

Прочитав эту книгу, вы научитесь создавать динамичные корпоративные приложения на основе последней версии СУБД PostgreSQL, которая позволяет аналитикам без труда проектировать физические и технические аспекты системной архитектуры.

Приводится введение в новые функциональные возможности PostgreSQL, благодаря которым можно разрабатывать эффективные и отказоустойчивые приложения. Подробно описываются передовые средства PostgreSQL, включая логическую репликацию, кластеры баз данных, оптимизацию производительности, мониторинг и управление пользователями. Не оставлен без внимания оптимизатор запросов PostgreSQL, конфигурирование СУБД для достижения высокого быстродействия и переход с Oracle на PostgreSQL. Рассматриваются транзакции, блокировка, индекса и оптимизация запросов. Кроме того, описана настройка сетевой безопасности, организация резервного копирования и репликации. В конце книги приведены сведения о полезных расширениях PostgreSQL.

Краткое содержание книги:

- передовые средства и возможности SQL в PostgreSQL 11;
- индексирование в PostgreSQL и оптимизация запросов;
- хранимые процедуры;
- резервное копирование и восстановление;
- репликация и методы отработки отказов;
- поиск и устранение неполадок с обзором типичных и не слишком типичных проблем;
- переход с MySQL и Oracle на PostgreSQL.

PostgreSQL 11 Мастерство разработки

Ганс-Юрген Шёниг



PostgreSQL 11 Мастерство разработки

Интернет -магазин:
www.dmkpress.com

Оптовая продажа:
КТК «Галактика»
e-mail: books@aliaskniga.ru

Packt
ДМК
ИЗДАТЕЛЬСТВО
www.дмк.рф

ISBN 978-5-97060-671-1

9 785970 606711 >

**Повысьте мастерство разработчика
в новейшей версии PostgreSQL**

ДМК
ИЗДАТЕЛЬСТВО

Ганс-Юрген Шениг

PostgreSQL 11

Мастерство разработки

Hans-Jürgen Schönig

Mastering PostgreSQL 11

Second Edition

*Expert techniques to build scalable, reliable,
and fault-tolerant database applications*



Ганс-Юрген Шениг

PostgreSQL 11

Мастерство разработки

Второе издание

*Как специалисты создают масштабируемые,
надежные и отказоустойчивые приложения базы данных*



Москва, 2019

УДК 004.65
ББК 32.972.134
Ш47

Шениг Г.-Ю.

Ш47 PostgreSQL 11. Мастерство разработки / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2019. – 352 с.: ил.

ISBN 978-5-97060-671-1

Книга описывает последние возможности PostgreSQL 11 для построения эффективных и отказоустойчивых приложений.

Подробно рассмотрены передовые аспекты PostgreSQL, включая логическую репликацию, кластеры баз данных, оптимизацию производительности, мониторинг и управление пользователями, процесс миграции с Oracle на PostgreSQL.

Издание рекомендовано ведущими специалистами в области PostgreSQL в России, будет полезно администраторам и разработчикам этой СУБД.

УДК 004.65
ББК 32.972.134

Authorized Russian translation of the English edition of Mastering PostgreSQL 11, Second Edition ISBN 9781789537819 © 2018 Packt Publishing.

This translation is published and sold by permission of Packt Publishing, which owns or controls all rights to publish and sell the same.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN 978-1-78953-781-9 (анг.)
ISBN 978-5-97060-671-1 (рус.)

© 2018 Packt Publishing
© Оформление, издание, перевод, ДМК Пресс, 2019

Содержание

Об авторе	11
О рецензенте	12
Предисловие	13
Глава 1. Обзор PostgreSQL	16
Что нового в PostgreSQL 11.0?	16
Новые средства администрирования базы данных	17
Усовершенствования в индексировании и оптимизации	18
Улучшенное управление кешем	20
Улучшенные оконные функции	20
Добавление JIT-компиляции	21
Улучшенное секционирование	21
Поддержка хранимых процедур	22
Улучшение команды ALTER TABLE	23
Резюме	24
Вопросы	24
Глава 2. Транзакции и блокировка	25
Работа с транзакциями в PostgreSQL	25
Обработка ошибок внутри транзакции	27
Использование команды SAVEPOINT	28
Транзакционные DDL-команды	29
Основы блокировки	30
Предотвращение типичных ошибок и явная блокировка	31
Использование фраз FOR SHARE и FOR UPDATE	33
Уровни изоляции транзакций	36
SSI-транзакции	38
Взаимоблокировки и смежные вопросы	38
Рекомендательные блокировки	40
Оптимизация хранилища и управление очисткой	41
Настройка VACUUM и autovacuum	41
Наблюдение за работой VACUUM	43
Ограничение длительности транзакций с помощью времени жизни снимка	46
Резюме	47
Вопросы	47
Глава 3. Использование индексов	49
Простые запросы и стоимостная модель	49
Использование команды EXPLAIN	50

Стоимостная модель PostgreSQL.....	52
Развертывание простых индексов	54
Сортировка результатов	55
Эффективное использование просмотра по битовой карте	57
Разумное использование индексов	57
Повышение быстродействия с помощью кластеризованных таблиц.....	59
Кластеризация таблиц	62
Просмотр только индекса.....	62
Дополнительные свойства B-деревьев.....	63
Комбинированные индексы	63
Добавление функциональных индексов	64
Уменьшение занятого места на диске	65
Добавление данных во время индексирования.....	66
Введение в классы операторов	67
Создание класса операторов для B-дерева	68
Типы индексов в PostgreSQL.....	73
Хеш-индексы	73
GiST-индексы.....	73
GIN-индексы.....	76
SP-GiST-индексы	77
BRIN-индексы.....	77
Добавление новых типов индексов	79
Получение более точных ответов с помощью нечеткого поиска.....	80
Расширение pg_trgm и его достоинства	80
Ускорение запросов с предикатом LIKE	82
Регулярные выражения	83
Полнотекстовый поиск.....	83
Сравнение строк.....	84
Определение GIN-индексов	85
Отладка поиска.....	86
Сбор статистики по словам	87
О пользе операторов исключения	87
Резюме.....	88
Вопросы.....	88

Глава 4. Передовые средства SQL	90
Введение в наборы группирования.....	90
Загрузка тестовых данных.....	90
Применение наборов группирования	91
Сочетание наборов группирования с фразой FILTER.....	94
Использование упорядоченных наборов.....	95
Гипотетические агрегаты.....	97
Оконные функции и аналитические средства	97
Разбиение данных.....	98
Упорядочение данных внутри окна.....	99
Скользящие окна.....	100
Абстрагирование окон.....	102

Использование встроенных оконных функций	103
Создание собственных агрегатов	109
Создание простых агрегатов	109
Добавление поддержки параллельных запросов	112
Повышение эффективности	113
Написание гипотетических агрегатов	114
Резюме	116

Глава 5. Журналы и статистика системы

Сбор статистических данных о работе системы	117
Системные представления в PostgreSQL	117
Создание файлов журналов	134
Конфигурационный файл postgresql.conf	134
Резюме	139
Вопросы	139

Глава 6. Оптимизация запросов для достижения максимальной производительности

Что делает оптимизатор	141
Оптимизация на примере	142
Разбираемся в планах выполнения	151
Систематический подход к планам выполнения	151
Выявление проблем	153
Соединения: осмысление и исправление	157
Как соединять правильно	157
Обработка внешних соединений	158
Параметр join_collapse_limit	159
Включение и выключение режимов оптимизатора	160
Генетическая оптимизация запросов	163
Секционирование данных	164
Создание секций	164
Применение табличных ограничений	166
Модификация наследуемой структуры	167
Перемещение таблицы в наследуемую структуру и из нее	168
Очистка данных	168
Секционирование в PostgreSQL 11.0	169
Настройка параметров для повышения производительности запросов	171
Ускорение сортировки	173
Ускорение административных задач	175
Распараллеливание запросов	176
Что PostgreSQL умеет делать параллельно?	179
Распараллеливание на практике	179
Введение в JIT-компиляцию	180
Настройка JIT	181
Выполнение запросов	181
Резюме	183

Глава 7. Написание хранимых процедур	184
Языки хранимых процедур	184
Фундаментальные основы – хранимые процедуры и функции	185
Анатомия функции	186
Языки хранимых процедур	189
Введение в PL/pgSQL	190
Создание хранимых процедур на PL/pgSQL	204
Введение в PL/Perl	205
Введение в PL/Python	211
Улучшение функций	214
Уменьшение числа вызовов функций	214
Резюме	218
Вопросы	218
Глава 8. Безопасность в PostgreSQL	220
Управление сетевой безопасностью	220
Подключения и адреса привязки	221
Файл pg_hba.conf	224
Безопасность на уровне экземпляра	228
Задание безопасности на уровне базы данных	232
Задание прав на уровне схемы	233
Работа с таблицами	235
Задание прав на уровне столбцов	236
Задание привилегий по умолчанию	237
Безопасность на уровне строк	238
Просмотр прав	242
Передача объектов и удаление пользователей	243
Резюме	245
Вопросы	245
Глава 9. Резервное копирование и восстановление	246
Простая выгрузка	246
Запуск pg_dump	247
Задание пароля и информации о подключении	248
Извлечение подмножества данных	250
Форматы резервной копии	250
Восстановление из резервной копии	252
Сохранение глобальных данных	253
Резюме	253
Вопросы	254
Глава 10. Резервное копирование и репликация	255
Что такое журнал транзакций	255
Знакомство с журналом транзакций	256
Контрольные точки	257
Оптимизация журнала транзакций	257

Архивация и восстановление журнала транзакций	259
Настройка архивации	259
Конфигурирование файла <code>pg_hba.conf</code>	260
Создание базовой резервной копии	261
Воспроизведение журнала транзакций	263
Очистка архива журналов транзакций	267
Настройка асинхронной репликации	268
Базовая настройка	268
Остановка и возобновление репликации	270
Проверка состояния репликации для обеспечения доступности	271
Отработка отказов и линии времени	274
Управление конфликтами	275
Повышение надежности репликации	276
Переход на синхронную репликацию	277
Настройка долговечности	279
Слоты репликации	280
Работа с физическими слотами репликации	281
Работа с логическими слотами репликации	283
Использование команд <code>CREATE PUBLICATION</code> и <code>CREATE SUBSCRIPTION</code>	285
Резюме	287
Вопросы	287

Глава 11. Полезные расширения 289

Как работают расширения	289
Проверка доступных расширений	290
Использование модулей из подборки contrib	293
Модуль <code>adminpack</code>	293
Применение фильтра Блума	294
Установка <code>btree_gist</code> и <code>btree_gin</code>	296
Dblink – пора расстаться	297
Доступ к файлам с помощью <code>file_fdw</code>	298
Анализ хранилища с помощью <code>pageinspect</code>	299
Анализ кеша с помощью <code>pg_buffercache</code>	301
Шифрование данных с помощью <code>pgcrypto</code>	302
Прогрев кеша с помощью <code>pg_prewarm</code>	302
Анализ производительности с помощью <code>pg_stat_statements</code>	304
Анализ хранилища с помощью <code>pgstattuple</code>	304
Нечеткий поиск с помощью <code>pg_trgm</code>	305
Подключение к удаленному серверу с помощью <code>postgres_fdw</code>	305
Другие полезные расширения	309
Резюме	310

Глава 12. Поиск и устранение неполадок 311

Первоначальное изучение незнакомой базы данных	311
Анализ результатов <code>pg_stat_activity</code>	311
Опрос <code>pg_stat_activity</code>	312
Выявление медленных запросов	314

Анализ отдельных запросов	315
Углубленный анализ с помощью perf	316
Анализ журнала	317
Анализ наличия индексов.....	318
Анализ памяти и ввода-вывода.....	318
О конкретных ошибочных ситуациях.....	320
Повреждение clog.....	320
Что означают сообщения о контрольной точке	321
Что делать с поврежденными страницами данных.....	322
Беззаботное управление подключениями	323
Борьба с разбуханием таблиц.....	323
Резюме.....	324
Вопросы.....	324
Глава 13. Переход на PostgreSQL.....	325
Перенос команд SQL в PostgreSQL.....	325
Латеральные соединения	325
Наборы группирования	326
Фраза WITH – общие табличные выражения	327
Фраза WITH RECURSIVE.....	328
Фраза FILTER.....	328
Оконные функции.....	329
Упорядоченные наборы – фраза WITHIN GROUP	329
Фраза TABLESAMPLE.....	330
Ограничение выборки и смещение	331
Фраза OFFSET	331
Темпоральные таблицы	332
Сопоставление с образцом во временных рядах.....	332
Переход с Oracle на PostgreSQL.....	332
Использование расширения oracle_fdw для переноса данных	333
Использование ora2pg для перехода с Oracle.....	334
Распространенные подводные камни	336
ora_migrator – быстрая миграция Oracle в PostgreSQL	337
Переход из MySQL или MariaDB на PostgreSQL	338
Обработка данных в MySQL и MariaDB.....	339
Перенос данных и схемы	343
Резюме.....	345
Предметный указатель	346

Об авторе

Ганс-Юрген Шениг 18 лет работает с PostgreSQL. Он является генеральным директором компании Cybertec Schönig and Schönig GmbH, оказывающей поддержку и консультационные услуги пользователям PostgreSQL. Компания успешно обслуживает бесчисленных заказчиков по всему миру. Перед тем как основать Cybertec Schönig and Schönig GmbH в 2000 году, он трудился разработчиком баз данных в частной исследовательской компании, изучавшей австрийский рынок труда, где занимался в основном добычей данных и прогностическими моделями. Написал несколько книг о PostgreSQL.

О рецензенте

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 International, оказывающей финансовые услуги и расположенной в Чикаго. В свободное время увлекается искусством, особенно музыкой, и путешествует со своей женой Мэрилин.

Предисловие

Второе издание этой книги призвано помочь вам в создании динамических решений на основе базы данных для корпоративных приложений, где в качестве базы используется последняя версия PostgreSQL, позволяющая аналитикам без труда проектировать физические и технические аспекты системной архитектуры.

Книга начинается введением в последние возможности PostgreSQL 11, которые дают возможность строить эффективные и отказоустойчивые приложения. Мы подробно рассмотрим передовые аспекты PostgreSQL, включая логическую репликацию, кластеры баз данных, оптимизацию производительности, мониторинг и управление пользователями. Мы также покажем, как пользоваться оптимизатором PostgreSQL, настраивать базу, так чтобы она работала максимально быстро, и как перейти с Oracle на PostgreSQL. В процессе чтения книги мы познакомимся с транзакциями, блокировкой, индексами и оптимизацией запросов.

Кроме того, вы научитесь настраивать сетевую безопасность и использовать резервные копии и репликацию. Мы также расскажем о полезных расширениях PostgreSQL, позволяющих увеличить производительность при работе с большими базами данных.

Прочитав эту книгу до конца, вы сможете выжать из своей базы все до капли благодаря высокотехнологичной реализации административных задач.

На кого рассчитана эта книга

В этой книге рассмотрены средства, включенные в PostgreSQL 11, и показано, как создавать более качественные приложения PostgreSQL и эффективно администрировать базу данных PostgreSQL. Вы освоите передовые возможности PostgreSQL и приобретете навыки, необходимые для создания эффективных решений.

Краткое содержание книги

Глава 1 «Обзор PostgreSQL» содержит введение в PostgreSQL и сведения о новых средствах, появившихся в PostgreSQL 11.

В главе 2 «Транзакции и блокировка» показано, как наиболее эффективно использовать транзакции в PostgreSQL.

В главе 3 «Использование индексов» обсуждаются индексы – их типы, сценарии использования и реализация собственной стратегии индексирования.

Глава 4 «Передовые средства SQL» посвящена современному SQL и его возможностям. Мы рассмотрим различные типы множеств и вопрос о написании собственных агрегатов.

В главе 5 «Журналы и статистика системы» объясняется, как извлечь полезный смысл из статистики базы данных.

В главе 6 «Оптимизация запросов для достижения максимальной производительности» показано, как писать более быстрые запросы. Мы также поговорим о том, что делает запрос плохим.

В главе 7 «Написание хранимых процедур» мы рассмотрим различия между процедурами и функциями. Обсуждаются также хранимые процедуры, использование расширений и некоторые продвинутые возможности PL/pgSQL.

В главе 8 «Безопасность в PostgreSQL» описаны типичные проблемы безопасности, с которыми сталкивается разработчик или администратор базы данных PostgreSQL.

Глава 9 «Резервное копирование и восстановление» посвящена восстановлению базы данных из резервной копии, а также вопросу о частичной выгрузке данных.

В главе 10 «Резервное копирование и репликация» мы рассмотрим журнал транзакций в PostgreSQL и объясним, как работать с ним, чтобы повысить качество и безопасность СУБД.

В главе 11 «Полезные расширения» обсуждаются некоторые широко распространенные расширения PostgreSQL.

Темой главы 12 «Поиск и устранение неполадок» являются анализ неизвестной базы данных, выявление узких мест, решение проблем, возникающих при повреждении системы хранения, и инспекция неработающих реплик.

Глава 13 «Переход на PostgreSQL» посвящена переходу с других СУБД на PostgreSQL.

КАК ВЫЖАТЬ МАКСИМУМ ИЗ ЭТОЙ КНИГИ

Книга написана для широкой аудитории. Для проработки приведенных примеров хорошо бы иметь некоторый опыт работы с SQL вообще и с PostgreSQL в частности (хотя это требование необязательно). Не мешает также знакомство с командной строкой UNIX.

ГРАФИЧЕСКИЕ ВЫДЕЛЕНИЯ

В этой книге применяется ряд соглашений о графическом выделении.

Код в тексте: зарезервированные слова, имена таблиц базы данных, имена папок и файлов, URL-адреса, данные, которые вводит пользователь, и адреса в Твиттере, например: «Я добавлю в таблицу одну строку с помощью простой команды INSERT».

Команды и их результаты набраны таким шрифтом:

```
test=# CREATE TABLE t_test (id int);
CREATE TABLE
test=# INSERT INTO t_test VALUES (0);
INSERT 0 1
```

Новые термины, важные фрагменты, а также элементы графического интерфейса в меню или диалоговых окнах набраны **полужирным шрифтом**, например «Выберите пункт **System info** на панели **Administration**».



Так будут оформляться предупреждения и важные примечания.



Так будут оформляться советы или рекомендации.

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 оставив 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.

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 принять меры.

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

Обзор PostgreSQL

Прошло уже немало времени с тех пор, как я написал последнюю книгу о PostgreSQL. Я прошел долгий путь и рад рассказать обо всем, что узнал, в этом издании «Осваиваем PostgreSQL», где освещены крутые новшества, вошедшие в PostgreSQL 11.

PostgreSQL – одна из самых передовых систем баз данных с открытым исходным кодом, в ней много возможностей, широко используемых как разработчиками, так и системными администраторами. А в версию PostgreSQL 11 добавлен ряд вещей, способных сделать этот исключительный продукт еще успешнее.

В данной книге мы подробно рассмотрим и обсудим многие из них.

Эта глава представляет собой введение в PostgreSQL и появившиеся в версии PostgreSQL 11 новшества. Детально описывается часть новой функциональности. Учитывая огромное количество изменений и сам размер проекта PostgreSQL, представленный список, конечно, далеко не полон, я старался выделить наиболее важные моменты, интересные большинству пользователей.

Рассматриваемые в этой главе средства можно разбить на несколько категорий:

- что нового в PostgreSQL 11;
- средства, относящиеся к SQL и к разработке;
- резервное копирование, восстановление и репликация;
- производительность.

Что нового в PostgreSQL 11.0?

Версия PostgreSQL 11, вышедшая осенью 2018 г., предлагает пользователям современные средства, полезные равно профессионалам и начинающим. PostgreSQL 11 – вторая основная версия, нумеруемая в соответствии с новой схемой, принятой сообществом PostgreSQL. Следующая основная версия будет иметь номер 12. Сервисные версии будут называться **PostgreSQL 11.1, 11.2, 11.3** и т. д. Это существенное изменение, по сравнению со схемой нумерации, действовавшей до выхода версии 10, поэтому на него следует обратить внимание.

Какую версию использовать? Рекомендуется работать с самой последней. Нет никакого смысла начинать работу, скажем, с версии PostgreSQL 9.6. Если вы только приступаете к использованию PostgreSQL, берите версию 11. В PostgreSQL не бывает дефектов – сообщество всегда дает вам работающий код, поэтому бояться PostgreSQL 10 или PostgreSQL 11 не надо. Они работают – и все тут.

Новые средства администрирования базы данных

В PostgreSQL 11 много новых средств, помогающих снизить нагрузку на администратора и сделать систему более надежной и стабильной.

Одно из таких средств – возможность задавать размер **сегмента WAL**.

Задание размера сегментов WAL

Первая версия PostgreSQL вышла 20 лет назад, и с тех пор размер одного файла предзаписи (файла WAL) был равен 16 МБ. В самом начале это ограничение даже было зашито в код, но впоследствии стало параметром времени компиляции. Начиная с версии PostgreSQL 11 размер сегментов WAL можно задавать в момент создания экземпляра, что дает администратору дополнительный рычаг конфигурирования и оптимизации PostgreSQL. В следующем примере показано, как задать размер сегмента WAL в момент запуска `initdb`:

```
initdb -D /pgdata --wal-segsize=32
```

Команда `initdb` создает экземпляр базы данных. Обычно этот вызов скрыт в каком-то скрипте операционной системы: вашего любимого дистрибутива Linux, Windows или иной. Теперь у `initdb` появился параметр, позволяющий задать требуемый размер сегмента WAL.

Как уже было сказано, по умолчанию размер равен 16 МБ, но в большинстве случаев для повышения производительности имеет смысл его увеличить. Использовать меньший размер стоит, только если вы работаете с совсем крохотной базой данных во встраиваемой системе.

Какого влияния на производительность следует ожидать? Как всегда, все зависит от того, что именно вы делаете. Если 99 % операций, выполняемых базой данных, – чтение, то эффект увеличения размера сегментов WAL будет нулевым. Да-да, вы не ослышались – **НУЛЕВЫМ**. Если операции записи выполняются, когда система простаивает 95 % времени и не подвергается серьезной нагрузке, то эффект будет нулевым или близким к тому. Выигрыш имеет место, только если в составе рабочей нагрузки преобладают операции записи. Лишь в этом случае размер стоит изменять. А если речь идет о парочке онлайн-форм, заполняемых случайным посетителем сайта, то к чему огород городить? Эта новая возможность заиграет во всю силу, только когда в базе производится много изменений и, стало быть, в WAL пишется много данных.

Увеличенный `queryid` в `pg_stat_statements`

Если вы хотите серьезно заняться производительностью PostgreSQL, то стоит приглядеться к представлению `pg_stat_statements`. Лично я считаю его бесцен-

ным подспорьем для всякого, кто хочет знать, что происходит в системе. Модуль `pg_stat_statements` загружается в момент запуска PostgreSQL, если прописан в параметре `shared_preload_libraries`, и собирает статистические сведения о запросах, выполняемых сервером. Представление сразу покажет, если что-то пошло не так.

В представлении `pg_stat_statements` имеется поле `queryid`, которое до сих пор содержало 32-разрядный идентификатор (внутренний хеш-код). Иногда это приводило к проблемам из-за коллизии ключей. Магнус Хагандер (Magnus Hagander) в одной из своих статей подсчитал, что после выполнения 3 млрд запросов следует ожидать примерно 50 000 коллизий. После перехода на 64-разрядный `queryid` это количество уменьшается до примерно 0.25 конфликта на 3 млрд запросов, что следует считать значительным улучшением.

Имейте в виду, что если вы используете в своих скриптах представление `pg_stat_statements` для диагностики проблем с производительностью, то после перехода на PostgreSQL 11 эти скрипты нужно будет обновить.

Усовершенствования в индексировании и оптимизации

PostgreSQL 11 предлагает целый спектр улучшенных функций для администрирования. Не остались в стороне и индексы. К ним как раз относятся одни из самых важных усовершенствований.

Статистика индекса по выражению

Для оптимизации простого запроса PostgreSQL анализирует внутреннюю статистику. Рассмотрим пример:

```
SELECT * FROM person WHERE gender = 'female';
```

В этом случае PostgreSQL смотрит на внутреннюю статистику и оценивает количество женщин в таблице `person`. Если оно мало, то PostgreSQL будет использовать индекс, а если большая часть записей относится к женщинам, то PostgreSQL предпочтет последовательный просмотр. Статистика собирается по каждому столбцу. Кроме того, в PostgreSQL 10 добавлена возможность собирать межстолбцовую статистику (посмотрите справку по команде `CREATE STATISTICS`). Хорошая новость состоит в том, что PostgreSQL может также собирать статистику для функциональных индексов:

```
CREATE INDEX idx_cos ON t_data (cos(data));
```

Но до сих пор было невозможно использовать более сложную статистику по функциональным индексам.

Рассмотрим пример индекса по нескольким столбцам:

```
CREATE INDEX coord_idx ON measured (x, y, (z + t));  
ALTER INDEX coord_idx ALTER COLUMN 3 SET STATISTICS 1000;
```

Здесь мы имеем индекс по двум физическим столбцам и еще одному виртуальному столбцу, представленному выражением. Новая возможность позволяет явно собрать больше статистики для третьего столбца, который в про-

тивном случае был бы покрыт неоптимально. В этом примере мы говорим PostgreSQL, что по третьему столбцу в системной статистике должно быть 1000 записей. Это позволит оптимизатору точнее вычислить оценку и, значит, создать более качественный план – весьма полезное прибавление эффективности в некоторых специализированных приложениях.

Покрывающие индексы

Во многих других системах баз данных давно уже существуют **покрывающие индексы**¹. Что это значит? Рассмотрим следующий пример, где мы просто выбираем из таблицы два столбца:

```
SELECT id, name FROM person WHERE id = 10;
```

Предположим, что имеется индекс по столбцу `id`. В таком случае PostgreSQL найдет ключ в этом индексе, а остальные поля прочтает из таблицы. Это называется просмотром индекса (*index scan*) и включает обращение и к индексу, и к базовой таблице для формирования строки. Раньше для решения проблемы нужно было бы создать индекс по двум столбцам, что позволило бы PostgreSQL выполнить просмотр только индекса (*index-only scan*), а не просмотр индекса. Если индекс содержит все необходимые столбцы, то дополнительных обращений к таблице не нужно (как правило).



Выбирайте только столбцы, которые действительно необходимы, иначе все может закончиться бессмысленным обращением к таблице. Запросы типа показанного ниже в общем случае считаются плохими с точки зрения производительности: `SELECT * FROM person WHERE id = 10;`

Проблема возникает, если `id` должен быть первичным ключом, но при этом хочется, чтобы при чтении дополнительного столбца выполнялся просмотр только индекса. Здесь-то и приходит на помощь новая возможность:

```
CREATE UNIQUE INDEX some_name ON person USING btree (id) INCLUDE (name);
```

PostgreSQL гарантирует, что столбец `id` будет уникальным, но при этом в индексе будет храниться дополнительное поле, так что при запросе обоих столбцов будет произведен просмотр только индекса. Если рабочая нагрузка преимущественно типа OLTP, то производительность может резко возрасти. Разумеется, точные оценки привести трудно, потому что таблицы и запросы у всех разные.

Параллельное построение индексов

Традиционно для построения индекса в PostgreSQL сервер использовал только одно процессорное ядро. Но PostgreSQL используется во все более крупных системах, поэтому создание индексов становилось камнем преткновения. И тогда сообщество начало думать о том, как улучшить сортировку. Первый шаг – раз-

¹ Автор использует термин «покрывающие индексы» как синоним INCLUDE-индексов. В другом распространенном значении «покрывающим» называется любой индекс, содержащий все необходимые для запроса значения. – *Прим. ред.*

решить параллельное построение **В-деревьев** – уже реализован в PostgreSQL 11. В будущих версиях PostgreSQL параллельная сортировка будет разрешена и для обычных операций (в версии 11 это, к сожалению, еще не поддерживается).

Параллельное построение может существенно ускорить создание индексов, и мы с нетерпением ждем дальнейших прорывов в этой области (например, поддержки для индексов других типов).

Улучшенное управление кешем

PostgreSQL 11 предлагает дополнительные способы управления кешем ввода-вывода (разделяемыми буферами). Особенно стоит отметить команду `pg_rewasm`.

Усовершенствованная команда `pg_rewarm`

Команда `pg_rewarm` позволяет восстановить содержимое кеша ввода-вывода PostgreSQL после перезапуска. Она существует уже довольно давно и широко применяется пользователями PostgreSQL. В версии 11 `pg_rewarm` дополнена и теперь допускает автоматическую выгрузку списка буферов с регулярными интервалами.

Можно также автоматически загрузить старое содержимое кеша, чтобы наблюдаемая производительность не ухудшалась после перезапуска. От этих улучшений выиграют системы с большим объемом оперативной памяти.

Улучшенные оконные функции

Оконные функции и средства аналитики – краеугольный камень любой современной реализации SQL, поэтому они широко используются профессионалами. PostgreSQL уже довольно давно поддерживает оконные функции, но некоторые возможности, упомянутые в стандарте SQL, до сих пор отсутствовали. Теперь, в версии PostgreSQL 11, поддерживается все, что требует стандарт SQL: 2011 в этой области.

Добавлены следующие возможности:

- RANGE BETWEEN:
 - раньше только ROWS;
 - теперь поддерживаются значения;
- исключение фрейма:
 - EXCLUDE CURRENT ROW;
 - EXCLUDE TIES.

