполнить два соединения: `access_log` с `car` и `car` с `car_model`. Это дорого.

Для экономии времени имеет смысл соединить таблицы `car` и `car_model` и сохранить результат в отдельной таблице. Она уже не будет иметь нормальную форму, потому что одна и та же модель встречается в ней много раз. Конечно, это означает дополнительный расход места на диске. Но это разумный компромисс, потому что опрос таблицы фактов станет быстрее, если производить соединение с одной этой таблицей, а не с двумя таблицами, `car` и `car_model`.

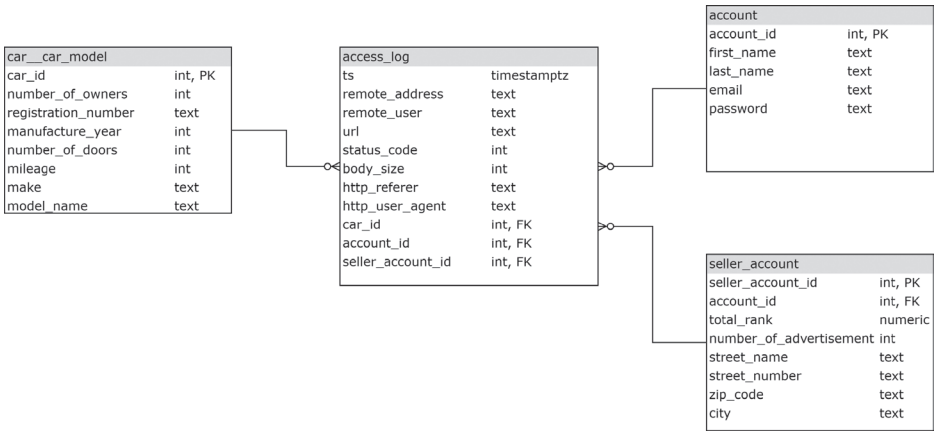
Можно было бы вместо этого завести поле `car_model_id` прямо в таблице фактов и заполнять его в процессе ETL. Такая структура тоже денормализована, она тоже потребляет больше места, но запрос становится проще.

Таблица фактов может ссылаться на таблицы измерений. А таблица измерений может, в свою очередь, ссылаться на другие таблицы измерений. Такая организация в терминологии OLAP называется **схемой типа «снежинка»** и может выглядеть следующим образом:



Модель хранилища данных типа «снежинка»

Если структура денормализована и таблицы измерений не соединяются между собой, то получается схема типа «звезда», вот такая:



Модель хранилища данных типа «звезда»

Плюсы и минусы есть у обоих подходов, и, разумеется, ничто не мешает использовать гибридные схемы, в которых «звезда» сочетается со «снежинкой». Обычно это вопрос компромисса между сложностью, производительностью и удобством сопровождения.

Агрегирование

Хранение отдельных записей в хранилище данных может оказаться слишком дорогим удовольствием. Количество HTTP-запросов в популярном веб-приложении может быть очень велико, а отдельные записи в таком наборе никому не интересны. Отдельные записи могут пригодиться для поиска неполадок в системе, но тогда хранить их стоит всего несколько дней, не дольше.

В таблице, содержащей информацию о вызовах HTTP API, могут быть следующие поля: количество отправленных байтов, время ответа, URL оконечной точки API, дата и время запроса, идентификатор пользователя. В этом случае количество отправленных байтов и время ответа – это показатели, а URL оконечной точки API, код состояния HTTP, дата и время запроса и идентификатор пользователя – измерения.

Данные группируются по полям измерений, а поля показателей агрегируются. Для количества отправленных байтов имеет смысл использовать агрегатную функцию `sum`, а для времени ответа – `avg`. Дополнительно можно подсчитать количество запросов. Для целей группировки от времени запроса можно оставить только часы. Результат такого агрегирования сохраняется в хранилище. Объем данных в агрегированной таблице гораздо меньше, чем занимали бы отдельные записи, поэтому работать она будет быстрее. Агрегирование может быть частью процесса ETL.

Если объем данных очень велик, то можно принести детальность в жертву производительности. Это означает, что какое-то измерение отбрасывается,

чтобы уменьшить количество комбинаций значений при группировке. Например, можно группировать по дням, а не по часам, или удалить идентификатор пользователя из таблицы, сократив тем самым количество строк, потому что все пользователи будут объединены в одну группу. Иногда разумно хранить таблицы с разными уровнями агрегирования и выбирать наиболее подходящую для конкретного отчета.

Секционирование

Данные загружаются в хранилище постоянно или периодически. База данных может стать очень большой. Чем она больше, тем медленнее работает. Размер базы ограничен емкостью дисковой системы хранения, поэтому время от времени данные нужно удалять. Но удаление из очень большой таблицы производится медленно.

Более свежие данные обычно запрашиваются чаще. Сотрудники компании могли бы каждое утро просматривать отчеты за прошлый день, каждый понедельник – отчеты за прошлую неделю и в начале месяца – отчеты за прошлый месяц. Принято сравнивать результаты периода с результатами соответствующего прошлого периода: текущий месяц с предыдущим или с таким же месяцем прошлого года. Маловероятно, что кому-то понадобятся данные десятилетней давности.

Было бы хорошо хранить свежие, часто запрашиваемые данные в одной сравнительно небольшой таблице, а старые – в другой таблице или в нескольких таблицах, и опрашивать только ту таблицу, в которой находятся данные за интересующий период. С другой стороны, генератор отчетов, который опрашивает разные таблицы в зависимости от параметров запроса, оказался бы слишком сложным.

PostgreSQL предлагает способ хранить данные в разных таблицах, но использовать общее имя в запросах к ним. Это называется **секционированием** и реализуется с помощью механизма наследования таблиц, который уже упоминался в главе 3. Таблица, наследующая одну или несколько других таблиц, называется **дочерней**, а таблица, которую она наследует, – **родительской**. При запросе к родительской таблице возвращаются данные из всех дочерних таблиц. В контексте секционирования дочерние таблицы называются **секциями**.

Для определения секций таблицы необходимо выбрать **секционный ключ**. Это поле или выражение (или список таковых), значение которого определяет, какой секции принадлежит запись. В PostgreSQL поддерживается две схемы секционирования: *по диапазонам* и *по списку*. Секционирование по диапазонам означает, что секции принадлежат все значения из некоторого диапазона.

Давайте определим секционированную структуру для хранения данных из журналов доступа по HTTP, которые мы импортировали в предыдущем разделе. Для простоты будем использовать только подмножество полей.

Сначала создадим родительскую таблицу:

```
CREATE TABLE dwh.access_log_partitioned (ts timestamptz, url text, status_code int)
PARTITION BY RANGE (ts);
```

Здесь выбрана схема секционирования по диапазонам, а поле ts является секционным ключом.

Теперь создадим секции для трех диапазонов значений поля ts: июль, август, сентябрь 2017:

```
CREATE TABLE dwh.access_log_2017_07 PARTITION OF dwh.access_log_partitioned
FOR VALUES FROM ('2017-07-01') TO ('2017-08-01');

CREATE TABLE dwh.access_log_2017_08 PARTITION OF dwh.access_log_partitioned
FOR VALUES FROM ('2017-08-01') TO ('2017-09-01');

CREATE TABLE dwh.access_log_2017_09 PARTITION OF dwh.access_log_partitioned
FOR VALUES FROM ('2017-09-01') TO ('2017-10-01');
```

Нижняя граница диапазона секции включается, верхняя – не включается. В примере выше секция access_log_2017_07 будет содержать записи, удовлетворяющие условию ts >= '2017-07-01' AND ts < '2017-08-01'.

Определив такую структуру, мы можем вставлять данные в родительскую таблицу access_log_partitioned, и они будут автоматически попадать в нужные секции в зависимости от значения секционного ключа.

Что произойдет, если кто-то попробует вставить запись, в которой значение секционного ключа не попадает ни в один из существующих диапазонов? Тогда PostgreSQL сообщит об ошибке:

```
car_portal=> INSERT INTO dwh.access_log_partitioned values ('2017-02-01',
'/test', 404);
  ERROR: no partition of relation "access_log_partitioned" found for row
  DETAIL: Partition key of the failing row contains (ts) = (2017-02-01
00:00:00+00).
```

Чтобы такую строку все-таки можно было вставить, необходимо создать еще одну секцию для февраля. Или же создать секцию для всех строк, в которых ключ больше или меньше некоторого значения. Для этой цели зарезервированы ключевые слова MAXVALUE и MINVALUE. Следующая команда создает секцию для всех записей до 1 июля 2017 г.:

```
CREATE TABLE dwh.access_log_min PARTITION OF dwh.access_log_partitioned
FOR VALUES FROM (MINVALUE) TO ('2017-07-01');
```

Секционирование по списку можно использовать для полей, принимающих не слишком большое количество значений, и все они заранее известны. Например, для кода состояния HTTP можно следующим образом объединить обе схемы секционирования, создав подсекции для существующих секций:

```
CREATE TABLE dwh.access_log_2017_10 PARTITION OF dwh.access_log_partitioned
FOR VALUES FROM ('2017-10-01') TO ('2017-11-01')
```

```
PARTITION BY LIST (status_code);

CREATE TABLE dwh.access_log_2017_10_200 PARTITION OF dwh.access_log_2017_10
FOR VALUES IN (200);

CREATE TABLE dwh.access_log_2017_10_400 PARTITION OF dwh.access_log_2017_10
FOR VALUES IN (400);
```

Теперь у нас имеется секция для октября с двумя подсекциями для кодов состояния 200 и 400. Отметим, что при такой конфигурации невозможно будет вставить в таблицу другие коды состояния. Однако при создании секции можно указать не одно значение секционного ключа, а целый список.

Схему секционирования можно определять не по полю, а по выражению. В следующем примере секции создаются по первой цифре кода состояния:

```
CREATE TABLE dwh.access_log_2017_11 PARTITION OF dwh.access_log_partitioned
FOR VALUES FROM ('2017-11-01') TO ('2017-12-01')
PARTITION BY LIST (left(status_code::text, 1));

CREATE TABLE dwh.access_log_2017_11_2XX PARTITION OF dwh.access_log_2017_11
FOR VALUES IN ('2');

CREATE TABLE dwh.access_log_2017_11_4XX PARTITION OF dwh.access_log_2017_11
FOR VALUES IN ('4');
```

С помощью традиционного наследования таблиц возможно реализовать и более сложные стратегии секционирования. Правда, потребуется поработать вручную: создать таблицы, определить отношения наследования, создать проверочные ограничения, чтобы сервер PostgreSQL знал, какие данные в какой секции находятся, и настроить триггеры или правила для распределения данных по секциям при вставке в родительскую таблицу.

Чтобы удалить секцию, достаточно просто удалить таблицу. Можно также отсоединить секцию от родительской таблицы, чтобы она стала обыкновенной таблицей:

```
car_portal=> ALTER TABLE dwh.access_log_partitioned
DETACH PARTITION dwh.access_log_2017_11;
ALTER TABLE
```

Наконец, можно превратить существующую таблицу в секцию какой-то другой таблицы:

```
car_portal=> ALTER TABLE dwh.access_log_partitioned
ATTACH PARTITION dwh.access_log_2017_11
FOR VALUES FROM ('2017-11-01') TO ('2017-12-01');
ALTER TABLE
```

Для иллюстрации преимуществ секционирования рассмотрим пример. Сначала для сравнения результатов создадим несекционированную таблицу с такой же структурой, как у секционированной:

```
car_portal=> CREATE TABLE dwh.access_log_not_partitioned (LIKE
dwh.access_log_partitioned);
CREATE TABLE
```

Вставим данные в секционированную и несекционированную таблицу, продублировав содержимое таблицы dwh.access_log 1000 раз:

```
car_portal=> INSERT INTO dwh.access_log_not_partitioned SELECT ts, url,
status_code FROM dwh.access_log, generate_series(1, 1000);
INSERT 0 15456000

car_portal=> INSERT INTO dwh.access_log_partitioned SELECT ts, url,
status_code FROM dwh.access_log, generate_series(1, 1000);
INSERT 0 15456000
```

Теперь посчитаем количество записей за последние 10 дней августа и измерим время выполнения запроса:

```
car_portal=> \timing
Timing is on.

car_portal=> SELECT count(*) FROM dwh.access_log_not_partitioned
WHERE ts >= '2017-08-22' AND ts < '2017-09-01';
 count
---------
 1712000
(1 row)
Time: 921.122 ms

car_portal=> SELECT count(*) FROM dwh.access_log_partitioned
WHERE ts >= '2017-08-22' AND ts < '2017-09-01';
 count
---------
 1712000
(1 row)
Time: 336.221 ms
```

Запрос к секционированной таблице оказался примерно в три раза быстрее. Это объясняется тем, что PostgreSQL знает, в какой секции находятся данные за август, и просматривает только ее. Это видно и из плана выполнения:

```
car_portal=> EXPLAIN SELECT count(*) FROM dwh.access_log_partitioned
  WHERE ts >= '2017-08-22' AND ts < '2017-09-01';
QUERY PLAN
------------------------------------------------------------------------------
------------------------------------------------------------------------------
--------
 Finalize Aggregate (cost=82867.96..82867.97 rows=1 width=8)
  -> Gather (cost=82867.75..82867.96 rows=2 width=8)
   Workers Planned: 2
    -> Partial Aggregate (cost=81867.75..81867.76 rows=1 width=8)
     -> Append (cost=0.00..80149.89 rows=687141 width=0)
      -> Parallel Seq Scan on access_log_2017_08 (cost=0.00..80149.89
rows=687141 width=0)
         Filter: ((ts >= '2017-08-22 00:00:00+00'::timestamp with time
zone) AND (ts < '2017-09-01 00:00:00+00'::timestamp with time zone))
(7 rows)
```

Последовательно просматривается секционная таблица access_log_2017_08.

Секционирование подробно описано на странице документации https://www.postgresql.org/docs/current/static/ddl-partitioning. html.

Параллельные запросы

PostgreSQL создает серверный процесс для каждого клиентского подключения. Это означает, что всю работу будет выполнять только одно процессорное ядро. Конечно, когда активно несколько подключений, ресурсы сервера будут использоваться интенсивно. Но в случае хранилищ данных количество одновременных сеансов обычно не очень велико, зато запросы большие и сложные. Поэтому имеет смысл задействовать несколько ядер для обработки запросов от одного клиента.

PostgreSQL поддерживает **параллельные запросы**, т. е. позволяет отдавать несколько процессоров под один запрос. Некоторые операции, в т. ч. просмотр таблицы, соединение и агрегирование, можно одновременно выполнять в нескольких процессах. Количество исполнителей, создаваемых для параллельного выполнения запроса, задает администратор. Если оптимизатор видит, что параллельное выполнение может принести выгоду, то он запрашивает дополнительных исполнителей, и если их удается получить (хотя бы частично), то запрос будет выполняться параллельно.

Чтобы оценить преимущества параллельного выполнения запросов, рассмотрим пример таблицы, в которую загружен журнал доступа по HTTP. Сначала отключим параллельные запросы в текущем сеансе, установив конфигурационный параметр `max_parallel_workers_per_gather` в 0. Отметим, что речь идет не об общем числе процессов, исполняющих запросы, а о количестве *дополнительных* исполнителей. Главный серверный процесс, обслуживающий текущий клиентский сеанс, всегда присутствует. Чтобы изменить параметр, выполните команду:

```
car_portal=> SET max_parallel_workers_per_gather = 0;
SET
```

Включим хронометраж и опросим таблицу:

```
car_portal=> SELECT count(*) FROM dwh.access_log_not_partitioned WHERE url ~ 'car';
  count
---------
 7030000
(1 row)
Time: 10876.095 ms (00:10.876)
```

Снова включим параллельные запросы, установив параметр `max_parallel_workers_per_gather` в 1. Это означает, что запрос будет исполняться двумя параллельными процессами. Повторим запрос:

```
car_portal=> SET max_parallel_workers_per_gather = 1;
SET
car_portal=> SELECT count(*) FROM dwh.access_log_not_partitioned WHERE url ~ 'car';
```

```
 count
---------
 7030000
(1 row)
Time: 6435.174 ms (00:06.435)
```

Запрос выполняется почти в два раза быстрее!

Показанный выше запрос интенсивно нагружает процессоры, поскольку во фразе `WHERE` используется регулярное выражение. Если бы узким местом был жесткий диск, то эффект задействования нескольких процессоров был бы не так ярко выражен.

Дополнительные сведения о параллельных запросах см. на странице https://www.postgresql.org/docs/current/static/parallel-query.html.

Просмотр только индексов

Мы уже обсуждали индексы в главе 4. Проще говоря, индекс ведет себя как алфавитный указатель в конце книги. Если мы хотим найти в книге какой-то термин, то будет быстрее заглянуть в указатель, а затем перейти на указанную в нем страницу. Поскольку указатель отсортирован по алфавиту, поиск в нем производится быстро. Но если нам нужно знать только, встречается ли термин в книге, то переходить на страницу вообще не нужно – достаточно просмотреть указатель.

PostgreSQL умеет делать то же самое. Если вся информация, необходимая для выполнения запроса, содержится в индексе, то сервер не станет просматривать данные в таблице, а ограничится индексом. Это называется **просмотром только индекса** (index-only scan).

Для демонстрации создадим индекс над таблицей `dwh.access_log_not_partitioned`:

```
CREATE INDEX on dwh.access_log_not_partitioned (ts, status_code);
```

Пусть требуется найти первый HTTP-запрос с кодом состояния 201, имевший место 1 августа. Запрос выглядит так:

```
car_portal=> SELECT min(ts) FROM dwh.access_log_not_partitioned
WHERE ts BETWEEN '2017-08-01' AND '2017-08-02' AND status_code = '201';
 min
-----------------------
 2017-08-01 01:30:57+00
(1 row)
Time: 0.751 ms
```

Запрос занял менее одной миллисекунды. Из плана выполнения видно, что сервер не стал просматривать всю таблицу, а выполнил просмотр только индекса:

```
car_portal=> EXPLAIN SELECT min(ts) FROM dwh.access_log_not_partitioned
  WHERE ts BETWEEN '2017-08-01' AND '2017-08-02' AND status_code = '201';
```

```
 QUERY PLAN
------------------------------------------------------------------------------
------------------------------------------------------------------------------
---------------------------------------------
 Result (cost=4.23..4.24 rows=1 width=8)
  InitPlan 1 (returns $0)
   -> Limit (cost=0.56..4.23 rows=1 width=8)
    -> Index Only Scan using
access_log_not_partitioned_ts_status_code_idx on
access_log_not_partitioned (cost=0.56..135923.57 rows=37083
width=8)
        Index Cond: ((ts IS NOT NULL) AND (ts >= '2017-08-01
00:00:00+00'::timestamp with time zone)
AND (ts <= '2017-08-02 00:00:00+00'::timestamp with time zone)
AND (status_code = 201))
(5 rows)
```

Чтобы оценить выгоду от использования этого механизма, выключим его и снова проведем хронометраж:

```
car_portal=> SET enable_indexonlyscan = off;
SET

car_portal=> SELECT min(ts) FROM dwh.access_log_not_partitioned
  WHERE ts BETWEEN '2017-08-01' AND '2017-08-02' AND status_code = '201';
min
2017-08-01 01:30:57+00
(1 row)
Time: 1.225 ms
```

Теперь запрос выполняется почти в два раза медленнее. И план выполнения несколько изменился (хотя общая стоимость осталась прежней):

```
car_portal=> EXPLAIN SELECT min(ts) FROM dwh.access_log_not_partitioned
  WHERE ts BETWEEN '2017-08-01' AND '2017-08-02' AND status_code = '201';
QUERY PLAN
------------------------------------------------------------------------------
------------------------------------------------------------------------------
---------------------------------------------
 Result (cost=4.23..4.24 rows=1 width=8)
  InitPlan 1 (returns $0)
   -> Limit (cost=0.56..4.23 rows=1 width=8)
    -> Index Scan using access_log_not_partitioned_ts_status_code_idx on
access_log_not_partitioned (cost=0.56..135923.57 rows=37083 width=8)
         Index Cond: ((ts IS NOT NULL) AND (ts >= '2017-08-01
00:00:00+00'::timestamp with time zone)
AND (ts <= '2017-08-02 00:00:00+00'::timestamp with time zone)
AND (status_code = 201))
(5 rows)
```

Этот механизм может работать, только если все поля, упоминаемые в запросе, являются частью индекса. Иногда оптимизатор может решить, что последовательный просмотр таблицы будет дешевле, даже если индекс содержит

все данные. Обычно так бывает, когда предполагается, что будет возвращено много записей.

На просмотр только индекса налагаются некоторые ограничения, связанные с тем, как PostgreSQL поддерживает изоляцию транзакций. Детали см. в главе 10. В двух словах, PostgreSQL проверяет карты видимости таблицы, чтобы понять, должны ли данные быть видны текущей транзакции. Если в картах видимости этой информации нет, обращаться к данным в самой таблице обязательно.

Не все индексы поддерживают механизм просмотра. Индексы типа B-tree поддерживают его всегда, типа GiST и SP-GiST – только для некоторых операторов, а типа GIN не поддерживают вовсе.

Эта тема относится не только к OLAP-решениям. Однако над таблицами в хранилище данных часто создается несколько индексов по разным комбинациям полей, и иногда они перекрываются. Понимать, как работает просмотр только индексов, важно для проектирования оптимальной структуры базы данных.

Дополнительные сведения о просмотре только индексов см. на странице https://www.postgresql.org/docs/current/static/indexes-index-only-scans.html.

Резюме

OLAP и хранилища данных применяются в тех случаях, когда база данных используется для анализа и отчетности. Акроним OLAP означает «оперативная аналитическая обработка». В отличие от OLTP, OLAP предполагает большие объемы данных и сравнительно меньшее число одновременных сеансов и транзакций, но размер транзакции при этом больше. Часто структура базы денормализуется во имя повышения производительности. База данных, являющаяся частью OLAP-решения, часто называется хранилищем, или складом данных.

В этой главе мы рассмотрели, как структурируются данные в хранилище, как в него загружаются данные и как оптимизировать производительность, применяя секционирование, параллельное выполнение запросов и просмотр только индексов.

В следующей главе мы рассмотрим расширенные типы данных, поддерживаемые PostgreSQL: массивы, JSON и другие. Они позволяют реализовать сложную бизнес-логику, пользуясь только встроенными в СУБД средствами.

Глава 9

За пределами традиционных типов данных

Благодаря развитой системе расширений PostgreSQL может работать со сложными типами данных. Данные, не укладывающиеся в строгую реляционную модель, например слабо структурированные, можно тем не менее хранить и обрабатывать, пользуясь либо встроенными, либо дополнительными типами. К тому же сообщество PostgreSQL акцентирует внимание не только на развитии реляционных механизмов, но и на поддержке таких типов данных, как массивы, XML, хранилища ключей и значений и JSON-документы. Смещение акцентов стало результатом учета изменений в жизненном цикле разработки ПО: развития гибких (agile) методов и поддержки быстро изменяющихся требований.

Благодаря нетрадиционным типам данных PostgreSQL может хранить географические, двоичные, бессхемные данные типа JSON-документов и выступать в роли хранилища ключей и значений. Поддержка некоторых типов данных, в т. ч. JSON, JSONB, XML, массивов и BLOB'ов, встроена в PostgreSQL. Но гораздо больше типов доступно с помощью расширений, например `hstore` и PostGIS.

JSON, JSONB и HStore позволяют PostgreSQL работать с бессхемными моделями, что, в свою очередь, дает разработчикам возможность автоматически и прозрачно вносить структурные корректировки в реальном времени, не прибегая к команде `ALTER`. Кроме того, мы получаем шанс обеспечить гибкость, не опираясь на модель **сущность-атрибут-значение** (entity-attribute-value – **EAV**), которая с трудом совмещается с реляционной моделью. Однако разработчики должны уделять внимание обеспечению целостности данных на уровне бизнес-логики, чтобы данные были чистыми и безошибочными.

В этой главе рассматриваются следующие типы данных:

- массив;
- тип данных Hstore, используемый для организации хранилищ ключей и значений;
- JSON;
- типы данных для полнотекстового поиска, включая tsquery и tsvector.

Рекомендуем также познакомиться с расширением PostGIS, поскольку оно поддерживает растровые и векторные форматы и предоставляет разнообразные функции для манипулирования ими. Интересен также тип данных `range`, который можно использовать, например, для определения диапазона дат.

Массивы

Массив состоит из набора элементов (значений или переменных), порядок элементов массива важен, каждый элемент идентифицируется своим индексом. Индексы элементов массива часто начинаются с 1, хотя в некоторых языках программирования, в т. ч. C и C++, индексирование начинается с 0. Все элементы массива должны иметь один и тот же тип, например `INT` или `TEXT`.

Поддерживаются также многомерные массивы. В массивах PostgreSQL допускаются дубликаты и значения null. В примере ниже показано, как инициализировать одномерный массив и получить первый элемент:

```
car_portal=> SELECT ('{red, green, blue}'::text[])[1] as red ;
 red
-----
 red
(1 row)
```

По умолчанию длина массива не задается, но ее можно указать, когда массивы используются для определения отношения. По умолчанию индексирование массива начинается с единицы, как в примере выше, но и это поведение можно изменить, определив размерность на этапе инициализации:

```
car_portal=> WITH test AS (SELECT '[0:1]={1,2}'::INT[] as arr) SELECT arr,
arr[0] from test;
         arr | arr
-------------+-----
 [0:1]={1,2} | 1
(1 row)
```

Для инициализации массива используется конструкция {}. Но есть и другой способ:

```
car_portal=> SELECT array['red','green','blue'] AS primary_colors;
  primary_colors
-----------------
 {red,green,blue}
(1 row)
```

PostgreSQL предоставляет много функций для манипулирования массивами, например `array_remove` для удаления элемента. Ниже продемонстрированы некоторые функции массивов:

```
car_portal=> SELECT
  array_ndims(two_dim_array) AS "Number of dimensions",
  array_dims(two_dim_array) AS "Dimensions index range",
```

```
  array_length(two_dim_array, 1) AS "The array length of 1st dimension",
  cardinality(two_dim_array) AS "Number of elements",
  two_dim_array[1][1] AS "The first element"
FROM
(VALUES ('{{red,green,blue}, {red,green,blue}}'::text[][])) AS
foo(two_dim_array);
-[ RECORD 1 ]----------------------+-----------
Number of dimensions              | 2
Dimensions index range            | [1:2][1:3]
The array length of 1st dimension | 2
Number of elements                | 6
The first element                 | red
```

Распространенное применение массивов – моделирование многозначных атрибутов. Например, у платья может быть несколько цветов, а у газетной статьи несколько категорий. Также они используются для моделирования хранилищ ключей и значений. Для этого заводят два массива: один содержит ключи, другой – значения, а для связывания ключа со значением используется индекс массива. Так, в таблице `pg_stats` этот подход применен для хранения информации о гистограмме типичных значений. В столбцах `most_common_vals` и `most_common_freqs` хранятся, соответственно, типичные значения в некотором столбце и частоты этих значений, как показано в примере ниже:

```
car_portal=> SELECT tablename, attname, most_common_vals, most_common_freqs
FROM pg_stats WHERE array_length(most_common_vals,1) < 10 AND schemaname
NOT IN ('pg_catalog','information_schema') LIMIT 1;
-[ RECORD 1 ]-----+-------------------------------------------------
tablename         | seller_account
attname           | zip_code
most_common_vals  | {10011,37388,94577,95111,99501}
most_common_freqs | {0.0136986,0.0136986,0.0136986,0.0136986,0.0136986}
```

Кроме того, массивы полезны для упрощения кода и реализации некоторых трюков на SQL, например передачи функции переменного числа аргументов с помощью ключевого слова `VARIADIC` или выполнения циклов с помощью функции `generate_series`. Это позволяет решать сложные задачи, не прибегая к языку PL/pgSQL.

Пусть имеется таблица `vehicle`, содержащая атрибуты, общие для всех транспортных средства. И пусть имеется несколько типов транспортных средств: грузовики, легковые автомобили, автобусы и т. д. Мы могли бы смоделировать эту ситуацию, заведя несколько столбцов, ссылающихся на таблицы легковых автомобилей, грузовиков и автобусы. Чтобы понять, как используется функция с переменным числом аргументов, смоделируем наследование от `vehicle`:

```
CREATE OR REPLACE FUNCTION null_count (VARIADIC arr int[]) RETURNS INT AS
$$
  SELECT count(CASE WHEN m IS NOT NULL THEN 1 ELSE NULL END)::int
  FROM unnest($1) m(n)
$$ LANGUAGE SQL
```

Чтобы воспользоваться этой функцией, нужно добавить в таблицу проверочное ограничение:

```
CREATE TABLE public.car (
  car_id SERIAL PRIMARY KEY,
  car_number_of_doors INT DEFAULT 5
);

CREATE TABLE public.bus (
  bus_id SERIAL PRIMARY KEY,
  bus_number_of_passengers INT DEFAULT 50
);

CREATE TABLE public.vehicle (
  vehicle_id SERIAL PRIMARY KEY,
  registration_number TEXT,
  car_id INT REFERENCES car(car_id),
  bus_id INT REFERENCES bus(bus_id),
  CHECK (null_count(car_id, bus_id) = 1)
);

INSERT INTO public.car VALUES (1, 5);
INSERT INTO public.bus VALUES (1, 25);
При вставке в таблицу vehicle необходимо задать либо car_id, либо bus_id:
car_portal=> INSERT INTO public.vehicle VALUES (default, 'a234', null, null);
ERROR: new row for relation "vehicle" violates check constraint
"vehicle_check"
DETAIL: Failing row contains (1, a234, null, null).
Time: 1,482 ms

car_portal=> INSERT INTO public.vehicle VALUES (default, 'a234', 1, 1);
ERROR: new row for relation "vehicle" violates check constraint
"vehicle_check"
DETAIL: Failing row contains (2, a234, 1, 1).
Time: 0,654 ms
car_portal=> INSERT INTO public.vehicle VALUES (default, 'a234', null, 1);
INSERT 0 1
Time: 1,128 ms
car_portal=> INSERT INTO public.vehicle VALUES (default, 'a234', 1, null);
INSERT 0 1
Time: 1,217 ms
```

Отметим, что при вызове функции null_count необходимо добавить к аргументам ключевое слово VARIADIC:

```
car_portal=> SELECT * FROM null_count(VARIADIC ARRAY [null, 1]);
 null_count
------------
          1
(1 row)
```

Проиллюстрируем еще один прием – генерацию подстроки текста; это оказывается полезно, когда нужно найти самый длинный совпадающий префикс. Такая задача очень важна, например, в телекоммуникационной отрасли – для

телефонных компаний и мобильных операторов, где самый длинный префикс определяет сеть. Вот как ее можно решить:

```
CREATE TABLE prefix (
  network TEXT,
  prefix_code TEXT NOT NULL
);

INSERT INTO prefix VALUES ('Palestine Jawwal', 97059), ('Palestine
Jawwal',970599), ('Palestine watania',970597);

CREATE OR REPLACE FUNCTION prefixes(TEXT) RETURNS TEXT[] AS $$
  SELECT ARRAY(SELECT substring($1,1,i) FROM generate_series(1,length($1))
g(i))::TEXT[];
$$ LANGUAGE SQL IMMUTABLE;
```

Функция `prefixes` возвращает массив, содержащий префикс строки. Чтобы протестировать ее, найдем самый длинный префикс строки `97059973456789`:

```
car_portal=> SELECT * FROM prefix WHERE prefix_code = any
(prefixes('97059973456789')) ORDER BY length(prefix_code) DESC limit 1;
 network          | prefix_code
------------------+-------------
 Palestine Jawwal | 970599
(1 row)
```

Функции и операторы массивов

Для массивов, как и для других типов данных, определены операторы. Например, оператор = сравнивает на равенство, а оператор || производит конкатенацию. В предыдущих главах мы также видели оператор &&, который возвращает true, если массивы пересекаются. Наконец, операторы @> и <@ возвращают true, если первый массив содержит второй или содержится во втором соответственно. Функция `unnest` возвращает множество элементов массива. Это полезно, когда требуется произвести над массивами теоретико-множественную операцию: `distinct`, `order by`, `intersect`, `union` и т. д. В следующем примере мы удаляем дубликаты и сортируем массив в порядке возрастания:

```
SELECT array(SELECT DISTINCT unnest (array [1,1,1,2,3,3]) ORDER BY 1);
```

Здесь к результату функции `unnest` применяются сортировка (`ORDER BY`) и удаление дубликатов (`DISTINCT`). Функция `array()` строит массив из множества. Массивы можно также использовать для агрегирования данных. Например, чтобы получить все модели некоторой марки, можно применить функцию `array_agg`:

```
car_portal=> SELECT make, array_agg(model) FROM car_model group by make;
 make          | array_agg
---------------+-----------------------------------------------------
 Volvo         | {S80,S60,S50,XC70,XC90}
 Audi          | {A1,A2,A3,A4,A5,A6,A8}
 UAZ           | {Patriot}
 Citroen       | {C1,C2,C3,C4,"C4 Picasso",C5,C6}
```

Функция ANY аналогична конструкции SQL IN () и служит для проверки принадлежности элемента массиву:

```
car_portal=> SELECT 1 in (1,2,3), 1 = ANY ('{1,2,3}'::INT[]);
 ?column? | ?column?
----------+----------
 t        | t
(1 row)
```

Полный перечень функций и операторов массивов довольно длинный, его можно найти на странице официальной документации https://www.postgresql.org/docs/current/static/functions-array.html. Помимо уже продемонстрированных функций unnest, array_agg, any, array_length, в следующей таблице перечислены наиболее употребительные функции:

Функция	Возвращаемый тип	Описание	Пример	Результат
array_to_string(anyarray, text [, text])	text	Преобразовать массив в текст. Можно указать разделитель, а также символ, подставляемый вместо NULL	array_to_string(ARRAY[1, NULL, 5], ',', 'x')	'1,,x,5'
array_remove(anyarray, anyelement)	anyarray	Удалить все элементы с указанным значением	array_remove(ARRAY[1,2,3], 2)	{1,3}
array_replace(anyarray, anyelement, anyelement)	anyarray	Заменить все элементы с указанным значением	array_replace(ARRAY[1,2,5,4], 5, 3)	{1,2,3,4}

Доступ к элементам массива и их модификация

К элементу массива обращаются по индексу; если элемент с указанным индексом не существует, то возвращается значение NULL:

```
CREATE TABLE color(
    color text []
);
INSERT INTO color(color) VALUES ('{red, green}'::text[]);
INSERT INTO color(color) VALUES ('{red}'::text[]);
```

Чтобы убедиться в том, что действительно возвращается NULL, выполним такой запрос:

```
car_portal=> SELECT color [3] IS NOT DISTINCT FROM null FROM color;
 ?column?
----------
 t
 t
(2 rows)
```

Можно получить срез массива, указав верхнюю и нижнюю границы:

```
car_portal=> SELECT color [1:2] FROM color;
    color
-------------
```

```
 {red,green}
 {red}
(2 rows)
```

Обновить можно массив целиком, срез массива или отдельный элемент. Можно также дописать элементы в конец массива, воспользовавшись оператором конкатенации ||, как в примере ниже:

```
car_portal=> SELECT ARRAY ['red', 'green'] || '{blue}'::text[] AS append;
      append
-----------------
 {red,green,blue}
(1 row)
Time: 0,848 ms

car_portal=> UPDATE color SET color[1:2] = '{black, white}';
UPDATE 2
car_portal=> table color ;
     color
--------------
 {black,white}
 {black,white}
(2 rows)
```

Для удаления всех элементов с указанным значением предназначена функция `array_remove`:

```
car_portal=> SELECT array_remove ('{Hello, Hello, World}'::TEXT[], 'Hello');
 array_remove
--------------
 {World}
(1 row)
```

Чтобы удалить элемент по индексу, можно воспользоваться фразой `WITH ORDINALITY`. Так, чтобы удалить первый элемент массива, следует написать:

```
car_portal=> SELECT ARRAY(
  SELECT unnest FROM unnest ('{Hello1, Hello2, World}'::TEXT[])
  WITH ordinality WHERE ordinality <> 1);
     array
----------------
 {Hello2,World}
(1 row)
```

Индексирование массивов

Для индексирования массивов используется GIN-индекс; в стандартном дистрибутиве PostgreSQL имеется класс операторов GIN для одномерных массивов. GIN-индекс поддерживают следующие операторы:

- оператор **содержит** @>;
- оператор **содержится в** <@;
- оператор **пересекается** &&;
- оператор **сравнения на равенство** =.

GIN-индекс по столбцу color создается следующим образом:

```
CREATE INDEX ON color USING GIN (color);
```

Проверим, что столбец проиндексирован:

```
car_portal=> SET enable_seqscan TO off; -- Чтобы заставить просматривать индекс
SET
car_portal=> EXPLAIN SELECT * FROM color WHERE '{red}'::text[] && color;
QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on color (cost=8.00..12.01 rows=1 width=32)
  Recheck Cond: ('{red}'::text[] && color)
  -> Bitmap Index Scan on color_color_idx (cost=0.00..8.00 rows=1 width=0)
        Index Cond: ('{red}'::text[] && color)
(4 rows)
```

Хранилище ключей и значений

Хранилище ключей и значений, или ассоциативный массив, – очень популярная структура данных в современных языках программирования, в частности **Java**, **Python** и **Node.js**. Имеются также базы данных, специально «заточенные» под эту структуру данных, например **Redis**.

PostgreSQL поддерживает хранилище ключей и значений – hstore – начиная с версии 9.0. Расширение hstore дает в руки разработчику лучшее из обоих миров, оно предлагает дополнительную гибкость, не жертвуя функциональностью PostgreSQL. Кроме того, hstore позволяет моделировать слабо структурированные данные и разреженные массивы, оставаясь в рамках реляционной модели.

Для создания расширения hstore нужно выполнить следующую команду от имени суперпользователя:

```
CREATE EXTENSION hstore;
```

Текстовое представление hstore содержит нуль или более пар ключ => значение, разделенных запятой, например:

```
SELECT 'tires=>"winter tires", seat=>leather'::hstore;
                  hstore
-------------------------------------------
 "seat"=>"leather", "tires"=>"winter tires"
(1 row)
```

Можно также сгенерировать одиночное значение в hstore с помощью функции hstore(key, value):

```
SELECT hstore(''Hello', 'World');
       hstore
------------------
 "Hello"=>"World"
(1 row)
```

Отметим, что все ключи в хранилище hstore уникальны, как доказывает следующий пример:

```
SELECT 'a=>1, a=>2'::hstore;
  hstore
----------
 "a"=>"1"
(1 row)
```

Пусть требуется поддержать несколько дополнительных атрибутов автомобиля: количество подушек безопасности, наличие кондиционера, усилителя руля и т. д. В традиционной реляционной модели пришлось бы изменить структуру таблицы, добавив новые столбцы. Но hstore позволяет сохранить эту информацию в хранилище ключей и значений, не изменяя каждый раз структуру таблицы:

```
car_portal=# ALTER TABLE car_portal_app.car ADD COLUMN features hstore;
ALTER TABLE
```

У hstore есть ограничение: это не настоящее хранилище документов, поэтому представить в нем вложенные объекты трудно. Вторая проблема – управление множеством ключей, поскольку ключи в hstore чувствительны к регистру.

```
car_portal=# SELECT 'color=>red, Color=>blue'::hstore;
              hstore
--------------------------------
"Color"=>"blue", "color"=>"red"
(1 row)
```

Чтобы получить значение ключа, используется оператор ->. Для добавления в hstore служит оператор конкатенации ||, а для удаления пары ключ-значение – оператор -. Для обновления хранилища hstore следует конкатенировать его с другим экземпляром hstore, который содержит новое значение. Ниже показано, как вставлять, обновлять и удалять ключи hstore:

```
CREATE TABLE features (
  features hstore
);
```

```
car_portal=# INSERT INTO features (features) VALUES
('Engine=>Diesel'::hstore) RETURNING *;
      features
--------------------
 "Engine"=>"Diesel"
(1 row)
```

```
INSERT 0 1
```

```
car_portal=# -- Добавить новый ключ
car_portal=# UPDATE features SET features = features || hstore ('Seat',
'Lether') RETURNING *;
               features
-------------------------------------
```

```
 "Seat"=>"Lether", "Engine"=>"Diesel"
(1 row)

UPDATE 1

car_portal=# -- Обновление ключа аналогично вставке
car_portal=# UPDATE features SET features = features || hstore ('Engine',
'Petrol') RETURNING *;
               features
--------------------------------------
 "Seat"=>"Lether", "Engine"=>"Petrol"
(1 row)

UPDATE 1

car_portal=# -- Удалить ключ
car_portal=# UPDATE features SET features = features - 'Seat'::TEXT
RETURNING *;
      features
--------------------
 "Engine"=>"Petrol"
(1 row)
```

Для типа данных hstore существует немало функций и операторов. Имеется несколько операторов для проверки содержимого hstore. Например, операторы ?, ?& и ?| проверяют, что hstore содержит ключ, множество ключей или любой из перечисленных ключей. Кроме того, объект hstore можно привести к типу массива, множества или JSON-документа.

Тип данных hstore преобразуется во множество с помощью функции each (hstore), и это открывает перед разработчиком возможность применять к hstore все операции реляционной алгебры, например DISTINCT, GROUP BY и ORDER BY.

Ниже показано, как получить уникальные ключи hstore, это может пригодиться для контроля ключей:

```
car_portal=# SELECT DISTINCT (each(features)).key FROM features;
  key
--------
 Engine
(1 row)
```

Чтобы получить hstore в виде множества, снова воспользуемся функцией each:

```
car_portal=# SELECT (each(features)).* FROM features;
key | value
--------+--------
Engine | Petrol
(1 row)
```

Индексирование hstore

Для индексирования данных типа hstore можно воспользоваться индексами GIN и GIST, а какой именно выбрать, зависит от ряда факторов: количества строк в таблице, наличного места, требуемой производительности поиска и об-

новления по индексу, характера запросов и т. д. Прежде чем остановиться на том или ином индексе, нужно провести тщательное эталонное тестирование.

Сначала создадим GIN-индекс для выборки записи по ключу:

```
CREATE INDEX ON features USING GIN (features);
```

Для проверки того, что индекс используется, выполним следующий код, в котором оператор ? проверяет, содержит ли hstore заданный ключ:

```
SET enable_seqscan to off;
car_portal=# EXPLAIN SELECT features->'Engine' FROM features WHERE features
? 'Engine';
QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on features (cost=8.00..12.02 rows=1 width=32)
  Recheck Cond: (features ? 'Engine'::text)
  -> Bitmap Index Scan on features_features_idx (cost=0.00..8.00 rows=1 width=0)
        Index Cond: (features ? 'Engine'::text)
(4 rows)
```

Разумеется, если некоторый оператор не поддерживается GIN-индексом, как, например, оператор ->, можно использовать и индекс типа B-tree:

```
CREATE INDEX ON features ((features->'Engine'));
car_portal=# EXPLAIN SELECT features->'Engine' FROM features
WHERE features->'Engine'= 'Diesel';
QUERY PLAN
------------------------------------------------------------------------------
 Index Scan using features_expr_idx on features (cost=0.12..8.14 rows=1 width=32)
   Index Cond: ((features -> 'Engine'::text) = 'Diesel'::text)
(2 rows)
```

Структура данных JSON

JSON – универсальная структура данных, понятная как человеку, так и машине. Она поддерживается почти всеми современными языками программирования и повсеместно применяется в качестве формата обмена данными в REST-совместимых веб-службах.

JSON и XML

И XML, и JSON используются для определения структуры данных передаваемых документов. Грамматически JSON проще, чем XML, вследствие чего JSON-документы компактнее и проще для чтения и написания. С другой стороны, структуру XML-документа можно определить на **языке определения схем XML** (**XSD**), а затем проконтролировать. У форматов JSON и XML различные применения. Наш опыт показывает, что JSON чаще используется внутри организации или в веб-службах и мобильных приложениях – в силу своей простоты, тогда как XML является основой для создания жестко структурированных документов и форматов, гарантирующих возможность обмена данными меж-

ду разными организациями. Например, в нескольких стандартах консорциума **Open Geospatial Consortium** (**OGC**), в частности в стандарте картографических служб, в качестве формата обмена используется XML с соответствующей XSD-схемой.