Для демонстрации новых возможностей я включил пример. Приведенный ниже код содержит две оконные функции. В первой используется то, что уже было в PostgreSQL 10 и раньше. Во второй функции `array_agg` исключается текущая строка, это новая возможность, появившаяся в PostgreSQL 11.

```
test=# SELECT *,
      array_agg(x) OVER (ORDER BY x ROWS BETWEEN
                        1 PRECEDING AND 1 FOLLOWING),
      array_agg(x) OVER (ORDER BY x ROWS BETWEEN
```

```

1 PRECEDING AND 1 FOLLOWING EXCLUDE CURRENT ROW)
FROM generate_series(1, 5) AS x;
x | array_agg | array_agg
---+-----+-----
1 | {1,2}      | {2}
2 | {1,2,3}    | {1,3}
3 | {2,3,4}    | {2,4}
4 | {3,4,5}    | {3,5}
5 | {4,5}      | {4}
(5 строк)

```

Исключение текущей строки – довольно обычное требование, так что это улучшение не следует недооценивать.

Добавление JIT-компиляции

Своевременная (just-in-time, JIT) компиляция – одна из жемчужин PostgreSQL 11. Добавлена обширная инфраструктура для поддержки дополнительных видов JIT-компиляции в будущем, а PostgreSQL 11 – первая версия, в которой эта современная техника используется на полную катушку. Но сначала разберемся, что, собственно, такое JIT-компиляция. При выполнении запроса многое становится известно только на этапе выполнения, а не на этапе компиляции запроса. Поэтому традиционный компилятор находится в невыгодном положении, т. к. не знает, что произойдет во время выполнения. А JIT-компилятор знает гораздо больше и может отреагировать соответственно.

JIT-компиляция реализована начиная с версии PostgreSQL 11 и особенно полезна для больших запросов. Детали будут рассмотрены в последующих главах.

Улучшенное секционирование

Первый вариант секционирования появился в PostgreSQL 10. Конечно, наследованием мы пользовались и раньше. Но PostgreSQL 10 стала первой версией, в которой это делается на современном уровне. В PostgreSQL 11 эта и так уже весьма мощная функциональность расширена, в частности появилась возможность создавать секцию по умолчанию, если ни одна из существующих не подходит.

Вот как это работает:

```

postgres=# CREATE TABLE default_part PARTITION OF some_table DEFAULT;
CREATE TABLE

```

В данном случае все строки, не попавшие ни в какую другую секцию, попадают в секцию default_part.

Но это еще не все. В PostgreSQL строку нельзя было (легко) переместить из одной секции в другую. Предположим, что мы завели по одной секции на страну. Если человек переехал, например, из Франции в Эстонию, то одной команды UPDATE раньше было недостаточно. Нужно было удалить старую строку и вставить новую. В PostgreSQL 11 эта проблема решена. Строки перемещаются из одной секции в другую совершенно прозрачно.

Раньше в PostgreSQL было много других недостатков. В прежних версиях секции приходилось индексировать по отдельности. Невозможно было создать один индекс для всех секций. В PostgreSQL 11 построение индекса над родительской таблицей автоматически гарантирует, что все дочерние таблицы тоже будут проиндексированы. И это замечательно, потому что теперь вряд ли удастся просто забыть про какой-то индекс. Кроме того, в PostgreSQL 11 можно добавить глобальный уникальный индекс¹, так что для секционированной таблицы стало возможно определить ограничение уникальности.

Вплоть до PostgreSQL 10 у нас было секционирование по диапазону и секционирование по списку. В PostgreSQL 11 добавилось секционирование по хеш-коду. Например:

```
test=# CREATE TABLE tab(i int, t text) PARTITION BY HASH (i);
CREATE TABLE
test=# CREATE TABLE tab_1 PARTITION OF tab FOR VALUES WITH (MODULUS 4, REMAINDER 0);
CREATE TABLE
```

Но добавлена не только новая функциональность, но и целый ряд улучшений производительности. Устранение секций теперь производится гораздо быстрее, а кроме того, появилась возможность соединений по секциям и агрегатов по секциям, а это именно то, что необходимо для аналитики и хранилищ данных.

Поддержка хранимых процедур

В PostgreSQL всегда были функции, которые часто назывались хранимыми процедурами. Однако между хранимой процедурой и функцией есть различие. Вплоть до PostgreSQL 10 у нас были только функции, а процедур не было.

Вся штука в том, что функция – это часть объемлющей структуры, транзакции. А процедура может содержать несколько транзакций, поэтому ее нельзя вызывать из объемлющей транзакции, т. е. она в каком-то смысле самостоятельна.

Приведем синтаксис команды CREATE PROCEDURE:

```
test=# \h CREATE PROCEDURE
Команда: CREATE PROCEDURE
Описание: создать процедуру
Синтаксис:
CREATE [ OR REPLACE ] PROCEDURE
    Имя ( [ [ режим_аргумента ] [ имя_аргумента ]
        тип_аргумента [ { DEFAULT | = } выражение_по_умолчанию ] [, ... ] ] )
{ LANGUAGE имя_языка
  | TRANSFORM { FOR TYPE имя_типа } [, ... ]
  | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
  | SET параметр_конфигурации { TO значение | = значение | FROM CURRENT }
  | AS 'определение'
  | AS 'объектный_файл', 'объектный_символ'
} ...
```

¹ Технически такой индекс не является глобальным: в каждой секции создаются собственные локальные индексы. – *Прим. ред.*

Ниже показано, как в одной процедуре можно выполнить две транзакции:

```
test=# CREATE PROCEDURE test_proc()
      LANGUAGE plpgsql
AS $$
BEGIN
    CREATE TABLE a (aid int);
    CREATE TABLE b (bid int);
    COMMIT;
    CREATE TABLE c (cid int);
    ROLLBACK;
END;
$$;
CREATE PROCEDURE
```

Отметим, что первые две команды зафиксированы, а вторая транзакция отменена. Чуть ниже мы увидим, к чему это приводит.

Для выполнения данной процедуры используется команда CALL:

```
test=# CALL test_proc();
CALL
```

Первые две таблицы созданы, а третья – нет, потому что внутри процедуры была команда ROLLBACK:

```
test=# \d
      Список отношений
  Схема | Имя | Тип  | Владелец
-----+-----+-----+-----
 public | a   | table | hs
 public | b   | table | hs
(2 строки)
```

Процедуры – важный шаг на пути к созданию полнофункциональной системы баз данных.

Улучшение команды ALTER TABLE

Команда ALTER TABLE используется, чтобы изменить определение таблицы. В PostgreSQL 11 поведение команды ALTER TABLE ... ADD COLUMN улучшено. В примерах ниже показано, как можно добавить столбцы в таблицу и как PostgreSQL будет работать с этими столбцами:

```
ALTER TABLE x ADD COLUMN y int;
ALTER TABLE x ADD COLUMN z int DEFAULT 57;
```

Первая из этих двух команд всегда работала быстро, поскольку в PostgreSQL столбец по умолчанию имеет значение NULL. Поэтому PostgreSQL просто добавляет описание столбца в системный каталог, а данные таблицы при этом не изменяются. Столбец добавляется в конец таблицы, поэтому если хранящаяся на диске строка оказывается слишком короткой, мы знаем, что последнее поле в ней содержит NULL.

Но во втором случае ситуация совершенно иная. Наличие фразы `DEFAULT 57` означает, что в строку нужно добавить реальные данные, а в PostgreSQL 10 и более ранних версиях это приводило к перезаписи всей таблицы. Если таблица мала, то ничего страшного в этом нет. Но если в таблице миллиарды строк, то нельзя просто заблокировать ее на время перезаписи – в профессиональной системе оперативной обработки транзакций (OLTP) простой недопустим.

Начиная с версии PostgreSQL 11 можно добавлять неизменяемые значения без перезаписи всей таблицы, что в значительной мере решает проблему изменения структуры данных.

РЕЗЮМЕ

В PostgreSQL 11 добавлено много возможностей, позволяющих быстрее и эффективнее выполнять еще более профессиональные приложения. Улучшению подверглись различные аспекты сервера базы данных. В будущем усовершенствование продолжится. Конечно, приведенный в этой главе перечень далеко не полон – есть еще много мелких изменений.

В следующей главе мы поговорим об индексировании и стоимостной модели в PostgreSQL. То и другое очень важно для достижения высокой производительности.

Вопросы

Какова самая важная функциональная возможность PostgreSQL 11?

Трудно сказать. Все зависит от того, как вы используете базу данных и какие средства наиболее важны для вашего приложения. Но любимцы есть у каждого. Лично для меня это параллельное построение индексов, что очень важно для заказчиков, работающих с гигантскими базами данных. В любом случае, вам решать, что вам нравится, а что нет.

Будет ли PostgreSQL 11 работать на моей платформе?

PostgreSQL 11 работает на всех распространенных платформах, включая Linux, Windows, Solaris, AIX, macOS X и некоторые другие. Сообщество стремится охватить максимальное количество платформ, чтобы никто не был отлучен от PostgreSQL. Для большинства популярных систем имеются даже готовые пакеты, содержащие PostgreSQL.

Изменилась ли модель лицензирования?

Нет, ничего не изменилось и вряд ли когда-нибудь изменится.

Когда ожидать выхода PostgreSQL 12?

Обычно основные версии выходят раз в год. Поэтому версию PostgreSQL 12 следует ожидать осенью 2019 года.

Глава 2

Транзакции и блокировка

Итак, введение позади, и самое время сосредоточиться на первой важной теме. Блокировка – важнейшая концепция в любой базе данных. Важно понимать, как она работает, не только чтобы писать правильные приложения, но и с точки зрения производительности. Если блокировки неправильно обрабатываются, то приложение может работать медленно, а то и вовсе неправильно, демонстрируя совершенно неожиданное поведение. На мой взгляд, блокировки – ключ к производительности, поэтому в них нужно как следует разобраться. Понимание блокировок важно как для администраторов, так и для разработчиков. В этой главе мы рассмотрим следующие вопросы:

- работа с транзакциями в PostgreSQL;
- основы блокировки;
- использование фраз `FOR SHARE` и `FOR UPDATE`;
- уровни блокировки транзакций;
- транзакции, использующие **сериализуемую изоляцию снимков** (serializable snapshot isolation – SSI);
- взаимоблокировки и смежные проблемы;
- оптимизация хранилища и управление очисткой.

Прочитав главу до конца, вы будете понимать, как устроены транзакции в PostgreSQL, и сможете максимально эффективно использовать их.

РАБОТА С ТРАНЗАКЦИЯМИ В PostgreSQL

PostgreSQL предоставляет развитый механизм транзакций, обладающий бесчисленными возможностями, полезными равно разработчикам и администраторам. В этом разделе мы рассмотрим само понятие транзакции.

Первым делом нужно понимать, что в PostgreSQL любая операция выполняется в составе транзакции. Даже один простой запрос, отправленный серверу, – уже транзакция. Например:

```
test=# SELECT now(), now();
              now                |              now
-----+-----
 2018-08-24 16:03:27.174253+02 | 2018-08-24 16:03:27.174253+02
(1 строка)
```

В данном случае команда SELECT составляет отдельную транзакцию. Если выполнить ее снова, будут возвращены другие временные метки.



Имейте в виду, что функция `now()` возвращает время начала транзакции. Поэтому данная команда SELECT всегда возвращает две одинаковые временные метки. Если вы хотите получить «настоящее время», то пользуйтесь функцией `clock_timestamp()`, а не `now()`.

Транзакция, содержащая несколько команд, должна начинаться командой BEGIN, синтаксис которой описан ниже:

```
test=# \h BEGIN
```

Команда: BEGIN

Описание: начать транзакцию

Синтаксис:

```
BEGIN [ WORK | TRANSACTION ] [ режим_транзакции [, ...] ]
```

где допустимый режим_транзакции:

```
ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }
READ WRITE | READ ONLY
[ NOT ] DEFERRABLE
```

Команда BEGIN позволяет включить в транзакцию несколько команд, например:

```
test=# BEGIN;
```

```
BEGIN
```

```
test=# SELECT now();
           now
```

```
-----
2018-08-24 16:04:08.105131+02
(1 строка)
```

```
test=# SELECT now();
           now
```

```
-----
2018-08-24 16:04:08.105131+02
(1 строка)
```

```
test=# COMMIT;
COMMIT
```

Обратите внимание на важный момент: обе временные метки одинаковы. Как уже было сказано, они содержат время начала транзакции.

Для завершения транзакции служит команда COMMIT:

```
test=# \h COMMIT
```

Команда: COMMIT

Описание: зафиксировать текущую транзакцию

Синтаксис:

```
COMMIT [ WORK | TRANSACTION ]
```

Можно писать COMMIT, COMMIT WORK или COMMIT TRANSACTION – все три команды эквивалентны. Если и этого недостаточно, то есть еще одна команда

```
test=# \h END
Команда: END
Описание: зафиксировать текущую транзакцию
Синтаксис:
END [ WORK | TRANSACTION ]
```

Команды END и COMMIT эквивалентны.

Противоположностью COMMIT является ROLLBACK. Эта команда не завершает транзакцию нормально, а прерывает ее, так что никакие произведенные изменения не будут видны другим транзакциям:

```
test=# \h ROLLBACK
Команда: ROLLBACK
Описание: прервать текущую транзакцию
Синтаксис:
ROLLBACK [ WORK | TRANSACTION ]
```

Иногда в приложениях используют команду ABORT вместо ROLLBACK. Это одно и то же.

Обработка ошибок внутри транзакции

Не всегда транзакции безошибочно отработывают от начала до конца. По разным причинам что-то может пойти не так. Но в PostgreSQL зафиксировать можно только транзакции, завершившиеся без ошибок. Ниже показана транзакция, в которой ошибка вызвана делением на ноль:

```
test=# BEGIN;
BEGIN
test=# SELECT 1;
?column?
-----
      1
(1 строка)
test=# SELECT 1 / 0;
ОШИБКА: деление на ноль
test=# SELECT 1;
ОШИБКА: текущая транзакция прервана, команды до конца блока транзакции игнорируются
test=# COMMIT;
ROLLBACK
```



В любой правильно написанной базе данных команда, показанная выше, вызывает немедленную ошибку и аварийное завершение оператора.

Подчеркнем, что PostgreSQL сообщит об ошибке, в отличие от MySQL, который ведет себя гораздо либеральнее. После ошибки больше никакие команды не принимаются, пусть даже они синтаксически и семантически правильны. Команду COMMIT выполнить можно, но PostgreSQL при этом откатит транзакцию, потому что в данной точке это единственно правильное действие.

Использование команды SAVEPOINT

В профессиональных приложениях бывает довольно трудно написать длинную транзакцию, гарантировав отсутствие ошибок. Для решения этой проблемы предназначена команда SAVEPOINT. Она организует безопасное место внутри транзакции, к которому можно будет вернуться, если вдруг все пойдет наперекосяк. Например:

```
test=# BEGIN;
BEGIN
test=# SELECT 1;
?column?
-----
      1
(1 строка)

test=# SAVEPOINT a;
SAVEPOINT
test=# SELECT 2 / 0;
ОШИБКА: деление на ноль
test=# SELECT 2;
ОШИБКА: текущая транзакция прервана, команды до конца блока транзакции игнорируются
test=# ROLLBACK TO SAVEPOINT a;
ROLLBACK
test=# SELECT 3;
?column?
-----
      3
(1 строка)

test=# COMMIT;
COMMIT
```

После первой команды SELECT я создал точку сохранения, чтобы приложение могло позже вернуться к этой точке, пока не вышло из транзакции. Как видите, у точки сохранения есть имя, по которому на нее можно сослаться.

После возврата к точке сохранения а транзакция может продолжиться нормально. Поток выполнения вернулся в точку, где ошибки еще не было, так что все хорошо.

Количество точек сохранения внутри транзакции практически не ограничено. Нам доводилось встречать заказчиков, у которых было свыше 250 000 точек сохранения в одной операции. PostgreSQL с этим легко справляется.

Если вы хотите удалить точку сохранения из транзакции, выполните команду RELEASE SAVEPOINT:

```
test=# \h RELEASE SAVEPOINT
Команда: RELEASE SAVEPOINT
Описание: удалить ранее определенную точку сохранения
Синтаксис:
RELEASE [ SAVEPOINT ] имя_точки_сохранения
```

Многие спрашивают, что произойдет, если попытаться обратиться к точке сохранения после завершения транзакции? Ответ простой – жизнь точки сохранения заканчивается вместе с транзакцией. Иными словами, вернуться к некоторому состоянию, после того как транзакция завершилась, невозможно.

Транзакционные DDL-команды

В PostgreSQL есть одна замечательная особенность, к сожалению, отсутствующая во многих коммерческих СУБД, – в блоке транзакции можно выполнять DDL-команды (т. е. команды, изменяющие структуру данных). В типичной коммерческой системе DDL-команда неявно фиксирует текущую транзакцию. В PostgreSQL такого не происходит.

За исключением (DROP DATABASE, CREATE TABLESPACE, DROP TABLESPACE и еще несколько команд), DDL-команды в PostgreSQL транзакционные, что очень удобно конечным пользователям. Приведем пример.

```
test=# \d
Отношений не найдено.
test=# BEGIN;
BEGIN
test=# CREATE TABLE t_test (id int);
CREATE TABLE
test=# ALTER TABLE t_test ALTER COLUMN id TYPE int8;
ALTER TABLE
test=# \d t_test
      Таблица "public.t_test"
  Столбец | Тип      | Модификаторы
-----+-----+-----
id       | bigint  |
```

```
test=# ROLLBACK;
ROLLBACK
test=# \d
Отношений не найдено.
```

В этом примере создана и модифицирована таблица, а затем вся транзакция прервана. Как видим, нет ни неявного COMMIT, ни еще какого-то странного поведения. PostgreSQL работает точно так, как мы ожидаем.

Транзакционные DDL-команды особенно важны, когда требуется развернуть какую-то программу. Рассмотрим, к примеру, **систему управления содержанием** (content management system – CMS). После выхода новой версии мы хотим обновиться. Работать со старой версией по-прежнему можно; работать с новой тоже можно, но мешать в кучу новую и старую мы не хотим. Поэтому разворачивать обновление в одной транзакции было бы весьма желательно, потому что такая операция атомарна.



Для поощрения правильных методов управления ПО мы можем включить несколько отдельных модулей из системы управления исходным кодом в одну транзакцию разворачивания.

ОСНОВЫ БЛОКИРОВКИ

В этом разделе мы рассмотрим основы механизма блокировки. Наша цель – понять, как блокировка работает в принципе и как правильно написать простое приложение.

Для демонстрации создадим простую таблицу и добавим в нее одну строку командой INSERT:

```
test=# CREATE TABLE t_test (id int);
CREATE TABLE
test=# INSERT INTO t_test VALUES (0);
INSERT 0 1
```

Первое, что следует отметить, – тот факт, что может одновременно выполняться несколько операций чтения одной таблицы. Из-за того, что много пользователей в один и тот же момент читают одни и те же данные, блокировки не произойдет. Поэтому PostgreSQL может обслуживать тысячи пользователей без всяких проблем.

i Несколько пользователей могут одновременно читать одни и те же данные, не блокируя друг друга.

А что, если одновременно производятся операции чтения и записи? Рассмотрим пример. Пусть таблица содержит одну строку и в ней `id = 0`:

Транзакция 1	Транзакция 2
BEGIN;	BEGIN;
UPDATE t_test SET id = id + 1 RETURNING *;	
Пользователь увидит 1	SELECT * FROM t_test;
	Пользователь увидит 0
COMMIT;	COMMIT;

Открыто две транзакции. Первая изменяет строку. Но никакой проблемы нет – вторая транзакция может продолжаться. Она вернет старую строку – в том виде, в каком она существовала до UPDATE. Такое поведение называется **многоверсионным управлением конкурентностью** (multi-version concurrency control – MVCC).

i Транзакция видит данные, только если они были зафиксированы в транзакции записи до начала транзакции чтения. Никакая транзакция не может увидеть изменения, произведенные в другой активной транзакции. Транзакция видит лишь те изменения, которые уже зафиксированы.

Есть еще один важный момент – многие базы данных, коммерческие и с открытым исходным кодом, до сих пор (в 2018 году) не умеют обрабатывать конкурентные операции чтения и записи. В PostgreSQL это не составляет никакой проблемы. Чтение и запись мирно сосуществуют.

i Транзакции записи не блокируют транзакции чтения.

После фиксации транзакции таблица будет содержать значение 1. А что, если два человека изменяют данные одновременно? Рассмотрим пример:

Транзакция 1	Транзакция 2
BEGIN;	BEGIN;
UPDATE t_test SET id = id + 1 RETURNING *;	
Будет возвращено 2	UPDATE t_test SET id = id + 1 RETURNING *;
	Будет ждать завершения транзакции 1
COMMIT;	Прочтет новое состояние строки, увидит значение 2, установит новое значение и вернет 3
	COMMIT;

Предположим, что мы хотим подсчитать количество заходов на сайт. Если код выглядит, как показано выше, то ни один заход не останется неучтенным, потому что второй UPDATE выполняется после первого.

i PostgreSQL блокирует только строки, затронутые командой UPDATE. Так что если в таблице 1000 строк, то теоретически можно выполнять в ней 1000 конкурентных изменений.

Отметим также, что конкурентные операции чтения можно выполнять всегда. Наши две операции записи не блокируют чтения.

Предотвращение типичных ошибок и явная блокировка

Работая профессиональным консультантом по PostgreSQL (<https://www.cybertec-postgresql.com>), я встречал ошибки, которые повторялись раз за разом. Вот моя любимая:

Транзакция 1	Транзакция 2
BEGIN;	BEGIN;
SELECT max(id) FROM product;	SELECT max(id) FROM product;
Пользователь видит 17	Пользователь видит 17
Пользователь решает добавить продукт 18	Пользователь решает добавить продукт 18
INSERT INTO product ... VALUES (18, ...)	INSERT INTO product ... VALUES (18, ...)
COMMIT;	COMMIT;

В данном случае будет либо нарушение ограничения уникальности ключа, либо две одинаковые записи. Ни то, ни другое не радует.

Один из способов решить проблему – воспользоваться явной табличной блокировкой. Ниже показан синтаксис команды LOCK:

```
test=# \h LOCK
```

```
Команда: LOCK
```

```
Описание: заблокировать таблицу
```


Синтаксис:

```
LOCK [ TABLE ] [ ONLY ] имя [ * ] [, ...] [ IN режим_блокировки MODE ] [ NOWAIT ]
```

где допустимый режим_блокировки:

```
ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE  
| SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE
```

Как видим, PostgreSQL предлагает восемь режимов блокировки всей таблицы. Блокировка может быть как весьма либеральной, например ACCESS SHARE, так и очень строгой, например ACCESS EXCLUSIVE. В перечне ниже показано, что делает каждый режим блокировки.

- ACCESS SHARE: этот режим устанавливается операциями чтения и конфликтует только с режимом ACCESS EXCLUSIVE, который устанавливается командой DROP TABLE и ей подобными. На практике это означает, что команда SELECT не может начаться, если таблицу собираются удалить. Это также означает, что команда DROP TABLE должна ждать завершения транзакции чтения.
- ROW SHARE: PostgreSQL устанавливает этот режим блокировки при выполнении команд SELECT FOR UPDATE и SELECT FOR SHARE. Он конфликтует с режимами EXCLUSIVE и ACCESS EXCLUSIVE.
- ROW EXCLUSIVE: этот режим устанавливается командами INSERT, UPDATE и DELETE. Он конфликтует с режимами SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE и ACCESS EXCLUSIVE.
- SHARE UPDATE EXCLUSIVE: этот режим устанавливается командами CREATE INDEX CONCURRENTLY, ANALYZE, ALTER TABLE VALIDATE и еще некоторыми вариантами команды ALTER TABLE, а также командой VACUUM (но не VACUUM FULL). Он конфликтует с режимами SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE и ACCESS EXCLUSIVE.
- SHARE: во время построения индекса захватываются блокировки типа SHARE. Этот режим конфликтует с ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE ROW EXCLUSIVE, EXCLUSIVE и ACCESS EXCLUSIVE.
- SHARE ROW EXCLUSIVE: устанавливается командой CREATE TRIGGER и некоторыми вариантами команды ALTER TABLE. Конфликтует со всеми режимами, кроме ACCESS SHARE.
- EXCLUSIVE: этот режим совместим только с режимом ACCESS SHARE, то есть параллельно с транзакцией, получившей блокировку в этом режиме, допускается только чтение таблицы.
- ACCESS EXCLUSIVE: самый ограничительный режим блокировки. Он защищает как от записи, так и от чтения. Если транзакция захватила такую блокировку, то больше никто не сможет ни читать соответствующую таблицу, ни писать в нее.

Принимая во внимание инфраструктуру блокировок в PostgreSQL, одно из решений описанной выше проблемы получения max выглядит следующим образом:

```
BEGIN;
LOCK TABLE product IN ACCESS EXCLUSIVE MODE;
INSERT INTO product SELECT max(id) + 1, ... FROM product;
COMMIT;
```

Имейте в виду, что такой способ выполнения операции нежелателен, потому что в течение всего времени выполнения никто больше не сможет ни читать таблицу, ни записывать в нее. Режим `ACCESS EXCLUSIVE` нужно всеми силами избегать.

Альтернативные решения

Есть и другие способы решить проблему. В качестве примера рассмотрим приложение, генерирующее номера счетов-фактур. Налоговая инспекция может потребовать, чтобы в последовательности номеров не было ни лакун, ни дубликатов. Как это сделать? Понятно, что одно из решений – заблокировать таблицу. Но есть способ лучше. Вот как поступил бы я:

```
test=# CREATE TABLE t_invoice (id int PRIMARY KEY);
CREATE TABLE
test=# CREATE TABLE t_watermark (id int);
CREATE TABLE
test=# INSERT INTO t_watermark VALUES (0);
INSERT 0
test=# WITH x AS (UPDATE t_watermark SET id = id + 1 RETURNING *)
      INSERT INTO t_invoice
      SELECT * FROM x RETURNING *;

 id
----
  1
(1 строка)
```

Я ввел еще одну таблицу `t_watermark`. Она содержит всего одну строку. Сначала выполняется команда `WITH`. Строка блокируется, поле в ней увеличивается на 1, и возвращается новое значение. В каждый момент времени это может сделать только один человек. Затем значение, возвращенное общим табличным выражением (СТЕ), вставляется в таблицу счетов. Оно гарантированно будет уникальным. Прелесть решения состоит в том, что для таблицы `t_watermark` захватывается простая строковая блокировка, что не мешает читать из нее. Конечно, такой подход масштабируется лучше.

ИСПОЛЬЗОВАНИЕ ФРАЗ FOR SHARE и FOR UPDATE

Иногда приложение выбирает данные из базы, как-то обрабатывает их, а затем записывает измененные данные в базу. Это классическое применение `SELECT FOR UPDATE`.

В следующем примере показано неправильное, но часто встречающееся употребление `SELECT`:

```
BEGIN;
SELECT * FROM invoice WHERE processed = false;
** здесь приложение что-то делает **
UPDATE invoice SET processed = true ...
COMMIT;
```

Проблема в том, что два человека могут выбрать одни и те же необработанные данные. И изменения, сделанные одним, будут перезаписаны изменениями, сделанными другим. Налицо состояние гонки.

Чтобы решить эту проблему, можно воспользоваться командой `SELECT FOR UPDATE`. В примере ниже показан типичный сценарий:

```
BEGIN;
SELECT * FROM invoice WHERE processed = false FOR UPDATE;
** здесь приложение что-то делает **
UPDATE invoice SET processed = true ...
COMMIT;
```

`SELECT FOR UPDATE` блокирует строки так же, как это сделала бы команда `UPDATE`. И следовательно, конкурентно произвести изменения нельзя. После `COMMIT` все блокировки, как обычно, освобождаются.

Если одна команда `SELECT FOR UPDATE` ждет завершения другой такой же команды, то она будет «висеть», пока первая начатая команда не завершится (в результате `COMMIT` или `ROLLBACK`). Если первая транзакция по какой-то причине не хочет завершаться, то вторая будет ждать бесконечно. Чтобы предотвратить такое развитие ситуации, можно воспользоваться командой `SELECT FOR UPDATE NOWAIT`.

Вот как это работает.

Транзакция 1	Транзакция 2
BEGIN;	BEGIN;
SELECT ... FROM tab WHERE ... FOR UPDATE NOWAIT;	
Какая-то обработка	SELECT ... FROM tab WHERE ... FOR UPDATE NOWAIT;
Какая-то обработка	ОШИБКА: не могу получить блокировку строки в отношении tab

Если гибкости `NOWAIT` для вас недостаточно, подумайте об использовании параметра `lock_timeout`. С его помощью задается время, в течение которого вы готовы ждать получения блокировки. Параметр можно задать на уровне сеанса:

```
test=# SET lock_timeout TO 5000;
SET
```

В данном случае задано значение 5 с.

Если `SELECT` по существу ничего не блокирует, то `SELECT FOR UPDATE` может вести себя довольно сурово. Рассмотрим такой бизнес-процесс: требуется заполнить самолет, в котором имеется 200 мест. Много людей пытается забронировать места одновременно. При этом может произойти вот что:

Транзакция 1	Транзакция 2
BEGIN;	BEGIN;
SELECT ... FROM flight LIMIT 1 FOR UPDATE;	
Ждет, пока пользователь введет данные	SELECT ... FROM flight LIMIT 1 FOR UPDATE;
Ждет, пока пользователь введет данные	Вынуждена ждать

Проблема в том, что в каждый момент времени можно забронировать только одно место. Всего имеется 200 мест, но все вынуждены ждать первого пользователя. И пока первое место заблокировано, никто не может ничего забронировать, хотя пассажирам безразлично, какое место они получат.

Эту проблему решает команда SELECT FOR UPDATE SKIP LOCKED. Сначала создадим тестовые данные:

```
test=# CREATE TABLE t_flight AS
        SELECT * FROM generate_series(1, 200) AS id;
SELECT 200
```

А вот теперь время волшебства.

Транзакция 1	Транзакция 2
BEGIN;	BEGIN;
SELECT * FROM t_flight LIMIT 2 FOR UPDATE SKIP LOCKED;	SELECT * FROM t_flight LIMIT 2 FOR UPDATE SKIP LOCKED;
Возвращает 1 и 2	Возвращает 3 и 4

Если каждый захочет выбрать две строки, то мы сможем обслужить 100 одновременных транзакций, не беспокоясь о блокировках.

☑ Помните, что ожидание – самая медленная форма выполнения. Если в каждый момент времени может быть активна только одна транзакция, то нет смысла покупать все более дорогие серверы – ведь проблема вызвана блокировкой и конфликтующими транзакциями, а не нехваткой ресурсов.

Но это еще не все. Иногда в FOR UPDATE возникают непредвиденные последствия. Мало кто знает, что FOR UPDATE тесно связана с внешними ключами. Допустим, что имеется две таблицы: в одной хранятся валюты, в другой – счета:

```
CREATE TABLE t_currency (id int, name text, PRIMARY KEY (id));
INSERT INTO t_currency VALUES (1, 'EUR');
INSERT INTO t_currency VALUES (2, 'USD');

CREATE TABLE t_account (
    id int,
    currency_id int REFERENCES t_currency (id)
                        ON UPDATE CASCADE
                        ON DELETE CASCADE,
    balance numeric);
INSERT INTO t_account VALUES (1, 1, 100);
INSERT INTO t_account VALUES (2, 1, 200);
```

Теперь мы хотим выполнить SELECT FOR UPDATE для таблицы счетов.

Транзакция 1	Транзакция 2
BEGIN;	
SELECT * FROM t_account FOR UPDATE;	BEGIN;
Ждет, пока пользователь закончит обработку	UPDATE t_currency SET id = id * 10;
Ждет, пока пользователь закончит обработку	Ждет транзакции 1

Хотя для таблицы счетов выполнена команда SELECT FOR UPDATE, команда UPDATE для таблицы валют оказалась заблокирована. И это необходимо, потому что иначе открылась бы возможность нарушить ограничение внешнего ключа. Таким образом, если структура данных достаточно сложна, то состязание может произойти там, где этого меньше всего ожидаешь (за сильно востребованные справочные таблицы).

Помимо фразы FOR UPDATE, существуют еще фразы FOR SHARE, FOR NO KEY UPDATE и FOR KEY SHARE. Ниже описано, что они означают.

- FOR NO KEY UPDATE: похожа на FOR UPDATE. Но эта фраза слабее и потому может сосуществовать с SELECT FOR SHARE.
- FOR SHARE: фраза FOR UPDATE является довольно сильным ограничением и работает в предположении, что мы действительно собираемся изменять строки. Фраза FOR SHARE отличается от нее тем, что удерживать блокировку такого типа одновременно может несколько транзакций.
- FOR KEY SHARE: аналогична FOR SHARE, но блокировка слабее. Она конфликтует с FOR UPDATE, но не конфликтует с FOR NO KEY UPDATE.

Лучше всего поэкспериментировать с различными режимами и посмотреть, что получится. Следует всячески стремиться улучшить поведение блокировки, поскольку это может резко повысить масштабируемость вашего приложения.

УРОВНИ ИЗОЛЯЦИИ ТРАНЗАКЦИЙ

До сих пор мы рассматривали обработку блокировок и некоторые общие вопросы конкурентности. В этом разделе мы поговорим об изоляции транзакций. Я считаю, что этой теме в современной разработке уделяется совершенно недостаточное внимание. Лишь малая доля программистов вообще знает об этой проблеме, что приводит к ошеломительным ошибкам.

Приведем пример того, что может произойти.

Транзакция 1	Транзакция 2
BEGIN;	
SELECT sum(balance) FROM t_account;	
Пользователь увидит 300	BEGIN;
	INSERT INTO t_account (balance) VALUES (100);

Транзакция 1	Транзакция 2
	COMMIT;
SELECT sum(balance) FROM t_account;	
Пользователь увидит 400	
COMMIT;	

Большинство пользователей ожидает, что левая транзакция всегда будет возвращать 300, что бы ни происходило в правой транзакции. Однако это не так. По умолчанию PostgreSQL работает в режиме изоляции READ COMMITTED. Это означает, что любая команда внутри транзакции получает новый снимок данных, который остается постоянным на всем протяжении запроса.

i SQL-команда работает с одним и тем же снимком данных и игнорирует все изменения, произведенные конкурентными транзакциями.

Если вы хотите избежать такого поведения, то можете установить режим изоляции транзакций REPEATABLE READ. В этом режиме транзакция будет пользоваться одним и тем же снимком на всем протяжении работы. Вот что тогда произойдет.

Транзакция 1	Транзакция 2
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;	
SELECT sum(balance) FROM t_account;	
Пользователь увидит 300	BEGIN;
	INSERT INTO t_account (balance) VALUES (100);
	COMMIT;
SELECT sum(balance) FROM t_account;	SELECT sum(balance) FROM t_account;
Пользователь увидит 300	Пользователь увидит 400
COMMIT;	

Как мы только что сказали, первая транзакция замораживает свой снимок данных и возвращает постоянные результаты на всем протяжении работы. Это особенно полезно при формировании отчетов. Первая и последняя страницы отчета должны относиться к одним и тем же данным. Поэтому режим повторяемого чтения – ключ к непротиворечивым отчетам.

Отметим, что ошибки, связанные с уровнем изоляции, не всегда проявляются мгновенно. Может случиться, что проблема выскакивает спустя много лет после передачи приложения в эксплуатацию.

i Повторяемое чтение обходится не дороже, чем чтение зафиксированных данных. Волноваться по поводу снижения производительности нет причин. В обычном режиме **оперативной обработки транзакций (OLTP)** режим READ COMMITTED имеет ряд преимуществ, потому что изменения видны гораздо раньше, и шансы столкнуться с неожиданными ошибками уменьшаются.

SSI-транзакции

Помимо режимов READ COMMITTED и REPEATABLE READ, PostgreSQL предлагает режим сериализуемой изоляции снимков (Serializable Snapshot Isolation – SSI). Так что всего PostgreSQL поддерживает три уровня изоляции. Отметим, что режим чтения незафиксированных данных READ UNCOMMITTED (который все еще является режимом по умолчанию в некоторых коммерческих СУБД) не поддерживается: если вы попытаете начать транзакцию в таком режиме, то PostgreSQL молча заменит режим на READ COMMITTED. Но вернемся к сериализуемой изоляции снимков.

Идея этого уровня изоляции проста: если известно, что транзакция работает правильно, когда имеется всего один пользователь, то она будет правильно работать и в конкурентной среде при выборе такого уровня изоляции. Однако пользователи должны быть готовы к тому, что транзакции могут завершаться с ошибкой (так и задумано). Кроме того, придется смириться со снижением производительности.

Желающие узнать больше об этом уровне изоляции могут обратиться к странице <https://wiki.postgresql.org/wiki/Serializable>.



Рассматривать сериализуемую изоляцию в качестве альтернативы следует, только если вы хорошо понимаете, что происходит внутри движка базы данных.

Взаимоблокировки и смежные вопросы

Взаимоблокировка – это важная проблема, которая может иметь место в любой известной мне базе данных. Это происходит, когда две транзакции вынуждены ждать друг друга.

В этом разделе мы увидим, как такое может случиться. Пусть имеется таблица, содержащая две строки:

```
CREATE TABLE t_deadlock (id int);  
INSERT INTO t_deadlock VALUES (1), (2);
```

Рассмотрим пример:

Транзакция 1	Транзакция 2
BEGIN;	BEGIN;
UPDATE t_deadlock SET id = id * 10 WHERE id = 1;	UPDATE t_deadlock SET id = id * 10 WHERE id = 2;
UPDATE t_deadlock SET id = id * 10 WHERE id = 2;	
Ждет транзакции 2	UPDATE t_deadlock SET id = id * 10 WHERE id = 1;
Ждет транзакции 2	Ждет транзакции 1
	Взаимоблокировка будет разрешена через одну секунду (deadlock_timeout)
COMMIT;	ROLLBACK;

Сразу после обнаружения взаимоблокировки появляется такое сообщение:

ОШИБКА: обнаружена взаимоблокировка

ПОДРОБНОСТИ: Процесс 91521 ожидает в режиме ShareLock блокировку "транзакция 903";
заблокирован процессом 77185.

Процесс 77185 ожидает в режиме ShareLock блокировку "транзакция 905";
заблокирован процессом 91521.

ПОДСКАЗКА: подробности запроса смотрите в протоколе сервера.

КОНТЕКСТ: при изменении кортежа (0,1) в отношении "t_deadlock"

PostgreSQL даже любезно сообщает, какая строка привела к конфликту. В этом примере корнем зла является кортеж (0, 1). Мы видим ctid – уникальный идентификатор строки в таблице. Он говорит о физическом положении строки внутри таблицы. В данном случае это первая строка в первом блоке (0).

Можно даже опросить эту строку, если она еще видна транзакции:

```
test=# SELECT ctid, * FROM t_deadlock WHERE ctid = '(0, 3)';
 ctid | id
-----+-----
(0,1) | 10
(1 строка)
```

Имейте в виду, что этот запрос может и не вернуть строку, если она уже была удалена или модифицирована.

Однако не только взаимоблокировки приводят к ошибкам в транзакциях. Невозможность сериализации может вызываться и другими причинами. В следующем примере показано, что еще может произойти. Предполагается, что в таблице все еще находятся две строки: id = 1 и id = 2.

Транзакция 1	Транзакция 2
BEGIN ISOLATION LEVEL REPEATABLE READ;	
SELECT * FROM t_deadlock;	
Возвращаются две строки	
	DELETE FROM t_deadlock;
SELECT * FROM t_deadlock;	
Возвращаются две строки	
DELETE FROM t_deadlock;	
Ошибка в транзакции	
ROLLBACK; -- COMMIT уже невозможен	

В этом примере одновременно работают две транзакции. Пока транзакция 1 только выбирает данные, все хорошо, потому что PostgreSQL легко может поддержать иллюзию статических данных. Но что, если вторая транзакция зафиксирует команду DELETE? Если в первой транзакции производятся только операции чтения, то проблема все равно не возникает. Неприятности начинаются, когда транзакция 1 пытается удалить или модифицировать данные, которые в этот момент фактически «мертвы». Единственное, что PostgreSQL может сделать в этой ситуации, – выдать ошибку из-за конфликта транзакций:


```
test=# DELETE FROM t_deadlock;
```

ОШИБКА: не удалось сериализовать доступ из-за параллельного изменения

На практике это означает, что конечные пользователи должны быть готовы к обработке ошибок в транзакциях. Если что-то пошло не так, то правильно написанное приложение должно повторить операцию.

РЕКОМЕНДАТЕЛЬНЫЕ БЛОКИРОВКИ

В PostgreSQL имеется весьма эффективный и прекрасно развитый механизм транзакций, который может обрабатывать мелкоструктурные блокировки. Несколько лет назад группе людей пришла в голову мысль использовать этот код для синхронизации приложений.

Так и появились на свет рекомендательные блокировки.

Важно понимать, что рекомендательные блокировки, в отличие от обычных, не исчезают после выполнения COMMIT¹. Следовательно, их нужно вовремя разблокировать, и пренебрегать этим ни в коем случае нельзя.

Применяя рекомендательную блокировку, вы на самом деле блокируете некое число. Не строки, не данные – просто число. Вот как это работает.

Транзакция 1	Транзакция 2
BEGIN;	
SELECT pg_advisory_lock(15);	
	SELECT pg_advisory_lock(15);
	Должна ждать
COMMIT;	Все еще должна ждать
SELECT pg_advisory_unlock(15);	Все еще ждет
	Блокировка захвачена

Первая транзакция блокирует число 15. Вторая транзакция должна ждать разблокировки этого числа. Она ждет даже после того, как первая транзакция зафиксирована. Ни в коем случае нельзя надеяться, что вместе с завершением транзакции все проблемы будут чудесным образом разрешены.

Для разблокировки всех заблокированных чисел PostgreSQL предоставляет функцию `pg_advisory_unlock_all()`:

```
test=# SELECT pg_advisory_unlock_all();
pg_advisory_unlock_all
```

```
-----
```

(1 строка)

Иногда возникает необходимость узнать, можно ли получить блокировку, и выдать ошибку, если это невозможно. Для этого в PostgreSQL есть функции;

¹ Существуют и рекомендательные блокировки на уровне транзакции, они устанавливаются функцией `pg_advisory_xact_lock`. – *Прим. ред.*

чтобы распечатать список всех имеющихся функций, относящихся к рекомендательным блокировкам, введите команду `\df *try*advisory*`.

ОПТИМИЗАЦИЯ ХРАНИЛИЩА И УПРАВЛЕНИЕ ОЧИСТКОЙ

Транзакции являются неотъемлемой частью PostgreSQL. Однако они обходятся не бесплатно. Как мы уже видели в этой главе, одновременно работающие пользователи могут получать различные данные в ответ на один и тот же запрос. Кроме того, командам DELETE и UPDATE не разрешено перезаписывать одни и те же данные, потому что тогда не сможет работать ROLLBACK. Если вы находитесь в середине большой операции DELETE, то нет гарантии, что COMMIT выполнится успешно. Более того, данные остаются видимыми во время выполнения DELETE, а иногда они видны спустя длительное время после завершения модификации.

Таким образом, очистка должна производиться асинхронно. Транзакция не может прибраться за собой, а окончательно удалять мертвые строки во время выполнения COMMIT и ROLLBACK может быть слишком рано.

Решением этой проблемы является команда VACUUM. Вот ее синтаксис:

```
test=# \h VACUUM
Команда: VACUUM
Описание: произвести сборку мусора и проанализировать базу данных
Синтаксис:
VACUUM [ ( { FULL | FREEZE | VERBOSE | ANALYZE } [, ...] ) ]
        [ имя_таблицы [ (имя_столбца [, ...]) ] ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] [ имя_таблицы ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ]
ANALYZE [ имя_таблицы [ (имя_столбца [, ...]) ] ] ]
```

VACUUM посещает все страницы, которые могут содержать модификации, и находит все мертвые кортежи. Свободное место запоминается в **карте свободного пространства** (Free Space Map – FSM) отношения.

Отметим, что в большинстве случаев VACUUM не уменьшает размер таблицы. Она просто находит и запоминает свободные участки в существующих файлах.

i Обычно после VACUUM размер таблицы не изменяется. В редких случаях, когда в конце таблицы нет действительных строк, файл сжимается. Но это скорее исключение, чем правило.

Что все это означает для пользователей, объяснено в разделе этой главы «Наблюдение за работой VACUUM».

Настройка VACUUM и autovacuum

Когда-то давным-давно запускать VACUUM приходилось вручную. По счастью, те времена прошли. Сегодня к услугам администратора имеется инструмент autovacuum, являющийся частью инфраструктуры сервера PostgreSQL. Он работает в фоновом режиме и автоматически производит очистку. Он просыпается один раз в минуту (параметр autovacuum_naptime = 1 в файле postgresql.conf) и проверяет, есть ли для него работа. Если есть, то autovacuum запускает до трех рабочих процессов (параметр autovacuum_max_workers в файле postgresql.conf).

Остается главный вопрос: что именно служит спусковым крючком для создания рабочего процесса?

i На самом деле процесс `autovacuum` не запускает процессы сам, а просит об этом главный процесс. Это делается, для того чтобы избежать образования процессов-зомби в случае ошибки и повысить общую стабильность системы.

Ответ на этот вопрос снова следует искать в файле `postgresql.conf`:

```
autovacuum_vacuum_threshold = 50
autovacuum_analyze_threshold = 50
autovacuum_vacuum_scale_factor = 0.2
autovacuum_analyze_scale_factor = 0.1
```

Параметр `autovacuum_vacuum_scale_factor` говорит PostgreSQL, что таблицу пора очищать, если в ней изменилось 20 % данных. Проблема в том, что если в таблице всего одна строка, то каждое изменение – это уже 100 %. Но запускать процесс для очистки всего одной строки бессмысленно. Поэтому параметр `autovacuum_vacuum_threshold` говорит, что вышеупомянутые 20 % в абсолютном исчислении должны содержать не менее 50 строк, иначе `VACUUM` не станет беспокоиться. Тот же механизм используется при сборе статистики для оптимизатора. Чтобы оправдать новый акт сбора статистики, нужно, чтобы изменилось не менее 10 % данных, а в абсолютном исчислении это должно быть не менее 50 строк. В идеале `autovacuum` собирает новую статистику во время обычной работы `VACUUM`, чтобы избежать ненужных просмотров таблицы.

Подробнее о заикливании идентификаторов транзакций

В файле `postgresql.conf` есть еще два параметра, чрезвычайно важных для понимания работы PostgreSQL:

```
autovacuum_freeze_max_age = 200000000
autovacuum_multixact_freeze_max_age = 400000000
```

Чтобы разобраться в сути проблемы, нужно понимать, как в PostgreSQL обрабатывается конкурентность. Механизм транзакций основан на сравнении идентификаторов и состояний транзакций.

Рассмотрим пример. Если я нахожусь в транзакции с идентификатором 4711, а вы – в транзакции 4712, то я вас не увижу, потому что ваша транзакция еще выполняется. Но если вы находитесь в транзакции 3900, то я вас увижу¹. Если ваша транзакция завершится с ошибкой, то я могу спокойно игнорировать все порожденные ей строки.

Беда в том, что количество идентификаторов транзакций конечно, и в какой-то момент произойдет заикливание, т. е. возврат к нулю. На практике это означает, что транзакция с номером 5 может быть позже транзакции с номером 800 миллионов. И как PostgreSQL узнает, какая из них начата раньше? Для

¹ Рассуждения упрощены до того, что стали неверными, но пример должен показать, что транзакции (обычно) получают возрастающие номера: чем позже началась транзакция, тем больше её номер. – *Прим. ред.*

этого она хранит отметку уровня. Рано или поздно эти отметки нужно корректировать, и вот тут-то в игру вступает VACUUM. Запуск VACUUM (или autovacuum) гарантирует, что отметка уровня будет скорректирована так, что в будущем у вас всегда будет достаточно идентификаторов транзакций.

i Не каждая транзакция увеличивает счетчик идентификаторов. Если транзакция только читает, то ее идентификатор виртуальный. Так сделано, чтобы идентификаторы транзакций не расходовались слишком быстро.

Параметр `autovacuum_freeze_max_age` определяет максимальное число транзакций (возраст), до которого может дорасти поле `pg_class.relFrozenxid`, ассоциированное с данной таблицей, прежде чем VACUUM вмешается с целью предотвратить заикливание идентификаторов транзакций. Значение этого параметра относительно мало, потому что он также влияет на очистку буфера состояния фиксации `clog` (`clog`, или `commit log` – это структура данных, в которой для каждой транзакции отведено два бита, показывающих ее состояние – работает, прервана, зафиксирована или все еще находится в подтранзакции).

Параметр `autovacuum_multixact_freeze_max_age` задает максимальный возраст поля `pg_class.relminmxid`, ассоциированного с таблицей, по достижении которого VACUUM вмешивается, чтобы предотвратить заикливание идентификатора мультитранзакций в данной таблице. Замораживание кортежей – важный элемент повышения производительности, мы еще вернемся к этому процессу в главе 6, когда будем обсуждать оптимизацию запросов.

Вообще говоря, стоит стремиться к уменьшению нагрузки со стороны VACUUM при сохранении операционной безопасности. Применение VACUUM к большим таблицам обходится дорого, поэтому полезно настроить эти параметры оптимальным для вашей системы образом.

Пара слов о VACUUM FULL

Вместо обычной команды VACUUM можно запустить VACUUM FULL. Но хочу отметить, что VACUUM FULL блокирует таблицу и перезаписывает ее целиком. Если таблица маленькая, то ничего страшного в этом нет. Однако блокировка большой таблицы может доставить вам массу неприятностей! VACUUM FULL блокирует все операции записи¹, поэтому у пользователей сложится впечатление, что база данных зависла. Так что применяйте с осторожностью.

Вместо VACUUM FULL я рекомендую обратить внимание на расширение `pg_squeeze` (<https://www.cybertec-postgresql.com/introducing-pg-squeeze-a-postgresql-extension-to-autorebuild-bloated-tables/>), которое умеет перезаписывать таблицу, не блокируя запись в нее.

Наблюдение за работой VACUUM

После этого введения самое время посмотреть на VACUUM в действии. Я включил этот раздел, потому что мой опыт консультирования и технической поддержки пользователей PostgreSQL (<https://www.cybertec-postgresql.com/>) показал,

¹ И операции чтения тоже, что особенно неприятно. – Прим. ред.

что народ в основном имеет весьма смутное представление о том, что происходит в хранилище.

Подчеркну еще раз, что в большинстве случаев VACUUM не уменьшает размер таблицы и не возвращает место файловой системе.

В примере ниже показано, как создать небольшую таблицу со специальными параметрами автоочистки. Первоначально в таблицу записывается 100 000 строк:

```
CREATE TABLE t_test (id int) WITH (autovacuum_enabled = off);
INSERT INTO t_test
  SELECT * FROM generate_series(1, 100000);
```

Отметим, что автоочистку отдельных таблиц можно выключать. Обычно это не рекомендуется, но в некоторых специальных случаях параметр `autovacuum_enabled = off` имеет смысл. Взять, к примеру, таблицы с очень коротким временем жизни. Зачем очищать кортежи, если разработчик заранее знает, что через несколько секунд таблицы не станет? В хранилищах данных так бывает, если таблица используется для хранения временных данных. В нашем примере VACUUM выключено, чтобы в фоновом режиме ничего не происходило. Все, что вы видите, – результат моих действий, а не работы какого-то стороннего процесса.

Для начала получим размер таблицы:

```
test=# SELECT pg_size_pretty(pg_relation_size('t_test'));
pg_size_pretty
-----
3544 kB
(1 строка)
```

Функция `pg_relation_size` возвращает размер таблицы в байтах, а функция `pg_size_pretty` принимает это число и преобразует его в нечто удобочитаемое.

Затем обновим все строки таблицы, воспользовавшись простой командой UPDATE:

```
test=# UPDATE t_test SET id = id + 1;
UPDATE 100000
```

То, что сейчас произойдет, очень важно для понимания PostgreSQL. Движок базы данных должен скопировать все строки. Почему? Во-первых, мы не знаем, завершится ли транзакция успешно, поэтому не вправе перезаписывать данные. А во-вторых, конкурентная транзакция может еще работать со старой версией данных.

Команда UPDATE копирует строки. Логично, что после этого изменения размер таблицы станет больше:

```
test=# SELECT pg_size_pretty(pg_relation_size('t_test'));
pg_size_pretty
-----
7080 kB
(1 строка)
```

После UPDATE можно попытаться вернуть место файловой системе:

```
test=# VACUUM t_test;
VACUUM
```

Но, как уже было сказано, VACUUM, как правило, не возвращает место, а лишь дает возможность использовать его повторно. Таким образом, таблица ничуть не уменьшилась:

```
test=# SELECT pg_size_pretty(pg_relation_size('t_test'));
pg_size_pretty
-----
7080 kB
(1 строка)
```

Однако следующая команда UPDATE не увеличит размер таблицы, т. к. воспользуется свободным местом, уже имеющимся внутри нее. И лишь вторая команда UPDATE приведет к новому росту таблицы, поскольку свободное место исчерпано и нужно дополнительное:

```
test=# UPDATE t_test SET id = id + 1;
UPDATE 1000000
test=# SELECT pg_size_pretty(pg_relation_size('t_test'));
pg_size_pretty
-----
7080 kB
(1 строка)

test=# UPDATE t_test SET id = id + 1;
UPDATE 1000000
test=# SELECT pg_size_pretty(pg_relation_size('t_test'));
pg_size_pretty
-----
10 MB
(1 строка)
```

Если бы мне нужно было выбрать одну-единственную вещь, которую следует запомнить в результате прочтения этой книги, то я бы остановился на этом факте. Знание того, как ведет себя подсистема хранения, – ключ к повышению производительности и администрированию в целом.

Выполним еще несколько запросов:

```
VACUUM t_test;
UPDATE t_test SET id = id + 1;
VACUUM t_test;
```

И снова размер не изменился. Посмотрим, что находится в таблице:

```
test=# SELECT ctid, * FROM t_test ORDER BY ctid DESC;
ctid | id
-----+-----
...
(1327, 46) | 112
```

```
(1327, 45) | 111
(1327, 44) | 110
...
(884, 20) | 99798
(884, 19) | 99797
...
```

Столбец `ctid` содержит информацию о физическом положении строки на диске. Фраза `ORDER BY ctid DESC` позволяет читать таблицу в обратном физическом порядке хранения. А какая нам разница? Дело в том, что в конце таблицы есть очень малые и очень большие значения. В следующей распечатке показано, как изменяется размер таблицы при удалении данных:

```
test=# DELETE FROM t_test
        WHERE id > 99000
        OR id < 1000;

DELETE 1999
test=# VACUUM t_test;
VACUUM
test=# SELECT pg_size_pretty(pg_relation_size('t_test'));
 pg_size_pretty 
-----
3504 kB
(1 строка)
```

Удалено лишь 2 % данных, но размер таблицы сократился на две трети. Причина в том, что если после некоторой точки в таблице `VACUUM` находит только мертвые строки, то она может вернуть место файловой системе. Это единственный случай, когда размер таблицы уменьшается. Конечно, обычные пользователи не могут контролировать физическое расположение данных на диске. Поэтому, скорее всего, после удаления строк размер таблицы не сильно изменится, если только не удаляются все вообще строки.



Но почему в конце таблицы так много малых и больших значений? После первоначальной записи 100 000 строк в таблицу последний блок оказался занят не полностью, поэтому первая команда `UPDATE` допишет в него изменения. При этом конец таблицы, естественно, будет немного перемешан. В данном тщательно сконструированном примере это и является причиной странной картины в конце таблицы.

В реальных приложениях важность этого наблюдения невозможно переоценить. Без понимания механизма хранения нечего и говорить об оптимизации производительности.

Ограничение длительности транзакций с помощью времени жизни снимка

`VACUUM` прекрасно справляется с работой и освобождает место по мере необходимости. Но в какой момент `VACUUM` вправе вычистить строки, превратив их в свободное место? Правило таково: если строку больше никто не может уви-

деть, значит, ее можно освободить. На практике это означает, что все строки, которые не видны самой старой из активных транзакций, можно считать окончательно умершими.

Отсюда также следует, что очень длинные транзакции могут привести к откладыванию очистки на длительный срок. Логическое последствие – разбухание таблицы. Таблицы неумеренно растут, а производительность снижается. По счастью, начиная с версии PostgreSQL 9.6 в базе данных есть полезная возможность, благодаря которой администратор может разумно ограничить длительность транзакции. Администраторам Oracle знакомо сообщение об ошибке «**snapshot too old**» (слишком старый снимок). В PostgreSQL 9.6 это сообщение также присутствует. Но это специально задуманная особенность, а не побочный эффект неправильной конфигурации (как в Oracle).

Чтобы ограничить время жизни снимков, в конфигурационном файле PostgreSQL `postgresql.conf` имеет параметр:

```
old_snapshot_threshold = -1
# 1min-60d; -1 - отключить; 0 - немедленно
```

Если этот параметр установлен, то транзакции будут завершаться с ошибкой по истечении заданного времени. Отметим, что параметр задается на уровне экземпляра, а не в сеансе. Благодаря ограничению возраста транзакции риск возникновения несуразно долгих транзакций заметно уменьшается.

РЕЗЮМЕ

В этой главе мы узнали о транзакциях, блокировке и ее логических последствиях, а также о том, как общая архитектура транзакций в PostgreSQL влияет на подсистему хранения, конкурентность и администрирование. Мы видели, как блокируются строки, и рассмотрели некоторые средства PostgreSQL.

В главе 3 «Использование индексов» мы обсудим один из важнейших аспектов работы базы данных: индексирование. Вы также узнаете об оптимизаторе запросов в PostgreSQL, о различных типах индексов и их поведении.

ВОПРОСЫ

Каково назначение транзакции?

Транзакции – основа основ любой современной реляционной базы данных. Идея в том, чтобы сделать операции атомарными – «все или ничего». Например, если мы хотим удалить миллион строк, то удалены должны быть все – нас не устроит, если парочка строк все-таки останется.

Насколько длинной может быть транзакция в PostgreSQL?

PostgreSQL не налагает никаких ограничений на максимальную длину транзакции. Так что разрешены (почти) бесконечно длинные транзакции, которые изменяют миллиарды строк и содержат сотни миллионов команд.

Что такое изоляция транзакций?

Не все транзакции равны. Поэтому во многих случаях мы должны управлять видимостью данных внутри транзакций. Именно для этого и предназначен механизм изоляции транзакций. Уровень изоляции позволяет сделать именно то, что нам нужно.

Следует ли избегать табличных блокировок?

Да, безусловно. Табличная блокировка запрещает доступ всем остальным пользователям, что может приводить к проблемам с производительностью. Чем большего количества табличных блокировок вы сможете избежать, тем лучше будет работать система.

Какое отношение транзакции имеют к VACUUM?

Транзакции и команда VACUUM тесно связаны. Весь смысл VACUUM в том, чтобы вычистить мертвые строки, которые больше никому не нужны. Если конечный пользователь плохо понимает, что такое транзакции, то неизбежно разбухание таблиц.