Типы данных JSON в PostgreSQL

PostgreSQL поддерживает два типа JSON: `JSON` и `JSONB`, тот и другой реализуют спецификацию RFC 7159. Оба типа применимы для контроля правил JSON. Они почти идентичны, но JSONB эффективнее, потому что является двоичным форматом и поддерживает индексы.

При работе с типом JSON рекомендуется задавать для базы данных кодировку UTF8, чтобы обеспечить его совместимость со стандартом RFC 7159. Документ, сохраняемый как объект типа `JSON`, хранится в текстовом формате. Если же JSON-объект сохраняется как `JSONB`, то примитивные типы JSON – string, boolean, number – отображаются, соответственно, на типы text, Boolean и numeric.

Доступ к объектам типа JSON и их модификация

Если текст приводится к типу `json`, то он сохраняется и отображается без какой-либо обработки, поэтому сохраняются все пробелы, форматирование чисел и другие детали. В формате `jsonb` эти детали не сохраняются, как показывает следующий пример:

```
car_portal=# SELECT '{"name":"some name", "name":"some name" }'::json;
                      json
-----------------------------------------------
 {"name":"some name", "name":"some name" }
(1 row)

car_portal=#
car_portal=# SELECT '{"name":"some name", "name":"some name" }'::jsonb;
         jsonb
----------------------
 {"name": "some name"}
(1 row)
```

JSON-объекты могут содержать вложенные JSON-объекты, массивы, вложенные массивы, массивы JSON-объектов и т. д. Глубина вложенности произвольна, так что можно конструировать весьма сложные JSON-документы. В одном массиве JSON-документа могут находиться элементы разных типов. В следующем примере показано, как создать учетную запись, в которой имя – текст, адрес – JSON-объект, а ранг – массив:

```
car_portal=# SELECT '{"name":"John", "Address":{"Street":"Some street",
"city":"Some city"}, "rank":[5,3,4,5,2,3,4,5]}'::JSONB;
jsonb
------------------------------------------------------------------------
```

```
----------------------------------
 {"name": "John", "rank": [5, 3, 4, 5, 2, 3, 4, 5], "Address": {"city":
"Some city", "Street": "Some street"}}
(1 row)
```

Можно получить поле JSON-объекта в виде JSON-объекта или текста. К полям можно также обращаться по индексу или по имени поля. В следующей таблице перечислены операторы доступа:

Json	Текст	Описание
->	->>	Возвращает поле JSON-объекта по индексу или по имени поля
#>	#>>	Возвращает поле JSON-объекта по указанному пути

Чтобы получить адрес и город из созданного ранее JSON-объекта, у нас есть два способа (отметим, что имена полей в JSON чувствительны к регистру):

```
CREATE TABLE json_doc ( doc jsonb );
INSERT INTO json_doc SELECT '{"name":"John", "Address":{"Street":"Some
street", "city":"Some city"}, "rank":[5,3,4,5,2,3,4,5]}'::JSONB ;
```

Если мы хотим вернуть города в текстовом формате, то к нашим услугам операторы ->> и #>>, например:

```
SELECT doc->'Address'->>'city', doc#>>'{Address, city}' FROM json_doc WHERE
doc->>'name' = 'John';
 ?column?  | ?column?
-----------+-----------
 Some city | Some city
(1 row)
```

В старых версиях Postgres, например 9.4, манипулировать JSON-документами было затруднительно. Но впоследствии появилось много операторов, в т. ч. || для конкатенации двух JSON-объектов и - для удаления пары ключ-значение. Раньше мы должны были преобразовать JSON-объект в текст, затем с помощью регулярных выражений найти и заменить или удалить элемент и, наконец, привести текст обратно к типу JSON. Так, чтобы удалить ранг из учетной записи, нужно было проделать следующие манипуляции:

```
SELECT (regexp_replace(doc::text, '"rank":(.*)],',''))::jsonb
FROM json_doc
WHERE doc->>'name' = 'John';
```

В версиях 9.5 и 9.6 соответственно появились функции `jsonb_set` и `json_insert`. Вот как можно вставить в JSON-объект пару ключ-значение:

```
car_portal=# UPDATE json_doc SET doc = jsonb_insert(doc,
'{hobby}','["swim", "read"]', true) RETURNING * ;
doc
------------------------------------------------------------------------
------------------------------------------------------------------
 {"name": "John", "rank": [5, 3, 4, 5, 2, 3, 4, 5], "hobby": ["swim",
"read"], "Address": {"city": "Some city", "Street": "Some street"}}
```

А вот так изменяется существующая пара ключ-значение:

```
car_portal=# update json_doc SET doc = jsonb_set(doc, '{hobby}','["read"]',
true) RETURNING * ;
doc
------------------------------------------------------------------------------
-----------------------------------------------------
 {"name": "John", "rank": [5, 3, 4, 5, 2, 3, 4, 5], "hobby": ["read"],
"Address": {"city": "Some city", "Street": "Some street"}}
```

И наконец, удалим пару ключ-значение:

```
car_portal=# update json_doc SET doc = doc -'hobby' RETURNING * ;
doc
------------------------------------------------------------------------------
-----------------------------------
 {"name": "John", "rank": [5, 3, 4, 5, 2, 3, 4, 5], "Address": {"city":
"Some city", "Street": "Some street"}}
```

Полный перечень функций и операторов, связанных с типом JSON, см. на странице https://www.postgresql.org/docs/current/static/functions-json.html.

Индексирование JSON-документов

Для индексирования JSONB-документов используется GIN-индекс, который поддерживает следующие операторы:

- @>: содержит ли JSON-документ в левой части значение в правой части?
- ?: существует ли указанный ключ в JSON-документе?
- ?&: существует ли в JSON-документе хотя бы один из элементов текстового массива?
- ?|: существуют ли в JSON-документе все элементы текстового массива (ключи)?

Чтобы убедиться, что индексирование таблицы `json_doc` действительно оказывает действие, создадим индекс и запретим последовательный просмотр:

```
CREATE INDEX ON json_doc(doc);
SET enable_seqscan = off;
```

Теперь проверим, что индекс используется:

```
car_portal=# EXPLAIN SELECT * FROM json_doc WHERE doc @> '{"name":"John"}';
QUERY PLAN
------------------------------------------------------------------------------
-------------
 Index Only Scan using json_doc_doc_idx on json_doc (cost=0.13..12.16
rows=1 width=32)
  Filter: (doc @> '{"name": "John"}'::jsonb)
(2 rows)
```

JSON-документ можно преобразовать во множество с помощью функции `json_to_record()`. Это полезно, потому что позволяет сортировать и фильтровать данные, как в обычных таблицах. Кроме того, мы можем агрегировать данные из нескольких строк и сконструировать массив JSON-объектов с помощью функции `jsosn_agg()`.

Реализация REST-совместимого интерфейса к PostgreSQL

REST-совместимый API удобен для организации интерфейса, позволяющего обмениваться данными, которые совместно используются несколькими приложениями. Пусть некоторая таблица используется в нескольких приложениях. Один из способов дать к ней доступ из приложения состоит в том, чтобы создать для нее **объект доступа к данным** (data access object – **DAO**), обернуть его библиотекой, а затем использовать эту библиотеку во всех приложениях. У такого подхода есть недостатки, например разрешение зависимостей от библиотеки и несовпадение версий библиотеки. Кроме того, для развертывания новых версий библиотеки требуются нетривиальные усилия: компиляция, тестирование и развертывание.

Преимущество REST-совместимого API к базе данных PostgreSQL состоит в том, что он обеспечивает простой доступ к данным и позволяет разработчику включить архитектуру микросервисов, что благотворно влияет на гибкость.

Существует несколько библиотек с открытым исходным кодом, позволяющих построить такой REST-совместимый интерфейс, в т. ч. `psql-api` и `PostgREST`. Для создания полного API со всеми операциями CRUD на базе веб-сервера имеется модуль PostgreSQL для Nginx – `ngx_postgres`, простой в использовании и хорошо поддерживаемый. К сожалению, он не оформлен в виде пакета Debian, а значит, придется откомпилировать и установить его вручную.

В следующем примере представлен REST-совместимый API для выборки данных на основе Nginx и memcached. Предполагается, что данные целиком умещаются в памяти. PostgreSQL отправляет данные серверу memcached, и Nginx запрашивает их у этого сервера. Memcached используется для кеширования данных и служит промежуточным уровнем между Postgres и Nginx.

Поскольку memcached представляет собой хранилище ключей и значений, необходимо определить ключ для доступа к данным. Nginx разбирает URI-адрес и использует переданные в нем аргументы для отображения запроса на значение, хранящееся в memcached. В рассматриваемом ниже примере мы не ставим задачу научить вас конструированию REST-совместимых API, это всего лишь демонстрации техники. Кроме того, на практике можно было бы совместить кеширование с модулем `ngx_postgres` для построения высокопроизводительной системы со всеми операциями CRUD. Объединив memcached с `ngx_postgres`, мы могли бы снять ограничение на размещение всех данных в оперативной памяти.

Для отправки данных серверу memcached нужен модуль PostgreSQL `pgmemcache`. Добавим его в базу данных `template1`, чтобы он был доступен всем новым базам. Кроме того, добавим его в базу `car_portal`:

```
$sudo apt-get install -y postgresql-10-pgmemcache
$psql template1 -c "CREATE EXTENSION pgmemcache"
$psql car_portal -c "CREATE EXTENSION pgmemcache"
```

Для установки Nginx и memcached в ОС на основе дистрибутивов Ubuntu или Debian выполните следующие команды:

```
$sudo apt-get install -y nginx
$sudo apt-get install -y memcached
```

А чтобы на постоянной основе связать серверы memcached и PostgreSQL, добавьте следующую переменную в блок пользовательских параметров в файле PostgreSQL.conf:

```
$echo "pgmemcache.default_servers =
'localhost'">>/etc/postgresql/10/main/postgresql.conf
$/etc/init.d/postgresql reload
```

Для тестирования memcached и расширения pgmemcache воспользуемся функциями memcached_add(key,value) и memcached_get(key), чтобы записать в словарь memcached значение, а затем извлечь его:

```
car_portal=# SELECT memcache_add('/1', 'hello');
 memcache_add
--------------
 t
(1 row)

car_portal=# SELECT memcache_get('/1');
 memcache_get
--------------
 hello
(1 row)
```

Для демонстрации мы воспользовались Linux из-за простоты установки. Если вы работаете в Windows, то можете установить какой-нибудь вариант Linux на виртуальную машину. Что касается сервера Nginx, то его можно установить в Windows, но функциональность будет ограничена. Дополнительные сведения см. на странице http://nginx.org/en/docs/windows.html.

Получить полный перечень функций memcached позволяет метакоманда \df memcache_*.

Чтобы разрешить Nginx доступ к memcached, необходимо изменить конфигурацию и перезагрузить Nginx. Ниже приведен минимальный конфигурационный файл Nginx, достаточный для этой цели. Отметим, что в Ubuntu конфигурационный файл Nginx находится в каталоге /etc/nginx/nginx.conf:

```
# cat /etc/nginx/nginx.conf
user www-data;
worker_processes 4;
pid /run/nginx.pid;
events {
  worker_connections 800;
}

http {
  server {
    location / {
      set $memcached_key "$uri";
      memcached_pass 127.0.0.1:11211;
      default_type application/json;
```

```
      add_header x-header-memcached true;
    }
  }
}
```

В этом примере веб-сервер Nginx обрабатывает ответы от сервера memcached, адрес которого определен переменной `memcached_pass`. В качестве ключа серверу memcached передается **универсальный идентификатор ресурса** (**URI**). По умолчанию ответ приходит в формате JSON. Наконец, в целях отладки добавляется заголовок `x-header-memcached`.

Чтобы проверить настройку сервера Nginx, получим значение ключа /1, сгенерированное расширением `pgmemcache`:

```
curl -I -X GET http://127.0.0.1/1
HTTP/1.1 200 OK
Server: nginx/1.10.3 (Ubuntu)
Date: Wed, 01 Nov 2017 17:29:07 GMT
Content-Type: application/json
Content-Length: 5
Connection: keep-alive
x-header-memcached: true
Accept-Ranges: bytes
```

Сервер успешно ответил на запрос, получен ответ типа JSON, и, как показывает заголовок, ответ пришел от memcached. Предположим, что мы хотим реализовать REST-совместимую веб-службу, которая возвращает информацию об учетной записи пользователя из таблицы `account`, включая идентификатор, имя, фамилию и адрес электронной почты. Для построения JSON-документа из строки реляционной таблицы предназначены функции `row_to_json()`, `to_json()` и `to_jsonb()`:

```
car_portal=# SELECT to_json (row(account_id,first_name, last_name, email))
FROM car_portal_app.account LIMIT 1;
                          to_json
-------------------------------------------------------
{"f1":1,"f2":"James","f3":"Butt","f4":"jbutt@gmail.com"}
(1 row)

car_portal=# SELECT to_json (account) FROM car_portal_app.account LIMIT 1;
to_json
-------------------------------------------------------------------------------
-------------------------------------------------------
{"account_id":1,"first_name":"James","last_name":"Butt","email":"jbutt@gmai
l.com","password":"1b9ef408e82e38346e6ebebf2dcc5ece"}
(1 row)
```

В этом примере из-за использования конструкции `row(account_id, first_name, last_ name, email)` функция `to_json` не смогла определить имена атрибутов, поэтому они были заменены на `f1`, `f2` и т. д.

Чтобы решить эту проблему, необходимо присвоить строке имя. Сделать это можно разными способами, например использовать подзапросы или сопоставить результатам псевдонимы. В примере ниже мы задали псевдонимы в CTE:

```
WITH account_info(account_id, first_name, last_name, email) AS (
    SELECT account_id,first_name, last_name, email
    FROM car_portal_app. Account
    LIMIT 1)
SELECT to_json(account_info) FROM account_info;
to_json
--------------------------------------------------------------------------------
{"account_id":1,"first_name":"James","last_name":"Butt","email":"jbutt@gmail.com"}
(1 row)
```

Чтобы сгенерировать записи, соответствующие таблице `account`, используя первичный ключ `account_id` в качестве ключа хеша, выполним запрос:

```
SELECT memcache_add('/'||account_id,
  (SELECT to_json(foo)
   FROM (SELECT account_id, first_name,last_name, email ) AS FOO )::text
  )
FROM car_portal_app.account;
```

После этого уже можно обращаться к данным через сервер Nginx. Чтобы убедиться в этом, получим JSON-представление учетной записи с `account_id`, равным 1:

```
$ curl -sD - -o -X GET http://127.0.0.1/2
HTTP/1.1 200 OK
Server: nginx/1.10.3 (Ubuntu)
Date: Wed, 01 Nov 2017 17:39:37 GMT
Content-Type: application/json
Content-Length: 103
Connection: keep-alive
x-header-memcached: true
Accept-Ranges: bytes
{"account_id":2,"first_name":"Josephine","last_name":"Darakjy","email":"jos
ephine_darakjy@darakjy.org"}
```

Необходимо также позаботиться об обработке откатов транзакций и соответствующей стратегии кеширования, чтобы устранить рассогласование кеша и базы данных. В примере ниже показано, как откат влияет на данные в memcached:

```
BEGIN
car_portal=# SELECT memcache_add('is_transactional?', 'No');
 memcache_add
--------------
t
(1 row)

car_portal=# Rollback;
ROLLBACK

car_portal=# SELECT memcache_get('is_transactional?');
 memcache_get
--------------
 No
(1 row)
```

Чтобы гарантировать согласованность таблицы account с данными на сервере memcached, можно добавить триггер AFTER INSERT OR UPDATE OR DELETE, который передаст в memcached изменения, произведенные в таблице. Триггер можно пометить флагами DEFERRABLE INITIALLY DEFERRED, чтобы отложить модификацию данных в memcached. Иными словами, состояние memcached изменяется, только если транзакция завершилась успешно.

Полнотекстовый поиск в PostgreSQL

PostgreSQL предоставляет средства полнотекстового поиска, которые позволяют заместить операторы сопоставления с текстом типа LIKE и ILIKE и резко повысить скорость поиска. Например, хотя индексирование текста с применением класса text_pattern_op поддерживается, индекс используется только тогда, когда образец привязан к началу столбца.

У традиционных операторов LIKE и ILIKE есть и еще одна проблема – отсутствие ранжирования на основе сходства и поддержки естественного языка. Эти операторы могут вернуть только булево значение: TRUE или FALSE.

Помимо ранжирования и поиска в любом месте текста, средства полнотекстового поиска в PostgreSQL обладают и многими другими возможностями. Благодаря поддержке словарей открывается возможность учитывать особенности языка, в т. ч. синонимию.

Типы данных tsquery и tsvector

Полнотекстовый поиск основан на типах данных tsvector и tsquery. Тип tsvector представляет нормализованный документ, а tsquery – запрос.

Тип данных tsvector

Тип данных tsvector представляет собой отсортированный список уникальных лексем. Лексема – это основа слова; проще говоря, это корень слова без суффикса, без учета форм склонения и грамматических вариантов. В примере ниже показано, как привести текст к типу tsvector:

```
car_portal=# SELECT 'A wise man always has something to say, whereas a fool
always needs to say something'::tsvector;
tsvector
------------------------------------------------------------------------
----------------
 'A' 'a' 'always' 'fool' 'has' 'man' 'needs' 'say' 'say,' 'something' 'to'
'whereas' 'wise'
(1 row)
```

Приведение текста к типу tsvector еще не означает полной нормализации документа, потому что не учитываются лингвистические правила. Для правильной нормализации нужно воспользоваться функцией to_tsvector():

```
car_portal=# SELECT to_tsvector('english', 'A wise man always has something
to say, whereas a fool always needs to say something');
```

```
to_tsvector
------------------------------------------------------------------------
------------
 'alway':4,12 'fool':11 'man':3 'need':13 'say':8,15 'someth':6,16
'wherea':9 'wise':2
(1 row)
```

Как видим, функция `to_tsvector` удалила некоторые буквы, например `s` из слова `always`, и добавила целочисленные позиции лексем, необходимые для ранжирования с учетом близости слов.

Тип данных tsquery

Тип данных `tsquery` используется для поиска лексем. Лексемы можно комбинировать с помощью операторов & (И), | (ИЛИ) и ! (НЕ). Оператор НЕ имеет наивысший приоритет, И – следующий по порядку, и самый меньший – ИЛИ. Для группировки лексем и операторов можно также использовать скобки. В примере ниже показано, как производится поиск лексем с помощью типов `tsquery`, `tsvector` и оператора сопоставления @@:

```
car_portal=# SELECT 'A wise man always has something to say, whereas a fool
always needs to say something'::tsvector @@ 'wise'::tsquery;
 ?column?
----------
 t
(1 row)
```

Для типа `tsquery` имеется также функция `to_tsquery`, которая преобразует текст в лексемы:

```
car_portal=# SELECT to_tsquery('english', 'wise & man');
   to_tsquery
----------------
 'wise' & 'man'
(1 row)
```

Реализован также поиск по фразам, когда результат считается положительным, если два слова соседние и расположены в указанном порядке. Для этого предназначен оператор <->:

```
car_portal=# SELECT to_tsvector('A wise man always has something to say,
whereas a fool always needs to say something') @@ to_tsquery('wise <->
man');
 ?column?
----------
 t
(1 row)
```

Сопоставление с образцом

Под сопоставлением с образцом понимается проверка вхождения некоторого образца в заданную последовательность слов. На результат операции влияет несколько факторов, в том числе:

- нормализация текста;
- словарь;
- ранжирование.

Если текст не нормализован, то поиск может не вернуть ожидаемый результат. В примерах ниже показано, как может завершиться неудачей поиск в ненормализованном тексте:

```
car_portal=# SELECT 'elephants'::tsvector @@ 'elephant';
 ?column?
----------
 f
(1 row)
```

В этом запросе приведение слова elephants к типу `tsvector` и неявное преобразование elephant в запрос не порождает нормализованных лексем из-за отсутствия информации о словаре. Чтобы добавить недостающую информацию, можно воспользоваться функциями `to_tsvector` и `to_tsquery`:

```
car_portal=# SELECT to_tsvector('english', 'elephants') @@
to_tsquery('english', 'elephant');
 ?column?
----------
 t
(1 row)

car_portal=#
car_portal=# SELECT to_tsvector('simple', 'elephants') @@
to_tsquery('simple', 'elephant');
 ?column?
----------
 f
(1 row)
```

Полнотекстовый поиск поддерживает сопоставление с образцом с учетом рангов. Лексемы в векторе `tsvector` можно снабдить метками `A`, `B`, `C`, `D`, где `D` – ранг по умолчанию, а `A` соответствует наивысшему рангу. Чтобы явно назначить вес вектору `tsvector`, можно воспользоваться функцией `setweight`:

```
car_portal=# SELECT setweight(to_tsvector('english', 'elephants'),'A') ||
setweight(to_tsvector('english', 'dolphin'),'B');
        ?column?
------------------------
'dolphin':2B 'eleph':1A
(1 row)
```

Для ранжирования предусмотрены две функции: `ts_rank` и `ts_rank_cd`. Функция `ts_rank` используется для стандартного ранжирования, а `ts_rank_cd` – для ранжирования с учетом плотности покрытия (cover density ranking). В следующем примере показаны результаты `ts_rank_cd` при поиске по словам `eleph` и `dolphin`:

```
car_portal=# SELECT ts_rank_cd
(setweight(to_tsvector('english','elephants'),'A') ||
setweight(to_tsvector('english', 'dolphin'),'B'),'eleph' );
 ts_rank_cd
------------
          1
(1 row)

car_portal=# SELECT ts_rank_cd
(setweight(to_tsvector('english','elephants'),'A') ||
setweight(to_tsvector('english', 'dolphin'),'B'),'dolphin' );
ts_rank_cd
------------
0.4
(1 row)
```

Ранжирование часто используется для обогащения, фильтрации и упорядочения результатов сопоставления с образцом. В реальных приложениях у разных частей документа могут быть разные веса. Например, когда мы ищем фильм, наибольший вес следует назначить названию фильма и главному герою, а краткому описанию сюжета можно назначить меньший вес.

Полнотекстовые индексы

Для полнотекстового поиска предпочтительны GIN-индексы. Чтобы создать индекс над документом, можно воспользоваться функцией `to_tsvector`:

```
CREATE INDEX ON <table_name> USING GIN (to_tsvector('english', <attribute name>));
-- или
CREATE INDEX ON <table_name> USING GIN (to_tsvector(<attribute name>));
```

Выбор индекса зависит от предиката в запросе. Так, если предикат имеет вид `to_tsvector('english',...) @@ to_tsquery (...)`, то при обработке запроса будет использоваться первый индекс.

Для индексирования `tsvector` и `tsquery` можно использовать GIST-индекс, тогда как GIN-индекс используется для индексирования одного лишь `tsvector`. В `GIST`-индексе теряется часть информации, и он может возвращать ложноположительные результаты. Поэтому PostgreSQL автоматически перепроверяет возвращенный результат, отбрасывая ложноположительные совпадения. Из-за расходов на доступ к записям производительность может снизиться. В GIN-индексе хранятся только лексемы вектора `tsvector` без весовых меток. Поэтому если в запросе участвуют веса, то можно считать, что и GIN-индексы теряют информацию.

Производительность GIN- и GIST-индексов зависит от количества уникальных слов, поэтому для уменьшения общего количества таковых рекомендуется использовать словари. Поиск по GIN-индексу работает быстрее, но сам индекс строится медленнее и занимает больше места, чем GIST-индекс. Увеличение параметра `maintenance_work_mem` может ускорить построение GIN-индекса, но никак не влияет на GIST-индекс.

Резюме

PostgreSQL обладает весьма развитой системой встроенных и дополнительных типов. Для расширения можно использовать языки C и C++. Более того, PostgreSQL предлагает инфраструктуру создания расширений PGXS, позволяющую собирать расширения для установленного сервера. Для обсуждения некоторых расширений, например PostGIS, понадобилась бы целая глава.

В PostgreSQL имеется обширный набор типов данных: XML, `hstore`, JSON, массив и т. д. Типы данных нужны для того, чтобы разработчик не изобретал велосипед, а мог воспользоваться всем богатством функций и операторов. Кроме того, некоторые типы данных, в т. ч. `hstore` и JSON, позволяют оперативнее реагировать на изменение требований, потому что не требуют внесения изменений в схему базы данных.

Массивы в PostgreSQL достигли высокой степени зрелости и обладают большим набором операторов и функций. PostgreSQL может работать с многомерными массивами разных базовых типов. Массивы полезны при моделировании многозначных атрибутов, а также для ряда задач, которые трудно решить, оставаясь в рамках строгой реляционной модели слабо структурированных данных, а также для обеспечения большей гибкости разработки.

Поддерживаются документы в форматах JSON и XML, что позволяет обмениваться документами в различных форматах. PostgreSQL предоставляет ряд функций для преобразования строк таблицы в формат JSON и обратно, что делает возможным использование сервера для поддержки REST-совместимых веб-служб.

PostgreSQL поддерживает полнотекстовый поиск, что полезно для решения задач лингвистики, а также повышает производительность поиска без привязки к началу текста. В результате пользователь доволен и счастлив.

В следующей главе мы подробно обсудим свойства ACID и их связь с управлением конкурентностью. Там же будут рассмотрены уровни изоляции и на примерах продемонстрированы их побочные эффекты. Наконец, мы поговорим о различных методах блокировки, включая пессимистичные стратегии: блокировку на уровне строк и рекомендательные блокировки.

Глава 10

Транзакции и управление параллельным доступом

В реляционной модели описана логическая единица обработки данных – *транзакция*. Можно сказать, что транзакция – это множество последовательно выполняемых операций. РСУБД предоставляет механизм блокировки, гарантирующий целостность транзакций.

В этой главе мы рассмотрим основные концепции, обеспечивающие правильное выполнение транзакций, а также обсудим проблему управления параллельным доступом, системы блокировки, взаимоблокировки и рекомендательные блокировки.

Транзакции

Транзакция – это множество операций, в состав которого могут входить операции обновления, удаления, вставки и выборки данных. Часто эти операции погружаются в язык более высокого уровня или явно обертываются в блок транзакции, заключенный между командами `BEGIN` и `END`. Транзакция считается успешно выполненной, если успешно выполнены все составляющие ее операции. Если какая-то операция транзакции завершается неудачно, то частично выполненные действия можно откатить.

Для явного управления транзакцией можно поставить команду `BEGIN` в ее начале и команду `END` или `COMMIT` в конце. В следующем примере показано, как выполнить команду SQL в транзакции:

```
BEGIN;
CREATE TABLE employee (id serial primary key, name text, salary numeric);
COMMIT;
```

Помимо обеспечения целостности данных транзакции позволяют легко отменить изменения, внесенные в базу в процессе разработки и отладки. В интерактивном режиме блок транзакции можно сочетать с командой `SAVEPOINT`, чтобы поставить отметку в точке сохранения состояния. Точка сохранения –

это способ откатить не всю транзакцию целиком, а только ее часть. В примере ниже демонстрируется использование SAVEPOINT:

```
BEGIN;
UPDATE employee set salary = salary*1.1;
SAVEPOINT increase_salary;
UPDATE employee set salary = salary + 500 WHERE name ='john';
ROLLBACK to increase_salary;
COMMIT;
```

Здесь команда `UPDATE employee SET salary = salary*1.1;` зафиксирована в базе данных, а команда `UPDATE employee SET salary = salary + 500 WHERE name = 'john';` откачена.

Все выполняемые PostgreSQL команды транзакционные, даже если блок транзакции не был открыт явно. Когда мы выполняем несколько команд, не обертывая их блоком транзакции, каждая команда выполняется в отдельной транзакции.

Поведением транзакций управляют драйверы базы данных и каркасы приложений типа Java EE или Spring. Например, в случае JDBC мы можем при желании задать режим автоматической фиксации.

Посмотрите в документации, добавляет ли клиент команды `BEGIN` и `COMMIT` автоматически.

Транзакции и свойства ACID

Гарантия атомарности, согласованности, изолированности и долговечности (ACID) операций – фундаментальное свойство реляционной базы данных.

Транзакция – это логическая единица выполнения, она неделима, или **атомарна**, т. е. либо выполнена целиком, либо не выполнена вовсе; и это утверждение справедливо вне зависимости от причины ошибки. Например, транзакция может завершиться неудачно из-за математической ошибки, неправильно написанного имени отношения и даже из-за аварии операционной системы.

После успешной фиксации транзакции все произведенные в ней изменения должны сохраняться даже после отказов оборудования; это свойство называется **долговечностью**. В многопользовательской среде пользователи могут одновременно выполнять несколько транзакций, каждая из которых содержит несколько операций. Любая транзакция должна выполняться, не мешая другим транзакциям, выполняющимся вместе с ней; это свойство называется **изолированностью**.

Наконец, **согласованность** – это не свойство транзакции как таковой, а желательное следствие изолированности и атомарности. Согласованность базы данных непосредственно связана с бизнес-требованиями, которые определяются с помощью правил, триггеров и ограничений. Если база данных находилась в согласованном состоянии до выполнения транзакции, то состояние должно быть согласовано и после ее завершения. Следить за согласованностью базы данных – задача разработчика, запрограммировавшего транзакцию.

Обеспечить свойства ACID нелегко. Например, **долговечность** – трудноисполнимое требование, потому что факторов, способных привести к потере данных, множество: крах операционной системы, прекращение электроснабжения, отказ жесткого диска и т. д. К тому же компьютерная архитектура весьма сложна, в частности до появления на жестком диске данные проходят через несколько уровней системы хранения: оперативная память, буферы ввода-вывода, кеш диска.

По умолчанию PostgreSQL пользуется системным вызовом `fsync`, который сбрасывает все модифицированные страницы кеша на диск, чтобы изменения были зафиксированы даже в случае краха или перезагрузки системы.

Транзакции и конкурентность

Конкурентность можно определить как чередование действий во времени для создания иллюзии одновременного выполнения. При этом необходимо решить несколько проблем, в т. ч. как обеспечить безопасный доступ к глобальным ресурсам и как обрабатывать ошибки, когда имеется несколько контекстов выполнения.

Конкурентность встречается в компьютерах повсеместно, начиная с низкоуровневой аппаратной логики и кончая доступом к общим данным из любой точки земного шара. Примеры конкурентности можно найти в организации конвейера команд в микропроцессорах и в многопоточной архитектуре операционной системы. Даже в своей повседневной практике мы сталкиваемся с конкурентностью, работая, например, с системой Git. Взять типичную ситуацию – несколько разработчиков работают с одной и той же кодовой базой, при этом возникают конфликты, которые в конечном счете разрешаются в процедуре объединения.

Не следует путать конкурентность и параллелизм. Под параллелизмом понимается использование дополнительных вычислительных устройств, которые могут выполнить больше работы в единицу времени, тогда как конкурентность касается логического доступа к разделяемым ресурсам. Можно считать, что физический параллелизм – частный случай конкурентности. Другой ее формой является квантование процессорного времени, т. е. виртуальный параллелизм.

Во многих базах данных параллелизм является дополнением к конкурентности. В PostgreSQL параллелизм используется начиная с версии 9.6, чтобы увеличить скорость выполнения запросов. В СУБД Greenplum, ответвившейся от PostgreSQL, также применяется **массово параллельная обработка** (massively parallel processing – **MPP**), чтобы повысить производительность и справиться с нагрузками, характерными для хранилищ данных.

Фундаментальной проблемой конкурентности является чередование действий во время доступа к общему глобальному ресурсу. Конфликта не произойдет, если код осуществляет доступ строго последовательно, а не конкурентно. Но на практике это вызывает значительные задержки.

Задача управления конкурентностью заключается в том, чтобы координировать выполнение конкурентных транзакций в многопользовательской среде и обрабатывать потенциальные проблемы: потерянные обновления, незафиксированные данные и несогласованную выборку.

Конкурентность – рыночное преимущество PostgreSQL, поскольку эта СУБД корректно и эффективно решает проблемы даже при очень высокой интенсивности операций чтения и записи. В PostgreSQL операции записи не блокируют операции чтения, и наоборот. Применяемый в PostgreSQL метод **управления параллельным доступом с помощью многоверсионности** (multiversion concurrency control – **MVCC**) используется и в нескольких других коммерческих и некоммерческих СУБД, включая Oracle, MySQL с движком InnoDB, Informix, Firebird и CouchDB.

Отметим, что MVCC – не единственный способ обеспечения конкурентности в РСУБД; в SQL Server вместо него применяется метод **сильной строгой двухфазной блокировки** (strong strict two-phase locking – **SS2PL**).

MVCC в PostgreSQL

Одновременный доступ к одним и тем же данным для чтения или записи может привести к несогласованному состоянию данных или к неправильному результату чтения. Например, читатель может получить частично записанные данные. Решить эту проблему проще всего, запретив читателю читать данные, пока писатель не закончит их записывать. Но при таком решении процессы состязаются за получение доступа, что приводит к снижению производительности, т. к. читатель должен ждать писателя.

В MVCC операция обновления создает новую версию данных, а исходные данные не перезаписываются. Таким образом, одновременно существует несколько версий данных, отсюда и слово «многоверсионность». К какой версии данных получит доступ транзакция, зависит от ее уровня изоляции, текущего состояния и версий кортежей.

В PostgreSQL любой транзакции присваивается идентификатор, называемый **XID**. **XID** увеличивается на единицу всякий раз, как начинается новая транзакция. Идентификаторы транзакций – четырехбайтовые целые числа без знака, так что всего их примерно 4,2 миллиарда. Вычисление новых идентификаторов производится по модулю, т. е. XID'ы изменяются по кругу. Все транзакции (примерно 2 миллиарда), предшествующие данной, считаются старыми и видны данной транзакции.

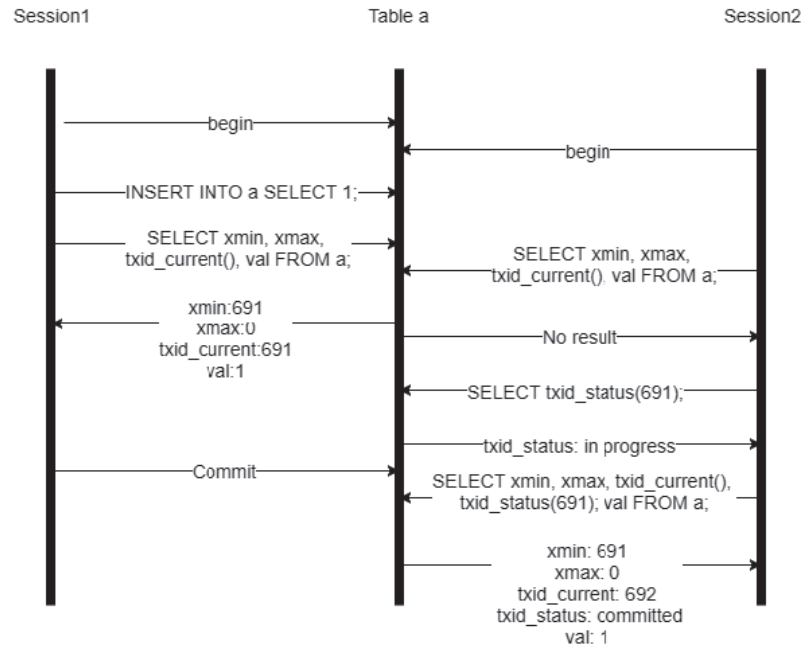
В примере ниже показано, как увеличиваются идентификаторы транзакций. Отметим, что если блок транзакции не открывается явно, то **XID** увеличивается после каждой команды.

```
test=# SELECT txid_current();
 txid_current
--------------
          682
(1 row)
```

```
test=# SELECT 1;
 ?column?
----------
        1
(1 row)

test=# SELECT txid_current();
 txid_current
--------------
          683
(1 row)

test=# BEGIN;
BEGIN
test=# SELECT txid_current();
 txid_current
--------------
          684
(1 row)

test=# SELECT 1;
 ?column?
----------
        1
(1 row)

test=# SELECT txid_current();
 txid_current
--------------
          684
(1 row)
```

Описанный механизм вычисления идентификаторов транзакций означает, что необходимы меры предосторожности. Во-первых – и это очень важно – никогда не отключайте процесс очистки (vacuum), иначе база данных рано или поздно остановится из-за циклического оборачивания идентификатора. Кроме того, заключайте команды массовой вставки или обновления в явные блоки транзакции. Наконец, имейте в виду, что активно обновляемые таблицы могут «разбухать» из-за большого числа мертвых строк.

PostgreSQL записывает идентификатор транзакции, создавшей кортеж, во внутреннее поле `xmin`, а идентификатор транзакции, удалившей этот кортеж, – в поле `xmax`. Операция обновления в MVCC реализуется как пара команд `DELETE` и `INSERT`, и в таком случае создается еще одна версия кортежа. Для решения проблем конкурентности PostgreSQL использует информацию о создании и удалении кортежа, состоянии транзакции (зафиксирована, выполняется, откачена), уровнях изоляции и видимости транзакции. На рисунке ниже показано, как идентификаторы транзакций записываются в поле `XMIN` при вставке записи:



Уровни изоляции транзакций

Разработчик может задать уровень изоляции транзакции, выполнив такую команду SQL:

```
SET TRANSACTION ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ
COMMITTED | READ UNCOMMITTED}
```

Команда `SET TRANSACTION ISOLATION LEVEL` должна вызываться внутри блока транзакции до начала запроса, иначе она не возымеет эффекта. Существует и другой способ:

```
BEGIN TRANSACTION ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ
COMMITTED | READ UNCOMMITTED}
```

Наконец, можно изменить уровень изоляции для всей базы данных, например:

```
ALTER DATABASE <DATABASE NAME> SET DEFAULT_TRANSACTION_ISOLATION TO SERIALIZABLE;
```

Как следует из этих примеров, существует четыре уровня изоляции:

- `SERIALIZABLE`: обеспечивает самую строгую согласованность и освобождает разработчика от необходимости думать о конкурентности. Но за это приходится расплачиваться производительностью. В стандарте SQL уровень `SERIALIZABLE` подразумевается по умолчанию. В PostgreSQL по умолчанию подразумевается уровень `READ COMMITTED`;

- REPEATABLE READ: следующий по строгости уровень изоляции. Он похож на READ COMMITTED тем, что допускает чтение только зафиксированных данных, но дополнительно гарантирует, что прочитанные данные не изменятся при повторном чтении;
- READ COMMITTED: уровень, подразумеваемый по умолчанию в PostgreSQL. Разрешает транзакции читать только зафиксированные данные. При таком уровне предпочтение отдается производительности, а не точности;
- READ UNCOMMITTED: самый нестрогий уровень изоляции. Допускает чтение незафиксированных данных.

Уровень READ UNCOMMITTED в PostgreSQL не поддерживается и трактуется так же, как READ COMMITTED. PostgreSQL поддерживает только три уровня изоляции.

Что такое уровень изоляции, проще объяснить, описав побочные эффекты, возникающие на каждом уровне:

- грязное чтение – это происходит, когда транзакция читает данные из кортежа, который был модифицирован другой работающей транзакцией и еще не зафиксирован. В PostgreSQL этого не может случиться, т. к. уровень READ UNCOMMITTED не поддерживается;
- неповторяемое чтение – это происходит, если в одной транзакции некоторая строка читается дважды и при этом получаются разные результаты. Такое бывает, если уровень изоляции равен READ COMMITTED, и зачастую является следствием многократного обновления строки другими транзакциями;
- фантомное чтение – это происходит, если на протяжении транзакции новая строка (или строки) сначала появляется, а потом исчезает. Часто это результат зафиксированной вставки, за которой следует зафиксированное удаление. Фантомное чтение имеет место, когда в одной транзакции несколько раз выбирается один и тот же набор строк, а результаты отличаются. Согласно стандарту SQL, фантомное чтение возможно, когда уровень изоляции равен READ COMMITTED и REPEATABLE READ, но в PostgreSQL оно возможно только в режиме READ COMMITTED;
- аномалия сериализации – результат выполнения группы транзакций зависит от порядка их выполнения. Это может случиться только на уровне REPEATABLE READ.

Стандарт SQL допускает фантомное чтение при уровне изоляции REPEATABLE READ. В PostgreSQL это не разрешено. Поэтому на уровне REPEATABLE READ возможна только аномалия сериализации.

Продемонстрируем эффект неповторяемого чтения на примере следующей таблицы test_tx_level:

```
postgres=# CREATE TABLE test_tx_level AS SELECT 1 AS val;
SELECT 1
postgres=# TABLE test_tx_level ;
```

```
 val
-----
   1
(1 row)
```

Будем использовать подразумеваемый по умолчанию уровень изоляции `READ COMMITTED`. T1, T2 и т. д. обозначают время в порядке возрастания.

	Сеанс 1	**Сеанс 2**

T1	`postgres=# BEGIN;`
	`BEGIN`
	`postgres=# SELECT * FROM test_tx_level ;`
	` val`
	`-----`
	`   1`
	`(1 row)`	

T2		`postgres=# BEGIN;`
		`BEGIN`
		`postgres=# UPDATE`
		`test_tx_level SET val = 2;`
		`UPDATE 1`
		`postgres=# COMMIT;`
		`COMMIT`

T3	`postgres=# SELECT * FROM test_tx_level ;`
	` val`
	`-----`
	`   2`
	`(1 row)`
	`postgres=# COMMIT;`
	`COMMIT`	

В сеансе 2 значение `val` изменилось с 1 на 2. Изменение, зафиксированное в сеансе 2, отражено в сеансе 1. Мы имеем пример неповторяемого чтения. Отметим также, что после `BEGIN` не был указан уровень изоляции, поэтому применен уровень, подразумеваемый по умолчанию.

Чтобы продемонстрировать фантомное чтение, снова воспользуемся уровнем изоляции по умолчанию.

	Сеанс 1	**Сеанс 2**

T1	`postgres=# BEGIN;`
	`BEGIN`
	`postgres=# SELECT count(*) FROM`
	`test_tx_level ;`
	` count`
	`-------`
	`     1`
	`(1 row)`	

T2		`postgres=# BEGIN;`
		`BEGIN`
		`postgres=# INSERT INTO`
		`test_tx_level SELECT 2;`
		`INSERT 0 1`
		`postgres=# COMMIT;`
		`COMMIT`

	Сеанс 1	Сеанс 2
T3	postgres=# SELECT count(*) FROM test_tx_level ; count ------- 2 (1 row) postgres=# COMMIT;	

Как видно из примеров, фантомное и неповторяемое чтение могут встретиться на уровне изоляции READ COMMITTED. Неповторяемое чтение обычно распространяется на строку или множество строк, а фантомное – на всю таблицу. Неповторяемое чтение является результатом зафиксированного в другой транзакции **обновления**, а фантомное – результатом зафиксированной вставки или удаления.

Если выполнить те же примеры с уровнем изоляции SERIALIZABLE или REPEATABLE READ, то мы увидим, что сеанс 2 не влияет на результат в сеансе 1:

	Сеанс 1	Сеанс 2
T1	BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE ; BEGIN postgres=# SELECT count(*) FROM test_tx_level ; count ------- 2 (1 row)	

T2		postgres=# BEGIN;
		BEGIN
		postgres=# INSERT INTO
		test_tx_level SELECT 2;
		INSERT 0 1
		postgres=# COMMIT;
		COMMIT

T3	postgres=# SELECT count(*) FROM test_tx_level ; count ------- 2 (1 row) postgres=# COMMIT;	

До версии 9.1 в PostgreSQL было только два уровня изоляции транзакций. В версии 9.1 был добавлен уровень SERIALIZABLE. Уровень изоляции SERIALIZABLE защищает от многих аномалий, в т. ч. от искажения записи. Искажение записи имеет место, когда две транзакции читают перекрывающиеся данные, одновременно производят обновление, а затем фиксируют изменения. Предположим, к примеру, что таблица содержит нули и единицы. Первая транзакция хочет заменить все единицы нулями, а вторая – все нули единицами. Если

транзакции выполняются последовательно, то мы получим либо все нули, либо все единицы в зависимости от того, какая транзакция выполнялась первой. Чтобы продемонстрировать эту аномалию, создадим и заполним таблицу:

```
postgres=# CREATE TABLE zero_or_one (val int);
CREATE TABLE
postgres=# INSERT INTO zero_or_one
SELECT n % 2 FROM generate_series(1,10) as foo(n) ;
INSERT 0 10
postgres=# SELECT array_agg(val) FROM zero_or_one ;
      array_agg
----------------------
 {1,0,1,0,1,0,1,0,1,0}
(1 row)
```

Чтобы наблюдать эффект искажения записи, запустим два сеанса с уровнем изоляции REPEATABLE READ:

	Сеанс 1	**Сеанс 2**

T1	postgres=# BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ ;
	BEGIN
	postgres=# UPDATE zero_or_one SET val = 1 WHERE val = 0;
	UPDATE 5	

T2		postgres=# BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ ;
		BEGIN
		postgres=# UPDATE zero_or_one SET val =0 WHERE val =1;
		UPDATE 5
		postgres=# COMMIT;
		COMMIT

T3	postgres=# COMMIT; COMMIT	

Посмотрим, что получилось в итоге:

```
postgres=# SELECT array_agg(val) FROM zero_or_one ;
      array_agg
----------------------
{1,1,1,1,1,0,0,0,0,0}
(1 row)
```

Чтобы узнать, что происходит в режиме SERIALIZABLE, усечем таблицы и выполним пример еще раз:

```
postgres=# truncate zero_or_one ;
TRUNCATE TABLE
postgres=# INSERT INTO zero_or_one SELECT n % 2 FROM generate_series(1,10)
as foo(n) ;
INSERT 0 10
```

	Сеанс 1	Сеанс 2

T1	postgres=# BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE ;
	BEGIN
	postgres=# UPDATE zero_or_one SET val = 1 WHERE val = 0;
	UPDATE 5	

T2		postgres=# BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE ;
		BEGIN
		postgres=# UPDATE zero_or_one SET val =0 WHERE val =1;
		UPDATE 5
		postgres=# COMMIT;

T3	postgres=# COMMIT ; ERROR: could not serialize access due to read/write dependencies among transactions DETAIL: Reason code: Canceled on identification as a pivot, during commit attempt. HINT: The transaction might succeed if retried.	

В режиме REPEATABLE READ обе транзакции выполнились без ошибок, но конечный результат оказался неверным. В режиме SERIALIZABLE при наличии искажения записи транзакции продолжаются, пока одна из них не попытается зафиксировать изменения. Первая транзакция, предпринявшая такую попытку, завершится успешно, остальные будут откачены. Правило «первый выигрывает» гарантирует, что мы хоть куда-то продвинемся. И еще раз подчеркнем, что успешно завершается только одна транзакция, остальных ждет печальная участь. Обратите также внимание на подсказку «the transaction might succeed if retried» (транзакция может завершиться успешно при следующей попытке). В итоге же получился такой результат:

```
postgres=# SELECT array_agg(val) FROM zero_or_one ;
       array_agg
-----------------------
 {0,0,0,0,0,0,0,0,0,0}
(1 row)
```

Дополнительные сведения об уровнях изоляции REPEATABLE READ и SERIALIZABLE см. на вики-странице о **сериализуемой изоляции методом мгновенного снимка** (serializable snapshot isolation – **SSI**) по адресу https://wiki.postgresql.org/wiki/SSI.

Явная блокировка

Если MVCC-блокировка не обеспечивает нужного поведения, то можно прибегнуть к явному управлению блокировкой. PostgreSQL предоставляет три механизма блокировки:

- блокировка на уровне таблиц;
- блокировка на уровне строк;
- рекомендательная блокировка.

Блокировка на уровне таблиц

Существует несколько режимов блокировки таблиц. Синтаксис команды LOCK имеет вид:

```
LOCK [ TABLE ] [ ONLY ] name [ * ] [, ...] [ IN lockmode MODE ] [ NOWAIT ]
```

где lockmode может принимать следующие значения:

```
ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE
| SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE
```

При выполнении любой команды PostgreSQL неявно блокирует таблицу. А чтобы не мешать конкурентному выполнению, блокировка производится в наименее ограничительном режиме. Если же программисту нужна более ограничительная блокировка, то он может воспользоваться командой LOCK.

Режимы табличной блокировки

Табличная блокировка часто ставится автоматически, но ее можно поставить и явно командой LOCK. Ниже перечислены все режимы блокировки:

- ACCESS SHARE: этот режим устанавливается командой SELECT;
- ROW SHARE: этот режим устанавливается командами SELECT FOR UPDATE и SELECT FOR SHARE;
- ROW EXCLUSIVE: этот режим устанавливается командой UPDATE, DELETE и INSERT;
- SHARE UPDATE EXCLUSIVE: этот режим служит для защиты таблицы от конкурентного изменения схемы. Устанавливается командами VACUUM (без FULL), ANALYZE, CREATE INDEX CONCURRENTLY, CREATE STATISTICS, ALTER TABLE VALIDATE и другими вариантами команды ALTER TABLE;
- SHARE: этот режим служит для защиты таблицы от конкурентного изменения данных. Устанавливается командой CREATE INDEX (без CONCURRENTLY);
- SHARE ROW EXCLUSIVE: этот режим защищает таблицу от конкурентного изменения данных и является монопольным, т. е. в каждый момент времени блокировкой может владеть только один сеанс. Устанавливается командами CREATE COLLATION, CREATE TRIGGER и различными вариантами команды ALTER TABLE;
- EXCLUSIVE: устанавливается командой REFRESH MATERIALIZED VIEW CONCURRENTLY. Разрешает только читать данные из таблицы;
- ACCESS EXCLUSIVE: в этом режиме гарантируется, что обращаться к таблице любым способом может только владелец блокировки. Он устанавливается командами DROP TABLE, TRUNCATE, REINDEX, CLUSTER, VACUUM FULL и REFRESH MATERIALIZED VIEW (без CONCURRENTLY), а также различными вариантами команды ALTER TABLE. Этот режим по умолчанию устанавливается командой LOCK TABLE, если явно не указан иной режим.

Для каждого режима важно знать, какие режимы с ним конфликтуют. Никакие две транзакции не могут поставить на одну таблицу конфликтующие блокировки.

Conflicting?	ACCESS SHARE	ROW SHARE	ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE	SHARE	SHARE ROW EXCLUSIVE	EXCLUSIVE	ACCESS EXCLUSIVE
ACCESS SHARE								yes
ROW SHARE							yes	yes
ROW EXCLUSIVE					yes	yes	yes	yes
SHARE UPDATE EXCLUSIVE				yes	yes	yes	yes	yes
SHARE			yes	yes	yes	yes	yes	yes
SHARE ROW EXCLUSIVE			yes	yes	yes	yes	yes	yes
EXCLUSIVE		yes	yes	yes	yes	yes	yes	yes
ACCESS EXCLUSIVE	yes	yes	yes	yes	yes	yes	yes	yes

Как видно из этой таблицы, режим ACCESS EXCLUSIVE конфликтует с ACCESS SHARE, а это значит, что невозможно выполнить команду SELECT для таблицы, заблокированной в режиме ACCESS EXCLUSIVE. И наоборот, нельзя удалить таблицу командой DROP, если ее кто-то читает. Отметим также, что во всех остальных режимах чтение командой SELECT возможно, т. е. запретить SELECT можно только в режиме ACCESS EXCLUSIVE.

В следующем примере показано, что произойдет при попытке удалить таблицу, когда другая транзакция читает ее.

	Сеанс 1	**Сеанс 2**

T1	BEGIN
	postgres=# SELECT COUNT(*) FROM test_tx_level ;
	 count
	------- 	
	3
	(1 row)
	postgres=# SELECT mode, granted FROM pg_locks WHERE
	relation ='test_tx_level'::regclass::oid;
	 mode
		granted
		-----------------+---------
		 AccessShareLock
			t
			(1 row)	

T2		BEGIN postgres=# DROP TABLE test_tx_level;
T3	postgres=# SELECT mode, granted FROM pg_locks where relation ='test_tx_level'::regclass::oid; mode	granted --------------------+--------- AccessShareLock	t AccessExclusiveLock	f (2 rows)	

Мы видим, что блокировка поставлена, когда данные выбирались из таблицы в сеансе 1, и затем не снята. Коль скоро блокировка поставлена, она обычно удерживается до конца транзакции. Для наблюдения за блокировками очень

полезна таблица pg_locks. Зачастую она используется для обнаружения узких мест в системах с высокой конкурентностью. Следующее представление показывает информацию о блокировках в более понятном виде:

```
CREATE OR REPLACE VIEW lock_info AS
SELECT
    lock1.pid as locked_pid,
    stat1.usename as locked_user,
    stat1.query as locked_statement,
    stat1.state as locked_statement_state,
    stat2.query as locking_statement,
    stat2.state as locking_statement_state,
    now() - stat1.query_start as locking_duration,
    lock2.pid as locking_pid,
    stat2.usename as locking_user
FROM pg_catalog.pg_locks lock1
    JOIN pg_catalog.pg_stat_activity stat1 on lock1.pid = stat1.pid
    JOIN pg_catalog.pg_locks lock2 on
(lock1.locktype,lock1.database,lock1.relation,lock1.page,lock1.tuple,lock1.
virtualxid,lock1.transactionid,lock1.classid,lock1.objid,lock1.objsubid) IS
NOT DISTINCT FROM
(lock2.locktype,lock2.DATABASE,lock2.relation,lock2.page,lock2.tuple,lock2.
virtualxid,lock2.transactionid,lock2.classid,lock2.objid,lock2.objsubid)
    JOIN pg_catalog.pg_stat_activity stat2 on lock2.pid = stat2.pid
WHERE NOT lock1.granted AND lock2.granted;
Чтобы посмотреть на блокировки, выполним запрос:
postgres=# SELECT * FROM lock_info ;
-[ RECORD 1 ]-----+-----------------------------------------------------
--------------------------
 locked_pid              | 3736
 locked_user             | postgres
 locked_statement        | DROP TABLE test_tx_level;
 locked_statement_state  | active
 locking_statement       | SELECT mode, granted FROM pg_locks where relation
='test_tx_level'::regclass::oid;
 locking_statement_state | idle in transaction
 locking_duration        | 00:09:57.32628
 locking_pid             | 3695
 locking_user            | postgres
```

Как видим, процесс 3736 пытается выполнить команду DROP TABLE и ждет завершения транзакции, начатой процессом 3695. Поскольку процесс 3695 ничего не делает во время работы предыдущего запроса, его состояние idle in transaction. Отметим, что это представление немного сбивает с толку, поскольку говорит, что блокировка поставлена командой SELECT mode, granted ..;, хотя это не так. Просто в таблице pg_state_activity находится последняя команда, выполненная процессом. Как уже было сказано, захваченная блокировка удерживается до конца транзакции. Поэтому процесс 3736 будет висеть, пока не закончится транзакция, начатая процессом 3695.

Блокировка на уровне строк

Блокировка на уровне строк вообще не блокирует команды SELECT, а служит для блокировки команд UPDATE и DELETE. Как и в случае блокировки на уровне таблиц, никакие две транзакции не могут поставить конфликтующие блокировки на одну строку. Блокировка строк полезна, когда приложение хочет сначала просмотреть значение строки, а затем обновить его. Другое применение – помешать пользователям обновить старое значение; например, если один пользователь редактирует документ, то всем остальным редактирование запрещено. В версии PostgreSQL 9.5 появился режим SKIP LOCKED, изменяющий поведение блокировки на уровне строк. Он полезен, когда нужно произвести пакетную обработку, не заставляя другие процессы долго ждать, а также при обработке очередей и пулов на стороне базы данных.

Режимы блокировки на уровне строк

В старых версиях, например в PostgreSQL 9.3, было всего два режима блокировки: FOR UPDATE и FOR SHARE. В режиме FOR UPDATE ставится монопольная блокировка на строку, при этом всем остальным транзакциям запрещено обновлять или удалять ее. Режим FOR SHARE не столь ограничительный, в нем другим транзакциям разрешено ставить на строку блокировку типа FOR SHARE, но по-прежнему запрещено удалять или обновлять строку. Режим FOR SHARE используется для предотвращения неповторяемого чтения на уровне изоляции READ COMMITTED. Например, если заблокировать строку в режиме FOR SHARE, то она гарантированно не изменится до фиксации транзакции.

Режим FOR SHARE не решает проблему потерянного обновления, поскольку разрешает другим транзакциям блокировать строку в том же режиме. Чтобы все-таки решить эту проблему, нужен режим FOR UPDATE.

Потерянное обновление имеет место, когда две транзакции пытаются одновременно обновить одну и ту же строку. Подробнее об этом см. в разделе, посвященном взаимоблокировкам.

PostgreSQL предлагает также режим блокировки FOR NO KEY UPDATE, похожий на FOR UPDATE, но более слабый, и режим FOR KEY SHARE, являющийся более слабой формой FOR SHARE. Конфликты между режимами блокировки на уровне строк приведены в следующей таблице.

Conflicting?	FOR KEY SHARE	FOR SHARE	FOR NO KEY UPDATE	FOR UPDATE
FOR KEY SHARE				yes
FOR SHARE			yes	yes
FOR NO KEY UPDATE		yes	yes	yes
FOR UPDATE	yes	yes	yes	yes

Чтобы протестировать блокировку на уровне строк, усечем таблицу test_tx_level и добавим в нее новую строку:

```
postgres=# truncate test_tx_level ;
TRUNCATE TABLE
postgres=# insert into test_tx_level Values(1), (2);
INSERT 0 2
```

а затем выполним такие команды:

	Сеанс 1	**Сеанс 2**

T1	postgres=# BEGIN;
	BEGIN
	postgres=# SELECT * FROM
	test_tx_level WHERE val = 1 FOR
	UPDATE;
	val
	----- 	
	1
	(1 row)	

T2		postgres=# BEGIN;
		BEGIN
		postgres=# update test_tx_level SET
		val =2 WHERE val =1;

Сеанс 2 ждет сеанса 1, поскольку тот захватил блокировку типа FOR UPDATE.

```
postgres=# SELECT * FROM lock_info ;
-[ RECORD 1 ]-----+-----------------------------------------------------
 locked_pid        | 3368
 locked_user       | postgres
 locked_statement  | update test_tx_level SET val =2 WHERE val =1;
 state             | active
 locking_statement | SELECT * FROM test_tx_level WHERE val = 1 FOR update;
 state             | idle in transaction
 locking_duration  | 00:04:04.631108
 locking_pid       | 3380
 locking_user      | postgres
```

Взаимоблокировки

Взаимоблокировка имеет место, когда каждая из двух или более транзакций удерживает блокировку, необходимую другой транзакции. Использование явных блокировок повышает вероятность возникновения взаимоблокировок. Чтобы смоделировать взаимоблокировку, воспользуемся блокировкой на уровне строк в режиме FOR SHARE:

	Сеанс 1	Сеанс 2

T1	postgres=# BEGIN;
	BEGIN
	postgres=# SELECT * FROM
	test_tx_level WHERE val = 1 FOR
	SHARE;
	val
	----- 	
	1
	(1 row)	

T2		postgres=# begin;
		BEGIN
		postgres=# SELECT * FROM test_tx_level
		WHERE val = 1 FOR SHARE;
		val
		----- 	
		1
		(1 row)

T3	postgres=# UPDATE test_tx_level SET val = 2 WHERE val=1;	
T4		postgres=# UPDATE test_tx_level SET val = 2 WHERE val=1; ERROR: deadlock detected DETAIL: Process 3368 waits for ExclusiveLock on tuple (0,1) of relation 139530 of database 13014; blocked by process 3380. Process 3380 waits for ShareLock on transaction 121862; blocked by process 3368. HINT: See server log for query details.
T5	UPDATE 1	

Чтобы избежать взаимоблокировок, первая захваченная в транзакции блокировка должна быть наиболее ограничительной. Так, если бы в сеансе 1 блокировка была поставлена в режиме FOR UPDATE, то сеанс 2 оказался бы заблокирован, и ошибка из-за взаимоблокировки не произошла бы.

Используя более ограничительный режим, мы заставляем другие транзакции ждать освобождения блокировки. Но если блокирующая транзакция остается открытой долго, то в приложении будут наблюдаться длительные задержки.

Рекомендательные блокировки

Рекомендательная блокировка (advisory lock) ставится приложением и служит для моделирования пессимистических стратегий блокировки. Она захватывается на уровне сеанса или транзакции и освобождается в конце сеанса или после фиксации транзакции.

Рекомендательные блокировки можно использовать для ограничения конкурентности одним процессом. Например, в начале работы процесс пытается захватить блокировку; если это получается, то он продолжает работать, ина-

че завершается. Рекомендательные блокировки позволяют разработчику рассматривать СУБД как однопользовательскую среду и забыть обо всех сложностях системы блокировок.

Рекомендательные блокировки хранятся в памяти, поэтому работать с ними нужно аккуратно, чтобы не исчерпать ресурсы кластера. Полный перечень рекомендательных блокировок см. на странице https://www.postgresql.org/docs/current/static/functions-admin.html#functionsadvisory-locks-table.

Сеанс 1	Сеанс 2
postgres=# SELECT pg_try_advisory_lock(1); pg_try_advisory_lock ---------- t (1 row)	
	postgres=# SELECT pg_try_advisory_lock(1); pg_try_advisory_lock ---------- f (1 row)
postgres=# select pg_advisory_unlock(1); pg_advisory_unlock ---------- t (1 row)	
	postgres=# SELECT pg_try_advisory_lock(1); pg_try_advisory_lock ---------- t (1 row)

В одном сеансе можно захватывать одну и ту же рекомендательную блокировку несколько раз. Но освобождать ее нужно столько же раз, сколько было захватов. Например:

```
SELECT pg_try_advisory_lock(1);
SELECT pg_try_advisory_lock(1);
-- Освобождаем
select pg_advisory_unlock(1);
select pg_advisory_unlock(1);
```

Резюме

PostgreSQL предлагает несколько механизмов блокировки для повышения уровня конкурентности и производительности. Среди них неявная блокировка средствами MVCC и явная блокировка: на уровне таблиц, на уровне строк и рекомендательная.

Модель MVCC – одно из самых важных рыночных преимуществ PostgreSQL, поскольку позволяет достичь высокой производительности. Вообще говоря, MVCC пригодна в большинстве распространенных сценариев работы с базой данных, и по возможности лучше использовать ее, чем явные блокировки.

Явные блокировки на уровне таблиц и строк позволяют разрешить несколько проблем несогласованности данных. Однако при недостаточно тщательном планировании повышается вероятность возникновения взаимоблокировок. Наконец, PostgreSQL предлагает рекомендательные блокировки, позволяющие моделировать пессимистические стратегии блокировки.

В следующей главе мы рассмотрим аутентификацию и авторизацию. Будут описаны методы аутентификации в PostgreSQL и объяснена структура конфигурационного файла аутентификации на уровне узлов. Мы также обсудим разрешения на создание различных объектов базы данных: схем, таблиц, представлений, индексов и столбцов. Наконец, мы покажем, как защитить секретные данные, например пароли, с помощью одностороннего и двустороннего шифрования.

Глава 11

Безопасность в PostgreSQL

Защита данных и безопасность обязательны для обеспечения непрерывности бизнеса. Защищать данные – не сахар, но этого требует законодательство. Закрытые данные – информация о пользователях, адреса электронной почты, географические адреса, сведения о платежах – должны быть защищены от любых взломов. У безопасности данных несколько аспектов, как то: конфиденциальность, сроки хранения и предотвращение утраты.

Существует несколько уровней защиты данных, зачастую они прописываются в политике защиты данных и в законах страны. Политика защиты данных обычно определяет правила передачи данных третьим сторонам, пользователей, имеющих право доступа к данным, и т. п. Данные следует защищать на разных уровнях, в т. ч. при передаче по каналам связи и при хранении (в зашифрованном виде). Безопасность данных – обширная тема, нередко для ее обеспечения создается специальное подразделение.

В этой главе рассматриваются следующие вопросы:

- аутентификация в PostgreSQL, включая аутентификацию по узлам и рекомендации;
- привилегии доступа по умолчанию и привилегии в схеме public;
- стратегия прокси-аутентификации;
- уровни безопасности в PostgreSQL, включая привилегии доступа к базе данных, схеме, таблице, столбцу и строке;
- шифрование и дешифрирование данных.

Аутентификация в PostgreSQL

Аутентификация отвечает на вопрос: кем является пользователь? PostgreSQL поддерживает несколько методов аутентификации:

- **trust:** любой подключившийся к серверу имеет право обращаться к базе (или к базам) данных в соответствии с тем, что прописано в конфигурационном файле `pg_hba.conf`. Часто используется, чтобы разрешить доступ к базе при подключении через Unix-сокет на однопользовательской ма-

шине. Этот метод работает и при подключении по протоколу TCP/IP, но редко бывает так, чтобы доступ был открыт для какого-нибудь IP-адреса, кроме localhost;

- **ident:** имя пользователя в клиентской операционной системе запрашивается у сервера ident, а затем используется для доступа к базе данных. Рекомендуется применять в закрытых сетях, где все клиентские машины находятся под полным контролем системного администратора;
- **peer:** метод аналогичен ident, но имя пользователя в клиентской операционной системе запрашивается у ядра;
- **GSSAPI:** этот отраслевой стандарт, определенный в RFC 2743, обеспечивает автоматическую аутентификацию (единая точка входа);
- **LDAP:** протокол LDAP (Lightweight Directory Access Protocol) используется только для проверки пар имя-пароль;
- **аутентификация по паролю:** реализовано три метода:
  - **scram-sha-256:** самый стойкий метод аутентификации по паролю, добавленный в PostgreSQL 10. Предотвращает прослушивание пароля при передаче по ненадежным соединениям;
  - **md5:** второй по стойкости метод, но для новых приложений рекомендуется scram-sha-256. PostgreSQL предлагает также средство миграции с md5 на scram-sha-256;
  - **password:** не рекомендуется, т. к. пароль передается серверу в открытом виде.

Существуют и другие методы аутентификации, полный перечень см. на странице https://www.postgresql.org/docs/current/static/authmethods.html.

Чтобы разобраться в аутентификации, надо знать следующее:

- аутентификация управляется файлом `pg_hba.conf`, где **hba** означает **host-based authentication** (аутентификация по узлам);
- следует знать, какие начальные параметры аутентификации установлены в дистрибутиве PostgreSQL;
- файл `pg_hba.conf` обычно находится в одном каталоге с данными, но его местонахождение можно задать в файле `postgresql.conf`;
- при изменении параметров аутентификации необходимо послать серверу сигнал **SIGHUP**, это делается различными способами в зависимости от платформы, на которой работает PostgreSQL. Отметим, что посылать сигнал имеет право системный пользователь `postgres` или `root` (на платформе Linux). Ниже показано несколько способов перезагрузить конфигурацию PostgreSQL:

```
psql -U postgres -c "SELECT pg_reload_conf();"
sudo service postgresql reload
sudo /etc/init.d/postgresql reload
sudo Kill -HUP <postgres process id>
```

- порядок записей в файле `pg_hba.conf` имеет значение. Данные, переданные при установлении сеанса, сравниваются с записями в `pg_hba.conf` последовательно, пока не будут отвергнуты или не встретится конец файла;

- наконец, важно просматривать журналы PostgreSQL, в которых отражаются ошибки, обнаруженные после перезагрузки конфигурационного файла.

Файл pg_hba.conf

Как и файл `postgresql.conf`, файл `pg_hba.conf` состоит из записей. Строки могут быть закомментированы знаком `#`, пробелы игнорируются. Каждая запись имеет следующую структуру:

```
host_type database user [IP-address | address] [IP-mask] auth-method [authoptions]
```

Поле `host_type` может принимать следующие значения:

- `Local`: используется в Linux-системах, чтобы разрешить доступ к PostgreSQL через Unix-сокет;
- `Host`: позволяет подключаться с других узлов, заданных доменным именем или IP-адресом, по протоколу TCP/IP с SSL-шифрованием или без;
- `Hostssl`: аналогично `host`, но соединение должно быть зашифровано по SSL;
- `Hostnossl`: аналогично `host`, но соединение не должно быть зашифровано.

Поле `database` содержит имя базы данных, к которой подключается пользователь. Разрешается перечислить через запятую несколько имен или задать слово `all`, означающее, что пользователю разрешен доступ к любой базе данных в кластере. Кроме того, значения `sameuser` и `samerole` означают, что имя базы данных должно совпадать с именем пользователя или с именем роли, членом которой является пользователь.

Поле `user` задает имя пользователя базы данных. Значение `all` разрешает доступ всем пользователям.

Поля `IP-address`, `address` и `IP-mask` определяют узлы, с которых разрешено подключаться. IP-адрес можно задавать в формате **CIDR** (Classless Inter-Domain Routing – бесклассовая междоменная маршрутизация) или в точечно-десятичной нотации.

Наконец, поле `auth-method` может принимать значения `trust`, `MD5`, `reject` и другие.

Ниже приведены типичные примеры конфигурирования аутентификации в PostgreSQL.

- **Пример 1:** любой пользователь кластера PostgreSQL может обращаться к любой базе данных через Unix-сокет:

```
#TYPE   DATABASE   USER   ADDRESS   METHOD
Local   all        all              trust
```

- **Пример 2:** любой пользователь кластера PostgreSQL может обращаться к любой базе данных через закольцованный IP-адрес:

```
#TYPE   DATABASE   USER   ADDRESS        METHOD
Host    all        all    127.0.0.1/32   trust
host    all        all    ::1/128        trust
```

- **Пример 3:** все подключения с адреса 92.168.0.53 отвергаются, а подключения с узлов из диапазона адресов 192.168.0.1/24 принимаются:

```
#TYPE   DATABASE   USER   ADDRESS          METHOD
Host    all        all    92.168.0.53/32   reject
Host    all        all    92.168.0.1/24    trust
```

PostgreSQL предлагает очень удобный способ просмотреть правила, определенные в файле `pg_hba.conf`, – через представление `pg_hba_file_rules`:

```
postgres=# SELECT * FROM pg_hba_file_rules limit 1;
 line_number | type  | database | user_name  | address | netmask |
auth_method | options | error
-------------+-------+----------+------------+---------+---------+-
------------+---------+-------
          85 | local | {all}    | {postgres} |         |         |
 peer        |         |
(1 row)
```

Прослушиваемые адреса

Прослушиваемые адреса определены в параметре `listen_addresses` в файле `postgresql-conf`. Этот параметр представляет собой список адресов, по которым сервер будет принимать подключения клиентов. Адреса – доменные имена или IP-адреса – перечисляются через запятую. После изменения этого параметра сервер необходимо перезагрузить. Отметим также, что:

- по умолчанию подразумевается единственный адрес localhost;
- пустой список означает, что к серверу разрешено подключаться только через Unix-сокет;
- значение * означает «любой».

Начинающие пользователи PostgreSQL часто забывают изменить `listen_addresses`.

Рекомендации по аутентификации

Как организовывать аутентификацию, зависит от настройки инфраструктуры в целом, от характера приложения, характеристик пользователей, уровня секретности данных и т. д. Многие стартапы начинают с такой конфигурации: приложение базы данных размещается на одной машине с сервером и используется только с одного физического компьютера сотрудниками компании.

Часто сервер базы данных изолируется от внешнего мира брандмауэром; в таком случае можно использовать метод аутентификации **scram-sha-256** и ограничить диапазон IP-адресов, с которых сервер принимает запросы на подключение. Отметим, что важно не подключаться к базе от имени суперпользователя или владельца базы данных, потому что если соответствующая учетная запись будет скомпрометирована, то под угрозой окажется весь кластер.

Если сервер приложений, на котором размещена бизнес-логика, и сервер баз данных находятся на разных машинах, то можно использовать стойкий метод аутентификации, например **LDAP** и **Kerberos**. Но для небольших приложений, в которых приложение и сервер базы располагаются на одной машине, метода аутентификации **scram-sha-256** и ограничения множества адресов подключения может оказаться достаточно.

Для аутентификации приложения рекомендуется использовать только одного пользователя, стараясь при этом уменьшить максимальное число допустимых подключений с помощью программ организации пулов подключений, чтобы более эффективно использовать ресурсы PostgreSQL. Еще один уровень безопасности может понадобиться, когда бизнес-логика приложения зависит от категории вошедшего пользователя. Для реальных пользователей предпочтительная аутентификация через LDAP или Kerberos.

Наконец, если к серверу баз данных обращаются из внешнего мира, то имеет смысл шифровать сеансы с помощью SSL-сертификатов, чтобы предотвратить перехват пакетов.

Не забывайте защищать серверы, которые доверяют всем локальным подключениям, потому что к такому серверу может подключиться любой, имеющий доступ к localhost.

Привилегии доступа по умолчанию

По умолчанию пользователи PostgreSQL – они же роли с возможностью входа – имеют доступ к схеме public. Отметим также, что на платформе Linux по умолчанию действует одноранговая политика аутентификации, разрешающая доступ ко всем базам с localhost. Пользователь может создавать объекты базы данных – таблицы, представления, функции и т. д. – в схеме public любой базы данных, к которой у него есть доступ. Наконец, пользователь может изменять некоторые параметры сеансов, например `work_mem`.

Пользователь не может обращаться к объектам других пользователей в схеме public, а также создавать новые базы данных и схемы. Однако он может узнать о существующих объектах базы данных, опросив системный каталог. Непривилегированный пользователь может получить информацию о других пользователях, структуре и владельце таблиц, некоторую статистику таблиц и др.

В примере ниже показано, как пользователь `test_user` может получить информацию о таблице, принадлежащей пользователю `postgres`; чтобы смоделировать эту ситуацию, создадим тестовую базу данных:

```
psql -U postgres -c 'CREATE ROLE test_user LOGIN;';
psql -U postgres -c 'CREATE DATABASE test;';
psql -U postgres -d test -c'CREATE TABLE test_permissions(id serial, name text);'
```

У пользователя `test_user` нет разрешения на доступ к самой таблице, но он может обращаться к системному каталогу. Чтобы убедиться в этом, подключимся к базе данных от имени `test_user`:

```
test=# SET ROLE test_user;
SET
test=> \d
                    List of relations
 Schema | Name                     | Type     | Owner
--------+--------------------------+----------+----------
 public | test_permissions         | table    | postgres
 public | test_permissions_id_seq | sequence | postgres
(2 rows)
test=> \du
                            List of roles
 Role name | Attributes                                                  | Member of
-----------+-------------------------------------------------------------+-----------
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
test_user | | {}
```

Пользователь может также вызывать функции, созданные другими пользователями в схеме public, при условии что они не обращаются к объектам, для которых у этого пользователя нет доступа.

Никто, кроме суперпользователя, не может создавать функции на ненадежных языках, в частности **plpythonu**. При попытке любого другого пользователя создать функцию на языке C или **plpythonu** будет напечатано сообщение об ошибке.

Чтобы запретить пользователю доступ к схеме public, можно отозвать соответствующие привилегии:

```
test=# SELECT session_user;
 session_user
--------------
 postgres
(1 row)

test=# REVOKE ALL PRIVILEGES ON SCHEMA PUBLIC FROM public;
REVOKE

test=# SET ROLE test_user;
SET

test=> CREATE TABLE b();
ERROR: no schema has been selected to create in
LINE 1: create table b();
```

У пользователя test_user имеются явные привилегии на доступ к схеме public; пользователь наследует их от роли public.

Владелец представления не сможет запросить через него данные, если не имеет прав доступа к соответствующим базовым таблицам.

Система ролей и прокси-аутентификация

Часто при проектировании приложения пользователь конфигурирует подключения к базе данных и средства подключения. Необходим и еще один уровень

безопасности: проверять, что каждый пользователь имеет право выполнять запрошенную операцию. Эта логика нередко встраивается в бизнес-логику приложения. Но для ее частичной реализации можно воспользоваться системой ролей в базе данных, делегировав авторизацию другой роли, после того как подключение установлено или использовано повторно. Для этого используется команда SET SESSION AUTHORIZATION или команда SET ROLE в блоке транзакции:

```
postgres=# SELECT session_user, current_user;
 session_user | current_user
--------------+--------------
 postgres     | postgres
(1 row)

postgres=# SET SESSION AUTHORIZATION test_user;
SET

postgres=> SELECT session_user, current_user;
 session_user | current_user
--------------+--------------
 test_user    | test_user
(1 row)
```

Для команды SET ROLE требуется быть членом роли, а для команды SET SESSION AUTHORIZATION нужны привилегии суперпользователя. Разрешать приложению подключаться от имени суперпользователя опасно, потому что действие команд SET SESSION AUTHORIZATION и SET ROLE можно отменить командами RESET ROLE и RESET SESSION соответственно, в результате чего приложение получит привилегии суперпользователя.

Чтобы понять, как использовать систему ролей PostgreSQL для реализации аутентификации и авторизации, обратимся к приложению для сайта торговли автомобилями, в котором можно выделить несколько групп пользователей: web_app_user, public_user, registered_user, seller_user и admin_user. Группа web_app_user используется для конфигурирования средств подключения к бизнес-логике, а группы public_user, registered_user и seller_user – чтобы различать различные категории пользователей. Группа public_user может обращаться только к открытой информации, например объявлениям, но не вправе ни ставить оценки, как registered_user, ни размещать объявления, как seller_user. Группа admin_user служит для управления всем контентом приложения, в т. ч. имеет право фильтровать спам и удалять пользователей, не соблюдающих правила сайта. Сразу после подключения к базе приложение car_portal имеет права web_app_user. Затем оно выполняет команду SET ROLE в соответствии с категорией пользователя. Такой способ называется **прокси-аутентификацией**.

Ниже приведены примеры использования системы ролей для реализации прокси-аутентификации. Первый шаг – создать роли, включить в них пользователей и назначить привилегии:

```
CREATE ROLE web_app_user LOGIN NOINHERIT;
CREATE ROLE public_user NOLOGIN;
```

```
GRANT SELECT ON car_portal_app.advertisement_picture,
  car_portal_app.advertisement_rating , car_portal_app.advertisement TO public_user;
GRANT public_user TO web_app_user;
GRANT USAGE ON SCHEMA car_portal_app TO web_app_user, public_user;
```

Слово NOINHERIT для роли web_app_user не позволяет наследовать разрешения от объемлющей роли, однако web_app_user может изменить роль на public_user, как в следующем примере:

```
$ psql car_portal -U web_app_user
psql (10.0)
Type "help" for help.

car_portal=> SELECT * FROM car_portal_app.advertisement;
ERROR: permission denied for relation advertisement

car_portal=> SET ROLE public_user;
SET
car_portal=> SELECT * FROM car_portal_app.advertisement;
 advertisement_id | advertisement_date | car_id | seller_account_id
------------------+--------------------+--------+-------------------
(0 rows)

car_portal=> SELECT session_user, current_user;
 session_user | current_user
--------------+--------------
 web_app_user | public_user
(1 row)
```

Уровни безопасности в PostgreSQL

В PostgreSQL определены различные уровни безопасности для объектов: табличного пространства, базы данных, схемы, таблицы, адаптера внешних данных, последовательности, домена, языка и большого объекта. О том, как задавать различные привилегии, расскажет метакоманда \h в psql:

```
Command: GRANT
Description: define access privileges
Syntax:
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] table_name [, ...]
         | ALL TABLES IN SCHEMA schema_name [, ...] }
    TO role_specification [, ...] [ WITH GRANT OPTION ]
```

Безопасность на уровне базы данных

Чтобы запретить всем пользователям подключаться к базе данных, нужно отозвать все разрешения у роли public:

```
$ psql car_portal -U postgres
psql (10.0)
Type "help" for help.
```

```
car_portal=# REVOKE ALL ON DATABASE car_portal FROM public;
REVOKE
car_portal=# \q
$ psql car_portal -U web_app_user
psql: FATAL: permission denied for database "car_portal"
DETAIL: User does not have CONNECT privilege.
```

Чтобы разрешить конкретному пользователю подключаться к базе, следует предоставить разрешение:

```
postgres=# GRANT CONNECT ON DATABASE car_portal TO web_app_user;
GRANT
postgres=# \l car_portal
                    List of databases
-[ RECORD 1 ]-----+---------------------------------
Name              | car_portal
Owner             | car_portal_app
Encoding          | UTF8
Collate           | en_US.UTF-8
Ctype             | en_US.UTF-8
Access privileges | car_portal_app=CTc/car_portal_app+
                  | web_app_user=c/car_portal_app
```

Можно также отозвать разрешения по умолчанию в шаблонной базе данных, тогда подключение ко всем вновь создаваемым базам по умолчанию будет запрещено. Если метакоманда `\l` выводит пустой список разрешений на уровне базы данных, значит, для этой базы действуют разрешения по умолчанию.

Будьте осторожны при выгрузке и восстановлении базы данных командами `pg_dump` и `pg_restore`. Разрешения на уровне базы данных не восстанавливаются, их надо будет предоставить явно.

Безопасность на уровне схемы

Пользователь может создавать объекты в схеме командой `CREATE` или обращаться к ним. Чтобы дать пользователю доступ к некоторой схеме, нужно предоставить ему разрешения:

```
GRANT USAGE ON SCHEMA car_portal_app TO web_app_user, public_user;
```

Безопасность на уровне таблицы

На уровне таблицы могут быть заданы разрешения `INSERT`, `UPDATE`, `DELETE`, `TRIGGER`, `REFERENCES` и `TRUNCATE`. Ключевое слово `ALL` позволяет предоставить все разрешения сразу:

```
GRANT ALL ON <table_name> TO <role>;
```

Разрешения `REFERENCES` и `TRIGGER` позволяют создавать ограничения внешнего ключа и триггеры соответственно. Можно задавать списки таблиц и ролей через запятую и даже предоставлять определенной роли разрешения сразу на все отношения, имеющиеся в схеме.

Безопасность на уровне столбца

PostgreSQL позволяет определять разрешения на уровне столбца. Для демонстрации создадим таблицу и роль следующим образом:

```
CREATE DATABASE test_column_acl;
\c test_column_acl;
CREATE TABLE test_column_acl AS SELECT * FROM (values (1,2), (3,4)) as n(f1, f2);
CREATE ROLE test_column_acl;
GRANT SELECT (f1) ON test_column_acl TO test_column_acl;
```

Для проверки попробуем получить из таблицы все данные:

```
test_column_acl=# SET ROLE test_column_acl ;
SET
test_column_acl=> TABLE test_column_acl;
ERROR: permission denied for relation test_column_acl
test_column_acl=> SELECT f1 from test_column_acl ;
 f1
----
  1
  3
(2 rows)
```

Безопасность на уровне строк

Безопасность на уровне строк (row-level security – **RLS**), или **строковая политика безопасности**, применяется для управления доступом к строкам таблицы с помощью команд `INSERT`, `UPDATE`, `SELECT` и `DELETE`. На команду `TRUNCATE`, а также на ограничения ссылочной целостности – первичного ключа, внешнего ключа и уникальности – строковая безопасность не распространяется. Кроме того, правила безопасности на уровне строк не применяются к суперпользователю. Чтобы включить безопасность на уровне строк, нужно выполнить для каждой таблицы такую команду `ALTER`:

```
ALTER TABLE <table_name> ENABLE ROW LEVEL SECURITY
```

И далее нужно определить политики доступа к определенному набору строк со стороны одной или нескольких ролей. По этой причине в таблицу часто включается столбец, содержащий имя роли. Ниже мы создаем двух пользователей и таблицу и включаем для этой таблицы безопасность на уровне строк:

```
CREATE DATABASE test_rls;
\c test_rls
CREATE USER admin;
CREATE USER guest;
CREATE TABLE account (
  account_name NAME,
  password TEXT
);
INSERT INTO account VALUES('admin', 'admin'), ('guest', 'guest');
GRANT ALL ON account to admin, guest;
ALTER TABLE account ENABLE ROW LEVEL SECURITY;
```

По умолчанию, если никакие политики не определены, то пользователю будет запрещен доступ к строкам:

```
test_rls=# SET ROLE admin;
test_rls=> table account;
 account_name | password
--------------+----------
(0 rows)
```

Политику можно было бы определить так:

```
CREATE POLICY account_policy_user ON account USING (account_name = current_user);
test_rls=# SET ROLE admin;
test_rls=> Table account;
 account_name | password
--------------+----------
 admin        | admin
(1 row)
```

Здесь мы просто сравниваем поле account_name каждой строки с current_user и, если они совпадают, возвращаем строку. Отметим, что в этой политике нет ограничений на операции, поэтому она распространяется также на команды INSERT, UPDATE и DELETE.

```
test_rls=# SET ROLE admin;
test_rls=> INSERT INTO account values('guest', 'guest');
ERROR: new row violates row-level security policy for table "account"
Синтаксис команды создания политики имеет вид:
CREATE POLICY name ON table_name
    [ FOR { ALL | SELECT | INSERT | UPDATE | DELETE } ]
    [ TO { role_name | PUBLIC | CURRENT_USER | SESSION_USER } [, ...] ]
    [ USING ( using_expression ) ]
    [ WITH CHECK ( check_expression ) ]
```

Команда CREATE POLICY весьма гибкая, она позволяет создавать политики для определенной операции и задавать произвольное условие в виде булева выражения. Фраза WITH CHECK служит для контроля новых или измененных строк. Например, если мы хотим, чтобы пользователи могли видеть все содержимое таблицы account, но изменять только собственные строки, то можем выполнить такую команду:

```
CREATE POLICY account_policy_write_protected ON account USING (true) WITH
CHECK (account_name = current_user);
test_rls=# SET ROLE admin;
test_rls=> Table account;
 account_name | password
--------------+----------
 admin        | admin
 guest        | guest
(2 rows)

test_rls=> INSERT INTO account values('guest', 'guest');
ERROR: new row violates row-level security policy for table "account"
```

По умолчанию политики *разрешительные*, т. е. объединяются связкой OR. Выше были созданы две политики: одна позволяет пользователю видеть только свои строки, а другая – видеть все строки. В итоге пользователь может видеть все строки:

```
\d account
             Table "public.account"
Column        | Type | Collation | Nullable | Default
--------------+------+-----------+----------+---------
 account_name | name |           |          |
 password     | text |           |          |
Policies:
    POLICY "account_policy_write_protected"
      USING (true)
      WITH CHECK ((account_name = CURRENT_USER))
    POLICY "account_user"
      USING ((account_name = CURRENT_USER))
```

PostgreSQL поддерживает также ограничительные политики, объединяемые связкой AND. Введем ограничение – доступ к таблице разрешен только в рабочее время:

```
CREATE POLICY account_policy_time ON account AS RESTRICTIVE USING (
date_part('hour', statement_timestamp()) BETWEEN 8 AND 16 ) WITH CHECK
(account_name = current_user);
test_rls=# set role admin;
test_rls=# select now();
              now
-------------------------------
 2017-10-07 17:42:34.663909+02
(1 row)

test_rls=> table account;
 account_name | password
--------------+----------
(0 rows)
```

Как видим, из-за ограничительной политики не было возвращено ни одной строки. Все строки отфильтрованы выражением, условие фильтрации вычислялось, когда была установлена роль admin:

Кортеж	account_policy	account_policy_write_protected	account_policy_time	Конечный результат
(admin,admin)	True	True	False	False
(guest,guest)	False	True	False	False

Все кортежи (строки) отфильтрованы из-за политики account_policy_time, конечный результат для кортежа (admin, admin) равен True OR True AND False, то есть False.

Шифрование данных

По умолчанию PostgreSQL самостоятельно шифрует секретные данные, в т. ч. пароли ролей. Но и пользователи могут шифровать и дешифрировать данные с помощью расширения pgcrypto.