Глава 3

Использование индексов

В главе 2 мы говорили о конкурентности и блокировках. В этой главе пришло время пойти в атаку на индексы. Важность данного вопроса невозможно переоценить – индексирование является (и, скорее всего, останется) одной из главных тем в жизни любого программиста баз данных.

После 18 лет профессиональной деятельности в области консультирования и круглосуточной поддержки пользователей PostgreSQL (www.cybertec-postgresql.com) могу с уверенностью сказать: плохие индексы – основной источник низкой производительности. Конечно, нужно правильно настраивать параметры памяти и все такое. Но все это будет тщетно, если индексы используются неправильно. Ничто не заменит отсутствующий индекс. Еще раз подчеркну: без подходящих индексов невозможно достичь высокой производительности, поэтому, столкнувшись с проблемами по этой части, прежде всего проверьте индексы.

Я посвятил целую главу индексированию не без причины. Это позволит осветить как можно больше различных аспектов.

В этой главе рассматриваются следующие темы:

- когда PostgreSQL использует индексы?
- как их учитывает оптимизатор?
- какие есть типы индексов и как они работают?
- применение собственных стратегий индексирования.

Прочитав эту главу, вы будете понимать, как в PostgreSQL использовать индексы с пользой.

Простые запросы и стоимостная модель

В этом разделе мы приступаем к изучению индексов. Нам потребуются тестовые данные. Создадим их:

```
test=# DROP TABLE IF EXISTS t_test;  
DROP TABLE  
test=# CREATE TABLE t_test (id serial, name text);  
CREATE TABLE  
test=# INSERT INTO t_test (name) SELECT 'hans'  
FROM generate_series(1, 2000000);
```

```
INSERT 0 2000000
test=# INSERT INTO t_test (name) SELECT 'paul'
        FROM generate_series(1, 2000000);
INSERT 0 2000000
```

В первой строке создается простая таблица с двумя столбцами. В первом, автоинкрементном, столбце хранятся последовательные числа, а во втором – статические строки.

i Функция `generate_series` порождает числа от 1 до 2 млн. Следовательно, в этом примере будет 2 млн строк с именем `hans` и 2 млн строк с именем `paul`.

Всего добавлено 4 млн строк:

```
test=# SELECT name, count(*) FROM t_test GROUP BY 1;
 name | count
-----+-----
 hans | 2000000
 paul | 2000000
(2 строки)
```

У этих 4 млн строк есть ряд интересных свойств, которые понадобятся нам в данной главе. Идентификаторы монотонно возрастают, а различных имен всего два.

Выполним простой запрос:

```
test=# \timing
Секундомер включен.
test=# SELECT * FROM t_test WHERE id = 432332;
 id | name
-----+-----
432332 | hans
(1 строка)
Время: 176.949 мс
```

Мы воспользовались командой `timing`, которая заставляет `psql` показывать время выполнения запроса.

i Это не истинное время выполнения на сервере, а время, измеренное `psql`. Если запрос очень короткий, то большая часть времени приходится на сетевые задержки, помните об этом.

Использование команды EXPLAIN

В этом примере чтение 4 млн строк заняло более 100 мс. С точки зрения производительности, это катастрофа. Чтобы понять, в чем проблема, PostgreSQL предлагает команду `EXPLAIN`:

```
test=# \h EXPLAIN
Команда: EXPLAIN
Описание: показать план выполнения оператора
Синтаксис:
```

EXPLAIN [(параметр [, ...])] оператор
 EXPLAIN [ANALYZE] [VERBOSE] оператор

где допустимый параметр:

```
ANALYZE [ boolean ]
VERBOSE [ boolean ]
COSTS [ boolean ]
BUFFERS [ boolean ]
TIMING [ boolean ]
FORMAT { TEXT | XML | JSON | YAML }
```

Если у вас возникло ощущение, что с запросом что-то не так, EXPLAIN поможет понять, в чем причина.

Вот как это работает:

```
test=# EXPLAIN SELECT * FROM t_test WHERE id = 432332;
               QUERY PLAN
-----
Gather (cost=1000.00..43463.92 rows=1 width=9)
  Workers Planned: 2
  -> Parallel Seq Scan on t_test (cost=0.00..42463.82 rows=1 width=9)
      Filter: (id = 432332)
(4 строки)
```

В распечатке показан план выполнения. В PostgreSQL выполнение SQL-команд состоит из четырех этапов:

- 1) **синтаксический анализатор** (parser) ищет синтаксические ошибки и очевидные проблемы;
- 2) система **перезаписывания** применяет правила (представления и прочее);
- 3) **оптимизатор** решает, как наиболее эффективно выполнить запрос, и формирует план выполнения;
- 4) предложенный оптимизатором план используется **исполнителем** для получения результата.

Цель EXPLAIN – показать план эффективного выполнения запроса, разработанный оптимизатором. В примере выше PostgreSQL будет производить параллельный последовательный просмотр. Это означает, что два исполнителя будут работать сообща, применяя условие фильтрации. Затем частичные результаты объединяются узлом **сбора** – компонентом, появившимся в версии PostgreSQL 9.6 (это часть инфраструктуры параллельного выполнения запросов). Взглянув на план более пристально, мы увидим, сколько строк PostgreSQL ожидает получить на каждом шаге выполнения плана (в данном случае rows = 1, т. е. будет возвращена одна строка).



В версиях PostgreSQL от 9.6 до 11.0 количество параллельных исполнителей определяется размером таблицы. Чем больше операция, тем больше исполнителей запустит сервер. Для совсем небольших таблиц распараллеливание не используется, поскольку накладные расходы были бы слишком велики.

Распараллеливание – не приговор. Всегда можно уменьшить количество параллельных исполнителей, чтобы смоделировать поведение PostgreSQL до выхода версии 9.6. Для этого достаточно присвоить 0 следующему параметру:

```
test=# SET max_parallel_workers_per_gather TO 0;
SET
```

Отметим, что у этого изменения нет побочных эффектов – оно действует только в текущем сеансе. Разумеется, можно изменить и файл `postgresql.conf`, но я не рекомендую так поступать, потому что вы потеряете важные преимущества, которые несет с собой распараллеливание запросов.

Стоимостная модель PostgreSQL

Если используется только один процессор, то план выполнения будет выглядеть следующим образом:

```
test=# EXPLAIN SELECT * FROM t_test WHERE id = 432332;
               QUERY PLAN
-----
Seq Scan on t_test (cost=0.00..71622.00 rows=1 width=9)
  Filter: (id = 432332)
(2 строки)
```

PostgreSQL последовательно читает (Seq Scan – последовательный просмотр) всю таблицу и применяет фильтр. Ожидается, что операция будет стоить 71622 штрафных балла. Но что это значит? Штрафные баллы (или стоимость) – это абстрактное понятие. Они нужны для сравнения различных способов выполнения запроса. Если запрос можно выполнить разными способами, то PostgreSQL выберет план, обещающий наименьшую стоимость. Вопрос в том, откуда взялось число 71622.

Объясняю:

```
test=# SELECT pg_relation_size('t_test') / 8192.0;
?column?
-----
21622.000000
(1 строка)
```

Функция `pg_relation_size` возвращает размер таблицы в байтах. В данном примере таблица содержит 21622 блока (по 8 К каждый). Следуя стоимостной модели, PostgreSQL прибавляет к стоимости единицу для каждого блока, который предстоит прочесть последовательно. Значение 1 берется из следующего параметра:

```
test=# SHOW seq_page_cost;
seq_page_cost
-----
1
(1 строка)
```

Однако недостаточно прочитать пару блоков с диска. Нужно еще применить фильтр и обработать строки с помощью процессора. Стоимости этих операций определяются параметрами:

```
test=# SHOW cpu_tuple_cost;
      cpu_tuple_cost
```

```
-----
0.01
(1 строка)
```

```
test=# SHOW cpu_operator_cost;
      cpu_operator_cost
```

```
-----
0.0025
(1 строка)
```

В итоге получается:

```
test=# SELECT 21622*1 + 4000000*0.01 + 4000000*0.0025;
      ?column?
```

```
-----
71622.0000
(1 строка)
```

Это и есть число, показанное в плане. Общая стоимость состоит из стоимости процессора и стоимости ввода-вывода, которые объединяются в одно число. Важно понимать, что стоимость не имеет никакого отношения к реальному вычислению, поэтому перевести ее в миллисекунды невозможно. Число, вычисляемое планировщиком, – всего лишь оценка.

Разумеется, есть и другие параметры. В PostgreSQL имеются специальные параметры для операций, в которых участвует индекс:

- `random_page_cost = 4`: если PostgreSQL использует индекс, то обычно производится много операций ввода-вывода с произвольной выборкой. При наличии традиционных вращающихся дисков операции чтения с произвольной выборкой гораздо важнее операций последовательного чтения, поэтому PostgreSQL сопоставляет им большую стоимость. Но с появлением SSD-дисков различие между чтением с произвольной выборкой и последовательным чтением исчезло, так что имеет смысл положить `random_page_cost = 1` в файле `postgresql.conf`;
- `cpu_index_tuple_cost = 0.005`: если используются индексы, то PostgreSQL считает, что на процессор приходится большая нагрузка.

Если выполнение запроса распараллеливается, то учитываются дополнительные параметры:

- `parallel_tuple_cost = 0.1`: это стоимость передачи одного кортежа от процесса параллельного исполнителя другому процессу. Параметр учитывает накладные расходы на перемещение строк внутри инфраструктуры;
- `parallel_setup_cost = 1000.0`: стоимость запуска исполнителя. Разумеется, запуск процессов для выполнения запросов обходится не бесплатно,

так что этот параметр – попытка включить в модель затраты, связанные с управлением процессами;

- `min_parallel_table_scan_size = 8 MB`: минимальный размер таблицы, при котором рассматривается возможность распараллеливания. Чем больше таблица, тем больше процессоров задействует PostgreSQL. Для запуска дополнительного исполнителя размер таблицы должен утроиться;
- `min_parallel_index_scan_size = 512 KB`: размер индекса, начиная с которого рассматривается параллельный просмотр.

Развертывание простых индексов

Запуск дополнительных исполнителей для просмотра постоянно растущих таблиц не всегда можно считать решением. Читать всю таблицу, чтобы найти всего одну строку, – обычно не лучшая мысль.

Поэтому имеет смысл создать индексы:

```
test=# CREATE INDEX idx_id ON t_test (id);
CREATE INDEX
test=# SELECT * FROM t_test WHERE id = 43242;
   id | name
-----+-----
 43242 | hans
(1 строка)
Время: 0.259 мс
```

В PostgreSQL используется В-дерево Лемана–Яо, обеспечивающее высокую степень конкурентности (<https://www.csd.uoc.gr/~hy460/pdf/p650-lehman.pdf>). С дополнительными оптимизациями, специфичными для PostgreSQL, оно дает великолепную производительность. Главное – что дерево Лемана–Яо позволяет одновременно выполнять много операций (чтения и записи) над одним и тем же индексом, что очень сильно повышает пропускную способность.

Однако индексы – не бесплатное удовольствие:

```
test=# \di+
                                Список отношений
 Схема | Имя | Тип | Владелец | Таблица | Размер | Описание
-----+-----+-----+-----+-----+-----+-----
 public | idx_id | индекс | hs      | t_test  | 86 MB  |
(1 строка)
```

Как видим, наш индекс, содержащий 4 млн строк, занимает 86 МБ на диске. Кроме того, замедляется запись в таблицу, потому что индекс всегда должен быть синхронизирован с данными.

Иначе говоря, построив над таблицей 20 индексов, вы должны будете обновлять их при каждой команде INSERT, что существенно замедляет запись.

i Начиная с версии 11 PostgreSQL поддерживает параллельное создание индексов. Теперь для построения индекса можно задействовать более одного процессора, что заметно ускоряет процесс. Пока что таким образом можно строить только обычные В-деревья,

другие типы индексов еще не поддерживаются. Но в будущем ситуация, скорее всего, изменится. Степень параллелизма управляется параметром `max_parallel_maintenance_workers`, равным максимально разрешенному числу параллельных процессов.

Сортировка результатов

Индексы типа В-дерево используются не только для поиска строк, но и для передачи отсортированных данных следующему шагу процесса:

```
test=# EXPLAIN SELECT *
FROM t_test
ORDER BY id DESC
LIMIT 10;

               QUERY PLAN
-----
Limit (cost=0.43..0.74 rows=10 width=9)
-> Index Scan Backward using idx_id on t_test
    (cost=0.43..125505.43 rows=4000000 width=9)
(2 строки)
```

В данном случае индекс уже возвращает данные в правильном порядке, поэтому дополнительно сортировать весь набор не нужно. Для ответа на запрос достаточно прочесть последние десять строк индекса. На практике это означает, что на поиск первых N строк таблицы уйдут доли миллисекунды.

Но `ORDER BY` – не единственная операция, которой нужны отсортированные данные. Функции `min` и `max` также нуждаются в сортировке, поэтому индекс может ускорить и эти операции. Например:

```
test=# EXPLAIN SELECT min(id), max(id) FROM t_test;

               QUERY PLAN
-----
Result (cost=0.93..0.94 rows=1 width=8)
InitPlan 1 (returns $0)
-> Limit (cost=0.43..0.46 rows=1 width=4)
-> Index Only Scan using idx_id on t_test
    (cost=0.43..135505.43 rows=4000000 width=4)
    Index Cond: (id IS NOT NULL)
InitPlan 2 (returns $1)
-> Limit (cost=0.43..0.46 rows=1 width=4)
-> Index Only Scan Backward using idx_id on t_test t_test_1
    (cost=0.43..135505.43 rows=4000000 width=4)
    Index Cond: (id IS NOT NULL)
(9 строк)
```

В PostgreSQL индекс (точнее, В-дерево) можно читать в прямом и обратном порядке. Нас сейчас интересует тот факт, что В-дерево можно рассматривать как отсортированный список. Поэтому не должно вызывать удивления, что наименьшее значение находится в начале, а наибольшее – в конце, и, значит, `min` и `max` – идеальные кандидаты на ускорение. Стоит также отметить, что в этом случае к базовой таблице обращаться вообще не придется.

Использование нескольких индексов одновременно

До настоящего момента мы видели, как используется один индекс. Но на практике этого зачастую отнюдь недостаточно.

PostgreSQL позволяет использовать несколько индексов в одном запросе. Понятно, что это имеет смысл, когда одновременно опрашивается несколько столбцов. Но это не единственный случай. Бывает, что один индекс многократно используется для обработки одного столбца. Рассмотрим пример.

```
test=# explain SELECT * FROM t_test WHERE id = 30 OR id = 50;
               QUERY PLAN
```

```
-----
Bitmap Heap Scan on t_test (cost=8.88..16.85 rows=2 width=9)
  Recheck Cond: ((id = 30) OR (id = 50))
    -> BitmapOr (cost=8.88..8.88 rows=2 width=0)
      -> Bitmap Index Scan on idx_idv (cost=0.00..4.44 rows=1 width=0)
          Index Cond: (id = 30)
      -> Bitmap Index Scan on idx_id (cost=0.00..4.44 rows=1 width=0)
          Index Cond: (id = 50)
```

(7 строк)

Здесь столбец `id` необходим дважды. Сначала мы ищем в нем значение 30, а затем 50. Как видим, PostgreSQL прибегла к просмотру по битовой карте.



Просмотр по битовой карте – не то же самое, что битовый индекс, о котором, возможно, знают читатели, хорошо знакомые с Oracle. Это совершенно разные вещи, не имеющие между собой ничего общего. Битовые индексы – один из типов индексов в Oracle, а просмотр по битовой карте – метод просмотра.

Идея просмотра по битовой карте заключается в том, что PostgreSQL просматривает первый индекс и собирает список блоков (т. е. страниц таблицы), содержащих данные. Затем просматривается следующий индекс и снова строится список блоков. Этот процесс производится столько раз, сколько необходимо. В случае `OR` списки потом объединяются, так что мы получаем большой список блоков, содержащих данные. После этого мы выбираем эти блоки из таблицы.

Проблема в том, что PostgreSQL выбрала гораздо больше данных, чем необходимо. В нашем случае запрос ищет две строки, а в результате просмотра по битовой карте может быть возвращено два блока. Поэтому исполнитель еще раз проверяет все возвращенные строки, отфильтровывая те, что не удовлетворяют условиям¹.

Просмотр по битовой карте работает также для связки `AND` и комбинации связок `AND` и `OR`. Но если PostgreSQL видит условие `AND`, ей необязательно переходить к просмотру по битовой карте. Предположим, что в запросе требуется найти среди всех проживающих в Австрии человека с конкретным идентификатором. В данном случае нет смысла использовать два индекса, потому что после поиска по идентификатору останется совсем немного данных. Просмотр обоих

¹ Если битовая карта помещается в отведенную ей память, то составляется список не блоков, а конкретных строк. Перепроверка в таком случае не выполняется, хоть и отображается в плане. – *Прим. ред.*

индексов окажется намного дороже, т. к. в Австрии проживает 8 млн человек (включая меня), а читать столько строк, чтобы найти всего одного человека, глупо с точки зрения производительности. А теперь хорошая новость – оптимизатор PostgreSQL принимает такие решения за вас, сравнивая стоимости различных вариантов и использования потенциальных индексов.

Эффективное использование просмотра по битовой карте

Естественно возникает вопрос: когда просмотр по битовой карте выгоден и когда его выбирает оптимизатор? На мой взгляд, есть всего два полезных сценария:

- чтобы избежать многократной выборки одного и того же блока;
- чтобы объединить относительно плохие условия.

Первый случай очень распространен. Допустим, мы ищем людей, говорящих на некотором языке. Для определенности предположим, что на этом языке говорят 10% людей. Поиск по индексу означает, что каждый блок таблицы нужно будет просмотреть несколько раз, потому в одном блоке может оказаться много искомых людей. А просмотр по битовой карте гарантирует, что один блок используется только один раз, что, конечно, повышает производительность.

Второй случай – совместное использование относительно слабых критериев. Допустим, что мы ищем людей от 20 до 30 лет, носящих желтые рубашки. Может случиться так, что у 15% всех людей возраст попадает в указанный диапазон и 15% людей носят желтые рубашки. Просматривать таблицу последовательно дорого, поэтому PostgreSQL, возможно, решит использовать два индекса, т. к. конечный результат может содержать всего 1% данных. Может получиться, что просмотр двух индексов дешевле чтения всех данных.

В версии PostgreSQL 10.0 поддерживается параллельный просмотр по битовой карте. Обычно просмотр по битовой карте применяется для сравнительно дорогих запросов. Поэтому реализация распараллеливания в этой области – огромный шаг вперед и определенно дает выгоды.

Разумное использование индексов

До сих пор применение индексов выглядело как Святой Грааль, который всегда волшебным образом повышает производительность. Однако это не так. В некоторых случаях индексы бесполезны.

Прежде чем переходить к обсуждению этой темы, создадим подходящую структуру данных. Напомним, что все идентификаторы уникальны, а различных имен только два.

```
test=# \d t_test
```

```

        Таблица "public.t_test"
  Столбец | Тип      | Модификаторы
-----+-----+-----
 id       | integer | not null default nextval('t_test_id_seq'::regclass)
 name     | text    |
Индексы:
 "idx_id" btree (id)
```

Сейчас построен один индекс по столбцу id. Далее мы собираемся выполнить запрос по столбцу name. Но сначала построим индекс по этому столбцу:

```
test=# CREATE INDEX idx_name ON t_test (name);
CREATE INDEX
```

Теперь посмотрим, правильно ли используется этот индекс:

```
test=# EXPLAIN SELECT * FROM t_test WHERE name = 'hans2';
               QUERY PLAN
-----
Index Scan using idx_name on t_test (cost=0.43..4.45 rows=1 width=9)
    Index Cond: (name = 'hans2'::text)
(2 строки)
```

Как и ожидалось, PostgreSQL решила использовать индекс. Но заметим, что в запросе указано имя hans2, а этого имени нет в таблице, и план запроса ясно отражает этот факт – rows=1 как раз и показывает, что планировщик ожидает, что этот запрос вернет очень мало строк.



В таблице нет ни одной подходящей строки, но PostgreSQL никогда не дает оценки 0, потому что это существенно усложнило бы последующие операции, сделав полезные вычисления других узлов плана практически невозможными.

Посмотрим, что произойдет, если поискать другие данные:

```
test=# EXPLAIN SELECT *
      FROM t_test
      WHERE name = 'hans'
             OR name = 'paul';
               QUERY PLAN
-----
Seq Scan on t_test (cost=0.00..81622.00 rows=3000011 width=9)
    Filter: ((name = 'hans'::text) OR (name = 'paul'::text))
(2 строки)
```

В этом случае PostgreSQL сразу переходит к последовательному просмотру. Но почему система игнорирует все индексы? Ответ прост: кроме hans и paul, в таблице нет никаких имен (PostgreSQL знает об этом из статистики). Поэтому PostgreSQL делает вывод, что таблицу все равно придется читать целиком. А раз так, то нет смысла читать и индекс, и таблицу – ведь чтения одной таблицы достаточно.

Иными словами, PostgreSQL не использует индекс только потому, что он существует. Индексы используются лишь тогда, когда это имеет смысл. Если число строк мало, то PostgreSQL будет рассматривать просмотр по битовой карте и обычный просмотр индекса:

```
test=# EXPLAIN SELECT *
      FROM t_test
      WHERE name = 'hans2'
             OR name = 'paul2';
               QUERY PLAN
```

```

-----
Bitmap Heap Scan on t_test (cost=8.88..12.89 rows=1 width=9)
  Recheck Cond: ((name = 'hans2'::text) OR (name = 'paul2'::text))
    -> BitmapOr (cost=8.88..8.88 rows=1 width=0)
      -> Bitmap Index Scan on idx_name (cost=0.00..4.44 rows=1 width=0)
        Index Cond: (name = 'hans2'::text)
      -> Bitmap Index Scan on idx_name (cost=0.00..4.44 rows=1 width=0)
        Index Cond: (name = 'paul2'::text)

```

Самое важное, что следует уяснить из этого примера, – тот факт, что планы выполнения зависят от входных значений, а не являются статическими. В реальных ситуациях изменение плана выполнения может стать причиной неожиданных результатов во время работы.

ПОВЫШЕНИЕ БЫСТРОДЕЙСТВИЯ С ПОМОЩЬЮ КЛАСТЕРИЗОВАННЫХ ТАБЛИЦ

В этом разделе мы поговорим о силе корреляции и кластеризованных таблиц. В чем идея? Предположим, что мы хотим прочитать целую область данных: временной диапазон, блок, множество идентификаторов и т. д.

Время работы таких запросов зависит от количества данных и их физического расположения на диске. Так что даже если в двух системах выполняется один и тот же запрос, возвращающий одни и те же данные, время ответа может различаться, поскольку физическая структура не одинакова. Приведем пример.

```

test=# EXPLAIN (analyze true, buffers true, timing true)
SELECT *
FROM t_test
WHERE id < 10000;

```

QUERY PLAN

```

-----
Index Scan using idx_id on t_test
  (cost=0.43..370.87 rows=10768 width=9)
  (actual time=0.011..2.897 rows=9999 loops=1)
  Index Cond: (id < 10000)
  Buffers: shared hit=85
Planning time: 0.078 ms
Execution time: 4.081 ms
(5 строк)

```

Как вы, наверное, помните, данные были загружены последовательно, в порядке возрастания идентификаторов. Поэтому можно ожидать, что и на диске они располагаются последовательно. Это действительно так, если данные загружались в пустую таблицу с автоинкрементным столбцом.

Мы уже видели команду EXPLAIN в действии. А в данном случае использована команда EXPLAIN (analyze true, buffers true, timing true). Если задан параметр

analyze, то система не только покажет план выполнения, но еще и выполнит запрос и покажет, что произошло на самом деле.

EXPLAIN в режиме analyze идеально подходит для сравнения оценок планировщика с реальностью. Это лучший способ узнать, прав планировщик или сбился с пути истинного. Параметр buffers true сообщает, сколько 8-килобайтных блоков затронул запрос. В данном случае затронуто 85 блоков. Слова shared hit означают, что данные взяты из кеша ввода-вывода PostgreSQL (разделяемые буферы). Всего на выборку данных у PostgreSQL ушло примерно 4 мс.

А что произойдет, если данные таблицы распределены по диску в случайном порядке? Изменится ли ситуация?

Чтобы создать таблицу, содержащую те же данные, но в случайном порядке, можно просто воспользоваться фразой ORDER BY random(). Тогда данные на диске будут перемешаны.

```
test=# CREATE TABLE t_random AS SELECT * FROM t_test ORDER BY random();
SELECT 4000000
```

Чтобы сравнение было честным, проиндексируем тот же столбец:

```
test=# CREATE INDEX idx_random ON t_random (id);
CREATE INDEX
```

Для правильной работы PostgreSQL нужна статистика оптимизатора. Она показывает, сколько всего данных, как распределены значения и коррелированы ли данные на диске. Чтобы еще ускорить дело, я предварительно вызвал команду VACUUM, хотя более подробно она будет обсуждаться ниже.

```
test=# VACUUM ANALYZE t_random;
VACUUM
```

Теперь выполним тот же запрос, что и раньше:

```
test=# EXPLAIN (analyze true, buffers true, timing true)
SELECT * FROM t_random WHERE id < 10000;
QUERY PLAN
```

```
-----
Bitmap Heap Scan on t_random
  (cost=203.27..18431.86 rows=10689 width=9)
  (actual time=5.087..13.822 rows=9999 loops=1)
    Recheck Cond: (id < 10000)
    Heap Blocks: exact=8027
    Buffers: shared hit=8057
  -> Bitmap Index Scan on idx_random
    (cost=0.00..200.60 rows=10689 width=0)
    (actual time=3.558..3.558 rows=9999 loops=1)
    Index Cond: (id < 10000)
    Buffers: shared hit=30
Planning time: 0.075 ms
Execution time: 14.411 ms
(9 строк)
```

Сделаем два замечания. Прежде всего пришлось прочитать аж 8057 блоков, и время работы взлетело до 14 с лишним миллисекунд. Единственное, что немного спасло производительность, – тот факт, что данные читались из памяти, а не с диска. А представьте, что нам пришлось бы 8057 раз обратиться к диску для ответа на этот запрос. Произошла бы катастрофа, потому что из-за времени ожидания диска все работало бы существенно медленнее.

Но это еще не все. Видно, что и план изменился. Теперь PostgreSQL использует просмотр по битовой карте, а не обычный просмотр индекса. Так делается, чтобы уменьшить количество необходимых запросу блоков и тем самым предотвратить еще худшее развитие событий.

Откуда планировщик знает, как данные хранятся на диске? Системное представление `pg_stats` содержит всю статистическую информацию о содержимом столбцов. Следующий запрос выводит интересные нас сведения:

```
test=# SELECT tablename, attname, correlation
        FROM pg_stats
        WHERE tablename IN ('t_test', 't_random')
        ORDER BY 1, 2;
```

tablename	attname	correlation
t_random	id	-0.0114944
t_random	name	0.493675
t_test	id	1
t_test	name	1

(4 строки)

Как видим, PostgreSQL учитывает все столбцы. Это представление заполняется командой `ANALYZE`, играющей ключевую роль для производительности:

```
test=# \h ANALYZE
Команда    ANALYZE
```

Описание: собрать статистику о базе данных

Синтаксис:

```
ANALYZE [ VERBOSE ] [ имя_таблицы [ ( имя_столбца [, ...] ) ] ]
```

Обычно `ANALYZE` выполняется в фоновом режиме процессом автоочистки, который мы рассмотрим ниже.

Но вернемся к нашему запросу. Как видим, в обеих таблицах есть два столбца (`id` и `name`). Для столбца `t_test.id` корреляция равна 1, это означает, что следующее значение зависит от предыдущего – в данном случае значения просто возрастают. То же относится и к столбцу `t_test.name`. Сначала идут все записи, содержащие имя `hans`, а затем записи с именем `raul`. Все одинаковые имена хранятся вместе.

В таблице `t_random` ситуация совершенно иная. Корреляция, близкая к нулю, означает, что данные перемешаны. Видно также, что для столбца `name` корреляция близка к 0.5. Это значит, что никакого порядка в следовании одинаковых имен не наблюдается – имя постоянно меняется при чтении таблицы в физическом порядке.

И почему же все это приводит к резкому увеличению количества блоков, затронутых запросом? Ответ понятен. Если нужные нам данные не хранятся вместе, а разбросаны по всей таблице, то для выборки одного и того же объема информации понадобится прочитать больше блоков, что ведет к снижению производительности.

Кластеризация таблиц

В PostgreSQL имеется команда `CLUSTER`, позволяющая перезаписать таблицу в нужном порядке. Мы можем указать некоторый индекс и попросить, чтобы система сохранила данные в том же порядке, что в этом индексе:

```
test=# \h CLUSTER
Команда: CLUSTER
Описание: перегруппировать таблицу по индексу
Синтаксис:
CLUSTER [VERBOSE] имя_таблицы [ USING имя_индекса ]
CLUSTER [VERBOSE]
```

Команда `CLUSTER` существует уже много лет и отлично справляется со своей задачей. Но прежде чем слепо запускать ее в производственной системе, следует помнить о нескольких моментах:

- команда `CLUSTER` блокирует таблицу на все время работы. Нельзя будет ни вставлять, ни изменять данные. В производственной системе это может оказаться неприемлемым;
- данные можно перегруппировать только в соответствии с одним индексом. Нельзя одновременно упорядочить таблицу по почтовому индексу, имени, идентификатору, дню рождения и т. д. Следовательно, выполнять `CLUSTER` имеет смысл только тогда, когда имеется преобладающий критерий поиска;
- имейте в виду, что приведенный выше пример демонстрирует худший случай. На практике различие в производительности между кластеризованной и некластеризованной таблицами зависит от рабочей нагрузки, объема выбираемых данных, коэффициента попадания в кеш и многих других вещей;
- состояние кластеризации не поддерживается при внесении в таблицу изменений в процессе выполнения типовых операций. Со временем корреляция может ухудшаться.