Шифрование паролей ролей в PostgreSQL

Подробные сведения о роли, созданной с опциями PASSWORD и LOGIN, можно найти в каталожной таблице pg_shadow. Не рекомендуется использовать для задания пароля такой формат команды:

```
CREATE ROLE <role_name> <with options> PASSWORD 'some_password';
```

Дело в том, что команда CREATE ROLE может появиться в таблице pg_stat_activity, а также в журналах сервера, например:

```
postgres=# SELECT query FROM pg_stat_activity;
                query
-----------------------------------
 SELECT query FROM pg_stat_activity;
 create role c password 'c';
```

Все пароли, хранящиеся в таблице pg_shadow, зашифрованы с «солью». После переименования учетной записи пароль сбрасывается:

```
postgres=# ALTER ROLE a RENAME TO b;
NOTICE: MD5 password cleared because of role rename
```

Для создания пользователя с паролем рекомендуется использовать метакоманду \password в psql, потому что в этом случае пароль в открытом виде не появится ни в истории команд psql, ни в журналах сервера, ни где-либо еще. Чтобы изменить пароль, следует выполнить эту метакоманду от имени суперпользователя:

```
postgres=#\password <some_role_name>
Enter new password:
Enter it again:
ERROR: role "<some_role_name>" does not exist
```

Расширение pgcrypto

Расширение pgcrypto предоставляет криптографические средства. Для шифрования и дешифрирования данных требуются ресурсы, поэтому важно соблюдать баланс между секретностью и сложностью. Существует два вида шифрования данных:

- в случае одностороннего шифрования некоторая функция генерирует хеш необратимым способом. Часто результирующий зашифрованный текст имеет фиксированную длину; например, алгоритм MD5 генерирует 16-байтовые хеши. Хорошая функция хеширования должна быть быстрой и не порождать один и тот же хеш для разных входных данных;

- двустороннее шифрование позволяет как шифровать, так и дешифрировать данные. В pgcrypto имеются функции для обоих видов шифрования и поддерживается несколько алгоритмов хеширования. Для установки pgcrypto выполните команду

```
CREATE EXTENSION pgcrypto;
```

Одностороннее шифрование

В случае одностороннего шифрования восстанавливать данные в открытом виде не требуется. Зашифрованный текст (свертка) нужен только для того, чтобы проверить, знает ли пользователь секретный текст. Часто одностороннее шифрование используется для хранения паролей. PostgreSQL изначально поддерживает алгоритм MD5, однако, поскольку его легко взломать, стоило бы использовать MD5 с «солью», как показано в примере ниже.

При проверке пароля обычно сравнивают сгенерированную MD5-свертку с хранимой:

```
CREATE TABLE account_md5 (id INT, password TEXT);
INSERT INTO account_md5 VALUES (1, md5('my password'));
SELECT (md5('my password') = password) AS authenticated FROM account_md5;
 authenticated
---------------
 t
(1 row)
```

pgcrypto предоставляет две функции для шифрования пароля: crypt и gen_salt – и при этом освобождает пользователя от необходимости хранить «соль». Функции crypt и gen_salt используются почти так же, как MD5:

```
CREATE TABLE account_crypt (id INT, password TEXT);
INSERT INTO account_crypt VALUES (1, crypt ('my password', gen_salt('md5')));
INSERT INTO account_crypt VALUES (2, crypt ('my password', gen_salt('md5')));
SELECT * FROM account_crypt;
 id | password
----+-----------------------------------
  1 | $1$ITT7yisa$FdRe4ihZ9kep1oU6wBr090
  2 | $1$HT2wH3UL$8DRdP6kLz5LvTXF3F2q610
(2 rows)

SELECT crypt ('my password', password) = password AS authenticated
FROM account_crypt;
 authenticated
---------------
 t
 t
(2 rows)
```

Мы видим, что свертки паролей отличаются из-за того, что была сгенерирована различная «соль». Отметим также, что хранить «соль» нигде не нужно. Наконец, пароль можно сделать более стойким ко взлому, изменив параметры генерации «соли». Например, можно было бы взять алгоритм Blowfish и задать

количество итераций. Чем больше количество итераций, тем медленнее происходит шифрование и тем больше времени потребуется для взлома (см. пример ниже).

```
\timing
SELECT crypt('my password', gen_salt('bf',4));
                              crypt
------------------------------------------------------------
 $2a$04$RZ5KWnI.IB4eLGNnT.37kuui4.Qi4Xh4TZmL7S0B6YW4LRpyZlP/K
(1 row)
Time: 1,801 ms

SELECT crypt('my password', gen_salt('bf',16));
                              crypt
------------------------------------------------------------
 $2a$16$/cUY8PX7v2GLPCQBCbnL6OtlFm4YACmShZaH1gDNZcHyAYBvJ.9jq
(1 row)
Time: 4686,712 ms (00:04,687)
```

Двустороннее шифрование

Двустороннее шифрование применяется для хранения конфиденциальной информации, например о платежах. Расширение pgcrypto содержит две функции – encrypt и decrypt:

```
test=# \df encrypt
                           List of functions
 Schema | Name    | Result data type | Argument data types | Type
--------+---------+------------------+---------------------+--------
 public | encrypt | bytea            | bytea, bytea, text | normal
(1 row)

test=# \df decrypt
                           List of functions
 Schema | Name    | Result data type | Argument data types | Type
--------+---------+------------------+---------------------+--------
 public | decrypt | bytea            | bytea, bytea, text  | normal
(1 row)
```

Функции encrypt и decrypt принимают три аргумента: данные, ключ и алгоритм шифрования. Ниже показано, как зашифровать и дешифрировать строку Hello World с помощью алгоритма aes:

```
test=# SELECT encrypt ('Hello World', 'Key', 'aes');
              encrypt
-----------------------------------
 \xf9d48f411bdee81a0e50b86b501dd7ba
(1 row)

test=# SELECT decrypt(encrypt ('Hello World', 'Key', 'aes'),'Key','aes');
          decrypt
-------------------------
 \x48656c6c6f20576f726c64
(1 row)
```

```
test=# SELECT convert_from(decrypt(encrypt ('Hello World', 'Key',
'aes'),'Key','aes'), 'utf-8');
 convert_from
--------------
 Hello World
(1 row)
```

У такой формы шифрования есть ограничения: например, команда может оказаться в таблице pg_stat_activity или в журнале сервера, а значит, посторонний может узнать ключ.

Существует два способа двустороннего шифрования: симметричное и асимметричное. Выше было показано, как работает симметричное шифрование, когда в функциях encrypt и decrypt используется один и тот же ключ. При асимметричном шифровании есть два ключа: открытый и закрытый. Открытым ключом данные шифруются, а закрытым – дешифрируются. Асимметричное шифрование безопаснее симметричного, но настроить его труднее. Сначала нужно сгенерировать ключи с помощью программы gpg. Эта программа просит задать парольную фразу; в примере ниже парольная фраза опущена, и для простоты мы взяли параметры по умолчанию:

```
gpg --gen-key
```

Затем, чтобы извлечь открытый и закрытый ключи, следует выполнить показанные ниже команды (если генерация ключей производилась от имени пользователя root, то нужно будет сделать владельцем ключей пользователя postgres):

```
$ gpg --list-secret-key
/var/lib/postgresql/.gnupg/secring.gpg
------------------------------------
sec 2048R/28502EEF 2017-10-08
uid Salahaldin Juba <juba@example.com>
ssb 2048R/D0B149A9 2017-10-08
$ gpg -a --export 28502EEF>/var/lib/postgresql/10/main/public.key
$ gpg -a --export-secret-key
D0B149A9>/var/lib/postgresql/10/main/secret.key
```

Флаг --list-secret-key выводит список идентификаторов закрытых ключей, а флаги --export-secret-key и -export экспортируют, соответственно, закрытый и открытый ключи. Флаг -a выгружает ключи в формате, пригодном для копирования и вставки. В кластере PostgreSQL мы должны выполнить функцию dearmor. Кроме того, ключи были перемещены в папку кластера базы данных для удобства работы с функцией pg_read_file. Сгенерировав ключи, мы можем создать обертку вокруг функций pgp_pub_encrypt и pgp_pub_decrypt, чтобы скрыть их местоположение:

```
CREATE OR REPLACE FUNCTION encrypt (text) RETURNS bytea AS
$$
BEGIN
    RETURN pgp_pub_encrypt($1, dearmor(pg_read_file('public.key')));
```

```
END;
$$ LANGUAGE plpgsql;
CREATE OR REPLACE FUNCTION decrypt (bytea) RETURNS text AS
$$
BEGIN
    RETURN pgp_pub_decrypt($1, dearmor(pg_read_file('secret.key')));
END;
$$ LANGUAGE plpgsql;
```

Для тестирования зашифруем строку `Hello World`:

```
test=# SELECT substring(encrypt('Hello World'), 1, 50);
substring
--------------------------------------------------------------------------------
----------------------------
\xc1c04c034db92a51d0b149a90107fe3dc44ec5dccc039aea2a44e1d811426583a265feb22
f68421355a3b4755a6cb19eb3fa
(1 row)

test=#
test=# SELECT decrypt(encrypt('Hello World'));
   decrypt
-------------
 Hello World
(1 row)
```

Резюме

В этой главе мы рассмотрели три аспекта безопасности в PostgreSQL: аутентификацию, авторизацию и шифрование данных. Но это не все – необходимо еще защитить код от внедрения SQL и других известных проблем, к которым можно отнести стоимость функции и барьер безопасности. PostgreSQL предоставляет ряд методов аутентификации, в т. ч. по паролю и полное доверие. Безопасность обеспечивается на различных уровнях: самой базы данных, схемы, таблицы, представления, функции, столбцов и строк. Наконец, конфиденциальные данные можно хранить в базе в зашифрованном виде, если воспользоваться расширением `pgcrypto`.

В следующей главе мы займемся системным каталогом PostgreSQL и дадим рекомендации по обслуживанию базы данных. В частности, мы покажем, как находить потенциальные проблемы, в т. ч. отсутствующие индексы, и устранять их.

Глава 12

Каталог PostgreSQL

Системный каталог PostgreSQL и функции системного администрирования помогают разработчикам и администраторам содержать базу в чистоте и поддерживать высокую производительность. Системный каталог используется для автоматизации ряда задач, в т. ч. нахождения таблиц без индексов, выявления зависимостей между объектами базы данных и проверки состояния базы данных: вычисления общего размера, наличия «разбухших» таблиц и т. д. Извлеченную из системного каталога информацию можно использовать в программах мониторинга, например Nagios, и в динамическом SQL. В этой главе мы обсудим некоторые повседневные задачи администрирования и разработки, такие как очистка данных и удаление неиспользуемых объектов, автоматическое построение индексов и мониторинг доступа к объектам базы. Будут рассмотрены следующие вопросы:

- общее описание системного каталога;
- системный каталог для администраторов;
- использование системного каталога для очистки базы данных;
- использование системного каталога для настройки производительности;
- выборочная выгрузка дерева зависимостей представления.

Системный каталог

В PostgreSQL все объекты базы данных – таблицы, представления, функции, индексы, адаптеры внешних данных, триггеры, ограничения, правила, пользователи, группы и т. д. – описываются с помощью метаданных, хранящихся в таблицах базы. Эта информация размещена в схеме `pg_catalog`, а чтобы людям было удобнее с ней работать, PostgreSQL предоставляет также схему `information_schema`, в которой метаданные обернуты представлениями.

В клиенте `psql` можно точно увидеть, что происходит за кулисами во время выполнения метакоманд, например `\z`. Для этого нужно включить режим `ECHO_HIDDEN` или задать при запуске флаг `-E`. Так можно изучить таблицы из системного каталога, например:

```
postgres=# \set ECHO_HIDDEN
postgres=# \d
********* QUERY **********
SELECT n.nspname as "Schema",
    c.relname as "Name",
    CASE c.relkind WHEN 'r' THEN 'table' WHEN 'v' THEN 'view' WHEN 'm' THEN
'materialized view' WHEN 'i' THEN 'index' WHEN 'S' THEN 'sequence' WHEN 's'
THEN 'special' WHEN 'f' THEN 'foreign table' WHEN 'p' THEN 'table' END as
"Type",
    pg_catalog.pg_get_userbyid(c.relowner) as "Owner"
FROM pg_catalog.pg_class c
     LEFT JOIN pg_catalog.pg_namespace n ON n.oid = c.relnamespace
WHERE c.relkind IN ('r','p','v','m','S','f','')
      AND n.nspname <> 'pg_catalog'
      AND n.nspname <> 'information_schema'
      AND n.nspname !~ '^pg_toast'
      AND pg_catalog.pg_table_is_visible(c.oid)
ORDER BY 1,2;
**************************
Did not find any relations.
```

Как видим, при выполнении метакоманды `\d` серверу отправляется запрос. Помимо режима `ECHO_HIDDEN`, представления из схем `information_schema` и `pg_catalog` можно опрашивать непосредственно:

```
SELECT * FROM information_schema.views
WHERE table_schema IN ('pg_catalog', 'information_schema');
```

В этих схемах находятся сотни представлений, таблиц и административных функций, поэтому мы сможем описать только те, что используются особенно часто.

Одна из главных таблиц в схеме `pg_catalog` – `pg_class`, в ней хранится информация о различных типах отношений: таблицах, индексах, представлениях, последовательностях и составных типах. Атрибут `relkind` определяет тип отношения и может принимать следующие значения:

- `r`: обычная таблица;
- `i`: индекс;
- `S`: последовательность;
- `t`: TOAST-таблица;
- `v`: представление;
- `m`: материализованное представление;
- `c`: составной тип;
- `f`: внешняя таблица;
- `p`: секционированная таблица.

Поскольку эта таблица универсальная, не все столбцы имеют смысл для всех типов.

Технику хранения больших атрибутов (The Oversized-Attribute Storage Technique – **TOAST**) можно рассматривать как метод вертикального секционирования. PostgreSQL

не разрешает кортежу занимать несколько страниц, а размер страницы обычно равен 8 КБ. Поэтому PostgreSQL разбивает большие объекты на части, сжимает их и хранит во вспомогательных TOAST-таблицах.

У каждого отношения есть идентификатор объекта. Эти идентификаторы играют роль первичных ключей в схеме pg_catalog, поэтому важно понимать, как преобразовать **идентификатор объекта** (**OID**) в текстовую форму и получить имя отношения. Отметим также, что существуют разные типы OID; например, тип regclass идентифицирует все отношения, хранящиеся в таблице pg_class, а тип regprocedure используется для идентификации функций. В примере ниже показано, как преобразовать имя таблицы в OID и наоборот:

```
postgres=# SELECT 'pg_catalog.pg_class'::regclass::oid;
  oid
------
 1259
(1 row)

postgres=# SELECT 1259::regclass::text;
   text
----------
 pg_class
(1 row)
```

Можно также получить oid с помощью таблиц pg_class и pg_namespace:

```
SELECT c.oid FROM pg_class c JOIN pg_namespace n ON (c.relnamespace = n.oid)
WHERE relname ='pg_class' AND nspname ='pg_catalog';
  oid
------
 1259
(1 row)
```

Еще одна важная таблица – pg_attribute, в ней хранится информация о таблицах и других объектах из таблицы pg_class. В таблице pg_index хранится информация об индексах. А в таблицах pg_depend и pg_rewrite – информация о зависимых объектах и правилах перезаписывания для таблиц и представлений.

Упомянем также набор таблиц и представлений pg_stat<*>; в них хранится статистическая информация о таблицах, индексах, столбцах, последовательностях и т. д. Она очень ценна для отладки проблем с производительностью и получения сведений о паттернах работы с базой. Следующий запрос показывает некоторые статистические отношения:

```
SELECT relname, case relkind WHEN 'r' THEN 'table' WHEN 'v' THEN 'VIEW' END as type
FROM pg_class
WHERE relname like 'pg_sta%' AND relkind IN ('r','v')
LIMIT 5 ;
 relname                    | type
----------------------------+-------
pg_statistic                | table
pg_stat_user_tables         | VIEW
```

```
pg_stat_xact_user_tables     | VIEW
pg_statio_all_tables         | VIEW
pg_statio_sys_tables         | VIEW
```

Системный каталог для администраторов

В этом разделе описаны некоторые функции, которые часто бывают нужны администратору базы данных. Их можно использовать в повседневной работе, например функцию `pg_reload_conf()`, которая перезагружает кластер после внесения изменений в файл `pg_hba.conf` или `postgresql.conf`, или функцию `pg_terminate_backend(pid)`, которая снимает указанный процесс.

Получение версии кластера баз данных и клиентских программ

Зная номер версии PostgreSQL, пользователь понимает, какие возможности поддерживаются, и может писать SQL-запросы, совместимые с различными версиями. Например, в версиях ниже 9.2 атрибут представления `pg_stat_activity`, содержащий идентификатор процесса, назывался `procpid`, а позже стал называться `pid`.

Чтобы получить версию кластера баз данных, воспользуемся функцией `version`:

```
postgres=# SELECT version();
version
------------------------------------------------------------------------------
-----------------------------------
 PostgreSQL 10.0 on x86_64-pc-linux-gnu, compiled by gcc (Ubuntu
5.4.0-6ubuntu1~16.04.4) 5.4.0 20160609, 64-bit
(1 row)
```

Важно также проверять версию клиентских программ, например `pg_restore`, `pg_dumpall` или `psql`, на предмет совместимости с сервером. Вот как это делается:

```
$ pg_dump --version
pg_dump (PostgreSQL) 10.0
```

Утилита `pg_dump` выгружает данные в файл, который можно восстановить в более свежую версию PostgreSQL. В принципе, `pg_dump` может выгружать данные из базы, версия которой младше ее собственной, но – и это логично – не из базы, версия которой старше ее собственной.

Завершение и отмена пользовательского сеанса

Администратору базы данных часто приходится снимать серверные процессы по разным причинам. Например, иногда очень медленные запросы, работающие на ведомом или ведущем узле, сконфигурированном для потоковой репликации, могут нарушить репликацию.

Есть несколько причин разорвать соединение с базой данных:

- превышено максимальное число подключений. Это бывает, когда сервер сконфигурирован неправильно, например соединения в состоянии `IDLE`

остаются открытыми слишком долго, или когда некоторые сеансы открыты, но в них давно ничего не происходит;
- требуется удалить базу данных, а этого нельзя сделать, пока к ней кто-то подключен. Такая необходимость иногда возникает при тестировании;
- очень медленные запросы. Это может оказать каскадный эффект на потоковую репликацию, а также и на другие транзакции.

Функция `pg_terminate_backend(pid)` разрывает соединение целиком, а функция `pg_cancel_backend(pid)` отменяет только текущий запрос. В следующем примере завершаются все соединения с текущей базой данных, за исключением текущего сеанса:

```
SELECT pg_terminate_backend(pid) FROM pg_stat_activity
WHERE datname = current_database() AND pid <> pg_backend_pid();
```

Функция `pg_cancel_backend` снимает текущий запрос, но приложение может запустить его заново. Функция `pg_terminate_backend` завершает соединение, но то же самое можно сделать с помощью команды Linux `kill`. Однако это опасно, потому что по неосторожности можно снять главный серверный процесс PostgreSQL, а не процесс, обслуживающий одно подключение.

Представление `pg_stat_activity` в сочетании с функцией `pg_terminate_backend` позволяет достичь большей гибкости. Например, если запущена потоковая репликация, то можно выявить запросы, которые тратят много времени на ведомых узлах, что приводит к большому отставанию репликации. Кроме того, если какое-то приложение сконфигурировано неправильно и в нем образуется много простаивающих соединений, то их можно снять, освободив тем самым память и дав возможность другим клиентам подключиться к базе данных.

PostgreSQL позволяет также завершать SQL-команду по тайм-ауту. Так, в некоторых случаях долго работающие команды блокируют всех остальных клиентов. Величину тайм-аута можно задать глобально в файле `postgresql.conf` или на уровне сеанса:

```
SET statement_timeout to 1000;
SELECT pg_sleep(1001);
ERROR: canceling statement due to statement timeout
```

Задание и получение параметров кластера баз данных

Конфигурационные параметры управляют различными аспектами работы кластера PostgreSQL. Администратор может задать тайм-аут команды, настройки памяти, количество подключений, параметры протоколирования, очистки и планировщика. Разработчику параметры помогают оптимизировать запросы. Узнать значение параметра позволяет функция `current_settings` или вспомогательная команда `show`:

```
SELECT current_setting('work_mem');
 current_setting
-----------------
 4MB
```

```
(1 row)

show work_mem;
 work_mem
----------
 4MB
(1 row)
```

Чтобы изменить значение конфигурационного параметра, следует вызвать функцию set_config(setting_name, new_value, is_local). Если аргумент is_local равен true, то новое значение будет действовать только в текущей транзакции, иначе в текущем сеансе.

```
SELECT set_config('work_mem', '8 MB', false);
```

При выполнении функции set_config может возникнуть ошибка, если указанный параметр неприменим в контексте сеанса, как, например, количество допустимых подключений или разделяемых буферов:

```
SELECT set_config('shared_buffers', '1 GB', false);
ERROR: parameter "shared_buffers" cannot be changed without restarting the server
```

Помимо set_config, PostgreSQL предоставляет для изменения конфигурационных параметров команду ALTER SYSTEM со следующим синтаксисом:

```
ALTER SYSTEM SET configuration_parameter { TO | = } { value | 'value' | DEFAULT }
ALTER SYSTEM RESET configuration_parameter
ALTER SYSTEM RESET ALL
```

Иногда для вступления значения в силу требуется перезагрузка системы или перезапуск. Для выполнения команды ALTER SYSTEM необходимы привилегии администратора. Команда ALTER SYSTEM вносит изменения в файл postgresql.auto.conf. Само название команды говорит, что ее действие носит глобальный характер, поэтому нужно перезагрузить конфигурацию сервера:

```
postgres=# ALTER SYSTEM SET work_mem TO '8MB';
ALTER SYSTEM
postgres=# SHOW work_mem;
 work_mem
----------
 4MB
(1 row)

postgres=# SELECT pg_reload_conf();
 pg_reload_conf
----------------
 t
(1 row)

postgres=# SHOW work_mem;
 work_mem
----------
 8MB
(1 row)
```

Следующая команда показывает содержимое postgresql.auto.conf:

```
$cat postgresql.auto.conf
# Do not edit this file manually!
# It will be overwritten by ALTER SYSTEM command.
work_mem = '8MB'
```

Наконец, просмотр всего файла postgresql.conf не вполне тривиален из-за большого количества параметров PostgreSQL. К тому же у большинства параметров есть начальные значения по умолчанию. Получить список параметров, значения которым присвоены в файле postgresql.conf, а не по умолчанию, легко:

```
SELECT name, current_setting(name), source FROM pg_settings
WHERE source IN ('configuration file');
 name                        | current_setting                          | source
-----------------------------+------------------------------------------+--------
------------
 cluster_name                | 10/main                                  | configuration
file
 DateStyle                   | ISO, DMY                                 | configuration
file
```

Получение размера базы данных и объекта базы данных

Чтобы управлять местом на диске и назначать таблицам и базам данных табличные пространства, нужно знать о размере базы и отдельных объектов. В процессе логического восстановления таблицы следить за ходом операции можно, сравнивая размер исходной таблицы с текущим размером восстанавливаемой. Наконец, зная размер объекта, часто можно сделать вывод о его «разбухании».

Под разбуханием понимается чрезмерное расходование места на диске из-за конкурентных манипуляций данными. Если таблица активно обновляется, то она часто оказывается разбухшей в силу самой природы MVCC. Разбухание обычно устраняется процессом очистки, поэтому выключать процесс автоматической очистки не рекомендуется.

Чтобы узнать размер базы данных, нужно найти ее oid в таблице pg_database и выполнить команду Linux du -h /data_directory/base/oid, где data_directory – папка кластера баз данных, прописанная в файле postgresql.conf.

Беглое знакомство с папками кластеров весьма полезно для уяснения основных концепций и операций PostgreSQL, в т. ч. с конфигурационными файлами. Например, дата создания базы данных совпадает с датой создания файла PG_VERSION в каталоге этой базы.

Дополнительно PostgreSQL предлагает функцию pg_database_size, которая возвращает размер базы данных, и функцию pg_size_pretty для отображения размера в понятном человеку виде:

```
SELECT pg_database.datname, pg_size_pretty(pg_database_size(pg_database.datname)) AS size
FROM pg_database;
 datname         | size
-----------------+---------
 postgres        | 8093 kB
 template1       | 7481 kB
 template0       | 7481 kB
```

Функция `pg_total_relation_size` возвращает размер таблицы со всеми индексами и TOAST-таблицами. Если вас интересует только размер самой таблицы, воспользуйтесь функцией `pg_relation_size`. Эта информация поможет следить за ростом таблицы и табличных пространств. Взгляните на следующий запрос:

```
SELECT tablename, pg_size_pretty(pg_total_relation_size(schemaname||'.'||tablename))
FROM pg_tables LIMIT 2;
 tablename    | pg_size_pretty
--------------+----------------
 pg_statistic | 280 kB
 pg_type      | 184 kB
(2 rows)
```

Наконец, размер индекса также вернет функция `pg_relation_size`:

```
SELECT indexrelid::regclass, pg_size_pretty(pg_relation_size(indexrelid::regclass))
FROM pg_index LIMIT 2;
 Indexrelid                  | pg_size_pretty
-----------------------------+----------------
 pg_toast.pg_toast_2604_index | 8192 bytes
 pg_toast.pg_toast_2606_index | 8192 bytes
(2 rows)
```

Для простоты в программе `psql` имеются метакоманды для получения размера базы данных, таблицы и индекса:

- `\l+`: выводит информацию о базе данных, в т. ч. и размер;
- `\dtis+`: буквы `t`, `i`, `s` означают соответственно «таблица», «индекс» и «последовательность». Эта метакоманда выводит список объектов, включая таблицы, индексы и последовательности. Знак `+` означает, что нужно показать место, занятое объектом на диске.

Размер таблицы на диске дает представление о ходе процесса восстановления данных. Допустим, что мы перемещаем таблицу из одной базы в другую с помощью команды `COPY`; сколько строк уже перемещено, мы узнать не можем, но можно сравнить размеры обеих таблиц и понять, сколько еще осталось.

Очистка базы данных

Зачастую база данных содержит неиспользуемые объекты или очень старые данные. Если их вычистить, то резервное копирование будет происходить быстрее. А с точки зрения разработчика, неиспользуемые объекты – просто шум, мешающий процессу рефакторинга.

Чтобы очистить базу, нужно выявить неиспользуемые объекты, включая таблицы, представления, индексы и функции. Найти пустые и неиспользуемые таблицы поможет статистика: количество «живых» строк, просмотров индекса и последовательных просмотров. Отметим, что поскольку результаты приведенных ниже запросов опираются на статистику, их надо перепроверять. Необходимую информацию содержит таблица `pg_stat_user_tables`, следующий запрос показывает пустые таблицы, исходя из оценки количества кортежей:

```
SELECT relname FROM pg_stat_user_tables WHERE n_live_tup= 0;
```

Любая информация, основанная на статистике, не стопроцентно надежна, поскольку статистика может быть неактуальной.

Чтобы найти пустые или неиспользуемые столбцы, можно взглянуть на атрибут `null_fraction` таблицы `pg_stats`. Если `null_fraction` равен 1, значит, в столбце вообще нет данных:

```
SELECT schemaname, tablename, attname FROM pg_stats
WHERE null_frac = 1 and schemaname NOT IN ('pg_catalog', 'information_schema');
```

Для нахождения бесполезных индексов придется сделать два шага:

1) сначала определим, является индекс дубликатом или частью другого индекса;
2) затем на основе статистики индекса выясним, используется ли он.

Следующий запрос позволяет на основе статистики оценить, используется ли индекс. Отметим, что индексы, необходимые для поддержания ограничений – уникальности и первичного ключа, – исключаются, потому что они необходимы, даже если не используются:

```
SELECT schemaname, relname, indexrelname
FROM pg_stat_user_indexes s
    JOIN pg_index i ON s.indexrelid = i.indexrelid
WHERE idx_scan=0 AND NOT indisunique AND NOT indisprimary;
```

Для выявления дублирующих индексов можно ориентироваться на пересечение множеств атрибутов. Следующий SQL-запрос сравнивает атрибуты индексов и возвращает те индексы, для которых множества атрибутов пересекаются:

```
WITH index_info AS (
SELECT
    pg_get_indexdef(indexrelid) AS index_def,
    indexrelid::regclass index_name ,
    indrelid::regclass table_name,
    array_agg(attname order by attnum) AS index_att
FROM
    pg_index i JOIN pg_attribute a ON i.indexrelid = a.attrelid
GROUP BY
    pg_get_indexdef(indexrelid), indrelid, indexrelid
) SELECT DISTINCT
    CASE WHEN a.index_name > b.index_name THEN a.index_def ELSE b.index_def
```

```
END AS index_def,
    CASE WHEN a.index_name > b.index_name THEN a.index_name ELSE
b.index_name END AS index_name,
    CASE WHEN a.index_name > b.index_name THEN b.index_def ELSE a.index_def
END AS overlap_index_def,
    CASE WHEN a.index_name > b.index_name THEN b.index_name ELSE
a.index_name END AS overlap_index_name,
    a.index_att = b.index_att as full_match,
    a.table_name
FROM
    index_info a INNER JOIN
    index_info b ON (a.index_name != b.index_name AND a.table_name =
      b.table_name AND a.index_att && b.index_att );
```

Для тестирования создадим пересекающиеся индексы и выполним запрос:

```
CREATE TABLE test_index_overlap(a int, b int);
CREATE INDEX ON test_index_overlap (a,b);
CREATE INDEX ON test_index_overlap (b,a);
```

Получится такой результат:

```
-[ RECORD 1 ]------+-----------------------------------------------------
-------------------------------------------------------------------------
index_def          | CREATE INDEX test_index_overlap_b_a_idx ON test_index_overlap
USING btree (b, a)
index_name         | test_index_overlap_b_a_idx
overlap_index_def  | CREATE INDEX test_index_overlap_a_b_idx ON
test_index_overlap USING btree (a, b)
overlap_index_name | test_index_overlap_a_b_idx
full_match         | f
table_name         | test_index_overlap
```

Вычистить неиспользуемые представления и функции несколько сложнее. По умолчанию PostgreSQL собирает статистику об индексах и таблицах, но не о функциях. Чтобы разрешить сбор статистики о функциях, нужно включить параметр `track_functions`. Статистика использования функций хранится в таблице `pg_stat_user_functions`.

О представлениях не собирается никакой статистики, если только они не материализованные. Разбираться, используется ли представление, нам придется самостоятельно. Это можно сделать, соединив представление с функцией, имеющей некоторый побочный эффект. Например, она может записывать в какую-нибудь таблицу, сколько раз обращались к представлению, или помещать сообщение в журнал. Для демонстрации этой техники создадим простую функцию, которая пишет в журнал:

```
CREATE OR REPLACE FUNCTION monitor_view_usage (view_name TEXT) RETURNS
BOOLEAN AS $$
BEGIN
  RAISE LOG 'The view % is used on % by % ', view_name, current_time, session_user;
  RETURN TRUE;
END;
$$LANGUAGE plpgsql cost .001;
```

Теперь предположим, что нам хотелось бы удалить следующее представление dummy_view, но нужна уверенность, что от него не зависят никакие приложения:

```
CREATE OR REPLACE VIEW dummy_view AS
SELECT dummy_text FROM (VALUES('dummy')) as dummy(dummy_text);
```

Чтобы удостовериться в том, что представление не используется, перепишем его, соединив с функцией monitor_view_usage:

```
-- Включить в представление функцию monitor_view_usage
CREATE OR REPLACE VIEW dummy_view AS
SELECT dummy_text FROM (VALUES('dummy')) as dummy(dummy_text) cross join
monitor_view_usage('dummy_view');
```

Если к представлению кто-то обратится, то в журнале появится такая запись:

```
$ tail /var/log/postgresql/postgresql-10-main.log
2017-10-10 16:54:15.375 CEST [21874] postgres@postgres LOG: The view
dummy_view is used on 16:54:15.374124+02 by postgres
```

Очистка данных в базе

Чтобы почистить разбухшие таблицы и индексы, нужно просто выполнить команду VACUUM. Узнать о том, как вычисляется степень разбухания таблиц и индексов, можно, заглянув в исходный код подключаемого к Nagios модуля check_postgres на сайте компании Bucardo: https://bucardo.org/wiki/Check_postgres.

Очистка данных – важная тема. Зачастую при создании приложения о жизненном цикле данных не думают, в результате накапливаются горы старых и никому не нужных данных. Наличие грязных данных мешает ряду процедур, в т. ч. рефакторингу базы. К тому же это влечет нежелательные последствия для всех процессов в компании: неточные отчеты, проблемы с выставлением счетов, несанкционированный доступ и т. д.[1]

Выше мы на примерах показали, как выявлять неиспользуемые объекты, но это еще не все. Сами данные также надо чистить, и хорошо бы определить их жизненный цикл. Грязные данные могут появляться разными способами, но мы рассмотрим только строки-дубликаты, возникающие из-за отсутствия ограничения уникальности или первичного ключа. Первым делом найдем таблицы, не имеющие таких ограничений. Это легко сделать с помощью схемы information_schema:

```
SELECT table_catalog, table_schema, table_name
FROM information_schema.tables
WHERE table_schema NOT IN ('information_schema','pg_catalog')
EXCEPT
SELECT table_catalog, table_schema, table_name
FROM information_schema.table_constraints
```

[1] Не вполне понятно, каким образом хранение старых данных может привести ко всем этим ужасам, но будем считать, что авторам виднее. – *Прим. перев.*

```
WHERE constraint_type IN ('PRIMARY KEY', 'UNIQUE') AND
      table_schema NOT IN ('information_schema', 'pg_catalog');
```

Далее выявим таблицы, которые действительно содержат дубликаты. Для этого нужно агрегировать данные. Создадим таблицу, содержащую дубликаты:

```
CREATE TABLE duplicate AS
SELECT (random () * 9 + 1)::INT as f1, (random () * 9 + 1)::INT as f2
FROM generate_series (1,40);

SELECT count(*), f1, f2 FROM duplicate GROUP BY f1, f2;
 count | f1 | f2
-------+----+----
     2 |  7 |  4
     3 |  5 |  4
     2 |  2 |  6
(3 rows)
```

Хитрость в том, как удалить дубликаты, ведь строки-то одинаковые. Для этого нужно пометить некоторые строки как остающиеся, а остальные удалить. Это можно сделать, воспользовавшись столбцом `ctid`. В PostgreSQL у каждой строки есть заголовок, а столбец `ctid` содержит физическое положение версии строки в таблице. Его можно использовать как временный идентификатор строки, поскольку он может изменяться в результате выполнения команд обслуживания, например `CLUSTER`.

Чтобы удалить дубликаты, воспользуемся командой `DELETE USING`:

```
SELECT ctid, f1, f2 FROM duplicate where (f1, f2) = (7,4)
 ctid   | f1 | f2
--------+----+----
 (0,11) |  7 | 4
 (0,40) |  7 | 4
(2 rows)
BEGIN;
DELETE FROM duplicate a USING duplicate b
WHERE a.f1= b.f1 and a.f2= b.f2 and a.ctid > b.ctid;
SELECT ctid, f1, f2 FROM duplicate where (f1, f2) = (7,4);
 ctid   | f1 | f2
--------+----+----
 (0,11) |  7 | 4
(1 row)
```

Не забывайте про явные блоки транзакции, когда манипулируете данными. Это позволит откатить изменения в случае ошибки.

Есть и другие способы удалить строки-дубликаты. Например, можно с помощью команд `CREATE TABLE` и `SELECT DISTINCT` создать таблицу, содержащую только уникальные строки, а затем удалить исходную таблицу и переименовать вновь созданную:

```
CREATE TABLE <tmp> AS SELECT DISTINCT * FROM <orig_tbl>;
DROP TABLE <orig_tbl>;
ALTER TABLE <tmp> RENAME TO <orig_tbl>;
```

или

```
CREATE UNLOGGED TABLE <tmp> AS SELECT DISTINCT * FROM <orig_tbl>;
DROP TABLE <orig_tbl>;
ALTER TABLE <tmp> RENAME TO <orig_tbl>;
    ALTER TABLE <tmp> SET LOGGED;
```

Этот способ быстрее описанного выше, но он не будет работать, если существуют объекты, зависящие от удаляемой таблицы: представления, индексы и т. п.

Если вы незнакомы с командой `DELETE USING`, то существует способ сделать то же самое с помощью CTEs (хотя первый запрос, скорее всего, будет работать быстрее):

```
WITH should_not_delete AS (
  SELECT min(ctid) FROM duplicate GROUP BY f1, f2
)
DELETE FROM duplicate WHERE ctid NOT IN (SELECT min FROM should_not_delete);
```

Оптимизация производительности

Для достижения высокой производительности PostgreSQL необходимо выбрать правильные конфигурационные параметры и позаботиться о физической схеме, включая индексы. Планы выполнения зависят от собранной статистики использования таблиц; к счастью, есть возможность управлять порядком сбора статистики.

Для разработчиков хорошая производительность очень важна. Мы дадим две рекомендации, касающиеся внешних ключей.

- **Всегда стройте индексы по внешним ключам.** Это позволит PostgreSQL выбирать из таблицы данные, применяя просмотр индексов.
- **Увеличивайте ориентир статистики по столбцам внешних ключей.** Это относится также ко всем предикатам, поскольку позволяет точнее оценить количество подходящих строк. По умолчанию ориентир равен 100, а максимальное значение равно 10 000. Чем больше ориентир, тем медленнее работает команда `ANALYZE`.

Для применения обеих рекомендаций необходимо выявить внешние ключи. Для этой цели пригодится таблица `pg_catalog.pg_constraint`. Чтобы найти все ограничения внешнего ключа, достаточно простого запроса:

```
SELECT * FROM pg_constraint WHERE contype = 'f';
```

Выше было показано, как найти пересекающиеся индексы; мы можем воспользоваться этой идеей, чтобы найти неиндексированные внешние ключи:

```
SELECT
  conrelid::regclass AS relation_name,
  conname AS constraint_name,
  reltuples::bigint AS number_of_rows,
  indkey AS index_attributes,
```

```
  conkey AS constraint_attributes,
  CASE WHEN conkey && string_to_array(indkey::text, ' ')::SMALLINT[] THEN
FALSE ELSE TRUE END as might_require_index
FROM
  pg_constraint JOIN pg_class ON (conrelid = pg_class.oid) JOIN
  pg_index ON indrelid = conrelid
WHERE contype = 'f';
```

Отметим, что если `indkey` пересекается с `conkey`, то индекс, возможно, добавлять и не надо, но это необходимо проверить, проанализировав, как он используется. В запросе также отбирается количество кортежей `reltuples`, поскольку это важный фактор, влияющий на решение о том, нужно ли строить индекс, – ведь последовательный просмотр больших таблиц обходится очень дорого. Можно вместо этого посмотреть статистику о количестве последовательных просмотров таблицы.

Найдя внешние ключи, мы можем построить индексы по ним командой `CREATE INDEX` и изменить подразумеваемые по умолчанию параметры сбора статистики командой `ALTER TABLE`.

Избирательная выгрузка

Если модифицируется какое-то представление, например добавляется новый столбец или изменяется тип столбца, то нужно модифицировать и все зависящие от него представления. К сожалению, PostgreSQL не предоставляет средств для логической выгрузки зависимого объекта.

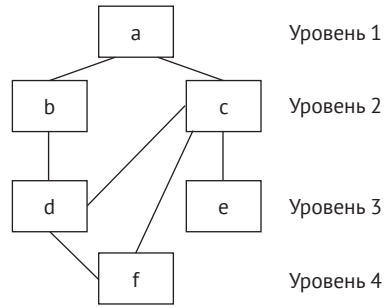
Команда `pg_dump` выгружает всю базу или некоторые объекты. Кроме того, рекомендуется хранить весь код в репозитории Git.

Заранее планируйте весь цикл разработки, включая кодирование, тестирование, развертывание в промежуточной и в производственной среде. Пользуйтесь инструментами управления версиями типа Git и средствами миграции баз данных типа Flyway. Это сэкономит уйму времени.

К сожалению, SQL-код унаследованных приложений зачастую не хранится в системе управления версиями или в инструменте миграции. В таком случае, если требуется изменить определение представления или тип столбца, придется найти все зависимые представления, удалить их, а затем восстановить.

Первым делом поймем, какие представления следует удалить и восстановить. В зависимости от характера задачи могут понадобиться разные скрипты; как правило, определения зависимых представлений удаляют. В таблицах `pg_catalog`, `pg_depend` и `pg_rewrite` хранится информация о зависимостях и правилах переписывания представлений. В более удобном для человека виде та же информация находится в таблице `information_schema.view_table_usage`.

Пусть имеется несколько зависящих друг от друга представлений (см. рисунок ниже) и требуется изменить представление `a`. Для этого придется удалить и заново создать зависимые объекты.



Чтобы сгенерировать такое дерево зависимостей, выполним следующие запросы:

```
CREATE TABLE test_view_dep AS SELECT 1;
CREATE VIEW a AS SELECT 1 FROM test_view_dep;
CREATE VIEW b AS SELECT 1 FROM a;
CREATE VIEW c AS SELECT 1 FROM a;
CREATE VIEW d AS SELECT 1 FROM b,c;
CREATE VIEW e AS SELECT 1 FROM c;
CREATE VIEW f AS SELECT 1 FROM d,c;
```

А чтобы разрешить дерево зависимостей, воспользуемся рекурсивным запросом:

```
CREATE OR REPLACE FUNCTION get_dependency (schema_name text, view_name text)
RETURNS TABLE (schema_name text, view_name text, level int) AS $$
WITH RECURSIVE view_tree(parent_schema, parent_view, child_schema,
child_view, level) AS
(
  SELECT parent.view_schema, parent.view_name ,parent.table_schema,
parent.table_name, 1
  FROM information_schema.view_table_usage parent
  WHERE parent.view_schema = $1 AND parent.view_name = $2
  UNION ALL
  SELECT child.view_schema, child.view_name, child.table_schema,
child.table_name, parent.level + 1
  FROM view_tree parent JOIN information_schema.view_table_usage child ON
child.table_schema = parent.parent_schema AND child.table_name
= parent.parent_view
)
SELECT DISTINCT parent_schema, parent_view, level
FROM (
  SELECT parent_schema, parent_view, max (level)
  OVER (PARTITION BY parent_schema, parent_view) as max_level,level
  FROM view_tree
) AS FOO
WHERE level = max_level
ORDER BY 3 ASC;
$$
LANGUAGE SQL;
```

Во внутренней части запроса вычисляются уровни зависимостей, а внешняя нужна для удаления дубликатов. Результат выполнения этой функции для представления a показан ниже:

```
SELECT * FROM get_dependency('public', 'a');
 schema_name | view_name | level
-------------+-----------+-------
 public      | a         |     1
 public      | b         |     2
 public      | c         |     2
 public      | d         |     3
 public      | e         |     3
 public      | f         |     4
(6 rows)
```

Чтобы выгрузить определение представления, можно воспользоваться утилитой `pg_dump` с флагом `-t`, который задает имя отношения. Таким образом, для выгрузки представлений из предыдущего примера можно написать такой скрипт на bash:

```
$relations=$(psql -t -c "SELECT string_agg (' -t ' ||
quote_ident(schema_name) || '.' || quote_ident(view_name), ' ' ORDER BY
level ) FROM get_dependency ('public'::text, 'a'::text)")

$echo $relations
-t public.a -t public.b -t public.c -t public.d -t public.e -t public.f

$pg_dump -s $relations
--
-- PostgreSQL database dump
--
-- Dumped from database version 10.0
-- Dumped by pg_dump version 10.0
SET statement_timeout = 0;
SET lock_timeout = 0;
SET idle_in_transaction_session_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
SET check_function_bodies = false;
SET client_min_messages = warning;
SET row_security = off;
SET search_path = public, pg_catalog;
--
-- Name: a; Type: VIEW; Schema: public; Owner: postgres
--
CREATE VIEW a AS
  SELECT 1
    FROM test_view_dep;
...
```

Команда `psql` с флагом `-t` возвращает только кортежи, а функция `string_agg` порождает список подлежащих выгрузке представлений, отсортированный по уровню зависимости. Таким образом, мы гарантируем, что представления будут затем восстановлены в нужном порядке.

Резюме

Каталожные таблицы дают бесценную информацию для автоматизации задач обслуживания и администрирования базы данных. Так, в предыдущем разделе нам удалось выгрузить дерево зависимых объектов в правильном порядке. Мы также умеем автоматизировать такие повседневные задачи, как обнаружение разбухших таблиц и висящих блокировок, вычисление размера и оценка состояния базы данных. Благодаря статистическим таблицам мы можем находить неиспользуемые индексы, а также узнавать, по каким столбцам было бы полезно построить индексы – например, по столбцам внешних ключей.

В каталоге PostgreSQL хранится метаинформация о базах данных и их объектах. Эту информацию можно получать с помощью команд SQL. Однако не рекомендуется работать с данными непосредственно в схеме `pg_catalog`. PostgreSQL поддерживает также схему `information_schema`, в которой информация представлена в более удобном виде.

Об одном лишь каталоге PostgreSQL можно было бы написать целую книгу рецептов. Полезно было бы иметь в своем арсенале рецепты по мониторингу состояния кластера баз данных: потребление памяти, простаивающие соединения, размер базы данных, состояние процессов. Кроме того, полезно знать, как работать с ролями и принадлежностью к ролям, с разрешениями на доступ к объектам, как обнаруживать блокировки и следить за состоянием потоковой репликации.

В следующей главе мы обсудим некоторые подходы к оптимизации производительности. Мы рассмотрим конфигурационные параметры кластера, используемые для настройки его производительности в целом. Также мы расскажем о типичных ошибках при написании запросов и поговорим о таких способах повышения производительности, как использование индексов, секционирование таблиц и исключение в силу ограничений.

Глава 13

Оптимизация производительности базы данных

Оптимизация производительности базы данных – тема многогранная, тут и конфигурация оборудования, и настройка сети, и конфигурация базы данных, и переписывание SQL-запросов, и обслуживание индексов, и многое другое. В этой главе мы будем говорить только о базовой конфигурации и о переписывании запросов.

Вообще говоря, для настройки производительности базы данных нужно знать о характере системы, например используется она для **оперативного анализа данных** (**OLAP**) или для **оперативной транзакционной обработки** (**OLTP**). Работа базы данных может быть ограничена скоростью ввода-вывода или быстродействием процессора; в зависимости от этого настраиваются такие стороны кластера, как количество и мощность процессоров, конфигурация RAID, объем оперативной памяти. Сконфигурировав сервер базы данных, можно прогнать эталонные тесты, например с помощью программы `pgbench`, и определить количество **транзакций в секунду** (transactions per second – **TPS**).

Следующий шаг оптимизации производительности производится, когда система запущена и работает, и обычно это делается периодически. На этой стадии можно настроить систему мониторинга, например `pgbadger`, анализатор рабочей нагрузки **PostgreSQL Workload Analyzer** (**PoWA**) и `pg_stat_statements` с целью поиска узких мест и медленных запросов.

Для оптимизации медленного запроса его нужно сначала проанализировать. Если запрос плохо написан, то, может быть, будет достаточно его переписать. В противном случае можно создать отсутствующие индексы, изменить конфигурационные параметры сервера, переработать физическую структуру и т. д.

В этой главе рассматриваются следующие вопросы:

- настройка конфигурационных параметров PostgreSQL;
- оптимизация производительности записи;

- оптимизация производительности чтения;
- обнаружение проблем в планах выполнения запросов;
- типичные ошибки при написании запросов;
- секционирование таблиц;
- переписывание запросов.

Настройка конфигурационных параметров PostgreSQL

Выставленные по умолчанию конфигурационные параметры PostgreSQL не годятся для работы в производственной среде, значения некоторых параметров слишком малы. В процессе разработки приложений рекомендуется иметь тестовую систему, очень близкую к реальной, – только тогда можно будет точно замерить производительность. В любом случае необходимо изменить следующие параметры.

Максимальное количество подключений

Максимальное количество подключений – важный параметр базы данных. Каждый клиент потребляет память, а следовательно, уменьшается объем памяти, доступной для других целей. По умолчанию параметр `max_connections` равен 100; уменьшив его значение, администратор сможет увеличить параметр `work_mem`. В общем случае имеет смысл установить программу организации пула соединений. Это уменьшит потребление памяти и повысит производительность, поскольку на завершение и установление соединений тратится время. Программ такого рода много, но наиболее зрелых две:

- PgBouncer;
- Pgpool-II.

Можно также организовать пул соединений на уровне приложения. Так, для Java есть много подобных решений, например: Hikari, пул соединений Apache Tomcat, dbcp2 и c3p0.

Параметры памяти

Существует несколько параметров для управления потреблением памяти.

- **Разделяемые буферы** (`shared_buffers`). По умолчанию под разделяемые буферы выделяется 32 МБ; но рекомендуется выделять примерно 25 процентов общего объема памяти, однако не больше 8 ГБ в Linux и не более 512 МБ в Windows. Иногда задание очень большого значения `shared_buffers` ведет к повышению производительности, потому что вся база данных целиком кешируется в памяти. Но, вообще говоря, у подобного подхода есть существенный недостаток: не хватает памяти для таких операций, как сортировка и хеширование.
- **Рабочая память** (`work_mem`). По умолчанию 4 МБ, для счетных задач это значение лучше увеличить. Параметр `work_mem` связан с количеством подключений, поэтому общий объем выделенной памяти равен количеству

подключений, умноженному на work_mem. Рабочая память используется для сортировки и хеширования, поэтому ее объем влияет на запросы с фразами ORDER BY, DISTINCT, UNION и EXCEPT. Для проверки можно проанализировать запрос с сортировкой и посмотреть, производится ли сортировка в памяти или на диске:

```
EXPLAIN ANALYZE SELECT n FROM generate_series(1,5) as foo(n) order by n;
Sort (cost=59.83..62.33 rows=1000 width=4) (actual time=0.075..0.075 rows=5 loops=1)
Sort Key: n
Sort Method: quicksort Memory: 25kB
  -> Function Scan on generate_series foo (cost=0.00..10.00
rows=1000 width=4) (actual time=0.018..0.018 rows=5 loops=1)"
Total runtime: 0.100 ms
```

Параметры жесткого диска

Существует несколько параметров, повышающих производительность ввода-вывода, однако такое повышение дается недаром. Так, параметр fsync заставляет каждую транзакцию сбрасывать данные на диск после фиксации. Если его выключить, то производительность возрастет, особенно в случае операций массовой загрузки. Небольшие значения параметров max_wal_size или checkpoint_segments могут снизить производительность в системах с интенсивной записью. С другой стороны, с их увеличением растет и время восстановления.

В отдельных случаях, например при массовой загрузке, производительность можно повысить, изменив параметры жесткого диска, установив минимальные параметры протоколирования и, наконец, отключив автоочистку. Однако по завершении загрузки обязательно восстановите конфигурацию сервера и запустите команду VACUUM ANALYZE.

Параметры планировщика

Эффективный размер кеша (параметр effective_cache_size) следует установить, исходя из оценки того, сколько памяти доступно для кеширования диска в операционной системе и внутри базы данных с учетом того, что уже используется самой ОС и другими приложениями. Для выделенного сервера PostgreSQL эта величина составляет от 50 до 70 процентов общего объема памяти.

Можно также поэкспериментировать с параметром random_page_cost, который отдает предпочтение просмотру индексов перед последовательным просмотром. По умолчанию значение random_page_cost равно 4.0. При использовании современных технологий сетевых хранилищ SAN/NAS можно установить значение 3, а для SSD-дисков – от 1.5 до 2.5. Это минимальный список; в действительности надо еще настроить протоколирование, контрольные точки, параметры WAL и очистки.

Отметим, что в производственной системе изменить некоторые параметры трудно, потому что необходим перезапуск. Это относится к max_connections, shared_buffers и fsync. Другие же параметры, например work_mem, можно задавать на уровне сеанса, так что у разработчика имеется возможность настраивать work_mem под конкретные запросы.

Эталонное тестирование вам в помощь

pgbench – простая программа, которая выполняет заранее подготовленный набор SQL-команд и вычисляет среднюю скорость транзакции (количество транзакций в секунду). Она является реализацией стандарта **TPC-B Совета по производительности транзакционной обработки** (Transaction Processing Performance Council – **TPC**). Программу pgbench можно также настроить с помощью скриптов. В общем случае клиент эталонного тестирования следует запускать в одиночестве, чтобы не отвлекать время и память тестируемого сервера на другие задачи. Кроме того, pgbench рекомендуется прогонять несколько раз с различными нагрузками и конфигурационными параметрами. Наконец, помимо pgbench, существуют открытые реализации и других эталонных стандартов, например **TPC-C** и **TPC-H**. Порядок вызова pgbench такой: pgbench [options] dbname. Флаг -i служит для создания в базе данных тестовых таблиц, а флаг -s определяет коэффициент масштабирования базы, или количество строк в каждой таблице.

Вывод pgbench с коэффициентом масштабирования по умолчанию на виртуальной машине с одним процессором выглядит примерно так:

```
$pgbench -i test_database
  creating tables...
  100000 of 100000 tuples (100%) done (elapsed 0.71 s, remaining 0.00 s).
  vacuum...
```

О других флагах команды можно прочитать на страницах руководства (pgbench --help).

Оптимизация производительности записи

Высокая нагрузка по записи может представать в разных видах. Например, это может быть результат записи событий в PostgreSQL или результат массовой загрузки выгруженной базы данных или задачи ETL. Для оптимизации производительности записи можно выполнить следующие действия:

- конфигурация оборудования:
  - следует использовать RAID 1+0, а не RAID 5 или 6. Для интенсивной записи RAID 10 показывает гораздо более высокую производительность. Кроме того, журналы транзакций (pg_xlog) лучше хранить на отдельном диске;
  - можно использовать SSD-диски, оснащенные **кешем с отложенной записью** (Write-back cache – WBC), это заметно повышает производительность записи. Но убедитесь, что SSD-диск не теряет кешированных данных в случае отказа электропитания;
- параметры PostgreSQL:
  - fsync. По умолчанию этот режим включен. Он гарантирует, что база данных сможет восстановиться после аппаратного сбоя, поскольку данные физически записаны на диск. Если вы доверяете своему оборудо-

ванию, то можете выключить этот параметр. Но помните, что в таком случае аппаратный сбой может привести к повреждению данных;
- `synchronous_commit` и `commit_delay`. По умолчанию параметр `synchronous_commit` включен и означает, что транзакция будет ждать записи WAL-файла на диск и только потом сообщит клиенту об успешном завершении. Это повышает производительность систем с большим числом одновременных мелких транзакций. Параметр `commit_delay` позволяет задержать сброс WAL-файла на указанное количество микросекунд. Сочетание обоих параметров снижает негативный эффект `fsync`. Но, в отличие от выключения `fsync`, аппаратный сбой не приведет к повреждению данных, хотя часть данных может быть потеряна;
- `max_wal_size` и `checkpoint_segments` (в PostgreSQL 9.4 и старше). Параметр `checkpoint_segments` в версии PostgreSQL 9.5 объявлен нерекомендуемым и заменен параметрами `min_wal_size` и `max_wal_size`. Между `max_wal_size` и `checkpoint_segment` имеется такое соотношение: `max_wal_size = (3 * checkpoint_segments) * 16` МБ. Увеличение `max_wal_size` дает выигрыш в производительности, потому что WAL записывается на диск не так часто. Это влияет на частоту контрольных точек и в случае сбоя замедляет восстановление;
- `wal_buffers`. По умолчанию этот параметр выключен. Он служит для хранения данных WAL, еще не записанных на диск. Его увеличение полезно, если имеется сильно загруженный сервер с несколькими одновременными клиентами. Максимальное значение равно 16 МБ;
- `maintenece_work_mem`. Этот параметр напрямую не влияет на производительность вставки, но увеличивает производительность создания и обслуживания индексов. Его увеличение косвенно способствует ускорению операций `INSERT`, особенно при вставке в индексированную таблицу;
- другие параметры. Для повышения производительности можно отключить еще несколько параметров, например протоколирование. Кроме того, на время массовой загрузки можно отключать автоочистку, чтобы она не прерывала процесса;

- команды DDL и DML:
  - существует много приемов для повышения производительности в процессе массовой загрузки. Например, можно отключить триггеры, индексы и внешние ключи для копируемой таблицы. Можно также создать нежурналируемую таблицу (`UNLOGGED`), а затем преобразовать ее в журналируемую командой `ALTER TABLE <table name> SET LOGGED`;
  - в ситуации, когда производится много операций вставки, тоже есть полезные трюки. Прежде всего можно увеличить размер пакета в одной транзакции. Это уменьшает задержку синхронной фиксации, а также экономит идентификаторы транзакций, вследствие чего реже будет запускаться процесс очистки, предотвращающий закольцовывание этих идентификаторов. Второй прием – использовать команду `COPY`.

Для этой цели можно применять JDBC-драйвер CopyManager. Наконец, всячески рекомендуем использовать подготовленные команды. Они работают быстрее, потому что заранее откомпилированы на стороне сервера;

– внешние инструменты. Для массовой загрузки имеет смысл попробовать очень быструю программу pg_bulkload.

Подведем итоги. В системах с большим количеством операций записи лучше писать порциями, а не вставлять по одной строке командой INSERT. Кроме того, предпочтительнее использовать команду COPY, а не INSERT. Наконец, операцию можно распараллелить, используя несколько сеансов вместо одного. Начиная с версии PostgreSQL 9.3 к нашим услугам команда COPY с опцией FREEZE, которая часто применяется для начальной загрузки данных. Опция FREEZE нарушает принципы MVCC тем, что данные становятся видны в других сеансах сразу после загрузки.

Для демонстрации влияния fsync подготовим SQL-скрипт для pgbench:

```
$ cat test.sql
\set aid random(1, 100000 * :scale)
\set bid random(1, 1 * :scale)
\set tid random(1, 10 * :scale)
\set delta random(-5000, 5000)
BEGIN;
UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid,
:bid, :aid, :delta, CURRENT_TIMESTAMP);
END;
```

Теперь, не внося никаких изменений, выполним pgbench, чтобы узнать базовую производительность системы. Все параметры в этом тесте имеют значения по умолчанию.

```
$pgbench -t 1000 -c 15 -f test.sql
starting vacuum...end.
transaction type: test.sql
scaling factor: 1
query mode: simple
number of clients: 15
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 15000/15000
latency average = 18.455 ms
tps = 812.801437 (including connections establishing)
tps = 813.032862 (excluding connections establishing)
```

Чтобы понять, как поведет себя система после выключения fsync, изменим параметры и перезагрузим сервер:

```
$ psql -U postgres << EOF
> ALTER SYSTEM RESET ALL;
> ALTER SYSTEM SET fsync to off;
```

```
> EOF
ALTER SYSTEM
ALTER SYSTEM

$/etc/init.d/postgresql restart
[ ok ] Restarting postgresql (via systemctl): postgresql.service.

$ pgbench -t 1000 -c 15 -f test.sql
starting vacuum...end.
transaction type: test.sql
scaling factor: 1
query mode: simple
number of clients: 15
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 15000/15000
latency average = 11.976 ms
tps = 1252.552937 (including connections establishing)
tps = 1253.082492 (excluding connections establishing)
```

Чтобы оценить влияние параметров `synchronous_commit` и `commit_delay`, произведем следующие изменения:

```
$ psql -U postgres << EOF
> ALTER SYSTEM RESET ALL;
> ALTER SYSTEM SET synchronous_commit to off;
> ALTER SYSTEM SET commit_delay to 100000;
> EOF
ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM

$ /etc/init.d/postgresql restart
[ ok ] Restarting postgresql (via systemctl): postgresql.service.

$ pgbench -t 1000 -c 15 -f test.sql
starting vacuum...end.
transaction type: test.sql
scaling factor: 1
query mode: simple
number of clients: 15
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 15000/15000
latency average = 12.521 ms
tps = 1197.960750 (including connections establishing)
tps = 1198.416907 (excluding connections establishing)
```

Оптимизация производительности чтения

PostgreSQL дает возможность выяснить, почему запрос работает медленно. За кулисами сервер анализирует таблицы, собирает по ним статистику гистограммы. Большое участие в этом принимает процесс автоочистки, который

освобождает место на диске, обновляет статистику таблиц и выполняет другие административные задачи, в т. ч. предотвращает закольцовывание идентификатора транзакции. Благодаря статистике таблиц PostgreSQL выбирает план выполнения с наименьшей стоимостью. При его вычислении учитывается стоимость ввода-вывода и, естественно, стоимость процессорных операций. Кроме того, PostgreSQL предоставляет команду EXPLAIN, которая показывает сгенерированный план выполнения.

Начинающим чрезвычайно полезно записывать один и тот же запрос разными способами и сравнивать результаты. Например, в некоторых случаях конструкцию `NOT IN` можно преобразовать в `LEFT JOIN` или `NOT EXIST`. А конструкцию `IN` можно переписать с использованием `INNER JOIN` или `EXISTS`. Благодаря таким упражнениям разработчик учится избегать определенных конструкций и начинает понимать, при каких условиях какая конструкция будет работать лучше. Вообще говоря, `NOT IN` часто вызывает проблемы с производительностью, потому что планировщик не может привлекать индексы для выполнения такого запроса.

Важно также постоянно следить за нововведениями в SQL. Сообщество PostgreSQL очень активно и нередко вносит модификации, направленные на решение типичных проблем. Например, появившуюся в PostgreSQL 9.3 конструкцию `LATERAL JOIN` можно использовать для оптимизации некоторых запросов с `GROUP BY` и `LIMIT`.

План выполнения и команда EXPLAIN

При оптимизации запросов в PostgreSQL первым делом нужно понять, как читать планы выполнения, сгенерированные командой `EXPLAIN`. Эта команда показывает, каким образом будут читаться данные из таблиц, например с помощью просмотра индекса или последовательного просмотра. Показываются также способы соединения таблиц и оценка количества строк.

У команды `EXPLAIN` есть несколько опций; при наличии опции `ANALYZE` команда выполняется и возвращает фактическое время и количество строк. Кроме того, `EXPLAIN` может рассказать об использовании буферов и кешировании. Ниже приведен синтаксис команды, в нем `option` может принимать значения `ANALYZE`, `VERBOSE`, `COSTS`, `BUFFERS`, `TIMING`, `SUMMARY` и `FORMAT`:

```
\h EXPLAIN
Description: show the execution plan of a statement
Syntax:
EXPLAIN [ ( option [, ...] ) ] statement
EXPLAIN [ ANALYZE ] [ VERBOSE ] statement
```

Для экспериментов создадим простую таблицу и заполним ее, как показано ниже:

```
postgres=# CREATE TABLE guru ( id INT PRIMARY KEY, name TEXT NOT NULL );
CREATE TABLE
postgres=# INSERT INTO guru SELECT n , md5 (random()::text) FROM
```

```
generate_series (1, 100000) AS foo(n);
INSERT 0 100000
postgres=# -- To update table statistics
postgres=# ANALYSE guru;
ANALYZE
```

Чтобы получить план выполнения запроса, который выбирает все записи из таблицы guru, воспользуемся командой EXPLAIN:

```
postgres=# EXPLAIN SELECT * FROM guru;
                            QUERY PLAN
-------------------------------------------------------------
 Seq Scan on guru (cost=0.00..1834.00 rows=100000 width=37)
(1 row)
```

План выполнения состоит из одного узла: последовательного просмотра таблицы. Оценка количества строк правильная, поскольку после вставки мы проанализировали таблицу. Для поддержания статистики базы данных в актуальном состоянии важно, что работал процесс автоочистки.

Разумеется, если статистика неправильная, то ожидать хороших планов выполнения не приходится. После массовой загрузки данных в базу рекомендуется выполнить команду ANALYZE для обновления статистики. Можно также управлять объемом выборки, производимой ANALYZE, с помощью следующей команды:

```
ALTER TABLE <table name> ALTER COLUMN <column name> SET STATISTICS <integer>;
```

Увеличение статистики по столбцу улучшает оценку строк, но замедляет процесс автоочистки. Стоимость запроса – это оценка затрат на его выполнение. В примере выше 0.00 – стоимость выборки первой строки, а 1834.00 – стоимость выборки всех строк, вычисляемая по формуле (relpages * seq_page cost) + (reltuples * cpu_tuple_cost). Количество страниц отношения (relpages) и строк отношения (reltuples) можно найти в таблице pg_class. Величины seq_page_cost и cpu_tuple_cost – это конфигурационные параметры планировщика. Таким образом мы и получаем число 1834:

```
postgres=# SELECT relpages*current_setting('seq_page_cost')::numeric +
reltuples*current_setting('cpu_tuple_cost')::numeric as cost FROM pg_class
WHERE relname='guru';
 cost
------
 1834
(1 row)
```

В этом простом примере последовательного просмотра вычислить стоимость не составляет труда. Но если запрос включает вычисление предикатов, группировку, сортировку и соединение, то получить оценку стоимости гораздо сложнее. Наконец, величина 37 в плане выполнения – это средняя ширина кортежа в байтах, которую можно найти в таблице pg_stats. Чтобы выполнить запрос и одновременно получить его стоимость, имеется команда EXPLAIN

(ANALYZE) или EXPLAIN ANALYZE. В следующем примере возвращаются все строки, идентификатор которых принадлежит диапазону от 10 до 20:

```
postgres=# EXPLAIN ANALYZE SELECT * FROM guru WHERE id >= 10 and id < 20;
QUERY PLAN
------------------------------------------------------------------------------
--------------------------------------
 Index Scan using guru_pkey on guru (cost=0.29..8.51 rows=11 width=37)
(actual time=0.007..0.010 rows=10 loops=1)
  Index Cond: ((id >= 10) AND (id < 20))
Planning time: 0.132 ms
Execution time: 0.028 ms
(4 rows)
```

Здесь планировщик получил оценку, очень близкую к реальным значениям: 11 вместо 10. Кроме того, в сгенерированном планировщиком плане теперь используется просмотр индекса, поскольку в запрос входит предикат. В плане выполнения имеется и другая информация, в т. ч. количество циклов и фактическое время. Отметим, что этот план занимает четыре строки с различными отступами. Читать его следует снизу вверх, начиная со строки с наибольшим отступом. Строка Index Cond: ((id >= 10) AND (id < 20)) имеет отступ и принадлежит узлу Index Scan.

В режиме EXPLAIN (BUFFERS) команда показывает эффект кеширования, а также информацию о том, используется ли кеш и правильно ли он настроен. Чтобы в полной мере оценить эффект кеширования, необходимо выполнить тестирование до и после прогрева кеша. Имеется также возможность управлять форматом плана выполнения. В следующем примере демонстрируется результат использования опций BUFFERS и FORMAT в режиме тестирования на холодном и горячем кеше. Чтобы протестировать на холодном кеше, остановим и снова запустим сервер:

```
$ /etc/init.d/postgresql stop
$ /etc/init.d/postgresql start
```

Иногда перезапуска PostgreSQL недостаточно для очистки кеша, поскольку PostgreSQL пользуется также буферами операционной системы. В ОС Ubuntu для очистки кеша нужно выполнить команду echo 3 > /proc/sys/vm/drop_caches.

Теперь прочитаем таблицу guru:

```
postgres=# EXPLAIN (ANALYZE, FORMAT YAML, BUFFERS) SELECT * FROM guru;
            QUERY PLAN
-------------------------------
 - Plan: +
      Node Type: "Seq Scan" +
      Parallel Aware: false +
      Relation Name: "guru" +
      Alias: "guru" +
      Startup Cost: 0.00 +
      Total Cost: 1834.00 +
```

```
      Plan Rows: 100000 +
      Plan Width: 37 +
      Actual Startup Time: 0.016+
      Actual Total Time: 209.332+
      Actual Rows: 100000 +
      Actual Loops: 1 +
      Shared Hit Blocks: 0 +
      Shared Read Blocks: 834 +
      Shared Dirtied Blocks: 0 +
      Shared Written Blocks: 0 +
      Local Hit Blocks: 0 +
      Local Read Blocks: 0 +
      Local Dirtied Blocks: 0 +
      Local Written Blocks: 0 +
      Temp Read Blocks: 0 +
      Temp Written Blocks: 0 +
   Planning Time: 0.660 +
   Triggers: +
   Execution Time: 213.879
(1 row)
```

Протестируем на прогретом кеше, выполнив тот же запрос:

```
postgres=# EXPLAIN (ANALYZE, FORMAT YAML, BUFFERS) SELECT * FROM guru;
             QUERY PLAN
------------------------------
 - Plan: +
      Node Type: "Seq Scan" +
      Parallel Aware: false +
      Relation Name: "guru" +
      Alias: "guru" +
      Startup Cost: 0.00 +
      Total Cost: 1834.00 +
      Plan Rows: 100000 +
      Plan Width: 37 +
      Actual Startup Time: 0.009+
      Actual Total Time: 10.105 +
      Actual Rows: 100000 +
      Actual Loops: 1 +
      Shared Hit Blocks: 834 +
      Shared Read Blocks: 0 +
      Shared Dirtied Blocks: 0 +
      Shared Written Blocks: 0 +
      Local Hit Blocks: 0 +
      Local Read Blocks: 0 +
      Local Dirtied Blocks: 0 +
      Local Written Blocks: 0 +
      Temp Read Blocks: 0 +
      Temp Written Blocks: 0 +
  Planning Time: 0.035 +
  Triggers: +
  Execution Time: 14.447
(1 row)
```

Первое, что бросается в глаза, – разница во времени выполнения. Во второй раз потребовалось почти на 93% меньше времени. Это объясняется тем, что данные читаются из памяти, а не с диска, о чем говорит поле `Shared Hit Blocks` (разделяемые блоки в кеше), красноречиво свидетельствующее, что все данные обнаружены в памяти.

Обнаружение проблем в планах выполнения запросов

Команда `EXPLAIN` может показать, почему запрос работает медленно, особенно если выполнять ее с опциями `BUFFER` и `ANALYZE`. Вот на что следует обращать внимание, пытаясь понять, хорош план выполнения или нет.

- **Оценка числа строк и истинное число строк.** Это важно, потому что планировщик ориентируется на эту величину, выбирая метод выполнения запроса. Есть два случая: оценка числа строк завышена или занижена. Неправильная оценка влияет на все алгоритмы: выборки данных с диска, сортировки, соединения и т. д. Вообще говоря, завышенная оценка – это нехорошо, но заниженная – гораздо хуже. С одной стороны, при выполнении соединения очень больших таблиц методом вложенных циклов время выполнения растет экспоненциально, если для простоты предположить, что стоимость этого алгоритма равна $O(n^2)$. С другой стороны, соединение с небольшой таблицей методом хеширования замедляет выполнение запроса, но эффект оказывается не таким чудовищным, как в первом случае.
- **Сортировка в памяти или на диске.** Сортировка выполняется при наличии таких фраз, как `DISTINCT`, `LIMIT`, `ORDER`, `GROUP BY`, и некоторых других. Если памяти достаточно, то сортировка производится в памяти, иначе на диске.
- **Буферный кеш.** Важно обращать внимание на то, сколько данных находится в буферах и сколько из них грязных. Чтение данных из буфера значительно повышает производительность.

Чтобы продемонстрировать плохой план выполнения, собьем планировщик PostgreSQL с толку, выполнив операцию над столбцом `id`:

```
postgres=# EXPLAIN SELECT * FROM guru WHERE upper(id::text)::int < 20;
                            QUERY PLAN
-------------------------------------------------------------
 Seq Scan on guru (cost=0.00..3334.00 rows=33333 width=37)
 Filter: ((upper((id)::text))::integer < 20)
(2 rows)
```

Здесь планировщик PostgreSQL не может вычислить выражение `upper(id::text)::int < 20` самостоятельно. И индексом воспользоваться также не может, потому что нет индекса, построенного по такому выражению, да и в любом случае планировщик не стал бы использовать индекс, поскольку оценка числа строк слишком велика. Если бы этот запрос был частью другого, то ошибка

распространилась бы каскадом, поскольку подзапрос, возможно, пришлось бы выполнить несколько раз. Мы создали проблему искусственно, но неверная оценка количества строк – одна из самых часто встречающихся ошибок. Быть может, это результат неправильной настройки автоочистки. А в больших таблицах причиной может быть слишком малый показатель статистики.

Наконец, для обнаружения глубинной причины падения производительности полезно знать о различных алгоритмах: соединение методом вложенных циклов, хеширования и слияния, просмотр индекса, просмотр битовой карты и т. д. В следующем примере иллюстрируется соединение с большой таблицей методом вложенных циклов. Сначала выполним запрос, не выключая параметры планировщика:

```
postgres=# EXPLAIN ANALYZE WITH tmp AS (SELECT * FROM guru WHERE id
<10000) SELECT * FROM tmp a inner join tmp b on a.id = b.id;
QUERY PLAN
------------------------------------------------------------------------------
-------------------------------------------------
 Merge Join (cost=2079.95..9468.28 rows=489258 width=72) (actual
time=8.193..26.330 rows=9999 loops=1)
  Merge Cond: (a.id = b.id)
  CTE tmp
    -> Index Scan using guru_pkey on guru (cost=0.29..371.40 rows=9892
width=37) (actual time=0.022..3.134 rows=9999 loops=1)
         Index Cond: (id < 10000)
    -> Sort (cost=854.28..879.01 rows=9892 width=36) (actual
time=6.238..8.641 rows=9999 loops=1)
         Sort Key: a.id
         Sort Method: quicksort Memory: 1166kB
         -> CTE Scan on tmp a (cost=0.00..197.84 rows=9892 width=36)
(actual time=0.024..5.179 rows=9999 loops=1)
    -> Sort (cost=854.28..879.01 rows=9892 width=36) (actual
time=1.950..2.658 rows=9999 loops=1)
         Sort Key: b.id
         Sort Method: quicksort Memory: 1166kB
         -> CTE Scan on tmp b (cost=0.00..197.84 rows=9892 width=36)
(actual time=0.001..0.960 rows=9999 loops=1)
 Planning time: 0.143 ms
 Execution time: 26.880 ms
(15 rows)
```

Чтобы протестировать метод вложенных циклов, мы должны выключить все прочие методы соединения и в первую очередь соединение слиянием и хешированием:

```
postgres=#set enable_mergejoin to off ;
SET
postgres=#set enable_hashjoin to off ;
SET
postgres=# EXPLAIN ANALYZE WITH tmp AS (SELECT * FROM guru WHERE id <10000)
SELECT * FROM tmp a inner join tmp b on a.id = b.id;
```

```
QUERY PLAN
------------------------------------------------------------------------------
-----------------------------------------------------
 Nested Loop (cost=371.40..2202330.60 rows=489258 width=72) (actual
time=0.029..15389.001 rows=9999 loops=1)
  Join Filter: (a.id = b.id)
  Rows Removed by Join Filter: 99970002
  CTE tmp
    -> Index Scan using guru_pkey on guru (cost=0.29..371.40 rows=9892
width=37) (actual time=0.022..2.651 rows=9999 loops=1)
         Index Cond: (id < 10000)
  -> CTE Scan on tmp a (cost=0.00..197.84 rows=9892 width=36) (actual
time=0.024..1.445 rows=9999 loops=1)
  -> CTE Scan on tmp b (cost=0.00..197.84 rows=9892 width=36) (actual
time=0.000..0.803 rows=9999 loops=9999)
 Planning time: 0.117 ms
 Execution time: 15390.996 ms
(10 rows)
```

Обратите внимание на огромную разницу между соединением методами слияния и вложенных циклов. Кроме того, как видно из примера, для соединения слиянием нужен отсортированный список.

Предыдущий пример можно переписать без CTE. Мы использовали CTE только для того, чтобы продемонстрировать крайние случаи, возникающие, когда количество строк занижено. Несмотря на то что мы запретили соединение слиянием и хешированием, PostgreSQL воспользовалась для выполнения запроса методом вложенных циклов с просмотром индекса. В данном случае вместо двух циклов она обошлась одним, что существенно повысило производительность, по сравнению с вариантом на основе CTE.

```
EXPLAIN ANALYZE SELECT * FROM guru as a inner join guru b on a.id = b.id
WHERE a.id < 10000;
QUERY PLAN
------------------------------------------------------------------------------
-----------------------------------------------------
 Nested Loop (cost=0.58..7877.92 rows=9892 width=74) (actual
time=0.025..44.130 rows=9999 loops=1)
  -> Index Scan using guru_pkey on guru a (cost=0.29..371.40 rows=9892
width=37) (actual time=0.018..14.139 rows=9999 loops=1)
       Index Cond: (id < 10000)
  -> Index Scan using guru_pkey on guru b (cost=0.29..0.76 rows=1
width=37) (actual time=0.003..0.003 rows=1 loops=9999)
       Index Cond: (id = a.id)
 Planning time: 0.123 ms
 Execution time: 44.873 ms
(7 rows)
```

Если вы столкнулись с очень длинным и сложным планом выполнения и не можете понять, в чем «затык», попробуйте запретить алгоритм с экспоненциальным временем работы, например соединение методом вложенных циклов.

Типичные ошибки при написании запросов

Есть ряд типичных порочных практик и ошибок, которые часто допускают разработчики.

Избыточные операции

Лишние операции, например просмотр диска, сортировка и фильтрация, могут появляться по разным причинам. К примеру, некоторые разработчики вставляют слово DISTINCT даже туда, где оно не нужно, или не понимают разницы между UNION, UNION ALL, EXCEPT, EXCEPT ALL и т. д. Это замедляет выполнение запроса, особенно если ожидаемое число строк велико. Следующие два запроса эквивалентны, поскольку в таблице есть первичный ключ, но запрос с DISTINCT работает гораздо медленнее:

```
postgres=# \timing
Timing is on.
postgres=# SELECT * FROM guru;
Time: 85,089 ms

postgres=# SELECT DISTINCT * FROM guru;
Time: 191,335 ms
```

Еще одна типичная ошибка – использование DISTINCT совместно с UNION:

```
postgres=# SELECT * FROM guru UNION SELECT * FROM guru;
Time: 267,258 ms
postgres=# SELECT DISTINCT * FROM guru UNION SELECT DISTINCT * FROM guru;
Time: 346,014 ms
```

В отличие от UNION ALL, команда UNION удаляет все дубликаты из результирующего множества. Поэтому в повторной сортировке и фильтрации нет никакой нужды.

Еще одна распространенная ошибка – включение ORDER BY в определение представления. Тогда при выборке из этого представления производится сортировка, возможно, совершенно лишняя:

```
postgres=# CREATE OR REPLACE VIEW guru_vw AS SELECT * FROM guru order by 1 asc;
CREATE VIEW
Time: 42,370 ms
postgres=#
postgres=# SELECT * FROM guru_vw;
Time: 132,292 ms
```

Здесь выборка данных из представления занимает 132 мс, а напрямую из таблицы – всего 85 мс.

Индексы отсутствуют или построены не так

Если по некоторому выражению, содержащему столбцы, отсутствует индекс, то таблица просматривается целиком. Существует несколько случаев, когда индексы способствуют повышению производительности. Например, всячески

рекомендуется строить индекс по внешним ключам. Для демонстрации создадим и заполним таблицу:

```
CREATE TABLE success_story (id int, description text, guru_id int
references guru(id));
INSERT INTO success_story (id, description, guru_id)
SELECT n, md5(n::text), random()*99999+1
FROM generate_series(1,200000) AS foo(n) ;
```

Чтобы получить историю успеха некоторого гуру, нужно написать простой запрос, соединяющий таблицы `guru` и `success_story`:

```
postgres=# EXPLAIN ANALYZE SELECT * FROM guru inner JOIN success_story on
guru.id = success_story.guru_id WHERE guru_id = 1000;
QUERY PLAN
------------------------------------------------------------------------------
-------------------------------------------
 Nested Loop (cost=0.29..4378.34 rows=3 width=78) (actual
time=0.030..48.011 rows=1 loops=1)
  -> Index Scan using guru_pkey on guru (cost=0.29..8.31 rows=1 width=37)
(actual time=0.017..0.020 rows=1 loops=1)
       Index Cond: (id = 1000)
  -> Seq Scan on success_story (cost=0.00..4370.00 rows=3 width=41)
(actual time=0.011..47.987 rows=1 loops=1)
       Filter: (guru_id = 1000)
       Rows Removed by Filter: 199999
 Planning time: 0.114 ms
 Execution time: 48.040 ms
(8 rows)
```

Обратите внимание, что таблица `success_story` просматривалась последовательно. Чтобы это исправить, мы можем создать индекс по внешнему ключу:

```
postgres=# CREATE index on success_story (guru_id);
CREATE INDEX
postgres=# EXPLAIN ANALYZE SELECT * FROM guru inner JOIN success_story on
guru.id =success_story.guru_id WHERE guru_id = 1000;
QUERY PLAN
------------------------------------------------------------------------------
-------------------------------------------------------------
 Nested Loop (cost=4.74..24.46 rows=3 width=78) (actual time=0.023..0.024
rows=1 loops=1)
  -> Index Scan using guru_pkey on guru (cost=0.29..8.31 rows=1 width=37)
(actual time=0.010..0.010 rows=1 loops=1)
       Index Cond: (id = 1000)
  -> Bitmap Heap Scan on success_story (cost=4.44..16.12 rows=3 width=41)
(actual time=0.009..0.010 rows=1 loops=1)
       Recheck Cond: (guru_id = 1000)
       Heap Blocks: exact=1
  -> Bitmap Index Scan on success_story_guru_id_idx (cost=0.00..4.44 rows=3
width=0) (actual time=0.006..0.006 rows=1 loops=1)
       Index Cond: (guru_id = 1000)
 Planning time: 0.199 ms
 Execution time: 0.057 ms
```

После построения индекса время выполнения разительно уменьшилось: с 48 мс до 0.057 мс.

Поиск по тексту также выигрывает от наличия индексов. Иногда требуется искать без учета регистра, например если речь идет о логине. Для этого можно привести текст к одному регистру – верхнему или нижнему. Для демонстрации создадим еще одну функцию, потому что алгоритм хеширования `md5` порождает буквы только в нижнем регистре:

```
CREATE OR REPLACE FUNCTION generate_random_text ( int ) RETURNS TEXT AS
$$
  SELECT
string_agg(substr('0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRST
UVWXYZ', trunc(random() * 62)::integer + 1, 1), '')
  FROM generate_series(1, $1)
$$
LANGUAGE SQL;
```

Для создания таблицы, содержащей логины, записанные буквами обоих регистров, выполним такой код:

```
CREATE TABLE login as SELECT n, generate_random_text(8) as login_name FROM
generate_series(1, 1000) as foo(n);
CREATE INDEX ON login(login_name);
VACUUM ANALYZE login;
```

Функция `generate_random_text()` генерирует случайный текст заданной длины. Пусть требуется проверить, существует ли в таблице некоторая запись. Можно было бы поступить так:

```
postgres=# EXPLAIN SELECT * FROM login WHERE login_name = 'jxaG6gjJ';
QUERY PLAN
------------------------------------------------------------------------
 Index Scan using login_login_name_idx on login (cost=0.28..8.29 rows=1
width=13)
   Index Cond: (login_name = 'jxaG6gjJ'::text)
(2 rows)
```

Как видим, применяется просмотр индекса, потому что по столбцу `login_name` построен индекс. Индекс используется и в случае, когда вызывается неволатильная функция от константного аргумента, поскольку оптимизатор вычисляет функцию самостоятельно:

```
postgres=# EXPLAIN SELECT * FROM login WHERE login_name = lower('jxaG6gjJ');
QUERY PLAN
------------------------------------------------------------------------
--------
 Index Scan using login_login_name_idx on login (cost=0.28..8.29 rows=1
width=13)
   Index Cond: (login_name = 'jxag6gjj'::text)
(2 rows)
```

Но если вызывается функция от столбца, то производится последовательный просмотр:

```
postgres=# EXPLAIN SELECT * FROM login WHERE lower(login_name) =
lower('jxaG6gjJ');
                        QUERY PLAN
-----------------------------------------------------
 Seq Scan on login (cost=0.00..21.00 rows=5 width=13)
 Filter: (lower(login_name) = 'jxag6gjj'::text)
(2 rows)
```

Обратите внимание, что и в этом случае оценка количества строк равна 5, потому что оптимизатор не может правильно вычислить предикат. Для решения проблемы добавим индекс:

```
postgres=# CREATE INDEX ON login(lower(login_name));
CREATE INDEX
postgres=# analyze login;
ANALYZE
postgres=# EXPLAIN SELECT * FROM login WHERE lower(login_name) = lower('jxaG6gjJ');
QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using login_lower_idx on login (cost=0.28..8.29 rows=1 width=13)
    Index Cond: (lower(login_name) = 'jxag6gjj'::text)
(2 rows)
```

Как именно индексировать текст, зависит от характера доступа. Вообще, существует два способа: в первом, рассчитанном на поиск строки в начале текста, используется `opclass`, во втором – типы `tsquery` и `tsvector`:

```
postgres=# CREATE INDEX on login (login_name text_pattern_ops);
CREATE INDEX
postgres=# EXPLAIN ANaLYZE SELECT * FROM login WHERE login_name like 'a%';
QUERY PLAN
-------------------------------------------------------------------------------
-------------------------------------------------
 Index Scan using login_login_name_idx1 on login (cost=0.28..8.30 rows=1
width=13) (actual time=0.019..0.063 rows=19 loops=1)
   Index Cond: ((login_name ~>=~ 'a'::text) AND (login_name ~<~ 'b'::text))
   Filter: (login_name ~~ 'a%'::text)
 Planning time: 0.109 ms
 Execution time: 0.083 ms
(5 rows)
Time: 0,830 ms
```

Наконец, если требуется искать по началу текста, но без учета регистра, то можно воспользоваться комбинацией функции `lower` и `text_pattern_ops`:

```
postgres=# CREATE INDEX login_lower_idx1 ON login (lower(login_name)
text_pattern_ops);
CREATE INDEX
postgres=# EXPLAIN ANaLYZE SELECT * FROM login WHERE lower(login_name) like 'a%';
QUERY PLAN
-------------------------------------------------------------------------------
-----------------------------------------------
```

```
 Bitmap Heap Scan on login (cost=4.58..11.03 rows=30 width=13) (actual
time=0.037..0.060 rows=30 loops=1)
   Filter: (lower(login_name) ~~ 'a%'::text)
   Heap Blocks: exact=6
   -> Bitmap Index Scan on login_lower_idx1 (cost=0.00..4.58 rows=30
width=0) (actual time=0.025..0.025 rows=30 loops=1)
        Index Cond: ((lower(login_name) ~>=~ 'a'::text) AND
(lower(login_name) ~<~ 'b'::text))
 Planning time: 0.471 ms
 Execution time: 0.080 ms
(7 rows)
```

Использование CTE без необходимости

Общие табличные выражения (CTE) позволяют разработчикам выражать очень сложную логику. Кроме того, в некоторых случаях CTE повышают производительность. Однако их использование может вызывать проблему при перемещении предиката в конец, поскольку PostgreSQL не производит оптимизацию за пределами CTE; каждое CTE работает изолированно от других. Чтобы понять, в чем состоит ограничение, взгляните на следующие два эквивалентных запроса и обратите внимание на различие в производительности:

```
\o /dev/null
\timing
postgres=# SELECT * FROM guru WHERE id = 4;
Time: 0,678 ms

postgres=# WITH gurus as (SELECT * FROM guru) SELECT * FROM gurus WHERE id
= 4;
Time: 67,641 ms
```

Использование процедурного языка PL/pgSQL

Кеширование в языке PL/pgSQL – замечательное средство повышения производительности, но если разработчик невнимателен, то оно может стать причиной плохих планов выполнения. Пусть, например, мы хотим обернуть функцией следующий запрос:

```
SELECT * FROM guru WHERE id <= <predicate>;
```

Здесь план выполнения кешировать не следует, потому что он может зависеть от значения предиката. Так, если в предикате указано значение 1, то планировщик предпочтет просмотр индекса, а если 90 000, то последовательный просмотр. Но давайте изменим запрос:

```
SELECT * FROM guru WHERE id = <predicate>;
```

Теперь план стоит кешировать, потому что он будет одинаков для всех предикатов, т. к. по столбцу `id` построен индекс.