Ниже приведен пример выполнения команды `CLUSTER`:

```
test=# CLUSTER t_random USING idx_random;
CLUSTER
```

Время кластеризации зависит от размера таблицы.

Просмотр только индекса

До сих пор мы говорили о том, когда индекс используется, а когда нет. Мы так же обсудили просмотр по битовым картам.

Однако этим тема индексирования не исчерпывается. Следующие два примера различаются несильно, но разница в производительности может быть очень заметной. Вот первый запрос:

```
test=# EXPLAIN SELECT * FROM t_test WHERE id = 34234;
               QUERY PLAN
```

```
-----
Index Scan using idx_id on t_test
  (cost=0.43..8.45 rows=1 width=9)
  Index Cond: (id = 34234)
```

Ничего необычного. PostgreSQL использует индекс, чтобы найти одну строку. А что произойдет, если мы выберем только один столбец?

```
test=# EXPLAIN SELECT id FROM t_test WHERE id = 34234;
               QUERY PLAN
```

```
-----
Index Only Scan using idx_id on t_test
  (cost=0.43..8.45 rows=1 width=4)
  Index Cond: (id = 34234)
(2 строки)
```

Теперь в плане запроса мы видим не просмотр индекса (index scan), а просмотр только индекса (index-only scan). В нашем примере индексирован столбец id, поэтому его содержимое, естественно, хранится в индексе. Как правило, нет смысла обращаться к таблице, если все данные можно и так взять из индекса. Обращение к таблице нужно только (ну, почти только) в том случае, когда запрашиваются дополнительные поля, чего в данном случае нет. Поэтому просмотр только индекса обещает значительно более высокую производительность, чем обычный просмотр индекса.

На практике иногда даже имеет смысл включить дополнительный столбец в индекс и получить выгоду от этой возможности. В MS SQL такие индексы с дополнительными столбцами называются **покрывающими**. Начиная с версии PostgreSQL 11 аналогичную функциональность можно получить, включив в команду CREATE INDEX ключевое слово INCLUDE.

Дополнительные свойства B-ДЕРЕВЬЕВ

Индексирование в PostgreSQL – обширная тема, связанная со многими аспектами работы базы данных. Как уже было сказано, индексирование – ключ к производительности. Без подходящих индексов ни о какой производительности не может быть и речи. Поэтому возможности, связанные с индексированием, стоит изучить повнимательнее.

Комбинированные индексы

Меня как человека, профессионально оказывающего техническую поддержку PostgreSQL, часто спрашивают о различии между комбинированными и индивидуальными индексами. В данном разделе я постараюсь пролить свет на этот вопрос.

Общее правило таково: если один индекс может ответить на ваш запрос, то и не надо искать от добра добра. Но невозможно же проиндексировать все возможные комбинации столбцов, по которым ищут пользователи. В таком случае имеет смысл воспользоваться свойствами комбинированных индексов, чтобы получить максимум возможного.

Рассмотрим таблицу с тремя столбцами: `postal_code`, `last_name`, `first_name`. Подобный комбинированный индекс можно было бы использовать в телефонном справочнике. Сначала данные сортируются по местоположению, а в пределах одного местоположения – по фамилии и имени.

В таблице ниже показано, какие операции возможны для этого индекса по трем столбцам.

Запрос	Возможно?	Замечания
<code>postal_code = 2700</code> <code>AND last_name = 'Schönig'</code> <code>AND first_name = 'Hans'</code>	Да	Идеальная ситуация для этого индекса
<code>postal_code = 2700</code> <code>AND last_name = 'Schönig'</code>	Да	Без ограничений
<code>last_name = 'Schönig'</code> <code>AND postal_code = 2700</code>	Да	PostgreSQL просто поменяет местами условия
<code>postal_code = 2700</code>	Да	То же самое, что индекс по <code>postal_code</code> ; комбинированный индекс просто занимает больше места на диске
<code>first_name = 'Hans'</code>	Да, но это другой сценарий	PostgreSQL больше не может воспользоваться отсортированностью этого индекса. Но в редких случаях (обычно для таблиц с очень большим количеством столбцов) PostgreSQL может просмотреть весь индекс, если это не дороже чтения всей таблицы

Если столбцы индексированы по отдельности, то, скорее всего, планировщик выберет просмотр по карте памяти. Разумеется, один специализированный индекс лучше.

Добавление функциональных индексов

До сих пор мы видели, как индексировать само содержимое столбца. Но не всегда это именно то, что нам нужно. Поэтому PostgreSQL допускает создание функциональных индексов. Идея очень проста: хранить в индексе не значение, а результат выполнения функции.

В примере ниже показано, как построить индекс по косинусу столбца `id`:

```
test=# CREATE INDEX idx_cos ON t_random (cos(id));
CREATE INDEX
test=# ANALYZE;
ANALYZE
```

Всего-то и нужно, что поместить функцию в список столбцов. Конечно, это работает не для всякой функции. Разрешаются лишь функции с неизменяе-

мым результатом – такие, которые возвращают одинаковое значение при одних и тех же аргументах.

```
test=# SELECT age('2010-01-01 10:00:00'::timestampz);
          age
-----
6 years 9 mons 14:00:00
(1 строка)
```

Функции типа `age` не годятся для индексирования, поскольку их результат может изменяться в зависимости от времени. Действительно, со временем возраст изменяется. PostgreSQL явно запрещает использовать функции, которые могут возвращать разные результаты при одних и тех же аргументах. Функция `cos` в этом отношении безопасна, поскольку косинус значения будет одинаковым что сегодня, что через тысячу лет.

Для проверки индекса я написал простой запрос, показывающий, что происходит:

```
test=# EXPLAIN SELECT * FROM t_random WHERE cos(id) = 10;
          QUERY PLAN
-----
Index Scan using idx_cos on t_random (cost=0.43..8.45 rows=1 width=9)
  Index Cond: (cos((id)::double precision) = '10'::double precision)
(2 строки)
```

Как и следовало ожидать, функциональный индекс используется, как любой другой.

Уменьшение занятого места на диске

Индексирование – вещь хорошая, его главная цель – максимально ускорить выполнение запросов. Но, как и все хорошие вещи, оно имеет свою цену: потребление места на диске. Чтобы вершить свою магию, индекс должен надлежащим образом организовать хранение значений. Если в таблице 10 млн целых чисел, то построенный над ней индекс будет содержать все эти 10 млн чисел плюс накладные расходы.

В В-дереве хранятся указатели на каждую строку таблицы, и, разумеется, это не бесплатно. Чтобы узнать, сколько места занимают индексы, выполните в `psql` команду `di+`:

```
test=# \di+
```

Список отношений					
Схема	Имя	Тип	Владелец	Таблица	Размер
public	idx_cos	индекс	hs	t_random	86 MB
public	idx_id	индекс	hs	t_test	86 MB
public	idx_name	индекс	hs	t_test	86 MB
public	idx_random	индекс	hs	t_random	86 MB

(4 строки)

В моей базе данных на хранение этих индексов ушло аж 344 МБ. А теперь сравним это с местом, занятым базовыми таблицами:

```
test=# \d+
```

Схема	Имя	Список отношений			Владелец	Размер
		Тип				
public	t_random	таблица			hs	169 MB
public	t_test	таблица			hs	169 MB
public	t_test_id_seq	последовательность			hs	8192 bytes

(3 строки)

Обе таблицы вместе занимают всего 338 МБ. Иными словами, для индексов требуется больше места, чем для самих данных. На практике это обычная и весьма вероятная ситуация. Недавно я был у клиента компании Cybertec в Германии и видел базу данных, где 64% места занимали индексы, которые никогда не использовались (ни разу за несколько месяцев). Так что избыточное индексирование, как и недостаточное, чревато проблемами. Напомню, что индексы не просто потребляют место. Каждая команда INSERT и UPDATE должна обновлять все индексы. В экстремальных случаях, как в нашем примере, это значительно снижает производительность.

Если в таблице немного различных значений, то можно воспользоваться частичными индексами:

```
test=# DROP INDEX idx_name;
DROP INDEX
test=# CREATE INDEX idx_name ON t_test (name)
      WHERE name NOT IN ('hans', 'paul');
CREATE INDEX
```

В этом случае большая часть строк исключена из индекса, так что нам на радость останется маленький и эффективный индекс:

```
test=# \di+ idx_name
```

Схема	Имя	Список отношений				Размер
		Тип	Владелец	Таблица		
public	idx_name	индекс	hs	t_test		8192 bytes

(1 строка)

Заметим, что исключать имеет смысл только очень часто встречающиеся значения, составляющие большую часть таблицы (по крайней мере 25%). Идеальные кандидаты на роль частичных индексов – пол (в предположении, что большинство людей – мужчины или женщины), национальность (в предположении, что большинство людей в вашей стране одной национальности) и т. д. Разумеется, для применения подобных трюков нужно очень хорошо знать свои данные, но иногда это окупается.

Добавление данных во время индексирования

Создать индекс легко. Но имейте в виду, что во время его построения модифицировать таблицу нельзя. Команда CREATE INDEX захватывает блокировку типа SHARE на всю таблицу, что препятствует любым изменениям. Для маленьких

таблиц в этом, конечно, нет ничего страшного, но при работе с большими таблицами в производственной системе возникают проблемы. Для индексирования терабайта данных нужно немало времени, поэтому блокировка таблицы надолго может стать источником неприятностей.

Решение дает команда `CREATE INDEX CONCURRENTLY`. Индекс будет строиться намного дольше (обычно, по крайней мере, в два раза), но зато в это время с таблицей можно работать. Приведем пример:

```
test=# CREATE INDEX CONCURRENTLY idx_name2 ON t_test (name);
CREATE INDEX
```

Заметим, что PostgreSQL не гарантирует успеха при использовании команды `CREATE INDEX CONCURRENTLY`. Если выполнявшиеся в системе операции как-то конфликтуют с построением индекса, то может оказаться, что индекс будет помечен как недействительный. Чтобы узнать, действительны ли индексы, воспользуйтесь командой `\d` для интересующей таблицы.

ВВЕДЕНИЕ В КЛАССЫ ОПЕРАТОРОВ

До сих пор перед нами стояла задача понять, что индексировать, а затем позволить системе использовать этот индекс при поиске по столбцу или группе столбцов. При этом мы молчаливо предполагали, что выполнено одно условие – порядок сортировки данных фиксирован и неизменен. На практике это не всегда так. Конечно, порядок сортировки чисел всегда один и тот же, но для других типов данных предопределенного фиксированного порядка может и не быть.

Чтобы убедить вас в своей правоте, я решил взять реальный пример. Взгляните на эти две записи:

```
1118 09 08 78
2345 01 05 77
```

Вопрос: расположены ли эти строки в правильном порядке? Возможно – ведь в лексикографическом порядке первая предшествует второй. И все же это не так, потому что у этих строк имеется скрытая семантика. Здесь показаны два австрийских номера социального страхования. 09 08 78 означает 9 августа 1978 года, а 01 05 77 – 1 мая 1977 года. Первые четыре цифры – это контрольная сумма и некое автоматически инкрементируемое трехзначное число. Поскольку 1977 меньше 1978, для получения правильного порядка строки следует переставить.

Проблема в том, что PostgreSQL понятия не имеет, что эти строки означают. Если столбец имеет тип `text`, то PostgreSQL будет применять к нему стандартные правила сортировки текста. Если же столбец числовой, то PostgreSQL будет применять правила сортировки чисел. Ни при каких условиях она не станет делать нечто подобное тому, что я описал выше. Если вы думаете, что это единственное, о чем нужно думать при сортировке вышеуказанных чисел, то вы неправы. Сколько месяцев в году? 12? А вот и нет. В австрийской системе

социального страхования может быть 14 месяцев. Почему? Потому что если у иммигранта или беженца нет документов и его день рождения неизвестен, то ему назначат искусственный день рождения в 13-м месяце.

А теперь вспомните, что три цифры – это просто автоинкрементное трехзначное число. Во время балканских войн в 1990 году Австрия предоставила убежище более чем 115 000 беженцам. Естественно, трех цифр не хватило, и тогда был добавлен 14-й месяц. И какой же стандартный тип данных пригоден для этого рудимента COBOL, оставшегося с начала 1970-х годов (именно тогда проектировался формат номера социального страхования)? Да никакой.

Чтобы разумно обрабатывать такие специальные поля, в PostgreSQL имеются классы операторов:

```
test=# \h CREATE OPERATOR CLASS
```

Команда: CREATE OPERATOR CLASS

Описание: создать класс операторов

Синтаксис:

```
CREATE OPERATOR CLASS имя [ DEFAULT ] FOR TYPE тип_данных
    USING метод_индекса [ FAMILY имя_семейства ] AS
    { OPERATOR номер_стратегии имя_оператора [ ( op_type, op_type ) ]
      [ FOR SEARCH | FOR ORDER BY семейство_сортировки ]
      | FUNCTION номер_опорной_процедуры [ ( тип_операции [ , тип_операции ] ) ]
        имя_функции ( тип_аргумента [ , ... ] )
      | STORAGE тип_хранения
    } [, ... ]
```

Класс операторов говорит, как должен вести себя индекс. Рассмотрим стандартное В-дерево. Для него имеется пять операций:

Стратегия	Оператор	Описание
1	<	Меньше
2	<=	Меньше или равно
3	=	Равно
4	>=	Больше или равно
5	>	Больше

Стандартные классы операторов поддерживают стандартные типы данных и стандартные операторы, которые встречались нам в этой книге. Если мы хотим обрабатывать номера социального страхования, то необходимо придумать собственные операторы, реализующие нужную нам логику. Эти специальные операторы можно будет затем использовать для образования класса операторов, представляющего собой не что иное, как стратегию, передаваемую индексу для задания его поведения.

Создание класса операторов для В-дерева

Чтобы привести практический пример класса операторов, я написал код для обработки номеров социального страхования. Для простоты я не стал обращать внимания на такие детали, как контрольная сумма.

Создание новых операторов

Первым делом следует определить требуемые операторы. Заметим, что всего нужно пять операторов, по одному для каждой стратегии. Стратегия индекса – это, по существу, подключаемый модуль, вместо которого мы вставляем собственный код.

Для начала я подготовил тестовые данные:

```
CREATE TABLE t_sva (sva text);
INSERT INTO t_sva VALUES ('1118090878');
INSERT INTO t_sva VALUES ('2345010477');
```

Теперь можно приступить к созданию операторов. Для этого PostgreSQL предлагает команду CREATE OPERATOR:

```
test=# \h CREATE OPERATOR
Команда: CREATE OPERATOR
Описание: создать оператор
Синтаксис:
CREATE OPERATOR name (
    PROCEDURE = имя_функции
    [, LEFTARG = тип_слева ] [, RIGHTARG = тип_справа ]
    [, COMMUTATOR = коммут_оператор ] [, NEGATOR = обратный_оператор ]
    [, RESTRICT = процедура_ограничения ] [, JOIN = процедура_соединения ]
    [, HASHES ] [, MERGES ]
)
```

Общая идея такова: оператор вызывает функцию, которая получает один или два параметра – для левого и для правого операндов.

Как видим, оператор – это не что иное, как вызов функции. Поэтому мы должны реализовать логику функций, стоящих за операторами. Чтобы исправить порядок сортировки, я написал опорную функцию `normalize_si`:

```
CREATE OR REPLACE FUNCTION normalize_si(text) RETURNS text AS $$
BEGIN
    RETURN substring($1, 9, 2) ||
           substring($1, 7, 2) ||
           substring($1, 5, 2) ||
           substring($1, 1, 4);
END; $$
LANGUAGE 'plpgsql' IMMUTABLE;
```

Вызовем эту функцию:

```
test=# SELECT normalize_si('1118090878');
normalize_si
-----
7808091118
(1 строка)
```

Мы просто переставили местами некоторые цифры. После этого можно применить обычную сортировку строк. На следующем шаге эту функцию уже можно использовать для сравнения номеров социального страхования.

Первой стратегии необходима функция сравнения на меньше:

```
CREATE OR REPLACE FUNCTION si_lt(text, text) RETURNS boolean AS $$
BEGIN
    RETURN normalize_si($1) < normalize_si($2);
END;
$$ LANGUAGE 'plpgsql' IMMUTABLE;
```

Следует отметить два момента.

- Функция должна быть написана не на SQL, а на процедурном или компилируемом языке. Дело в том, что SQL-функции при некоторых условиях встраиваются, а это свело бы на нет все наши усилия (На SQL тоже все работает. – *Прим. ред.*).
- Необходимо придерживаться используемого в этой главе соглашения об именовании, поскольку оно принято сообществом. Функция сравнения на меньше должна называться `si_lt`, сравнения на меньше или равно – `si_le` и т. д.

Теперь мы можем определить остальные функции, которые понадобятся будущим операторам:

```
-- меньше или равно
CREATE OR REPLACE FUNCTION si_le(text, text)
    RETURNS boolean AS
$$
BEGIN
    RETURN normalize_si($1) <= normalize_si($2);
END;
$$
LANGUAGE 'plpgsql' IMMUTABLE;

-- больше или равно
CREATE OR REPLACE FUNCTION si_ge(text, text)
    RETURNS boolean AS
$$
BEGIN
    RETURN normalize_si($1) >= normalize_si($2);
END;
$$
LANGUAGE 'plpgsql' IMMUTABLE;

-- больше
CREATE OR REPLACE FUNCTION si_gt(text, text)
    RETURNS boolean AS
$$
BEGIN
    RETURN normalize_si($1) > normalize_si($2);
END;
$$
LANGUAGE 'plpgsql' IMMUTABLE;
```

Мы определили четыре функции. Пятая – для оператора сравнения на равенство – не нужна. Мы можем воспользоваться существующим оператором, т. к. равенство от порядка сортировки не зависит.

Имея все необходимые функции, мы можем определить самих операторов:

```
-- определить операторов
CREATE OPERATOR <# ( PROCEDURE=si_lt,
                    LEFTARG=text,
                    RIGHTARG=text);
```

Структура оператора очень проста. У оператора должно быть имя (в моем случае <#), вызываемая процедура, а также типы данных левого и правого операндов. При вызове оператора левый операнд станет первым аргументом si_lt, а правый – вторым.

Остальные операторы устроены точно так же:

```
CREATE OPERATOR <=# ( PROCEDURE=si_le,
                     LEFTARG=text,
                     RIGHTARG=text);

CREATE OPERATOR >=# ( PROCEDURE=si_ge,
                     LEFTARG=text,
                     RIGHTARG=text);

CREATE OPERATOR ># ( PROCEDURE=si_gt,
                     LEFTARG=text,
                     RIGHTARG=text);
```

В зависимости от типа индекса могут понадобиться дополнительные опорные функции. Для стандартных B-деревьев нужна всего одна функция, которая служит для ускорения работы:

```
CREATE OR REPLACE FUNCTION si_same(text, text) RETURNS int AS $$
BEGIN
    IF normalize_si($1) < normalize_si($2)
    THEN
        RETURN -1;
    ELIF normalize_si($1) > normalize_si($2)
    THEN
        RETURN +1;
    ELSE
        RETURN 0;
    END IF;
END;
$$ LANGUAGE 'plpgsql' IMMUTABLE;
```

Функция si_same возвращает -1, если первый аргумент меньше второго, 0 – если аргументы равны, и 1 – если первый аргумент больше. На внутреннем уровне она играет роль рабочей лошади, поэтому ее код должен быть как можно лучше оптимизирован.

Создание классов операторов

Ну вот, все компоненты на месте, и мы можем создать необходимый индексу класс операторов:


```
CREATE OPERATOR CLASS sva_special_ops
FOR TYPE text USING btree
AS
    OPERATOR 1 <# ,
    OPERATOR 2 <=# ,
    OPERATOR 3 = ,
    OPERATOR 4 >=# ,
    OPERATOR 5 ># ,

    FUNCTION 1 si_same(text, text);
```

Команда `CREATE OPERATOR CLASS` соединяет стратегии с операторами. Фраза `OPERATOR 1 <#` означает, что стратегия 1 будет использовать оператор `<#`. И напоследок с классом операторов соединяется функция `si_same`.

Отметим, что у класса операторов есть имя и что в определении явно указано, что он предназначен для работы с B-деревьями. Теперь мы уже можем использовать созданный нами класс операторов при создании индекса:

```
CREATE INDEX idx_special ON t_sva (sva sva_special_ops);
```

Команда создания индекса выглядит немного иначе, чем раньше: `sva sva_special_ops` означает, что столбец `sva` индексируется с применением класса операторов `sva_special_ops`. Если опустить явное указание класса `sva_special_ops`, то PostgreSQL будет использовать класс операторов по умолчанию.

Тестирование пользовательских классов операторов

В нашем примере тестовые данные состоят всего из двух строк. Поэтому PostgreSQL по своей воле не стала бы использовать индекс, т. к. таблица слишком мала, чтобы оправдать накладные расходы на открытие индекса. Чтобы все-таки протестировать класс операторов, не загружая слишком много данных, мы можем запретить оптимизатору последовательный просмотр. Для этого выполним в текущем сеансе команду:

```
SET enable_seqscan TO off;
```

Индекс работает, как и ожидалось:

```
test=# explain SELECT * FROM t_sva WHERE sva = '0000112273';
               QUERY PLAN
```

```
-----
Index Only Scan using idx_special on t_sva (cost=0.13..8.14 rows=1 width=32)
  Index Cond: (sva = '0000112273'::text)
(2 строки)
```

```
test=# SELECT * FROM t_sva;
      sva
```

```
-----
2345010477
1118090878
(2 строки)
```

Типы индексов в PostgreSQL

До сих пор мы обсуждали только B-деревья. Но во многих случаях их недостаточно. Почему так? Как было сказано в этой главе, B-деревья основаны на сортировке и, стало быть, на операторах $<$, $<=$, $=$, $>=$ и $>$. Проблема, однако, в том, что не для любого типа данных можно разумно определить сортировку. Взять, к примеру, многоугольник. Как можно было бы отсортировать объекты такого типа? Можно, конечно, сортировать по площади, по периметру и т. д., но это не поможет найти объект с помощью геометрического поиска.

Решение заключается в том, чтобы иметь разные типы индексов. Каждый индекс служит конкретной цели и делает ровно то, что от него требуется. В версии PostgreSQL 11.0 имеются следующие типы индексов:

```
test=# SELECT * FROM pg_am;
 amname | amhandler | amtype 
-----+-----+-----
 btree  | bthandler | i
 hash   | hashhandler | i
 gist   | gisthandler | i
 gin    | ginhandler | i
 spgist | spghandler | i
 brin   | brinhandler | i
(6 строк)
```

Существует шесть типов индексов. B-деревья мы уже подробно обсудили, но для чего нужны остальные? В следующих разделах мы кратко опишем все типы индексов, поддерживаемые PostgreSQL.

Отметим, что существуют еще и расширения: типы индексов *rum*, *vodka* и в будущем *cognac*.

Хеш-индексы

Хеш-индексы существуют уже много лет. Идея в том, чтобы вычислить хеш входного значения и сохранить его для последующего поиска. Наличие хеш-индексов оправдано, но в PostgreSQL 10.0 было рекомендовано не пользоваться ими, потому что механизм WAL их не поддерживал. В PostgreSQL 11.0 ситуация изменилась – теперь все операции с хеш-индексами протоколируются, поэтому они пригодны для репликации и, стало быть, на 100% отказоустойчивы.

В общем случае хеш-индексы занимают немного больше места, чем B-деревья. Если индексируется 4 млн целых значений, то B-дерево займет примерно 90 МБ, а хеш-индекс – 125 МБ. Поэтому популярное предположение о том, что хеш якобы занимает очень мало места на диске, как правило, неверно.

GiST-индексы

Обобщенные деревья поиска (Generalized Search Tree – **GiST**) – чрезвычайно важный тип индекса, поскольку они используются для самых разных вещей. GiST-индексы можно использовать для реализации R-дерева, они могут даже

выступать в роли В-дерева. Но применять GiST-индексы в качестве В-деревьев не рекомендуется – это было бы профанацией идеи.

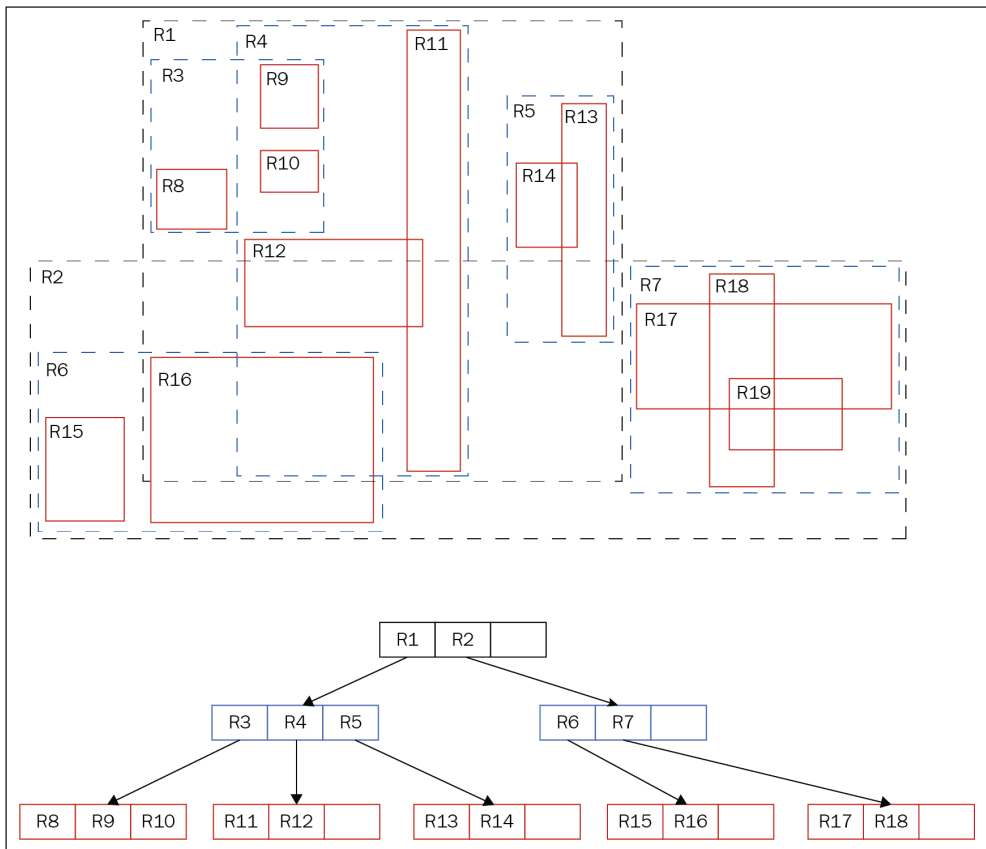
Перечислим типичные сценарии применения GiST-индексов:

- диапазонные типы;
- геометрические индексы (например, в очень популярном расширении PostGIS);
- нечеткий поиск.

Как работает GiST

Для многих GiST-индексы все еще представляются черным ящиком. Поэтому я решил включить в эту главу раздел о внутреннем механизме работы GiST.

Рассмотрим следующий рисунок.



Взгляните на это дерево. Мы видим, что **R1** и **R2** находятся наверху. **R1** и **R2** – ограничивающие прямоугольники, содержащие внутри себя все остальное. **R3**, **R4** и **R5** находятся внутри **R1**; **R8**, **R9** и **R10** – внутри **R3** и т. д. Таким образом,

GiST-индекс организован иерархически. На рисунке можно видеть некоторые операции, не поддерживаемые B-деревьями: пересекается, находится слева от, находится справа от и т. д. Структура GiST-дерева идеальна для геометрического индексирования.

Расширение GiST-индексов

Разумеется, мы можем определить свои собственные классы операторов. Поддерживаются следующие стратегии:

Операция	Номер стратегии
Строго слева от	1
Не простирается правее	2
Пересекается с	3
Не простирается левее	4
Строго справа от	5
Одинаковы	6
Содержит	7
Содержится в	8
Не простирается выше	9
Строго ниже	10
Строго выше	11
Не простирается ниже	12

Если вы захотите написать классы операторов для GiST-индекса, то придется предоставить несколько опорных функций. В случае B-дерева нужна была только функция `same`, а в GiST-индексах их намного больше:

Функция	Описание	Номер опорной функции
<code>consistent</code>	Определяет, удовлетворяет ли ключ условию запроса. На внутреннем уровне просматриваются стратегии	1
<code>union</code>	Вычисляет объединение набора ключей. В случае числовых значений просто вычисляет минимальное и максимальное значения или диапазон. Особенно важна для геометрических приложений	2
<code>compress</code>	Вычисляет сжатое представление ключа или значения	3
<code>decompress</code>	Противоположность <code>compress</code>	4
<code>penalty</code>	В процессе вставки вычисляется стоимость вставки в дерево. Эта стоимость определяет, в какое место дерева попадет новая запись. Поэтому хорошая функция <code>penalty</code> – ключ к высокой производительности индекса	5
<code>picksplit</code>	Определяет, какие записи страницы должны быть перемещены в новую страницу в случае расщепления. Некоторые записи останутся в старой странице, другие перейдут в новую. Хорошая функция <code>picksplit</code> также важна для высокой производительности индекса	6
<code>equal</code>	Похожа на функцию <code>same</code> для B-деревьев	7
<code>distance</code>	Вычисляет расстояние (число) между ключом и искомым значением. Функция <code>distance</code> необязательна, но нужна для поддержки поиска ближайших соседей	8
<code>fetch</code>	Вычисляет исходное представление сжатого ключа. Нужна для поддержки просмотра только индекса, реализованного в последней версии PostgreSQL	9

Классы операторов для GiST-индексов обычно пишутся на языке C. Если вы ищете хороший пример, рекомендую модуль `btree_gist` в каталоге `contrib`. Там показано, как индексировать стандартные типы данных с помощью GiST, код может служить источником собственных идей.