Еще отметим, что обработка исключений в PL/pgSQL обходится очень дорого, поэтому применять эту возможность следует с осторожностью.

Межстолбцовая корреляция

Из-за межстолбцовой корреляции возможны ошибки в оценке количества строк, поскольку PostgreSQL предполагает, что все столбцы независимы. Но на самом деле это далеко не всегда так. Например, в некоторых культурах существует корреляция между именем и фамилией. Другой пример – страна и язык в предпочтениях клиентов. Чтобы понять, в чем опасность межстолбцовой корреляции, создадим таблицу клиентов:

```
CREATE TABLE client (
    id serial primary key,
    name text,
    country text,
    language text
);
```

Для демонстрации коррелированной статистики сгенерируем тестовые данные следующим образом:

```
INSERT INTO client(name, country, language)
SELECT generate_random_text(8), 'Germany', 'German'
FROM generate_series(1, 10);

INSERT INTO client(name, country, language)
SELECT generate_random_text(8), 'USA', 'English'
FROM generate_series(1, 10);

VACUUM ANALYZE client;
```

Если мы захотим найти пользователей, для которых язык `language` равен `German`, а страна `country` – `Germany`, то получим неверную оценку строк, поскольку эти столбцы коррелированы:

```
postgres=# EXPLAIN SELECT * FROM client WHERE country = 'Germany' and
language='German';
QUERY PLAN
------------------------------------------------------------------------
 Seq Scan on client (cost=0.00..1.30 rows=5 width=26)
   Filter: ((country = 'Germany'::text) AND (language = 'German'::text))
(2 rows)
```

Здесь оценка количества строк равна 5, а не 10, поскольку вычисляется она по формуле: *оценка количества строк = общее число строк * избирательность country * избирательность language = 20 * 0.5 * 0.5 = 5*. До версии PostgreSQL 10 простой способ исправить оценку заключался в том, чтобы изменить физическую структуру таблицы, объединив два поля в одно:

```
CREATE TABLE client2 (
    id serial primary key,
    name text,
    client_information jsonb
);
INSERT INTO client2(name, client_information)
```

```
SELECT generate_random_text(8),'{"country":"Germany", "language":"German"}'
FROM generate_series(1,10);

INSERT INTO client2(name, client_information)
SELECT generate_random_text(8),'{"country":"USA", "language":"English"}'
FROM generate_series(1, 10);

VACUUM ANALYZE client2;
```

Здесь country и language обернуты типом jsonb; теперь план выполнения показывает правильную оценку строк:

```
postgres=# EXPLAIN SELECT * FROM client2 WHERE client_information =
'{"country":"USA","language":"English"}';
QUERY PLAN
------------------------------------------------------------------------------
----------
 Seq Scan on client2 (cost=0.00..1.25 rows=10 width=60)
   Filter: (client_information = '{"country": "USA", "language":
"English"}'::jsonb)
(2 rows)
```

В PostgreSQL 10 добавлена перекрестная статистика столбцов, так что больше менять физическую структуру таблицы не нужно. Однако необходимо уведомить планировщик о наличии таких зависимостей. Чтобы расширить статистику, выполним такую команду:

```
CREATE STATISTICS stats (dependencies) ON country, language FROM client;
```

Чтобы увидеть новый объект statistic, сначала проанализируем таблицу, а затем опросим схему pg_catalog:

```
postgres=# analyze client;
ANALYZE
Time: 5,007 ms

postgres=# SELECT stxname, stxkeys, stxdependencies FROM pg_statistic_ext
WHERE stxname = 'stats';
 stxname | stxkeys | stxdependencies
---------+---------+----------------------------------------
 stats   | 3 4     | {"3 => 4": 1.000000, "4 => 3": 1.000000}
(1 row)
Time: 0,671 ms
```

Столбец stxkey содержит информацию о положении столбцов в таблице, в данном случае 3 and 4. Столбец stxdependencies определяет корреляционные зависимости – здесь имеется 100-процентная зависимость между language и country, и наоборот. Наконец, еще раз выполним предыдущий запрос, чтобы понять, помогла ли расширенная статистика решить проблему межстолбцовой корреляции:

```
postgres=# EXPLAIN SELECT * FROM client WHERE country = 'Germany' and
language='German';
                                QUERY PLAN
```

```
------------------------------------------------------------------------
 Seq Scan on client (cost=0.00..1.30 rows=10 width=26)
   Filter: ((country = 'Germany'::text) AND (language = 'German'::text))
(2 rows)
```

Секционирование таблиц

Секционирование таблиц помогает повысить производительность за счет физической организации данных на диске в соответствии с заданными критериями группировки. Существует два метода секционирования таблиц:

- **вертикальное секционирование** – одна таблица разбивается на несколько с целью уменьшить размер строки. Это ускоряет последовательный просмотр частичных таблиц, поскольку на одной странице помещается больше строк. Для примера предположим, что мы хотим хранить в таблице `client` фотографии клиентов в столбце типа `byte` или `blob`. Поскольку в таблице появилась большая строка, количество строк на страницу уменьшилось. Чтобы решить проблему, создадим другую таблицу, ссылающуюся на `client`:

```
CREATE TABLE client_picture (
 id int primary key,
 client_id int references client(id),
 picture bytea NOT NULL
);
```

- **горизонтальное секционирование** – применяется для того, чтобы уменьшить размер всей таблицы, распределив строки по нескольким таблицам. Этот подход реализован с помощью наследования таблиц и исключения в силу ограничений. Зачастую родительская таблица служит просто связующим звеном, а собственно данные хранятся в дочерних таблицах. Наследование таблиц позволяет достичь секционирования как такового, а исключение в силу ограничений служит для оптимизации производительности, поскольку при выполнении запроса производится доступ только к тем дочерним таблицам, которые содержат нужные данные.

Недостатки механизма исключения в силу ограничений

Иногда механизм исключения в силу ограничений не включается в работу, что приводит к очень медленным запросам. Это бывает в следующих случаях:

- исключение в силу ограничений запрещено;
- механизм не работает, если выражение во фразе `WHERE` не является сравнением на равенство или проверкой вхождения в диапазон;
- механизм работает только с неизменяемыми функциями, т. е. если предикат во фразе `WHERE` содержит функцию, которую нужно вычислять во время выполнения, например `CURRENT_TIMESTAMP`, то исключение в силу ограничений работать не будет;

- наличие сильно секционированных таблиц может снизить производительность из-за увеличения времени планирования.

Пусть мы хотим секционировать таблицу по текстовому образцу, например `pattern LIKE 'a%'`. Это можно сделать, переписав оператор `LIKE` в виде сравнения с диапазоном: `pattern >='a ' AND pattern < 'b'`. Таким образом, проверочное ограничение на дочерних таблицах будет записано не в виде `LIKE`, а с помощью диапазонов. А если пользователь выполнит команду `SELECT` с оператором `LIKE`, то ограничение в силу исключений не будет задействовано.

Переписывание запросов

Записывая запрос разными способами, разработчик учится избегать порочных практик и начинает лучше разбираться в параметрах планировщика и методах оптимизации. Запрос, который возвращает идентификаторы гуру, их имена и количество историй успеха, можно записать следующими способами:

```
postgres=# \o /dev/null
postgres=# \timing
Timing is on.
postgres=# SELECT id, name, (SELECT count(*) FROM success_story where
guru_id=id) FROM guru;
Time: 144,929 ms

postgres=# WITH counts AS (SELECT count(*), guru_id FROM success_story
group by guru_id) SELECT id, name, COALESCE(count,0) FROM guru LEFT JOIN
counts on guru_id = id;
Time: 728,855 ms

postgres=# SELECT guru.id, name, COALESCE(count(*),0) FROM guru LEFT JOIN
success_story on guru_id = guru.id group by guru.id, name ;
Time: 452,659 ms

postgres=# SELECT id, name, COALESCE(count,0) FROM guru LEFT JOIN ( SELECT
count(*), guru_id FROM success_story group by guru_id ) as counts on guru_id = id;
Time: 824,164 ms

postgres=# SELECT guru.id, name, count(*) FROM guru LEFT JOIN success_story
on guru_id = guru.id group by guru.id, name ;
Time: 454,865 ms

postgres=# SELECT id, name, count FROM guru , LATERAL (SELECT count(*) FROM
success_story WHERE guru_id=id ) as foo(count);
Time: 137,902 ms

postgres=# SELECT id, name, count FROM guru LEFT JOIN LATERAL (SELECT
count(*) FROM success_story WHERE guru_id=id ) AS foo ON true;
Time: 156,294 ms
```

Здесь вариант с `LATERAL` оказывается самым быстрым, а варианты, основанные на CTE и на соединении с подзапросом, – самыми медленными. Отметим, что результат может зависеть от характеристик оборудования, в частности жесткого диска. Конструкция `LATERAL` очень эффективна в запросах, где нужно

ограничить количество результатов, например: *найти десять лучших статей для каждого журналиста*. Кроме того, LATERAL весьма полезна при соединении с табличными функциями.

Резюме

Настройка производительности PostgreSQL – многогранная деятельность. Нужно учитывать конфигурацию оборудования, параметры сети и самого сервера PostgreSQL. Обычно PostgreSQL поставляется в конфигурации, непригодной для производственной системы. Поэтому нужно настроить, по крайней мере, параметры буферов и оперативной памяти, количество подключений и журналирование. Отметим, что некоторые параметры PostgreSQL взаимосвязаны, например параметры памяти и количество подключений. Кроме того, следует с осторожностью относиться к параметрам, после изменения которых нужно перезапускать сервер, т. к. в производственной системе это может оказаться проблематично. Как правило, PostgreSQL генерирует хороший план выполнения, если база данных нормализована и запрос правильно написан. Но так бывает не всегда. Для разрешения проблем с производительностью имеется команда EXPLAIN, которая объясняет детали плана выполнения. У этой команды несколько опций, в т. ч. ANALYZE и BUFFERS. Для написания хороших запросов необходимо знать о таких особенностях PostgreSQL, как межстолбцовая статистика, оптимизация в границах CTE и возможности PL/pgSQL. Важно также отчетливо понимать семантику таких команд SQL, как UNION, UNION ALL, DISTINCT и т. д. Учитесь записывать запросы разными способами и сравнивать их производительность.

В следующей главе мы поговорим о некоторых аспектах тестирования ПО и его особенностях в применении к базам данных. Автономные тесты для баз данных можно оформлять в виде SQL-скриптов или хранимых процедур.

Глава 14

Тестирование

Тестирование программного обеспечения – это процесс анализа программных компонентов, программ и систем с целью выявления в них ошибок и определения или проверки их технических ограничений и требований.

База данных – особая система со своими подходами к тестированию. Это связано с тем, что поведение программных компонентов базы данных (представлений, хранимых процедур, функций) может зависеть не только от их кода, но и от данных. Часто функции не являются неизменяемыми, т. е. их многократное выполнение с одними и теми же параметрами дает различные результаты.

Поэтому для тестирования модулей базы данных нужны специальные приемы. PostgreSQL предоставляет некоторые средства в этом направлении.

В архитектуре программной системы база данных обычно находится на самом нижнем уровне. Компоненты пользовательского интерфейса отображают информацию и передают команды серверным системам. Серверные системы реализуют бизнес-логику и манипулируют данными. Данные хранятся в базе. Поэтому любые изменения в схеме базы данных зачастую отражаются на многих программных компонентах. Но компания развивается, поэтому изменения необходимы. Создаются новые таблицы и поля, удаляются старые, вносятся улучшения в модель.

Разработчик должен быть уверен, что изменение структуры базы данных не «поломает» существующих приложении и что приложения смогут корректно воспользоваться новыми структурами.

В этой главе мы обсудим некоторые методы тестирования объектов базы данных. Они применимы как к реализации изменений в структуре данных сложных систем, так и к разработке интерфейсов к базе данных.

Примеры из этой главы можно выполнять в той же базе данных `car_portal`, но они не зависят от объектов этой базы. Чтобы создать базу, выполните скрипт `schema.sql`, находящийся в сопровождающем главу файле, а чтобы заполнить ее тестовыми данными – скрипт `data.sql` оттуда же. Все примеры кода можно найти в файле `examples.sql`.

Автономное тестирование

Автономное тестирование (unit testing) – это часть процесса разработки, связанная с поиском ошибок в отдельных компонентах или модулях программы.

В базе данных такими компонентами являются хранимые процедуры, функции, триггеры и т. д. Объектом автономного тестирования может быть даже определение представления или код запроса.

Идея автономного тестирования заключается в том, что для каждого модуля программной системы, например класса или функции, должен быть создан набор тестов, которые вызывают этот модуль с определенными входными данными и проверяют, совпадает ли результат вызова с ожидаемым. Если тестируемый модуль должен взаимодействовать с внешними системами, то их можно эмулировать с помощью каркаса тестирования и тем самым проверить также взаимодействие с ними.

Набор тестов должен быть достаточно велик для покрытия максимально возможной части исходного кода. Этого можно добиться, если тесты вызывают тестируемый код со всеми логически допустимыми комбинациями входных параметров. Если модуль должен как-то реагировать на недопустимые данные, то следует включать также тесты, проверяющие это. Выполнение тестов должно быть автоматизировано, чтобы можно было прогонять их при выпуске каждой новой версии тестируемого модуля.

Благодаря такому подходу мы можем быстро проверить, удовлетворяет ли модифицированное ПО старым требованиям. Это называется регрессионным тестированием.

Существует методика разработки программного обеспечения **через тестирование**. Она подразумевает, что сначала пишутся тесты, выражающие требования к компоненту, а затем код, при котором эти тесты проходят. В конечном итоге получается работающий код, покрытый тестами, и хороший набор формально определенных функциональных требований к модулю.

Специфика автономного тестирования в базе данных

Особенность автономных тестов базы данных состоит в том, что входами и выходами тестируемого модуля могут быть не только параметры функции, но и данные, хранящиеся в таблицах. Более того, выполнение одного теста может повлиять на результаты последующих вследствие изменения данных, поэтому не исключено, что тесты следует прогонять в определенном порядке.

Таким образом, каркас тестирования должен уметь вставлять данные в базу, прогонять тесты и анализировать новые данные. Кроме того, возможно, от каркаса потребуется управлять транзакциями, в контексте которых прогоняются тесты. Добиться этого будет проще всего, если оформлять тесты в виде SQL-скриптов. Во многих случаях их удобно обертывать хранимыми процедурами (в случае PostgreSQL – функциями). Эти процедуры могут храниться в той же базе, где находятся тестируемые компоненты.

Тестовые функции могут выбирать тестовые примеры из некоторой таблицы по одному. Тестовых функций может быть много, и должна существовать одна специальная функция, которая выполняет их по очереди, а затем формирует протокол испытаний.

Рассмотрим простой пример. Пусть имеется таблица в базе данных и функция, выполняющая некоторое действие над данными из этой таблицы:

```
car_portal=> CREATE TABLE counter_table(counter int);
CREATE TABLE
car_portal=> CREATE FUNCTION increment_counter() RETURNS void AS $$
  BEGIN
    INSERT INTO counter_table SELECT count(*) FROM counter_table;
  END;
$$ LANGUAGE plpgsql;
CREATE FUNCTION
```

В таблице всего один столбец, содержащий целое число. Функция подсчитывает количество записей в таблице и вставляет эту величину в ту же таблицу. Таким образом, последовательные вызовы функции приведут к вставке чисел 0, 1, 2 и т. д. Мы хотим протестировать эту функциональность. Тестовую функцию можно написать так:

```
CREATE FUNCTION test_increment() RETURNS boolean AS $$
  DECLARE
    c int; m int;
  BEGIN
    RAISE NOTICE '1..2';
   -- Отделить тестовый пример от тестового окружения
    BEGIN
    -- Тест 1. Вызвать функцию increment_counter
      BEGIN
        PERFORM increment_counter();
        RAISE NOTICE 'ok 1 - Call increment function';
      EXCEPTION WHEN OTHERS THEN
        RAISE NOTICE 'not ok 1 - Call increment function';
      END;
    -- Тест 2. Проверить результаты
      BEGIN
        SELECT COUNT(*), MAX(counter) INTO c, m FROM counter_table;
        IF NOT (c = 1 AND m = 0) THEN
          RAISE EXCEPTION 'Test 2: wrong values in output data';
        END IF;
        RAISE NOTICE 'ok 2 - Check first record';
      EXCEPTION WHEN OTHERS THEN
        RAISE NOTICE 'not ok 2 - Check first record';
      END;
      -- Откатить произведенные в тесте изменения
      RAISE EXCEPTION 'Rollback test data';
    EXCEPTION
      WHEN raise_exception THEN RETURN true;
      WHEN OTHERS THEN RETURN false;
    END;
END;
$$ LANGUAGE plpgsql;
```

Эта тестовая функция работает следующим образом:

- весь тестовый пример выполняется в собственном блоке BEGIN-EXCEPTION-END, чтобы изолировать тест от транзакции, в которой он выполняется, и дать возможность выполнить следующий тест с теми же данными;
- каждая часть тестового примера также выполняется в отдельном блоке BEGIN-EXCEPTION-END, чтобы тестирование можно было продолжить, даже если какая-то часть завершится с ошибкой;
- сначала вызывается функция increment_counter(). Эта часть считается успешной, если функция выполнилась без ошибки, и неудачной в случае возникновения любой ошибки;
- затем из таблицы выбираются данные, и проверяется, совпадают ли они с ожидаемыми. Если нет или если команда SELECT завершится с ошибкой, то тест считается неудачным;
- результаты тестирования выводятся на консоль командами RAISE NOTICE. Формат вывода следует требованиям протокола **Test Anything Protocol (TAP)**, чтобы результаты могла обработать любая тестовая обвязка (внешний каркас тестирования), например Jenkins;
- если тесты выполнены без ошибок (каковы бы ни были их результаты), то функция возвращает true, иначе false.

Запустив тестовую функцию, получим:

```
car_portal=> SELECT test_increment();
NOTICE: 1..2
NOTICE: ok 1 - Call increment function
NOTICE: ok 2 - Check first record
 test_increment
----------------
 t
(1 row)
```

Тест успешно прошел!

Допустим, что требования изменились, и в таблицу добавлено поле, содержащее время вставки значения:

```
car_portal=> ALTER TABLE counter_table ADD insert_time timestamp with time
zone NOT NULL;
ALTER TABLE
```

Внеся изменение, прогоним тест еще раз, чтобы убедиться, что функция по-прежнему работает:

```
car_portal=> SELECT test_increment();
NOTICE: 1..2
NOTICE: not ok 1 - Call increment function
NOTICE: not ok 2 - Check first record
 test_increment
----------------
 t
(1 row)
```

Тест не проходит, поскольку функция `increment_counter()` ничего не знает о новом поле. Изменим ее:

```
car_portal=> CREATE OR REPLACE FUNCTION increment_counter() RETURNS void AS
$$
BEGIN
INSERT INTO counter_table SELECT count(*), now() FROM counter_table;
END;
$$ LANGUAGE plpgsql;
CREATE FUNCTION
```

Теперь тест снова проходит:

```
car_portal=> SELECT test_increment();
NOTICE: 1..2
NOTICE: ok 1 - Call increment function
NOTICE: ok 2 - Check first record
test_increment
---------------
t
(1 row)
```

Написанная выше тестовая функция несовершенна. Во-первых, она не проверяет, действительно ли `increment_counter()` подсчитывает записи. Тест завершится успешно, даже если `increment_counter()` просто вставляет в таблицу постоянное значение 0. Чтобы исправить это, нужно запустить `increment_counter()` дважды и проверить, что получилось во второй раз.

Во-вторых, если тест не проходит, то хорошо было бы знать, в чем причина. Тестовая функция могла бы запросить эту информацию у PostgreSQL с помощью команды `GET STACKED DIAGNOSTICS` и показать ее командой `RAISE NOTICE`. Улучшенный вариант тестовой функции имеется в сопроводительном файле `examples.sql`. Он слишком длинный, поэтому здесь мы его не приводим.

В сложных программных системах очень полезно иметь автономные тесты для компонентов базы данных, поскольку зачастую эти компоненты сообща используются несколькими службами или модулями. Любые изменения, внесенные в базу в интересах одной из этих служб, могут привести к отказу остальных. Во многих случаях непонятно, какая служба каким объектом базы пользуется и как именно. Потому-то так важно иметь автономные тесты, имитирующие использование базы данных каждой внешней службой. Тогда в процессе изменения структуры данных разработчик сможет прогнать эти тесты и убедиться, что система в целом может работать с новой структурой.

Тесты можно прогонять в специально созданном тестовом окружении. В таком случае установочный скрипт должен создавать тестовые данные. Можно вместо этого прогонять тесты на копии производственной базы. Тестовый скрипт мог бы также содержать код очистки.

Фреймворки юнит-тестирования

У примера из предыдущего раздела есть и другие недостатки. Например, если тестируемая функция сама выдает предупреждения или уведомления, то они

будут портить протокол тестирования. Кроме того, тестовый код не назовешь чистым. Одни и те же куски повторяются несколько раз, блоки BEGIN-END слишком громоздкие, а протокол тестирования плохо отформатирован. Все эти вещи можно автоматизировать с помощью **каркасов автономного тестирования**.

В комплекте поставки PostgreSQL нет каркасов автономного тестирования, но их предлагает сообщество, причем сразу несколько.

Один из самых распространенных – pgtap (http://pgtap.org). Вы можете скачать его с GitHub (https://github.com/theory/pgtap/), откомпилировать и установить в тестовую базу данных. Установка в ОС не вызывает трудностей и хорошо описана в документации. Вам понадобится предварительно установить пакет postgresql-server-dev-10. Чтобы установить каркас в определенную базу данных, нужно будет создать расширение:

```
car_portal=> CREATE EXTENSION pgtap;
CREATE EXTENSION
```

Тесты пишутся в виде SQL-скриптов и могут прогоняться пакетами с помощью утилиты pg_prove, входящей в состав pgtap. Можно также писать тесты в виде хранимых функций на языке PL/pgSQL.

Каркас pgtap предоставляет пользователю набор вспомогательных функций для обертывания тестового кода. Они также записывают результаты во временную таблицу, которая впоследствии используется для формирования протокола тестирования. Например, функция ok() сообщает, что тест прошел успешно, если ее аргумент равен true, и что он завершился ошибкой – в противном случае. Функция has_relation() проверяет, существует ли в базе данных указанное отношение. Таких функций порядка сотни.

С помощью pgtap тестовый пример из предыдущего раздела можно реализовать следующим образом:

```
-- Изолировать тестовый пример в отдельной транзакции
BEGIN;
-- Сообщить, что будет выполнено 2 теста
SELECT plan(2);
-- Тест 1. Вызвать функцию инкремента
SELECT lives_ok('SELECT increment_counter()','Call increment function');
-- Тест 2. Проверить результаты
SELECT is( (SELECT ARRAY [COUNT(*), MAX(counter)]::text FROM
counter_table), ARRAY [1, 0]::text,'Check first record');
-- Сообщить о завершении
SELECT finish();
-- Откатить произведенные тестом изменения
ROLLBACK;
```

Код стал гораздо чище. Этот скрипт находится в файле pgtap.sql, сопровождающем данную главу. При запуске из psql его результаты выглядят, как показано ниже. Чтобы протокол был более компактным, включен режим Tuples only:

```
car_portal=> \t
Tuples only is on.
car_portal=> \i pgtap.sql
BEGIN
  1..2
  ok 1 - Call increment function
  ok 2 - Check first record
ROLLBACK
```

Упомянем также каркас автономного тестирования `plpgunit`. В нем тесты пишутся в виде функций на PL/pgSQL, которые вызывают предоставляемые каркасом вспомогательные функции: например, `assert.is_equal()` проверяет, что два аргумента равны. Вспомогательные функции форматируют результаты тестов и выводят их на консоль. Управляющая функция `unit_tests.begin()` вызывает все тестовые функции, записывает их результаты в таблицу и формирует протокол.

Достоинством `plpgunit` является простота. Он весит очень мало и легко устанавливается – чтобы включить каркас в свою базу, нужно выполнить всего один SQL-скрипт.

Каркас `plpgunit` можно скачать по адресу *https://github.com/mixerp/plpgunit*.

Различие схем

Иногда при работе над изменениями в схеме базы данных необходимо понять, чем старая схема отличается от новой. Анализ этой информации поможет определить, окажут ли изменения нежелательное воздействие на другие приложения и надо ли отражать их в документации.

Искать различия можно с помощью стандартных командных утилит.

Допустим, к примеру, что мы изменили структуру базы данных `car_portal`.

Сначала создадим другую базу данных, содержащую новую схему. Будем предполагать, что пользователь может обращаться к базе, работающей на локальной машине, что ему разрешено создавать базы и подключаться к любой базе. Тогда для создания новой базы по шаблону старой достаточно выполнить такую команду:

```
user@host:~$ createdb -h localhost car_portal_new -T car_portal -O car_portal_app
```

Теперь у нас имеется две одинаковые базы. Подключимся к новой и развернем в ней изменения схемы:

```
user@host:~$ psql -h localhost car_portal_new
psql (10.0)
Type "help" for help.
car_portal_new=# ALTER TABLE car_portal_app.car ADD insert_date timestamp
with time zone DEFAULT now();
ALTER TABLE
```

Теперь структуры баз различаются. Чтобы найти отличия, выгрузим схемы обеих баз в файлы:

```
user@host:~$ pg_dump -h localhost -s car_portal > old_db.sql
user@host:~$ pg_dump -h localhost -s car_portal_new > new_db.sql
```

Файлы `old_db.sql` и `new_db.sql`, созданные этими командами, имеются в приложении к данной главе. Их можно сравнить стандартными утилитами. В Linux это можно сделать командой `diff`:

```
user@host:~$ diff -U 7 old_db.sql new_db.sql
--- old_db.sql 2017-09-25 21:34:39.217018000 +0200
+++ new_db.sql 2017-09-25 21:34:46.649018000 +0200
@@ -351,15 +351,16 @@
  CREATE TABLE car (
  car_id integer NOT NULL,
  number_of_owners integer NOT NULL,
  registration_number text NOT NULL,
  manufacture_year integer NOT NULL,
  number_of_doors integer DEFAULT 5 NOT NULL,
  car_model_id integer NOT NULL,
- mileage integer
+ mileage integer,
+ insert_date timestamp with time zone DEFAULT now()
);
```

А в Windows – командой `fc`:

```
c:\dbdumps>fc old_db.sql new_db.sql
Comparing files old_db.sql and NEW_DB.SQL
***** old_db.sql
 car_model_id integer NOT NULL,
 mileage integer
);
***** NEW_DB.SQL
 car_model_id integer NOT NULL,
 mileage integer,
 insert_date timestamp with time zone DEFAULT now()
);
*****
```

В обоих случаях видно, что одна строка схемы изменилась и еще одна была добавлена. Для сравнения файлов есть много более удобных способов, например такие текстовые редакторы, как **Vim** или **Notepad++**.

Во многих случаях просто видеть различия недостаточно, а нужно еще синхронизировать две схемы. Для этой цели существуют коммерческие продукты, например *EMS DB Comparer* для PostgreSQL.

Интерфейсы абстрагирования базы данных

Если большая база данных используется многими приложениями, то иногда бывает трудно понять, кто чем пользуется и что произойдет, если схема базы изменится. С другой стороны, если база данных большая и сложная, то изменения в нее вносятся постоянно: меняются бизнес-требования, добавляются но-

вые возможности, база перерабатывается в целях лучшей нормализации и т. п.

В таком случае имеет смысл применить в системе многоуровневую архитектуру. Физическая структура расположена на нижнем уровне, и приложения не обращаются к ней напрямую.

На втором снизу уровне находятся структуры, которые абстрагируют логические сущности от их физической реализации. Они играют роли интерфейсов абстрагирования данных. Реализовать их можно по-разному. Один из способов – функции в самой базе данных. Тогда приложению разрешено только вызывать эти функции. Другой подход – использовать обновляемые представления. В этом случае приложения могут обращаться к логическим сущностям с помощью обычных SQL-команд.

Интерфейс можно также реализовать вне базы данных в виде относительно простой службы, которая обрабатывает запросы от систем верхнего уровня, запрашивает данные из базы и вносит в нее изменения. У каждого подхода есть свои плюсы и минусы:

- функции позволяют оставить логику абстрагирования данных в самой базе, их легко тестировать, но они скрывают логическую модель и иногда с трудом поддаются интеграции с приложениями верхнего уровня;
- обновляемые представления раскрывают реляционную модель (хотя не имеют внешних ключей), но реализация логики, выходящей за пределы простых команд INSERT, UPDATE и DELETE, может оказаться очень сложным и противоречащим интуиции делом;
- микросервисы, работающие вне базы данных, труднее реализовать и тестировать, зато они дают дополнительную гибкость, в т. ч. доступ к HTTP REST API.

На верхнем уровне находятся приложения, которые реализуют бизнес-логику. Им безразлична физическая структура данных, они взаимодействуют с базой только через интерфейсы абстрагирования данных.

При таком подходе уменьшается количество агентов, имеющих доступ к физическим структурам данных. Становится проще понять, как база данных используется или может использоваться. В документации по базе данных должны быть описаны все эти интерфейсы. Поэтому разработчик, работающий над рефакторингом схемы, должен только следить за тем, чтобы интерфейсы не отклонялись от спецификации, и, пока это так, может изменять в базе все, что ему заблагорассудится.

Благодаря наличию интерфейсов становится проще писать автономные тесты: ясно, что и как тестировать, поскольку имеется спецификация. Если применяется методика разработки через тестирование, то сами тесты играют роль спецификации интерфейсов.

Отличия в данных

Самый простой способ создать интерфейс абстрагирования базы данных – использовать для доступа к данным представления. Тогда если кто-то захочет

изменить структуру таблицы, то сможет сделать это, не меняя код приложений. Нужно будет только модифицировать определения интерфейсных представлений. Более того, PostgreSQL пресечет попытку удалить из таблицы столбец, используемый в каком-то представлении. Поэтому те объекты базы данных, к которым обращаются приложения, оказываются защищены.

Тем не менее если структура базы данных изменилась и определения представлений соответственно модифицированы, важно убедиться, что новые представления возвращают те же данные, что и старые.

Иногда можно реализовать новое представление в той же базе. В таком случае нужно просто создать копию производственной базы данных в тестовой среде или подготовить тестовую базу данных, содержащую все возможные комбинации атрибутов сущностей предметной области. Затем можно будет развернуть новую версию представления под другим именем и выполнить следующий запрос, чтобы убедиться, что оба представления возвращают одни и те же данные:

```
WITH
    n AS (SELECT * FROM new_view),
    o AS (SELECT * FROM old_view)
SELECT 'new', * FROM (SELECT * FROM n EXCEPT ALL SELECT * FROM o) a
UNION ALL
SELECT 'old', * FROM (SELECT * FROM o EXCEPT ALL SELECT * FROM n) b;
```

Здесь `new_view` и `old_view` – имена нового и старого представлений. Если оба представления возвращают один и тот же результат, то запрос не вернет ни одной строки.

Однако этот способ годится, только если оба представления находятся в одной и той же базе данных и старое представление работает так же, как до рефакторинга. Если же изменяется структура базовых таблиц, старое представление не может работать так, как раньше, поэтому сравнение неприменимо. Проблему можно решить, создав временную таблицу по данным, возвращенным старым представлением до рефакторинга, а затем сравнив ее с новым представлением.

Можно также сравнивать данные из разных баз – старой до рефакторинга и новой. Для этого можно применить внешние инструменты. Например, данные из обеих баз выгружаются в файлы с помощью `psql`, а затем эти файлы сравниваются посредством `diff` (это будет работать, только если порядок строк одинаковый). Существуют также коммерческие инструменты, предлагающие аналогичную функциональность.

Еще один подход – соединить две базы данных, выполнить запросы и произвести сравнение внутри базы. Может показаться сложновато, но на самом деле это самый быстрый и надежный способ. Существует два метода соединить две базы данных: расширения `dblink` и `postgres_fdw` (адаптер внешних данных).

Использование `dblink`, на первый взгляд, проще и позволяет выполнять запросы к разным объектам. Но эта технология уже устарела, применяемый

в ней синтаксис не соответствует стандартам, к тому же имеются проблемы с производительностью, особенно когда опрашиваются большие таблицы или представления.

С другой стороны, `postgres_fdw` требует создавать в локальной базе объект для каждого объекта в удаленной базе, к которому мы собираемся обращаться, что не слишком удобно. Но зато потом легко использовать в запросах удаленные таблицы наравне с локальными, и работает всё быстрее.

В примере из предыдущего раздела мы создали еще одну базу данных на основе исходной базы `car_portal` и добавили поле в таблицу `car_portal_app.car`. Давайте выясним, привела ли эта операция к каким-либо изменениям в данных. Выполните следующие действия:

1) создайте новую базу данных от имени суперпользователя:

```
user@host:~$ psql -h localhost -U postgres car_portal_new
psql (10beta3)
Type "help" for help.
car_portal_new=#
```

2) создайте расширение, реализующее адаптер внешних данных. Его двоичный код включен в состав пакета сервера PostgreSQL:

```
car_portal_new=# CREATE EXTENSION postgres_fdw;
CREATE EXTENSION
```

3) создайте объект сервера и отображение пользователей:

```
car_portal_new=# CREATE SERVER car_portal_original FOREIGN DATA
WRAPPER postgres_fdw OPTIONS (host 'localhost', dbname 'car_portal');
CREATE SERVER
car_portal_new=# CREATE USER MAPPING FOR CURRENT_USER SERVER
car_portal_original;
CREATE USER MAPPING
```

4) создайте внешнюю таблицу и проверьте, можно ли ее опросить:

```
car_portal_new=# CREATE FOREIGN TABLE car_portal_app.car_orignal
(car_id int, number_of_owners int,
   registration_number text, manufacture_year int, number_of_doors int,
   car_model_id int, mileage int)
SERVER car_portal_original OPTIONS (table_name 'car');
CREATE FOREIGN TABLE
car_portal_new=# SELECT car_id FROM car_portal_app.car_orignal limit 1;
 car_id
--------
      1
(1 row)
```

Теперь мы можем опрашивать таблицу, находящуюся в другой базе данных, так, будто это обычная таблица. Она может принимать участие в операциях соединения, фильтрации, группировки и любых других.

Итак, таблица готова, и к ней можно предъявлять запросы. Чтобы сравнить данные, мы можем использовать тот же запрос, что и раньше для старого и нового представлений:

```
car_portal_new=# WITH n AS (
    SELECT car_id, number_of_owners, registration_number, manufacture_year,
number_of_doors, car_model_id, mileage
    FROM car_portal_app.car),
        o AS (SELECT * FROM car_portal_app.car_orignal)
SELECT 'new', * FROM (SELECT * FROM n EXCEPT ALL SELECT * FROM o) a
UNION ALL
SELECT 'old', * FROM (SELECT * FROM o EXCEPT ALL SELECT * FROM n) b;
 ?column? | car_id | number_of_owners | registration_number |
manufacture_year | number_of_doors | car_model_id | mileage
----------+--------+------------------+---------------------+-------------
----+-----------------+-------------+---------
(0 rows)
```

Результат пустой. Значит, данные в обеих таблицах одинаковы.

Тестирование производительности

В отношении базы данных можно задать важный вопрос: насколько она быстрая? Сколько транзакций в секунду она может обработать, сколько времени работает конкретный запрос? Вопрос о производительности базы данных обсуждался в главе 13. Здесь мы поговорим только о том, как ее измерять.

Для измерения времени работы SQL-команды в `psql` есть метакоманда `\timing`. Если хронометраж включен, то `psql` показывает время работы каждой команды:

```
car_portal=> \timing
Timing is on.
car_portal=# SELECT count(*) FROM car_portal_app.car;
 count
-------
   229
(1 row)
Time: 0.643 ms
```

Обычно этого достаточно, чтобы понять, какой запрос быстрее и в правильном ли направлении вы движетесь в процессе оптимизации. Но не стоит полагаться на хронометраж, если вы пытаетесь оценить, сколько запросов сервер может обработать в секунду. Дело в том, что время выполнения одного запроса зависит от множества случайных факторов: текущей загрузки сервера, состояния кеша и т. д.

PostgreSQL предлагает специальную утилиту `pgbench`, которая подключается к серверу и много раз выполняет тестовый скрипт. По умолчанию `pgbench` создает собственную небольшую базу данных и гоняет в ней SQL-скрипт, имитируя работу типичного OLTP-приложения. Этого уже достаточно, чтобы понять,

насколько мощным является сервер и как влияют на производительность изменения конфигурационных параметров.

Чтобы получить более точные результаты, следует подготовить базу данных примерно такого же размера, как производственная, и тестовый скрипт, содержащий запросы, такие же или похожие на те, что выполняются в производственной системе.

Например, предположим, что база данных `car_portal` используется в веб-приложении. Типичный сценарий – запросить количество записей в таблице `car`, а затем выбрать первые 20 записей для отображения на экране. Ниже показан тестовый скрипт:

```
SELECT count(*) FROM car_portal_app.car;
SELECT * FROM car_portal_app.car INNER JOIN car_portal_app.car_model
USING (car_model_id) ORDER BY car_id LIMIT 20;
```

Этот скрипт находится в прилагаемом файле `test.sql`, предназначенном для демонстрации программы `pgbench`.

Необходимо инициализировать тестовые данные для `pgbench` (предполагается, что база данных работает на той же машине и к ней может обратиться текущий пользователь):

```
user@host:~$ pgbench -h localhost -i car_portal
NOTICE: table "pgbench_history" does not exist, skipping
NOTICE: table "pgbench_tellers" does not exist, skipping
NOTICE: table "pgbench_accounts" does not exist, skipping
NOTICE: table "pgbench_branches" does not exist, skipping
creating tables...
100000 of 100000 tuples (100%) done (elapsed 0.17 s, remaining 0.00 s).
vacuum...
set primary keys...
done.
```

Теперь запускаем тест:

```
user@host:~$ pgbench -h localhost -f test.sql -T 60 car_portal
starting vacuum...end.
transaction type: test.sql
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
duration: 60 s
number of transactions actually processed: 98314
latency average = 0.610 ms
tps = 1638.564670 (including connections establishing)
tps = 1638.887817 (excluding connections establishing)
```

Как видим, производительность составляет приблизительно 1.6 тысячи транзакций в секунду, для небольшого сайта торговли автомобилями это более чем достаточно. При этом `pgbench` работала на одной машине с сервером базы

данных, и они разделяли один процессор. С другой стороны, сетевые задержки были минимальны.

Чтобы получить более реалистичную картину, нужно запустить pgbench на той машине, где установлен сервер приложений, а количество соединений задать таким же, как в конфигурации этого сервера. Обычно для таких простых запросов время установления соединения по сети и время передачи играют большую роль, чем время обработки базой данных.

Утилита pgbench позволяет задать количество подключений к серверу базы данных и даже умеет генерировать случайные запросы. Так что это весьма полезный инструмент для оценки производительности и настройки параметров сервера.

Резюме

В этой главе мы рассмотрели ряд аспектов тестирования ПО вообще и применительно к базам данных в частности.

Методы автономного тестирования применимы к разработке кода, хранимого в базе данных, например функций или триггеров. Методика разработки через тестирование – очень хороший подход к разработке ПО. Автономное тестирование в базах данных имеет свои особенности. Автономные тесты можно писать в виде SQL-скриптов или хранимых функций. Существует несколько каркасов, помогающих писать автономные тесты и обрабатывать результаты тестирования.

Еще один аспект тестирования баз данных – сравнение данных в одной или в разных базах для проверки результатов рефакторинга модели данных. Иногда для этого можно воспользоваться SQL-запросами, а иногда приходится устанавливать соединение между базами – с помощью расширения dblinks или адаптеров внешних данных.

Схемы базы данных легко сравнить с помощью командных утилит, предоставляемых PostgreSQL и операционной системой. Существуют и более развитые программные продукты для этой цели.

Утилита pgbench, входящая в комплект поставки PostgreSQL, применяется для оценки производительности СУБД.

В следующей главе мы продолжим разговор об интеграции PostgreSQL с другими продуктами и обсудим разработку приложений базы данных на языке Python.

Глава 15

PostgreSQL в приложениях на Python

База данных – это компонент программного решения, отвечающий за хранение данных. Однако он не только хранит информацию, но и гарантирует согласованность данных, при условии что реляционная модель построена правильно. Дополнительно база данных может с помощью триггеров и правил поддержать сложную логику, выходящую за пределы нормализации. Бизнес-логика может быть реализована с помощью функций, написанных на PL/pgSQL и других языках, поддерживаемых СУБД. Стоит ли это делать, сомнительно с архитектурной точки зрения, но нас сейчас интересует не это.

Однако же задачи, относящиеся к взаимодействию с пользователями или к вводу-выводу, базе данных не под силу. Этим занимаются внешние приложения. В данной главе мы научимся подключаться к базе данных и взаимодействовать с ней из приложения, написанного на Python.

Язык **Python** очень прост для изучения и одновременно обладает колоссальной выразительной мощностью. Он активно поддерживается сообществом и располагает огромным архивом модулей, расширяющих функциональность стандартной библиотеки. Читать и понимать код, написанный на Python, очень легко. С другой стороны, язык интерпретируемый и потому довольно медленный. Типизация в Python динамическая. Это удобно, но чревато ошибками.

В этой главе рассматриваются следующие вопросы:

- Python DB API 2.0 – интерфейс прикладных программ, применяемый для работы с базами данных из приложений, написанных на Python;
- низкоуровневый доступ к базам данных с помощью `psycopg2` – самого распространенного драйвера для PostgreSQL на Python;
- альтернативные драйверы для PostgreSQL;
- концепция объектно-реляционного отображения и ее реализация в `SQLAlchemy`.

Для чтения этой главы необходимо знакомство с базовыми понятиями объектно-ориентированного программирования. Чтобы выполнять примеры, понадобится установить Python. Все примеры рассчитаны на версию Python 3.5.

Для Windows скачать установочный пакет можно с сайта https://www.Python.org. Установите его в папку, предлагаемую по умолчанию, и не забудьте также установить PIP – установщик дополнительных модулей. Путь к папке, в которую установлен Python, должен быть включен в переменную окружения PATH. Для входа в оболочку Python введите слово python в командной строке. Имеется также интерактивная оболочка Python, доступная из меню Пуск под именем idle.

Python включен в стандартные репозитории многих дистрибутивов Linux. Для установки Python и PIP выполните команду sudo aptget install python3.5 python3-pip. Чтобы войти в оболочку Python, введите в командной строке python3.

В примерах мы будем использовать ту же базу данных car_portal, что и в других главах. Скрипты для создания и заполнения базы тестовыми данными находятся в прилагаемых файлах schema.sql и data.sql. Примеры будут работать без модификации, если сервер работает на локальной машине (localhost), а пользователю car_portal_app разрешено подключаться к базе данных car_portal без пароля (с использованием метода аутентификации trust). В противном случае нужно будет изменить параметры подключения в скриптах.

Python DB API 2.0

PostgreSQL – не единственная на свете реляционная база данных и, уж конечно, не единственная, к которой возможен доступ из Python. **Python Database API** предназначен для унификации работы приложений с базами данных. К моменту написания этой книги вышла уже вторая спецификация, определенная в документе **Python Enhancement Proposal (PEP) 249**. Этот документ доступен по адресу https://www.python.org/dev/peps/pep-0249/.

В API определены следующие объекты, используемые при подключении к базе данных и взаимодействии с ней:

- connection: реализует логику подключения к серверу базы данных, аутентификации и управления транзакциями. Объект connection создается при обращении к функции connect() с параметрами, зависящими от базы данных и драйвера;
- cursor: представляет курсор в базе данных. Этот объект используется для управления контекстом выполнения SQL-команды. Например, он позволяет выполнять запрос SELECT и выбирать строки по одной или порциями. В некоторых реализациях допускается изменение позиции курсора в результирующем наборе. Объект cursor создается в результате вызова метода connect() объекта connection;
- исключения: в API определена иерархия исключений, возможных во время выполнения. Например, исключение класса IntergrityError (подкласс DatabaseError) возбуждается, когда база данных сообщает о нарушении ограничения целостности.

API упрощает переход к другому драйверу базы данных или даже вообще к другой базе. Он также помогает стандартизовать логику, а значит, сделать ее

более понятной другим разработчикам и удобной для сопровождения. Однако детали могут зависеть от драйвера базы данных.

Ниже в качестве примера приведена небольшая программа на Python, которая с помощью модуля psycopg2 выбирает данные из базы, а затем печатает их на консоли (как установить psycopg2, объяснено в следующем разделе).

```
#!/usr/bin/python3
from psycopg2 import connect

conn = connect(host="localhost", user="car_portal_app", dbname="car_portal")
with conn.cursor() as cur:
    cur.execute("SELECT DISTINCT make FROM car_portal_app.car_model")
    for row in cur:
        print(row[0])

conn.close()
```

Эта программа находится в файле print_makes.py прилагаемого архива.

Существует несколько Python-драйверов для PostgreSQL. Мы рассмотрим следующие:

- psycopg2: один из самых популярных. Пользуется «родной» библиотекой PostgreSQL libpq, поэтому очень эффективен, но недостаточно хорошо переносим;
- pg8000: этот драйвер написан на чистом Python и не зависит от libpq;
- asyncpg: высокопроизводительный драйвер, в котором для достижения максимальной производительности применяется двоичный протокол. Однако этот драйвер не реализует DB API.

Низкоуровневый доступ к базе данных с помощью psycopg2

psycopg2 – один из самых популярных драйверов для PostgreSQL, применяемых в Python-программах. Он совместим со спецификацией DB API 2.0. По большей части он написан на C и пользуется библиотекой libpq. Драйвер потокобезопасный, т. е. один и тот же объект connection может использоваться в нескольких потоках. Работает с версиями Python 2 и Python 3.

Официальная страница библиотеки – http://initd.org/psycopg/.

В Linux драйвер psycopg2 можно установить с помощью PIP из командной строки:

```
user@host:~$ sudo pip3 install psycopg2
[sudo] password for user:
Collecting psycopg2
  Downloading psycopg2-2.7.3.1-cp35-cp35m-manylinux1_x86_64.whl (2.6MB)
  100% |████████████████████████████████| 2.6MB 540kB/s
Installing collected packages: psycopg2
Successfully installed psycopg2-2.7.3.1
```

В Windows введите такую команду:

```
C:\Users\User>python -m pip install psycopg2
Collecting psycopg2
  Downloading psycopg2-2.7.3.1-cp35-cp35m-win_amd64.whl (943kB)
  100% |################################| 952kB 1.1MB/s
Installing collected packages: psycopg2
Successfully installed psycopg2-2.7.3.1
```

Разберем построчно скрипт из предыдущего раздела.

```
#!/usr/bin/python3
```

Первые два символа #!, за которыми следует путь к файлу, означают, что это скрипт, который должен интерпретироваться указанной программой. В Linux такой файл можно сделать исполняемым и просто выполнить. Оболочка Windows игнорирует эту строку, но она ничем не вредит, так что лучше оставить ее для переносимости. В данном случае строка означает, что будет вызван интерпретатор Python 3.

```
from psycopg2 import connect
```

Здесь из модуля `psycopg2` импортируется в текущее пространство имен функция `connect()`. Можно вместо этого написать предложение `import psycopg2`, но тогда импортируется весь модуль, и к функции следует обращаться по имени `psycopg.connect()`.

```
conn = connect(host="localhost", user="car_portal_app",
dbname="car_portal")
```

В этой строке устанавливается соединение с базой данных. Функция `connect()` возвращает объект `connection`, на который ссылается переменная `conn`.

```
with conn.cursor() as cur:
```

Метод соединения `cursor()` создает объект курсора. Конструкция `with ... as` создает **контекстный менеджер**. Этот прием очень удобен для выделения и освобождения ресурсов. В нашем случае создается курсор, на который указывает переменная `cur`. Как только эта переменная покидает область видимости блока with, курсор автоматически закрывается. Это произойдет независимо от того, каким способом случился выход из блока: нормально или в результате исключения.

```
    cur.execute("SELECT DISTINCT make FROM car_portal_app.car_model")
```

Выполняется запрос `SELECT DISTINCT make FROM car_portal_app.car_model`, и курсор представляет контекст его выполнения. Обратите внимание на отступы в начале этой и двух следующих строк. В Python отступы обозначают вложенность кода. В нашем случае эти три строки находятся внутри блока with. Отступы в Python заменяют фигурные скобки в языках Java и C. В руководстве по стилю оформления программ на Python рекомендуется делать отступы шириной четыре пробела. Подробнее см. страницу https://www.python.org/dev/peps/pep-008/.

```
    for row in cur:
        print(row[0])
```

Эти две строки – также часть блока with. В первой строке определен цикл for, в котором перебираются записи, возвращаемые курсором. В теле цикла на текущую строку ссылается переменная `row`. Во второй строке первое поле записи печатается на консоль. Переменная `row` имеет тип `tuple`, который в данном случае предстает в виде индексируемого, начиная с 0 массива, содержащего данные записи, а выражение `row[0]` возвращает значение первого поля.

```
conn.close()
```

Этот вызов закрывает соединение с базой данных. Обратите внимание, что в этой строке нет отступа, а значит, выполнение покинуло блок with и курсор неявно закрылся.

Теперь рассмотрим некоторые аспекты работы с `psycopg2` более подробно.

Соединение с базой данных

За соединение с базой данных отвечают объекты класса `connection`, представляющие сеанс работы. Такой объект создается в результате обращения к функции `connect()`, принадлежащей модулю `psycopg2`. Этот способ соединения определен в **DB API 2.0**.

Для задания местоположения базы данных и аутентификации в ней используется строка соединения:

```
conn = connect("host=db_host user=some_user dbname=database password=$ecreT")
```

Можно вместо этого задавать именованные параметры:

```
conn = connect(host="db_host", user="some_user", dbname="database",
               password="$ecreT")
```

Здесь устанавливается соединение с базой данных `database`, размещенной на сервере `db_host`, от имени пользователя `some_user` с паролем `$ecreT`.

Функция `connect()` принимает два необязательных параметра, `connection_factory` и `cursor_factory`. Они позволяют задать специальные функции для создания соединений или курсоров, что бывает полезно для расширения стандартной функциональности. Например, если вы хотите протоколировать все SQL-запросы, выполняемые приложением, то подойдет специализированный класс `cursor`.

В предыдущих примерах на объект соединения, возвращаемый функцией `connect`, ссылалась переменная `conn`. Этот объект используется для создания объектов курсоров и управления сеансом работы с базой. Перечислим наиболее употребительные методы класса `connection`:

- `commit()`: фиксирует текущую транзакцию. По умолчанию `psycopg2` открывает транзакцию перед выполнением любой команды. Поэтому не существует явного метода начала транзакции;
- `rollback()`: откатывает текущую транзакцию;

- `close()`: разрывает соединение с базой данных и закрывает объект `connection`;
- `cancel()`: отменяет текущую команду. Этот метод допускается вызывать из другого потока. В DB API 2.0 он не определен.

Существует также ряд методов для выполнения двухфазной фиксации. **Протокол двухфазной фиксации** применяется, когда несколько систем должны координировано фиксировать некое действие или откатить его. В классе `connection` имеется еще много полей (некоторые из них допускают только чтение) и методов, позволяющих проверить состояние соединения, получить и установить его свойства, задавать уровень изоляции транзакций и выполнять другие операции.

Объект класса `connection` можно создать, применив синтаксис контекстного менеджера:

```
with connect(host="db_host", user="some_user", dbname="database",
             password="$ecreT") as conn:
    cur = conn.cursor()
    ...
```

Когда выполнение покинет блок with любым способом – нормально или в результате исключения, – соединение будет автоматически закрыто.

Если соединение установлено в блоке with, то при нормальном выходе из блока транзакция будет зафиксирована перед закрытием соединения. Если же в блоке возникнет исключение, то транзакция будет откачена.

Пул соединений

Установление соединения с базой данных – накладный процесс. Необходимо установить соединение на сетевом уровне, включая квитирование, произвести аутентификацию и выделить новому сеансу ресурсы сервера. Представьте себе веб-приложение, которое должно было бы подключаться к базе при каждом запросе пользователя, отправлять запрос, а затем разрывать соединение. Каждая такая процедура занимает несколько миллисекунд, но когда запросов много, затраты суммируются и отнимают много времени. А если запросы поступают одновременно, то может быть создано слишком много соединений, и сервер захлебнется, т. к. под каждое соединение нужно выделять ресурсы, и даже если в сеансе ничего не происходит, все равно количество сеансов ограничено.

В веб-приложениях (и не только в них) принято сразу создавать фиксированное количество соединений, а затем использовать их для разных задач. Если задач больше, чем соединений, то они будут блокироваться, ожидая, пока какое-нибудь соединение не освободится. Это называется **пулом соединений**. У этой стратегии есть несколько преимуществ:

- не тратится время на подключение к базе данных, поскольку соединение уже создано заранее;
- количество соединений ограничено, поэтому становится проще управлять ресурсами сервера.

Модуль psycopg2 предоставляет пул соединений с простым API. Есть даже три разные реализации:

- простой пул, предназначенный для использования в однопоточных приложениях;
- многопоточный пул для использования в многопоточных приложениях;
- постоянный пул, предназначенный для работы с **Zope**, сервером веб-приложений.

К сожалению, psycopg2 не поддерживает блокировку потока, пытающегося получить соединение из уже исчерпанного пула, поэтому при работе с этим пулом нужна осторожность. Не поддерживается также синтаксис контекстного менеджера, так что работать с ним не так удобно, как могло бы быть.

В прилагаемом архиве имеется скрипт psycopg_pool.py, демонстрирующий работу с пулом соединений из psycopg2. Он слишком длинный, чтобы помещать его здесь целиком. Мы еще вернемся к этой теме, когда будем говорить о библиотеке SQLAlchemy.

Выполнение SQL-команд

Пусть соединение установлено и имеется переменная conn, ссылающаяся на экземпляр класса connection. Чтобы выполнить SQL-команду, необходимо сначала создать объект курсора, представляющий контекст команды. Это делается путем вызова метода connection.cursor():

```
cur = conn.cursor()
```

Теперь переменная cur ссылается на объект курсора. Метод cursor() может принимать следующие факультативные параметры:

- name: если имя задано, то создается курсор на стороне сервера. Разница между клиентским и серверным курсорами состоит в том, что первый обычно загружает весь результирующий набор и хранит его в памяти, а последний просит сервер выдавать данные порциями по мере готовности приложения к их обработке;
- cursor_factory: используется для создания нестандартных курсоров;
- scrollable: применим только к серверным курсорам. Если параметр равен true, то курсор можно прокручивать назад в процессе обхода результирующего набора;
- withhold: применим только к серверным курсорам. Если параметр равен true, то курсор может выбирать данные даже после фиксации транзакции (но не после ее отката).

После того как курсор создан, для выполнения SQL-команды нужно вызвать его метод cursor.execute(). Например, для выборки или удаления строк можно выполнить такой код:

```
cur.execute("SELECT * FROM car_portal_app.car_model")
cur.execute("DELETE FROM car_portal_app.car_model WHERE car_model_id = 2")
```

Если у запроса есть параметры, то их следует указать в запросе и передать значения во втором аргументе метода `cursor.execute()`. Например, вот как можно вставить запись в таблицу:

```
new_make = "Ford"
new_model = "Mustang"
sql_command = "INSERT INTO car_portal_app.car_model (make, model) " \
    "VALUES (%s, %s)"
    cur.execute(sql_command, [new_make, new_model])
```

Здесь применена **позиционная нотация**, означающая, что значения параметров должны быть перечислены в том же порядке, в каком они встречаются в строке запроса. Обратите внимание на знак \ в конце третьей строки. Так мы сообщаем интерпретатору Python, что выражение продолжается на следующей строке.

Модуль `psycopg2` поддерживает также именованные параметры. Тот же пример можно было бы записать и в таком виде:

```
new_make = "Ford"
new_model = "Mustang"
sql_command = "INSERT INTO car_portal_app.car_model (make, model) " \
    "VALUES (%(make)s, %(model)s)"
cur.execute(sql_command, {"model": new_model, "make": new_make})
```

Задание параметров в SQL-командах может показаться более сложным, чем включение их непосредственно в текст команды, например:

```
new_make = "Ford"
new_model = "Mustang"
sql_command = "INSERT INTO car_poral_app.car_model (make, model) " \
    "VALUES ('" + new_make + "', '" + new_model + "')"
cur.execute(sql_command)
```

Но это никуда не годится! Никогда не подставляйте данные прямо в SQL-команду. Это может стать причиной серьезных проблем с безопасностью. Допустим, что значением переменной `new_model` является строка, содержащая одну одиночную кавычку '. Тогда серверу будет отправлена команда `INSERT INTO car_portal_app.car_model (make, model) VALUES ('Ford', ''')` – заведомо некорректная. База данных при этом вернет ошибку. В веб-приложении такие вещи могут привести к уничтожению данных или к похищению секретов. А если мы воспользуемся параметрами, то `psycopg2` позаботится о том, чтобы правильно вставить ' в таблицу, как если бы это было название модели автомобиля.

Такая ошибка делает программу уязвимой к атаке **внедрением SQL**.

Один и тот же объект курсора можно использовать многократно для выполнения разных запросов. Однако курсоры не являются потокобезопасными, т. е. один курсор нельзя использовать из разных потоков. В таком случае нужно создавать несколько курсоров. Отметим заодно, что объект соединения потокобезопасный.

В прилагаемом архиве имеется скрипт `psycopg2_insert_data.py`, который вставляет запись в таблицу `car_model`.

Чтение данных из базы

Если на курсоре выполняется запрос, возвращающий данные, то к этим данным может получить доступ приложение. Это касается не только запросов SELECT, но и любых изменяющих данные запросов с фразой RETURNING.

Пусть имеется курсор, на котором выполняется запрос:

```
cur = conn.cursor()
cur.execute("SELECT make, model FROM car_portal_app.car_model")
```

Объект курсора предоставляет несколько способов получить данные.

- Использовать сам объект как итератор:

  ```
  for record in cur:
      print("Make: {}, model: {}".format(record[0], record[1]))
  ```

  Здесь record – кортеж, а к значениям полей можно обратиться, указав номер поля в квадратных скобках.
- Получить запись, вызвав метод курсора cursor.fetchone(). Последующие вызовы будут возвращать следующие записи. Когда записей не останется, метод вернет None:

  ```
  while True:
      record = cur.fetchone()
      if record is None:
          break
      print("Make: {}, model: {}".format(record[0], record[1]))
  ```

- Получить следующую порцию записей, вызвав метод cursor.fetchmany(). Метод принимает факультативный параметр, задающий количество записей в порции. Если он не задан, то количество записей берется из поля cursor.arraysize. Если записей больше не осталось, метод возвращает пустой набор. Например:

  ```
  while True:
      records = cur.fetchmany()
      if len(records) == 0:
          break
      print(records)
  ```

После того как данные выбраны и обработаны, курсор лучше закрыть. Это произойдет автоматически, если курсор был создан в блоке with. Незакрытые курсоры в конечном итоге будут удалены сборщиком мусора или при закрытии базы данных, но до тех пор они будут потреблять системные ресурсы.

В прилагаемом архиве имеется скрипт psycopg_query_data.py, в котором демонстрируются все три метода.

Команда COPY

Модуль psycopg2 умеет также выполнять команду COPY, чтобы быстро прочитать из базы или записать в базу большой объем данных. Методы copy_to()

и copy_from() класса cursor соответственно копируют данные из таблицы в файлоподобный объект или из файлоподобного объекта в таблицу. Имеется также метод cursor.copy_expert(), представляющий значительно более развитую функциональность, – столь же гибкий, как сама команда COPY.

Метод cursor.copy_to() принимает следующие параметры:

- file: файлоподобный объект, принимающий данные. Это может быть файл, или объект StringIO, или любой другой объект, поддерживающий метод write();
- table: имя копируемой таблицы;
- sep: разделитель полей. По умолчанию знак табуляции;
- null: текстовое представление значений NULL;
- columns: список столбцов, подлежащих копированию.

Записи в объекте file будут разделены знаками новой строки. Ниже демонстрируется применение метода cursor.copy_to():

```
import io
with io.StringIO() as s:
    cur.copy_to(table='car_portal_app.car_model', file=s)
    print(s.getvalue())
```

В результате на консоль будет выведено содержимое таблицы car_portal_app.car_model, как это сделала бы команда COPY car_portal_app.car_model TO STDOUT;, выполненная в psql.

Метод cursor.copy_from() принимает такие же параметры и дополнительно параметр size, задающий размер буфера для чтения из объекта file. Объект file должен поддерживать методы read() и readline(). Вот пример того, как вставить записи в таблицу car_portal_app.car_model с помощью команды COPY:

```
import io
with io.StringIO('Tesla\tModel-X\n) as s:
    cur.copy_from(table='car_portal_app.car_model', file=s,
                  columns=['make', 'model'])
```

В прилагаемом архиве имеется скрипт psycopg2_copy_data.py, в котором оба этих метода используются для копирования данных в базу и из базы.

Асинхронный доступ

В предыдущих примерах метод cursor.execute() блокировал программу на все время выполнения запроса. Если запрос сложный, то программа будет заблокирована надолго, что не всегда желательно. Модуль psycopg2 предоставляет способ асинхронного выполнения запросов. Это означает, что пока запрос выполняется, программа может заниматься другими делами.

Чтобы воспользоваться данной возможностью, необходимо создать асинхронное соединение. Для этого функции connect() передается аргумент async=1. Сам процесс соединения также асинхронный. Отметим, что программа должна ждать установления соединения. Это можно сделать с помощью метода connection.poll() и функции select() из библиотеки Python, являющейся оберткой

соответствующего системного вызова. Всё это довольно сложно и выходит за рамки книги. По счастью, psycopg2 предоставляет вспомогательную функцию, которая блокирует выполнение и ждет завершения асинхронной операции. Это функция psycopg2.extras.wait_select(conn), где conn – объект соединения с базой данных.

Ниже приведен упрощенный пример асинхронного выполнения запроса SELECT:

```
from psycopg2 import connect
from psycopg2.extras import wait_select

aconn = connect(host="localhost", user="car_portal_app",
                dbname="car_portal", async=1)
wait_select(aconn)
acur = aconn.cursor()

# Предполагается, что следующий запрос занимает много времени
acur.execute("SELECT DISTINCT make FROM car_portal_app.car_model")

# Здесь делается что-то еще
# ...
# Закончив другие дела, ждем завершения запроса
wait_select(aconn)
for row in acur:
    print(row[0])
acur.close()
aconn.close()
```

Более интересные примеры асинхронного выполнения имеются в файле print_makes_async.py из прилагаемого архива.

Дополнительные сведения об асинхронных операциях в psycopg2 см. на странице http://initd.org/psycopg/docs/advanced.html#asynchronoussupport.

Альтернативные драйверы для PostgreSQL

В этом разделе мы подробнее познакомимся еще с двумя драйверами для PostgreSQL. Первый, pg8000, также реализует DB API 2.0 и очень похож на psycopg2. Различие, однако, в том, что он не зависит от библиотеки libpq и написан полностью на Python. Поэтому модуль весит очень немного и использующие его приложения переносимы.

Второй, asyncpg, не реализует Python DB API, а взаимодействует с PostgreSQL по двоичному протоколу и предоставляет асинхронный API приложениям. Поэтому он позволяет создавать очень быстрые приложения, способные выполнять чрезвычайно много команд в единицу времени.

pg8000

Для установки пакета pg8000 нужно использовать pip так же, как в случае psycopg2. Вот как это выглядит в Windows:

```
c:\Users\user>python -m pip install pg8000
Collecting pg8000
  Downloading pg8000-1.11.0-py2.py3-none-any.whl
Collecting six>=1.10.0 (from pg8000)
  Downloading six-1.11.0-py2.py3-none-any.whl
Installing collected packages: six, pg8000
Successfully installed pg8000-1.11.0 six-1.11.0
```

В Linux следует выполнить команду `pip3 install pg8000`, которая печатает практически то же самое.

Поскольку библиотека также реализует DB API 2.0, используется он почти так же, как `psycopg2`. Следующая небольшая программа подключается к базе данных, отправляет запрос и выводит на консоль список марок и моделей машин, имеющихся в системе:

```
#!/usr/bin/python3

from pg8000 import connect

conn = connect(host="localhost", user="car_portal_app",
               database="car_portal")
query = "SELECT make, model FROM car_portal_app.car_model"
with conn.cursor() as cur:
    cur.execute(query)
    for record in cur:
        print("Make: {}, model: {}".format(record[0], record[1]))

conn.close()
```

Единственное отличие от `psycopg2` – имя импортируемого модуля в начале скрипта. Существуют также некоторые отличия в функциях, не являющихся частью DB API 2.0, – например, команда `COPY` в `pg8000` реализована посредством вызова метода `cursor.execute()`. Соответствие между типами данных базы и Python тоже несколько различается. `pg8000` не предлагает асинхронного API и не поддерживает пулы соединений.

Основное преимущество `pg8000` над `psycopg2` – отсутствие внешних зависимостей от драйверов PostgreSQL и библиотек. Основной недостаток – низкая производительность. На некоторых задачах `pg8000` может работать значительно медленнее.

asyncpg

Библиотека `asyncpg` не реализует Python DB API. Вместо этого она предоставляет асинхронный API, предназначенный для работы совместно с `asyncio` – библиотекой, используемой для написания конкурентного кода на Python. Изучение асинхронного программирования выходит за рамки этой книги, поэтому мы лишь приведем два упрощенных примера с пояснениями.

Как уже было сказано выше, асинхронная работа означает, что мы запускаем задачу и, пока она выполняется, занимаемся чем-то другим. Функция `asyncio.`

connect(), используемая для подключения к базе данных, асинхронна. Функции для выполнения запросов и закрытия соединения также асинхронны.

В асинхронном программировании существует понятие **обратного вызова**. Так называется функция, входящая в состав приложения, которая автоматически вызывается, когда происходит некоторое событие. Вот пример использования функции обратного вызова, которая реагирует на сообщения от базы данных:

```
import asyncio
import asyncpg

async def main():
    conn = await asyncpg.connect(host='localhost', user='car_portal_app',
                                 database='car_portal')
    conn.add_log_listener(lambda conn, msg: print(msg))
    print("Executing a command")
    await conn.execute('''DO $$ BEGIN RAISE NOTICE 'Hello'; END; $$;''')
    print("Finished execution")
    await conn.close()

asyncio.get_event_loop().run_until_complete(main())
```

Для читателей, незнакомых с асинхронными средствами Python, поясним некоторые ключевые слова и выражения, встречающиеся в этом коде:

- ключевое слово async означает, что определенная далее функция является **сопрограммой**, т. е. асинхронна и должна выполняться особым образом;
- ключевое слово await служит для синхронного выполнения сопрограмм. Если сопрограмма вызывается в предложении await, то она работает как обычная функция, т. е. выполнение продолжается только после возврата из нее;
- ключевое слово lambda определяет встраиваемую функцию. В примере выше выражение lambda conn, msg: print(msg) определяет функцию с двумя параметрами, conn и msg, которая печатает значение msg. Лямбда-выражения встречаются не только в асинхронном программировании, но в этом контексте они очень удобны в качестве коротких обратных вызовов.

Скрипт выполняет в базе данных код, написанный на PL/pgSQL. Этот код всего лишь генерирует уведомление NOTICE. В базе данных он выполняется асинхронно. Хотя скрипт ждет завершения функции conn.execute(), функция обратного вызова вызывается сразу после генерации NOTICE. Чтобы убедиться в этом, можете вставить в выражение DO задержку с помощью функции pg_sleep().

В последней строке кода вызывается функция main(). Функция определена как сопрограмма, поэтому просто так вызвать ее невозможно. В нашем примере выражение в последней строке вызывает функцию и ждет ее завершения.

Другие примеры имеются в файле asyncpg_raise_notice.py из прилагаемого архива.

Преимуществом asyncpg, по сравнению с psycopg2, является производительность. Разработчики asyncpg утверждают, что она работает примерно в три раза

быстрее psycopg2. Наличие библиотек PostgreSQL в системе не требуется. Благодаря асинхронности открывается возможность создавать очень эффективные приложения.

Недостаток же в том, что разрабатывать и отлаживать асинхронный код весьма сложно. Библиотека несовместима с DB API 2.0, поэтому другие библиотеки, использующие этот API абстрагирования базы данных, не могут с ней работать. Поскольку библиотека довольно новая, о ней пока мало что известно, и примеров недостаточно.

SQLAlchemy – библиотека объектно-реляционного отображения

Выше были описаны библиотеки низкого уровня. Чтобы их использовать, разработчик должен понимать, как работают базы данных, и знать язык SQL. С другой стороны, если база данных – всего лишь один из компонентов программного решения, обеспечивающий хранение данных, а вся логика сосредоточена в приложениях верхнего уровня, то разработчик должен заниматься бизнес-логикой, а не реализацией взаимодействия с базой данных на уровне отдельных запросов.

В приложениях верхнего уровня бизнес-объекты представлены классами и их экземплярами. Методы классов соответствуют операциям предметной области. Задачи сохранения состояния объекта в базе данных и загрузки оттуда к таким операциям не относятся.

В разработке ПО существует концепция **объектно-реляционного отображения** (object relational mapping – **ORM**). Так называется программный слой, который представляет хранящиеся в базе записи в виде экземпляров классов. При создании такого экземпляра запись вставляется в таблицу, а в результате его модификации запись обновляется.

Для Python существует библиотека SQLAlchemy, которая реализует ORM и умеет работать с разными базами данных, в т. ч. с PostgreSQL.

SQLAlchemy состоит из двух основных компонентов: *Core* и *ORM*. *Core* отвечает за взаимодействие с базой данных и выполнение команд. *ORM* работает поверх *Core* и реализует объектно-реляционное отображение. В следующих разделах кратко описаны оба компонента.

Для получения дополнительных сведений о SQLAlchemy обратитесь к официальному сайту https://www.sqlalchemy.org.

Установка SQLAlchemy производится так же, как для других продуктов, описанных в этой главе. В Windows выполните команду:

```
> python -m pip install sqlalchemy
```

А в Linux – команду:

```
$ sudo pip3 install sqlalchemy
```

Основные компоненты SQLAlchemy

Библиотека SQLAlchemy работает поверх DB API. Она может использовать для подключения к PostgreSQL как psycopg2, так и любой другой драйвер, реализующий этот API, например pg8000. В состав *Core* входит несколько компонентов SQLAlchemy.

- *Диалекты* служат для взаимодействия с конкретным драйвером базы данных. В SQLAlchemy имеются диалекты для нескольких баз данных, в т. ч. Oracle, MS SQL Server, PostgreSQL и MySQL. Для каждого диалекта необходимо установить соответствующую библиотеку.
- *Пул соединений* отвечает за установление соединений с базой данных посредством диалекта, за управление пулом соединений и за предоставление API подключения компоненту *Engine*.
- *Engine* представляет базу данных другим компонентам, выполняющим команды SQL. Это отправная точка для приложения, использующего SQLAlchemy.
- *Язык SQL Expression* – это уровень абстракции, преобразующий вызовы высокоуровневого API в команды SQL, который понимает *Engine*. Этот компонент непосредственно доступен приложениям, работающим с SQLAlchemy.
- *Schema* и *Type* – объекты, определяющие логическую модель данных. Они могут использоваться как компонентом *ORM*, так и прямо из языка SQL Expression для манипулирования данными.

Компонент *Object Relational Mapper*, или *ORM*, расположен выше компонентов *Core* и пользуется ими. Он реализует представление записей базы данных в виде экземпляров классов, определяющих сущности предметной области. *ORM* также управляет связями между классами, реализованными с помощью внешних ключей в базе данных, хотя это и необязательно.

Подключение к базе и выборка данных с помощью языка SQL Expression

Компонент *SQL Expression* служит для манипулирования данными в базе.

Чтобы подключиться к базе, нужно прежде всего создать объект engine, для чего предназначена функция create_engine():

```
from sqlalchemy import *

engine = create_engine(
    "postgresql+pg8000://car_portal_app@localhost/car_portal", echo=True)
```

Здесь мы воспользовались строкой соединения, имеющей формат dialect[+driver]://user:password@host/dbname. Параметр echo, равный True, означает, что SQLAlchemy должна протоколировать все выполненные команды SQL для отладки. В примере выше для подключения к базе использован драйвер pg8000. Заметим, что в этот момент приложение еще не подключилось к базе. Оно соз-

дало пул соединений, и, когда библиотеке SQLAlchemy понадобится выполнить команду, она запросит соединение из этого пула. Если соединение еще не существует, оно будет установлено.

Далее следует создать объект MetaData, в котором будет храниться информация о структурах данных, с которыми работает приложение:

```
metadata = MetaData()
```

Затем определим таблицу, с которой будем работать:

```
car_model = Table('car_model', metadata,
                  Column('car_model_id', Integer, primary_key=True),
                  Column('make', String),
                  Column('model', String),
                  schema='car_portal_app')
```

Теперь объект metadata знает о структуре данных. Но пока никаких взаимодействий с базой еще не было.

Чтобы реализовать логическую структуру, определенную в метаданных, нужно вызвать метод metadata.create_all(engine). Можно также загрузить физическую структуру данных в объект MetaData, это называется **отражением** (reflection). В следующем предложении в объект metadata загружается таблица car:

```
car = Table('car', metadata, schema='car_portal_app', autoload=True,
            autoload_with=engine)
```

Теперь получим из базы данные. Для этого следует получить объект соединения, выполнив метод engine.connect(), который запрашивает соединение из пула. Приложение уже подключилось к базе, когда получало информацию о таблице car. Теперь то же самое соединение используется повторно.

```
conn = engine.connect()
```

Переменная conn ссылается на объект соединения.

Для определения запроса используется объект типа Select, который создается функцией select(). Для выполнения запроса вызывается метод connection.execute():

```
query = select([car_model])
result = conn.execute(query)
```

Результатом является объект класса ResultProxy, представляющий курсор. Его можно использовать как итератор для получения записей:

```
for record in result:
    print(record)
```

Чтобы вставить новые данные, выполним такой код:

```
ins = car_model.insert()
conn.execute(ins, [
    {'make': 'Jaguar', 'model': 'XF'},
    {'make': 'Jaguar', 'model': 'XJ'}])
```

Логика похожа. Сначала создается объект типа `Insert`. Затем он выполняется на имеющемся соединении. Метод `conn.execute()` принимает список параметров запроса. `SQLAlchemy` понимает, что здесь вставляется две записи и что в списке словарей заданы значения полей `make` и `model`.

Пока что мы не написали ни строчки SQL. Однако за кулисами `SQLAlchemy` выполняет SQL-команды, потому что это единственный язык, который понимает база данных. `SQLAlchemy` позволяет легко узнать, какая SQL-команда стоит за каждым высокоуровневым объектом. Достаточно всего лишь напечатать его! Так, `print(ins)` выводит на консоль такую команду:

```
INSERT INTO car_portal_app.car_model (car_model_id, make, model) VALUES
(:car_model_id, :make, :model)
```

Язык *SQL Expression* дает возможность фильтровать результаты запроса. У объекта, представляющего запрос, имеется метод `where()`, предназначенный для фильтрации. В `SQLAlchemy` определено множество операторов и функций, соответствующих операторам и функциям SQL. Вот, например, как с помощью оператора == профильтровать строки:

```
query = select([car_model]).where(car_model.c.make == "Jaguar")
```

Этот запрос вернет только модели марки Jaguar.

В объектах запросов имеются также методы `order_by()` и `group_by()`, реализующие соответствующие фразы SQL.

Уже на этом уровне библиотека может сделать гораздо больше. Например, можно соединять таблицы, использовать подзапросы, выполнять теоретико-множественные операции (UNION) и даже вызывать оконные функции. Дополнительные сведения о языке *SQL Expression* можно почерпнуть из пособия по адресу http://docs.sqlalchemy.org/en/latest/core/tutorial.html.

Демонстрационный скрипт, выполняющий операции, описанные в этом разделе, находится в файле *sqlalchemy_sql_expression_language.py* из прилагаемого архива.

ORM

Хотя язык *SQL Expression* уже предоставляет довольно высокий уровень абстракции, он все-таки работает с физической моделью данных, а не с сущностями предметной области.

Компонент *ORM* позволяет отобразить объекты предметной области на структуры данных в базе. Для демонстрации создадим класс `Car` и определим отображение.

При работе с *ORM* мы описываем структуру базы данных и одновременно определяем классы, представляющие сущности предметной области. Это делается с помощью подсистемы `declarative`. В `SQLAlchemy` она инициализируется путем создания класса, который затем используется как базовый для других классов:

```
from sqlalchemy.ext.declarative import declarative_base
Base = declarative_base()
```

Теперь можно определить класс верхнего уровня:

```
class Car(Base):
    __tablename__ = "car"
    __table_args__ = {'schema': 'car_portal_app'}
    car_id = Column(Integer, primary_key=True)
    registration_number = Column(String, nullable=False)
    def __repr__(self):
        return "Car {}: '{}'".format(self.car_id, self.registration_number)
```

Для экономии места мы определили только два поля. Но чтобы подчеркнуть, что это сущность предметной области, а не просто определение структуры данных, мы включили в класс метод специализированного представления. Он будет вызываться, когда программа запросит текстовое представление экземпляра класса, например чтобы напечатать его. Полный код класса находится в прилагаемом файле `sqlalchemy_orm.py`.

Чтобы запросить данные из базы и получить экземпляры определенного класса в приложении, мы должны создать объект класса `Session`. Этот класс реализует сеанс – диалог с базой данных, относящийся к текущей выполняемой операции предметной области. Если приложение обращается к нескольким объектам при выполнении одной операции, то для всех используется один и тот же сеанс. Например, на сайте торговли автомобилями приложение могло бы запросить автомобили и соответствующие им модели в сеансе, где пользователь ищет автомобиль. Одновременно в другом сеансе другой пользователь мог бы создавать себе учетную запись. Во многих случаях сеанс связан с одной транзакцией.

Сеанс создается следующим образом: сначала создать класс `Session`, привязанный к созданному ранее объекту `engine`, а затем – объект этого класса:

```
from sqlalchemy.orm import sessionmaker
Session = sessionmaker(bind=engine)
session = Session()
```

Объект `session` служит для запрашивания данных. Делается это примерно так, как при использовании языка *SQL Expression*, а в процессе привязки запроса употребляются те же самые операторы и методы. Ниже показан запрос, который выбирает пять автомобилей с наименьшими идентификаторами:

```
query = session.query(Car).order_by(Car.car_id).limit(5)
```

Если этот запрос используется для выборки данных, то возвращаются экземпляры класса `Car`:

```
for car in query.all():
    print(car)
```

При печати объектов используется текстовое представление, определенное в классе, поэтому на консоли мы видим:

```
Car 1: 'MUWH4675'
Car 2: 'VSVW4565'
Car 3: 'BKUN9615'
...
```

Теперь обновим данные об одном автомобиле. Чтобы получить первый объект класса `Car`, возвращенный запросом, вызовем метод `query.first()`:

```
car = query.first()
```

Чтобы изменить регистрационный номер автомобиля, просто изменим значение поля:

```
car.registration_number = 'BOND007'
```

Атрибут этого экземпляра изменился. *ORM* знает, что этот экземпляр отображен на строку таблицы, в которой `car_id = 1`, и выполняет соответствующую команду `UPDATE` в базе данных.

Наконец, чтобы зафиксировать транзакцию и закрыть сеанс, вызовем следующие методы:

```
session.commit()
session.close()
```

Реляционная база данных состоит из отношений и связей. *ORM* не забывает и о связях. В таблице `car` имеется столбец `car_model_id`, указывающий на запись в таблице `car_model`. Логически это означает, что в классе `Car` имеется атрибут `car_model`. Физическая реализация (конкретно: хранение в другой таблице) для бизнес-логики несущественна.

Обратите внимание, что в следующих далее примерах определение класса `Car` будет изменено. `SQLAlchemy` не позволяет легко переопределять классы, отображенные на базу данных, а значит, если вы привыкли вводить код на консоли Python, то ее придется закрыть и начать заново: импортировать модуль `sqlalchemy`, инициализировать `engine` и создать классы `Base` и `Session`:

```
from sqlalchemy import *
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

engine = create_engine(
    "postgresql+pg8000://car_portal_app@localhost/car_portal", echo=True)
Base = declarative_base()
Session = sessionmaker(bind=engine)
```

Чтобы реализовать интересующую нас связь, создадим еще один класс, который будет представлять модели машин, `Car_model`. Его следует определить раньше класса `Car`:

```
class Car_model(Base):
    __tablename__ = "car_model"
    __table_args__ = {'schema': 'car_portal_app'}
    car_model_id = Column(Integer, primary_key=True)
```

```
    make = Column(String)
    model = Column(String)
```

Изменим класс `Car`: добавим еще два атрибута и переопределим метод `__repr__`:

```
from sqlalchemy.orm import relationship

class Car(Base):
    __tablename__ = "car"
    __table_args__ = {'schema': 'car_portal_app'}
    car_id = Column(Integer, primary_key=True)
    registration_number = Column(String, nullable=False)
    car_model_id = Column(Integer, ForeignKey(Car_model.car_model_id))
    car_model = relationship(Car_model)
    def __repr__(self):
        return "Car {}: {} {}, '{}'".format(
               self.car_id, self.car_model.make, self.car_model.model,
               self.registration_number)
```

Теперь еще раз выполним запрос:

```
session = Session()
query = session.query(Car).order_by(Car.car_id).limit(5)
for car in query.all():
    print(car)
```

На консоли будет напечатано:

```
Car 1: Peugeot 308, 'BOND007'
Car 2: Opel Corsa, 'VSVW4565'
Car 3: Citroen C3, 'BKUN9615'
    ...
```

ORM можно также использовать для создания новых объектов, сохранения их в базе данных и удаления.

Создадим новую модель автомобиля:

```
new_car_model = Car_model(make="Jaguar", model="XE")
session = Session()
session.add(new_car_model)
session.commit()
session.close()
```

А затем удалим ее:

```
session = Session()
old_car_model = session.query(Car_model).filter(
    and_(Car_model.make == "Jaguar", Car_model.model == "XE")).one()
session.delete(old_car_model)
session.commit()
session.close()
```

`SQLAlchemy` предоставляет очень мощный и гибкий API для работы с объектами, отображенными на таблицы базы данных. Связи можно настроить, напри-

мер так, что при удалении модели автомобиля будут удалены все автомобили этой модели, т. е. произведено каскадное удаление. В классе `Car_model` мог бы существовать атрибут `cars`, представляющий список экземпляров класса `Car`, соответствующих этой модели. Есть возможность сделать так, чтобы `SQLAlchemy` соединяла таблицу `car_model` с таблицей `car` в момент, когда объект `Car` опрашивается, – так будет работать быстрее.

Скрипт, демонстрирующий манипуляции с данными с помощью `SQLAlchemy` ORM, находится в приложенном файле `sqlalchemy_orm.py`.

Резюме

База данных часто является компонентом комплексного программного решения, отвечающим за хранение данных. Даже если бы всю сложную логику можно было реализовать в самой базе с помощью триггеров и функций, все равно остается необходимость в пользовательском интерфейсе приложения, внешнем по отношению к базе. Часто разделяют логику хранения и бизнес-логику. В таком случае бизнес-логика тоже выносится во внешнее приложение.

В этой главе мы научились подключаться к базе данных PostgreSQL из приложения, написанного на языке Python. Существуют библиотеки на Python, предоставляющие различные интерфейсы программирования. Одни из них реализуют стандартный Python DB API, другие – нет. Для Python имеются реализации ORM (объектно-реляционного отображения), прекрасно работающие с PostgreSQL.

В этой главе описаны низкоуровневые библиотеки: `psycopg2`, `pg8000` и `asyncpg`. Они умеют подключаться к базе данных из приложения и выполнять SQL-команды для выборки или модификации данных.

В последнем разделе была описана также библиотека `SQLAlchemy`, которая предоставляет средства объектно-реляционного отображения, упрощающие реализации бизнес-логики благодаря отображению сущностей предметной области на таблицы базы данных. Это позволяет разработчику сосредоточиться на логике приложения, а не на низкоуровневых деталях взаимодействия с базой данных. Хотя разработчик должен знать, что происходит в системах, с которыми он работает, и понимать, как работают базы данных, использование библиотек ORM устраняет необходимость писать SQL-код.

Из последней главы книги вы узнаете, как организовать совместную работу нескольких серверов PostgreSQL для достижения более высокой производительности, доступности или того и другого. Это сделает решение масштабируемым, способным расти вместе с компанией.

Глава 16

Масштабируемость

Последняя глава книги посвящена проблеме **масштабируемости**. Под этим понимается способность системы расти вместе с ростом компании, которая ею пользуется. PostgreSQL предоставляет некоторые средства для построения масштабируемых решений, но, строго говоря, сама СУБД PostgreSQL не масштабируема. Она может эффективно использовать ресурсы только одной машины.

- СУБД задействует несколько процессорных ядер, чтобы ускорить выполнение запроса путем распараллеливания.
- При правильной настройке она может использовать всю доступную память для кеширования.
- PostgreSQL не налагает ограничений на размер базы данных. Благодаря поддержке табличных пространств она может использовать несколько жестких дисков, а секционирование позволяет делать это быстро и прозрачно для пользователей.

Однако развертывание решения на основе базы данных на нескольких машинах проблематично, потому что стандартный сервер PostgreSQL может работать только на одной машине. Поэтому в этой главе мы будем говорить не только о самой PostgreSQL, но и о том, куда двигаться дальше.

Будут рассмотрены следующие вопросы:

- проблема масштабируемости и теорема CAP;
- физическая и логическая репликация в PostgreSQL;
- различные сценарии масштабирования и их реализация в PostgreSQL.

Это весьма обширная тема, заслуживающая отдельной книги. Поэтому мы не станем приводить много практических рецептов, а обсудим базовые концепции, чтобы вы понимали, какие есть варианты, и дальше уже изучали их самостоятельно.

Мы можем порекомендовать другие книги, выпущенные издательством Packt Publishing на эту тему:

- Hans-Jürgen Schönig «PostgreSQL Replication», Second Edition. Очень хорошо объяснены внутренние механизмы работы репликации в PostgreSQL, и приводится много практических примеров их использования;
- Shaun M. Thomas «PostgreSQL High Availability Cookbook». Обширный сборник практических рецептов построения высокодоступных решений на базе PostgreSQL с использованием средств репликации.

Примеры в этой главе предполагают, что «поднято» и настроено несколько экземпляров базы данных. Поэтому в демонстрационных сценариях мы используем композиции Docker. Если вы незнакомы с Docker, обратитесь к документации по адресу https://docs.docker.com/get-started/. Для выполнения примеров понадобится установить `docker engine` и `docker-compose`.

Примеры основаны на базе данных `car_portal`, используемой всюду в этой книге.