GIN-индексы

Обобщенные инвертированные индексы (*generalized inverted* – **GIN**) дают удобный способ индексирования текста. Допустим, мы хотим проиндексировать миллион текстовых документов. Некоторое слово может встречаться миллионы раз. В обычном B-дереве это означало бы, что ключ нужно хранить миллионы раз. В GIN-индексе дело обстоит иначе. Каждый ключ (слово) хранится только один раз, и с ним связывается список документов. Ключи организованы в виде стандартного B-дерева. В каждой записи хранится список документов, содержащий указатели на все записи таблицы с тем же ключом. GIN-индекс очень компактный. Но ему недостает важного свойства, которым обладают данные, отсортированные по B-дереву. В GIN-индексе список указателей на документы, ассоциированные с определенным ключом, отсортирован по позиции строки в таблице, а не по какому-то произвольному критерию.

Расширение GIN-индексов

GIN-индексы, как и любые другие, можно расширять. Определены следующие стратегии:

Операция	Номер стратегии
Пересекается с	1
Содержит	2
Содержится в	3
Равно	4

Определены также следующие опорные функции:

Функция	Описание	Номер опорной функции
<code>compare</code>	Аналогична одноименной функции для B-дерева. Возвращает -1, если первый аргумент меньше второго, 0 – если аргументы равны, и 1 – если первый аргумент больше второго	1
<code>extractValue</code>	Извлекает ключи из индексированного значения. Значение может содержать много ключей. Например, текст может содержать более одного слова	2
<code>extractQuery</code>	Извлекает ключи из условия запроса	3
<code>consistent</code>	Проверяет, соответствует ли значение условию запроса	4
<code>comparePartial</code>	Сравнивает частичный ключ из запроса с ключом из индекса. Возвращает -1, 0 или 1 (как одноименная функция для B-деревьев)	5
<code>triConsistent</code>	Определяет, соответствует ли значение условию запроса (троичный вариант). Необязательна, если присутствует опорная функция <code>consistent</code>	6

Тем, кто ищет хороший пример расширения GIN-индекса, рекомендую модуль `btree_gin` в каталоге `contrib` дистрибутива PostgreSQL. Это ценный источник информации и неплохая отправная точка для собственной реализации. Если вас интересует полнотекстовый поиск, то далее в данной главе имеется дополнительная информация на эту тему.

SP-GiST-индексы

GiST-деревья разбиения пространства (space partitioned GiST – SP-GiST) проектировались преимущественно для использования в памяти. Дело в том, что при хранении SP-GiST на диске требуется довольно много обращений к диску, а это куда дороже парочки переходов по указателям в памяти.

Красота SP-GiST-деревьев состоит в том, что их можно использовать для реализации различных типов деревьев, включая квадродеревья, k-d-деревья и префиксные деревья (trie-деревья).

Определены следующие стратегии:

Операция	Номер стратегии
Строго слева от	1
Строго справа от	5
Одинаковы	6
Содержится в	8
Строго ниже	10
Строго выше	11

Для написания собственных классов операторов для SP-GiST-индексов необходимо несколько опорных функций:

Функция	Описание	Номер опорной функции
<code>config</code>	Предоставляет основную информацию о классе операторов	1
<code>choose</code>	Определяет, как вставить новое значение во внутренний кортеж	2
<code>picksplit</code>	Определяет, как разделить множество значений	3
<code>inner_consistent</code>	Определяет, в каких внутренних ветвях нужно искать заданное значение	4
<code>leaf_consistent</code>	Определяет, удовлетворяет ли ключ условию запроса	5

BRIN-индексы

Индексы блочных диапазонов (block range index – BRIN) представляют большую практическую ценность. Все индексы, рассмотренные до сих пор, занимают очень много места на диске. И хотя на уменьшение размера GIN-индексов и им подобных было потрачено много усилий, проблема осталась, поскольку для каждой записи необходим указатель в индексе. Так что если в таблице 10 млн записей, то понадобится 10 млн указателей. Экономия места на диске –

основная проблема, решаемая BRIN-индексами. В BRIN-индексе не хранятся элементы, соответствующие каждому кортежу, хранятся лишь минимальное и максимальное значения в 128 (по умолчанию) блоках данных (общим размером 1 МБ). Поэтому индекс очень маленький, но при этом содержит неполную информацию. Просмотр индекса возвращает больше данных, чем запрашивалось. На следующем шаге PostgreSQL должна отфильтровать лишние строки.

В примере ниже демонстрируется, насколько в действительности мал BRIN-индекс:

```
test=# CREATE INDEX idx_brin ON t_test USING brin(id);
CREATE INDEX
test=# \di+ idx_brin
```

Список отношений					
Схема	Имя	Тип	Владелец	Таблица	Размер
public	idx_brin	index	hs	t_test	48 KB

(1 строка)

В этом примере BRIN-индекс в 2000 раз меньше стандартного B-дерева. Естественно возникает вопрос: почему же мы не используем BRIN-индексы всегда? Чтобы ответить на него, важно принять во внимание структуру BRIN-индекса, в котором хранятся минимальные и максимальные значения в порции размером 1 МБ. Если данные отсортированы (корреляция высока), то BRIN-индекс довольно эффективен, поскольку мы можем выбрать 1 МБ данных, просмотреть его и получить искомым результат. А если данные перемешаны? В таком случае BRIN-индекс не сможет исключить блоки данных, потому что с большой вероятностью внутри порции размером 1 МБ находятся значения, близкие общему минимуму или максимуму. Таким образом, BRIN подходит в основном для сильно коррелированных данных, которые чаще всего встречаются в приложениях для организации хранилищ данных. Зачастую данные загружаются каждый день, и потому даты сильно коррелированы.

Расширение BRIN-индексов

BRIN-индексы поддерживают те же стратегии, что B-дерева, и, стало быть, нуждаются в том же наборе операторов¹. Так что код можно использовать повторно.

Операция	Стратегия
Меньше	1
Меньше или равно	2
Равно	3
Больше или равно	4
Больше	5

¹ Это верно, если данные допускают сортировку. Но BRIN-индекс, аналогично GiST, можно использовать и в более общем случае; тогда используется другой набор операторов. – Прим. ред.

Для BRIN нужны следующие опорные функции:

Функция	Описание	Номер опорной функции
opcInfo	Возвращает внутреннюю информацию об индексированных столбцах	1
add_value	Добавляет новую запись в существующий сводный кортеж	2
consistent	Определяет, удовлетворяет ли значение условию запроса	3
union	Вычисляет объединение двух сводных кортежей (минимальных и максимальных значений)	4

Добавление новых типов индексов

Начиная с версии PostgreSQL 9.6 появился простой способ разворачивать совершенно новые типы индексов в виде расширений. Это замечательно, потому что если встроенных в PostgreSQL типов индексов не хватает, то можно добавить свои, точно отвечающие решаемой задаче. Для этого предназначена команда `CREATE ACCESS METHOD`:

```
test=# \h CREATE ACCESS METHOD
Команда: CREATE ACCESS METHOD
Описание: создать новый метод доступа
Синтаксис:
CREATE ACCESS METHOD имя
    TYPE тип_метода_доступа
    HANDLER функция_обработки
```

Особо не переживайте по поводу этой команды – на случай, если вы когда-нибудь захотите сами развернуть тип индекса, он будет оформлен в виде готового к работе расширения.

Одно из таких расширений реализует фильтр Блума. Фильтр Блума – это вероятностная структура данных. Иногда он возвращает слишком много строк, но никогда не возвращает слишком мало. Поэтому фильтр Блума – хороший метод предварительной фильтрации данных.

Как он работает? Фильтр Блума определен для нескольких столбцов. На основе входных значений вычисляется битовая маска, которая затем сравнивается с запросом. Плюсом фильтра Блума является тот факт, что можно проиндексировать сколько угодно столбцов, а минусом – что необходимо загрузить в память фильтр целиком. Но поскольку фильтр Блума меньше исходных данных, то во многих случаях это весьма выгодная сделка.

Для использования фильтров Блума достаточно активировать расширение, входящее в состав пакета `contrib`:

```
test=# CREATE EXTENSION bloom;
CREATE EXTENSION
```

Как уже сказано, идея фильтра Блума заключается в том, что он позволяет индексировать столько столбцов, сколько нужно. Во многих реальных прило-

жениях возникает проблема – индексировать много столбцов, не зная точно, какие комбинации понадобятся пользователям. Если таблица велика, то совершенно невозможно построить стандартное B-дерево, скажем, по 80 полям или того больше. В этом случае фильтр Блума может составить альтернативу:

```
test=# CREATE TABLE t_bloom (x1 int, x2 int, x3 int, x4 int, x5 int, x6 int, x7 int);
CREATE TABLE
```

Построить индекс просто:

```
test=# CREATE INDEX idx_bloom ON t_bloom USING bloom(x1, x2, x3, x4, x5, x6, x7);
CREATE INDEX
```

Запретив последовательный просмотр, мы сможем увидеть индекс в действии:

```
test=# SET enable_seqscan TO off;
SET
test=# explain SELECT * FROM t_bloom WHERE x5 = 9 AND x3 = 7;
               QUERY PLAN
-----
Bitmap Heap Scan on t_bloom (cost=18.50..22.52 rows=1 width=28)
  Recheck Cond: ((x3 = 7) AND (x5 = 9))
    -> Bitmap Index Scan on idx_bloom (cost=0.00..18.50 rows=1 width=0)
      Index Cond: ((x3 = 7) AND (x5 = 9))
```

Обратите внимание, что в запросе участвует комбинация случайно выбранных столбцов, они никак не соотносятся с фактическим порядком хранения ключей в индексе. Фильтр Блума все равно дает выигрыш.

Если вы заинтересовались фильтром Блума, почитайте статью https://en.wikipedia.org/wiki/Bloom_filter.

ПОЛУЧЕНИЕ БОЛЕЕ ТОЧНЫХ ОТВЕТОВ С ПОМОЩЬЮ НЕЧЕТКОГО ПОИСКА

В наши дни пользователи не довольствуются точным поиском. Современные сайты приучили пользователей к тому, что на ответ можно рассчитывать всегда, как бы ни был введен запрос. Google даст ответ, даже если запрос пользователя неправилен, изобилует опечатками или попросту не имеет смысла.

Расширение pg_trgm и его достоинства

Для выполнения нечеткого поиска в PostgreSQL можно добавить расширение pg_trgm. Чтобы его активировать, выполните команду:

```
test=# CREATE EXTENSION pg_trgm;
CREATE EXTENSION
```

Расширение pg_trgm обладает целым рядом возможностей, и, чтобы их продемонстрировать, я подготовил тестовые данные, содержащие названия

2354 австрийских городов и деревень. Эти данные будут храниться в простой таблице:

```
test=# CREATE TABLE t_location (name text);
CREATE TABLE
```

Все данные находятся на сайте моей компании, и PostgreSQL позволяет загрузить непосредственно оттуда:

```
test=# COPY t_location FROM PROGRAM
        'curl https://www.cybertec-postgresql.com/secret/orte.txt';
COPY 2354
```



Команду `curl` (командную утилиту для загрузки данных из интернета) необходимо установить. Если у вас ее нет, скачайте файл и поместите его в локальную файловую систему.

Загрузив данные, проверим содержимое таблицы:

```
test=# SELECT * FROM t_location LIMIT 4;
        name
```

```
-----
Eisenstadt
Rust
Breitenbrunn am Neusiedler See
Donnerskirchen
(4 строки)
```

Если вы не говорите по-немецки, то произнести некоторые названия без ошибок вряд ли удастся.

`pg_trgm` предлагает оператор `<->`, который вычисляет расстояние между двумя строками:

```
test=# SELECT 'abcde' <-> 'abdeacb';
?column?
-----
0.833333
(1 строка)
```

Расстояние – это число между 0 и 1. Чем оно меньше, тем более похожи строки.

Как это работает? Функция `show_trgm` принимает строку и разбивает ее на триграммы – последовательности по три символа:

```
test=# SELECT show_trgm('abcdef');
        show_trgm
-----
{" a", " ab", abc,bcd,cde,def,"ef "}
```

Вот на основе этих последовательностей и вычисляется расстояние. Конечно, оператор расстояния можно использовать внутри запроса для поиска ближайшего соответствия:

```
test=# SELECT *
FROM t_location
ORDER BY name <-> 'Kramertneusiedel'
LIMIT 3;
      name
-----
Gramatneusiedl
Klein-Neusiedl
Pötzneusiedl
(3 строки)
```

Строки Gramatneusiedl и Kramertneusiedel довольно близки. Они произносятся похоже, а написание К вместо G – довольно распространенная ошибка. В Google вы наверняка видели сообщение «Возможно, вы имели в виду». Очень может статься, что реализация этого механизма основана на триграммах.

В PostgreSQL для индексирования текста по триграммам можно воспользоваться GiST-индексом:

```
test=# CREATE INDEX idx_trgm ON t_location
      USING GiST(name GiST_trgm_ops);
CREATE INDEX
```

Расширение pg_trgm включает класс операторов gist_trgm_ops, предназначенный специально для поиска по сходству. Из распечатки ниже видно, что индекс используется, как положено:

```
test=# EXPLAIN SELECT *
FROM t_location
ORDER BY name <-> 'Kramertneusiedel'
LIMIT 5;
                                QUERY PLAN
-----
Limit (cost=0.14..0.58 rows=5 width=17)
  -> Index Scan using idx_trgm on t_location (cost=0.14..207.22 rows=2354 width=17)
      Order By: (name <-> 'Kramertneusiedel'::text)
(3 строки)
```

Ускорение запросов с предикатом LIKE

Запросы с предикатом LIKE, безусловно, являются причиной худших проблем с производительностью, которые только можно вообразить. В большинстве СУБД LIKE работает медленно и требует последовательного просмотра. Мало того, пользователи очень быстро обнаруживают, что нечеткий поиск во многих случаях дает лучшие результаты, чем точные запросы. Поэтому всего один запрос с предикатом LIKE к большой таблице, выполняемый достаточно часто, может поставить крест на производительности всего сервера.

По счастью, PostgreSQL предлагает решение проблемы, и оно даже уже установлено:

```
test=# EXPLAIN SELECT * FROM t_location WHERE name LIKE '%neusi%';
                                QUERY PLAN
-----
```

```
Bitmap Heap Scan on t_location (cost=4.33..19.05 rows=24 width=13)
  Recheck Cond: (name ~~ '%neusi% '::text)
  -> Bitmap Index Scan on idx_trgm (cost=0.00..4.32 rows=24 width=0)
    Index Cond: (name ~~ '%neusi% '::text)
(4 строки)
```

Триграммный индекс, построенный в предыдущем разделе, подходит и для ускорения запросов с LIKE. Обратите внимание, что символы % можно использовать в любом месте поисковой строки. Это значительный шаг вперед, по сравнению с В-деревьями, которые позволяют эффективно обрабатывать запросы с LIKE, только если метасимвол находится в конце поисковой строки.

Регулярные выражения

Но и это еще не все. Триграммный индекс способен даже ускорить поиск по простым регулярным выражениям. Ниже приведен пример:

```
test=# SELECT * FROM t_location WHERE name ~ '[A-C].*neu.*';
      name
-----
Bruckneudorf
(1 строка)

test=# explain SELECT * FROM t_location WHERE name ~ '[A-C].*neu.*';
              QUERY PLAN
-----
Index Scan using idx_trgm on t_location (cost=0.14..8.16 rows=1 width=13)
  Index Cond: (name ~ '[A-C].*neu.* '::text)
(2 строки)
```

PostgreSQL анализирует регулярное выражение и использует индекс для ответа на запрос.



PostgreSQL преобразует регулярное выражение в граф, а затем организует соответствующий графу просмотр индекса.

Полнотекстовый поиск

При поиске имен или простых строк обычно спрашивается все содержимое поля. Полнотекстовый поиск (ПТП) организован иначе. Его цель – найти в тексте слова или группы слов. Поэтому ПТП ближе к операции «содержит», поскольку мы почти никогда не ищем точную строку.

В PostgreSQL для полнотекстового поиска можно использовать GIN-индексы. Идея заключается в том, чтобы выделить из текста ценные *лексемы*, т. е. предварительно обработанные канонические формы слов, и индексировать эти элементы, а не исходный текст. Например:

```
test=# SELECT to_tsvector('english', 'A car, I want a car. I would not even
mind having many cars');
           to_tsvector
```

```
-----
'car':2,6,14 'even':10 'mani':13 'mind':11 'want':4 'would':8
(1 строка)
```

В этом примере показано простое предложение. Функция `to_tsvector` принимает строку, применяет к ней правила английского языка (`english`), отбрасывает стоп-слова и производит стемминг. Например, слова **car** и **cars** преобразуются в **car**. Заметим, что речь идет не просто о выделении основы слова. Слово **many** PostgreSQL преобразует в строку **mani**, применяя стандартные правила, ориентированные на английский язык.

Результат функции `to_tsvector` сильно зависит от языка. Если мы попросим PostgreSQL интерпретировать ту же строку на голландском языке, то результат будет совсем другим:

```
test=# SELECT to_tsvector('dutch', 'A car, I want a car. I would not even
mind having many cars');
               to_tsvector
```

```
-----
'a':1,5 'car':2,6,14 'even':10 'having':12 'i':3,7 'many':13
'mind':11 'not':9 'would':8
(1 строка)
```

Чтобы узнать, какие конфигурации поддерживаются, выполните такой запрос:

```
SELECT cfgname FROM pg_ts_config;
```

Сравнение строк

Немного познакомившись с процессом стемминга, посмотрим, как подвергнутый ему текст можно сравнить с запросом пользователя. В следующем фрагменте показано, как искать слово `wanted`:

```
test=# SELECT to_tsvector('english', 'A car, I want a car. I would not even
mind having many cars') @@ to_tsquery('english', 'wanted');
?column?
-----
t
(1 строка)
```

Отметим, что слово **wanted** не фигурирует в исходном тексте. И тем не менее PostgreSQL возвращает `true`. Причина в том, что слова `want` и `wanted` преобразуются в одну и ту же лексему. Это, безусловно, имеет практический смысл. Представьте, что вы ищете «автомобиль» через Google. Если вы найдете страницы сайтов, торгующих «автомобилями», то будете довольны. Таким образом, поиск общих лексем – здравая идея.

Иногда требуется найти не одно слово, а группу слов. Функция `to_tsquery` поддерживает и такую возможность:

```
test=# SELECT to_tsvector('english', 'A car, I want a car. I would not even
mind having many cars') @@ to_tsquery('english', 'wanted & bmw');
?column?
```

```
-----
f
(1 строка)
```

В этом случае возвращено `false`, потому что слово `bmw` отсутствует во входной строке. В функции `to_tsquery` символ `&` означает `and`, а символ `|` – `or`. Таким образом, легко построить сложную поисковую строку.

Определение GIN-индексов

Если вы хотите применить полнотекстовый поиск к столбцу или группе столбцов, то у вас есть две возможности:

- создать функциональный GIN-индекс;
- добавить столбец, содержащий уже готовые к использованию векторы `tsvector`, и триггер, гарантирующий их синхронизацию с другими столбцами.

В этом разделе рассматриваются оба подхода. Для демонстрации я подготовил тестовые данные:

```
test=# CREATE TABLE t_fts AS SELECT comment
      FROM pg_available_extensions;
SELECT 43
```

Разбор столбца прямо при построении функционального индекса, конечно, медленнее, зато требует меньше места на диске:

```
test=# CREATE INDEX idx_fts_func ON t_fts
      USING gin(to_tsvector('english', comment));
CREATE INDEX
```

Добавление физического столбца требует больше места, но индекс строится быстрее.

```
test=# ALTER TABLE t_fts ADD COLUMN ts tsvector;
ALTER TABLE
```

Правда, возникает вопрос о синхронизации этого столбца. Для этого создадим триггер:

```
test=# CREATE TRIGGER tsvectorupdate
      BEFORE INSERT OR UPDATE ON t_fts
      FOR EACH ROW
      EXECUTE PROCEDURE
      tsvector_update_trigger(somename, 'pg_catalog.english', 'comment');
```

По счастью, PostgreSQL уже предоставляет написанную на C функцию, которую можно вызывать из триггера для синхронизации столбца `tsvector`. Просто передайте ей нужный язык и имена столбцов, а она позаботится обо всем остальном. Отметим, что триггер всегда работает в контексте той транзакции, которая содержит команду обновления. Поэтому риска рассогласования нет.

Отладка поиска

Иногда не вполне понятно, почему поиск нашел совпадение с данной строкой. Для отладки запроса PostgreSQL предлагает функцию `ts_debug`. С точки зрения пользователя, она очень похожа на `to_tsvector`, но проливает свет на то, как устроена инфраструктура ПТП:

```
test=# \x
Расширенный вывод включен.
test=# SELECT * FROM ts_debug('english', 'go to www.postgresql-support.de');
= 91 =
-[ RECORD 1 ]+-----
alias      | asciiword
description | Word, all ASCII
token      | go
dictionaries | {english_stem}
dictionary | english_stem
lexemes    | {go}
-[ RECORD 2 ]+-----
alias      | blank
description | Space symbols
token      |
dictionaries | {}
dictionary |
lexemes    |
-[ RECORD 3 ]+-----
alias      | asciiword
description | Word, all ASCII
token      | to
dictionaries | {english_stem}
dictionary | english_stem
lexemes    | {}
-[ RECORD 4 ]+-----
alias      | blank
description | Space symbols
token      |
dictionaries | {}
dictionary |
lexemes    |
-[ RECORD 5 ]+-----
alias      | host
description | Host
token      | www.postgresql-support.de
dictionaries | {simple}
dictionary | simple
lexemes    | {www.postgresql-support.de}
```

`ts_debug` выводит список всех найденных лексем и сведения о каждой лексеме. Вы увидите, какую лексему нашел анализатор, какой словарь использовал, а также тип объекта. В данном примере обнаружены пробелы, слова и доменные имена. Можно также встретить числа, адреса электронной почты и многое

другое. Способ обработки строки зависит от ее типа. Например, подвергать стеммингу доменные имена и почтовые адреса, очевидно, бессмысленно.

Сбор статистики по словам

В процессе полнотекстового поиска может обрабатываться огромный объем данных. Чтобы дать пользователям возможность лучше разобраться в своих текстовых данных, PostgreSQL предлагает функцию `pg_stat`, которая возвращает список слов:

```
SELECT * FROM ts_stat('SELECT to_tsvector(''english'', comment)
                     FROM pg_available_extensions')
        ORDER BY 2 DESC
        LIMIT 3;
```

word	ndoc	nentry
function	10	10
data	10	10
type	7	7

(3 строки)

Столбец `word` содержит результат стемминга слова, `ndoc` – количество документов, в которых встречается слово, а `nentry` – сколько раз всего встретилось слово.

О пользе операторов исключения

До сих пор мы использовали индексы, чтобы ускорить работу и обеспечить уникальность. Но однажды кому-то пришла в голову мысль расширить область применения индексов. В этой главе мы видели, что GiST-индексы поддерживают операции «пересекается», «содержит» и многие другие. Так почему бы не воспользоваться этим для контроля над целостностью данных?

Приведем пример:

```
test=# CREATE EXTENSION btree_gist;
test=# CREATE TABLE t_reservation (
    room int,
    from_to tsrange,
    EXCLUDE USING GiST (room with =, from_to with &&)
);
CREATE TABLE
```

Во фразе `EXCLUDE USING GiST` определены дополнительные ограничения. Если вы сдаете в аренду квартиры, то можете разрешить бронирование разных квартир на один срок, но одна и та же квартира не должна быть забронирована дважды на один период. В приведенном выше примере фраза `EXCLUDE` означает: если квартира дважды забронирована на один и тот же период, то должна быть выдана ошибка (диапазоны `from_to`, относящиеся к одной квартире, не должны пересекаться).

Следующие две строки не нарушают это ограничение:

```
test=# INSERT INTO t_reservation
VALUES (10, '["2017-01-01", "2017-03-03"]');
INSERT 0 1
test=# INSERT INTO t_reservation
VALUES (13, '["2017-01-01", "2017-03-03"]');
INSERT 0 1
```

Однако такая команда INSERT вызовет нарушение, потому что диапазоны пересекаются:

```
test=# INSERT INTO t_reservation
VALUES (13, '["2017-02-02", "2017-08-14"]');
ОШИБКА: конфликтующее значение ключа нарушает ограничение исключения
"t_reservation_room_from_to_excl"
ПОДРОБНОСТИ: Key (room, from_to)=(13, ["2017-02-02 00:00:00", "2017-08-14
00:00:00"]) conflicts with existing key (room, from_to)=(13, ["2017-01-01
00:00:00", "2017-03-03 00:00:00"]).
```

Использование операторов исключения очень полезно, поскольку предоставляет мощные средства поддержания целостности данных.

РЕЗЮМЕ

Эта глава целиком была посвящена индексам. Мы узнали, когда PostgreSQL решает воспользоваться индексом и какие существуют типы индексов. Можно также реализовать собственные способы ускорения работы приложений, прибегнув к помощи пользовательских операторов и стратегий индексирования.

Тем, кто готов идти до конца, PostgreSQL предлагает также механизм пользовательских методов доступа.

Глава 4 посвящена передовым средствам SQL. Многие не знают обо всем, на что способен SQL, поэтому я собираюсь продемонстрировать ряд эффективных и эффектных приемов.

Вопросы

Всегда ли индексы улучшают производительность?

Безусловно, нет. Если бы какое-то средство всегда давало хороший результат, то оно было бы включено по умолчанию. Индексы могут ускорить многие операции, но могут и существенно замедлить работу. Правило одно: думайте о том, что делаете и чего хотите достичь.

Правда ли, что индекс занимает много места?

Зависит от типа индекса. BRIN-индексы совсем небольшие и сравнительно дешевые, но остальные индексы обычно потребляют гораздо больше места. К примеру, B-дерева примерно в 2000 раз объемнее BRIN-индексов. А индексы, основанные на триграммах, как правило, еще больше.

Как узнать, каких индексов не хватает?

На мой взгляд, самый лучший способ – проанализировать таблицы `pg_stat_statements` и `pg_stat_user_tables`. Особенно интересен столбец `seq_tup_read`. Если система читает очень много строк, значит, какого-то индекса, возможно, не хватает. Чтобы понять, что на самом деле происходит, необходимо пристально изучить запрос. `EXPLAIN` всегда придет вам на помощь.

Можно ли строить индексы параллельно?

Да, начиная с версии PostgreSQL включена поддержка параллельного построения индексов. Это может значительно ускорить создание индекса.

Глава 4

Передовые средства SQL

В главе 3 мы узнали об индексировании и об имеющейся в PostgreSQL возможности создавать пользовательские индексы для ускорения запросов. У большинства читателей этой книги есть некоторый опыт использования SQL. Но, как выясняется, передовые средства, описанные в этой книге, известны не слишком широко, поэтому имеет смысл рассмотреть их, дав людям возможность решать свои задачи быстрее и эффективнее. Давно уже спорят о том, является ли база данных просто хранилищем или в нее должны быть встроены средства бизнес-логики. Быть может, данная глава прольет свет на этот вопрос и покажет, чем в действительности является современная база данных.

Эта глава посвящена современному языку SQL и его возможностям. Подробно представлены разнообразные, достаточно изощренные средства SQL. Мы рассмотрим следующие темы:

- наборы группирования;
- упорядоченные наборы;
- гипотетические агрегаты;
- оконные функции и аналитические средства.

ВВЕДЕНИЕ В НАБОРЫ ГРУППИРОВАНИЯ

Любой квалифицированный пользователь SQL должен быть знаком с фразами GROUP BY и HAVING. Но знакомы ли вы также с CUBE, ROLLUP и GROUPING SETS? Если нет, то эту главу следует прочитать обязательно.

Загрузка тестовых данных

Чтобы читать эту главу было интереснее, я подготовил тестовые данные, взятые из отчета по производству энергоносителей, составленному компанией BP: <http://www.bp.com/en/global/corporate/energy-economics/statistical-review-of-world-energy/downloads.html>.

Ниже описана структура данных:

```
test=# CREATE TABLE t_oil (  
      region      text,  
      country     text,
```

```

year      int,
production int,
consumption int
);
CREATE TABLE

```

Сами данные можно скачать с нашего сайта с помощью `curl`:

```

test=# COPY t_oil FROM PROGRAM '
curl https://www.cybertec-postgresql.com/secret/oil_ext.txt ';
COPY 644

```

В некоторых операционных системах утилита `curl` отсутствует или не устанавливается по умолчанию, в таком случае ее необходимо предварительно скачать и установить.

В отчете приведены данные за период с 1965 по 2010 год для 14 стран в двух регионах мира:

```

test=# SELECT region, avg(production) FROM t_oil GROUP BY region;
   region   |      avg
-----+-----
Middle East | 1992.6036866359447005
North America | 4541.3623188405797101
(2 строки)

```

Применение наборов группирования

Фраза `GROUP BY` преобразует группу строк в одну. Однако в реальных отчетах может возникнуть также необходимость в среднем по всему набору. То есть нужна одна дополнительная строка.

Вот как это можно сделать:

```

test=# SELECT region, avg(production)
      FROM t_oil
      GROUP BY ROLLUP (region);
   region   |      avg
-----+-----
Middle East | 1992.6036866359447005
North America | 4541.3623188405797101
              | 2607.5139860139860140
(3 строки)

```

`ROLLUP` вставляет дополнительную строку, содержащую среднее по всему множеству. В отчете такая сводная строка, скорее всего, понадобится. И чтобы предоставить эти данные, PostgreSQL достаточно одного запроса, а не двух. Обратим внимание еще на один момент: в разных версиях PostgreSQL данные могут возвращаться в разном порядке. Причина в том, что в PostgreSQL 10.0 реализация наборов группирования была существенно улучшена. В версии 9.6 и ранее PostgreSQL должна была выполнять большой объем сортировки. Начиная с версии 10.0 стало возможно заменить сортировку хешированием, что во многих случаях ведет к значительному ускорению:

```
test=# EXPLAIN SELECT region, avg(production)
      FROM t_oil
      GROUP BY ROLLUP (region);
      QUERY PLAN
-----
MixedAggregate (cost=0.00..17.31 rows=3 width=44)
  Hash Key: region
  Group Key: ()
  -> Seq Scan on t_oil (cost=0.00..12.44 rows=644 width=16)
(4 строки)
```

Если мы хотим, чтобы данные были отсортированы и во всех версиях возвращались в одном и том же порядке, то следует включить в запрос фразу ORDER BY.

Конечно, эта операция применима и к группировке по нескольким столбцам:

```
test=# SELECT region, country, avg(production)
      FROM t_oil
      WHERE country IN ('USA', 'Canada', 'Iran', 'Oman')
      GROUP BY ROLLUP (region, country);
      region | country | avg
-----+-----+-----
Middle East | Iran    | 3631.6956521739130435
Middle East | Oman    | 586.4545454545454545
Middle East |         | 2142.9111111111111111
North America | Canada | 2123.2173913043478261
North America | USA     | 9141.3478260869565217
North America |         | 5632.2826086956521739
              |         | 3906.7692307692307692
(7 строк)
```

В этом примере PostgreSQL вставляет в результирующий набор три строки: одну для Ближнего Востока (Middle East), другую – для Северной Америки (North America) и третью – для среднего по всему набору. Для веб-приложения это идеально, потому что мы легко можем построить графический интерфейс для анализа результирующего набора, отфильтровав значения null.

Группировка ROLLUP удобна, если нужно отобразить результат немедленно. Лично я всегда использую ее для показа конечных результатов пользователям. Однако в отчете пользователи могут захотеть большей гибкости, и, стало быть, понадобится предварительно вычислить больше данных. Быть может, в этом вам поможет ключевое слово CUBE:

```
test=# SELECT region, country, avg(production)
      FROM t_oil
      WHERE country IN ('USA', 'Canada', 'Iran', 'Oman')
      GROUP BY CUBE (region, country);
      region | country | avg
-----+-----+-----
Middle East | Iran    | 3631.6956521739130435
Middle East | Oman    | 586.4545454545454545
```

Middle East		2142.9111111111111111
North America	Canada	2123.2173913043478261
North America	USA	9141.3478260869565217
North America		5632.2826086956521739
		3906.7692307692307692
	Canada	2123.2173913043478261
	Iran	3631.6956521739130435
	Oman	586.4545454545454545
	USA	9141.3478260869565217

(11 строк)

Заметьте, что в результирующий набор добавлено еще больше строк. CUBE генерирует те же данные, что GROUP BY region, country + GROUP BY region + GROUP BY country + среднее по всему набору. То есть идея в том, чтобы сразу произвести агрегирование на разных уровнях. Результирующий куб содержит все возможные комбинации групп.

ROLLUP и CUBE – просто удобные вспомогательные средства, построенные на основе фразы GROUPING SETS. Эта фраза позволяет явно перечислить интересующие нас агрегаты:

```
test=# SELECT region, country, avg(production)
        FROM t_oil
        WHERE country IN ('USA', 'Canada', 'Iran', 'Oman')
        GROUP BY GROUPING SETS ( (), region, country);
   region   | country |      avg
```

Middle East		2142.9111111111111111
North America		5632.2826086956521739
		3906.7692307692307692
	Canada	2123.2173913043478261
	Iran	3631.6956521739130435
	Oman	586.4545454545454545
	USA	9141.3478260869565217

(7 строк)

В этом примере я задал три набора группирования: среднее по всему набору, GROUP BY region и GROUP BY country. Если вы хотите сгруппировать еще комбинации регионов и стран, то следует добавить (region, country).

Изучение производительности

Наборы группирования – мощное средство, помогающее уменьшить количество дорогостоящих запросов. На внутреннем уровне PostgreSQL прибегает к традиционному узлу плана GroupAggregate. Этот узел нуждается в отсортированных данных, поэтому будьте готовы к тому, что PostgreSQL будет выполнять много промежуточных операций сортировки:

```
test=# EXPLAIN SELECT region, country, avg(production)
        FROM t_oil
        WHERE country IN ('USA', 'Canada', 'Iran', 'Oman')
        GROUP BY GROUPING SETS ( (), region, country);
```

QUERY PLAN

```
-----
GroupAggregate (cost=22.58..32.69 rows=34 width=52)
  Group Key: region
  Group Key: ()
  Sort Key: country
    Group Key: country
  -> Sort (cost=22.58..23.04 rows=184 width=24)
    Sort Key: region
    -> Seq Scan on t_oil (cost=0.00..15.66 rows=184 width=24)
      Filter: (country = ANY ('{USA,Canada,Iran,Oman}'::text[]))
(9 строк)
```

В PostgreSQL хеш-агрегаты поддерживаются только для обычных фраз GROUP BY без наборов группирования. В PostgreSQL 10.0 планировщик располагает большими возможностями, чем в PostgreSQL 9.6. Следует ожидать, что быстрое действие наборов группирования в новой версии повысится.

Сочетание наборов группирования с фразой FILTER

В реальных приложениях наборы группирования часто сочетаются с фразами FILTER, которые позволяют вычислять частичные агрегаты, например:

```
test=# SELECT region,
  avg(production) AS all,
  avg(production) FILTER (WHERE year < 1990) AS old,
  avg(production) FILTER (WHERE year >= 1990) AS new
FROM t_oil
GROUP BY ROLLUP (region);
 region |      all      |      old      |      new
-----+-----+-----+-----
Middle East | 1992.603686635 | 1747.325892857 | 2254.233333333
North America | 4541.362318840 | 4471.653333333 | 4624.349206349
           | 2607.513986013 | 2430.685618729 | 2801.183150183
(3 строки)
```

Идея в том, что в разных столбцах агрегируются разные данные. Фраза FILTER позволяет избирательно передавать данные агрегатным функциям. В этом случае во втором агрегате рассматриваются только до 1990 года, в третьем агрегате – недавние данные, а в первом – все вообще данные.



Если есть возможность перенести условия во фразу WHERE, то так и следует поступить, поскольку тогда из таблиц будет выбираться меньше данных. Фраза FILTER полезна, только если не все данные, отобранные WHERE, нужны в каждом агрегате.

FILTER работает для всех агрегатных функций и предлагает простой способ развернуть данные: поменять местами строки и столбцы. Кроме того, FILTER работает быстрее его имитации с помощью конструкции CASE WHEN ... THEN NULL ... ELSE END. Результаты сравнения на реальных данных можно найти по адресу <https://www.cybertec-postgresql.com/en/postgresql-9-4-aggregation-filters-they-do-pay-off/>.

ИСПОЛЬЗОВАНИЕ УПОРЯДОЧЕННЫХ НАБОРОВ

Упорядоченные наборы – очень полезная вещь, но они плохо известны сообществу разработчиков. Идея чрезвычайно проста: данные группируются как обычно, а затем данные внутри каждой группы упорядочиваются в соответствии с заданным условием. После этого с отсортированными данными производится некоторое вычисление.

Классический пример – вычисление медианы.

i Медиана – это значение, находящееся посередине. Если, к примеру, у вас медианный доход, значит, количество людей, зарабатывающих больше и меньше вас, одинаково.

Один из способов вычислить медиану – отсортировать данные и отобрать 50%. Именно это делает фраза `WITHIN GROUP`:

```
test=# SELECT region,
      percentile_disc(0.5) WITHIN GROUP (ORDER BY production)
FROM t_oil
GROUP BY 1;
```

region	percentile_disc
Middle East	1082
North America	3054

(2 строки)

Функция `percentile_disc` пропускает 50% группы и возвращает требуемое значение.

i Отметим, что медиана может существенно отличаться от среднего значения.

В экономике разность между медианным и средним доходами может использоваться как индикатор социального неравенства. Чем больше медиана по сравнению со средним, тем выше неравенство доходов. Для пущей гибкости стандарт ANSI предлагает не просто функцию вычисления медианы, но и задавание в функции `percentile_disc` любого параметра от 0 до 1.

Прелесть в том, что упорядоченные наборы можно даже использовать в сочетании с наборами группирования:

```
test=# SELECT region,
      percentile_disc(0.5) WITHIN GROUP (ORDER BY production)
FROM t_oil
GROUP BY ROLLUP (1);
```

region	percentile_disc
Middle East	1082
North America	3054
	1696

(3 строки)

В данном случае PostgreSQL снова вставляет дополнительные строки в результирующий набор.

В соответствии со стандартом ANSI SQL PostgreSQL предлагает две функции `percentile_`. Функция `percentile_disc` возвращает значение, которое действительно присутствует в наборе данных. А функция `percentile_cont` возвращает интерполированное значение, если точное отсутствует в наборе. Различие демонстрируется в следующем примере:

```
test=# SELECT percentile_disc(0.62) WITHIN GROUP (ORDER BY id),
           percentile_cont(0.62) WITHIN GROUP (ORDER BY id)
FROM generate_series(1, 5) AS id;
 percentile_disc | percentile_cont 
-----+-----
4                | 3.48
(1 строка)
```

Значение 4 присутствует в наборе, а 3.48 интерполировано. PostgreSQL предоставляет не только функции `percentile_`. Для нахождения самого часто встречающегося в группе значения имеется функция `mode`. Прежде чем приводить пример использования `mode`, я выполнил запрос, который расскажет нам чуть подробнее о содержимом таблицы:

```
test=# SELECT production, count(*)
FROM t_oil
WHERE country = 'Other Middle East'
GROUP BY production
ORDER BY 2 DESC
LIMIT 4;
 production | count 
-----+-----
50          | 5
48          | 5
52          | 5
53          | 4
(4 строки)
```

Три разных значения встречаются ровно по пять раз. Конечно, функция `mode` может вернуть только одно из них:

```
test=# SELECT country, mode() WITHIN GROUP (ORDER BY production)
FROM t_oil
WHERE country = 'Other Middle East'
GROUP BY 1;
 country | mode 
-----+-----
Other Middle East | 48
(1 строка)
```

Возвращено самое часто встречающееся значение, но SQL не говорит, сколько именно раз оно встречается, – может быть, всего один.

ГИПОТЕТИЧЕСКИЕ АГРЕГАТЫ

Гипотетические агрегаты напоминают стандартные упорядоченные наборы. Однако они отвечают на другой вопрос: что произошло бы, если бы значение присутствовало в данных? Как видим, речь идет не о значениях, фактически представленных в базе данных, а о том, что получилось бы, если бы данные были представлены.

PostgreSQL предоставляет только одну гипотетическую функцию – `rank`:

```
test=# SELECT region,
        rank(9000) WITHIN GROUP (ORDER BY production DESC NULLS LAST)
FROM t_oil
GROUP BY ROLLUP (1);
   region   | rank
-----+-----
Middle East |  21
North America | 27
            | 47
(3 строки)
```

Это означает, что если бы какая-то страна производила 9000 баррелей в сутки, то она получила бы ранг 27 в Северной Америке и ранг 21 на Ближнем Востоке.



В этом примере использована фраза `NULLS LAST`. При сортировке данных в порядке возрастания значения `null` оказываются в конце. Но и при смене порядка сортировки они должны быть в конце. Именно это и гарантирует `NULLS LAST`.

ОКОННЫЕ ФУНКЦИИ И АНАЛИТИЧЕСКИЕ СРЕДСТВА

Обсудив упорядоченные наборы, мы можем перейти к оконным функциям. Принцип работы агрегатных функций довольно прост: взять много строк и преобразовать их в меньшее количество агрегированных строк. Оконная функция ведет себя иначе. Она сравнивает текущую строку со всеми строками в группе. Количество возвращенных строк не изменяется.

Рассмотрим пример:

```
test=# SELECT avg(production) FROM t_oil;
   avg
-----
2607.5139
(1 строка)

test=# SELECT country, year, production,
        consumption, avg(production) OVER ()
FROM t_oil
LIMIT 4;
country | year | production | consumption | avg
-----+-----+-----+-----+-----
USA     | 1965 |      9014 |      11522 | 2607.5139
```

USA		1966		9579		12100		2607.5139
USA		1967		10219		12567		2607.5139
USA		1968		10600		13405		2607.5139

(4 строки)

Среднее производство в этом наборе данных составляет приблизительно 2.6 млн баррелей в сутки. Цель запроса – добавить это значение в качестве отдельного столбца. Теперь легко сравнить текущую строку со средним по всему набору.

Подчеркнем, что фраза `OVER` обязательна. Не будь ее, PostgreSQL отказалась бы обрабатывать запрос:

```
test=# SELECT country, year, production, consumption, avg(production) FROM t_oil;
ОШИБКА: столбец "t_oil.country" должен фигурировать во фразе GROUP BY или
использоваться в агрегатной функции
СТРОКА 1: SELECT country, year, production, consumption, avg(productio...
```

И это правильно, потому что необходимо точно указать, по каким данным считать среднее. Сама база данных об этом догадаться не может.

i Другие базы данных согласны обрабатывать агрегатные функции без фразы `OVER` и даже без `GROUP BY`. Но логически это неправильно и, более того, нарушает стандарт SQL.

Разбиение данных

Предыдущий запрос можно было бы легко реализовать и с помощью **подзапроса**. Но если требуется нечто большее, чем среднее по всему набору, то использование подзапросов резко усложняет запрос, превращая его в ночной кошмар. Допустим, нам нужно не среднее по всему набору, а среднее по текущей стране. Задачу поможет решить фраза `PARTITION BY`:

```
test=# SELECT country, year, production, consumption,
          avg(production) OVER (PARTITION BY country)
FROM t_oil;
```

country		year		production		consumption		avg
Canada		1965		920		1108		2123.2173
Canada		2010		3332		2316		2123.2173
Canada		2009		3202		2190		2123.2173
...								
Iran		1966		2132		148		3631.6956
Iran		2010		4352		1874		3631.6956
Iran		2009		4249		2012		3631.6956
...								

Обратите внимание, что каждой стране сопоставлено среднее значение по годам для этой страны. Фраза `OVER` определяет окно. В данном случае окном является страна, которой принадлежит строка. Иными словами, запрос сравнивает каждую строку со всеми строками, относящимися к той же стране.

i Результат не отсортирован по столбцу `year`. Поскольку в запросе явно не указан порядок сортировки, то данные могут быть возвращены в любом порядке. Помните, что SQL не обещает отсортировать результат, если вы этого не просили.

В принципе, фраза `PARTITION BY` принимает любое выражение. Обычно разбиение производится по столбцу. Но вот вам другой пример:

```
test=# SELECT year, production,
          avg(production) OVER (PARTITION BY year < 1990)
```

```
FROM t_oil
```

```
WHERE country = 'Canada'
```

```
ORDER BY year;
```

year	production	avg
1965	920	1631.6000000000000000
1966	1012	1631.6000000000000000
...		
1990	1967	2708.4761904761904762
1991	1983	2708.4761904761904762
1992	2065	2708.4761904761904762
...		

Здесь данные разбиваются по значению выражения. Выражение `year < 1990` может возвращать два значения: `true` и `false`. В зависимости от того, в какую группу попадает год, он будет учитываться в среднем за годы до 1990-го или в среднем за годы, начиная с 1990-го. В этом плане PostgreSQL проявляет немалую гибкость. Использование функций для задания принадлежности к группе часто встречается в реальных приложениях.

Упорядочение данных внутри окна

Фраза `PARTITION BY` – не единственное, что может находиться внутри `OVER`. Иногда необходимо отсортировать данные внутри окна. Для этого служит фраза `ORDER BY`, которая располагает агрегируемые данные в определенном порядке. Например:

```
test=# SELECT country, year, production,
          min(production) OVER (PARTITION BY country ORDER BY year)
```

```
FROM t_oil
```

```
WHERE year BETWEEN 1978 AND 1983 AND country IN ('Iran', 'Oman');
```

country	year	production	min
Iran	1978	5302	5302
Iran	1979	3218	3218
Iran	1980	1479	1479
Iran	1981	1321	1321
Iran	1982	2397	1321
Iran	1983	2454	1321
Oman	1978	314	314
Oman	1979	295	295

```
Oman | 1980 |      285 | 285
Oman | 1981 |      330 | 285
...
```

Из всего набора данных мы отобрали две страны (Иран и Оман) за период с 1978 по 1983 год. Имейте в виду, что в 1979 году в Иране произошла революция, что не могло не сказаться на производстве нефти. Данные отражают этот факт.

Этот запрос вычисляет минимальное производство до некоторой точки в нашем временном ряду. Тут надо хорошо понимать, что делает фраза ORDER BY внутри OVER. В данном примере PARTITION BY создает по одной группе для каждой страны и упорядочивает данные внутри группы. Функция min пробегает по отсортированным данным и вычисляет требуемые минимумы.

Если вы раньше не имели дела с оконными функциями, то запомните одну вещь: наличие фразы ORDER BY полностью меняет результат:

```
test=# SELECT country, year, production,
           min(production) OVER (),
           min(production) OVER (ORDER BY year)
```

```
FROM t_oil
```

```
WHERE year BETWEEN 1978 AND 1983
```

```
AND country = 'Iran';
```

country	year	production	min	min
Iran	1978	5302	1321	5302
Iran	1979	3218	1321	3218
Iran	1980	1479	1321	1479
Iran	1981	1321	1321	1321
Iran	1982	2397	1321	1321
Iran	1983	2454	1321	1321

```
(6 строк)
```

Если агрегат используется без ORDER BY, то он вычисляет минимум по всему набору в окне. Если же ORDER BY присутствует, то вычисляется минимум только по подмножеству до текущей точки, а это подмножество определяется заданным порядком.

Скользящие окна

До сих пор все окна были статическими. Но, например, для вычисления скользящего среднего этого недостаточно. Для скользящего среднего нужно скользящее окно, которое двигается по мере обработки данных.

В примере ниже показано, как вычисляется скользящее среднее:

```
test=# SELECT country, year, production, min(production)
           OVER (PARTITION BY country
                 ORDER BY year ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
```

```
FROM t_oil
```

```
WHERE year BETWEEN 1978 AND 1983
```

```
AND country IN ('Iran', 'Oman');
```

country	year	production	min
Iran	1978	5302	3218
Iran	1979	3218	1479
Iran	1980	1479	1321
Iran	1981	1321	1321
Iran	1982	2397	1321
Iran	1983	2454	2397
Oman	1978	314	295
Oman	1979	295	285
Oman	1980	285	285
Oman	1981	330	285
Oman	1982	338	330
Oman	1983	391	338

```
FROM generate_series(1, 5) AS id;
id | array_agg
```

```
-----+-----
1 | {1}
2 | {1,2}
3 | {1,2,3}
4 | {1,2,3,4}
5 | {1,2,3,4,5}
(5 строк)
```

Ключевые слова `UNBOUNDED PRECEDING` означают, что все находящееся до текущей строки принадлежит окну. Противоположностью `UNBOUNDED PRECEDING` является `UNBOUNDED FOLLOWING`. Рассмотрим пример:

```
test=# SELECT *,
        array_agg(id) OVER (ORDER BY id ROWS BETWEEN 2 FOLLOWING AND UNBOUNDED FOLLOWING)
FROM generate_series(1, 5) AS id;
```

```
id | array_agg
-----+-----
1 | {3,4,5}
2 | {4,5}
3 | {5}
4 |
5 |
(5 строк)
```

Как видим, можно использовать окно, находящееся в будущем. PostgreSQL в этом отношении проявляет большую гибкость.

Абстрагирование окон

Оконная функция позволяет добавлять в результирующий набор динамически вычисляемые столбцы. Однако нередко бывает так, что многие столбцы основаны на одном и том же окне. Многократно включать в запрос одни и те же фразы – не лучшая мысль, поскольку такие запросы трудно читать и сопровождать.

Фраза `WINDOW` позволяет заранее определить окно и использовать его в разных местах запроса, например:

```
SELECT country, year, production,
       min(production) OVER (w),
       max(production) OVER (w)
FROM t_oil
WHERE country = 'Canada'
      AND year BETWEEN 1980 AND 1985
WINDOW w AS (ORDER BY year);
country | year | production | min | max
-----+-----+-----+-----+-----
Canada | 1980 |          1764 | 1764 | 1764
Canada | 1981 |          1610 | 1610 | 1764
Canada | 1982 |          1590 | 1590 | 1764
Canada | 1983 |          1661 | 1590 | 1764
```

```
Canada | 1984 |      1775 | 1590 | 1775
Canada | 1985 |      1812 | 1590 | 1812
(6 строк)
```

В этом примере функции `min` и `max` вычисляются по одному и тому же окну. Разумеется, с помощью фразы `WINDOW` можно определить несколько окон – PostgreSQL не налагает серьезных ограничений в этом отношении.

Использование встроенных оконных функций

Познакомившись с основными понятиями, мы можем взглянуть на встроенные в PostgreSQL оконные функции. Мы уже видели, что оконные варианты существуют у всех стандартных агрегатных функций. Но, помимо них, PostgreSQL предлагает ряд дополнительных функций, употребляемых исключительно в оконном контексте для решения аналитических задач. В этом разделе мы обсудим наиболее важные из них.

Функции `rank` и `dense_rank`

На мой взгляд, функции `rank()` и `dense_rank()` самые интересные в SQL. Функция `rank()` возвращает номер текущей строки в своем окне. Нумерация начинается с 1.

```
test=# SELECT year, production, rank() OVER (ORDER BY production)
FROM t_oil
WHERE country = 'Other Middle East'
ORDER BY rank
LIMIT 7;
```

year	production	rank
2001	47	1
2004	48	2
2002	48	2
1999	48	2
2000	48	2
2003	48	2
1998	49	7

(7 строк)

В столбце `rank` находится номер кортежа в наборе данных. В этом примере много одинаковых строк, поэтому ранг изменяется с 2 сразу на 7. Чтобы избежать этого, воспользуемся функцией `dense_rank()`:

```
test=# SELECT year, production, dense_rank() OVER (ORDER BY production)
FROM t_oil
WHERE country = 'Other Middle East'
ORDER BY dense_rank
LIMIT 7;
```

year	production	dense_rank
2001	47	1
2004	48	2


```
...
2003 |      48 |      2
1998 |      49 |      3
(7 строк)
```

Теперь PostgreSQL присваивает ранги без пропусков.

Функция *ntile()*

В некоторых приложениях нужно разбивать данные на равные группы. Именно для этого и предназначена функция *ntile()*, демонстрируемая в примере ниже:

```
test=# SELECT year, production, ntile(4) OVER (ORDER BY production)
FROM t_oil
WHERE country = 'Iraq'
AND year BETWEEN 2000 AND 2006;
 year | production | ntile
-----+-----+-----
2003 |      1344 |      1
2005 |      1833 |      1
2006 |      1999 |      2
2004 |      2030 |      2
2002 |      2116 |      3
2001 |      2522 |      3
2000 |      2613 |      4
(7 строк)
```

Этот запрос разбивает данные на четыре группы. Поскольку выбрано всего семь строк, получить идеально одинаковые группы невозможно. Как видим, PostgreSQL делает первые три группы одинаковыми, а последнюю – более короткой. Так бывает всегда, когда не удастся разбить набор на равные группы.

i В этом примере количество строк невелико. В реальных приложениях строк миллионы, поэтому небольшой перекося в размере последней группы не составляет проблемы.

Функция *ntile()* редко используется сама по себе. Конечно, она позволяет присвоить строке номер группы. Но в реальных приложениях требуется выполнять какие-то вычисления с этими группами. Предположим, что мы хотим создать разбиение данных по *квартелям*. Вот как это делается:

```
test=# SELECT grp, min(production), max(production), count(*)
FROM (
    SELECT year, production, ntile(4) OVER (ORDER BY production) AS grp
    FROM t_oil
    WHERE country = 'Iraq'
) AS x
GROUP BY ROLLUP (1);
 grp | min  | max  | count
-----+-----+-----+-----
  1  | 285  | 1228 |    12
  2  | 1313 | 1977 |    12
```

3		1999		2422		11
4		2428		3489		11
		285		3489		46

(5 строк)

Здесь самое важное – что вычисление невозможно произвести за один шаг. Проводя учебные курсы по SQL в компании Cybertec (<https://www.cybertec-postgresql.com>), я стараюсь объяснить студентам, что если непонятно, как сделать все сразу, нужно подумать об использовании подзапроса. Обычно эта идея приносит плоды в аналитических расчетах. В данном случае мы первым делом сопоставили метку каждой группе. А затем эти группы обрабатываются в главном запросе.

Полученный результат уже можно использовать в реальном приложении (быть может, нанести на график в виде пояснительной надписи или сделать еще что-то в этом роде).

Функции *lead()* и *lag()*

Если функция `ntile()` нужна для разбиения набора данных на группы, то `lead()` и `lag()` служат для перемещения строк внутри результирующего набора. Типичное применение – вычисление разности показателя от года к году, как в следующем примере:

```
test=# SELECT year, production, lag(production, 1) OVER (ORDER BY year)
FROM t_oil
WHERE country = 'Mexico'
LIMIT 5;
```

year		production		lag
-----+-----+-----				
1965		362		
1966		370		362
1967		411		370
1968		439		411
1969		461		439

(5 строк)

Прежде чем вычислять изменение произведенной продукции, имеет смысл разобраться в том, что же делает функция `lag()`. Легко видеть, что столбец `lag` сдвинут относительно `production` на одну строку. Характер сдвига данных определен фразой `ORDER BY`. В примере выше данные сдвигаются вниз. Если бы мы написали `ORDER BY DESC`, то данные сдвигались бы вверх.

После этого все уже просто:

```
test=# SELECT year, production,
               production - lag(production, 1) OVER (ORDER BY year)
FROM t_oil
WHERE country = 'Mexico'
LIMIT 3;
```

year		production		?column?
-----+-----+-----				
1965		362		

```
1966 |      370 |      8
1967 |      411 |     41
(3 строки)
```

Нужно только вычислить разность двух столбцов. Отметим, что у функции `lag()` два параметра. Первый показывает, как столбец отображать, а второй – на сколько строк сдвигать. Если бы второй параметр был равен 7, то мы сдвигали бы на семь строк.

Отметим, что первое значение равно `Null` (как было бы во всех сдвинутых строках, для которых нет предыдущего значения).

Функция `lead()` – противоположность `lag()`; она не опускает, а поднимает строки:

```
test=# SELECT year, production,
              production - lead(production, 1) OVER (ORDER BY year)
FROM t_oil
WHERE country = 'Mexico'
LIMIT 3;
 year | production | ?column?
-----+-----+-----
 1965 |        362 |      -8
 1966 |        370 |     -41
 1967 |        411 |     -28
(3 строки)
```

PostgreSQL допускает отрицательное значение второго аргумента в функциях `lead` и `lag`. Таким образом, `lag(production, -1)` – то же самое, что `lead(production, 1)`. Однако, безусловно, лучше использовать функцию, соответствующую направлению сдвига.

До сих пор мы видели, как сдвинуть один столбец. И в большинстве приложений этого достаточно. Но PostgreSQL на этом не ограничивается. Разрешается сдвигать строки целиком:

```
test=# \x
Расширенный вывод включен.
test=# SELECT year, production, lag(t_oil, 1) OVER (ORDER BY year)
FROM t_oil
WHERE country = 'USA'
LIMIT 3;
-[ RECORD 1 ]-----
 year      | 1965
 production | 9014
 lag       |
-[ RECORD 2 ]-----
 year      | 1966
 production | 9579
 lag       | ("North America",USA,1965,9014,11522)
-[ RECORD 3 ]-----
 year      | 1967
 production | 10219
 lag       | ("North America",USA,1966,9579,12100)
```

Хорошо здесь то, что с предыдущей строкой можно сравнивать не только одно значение. Проблема, однако, в том, что PostgreSQL возвращает всю строку в виде составного типа, с которым трудно работать. Чтобы разложить составной тип на части, можно воспользоваться скобками и звездочкой:

```
test=# SELECT year, production, (lag(t_oil, 1) OVER (ORDER BY year)).*
FROM t_oil
WHERE country = 'USA'
LIMIT 3;
```

year	prod	region	country	year	prod	consumption
1965	9014					
1966	9579	N. America	USA	1965	9014	11522
1967	10219	N. America	USA	1966	9579	12100

(3 строки)

Почему это полезно? Отставание на целую строку позволяет увидеть, были ли данные вставлены более одного раза. Это простой способ обнаружить строки-дубликаты (или почти дубликаты) во временном ряде.

Рассмотрим следующий пример:

```
test=# SELECT *
      FROM (SELECT t_oil, lag(t_oil) OVER (ORDER BY year)
            FROM t_oil
            WHERE country = 'USA'
           ) AS x
      WHERE t_oil = lag;
 t_oil | lag
-----+-----
(0 строк)
```

Разумеется, в тестовых данных дубликатов нет. Но в реальном приложении это вполне может случиться, и теперь вы знаете, как это обнаружить, даже не имея первичного ключа.

i Столбец `t_oil` содержит всю строку. Результат функции `lag` в подзапросе также содержит всю строку. В PostgreSQL составные типы можно сравнивать непосредственно, если ответственные поля одинаковы. При этом поля просто сравниваются последовательно.