Проблема масштабируемости и теорема CAP

Требование о масштабируемости системы означает, что система должна без потери качества поддерживать бизнес компании в условиях его роста. Допустим, что в базе данных хранится 1 ГБ данных, и она эффективно обрабатывает 100 запросов в секунду. Что произойдет, если бизнес вырастет в 100 раз? Сможет ли система поддержать 10 000 запросов в секунду, когда объем данных вырастет до 100 ГБ? Быть может, не прямо сейчас и не в текущей конфигурации. Но в любом случае масштабируемое решение должно быть готово к расширению, чтобы обработать возросшую нагрузку.

Обычно масштабируемость неотделима от распределенной архитектуры системы. Если бы база данных могла задействовать мощности нескольких компьютеров, то для масштабирования было бы достаточно добавить компьютеры в кластер. Такие решения существуют. На самом деле многие базы данных NoSQL (даже если они реализуют SQL) являются распределенными системами. Один из самых впечатляющих примеров масштабируемого решения дает **Cassandra** – распределенная база данных NoSQL. Существуют кластеры, состоящие из десятков тысяч узлов, которые оперируют петабайтами данных и выполняют сотни миллионов операций в день. Одному серверу такое, очевидно, не под силу.

С другой стороны, эти системы менее гибки. Проще говоря, Cassandra предоставляет хранилище ключей и значений, но не обеспечивает согласованности в смысле принципов ACID. Напротив, PostgreSQL – реляционная система управления базами данных, которая поддерживает крупные конкурентные изолированные транзакции и ограничения целостности.

В информатике существует **теорема CAP**, сформулированная в 2000 году Эриком Брюэром, которая утверждает, что распределенное хранилище данных может обладать только двумя из трех свойств:

- **согласованность** (consistency) (определенная иначе, чем в принципах ACID): любая операция чтения возвращает самое актуальное состояние данных, образовавшееся после последней операции записи, независимо от того, какой узел опрашивается;
- **доступность** (availability): любой запрос к системе завершается успешно (но результат может оказаться несогласованным);
- **устойчивость к разделению** (partition tolerance): система продолжает работать, даже если некоторые ее части недоступны или кластер распался на несвязанные части.

Теорема названа по первым буквам английских названий этих свойств: CAP. Она утверждает, что обеспечить сразу три свойства невозможно. Поскольку речь идет о распределенной системе, естественно ожидать, что она может работать, когда некоторые узлы недоступны. Чтобы обеспечить согласованность, система должна координировать все операции чтения и записи, даже если они производятся в разных узлах. Чтобы гарантировать доступность, система должна хранить копии одних и тех же данных в разных узлах и синхронизировать их. И эти два свойства нельзя гарантировать одновременно, если какие-то узлы недоступны.

Cassandra гарантирует доступность и устойчивость к разделению, т. е. принадлежит классу **cAP**:

- **не согласована:** может случиться, что операция чтения, выполненная в одном узле, не увидит данных, записанных в другом узле, пока данные не будут реплицированы на все узлы, где должны находиться. И это не является причиной для блокировки операции чтения;
- **доступна:** операции чтения и записи всегда завершаются успешно;
- **устойчива к разделению:** выход из строя какого-то узла не приводит к отказу базы данных в целом. Когда отказавший узел вернется в кластер, он автоматически получит все изменения данных.

Уровень согласованности можно повысить, сделав Cassandra абсолютно согласованной, но тогда она перестанет быть устойчивой к разделению.

Реляционные базы данных, отвечающие принципам ACID, согласованы и доступны. PostgreSQL принадлежит классу **CAp**:

- **согласована:** после того как транзакция зафиксирована в одном сеансе, все остальные сеансы сразу видят результаты. Промежуточные состояния данных не видны;
- **доступна:** пока база данных работает, все корректные запросы завершаются успешно;
- **неустойчива к разделению:** если какая-то часть данных оказывается недоступной, база данных прекращает работать.

Например, для банковских операций согласованность – непререкаемое требование. При переводе денежных средств с одного счета на другой мы ожидаем, что изменены балансы обоих счетов или балансы не изменены вовсе. Если по какой-то причине приложение «падает», обновив только один счет, то вся транзакция отменяется, и данные возвращаются в предыдущее согласованное состояние. Доступность данных также важна. Что произойдет, если транзакция зафиксирована, но диск вышел из строя и вся база данных «накрылась»? Чтобы справиться с такой ситуацией, все операции должны реплицироваться в резервном хранилище, и транзакция должна считаться успешной, только если она согласована и долговечна. Перевод денежных средств в банке может занять некоторое время, потому что согласованность чрезвычайно важна, а приоритет производительности ниже.

Если онлайновая банковская система недоступна в течение какого-то времени, поскольку база данных восстанавливается из резервной копии, то клиенты могут смириться с неудобствами при условии, что не потеряют свои деньги.

С другой стороны, когда человек «лайкает» фотографию в Instagram, система отслеживает это действие в контексте фотографии и в контексте пользователя. Если какая-то из этих операций завершится неудачно, то данные окажутся несогласованными, но это не критично. Это не означает, что данные, которыми оперирует Instagram, стали менее ценными. Просто здесь требования другие. Пользователей миллионы, фотографий миллиарды, и никто не хочет стоять в очереди за обслуживанием. Некорректность некоторых лайков не имеет решающего значения, но если система в целом оказывается недоступна, пока восстанавливает корректное состояние после сбоя, то пользователь может уйти.

Таким образом, для разных требований нужны разные решения, но, к сожалению, существует естественное ограничение, не позволяющее получить все и сразу.

Репликация данных в PostgreSQL

Если требуется обеспечить наивысшую производительность, то обычно настраивают дополнительные серверы, между которыми распределяется нагрузка. Все эти серверы получают одни и те же данные от ведущего сервера. Если требуется обеспечить высокую доступность, то обычно также копируют данные на резервный сервер, чтобы он мог перехватить управление в случае отказа ведущего сервера.

Журнал транзакций

Прежде чем переходить к деталям настройки репликации, скажем несколько слов о том, как PostgreSQL обрабатывает изменение данных на нижнем уровне.

В ходе обработки команды, изменяющей данные в базе, PostgreSQL записывает новые данные на диск, чтобы они сохранились в случае отказа. Данные записываются в два места.

- В Linux файлы данных по умолчанию находятся в каталоге `/var/lib/postgresql/10/main/base`. Здесь хранятся таблицы, индексы, временные таблицы и прочие объекты. Размер этого каталога ограничен только размером диска.
- Журнал транзакций по умолчанию находится в каталоге `/var/lib/postgresql/10/main/pg_wal`. В него записывается информация о последних изменениях, внесенных в файлы данных. Его размер задается в конфигурационном файле и по умолчанию составляет приблизительно 1 ГБ.

В оба места записываются одни и те же данные. У такой кажущейся избыточности есть основательная причина.

Представьте, что некоторая транзакция вставляет в большую таблицу текстовое значение *test*, и случилось так, что в середине транзакции сервер «упал». Буквы *te* он успел записать, а остальное на диск не попало. Когда база данных «поднимется», она не сможет сказать, что запись повреждена, потому что не

знает, что должно быть в поле: две буквы *te* или слово *test*. Ну ладно, эту проблему можно решить с помощью контрольных сумм. Но как база данных будет искать поврежденную запись? Проверять все вообще контрольные суммы после неожиданного перезапуска станет очень дорого.

Но решение есть: перед тем как производить запись в файлы данных, PostgreSQL всегда пишет в журнал транзакций. **Журнал транзакций** (его также называют **журналом предзаписи**) – это список изменений, внесенных PostgreSQL в файлы данных. Он представлен набором файлов размером 16 МБ каждый (**WAL-файлов**), находящихся в подкаталоге `pg_wal` каталога, где размещена база данных. Каждый файл содержит записи, сообщающие, какой файл данных и каким образом нужно изменить. И лишь после того как журнал транзакций сохранен на диске, производится запись в файлы данных. Когда журнал транзакций заполнится, PostgreSQL удаляет самый старый сегмент, чтобы освободить место на диске. Журнал транзакций относительно невелик, так что сервер может просмотреть его в случае неожиданной остановки.

Вот что происходит в процессе перезапуска базы данных:

- если система «упала» во время записи в журнал транзакций, то PostgreSQL отыщет неполную запись журнала, поскольку не будет совпадать контрольная сумма. Эта запись будет отброшена, а транзакции, которые помещали в нее данные, будут откачены;
- если система «упала» во время записи в файлы данных, но журнал транзакций не поврежден, то PostgreSQL просмотрит весь журнал транзакций, проверит, что его содержимое помещено в файлы данных, и при необходимости исправит файлы данных. Нет нужды просматривать все файлы данных, поскольку из журнала транзакций известно, какую часть какого файла предполагалось изменить и как именно.

Процесс воспроизведения журнала транзакций называется восстановлением. Если имеется полный журнал транзакций от момента инициализации сервера до текущего момента, то можно восстановить состояние базы на любой момент времени в прошлом. Чтобы обеспечить такую функциональность, можно сконфигурировать PostgreSQL, так чтобы старые WAL-файлы не удалялись, а где-то архивировались. Тогда архив можно будет использовать для **восстановления базы данных на момент времени** на другой машине.

Физическая репликация

Записи журнала транзакций можно взять с одного сервера базы данных – ведущего – и применить к файлам данных на другом сервере – ведомом. В таком случае ведомый сервер будет иметь точную копию базы данных на ведущем сервере. Процесс передачи записей из журнала транзакций на другой сервер и их применения называется **физической репликацией**. Слово «физическая» означает, что журнал транзакций работает на низком уровне и реплика базы данных на ведомом сервере будет точной побайтовой копией базы данных на ведущем сервере.

Физическая репликация работает для всех баз данных в кластере. Создание новой базы отражается в журнале транзакций и потому реплицируется на ведомый сервер.

Трансляция журналов

Один из способов физической репликации – постоянно передавать новые WAL-файлы с ведущего сервера на ведомый и применять их там для получения синхронизированной копии базы данных. Это называется **трансляцией журналов** (log shipping).

Чтобы настроить трансляцию журналов, нужно выполнить следующие действия:

- на ведущем сервере:
  - гарантировать, что в WAL-файлах достаточно информации для репликации: в файле `postgresql.conf` установить параметр `wal_level` равным `replica` или `logical`;
  - включить архивацию WAL-файлов, присвоив параметру `archive_mode` значение `on`;
  - заставить PostgreSQL архивировать WAL-файлы в безопасное место, задав конфигурационный параметр `archive_command`. Сервер будет применять эту команду ОС к каждому WAL-файлу, нуждающемуся в архивации. Команда может, например, сжимать файл и копировать его на сетевой диск. Если команда не задана, но режим архивации WAL-файлов включен, то эти файлы будут накапливаться в каталоге `pg_wal`;
- на ведомом сервере:
  - восстановить базу из резервной копии, снятой на ведущем сервере. Проще всего сделать это с помощью программы `pg_basebackup`. Вот, например, как может выглядеть команда, выполняемая на ведомом сервере:

```
postgres@standby:~$ pg_pasebackup -D /var/lib/postgresql/10/main -h
master -U postgres
```

    Подробное описание утилиты `pg_basebackup` приведено на странице https://www.postgresql.org/docs/10/static/app-pgbasebackup.html;
  - создать файл `recovery.conf` в каталоге данных. В нем описывается, как сервер должен выполнять восстановление и должен ли он работать в качестве ведомого сервера. Чтобы сервер выступал в роли ведомого, в файле должны присутствовать как минимум такие две строки:

```
standby_mode = on
restore_command = 'cp /wal_archive_location/%f %p'
```

    Значение параметра `restore_command` зависит от положения архива WAL-файлов. Это команда ОС, которая должна скопировать WAL-файл из архива туда, где находится журнал транзакций. Дополнительные сведения о файле `recovery.conf` см. на странице https://www.postgresql.org/docs/10/static/recovery-config.html.

После того как все будет настроено и оба сервера начнут работать, ведущий сервер будет копировать все WAL-файлы в архивный каталог, а ведомый сервер будет брать их оттуда и воспроизводить в своей базе. Этот процесс продолжается, пока в файле `recovery.conf` имеется строка `standby_mode = on`.

Если ведущий сервер «упадет» и нужно будет переключиться на ведомый, его следует «повысить в чине». Это означает, что он должен прекратить восстановление и разрешить транзакции чтения-записи. Для этого достаточно просто удалить файл `recovery.conf` и перезапустить сервер. В результате ведомый сервер становится новым ведущим. После восстановления старого ведущего сервера имеет смысл сделать его ведомым, чтобы сохранить резервирование кластера. В таком случае нужно повторить описанную выше последовательность действий, поменяв серверы ролями. Ни в коем случае не запускайте сервер как ведущий, если ведомый сервер был повышен. Это может привести к различиям в репликах и в конечном итоге к потере данных!

Перечислим преимущества репликации методом трансляции журналов:

- ее сравнительно легко настроить;
- нет необходимости устанавливать соединение между ведущим и ведомым серверами;
- ведущий сервер не знает о существовании ведомого и не зависит от него;
- ведомых серверов может быть несколько. На самом деле ведомый сервер может выступать в роли ведущего для других серверов. Это называется каскадной репликацией.

С другой стороны, есть и проблемы:

- необходимо предоставить сетевой каталог для архива WAL-файлов, к которому имели бы доступ ведущий и ведомый серверы. Это значит, что нужно вовлекать в процесс третью сторону. Можно, конечно, архивировать WAL-файлы прямо на ведомом сервере, но тогда нарушилась бы симметрия, и в случае выхода из строя ведущего сервера и повышения ведомого настройка нового ведомого стала бы несколько сложнее;
- ведомый сервер может воспроизвести WAL-файл только после его архивации. Это произойдет лишь после того, как файл будет закрыт, т. е. достигнет размера 16 МБ. А значит, последние транзакции, выполненные на ведущем сервере, могут не найти отражения на ведомом, а если транзакции на ведущем сервере небольшие и происходят не слишком часто, то резервная база данных может далеко отстать от основной. Поэтому в случае сбоя часть данных может быть потеряна.

Потоковая репликация

Существует еще один вид физической репликации, который может работать поверх трансляции журналов или вообще без трансляции журналов. Это **потоковая репликация**. В подобном случае предполагается наличие соединения между ведущим и ведомым серверами, по которому ведущий посылает все записи из журнала транзакций ведомому. В результате ведомый сервер получает самые последние изменения, не дожидаясь архивации WAL-файлов.

Чтобы настроить потоковую репликацию, на ведущем сервере нужно выполнить следующие действия в дополнение к вышеупомянутым:

- в базе данных следует создать роль для репликации, например:

```
postgres=# CREATE USER streamer REPLICATION;
CREATE USER
```

- в файле `pg_hba.conf` необходимо разрешить этому пользователю подключение к виртуальной базе данных `replication`, например:

```
host replication streamer 172.16.0.2/32 md5
```

А на ведомом сервере нужно выполнить следующие действия:

- добавить в файл `recovery.conf` параметр `primary_conninfo`, описывающий подключение ведомого сервера к ведущему:

```
primary_conninfo = 'host=172.16.0.1 port=5432 user=streamer password=secret'
```

Уже и этого достаточно. Если архивирование WAL-включено и настроено, то сначала будет произведена трансляция журналов, чтобы применить все архивированные WAL-файлы к резервной базе данных. Когда файлов не останется, ведомый сервер подключится к ведущему и начнет принимать новые записи журнала транзакций прямо от сервера. Если соединение разорвется или ведомый сервер будет перезапущен, то весь процесс начнется сначала.

Существует возможность настроить потоковую репликацию вообще без архивирования WAL-файлов. Ведущий сервер по умолчанию хранит последние 64 WAL-файла в каталоге `pg_wal`. Он может отправлять их ведомому по мере необходимости. Размер каждого файла составляет 16 МБ. Это означает, что пока объем изменений файлов данных не превышает 1 ГБ, потоковая репликация может воспроизвести их в резервной базе данных без помощи архива.

Кроме того, PostgreSQL позволяет ведущему серверу узнать, какие WAL-файлы уже обработаны ведомым, а какие нет. Если ведомый работает медленно (или просто отключился), то ведущий не будет удалять необработанные файлы, даже если их число превысит 64. Это делается путем создания слотов репликации на ведущем сервере. Тогда ведомый сервер будет использовать слот репликации, а ведущий – отслеживать, какие WAL-файлы обработаны для данного слота.

Чтобы воспользоваться этой возможностью, нужно проделать на ведущем сервере следующие действия:

- создать слот репликации, выполнив в базе данных функцию `pg_create_physical_replication_slot()`:

```
postgres=# SELECT * FROM
pg_create_physical_replication_slot('slot1');
 slot_name | lsn
-----------+-----
 slot1     |
(1 row)
```

- убедиться, что значение конфигурационного параметра `max_replication_slots` в файле `postgresql.conf` достаточно велико.

А на ведомом сервере нужно:

- добавить параметр `primary_slot_name` в файл `recovery.conf`:

```
primary_slot_name = 'slot1'
```

После этого ведущий сервер не станет удалять WAL-файлы, не полученные ведомым, даже если ведомый не подключен. Когда ведомый сервер снова подключится, он получит все недостающие записи из WAL-файлов, после чего ведущий сервер удалит старые файлы.

Потоковая репликация обладает следующими преимуществами, по сравнению с трансляцией журналов:

- разрыв между ведущим и ведомым серверами меньше, поскольку записи WAL-файлов отправляются сразу после фиксации транзакции на ведущем сервере, не дожидаясь архивации;
- можно настроить репликацию вообще без архивации, тогда не будет необходимости выделять в сетевом хранилище или где-то еще место, доступное обоим серверам.

Синхронная репликация

По умолчанию потоковая репликация асинхронная. Это означает, что пользователь, зафиксировавший транзакцию, получает подтверждение немедленно, а репликация происходит спустя некоторое время. И значит, остается возможность, что транзакция, завершенная на ведущем сервере, не будет реплицирована на ведомый, если ведущий «грохнется» сразу после фиксации, не успев отправить записи из WAL-файла.

Если необходима высокая доступность в том смысле, что никакая потеря данных недопустима, то можно установить синхронный режим потоковой репликации.

Чтобы включить этот режим на ведущем сервере, нужно записать в параметр `synchronous_standby_names` в конфигурационном файле `postgresql.conf` имя ведомого сервера, например `synchronous_standby_names = 'standby1'`.

Затем на ведомом сервере то же самое имя нужно вписать в строку соединения в файле `recovery.conf`:

```
primary_conninfo = 'host=172.16.0.1 port=5432 user=streamer password=secret
application_name=standby1'
```

После этого ведущий сервер будет ждать от ведомого подтверждения успешной обработки каждой записи WAL-файла и только потом подтвердит фиксацию. Конечно, это несколько замедлит фиксацию транзакций, а если ведомый сервер отключится, то все транзакции на ведущем будут заблокированы. Но это не отказ, просто ведущий сервер будет ждать, когда ведомый снова подключится. Запросы на чтение будут при этом работать нормально.

Преимущество синхронной репликации заключается в том, что есть уверенность: если транзакция завершена и команда `COMMIT` вернула управление, то

данные реплицированы на ведомый сервер. Недостаток – снижение производительности и зависимость ведущего сервера от ведомого.

В прилагаемом архиве имеется композиция Docker, в которой реализована настройка потоковой репликации. Чтобы протестировать ее в Linux, перейдите в каталог `streaming_replication` и поднимите композицию, выполнив команду `docker-compose up`:

```
user@host:~/learning_postgresql/scalability/streaming_replication$ docker-compose up
Creating network "streamingreplication_default" with the default driver
Creating streamingreplication_master_1
Creating streamingreplication_standby_1
Attaching to streamingreplication_master_1, streamingreplication_standby_1
...
```

Теперь имеется два экземпляра базы данных, работающих в контейнерах Docker `master` и `standby`. Синхронная репликация уже включена. База данных на ведущем сервере пуста. В окно терминала будут выводиться журналы обоих серверов. Откройте еще одно окно терминала, перейдите в каталог `streaming_replication`, подключитесь к контейнеру `master`, запустите консоль `psql` и создайте базу данных `car_portal`:

```
user@host:~/learning_postgresql/scalability/streaming_replication$ dockercompose
exec master bash
root@master:/# psql -h localhost -U postgres
psql (10.0)
SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384,
bits: 256, compression: off)
Type "help" for help.
postgres=# \i schema.sql
...
car_portal=> \i data.sql
...
```

Теперь на ведущем сервере создана база данных, реплицируемая на ведомый. Проверим это. Выйдите из сеанса оболочки в контейнере `master`, подключитесь к контейнеру `standby`, запустите `psql` и выполните несколько запросов:

```
user@host:~/learning_postgresql/scalability/streaming_replication$ dockercompose
exec standby bash
root@standby:/# psql -h localhost -U car_portal_app car_portal
psql (10.0)
SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384,
bits: 256, compression: off)
Type "help" for help.
car_portal=> SELECT count(*) FROM car;
 count
------
   229
(1 row)
car_portal=> UPDATE car set car_id = 0;
ERROR: cannot execute UPDATE in a read-only transaction
```

Данные реплицируются, но в режиме чтения; сервер работает в режиме восстановления и не допускает никаких изменений.

Дополнительные сведения можно найти в документации на странице https://www.postgresql.org/docs/current/static/high-availability.html.

Логическая репликация

У физической репликации есть один маленький недостаток: синхронизируемые серверы должны иметь идентичную конфигурацию. Проще говоря, данные должны находиться в одних и тех же местах файловой системы. И еще одно – реплицируется любое изменение в файлах данных, даже если сами данные не изменяются. Например, это происходит, когда на ведущем сервере выполняется команда `VACUUM`, которая удаляет из таблиц мертвые кортежи, или команда `CLUSTER`, которая реорганизует строки. Даже если над таблицей строится индекс, ведомому серверу передается не команда `CREATE INDEX`, а содержимое индексного файла. Это создает излишнюю нагрузку на сеть и может оказаться узким местом в сильно нагруженной системе.

В случае **логической репликации** передаются не результаты SQL-команд, а сами команды. В этом случае объем данных, передаваемых по сети, гораздо меньше, а серверы не обязательно должны быть идентичными. Более того, даже структуры данных на серверах могут различаться.

Например, если на ведущем сервере выполнена команда `INSERT INTO table_a (a, b) VALUES (1, 2)` и эта команда реплицирована на ведомый, то не возникнет никакой проблемы, если в таблице на ведомом сервере имеются еще какие-то столбцы, кроме `a` и `b`. SQL-команда правильна, поэтому может быть выполнена; дополнительные столбцы получат значения по умолчанию.

PostgreSQL поддерживает логическую репликацию. Устроена она таким образом, что один и тот же сервер может получать данные от одного сервера и передавать другому. Поэтому, когда говорят о логической репликации, не употребляют термины ведущий и ведомый; существует издатель – сервер, отправляющий данные, и подписчик – сервер, который данные получает. Один и тот же сервер может быть одновременно издателем одних таблиц и подписчиком на другие. Подписчик может получать данные от разных издателей.

Логическая репликация работает на уровне отдельных таблиц или наборов таблиц. Можно также настроить ее для всех таблиц в базе данных, и тогда она автоматически распространяется на новые таблицы. Однако логическая репликация не затрагивает других объектов схемы, например последовательностей, индекса и представлений.

Чтобы настроить логическую репликацию, на стороне издателя нужно выполнить следующие действия:

- создать в базе данных роль для репликации или включить репликацию для существующего пользователя:

```
postgres=# ALTER USER car_portal_app REPLICATION;
ALTER USER
```

- в файле pg_hba.conf разрешить этому пользователю подключаться к виртуальной базе данных replication:

```
host replication car_portal_app 172.16.0.2/32 md5
```

- присвоить конфигурационному параметру wal_level в файле postgresql.conf значение logical. Это необходимо, чтобы PostgreSQL записывал в WAL-файлы достаточно информации для репликации;
- убедиться, что значение конфигурационного параметра max_replication_slots больше или равно количеству подписчиков, которые могут подключиться к серверу-издателю;
- присвоить конфигурационному параметру max_wal_senders значение, не меньшее max_replication_slots;
- создать объект публикации. Публикация – это именованный набор таблиц. Если он создан, то сервер будет отслеживать изменения в этих таблицах и передавать их любому серверу, подписавшемуся на эту публикацию. Для этой цели используется команда CREATE PUBLICATION. В примере ниже создается публикация, охватывающая все таблицы в базе данных car_portal:

```
car_portal=> CREATE PUBLICATION car_portal FOR ALL TABLES;
CREATE PUBLICATION
```

На стороне подписчика следует создать подписку. Это специальный объект, описывающий подключение к существующей публикации издателя. Подписка создается командой CREATE SUBSCRIPTION, как показано в примере ниже:

```
car_portal=# CREATE SUBSCRIPTION car_portal CONNECTION 'dbname=car_portal
host=publisher user=car_portal_app' PUBLICATION car_portal;
CREATE SUBSCRIPTION
```

Подлежащие репликации таблицы определены в публикации. Таблицы должны быть заранее созданы на стороне подписчика. В примере выше подписка car_portal подключается к публикации car_portal, которая публикует все таблицы в базе данных. Поэтому такие же таблицы должны существовать на стороне подписчика.

Как только подписка создана, репликация начинается автоматически. Сначала PostgreSQL по умолчанию реплицирует всю таблицу, а после этого асинхронно реплицирует изменения, по мере того как они появляются на стороне издателя. Если любой из серверов перезапускается, то соединение автоматически восстанавливается. Если подписчик в течение некоторого времени не подключен, то издатель запоминает все неотправленные изменения и отправляет их, как только подписчик снова подключится. Эта функциональность реализована с помощью слотов репликации, примерно так же, как в случае потоковой репликации.

Логическую репликацию можно рассматривать так, будто издатель отправляет подписчику все изменяющие данные SQL-команды, относящиеся к публикуемым таблицам. Это происходит на уровне SQL, а не на физическом

уровне. Как уже отмечалось, структуры данных на стороне издателя и подписчика могут различаться: коль скоро SQL-команды можно выполнить, все будет работать.

Логическая репликация не распространяется на последовательности, на команды DDL и на команду `TRUNCATE`. Первичные ключи, ограничения `UNIQUE` и `CHECK` на конечных таблицах принимаются во внимание, но ограничения внешнего ключа игнорируются.

Логическую репликацию, как и потоковую, можно запустить в синхронном режиме. Для этого на стороне издателя нужно задать имя подписчика в конфигурационном параметре `synchronous_standby_names` в файле `postgresql.conf`, а на стороне подписчика прописать то же имя в строке соединения при выполнении команды `CREATE SUBSCRIPTION`.

Логическая репликация обладает следующими преимуществами, по сравнению с физической:

- она легко настраивается;
- ее можно настроить очень гибко:
  - не требуется, чтобы схемы базы данных на обоих серверах были идентичны, да и вообще серверы могут быть сконфигурированы поразному;
  - один и тот же сервер может быть одновременно издателем и подписчиком;
  - одна и та же таблица может участвовать в нескольких подписках, так что в нее будут попадать данные с нескольких серверов;
  - публикацию можно настроить так, чтобы реплицировались только операции определенного типа (скажем, `INSERT` и `DELETE`, но не `UPDATE`);
  - конечная таблица на стороне подписчика доступна для записи;
- логическую репликацию теоретически можно использовать с разными основными версиями PostgreSQL (правда, пока что она поддерживается только для версии PostgreSQL 10);
- для нее не требуется стороннее программное обеспечение или оборудование, она включена в комплект поставки PostgreSQL;
- изменения данных, произведенные на физическом уровне (например, `VACUUM` или `CLUSTER`), не реплицируются.

С другой стороны, спутником гибкости неизменно является сложность. При внедрении логической репликации следует иметь в виду ряд моментов:

- не принимаются во внимание внешние ключи, поэтому конечная база данных может оказаться в несогласованном состоянии;
- если на стороне издателя схема изменяется, так что нарушается совместимость со схемой на стороне подписчика, то репликация может внезапно перестать работать;
- подписчику реплицируются только изменения, произведенные в базе данных издателя. Если кто-то внесет изменения непосредственно в базу данных подписчика, то репликация не сможет восстановить синхронизацию таблиц;

- из всех объектов схемы реплицируются только таблицы. Это может стать проблемой, если в базах данных используются автоинкрементные поля на основе последовательностей.

В прилагаемом архиве имеется композиция Docker, реализующая логическую репликацию. Для ее тестирования перейдите в каталог `logical_replication` и выполните команду `docker-compose up`. В результате будут подняты два экземпляра PostgreSQL: `publisher` и `subscriber`, и на стороне издателя будут созданы база `car_portal` и публикация. На стороне подписчика также имеется база данных, пока пустая. Далее в другом терминале запустите оболочку bash в экземпляре `subscriber` =, откройте `psql` и выполните следующие команды:

```
user@host:~/learning_postgresql/scalability/logical_replication$ dockercompose
exec subscriber bash
root@subscriber:/# psql -h localhost -U postgres car_portal
psql (10.0)
SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384,
bits: 256, compression: off)
Type "help" for help.
car_portal=# CREATE SUBSCRIPTION car_portal CONNECTION 'dbname=car_portal
host=publisher user=car_portal_app' PUBLICATION car_portal;
psql:subscriber.sql:2: NOTICE: created replication slot "car_portal" on
publisher
CREATE SUBSCRIPTION
```

В этот момент в окне первого терминала побегут сообщения о том, что все таблицы, имеющиеся в базе `car_portal`, реплицируются подписчику. Теперь выполним запрос на стороне подписчика:

```
car_portal=# select count(*) from car_portal_app.car;
 count
-------
   229
(1 row)
```

Репликация работает. Теперь можете открыть сеанс в контейнере `publisher`, запустить `psql` и вставить данные в базу. А затем проверьте, как данные реплицировались подписчику.

В каталоге `logical_replication_multi_master` есть еще один пример, который показывает, как в одну таблицу записываются изменения от разных издателей.

Дополнительные сведения о логической репликации можно найти в документации по адресу https://www.postgresql.org/docs/current/static/logical-replication. html.

Применение репликации для масштабирования PostgreSQL

Репликацию можно использовать для масштабирования в разных ситуациях. Ее основная цель – конечно, создать и поддерживать резервную базу данных

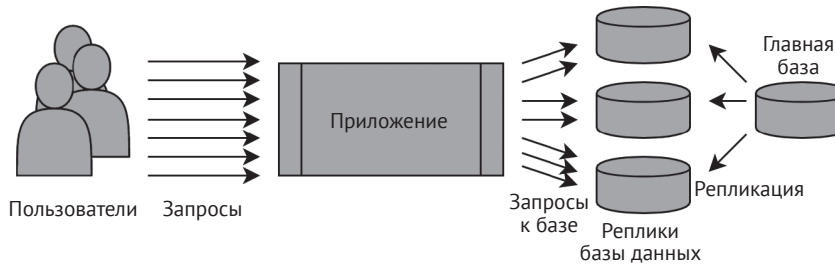
на случай выхода системы из строя. Особенно это относится к физической репликации. Однако репликацию можно использовать и для повышения производительности решения на основе PostgreSQL. Иногда для реализации комплексных сценариев масштабирования выбирают сторонние инструменты.

Масштабирование на большое количество запросов

Рассмотрим систему, рассчитанную на обработку очень большого количества запросов на чтение. Например, это может быть приложение, реализующее оконечную точку HTTP API, которая призвана поддержать автозавершение на веб-сайте. Всякий раз, как пользователь вводит символ в форму, система ищет в базе данных объекты, названия которых начинаются уже введенной строкой. Таких запросов может быть очень много, потому что одновременно работает много пользователей, каждый из которых порождает все новые запросы. Чтобы справиться с этой лавиной, база данных должна задействовать несколько процессорных ядер. Если число запросов по-настоящему велико, то количество ядер может превысить возможности одного компьютера.

То же относится к системе, которая должна обрабатывать несколько «тяжелых» запросов одновременно. Даже если запросов немного, но каждый из них сложен, задействование максимально возможного количества процессоров может дать выигрыш в производительности. Особенно если выполнение запросов распараллеливается.

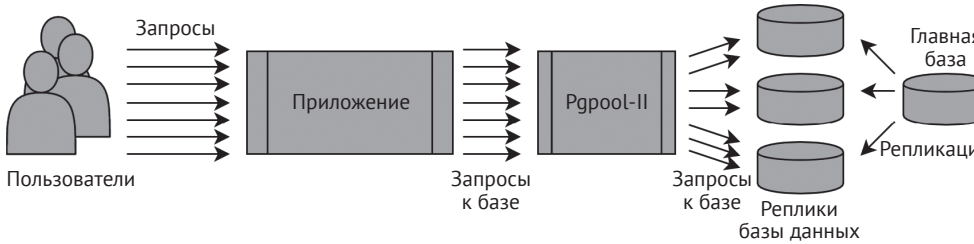
В подобных ситуациях, когда одна база данных не справляется с нагрузкой, можно завести несколько баз, настроить репликацию в них главной базы и организовать приложение так, чтобы оно направляло разные запросы разным базам. Приложение можно научить запрашивать данные у разных баз, но для этого необходима специальная архитектура уровня доступа к данным, например такая, как показана на рисунке ниже.



Приложение, опрашивающее несколько реплик базы данных

Другой вариант предлагает программа **Pgpool-II**, он может работать как балансировщик нагрузки, поставленный перед несколькими базами данных PostgreSQL. Эта программа предоставляет SQL-интерфейс, так что приложения могут работать так, будто это настоящий сервер PostgreSQL. А программа

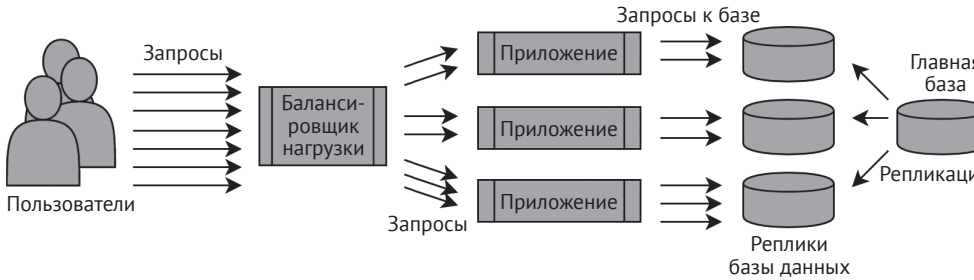
направляет запросы базам, у которых в данный момент меньше всего запросов, т. е. балансирует нагрузку.



Применение Pgpool-II для балансировки нагрузки между серверами баз данных

Дополнительные сведения о Pgpool-II см. на сайте http://www.pgpool.net.

Еще одна возможность – масштабировать приложение вместе с базами данных, так чтобы каждый экземпляр приложения подключался к своему экземпляру базы данных. В таком случае пользователь должен был бы подключаться к одному из нескольких экземпляров, что легко реализовать с помощью балансировщика HTTP-нагрузки:



Разделение данных

Если проблема заключается не в количестве одновременных запросов, а в размере базы данных и скорости выполнения одного запроса, то нужен другой подход. Данные можно разделить между несколькими серверами, которые будут опрашиваться параллельно, а объединение частичных результатов в окончательный будет производиться за пределами базы данных. Эта идея называется **разделением данных** (data sharding).

PostgreSQL предлагает метод разделения, основанный на секционировании таблиц, когда секции размещаются на разных серверах, а главный сервер рассматривает их как внешние таблицы. Мы уже касались темы секционирования в главе 8, посвященной OLAP, а темы внешних таблиц – в главе 14. Видя запрос, адресованный родительской таблице, главный сервер может проанализировать фразу `WHERE` и определения секций, чтобы понять, какие секции содержат

запрашиваемые данные, и только их и опрашивать. Иногда, в зависимости от запроса, соединение и агрегирование можно выполнить на удаленных серверах. К сожалению, PostgreSQL не может опрашивать различные секции параллельно. Но работа над этим ведется, и, быть может, в будущих версиях такая возможность появится. Но и то, что есть сейчас, уже позволяет построить решение, в котором приложения будут подключаться к одной базе данных, а та физически перенаправлять запросы к различным базам в зависимости от того, какие данные запрашиваются. Если из различных сеансов запрашиваются разные данные и секционирование проектировалось с учетом этого, то выигрыш в производительности налицо.

Алгоритмы разделения данных можно встроить и в приложения на основе PostgreSQL. Суть в том, что приложение должно знать, в какой базе какие данные находятся. Это значительно усложняет приложение.

Еще одна возможность – воспользоваться одним из имеющихся на рынке решений с разделением данных: коммерческих или с открытым исходным кодом. У всех них свои плюсы и минусы, но есть одна общая черта – они основаны на старых версиях PostgreSQL и не используют последние новшества (иногда предлагая вместо них собственные средства).

Одно из самых популярных решений на основе разделения данных – **Postgres-XL** (https://www.postgres-xl.org), в котором реализована архитектура без совместно используемых ресурсов с несколькими серверами PostgreSQL. Система состоит из нескольких компонентов:

- несколько узлов данных: используются для хранения данных;
- единственный **глобальный монитор транзакций** (**ГМТ**): управляет кластером, обеспечивая глобальную согласованность транзакций;
- несколько **узлов-координаторов**: поддерживают подключение пользователей, построение планов выполнения запросов и взаимодействие с ГМТ и узлами данных.

Postgres-XL предоставляет такой же API, как PostgreSQL, поэтому приложению не нужно обращаться к нему каким-то особым образом. Решение совместимо с ACID, т. е. поддерживает транзакции и ограничения целостности. Команда `COPY` тоже поддерживается.

Перечислим основные достоинства Postgres-XL:

- чтение масштабируется посредством добавления узлов данных;
- запись масштабируется посредством добавления узлов-координаторов;
- текущая версия Postgres-XL (на момент написания книги) основана на версии PostgreSQL 9.5, относительно новой. Уже доступна альфа-версия Postgres-XL, основанная на PostgreSQL 10.

Основной недостаток программы Postgres-XL состоит в том, что в нее не встроены средства обеспечения высокой доступности. С увеличением количества серверов в кластере возрастает и вероятность отказа любого из них. Поэтому нужно аккуратно создавать резервные копии или самостоятельно организовать репликацию узлов данных.

Postgres-XL поставляется с открытым исходным кодом, но предлагается также коммерческая поддержка.

Упомянем также программу **Greenplum** (http://greenplum.org). Она позиционируется как реализация массово-параллельной базы данных, предназначенной специально для хранилищ данных. Система состоит из следующих компонентов:

- **главный узел:** управляет подключением пользователей, строит планы выполнения запросов, управляет транзакциями;
- **узлы данных:** хранят данные и выполняют запросы.

Greenplum также реализует PostgreSQL API, так что приложение может подключаться к базе данных Greenplum без каких-либо изменений. Транзакции поддерживаются, но поддержка ограничений целостности ограничена. Команда `COPY` поддерживается.

Перечислим основные достоинства Greenplum:

- чтение масштабируется посредством добавления узлов данных;
- поддерживается столбцовая организация таблиц, что полезно для хранилищ данных;
- поддерживается сжатие данных;
- средства обеспечения высокой доступности встроены изначально. Можно (и рекомендуется) добавить вспомогательный главный сервер, который будет перехватывать управление в случае отказа основного. Можно также включить зеркала узлов данных для предотвращения потери данных.

Но есть и недостатки:

- запись не масштабируется. Любая операция записи проходит через единственный главный сервер – и добавление узлов данных ничего не даст;
- в основе лежит PostgreSQL 8.4. В Greenplum реализовано множество улучшений и новых возможностей, но по существу это все-таки очень старая версия;
- Greenplum не поддерживает внешние ключи, а поддержка ограничений уникальности ограничена.

Существует коммерческое и открытое издание Greenplum.

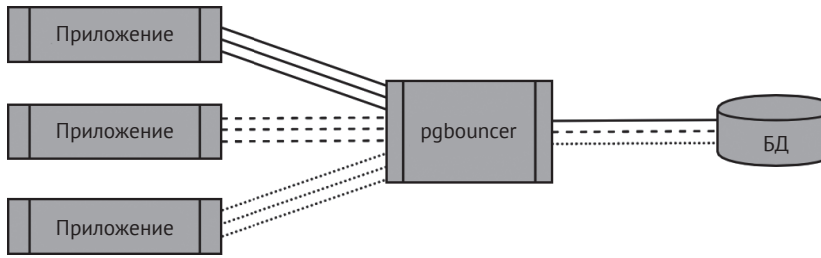
Масштабирование с ростом числа подключений

Еще один вид масштабирования возникает, когда число подключений к базе данных очень велико. В главе 15 мы уже говорили о пуле соединений: почему он необходим и в чем его преимущества. Но когда одна база данных используется в среде, где работает множество микросервисов и у каждого есть свой пул соединений, то даже если они в совокупности выполняют не так уж много запросов, количество подключений может достигать сотен, а то и тысяч. Каждое подключение потребляет ресурсы сервера, так что одно лишь требование поддерживать громадное число соединений может составить проблему, даже если не принимать во внимание запросы.

Если бы приложения не использовали пул соединений, а открывали бы соединения, только когда нужно опросить базу данных, а потом сразу закрывали, то возникла бы другая проблема. Установление соединения с базой данных занимает время. Не очень много, но когда число таких операций огромно, издержки могут оказаться весьма заметными.

Существует инструмент **PgBouncer** (https://PgBouncer.github.io/), который реализует функциональность пула соединений. Он может принимать запросы на подключение от приложений так, будто является сервером PostgreSQL, и открывать ограниченное число соединений с базой данных. Он повторно использует одни и те же соединения для различных приложений. Установление соединения с PgBouncer происходит гораздо быстрее, чем с настоящей базой, потому что у PgBouncer нет необходимости инициализировать серверный процесс, который будет обслуживать сеанс.

Схема работы PgBouncer представлена на рисунке ниже:



PgBouncer в роли пула соединений

PgBouncer устанавливает несколько соединений с базой данных. Когда приложение подключается к PgBouncer и начинает транзакцию, PgBouncer выделяет ему существующее соединение, перенаправляет все SQL-команды базе данных и возвращает результаты приложению. По завершении транзакции PgBouncer отбирает соединение у приложения, но не закрывает его. Если другое приложение начнет транзакцию, то может быть использовано то же самое соединение.

Pgbouncer может также выделять приложению соединение в начале каждой SQL-команды. Это полезно, когда приложение отправляет только запросы на чтение, но число таких запросов велико. Еще один вариант – выделить соединение на все время жизни приложения. При этом сокращаются накладные расходы на установление соединений, но количество самих соединений не уменьшается.

Резюме

В этой главе мы обсудили проблемы создания масштабируемых приложений на основе PostgreSQL, когда задействуются ресурсы нескольких серверов. У та-

ких систем есть естественное ограничение – невозможно одновременно обеспечить производительность, надежность и согласованность. Можно улучшить какой-то один аспект, но остальные при этом пострадают.

PostgreSQL предлагает несколько способов организовать репликацию, когда поддерживается актуальная копия базы данных на другом сервере или нескольких серверах. Репликацию можно использовать для создания резервной копии или резервного сервера, который берет на себя управление в случае отказа основного. С помощью репликации можно также повысить производительность программной системы, распределив нагрузку между несколькими серверами баз данных.

Иногда возможностей репликации, предлагаемых PostgreSQL, недостаточно. Существуют сторонние решения, реализующие дополнительную функциональность. Например, Pgpool-II играет роль пула соединений, а PgBouncer работает как балансировщик нагрузки. Есть и более сложные решения на основе кода PostgreSQL, способные хранить части данных в разных узлах и опрашивать их одновременно. Тем самым создается многосерверная распределенная база данных, способная обрабатывать огромные массивы данных и справляться с очень высокой нагрузкой. Мы упомянули Postgres-XL и Greenplum, но есть и другие решения – коммерческие и с открытым исходным кодом.

Предметный указатель

У

Ф

Х

Ш

Я

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Салахалдин Джуба, Андрей Волков

Изучаем PostgreSQL 10

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Слинкин А. А.*
Корректоры *Шемяк С. Н., Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Докучаева А. Е.*

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать офсетная.
Усл. печ. л. 32,5. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**