Функции `first_value()`, `nth_value()` и `last_value()`

Иногда необходимо вычислять данные на основе первого значения в окне. Неудивительно, что для этого существует функция `first_value()`:

```
test=# SELECT year, production, first_value(production) OVER (ORDER BY year)
FROM t_oil
WHERE country = 'Canada'
LIMIT 4;
```

year	production	first_value
1965	920	920
1966	1012	920

```
1967 |      1106 |      920
1968 |      1194 |      920
(4 строки)
```

И в этом случае необходимо указывать порядок сортировки, чтобы система знала, что значит «первое». В результате PostgreSQL помещает в последний столбец одно и то же значение. Чтобы найти последнее значение в окне, нужно было бы написать `last_value()` вместо `first_value()`.

Если нас интересует не первое и не последнее значение, а что-то среднее, то PostgreSQL предлагает функцию `nth_value()`:

```
test=# SELECT year, production, nth_value(production, 3) OVER (ORDER BY year)
FROM t_oil
WHERE country = 'Canada';
 year | production | nth_value
-----+-----+-----
 1965 |      920 |
 1966 |     1012 |
 1967 |     1106 |     1106
 1968 |     1194 |     1106
...
```

В данном случае в последний столбец помещается третье значение. Обратите внимание, что первые две строки пусты. Дело в том, что когда PostgreSQL начинает проход по данным, третье значение еще неизвестно, поэтому вместо него выводится `null`. Возникает вопрос: как пополнить временной ряд, заменив два значения `null` значениями, которые появятся позже? Вот один из способов:

```
test=# SELECT *, min(nth_value) OVER ()
FROM (
    SELECT year, production, nth_value(production, 3) OVER (ORDER BY year)
    FROM t_oil
    WHERE country = 'Canada'
) AS x
LIMIT 4;
 year | production | nth_value | min
-----+-----+-----+-----
 1965 |      920 |          | 1106
 1966 |     1012 |          | 1106
 1967 |     1106 |     1106 | 1106
 1968 |     1194 |     1106 | 1106
(4 строки)
```

В подзапросе создается неполный временной ряд. А главный запрос дополняет результат. В общем случае пополнение может быть более сложным, и подзапросы открывают возможность реализовать более сложную логику, для которой одного шага недостаточно.

Функция `row_number()`

Последняя функция, которую мы обсудим в этом разделе, – `row_number()`, ее можно использовать, чтобы вернуть виртуальный идентификатор. Вот так:

```
test=# SELECT country, production, row_number() OVER (ORDER BY production)
FROM t_oil
LIMIT 3;
```

country	production	row_number
Yemen	10	1
Syria	21	2
Yemen	26	3

(3 строки)

Функция `row_number()` просто присваивает номер строке. Очевидно, что никаких дубликатов тут быть не может.

Интересно отметить, что даже задавать порядок не нужно (если он вам не важен):

```
test=# SELECT country, production, row_number() OVER()
FROM t_oil
LIMIT 3;
```

country	production	row_number
USA	9014	1
USA	9579	2
USA	10219	3

(3 строки)

СОЗДАНИЕ СОБСТВЕННЫХ АГРЕГАТОВ

В этой книге описаны многие встроенные в PostgreSQL функции. Но того, что предлагает SQL, не всегда хватает. Однако есть и хорошая новость: мы можем написать собственный агрегат. Ниже показано, как это делается.

Создание простых агрегатов

В этом примере мы решим очень простую задачу. Человек, заказавший такси, обычно платит за посадку, например, 2,50 евро. И предположим, что каждый километр стоит 2,20 евро. Вопрос: как вычислить полную стоимость поездки?

Конечно, эту задачу легко решить без всяких пользовательских агрегатов, но мы все-таки продемонстрируем технику. Сначала создадим тестовые данные:

```
test=# CREATE TABLE t_taxi (trip_id int, km numeric);
CREATE TABLE
test=# INSERT INTO t_taxi
VALUES (1, 4.0), (1, 3.2), (1, 4.5), (2, 1.9), (2, 4.5);
INSERT 0 5
```

Для создания агрегатов в PostgreSQL имеется команда `CREATE AGGREGATE`. Ее синтаксис со временем стал таким развитым и длинным, что включать его в книгу

целиком не имеет смысла. Рекомендую обратиться к документации PostgreSQL по адресу <https://www.postgresql.org/docs/devel/static/sql-createaggregate.html>¹.

При написании агрегатной функции первым делом нужно создать функцию, которая будет вызываться для каждой строки. Она принимает промежуточное значение и данные, взятые из обрабатываемой строки. Например:

```
test=# CREATE FUNCTION taxi_per_line (numeric, numeric)
RETURNS numeric AS
$$
BEGIN
    RAISE NOTICE 'intermediate: %, per row: %', $1, $2;
    RETURN $1 + $2*2.2;
END;
$$
LANGUAGE 'plpgsql';
CREATE FUNCTION
```

Вот теперь можно создать простой агрегат:

```
test=# CREATE AGGREGATE taxi_price (numeric)
(
    INITCOND = 2.5,
    SFUNC = taxi_per_line,
    STYPE = numeric
);
CREATE AGGREGATE
```

Как и было сказано, стоимость любой поездки включает 2,50 евро за посадку в такси. Эта величина определена константой INITCOND – начальным значением для любой группы. Затем для каждой строки в группе вызывается функция – в данном случае определенная выше функция taxi_per_line. Она принимает как минимум два параметра. Первый – промежуточное значение, а остальные – значения входных данных.

Следующая команда показывает, какие данные передаются, когда и как:

```
test=# SELECT trip_id, taxi_price(km) FROM t_taxi GROUP BY 1;
NOTICE: intermediate: 2.5, per row: 4.0
NOTICE: intermediate: 11.30, per row: 3.2
NOTICE: intermediate: 18.34, per row: 4.5
NOTICE: intermediate: 2.5, per row: 1.9
NOTICE: intermediate: 6.68, per row: 4.5
 trip_id | taxi_price
-----+-----
       1 |      28.24
       2 |      16.58
(2 строки)
```

Начинаются данные о поездке 1, и функции передается начальное значение 2.50 и протяженность поездки, 4 км. Общая стоимость равна $2.50 + 4 \times 2.2$. Сле-

¹ На русском языке <https://postgrespro.ru/docs/postgresql/11/sql-createaggregate>. – Прим. перев.

дующая строка прибавляет 3.2×2.2 и т. д. Таким образом, стоимость поездки составила 28,24 евро.

Затем начинается следующая поездка. И снова – начальное значение и вызов функции для каждой строки.

В PostgreSQL любую агрегатную функцию автоматически можно использовать в качестве оконной, никаких дополнительных действий для этого не нужно:

```
test=# SELECT *, taxi_price(km) OVER (PARTITION BY trip_id ORDER BY km)
        FROM t_taxi;
```

```
NOTICE: intermediate: 2.5, per row: 3.2
NOTICE: intermediate: 9.54, per row: 4.0
NOTICE: intermediate: 18.34, per row: 4.5
NOTICE: intermediate: 2.5, per row: 1.9
NOTICE: intermediate: 6.68, per row: 4.5
```

trip_id	km	taxi_price
1	3.2	9.54
1	4.0	18.34
1	4.5	28.24
2	1.9	6.68
2	4.5	16.58

(5 строк)

Этот запрос возвращает частичные стоимости поездки на каждом отрезке.

Определенный нами агрегат вызывает одну функцию для каждой строки. Но как тогда вычислить среднее? Без добавления функции FINALFUNC такого рода вычисления были бы невозможны. Для демонстрации FINALFUNC пример необходимо расширить. Предположим, что в конце поездки клиент хочет дать таксисту на чай 10 % от стоимости поездки. Для этого нужно знать окончательную стоимость. Здесь-то и приходит на помощь функция FINALFUNC. Работает она следующим образом:

```
test=# DROP AGGREGATE taxi_price(numeric);
DROP AGGREGATE
```

Для начала удалим старый агрегат. Затем определим функцию FINALFUNC. Она получает на входе промежуточный результат и делает то, что от нее требуется:

```
test=# CREATE FUNCTION taxi_final (numeric)
        RETURNS numeric AS
$$
    SELECT $1 * 1.1;
$$
LANGUAGE sql IMMUTABLE;
CREATE FUNCTION
```

Само вычисление не представляет никаких сложностей – мы просто прибавляем 10 % к окончательной сумме.

Имея функцию, мы можем пересоздать агрегат:

```
test=# CREATE AGGREGATE taxi_price (numeric)
(
  INITCOND = 2.5,
  SFUNC = taxi_per_line,
  STYPE = numeric,
  FINALFUNC = taxi_final
);
CREATE AGGREGATE
```

В итоге стоимость будет немного больше, чем раньше:

```
test=# SELECT trip_id, taxi_price(km) FROM t_taxi GROUP BY 1;
NOTICE: intermediate: 2.5, per row: 4.0
...
trip_id | taxi_price
-----+-----
      1 |      31.064
      2 |      18.238
(2 строки)
```

PostgreSQL позаботится о группировке и обо всем остальном автоматически.

В случае простых вычислений для хранения промежуточного результата достаточно простых типов данных. Но не все операции можно выполнить, передавая лишь числа и текст. По счастью, PostgreSQL позволяет использовать для промежуточных результатов составные типы данных.

Допустим, требуется вычислить среднее для некоторого набора данных, быть может, для временного ряда. Промежуточный результат мог бы выглядеть так:

```
test=# CREATE TYPE my_intermediate AS (c int4, s numeric);
CREATE TYPE
```

Ничто не мешает создать произвольный составной тип. Просто передайте его в качестве первого аргумента и добавьте остальные параметры.

Добавление поддержки параллельных запросов

Выше мы видели простой агрегат, который не поддерживает параллельных запросов и всего, что с ними связано. Для решения этой проблемы мы улучшим и ускорим приведенные выше примеры.

Создавая агрегат, можно факультативно определить следующий параметр:

```
[ , PARALLEL = { SAFE | RESTRICTED | UNSAFE } ]
```

По умолчанию агрегатная функция не поддерживает параллельных запросов. Но ради повышения производительности имеет смысл явно указать, что умеет делать агрегат:

- UNSAFE: в этом режиме параллельные запросы не поддерживаются;
- RESTRICTED: в этом режиме агрегат можно использовать в параллельных запросах, но выполнение ограничено только ведущим процессом группы;

- SAFE: в этом режиме параллельные запросы поддерживаются в полном объеме.

Помечая функцию как SAFE, помните, что она не должна иметь побочных эффектов. Результат выполнения запроса не должен зависеть от порядка вызовов функции. Только в этом случае PostgreSQL следует разрешать распараллеливание операций. Примерами функций без побочных эффектов являются $\sin(x)$ и $\text{length}(s)$. Подходящими кандидатами являются неизменяемые (IMMUTABLE) функции, потому что они гарантированно возвращают один и тот же результат для одних и тех же аргументов. Функции, помеченные как STABLE, тоже будут работать при некоторых ограничениях.

Повышение эффективности

Определенные до сих пор агрегаты уже способны на многое. Но если используются скользящие окна, то количество вызовов функции растет экспоненциально. Вот что происходит:

```
test=# SELECT taxi_price(x::numeric) OVER (ROWS BETWEEN 0 FOLLOWING AND 3 FOLLOWING)
FROM generate_series(1, 5) AS x;
NOTICE: intermediate: 2.5, per row: 1
NOTICE: intermediate: 4.7, per row: 2
NOTICE: intermediate: 9.1, per row: 3
NOTICE: intermediate: 15.7, per row: 4
NOTICE: intermediate: 2.5, per row: 2
NOTICE: intermediate: 6.9, per row: 3
NOTICE: intermediate: 13.5, per row: 4
NOTICE: intermediate: 22.3, per row: 5
...
```

Для каждой строки PostgreSQL обрабатывает все окно целиком. Если скользящее окно велико, то эффективность падает. Чтобы исправить это, расширим определенный выше агрегат. Но сначала удалим старый:

```
DROP AGGREGATE taxi_price(numeric);
```

Нам понадобятся две функции. Функция `msfunc` прибавляет к промежуточному результату следующую строку окна:

```
CREATE FUNCTION taxi_msfunc(numeric, numeric)
  RETURNS numeric AS
$$
BEGIN
  RAISE NOTICE 'taxi_msfunc called with % and %', $1, $2;
  RETURN $1 + $2;
END;
$$ LANGUAGE 'plpgsql' STRICT;
```

Функция `minvfunc` исключает из промежуточного результата значение, вышедшее за пределы окна:

```
CREATE FUNCTION taxi_minvfunc(numeric, numeric) RETURNS numeric AS
$$
```

```
BEGIN
  RAISE NOTICE 'taxi_minvfunc called with % and %', $1, $2;
  RETURN $1 - $2;
END;
$$
LANGUAGE 'plpgsql' STRICT;
```

В этом примере достаточно операций сложения и вычитания. Но, в принципе, вычисление может быть сколь угодно сложным.

Далее пересоздадим агрегат:

```
CREATE AGGREGATE taxi_price (numeric)
(
  INITCOND = 0,
  STYPE = numeric,
  SFUNC = taxi_per_line,
  MSFUNC = taxi_msfunc,
  MINVFUNC = taxi_minvfunc,
  MSTYPE = numeric
);
```

И снова выполним тот же самый запрос:

```
test# SELECT taxi_price(x::numeric) OVER (ROWS BETWEEN 0 FOLLOWING AND 3 FOLLOWING)
FROM generate_series(1, 5) AS x;
NOTICE: taxi_msfunc called with 1 and 2
NOTICE: taxi_msfunc called with 3 and 3
NOTICE: taxi_msfunc called with 6 and 4
NOTICE: taxi_minvfunc called with 10 and 1
NOTICE: taxi_msfunc called with 9 and 5
NOTICE: taxi_minvfunc called with 14 and 2
NOTICE: taxi_minvfunc called with 12 and 3
NOTICE: taxi_minvfunc called with 9 and 4
```

Количество вызовов функций существенно уменьшилось. Для каждой строки функция вызывается фиксированное (небольшое) число раз. Больше нет необходимости пересчитывать один и тот же фрейм окна многократно.

Написание гипотетических агрегатов

Писать агрегаты нетрудно, а выигрыш при выполнении сложных операций может быть заметным. В этом разделе мы напомним гипотетический агрегат, который обсуждали выше.

Реализация гипотетических агрегатов не сильно отличается от написания нормальных. По-настоящему трудная задача – понять, когда их использовать. Чтобы сделать этот раздел максимально понятным, я взял тривиальный пример: каким будет результат добавления *abc* в конец строки при заданном порядке сортировки?

Вот как это работает:

```
CREATE AGGREGATE имя ( [ [ режим_аргумента ] [ имя_аргумента ]
                      тип_данных_аргумента [ , ... ] ]
```

```
ORDER BY [ режим_аргумента ] [ имя_аргумента ] тип_данных_аргумента [ , ...])
(
  SFUNC = sfunc,
  STYPE = тип_данных_состояния
  [ , SSPACE = размер_данных_состояния ] [ , FINALFUNC = функция_завершения ]
  [ , FINALFUNC_EXTRA ]
  [ , INITCOND = начальное_условие ]
  [ , PARALLEL = { SAFE | RESTRICTED | UNSAFE } ] [ , HYPOTHETICAL ]
)
```

Нам понадобятся две функции. Функция `sfunc` будет вызываться для каждой строки:

```
CREATE FUNCTION hypo_sfunc(text, text)
  RETURNS text AS
$$
  BEGIN
    RAISE NOTICE 'hypo_sfunc called with % and %', $1, $2;
    RETURN $1 || $2;
  END;
$$ LANGUAGE 'plpgsql';
```

Этой функции передается два параметра, обрабатываемых так же, как и раньше. И точно так же можно определить функцию завершения:

```
CREATE FUNCTION hypo_final(text, text, text)
  RETURNS text AS
$$
  BEGIN
    RAISE NOTICE 'hypo_final called with %, %, and %', $1, $2, $3;
    RETURN $1 || $2;
  END;
$$ LANGUAGE 'plpgsql';
```

Имея эти две функции, уже можно создать гипотетический агрегат:

```
CREATE AGGREGATE whatif(text ORDER BY text)
(
  INITCOND = 'START',
  STYPE = text,
  SFUNC = hypo_sfunc,
  FINALFUNC = hypo_final,
  FINALFUNC_EXTRA = true,
  HYPOTHETICAL
);
```

Отметим, что агрегат помечен как гипотетический, чтобы PostgreSQL знала, о чем идет речь.

Агрегат создан, теперь можно его выполнить:

```
test=# SELECT whatif('abc'::text) WITHIN GROUP (ORDER BY id::text)
FROM generate_series(1, 3) AS id;
NOTICE: hypo_sfunc called with START and 1
NOTICE: hypo_sfunc called with START1 and 2
```

```
NOTICE: hypo_sfunc called with START12 and 3
NOTICE: hypo_final called with START123, abc, and <NULL>
whatif
-----
START123abc
(1 строка)
```

Чтобы разобраться во всех таких агрегатах, нужно ясно понимать, когда какая функция вызывается и как работает механизм в целом.

РЕЗЮМЕ

В этой главе мы узнали о дополнительных возможностях SQL. Помимо простых агрегатов, PostgreSQL предоставляет упорядоченные наборы, наборы группирования, оконные функции, а также интерфейс для создания пользовательских агрегатов. Агрегирование внутри базы данных имеет то преимущество, что код легко читать, а производительность обычно выше, по сравнению с внешним агрегированием.

В главе 5 мы переключим внимание на административные задачи: управление файлами журналов, осмысление статистики системы и реализацию мониторинга.

Глава 5

Журналы и статистика системы

В главе 4 мы многое узнали о передовых средствах SQL и смогли увидеть SQL в новом свете. Но база данных не сводится к одному лишь написанию хитроумного SQL-кода. Иногда приходится заботиться о том, чтобы все работало как надо. Для этого важно следить за статистикой системы, файлами журналов и т. д. Мониторинг – ключ к профессиональной эксплуатации СУБД. По счастью, PostgreSQL располагает многочисленными средствами для мониторинга баз данных, и в этой главе мы научимся ими пользоваться.

В этой главе рассматриваются следующие вопросы:

- сбор статистических данных о работе системы;
- создание файлов журналов;
- сбор важной информации;
- интерпретация статистики системы.

Прочитав эту главу, вы сможете профессионально настраивать инфраструктуру ведения журналов в PostgreSQL и пользоваться находящейся в журналах информацией.

СБОР СТАТИСТИЧЕСКИХ ДАННЫХ О РАБОТЕ СИСТЕМЫ

Первое, что нужно усвоить, – как интерпретировать и использовать встроенную в PostgreSQL статистику. Лично я считаю, что невозможно повысить производительность и надежность системы, не собрав предварительно данных для принятия обоснованных решений.

В этом разделе мы рассмотрим, какую статистику собирает PostgreSQL во время работы, и подробно объясним, как получить дополнительную информацию от СУБД.

Системные представления в PostgreSQL

PostgreSQL предлагает обширный набор системных представлений, позволяющих администраторам и разработчикам глубже изучить, что происходит внут-

ри системы. Беда в том, что многие собирают все эти данные, но не знают, что с ними делать. Общее правило таково: если чего-то не понимаешь, то и наносить это на график нет смысла. Цель этого раздела – пролить свет на то, что все-таки предлагает PostgreSQL, в надежде помочь вам правильно распорядиться предлагаемым богатством.

Динамическая проверка выполняемых запросов

Приступая к инспекции системы, я всегда начинаю с одного и того же системного представления. Речь, конечно, идет о представлении `pg_stat_activity`. Его идея – показать, что происходит прямо сейчас.

```
test=# \d pg_stat_activity
```

Представление "pg_catalog.pg_stat_activity"			
Столбец	Тип	Правило сортировки	Допустимость NULL
datid	oid		
datname	name		
pid	integer		
usesysid	oid		
username	name		
application_name	text		
client_addr	inet		
client_hostname	text		
client_port	integer		
backend_start	timestamp with time zone		
xact_start	timestamp with time zone		
query_start	timestamp with time zone		
state_change	timestamp with time zone		
wait_event_type	text		
wait_event	text		
state	text		
backend_xid	xid		
backend_xmin	xid		
query	text		
backend_type	text		

Представление `pg_stat_activity` содержит одну строку на каждое активное соединение. Вы видите идентификатор объекта базы данных (`datid`), имя базы данных, с которой установлено соединение, и идентификатор процесса, обслуживающего это соединение (`pid`). PostgreSQL также сообщает, кто именно подключился к базе (`username`; обратите внимание на отсутствующую букву `r`) и идентификатор внутреннего объекта этого пользователя (`usesysid`).

Далее следует поле `application_name`, о котором имеет смысл сказать несколько слов. Вообще говоря, имя приложения (`application_name`) устанавливается пользователем:

```
test=# SET application_name TO 'www.cybertec-postgresql.com';
SET
test=# SHOW application_name;
application_name
```

 www.cybertec-postgresql.com
 (1 строка)

Объясним, в чем его смысл. Предположим, что с одного IP-адреса открыто несколько тысяч соединений. Может ли администратор сказать, чем занимается конкретное соединение? Невозможно помнить SQL-код каждого приложения наизусть. Если клиент был настолько любезен, что задал имя приложения, то понять, для чего открыто соединение, гораздо проще. В примере выше я указал имя домена, из которого открыто соединение. В случае проблемы это упростит поиск соединений, которые могут вызывать аналогичные проблемы.

Следующие три столбца (`client_`) позволяют узнать, откуда было открыто соединение. PostgreSQL показывает IP-адреса и (если это задано в конфигурации) даже имена хостов.

Поле `backend_start` сообщает, когда было открыто соединение, а `xact_start` – когда начата транзакция. Имеются также поля `query_start` и `state_change`. Когда-то давным-давно PostgreSQL показывала только активные запросы. В те дни, когда выполнение запросов занимало куда больше времени, чем сейчас, это имело смысл. На современном оборудовании OLTP-запросы иногда выполняются за доли миллисекунды, поэтому трудно поймать момент, когда такой запрос потенциально может нанести вред. Решение – показывать либо активный, либо предыдущий запрос на данном соединении.

Вот возможный результат запроса:

```
test=# SELECT pid, query_start, state_change, state, query
FROM pg_stat_activity;
```

```
...
-[ RECORD 2 ] +-----
pid          | 28001
query_start  | 2018-11-05 10:03:57.575593+01
state_change | 2018-11-05 10:03:57.575595+01
state        | active
query        | SELECT pg_sleep(10000000);
```

Мы видим, что на втором соединении выполняется функция `pg_sleep`. По завершении этого запроса картина изменится следующим образом:

```
-[ RECORD 2 ] +-----
pid          | 28001
query_start  | 2018-11-05 10:03:57.575593+01
state_change | 2018-11-05 10:05:10.388522+01
state        | idle
query        | SELECT pg_sleep(10000000);
```

Теперь запрос помечен как `idle` (не выполняется). Разность между значениями `state_change` и `query_start` равна времени выполнения запроса.

Таким образом, `pg_stat_activity` дает общее представление о происходящем в системе. Благодаря новому полю `state_change` находить дорогостоящие запросы стало гораздо проще.

Но теперь возникает вопрос: как избавиться от обнаруженных плохих запросов? Для этого PostgreSQL предоставляет две функции: `pg_cancel_backend` и `pg_terminate_backend`. Функция `pg_cancel_backend` снимает запрос, но не закрывает соединение. Функция `pg_terminate_backend` ведет себя более радикально – она закрывает соединение вместе с запросом.

Если вы хотите отключить всех пользователей, кроме себя самого, нужно выполнить такой запрос:

```
test=# SELECT pg_terminate_backend(pid)
        FROM pg_stat_activity
        WHERE pid <> pg_backend_pid() AND backend_type = 'client backend'
pg_terminate_backend
-----
t
t
(2 строки)
```

Если отключили вас, то вы увидите такое сообщение:

```
test=# SELECT pg_sleep(10000000);
FATAL: terminating connection due to administrator command server closed
the connection unexpectedly
```

Скорее всего, это означает, что сервер аварийно остановился до или во время выполнения запроса. Соединение с сервером разорвано.



После разрыва соединения `psql` попытается восстановить работу. Но это относится только к `psql`, а не к другим клиентам и уж тем более не к клиентским библиотекам.

Получение информации о базах данных

Изучив активные соединения с сервером, можно копнуть глубже и поинтересоваться статистикой на уровне отдельных баз данных. Представление `pg_stat_database` возвращает по одной строке на каждую базу данных, обслуживаемую экземпляром PostgreSQL.

Вот что вы увидите:

```
test=# \d pg_stat_database
```

Столбец	Тип	Правило сортировки	Допустимость NULL
<code>datid</code>	<code>oid</code>		
<code>datname</code>	<code>name</code>		
<code>numbackends</code>	<code>integer</code>		
<code>xact_commit</code>	<code>bigint</code>		
<code>xact_rollback</code>	<code>bigint</code>		
<code>blks_read</code>	<code>bigint</code>		
<code>blks_hit</code>	<code>bigint</code>		
<code>tup_returned</code>	<code>bigint</code>		
<code>tup_fetched</code>	<code>bigint</code>		
<code>tup_inserted</code>	<code>bigint</code>		
<code>tup_updated</code>	<code>bigint</code>		

tup_deleted	bigint		
conflicts	bigint		
temp_files	bigint		
temp_bytes	bigint		
deadlocks	bigint		
blk_read_time	double precision		
blk_write_time	double precision		
stats_reset	timestamp with time zone		

После идентификатора и имени базы данных следует столбец `numpbackends`, который показывает, сколько соединений с базой данных открыто в данный момент.

Далее идут столбцы `xact_commit` и `xact_rollback`. Они показывают, что приложение делает чаще: фиксацию или откат. Столбцы `blks_hit` и `blks_read` сообщают о количестве попаданий и промахов кеша. Имейте в виду, что речь обычно идет о кеше разделяемых буферов. На уровне базы данных невозможно отличить чтение из кеша файловой системы и с физического диска. В компании Cybertec (<https://www.cybertec-postgresql.com>) мы часто анализируем выдачу `pg_stat_database` совместно со сведениями о времени ожидания диска и о промахах системного кеша. Это позволяет лучше понять, что происходит в системе.

Столбцы с префиксом `tup_` позволяют узнать, что превалирует в системе: операции чтения или операции записи.

Далее следуют столбцы `temp_files` и `temp_bytes`. Они весьма важны, потому что показывают, приходится ли системе создавать временные файлы на диске, что неизбежно замедляет работу. Каковы возможные причины использования временных файлов? Перечислим основные:

- **неправильная настройка:** если параметр `work_mem` слишком мал, то нет возможности выполнять операции в оперативной памяти, так что PostgreSQL обращается к диску;
- **дурацкие операции:** довольно часто пользователи мучат систему дорогостоящими, но бессмысленными запросами. Если в OLTP-системе создается много временных файлов, поищите «тяжелые» запросы;
- **индексирование и другие административные задачи:** время от времени приходится строить индексы или выполнять DDL-команды. При этом могут создаваться временные файлы, но это необязательно признак проблемы (в большинстве случаев).

Короче говоря, временные файлы могут создаваться, даже если система в полном порядке. Но за ними все же нужно приглядывать и следить, чтобы они не создавались слишком часто.

Наконец, есть два более важных поля: `blk_read_time` и `blk_write_time`. По умолчанию они пусты, т. к. данные для них не собираются. Идея в том, чтобы показать, сколько времени было потрачено на ввод-вывод. А пусты они потому, что по умолчанию параметр `track_io_timing` равен `off`. И тому есть основательные причины. Представьте, что вы хотите узнать, сколько времени уходит на чтение 1 млн блоков. Для этого нужно для каждого блока дважды вызвать функ-

цию `time` из стандартной библиотеки `C`, так что мы получаем 2 млн дополнительных вызовов функции, только чтобы прочитать 8 ГБ данных. Насколько велики такие накладные расходы, зависит от быстродействия вашей системы.

По счастью, существует инструмент, позволяющий оценить стоимость хронометража:

```
[hs@zenbook ~]$ pg_test_timing
Testing timing overhead for 3 seconds.
Per loop time including overhead: 23.16 nsec
Histogram of timing durations:
< usec    % of total    count
1         97.70300    126549189
2         2.29506     2972668
4         0.00024      317
8         0.00008      101
16        0.00160     2072
32        0.00000      5
64        0.00000      6
128       0.00000      4
256       0.00000      0
512       0.00000      0
1024      0.00000      4
2048      0.00000      2
```

В моем случае накладные расходы, связанные с включением режима `track_io_timing` в отдельном сеансе или в файле `postgresql.conf`, составляют примерно 23 нс, что приемлемо. Профессиональные высококлассные серверы могут снизить это время до 14 нс, а плохо настроенная виртуализация – увеличить до 1400 или даже 1900 нс. Работая в облаке, следует ожидать величины порядка 100–120 нс (в большинстве случаев). Если вы видите четырехзначные числа, то включение хронометража ввода-вывода наверняка приведет к ощутимым накладным расходам, из-за которых работа система замедлится. Общее правило таково: на физическом оборудовании хронометраж не вызывает проблем, при работе в виртуальной системе надо проверять.



Можно также включать различные параметры избирательно с помощью команд `ALTER DATABASE`, `ALTER USER` и т. п.

Получение информации о таблицах

Уяснив, что происходит на уровне базы данных, можно копнуть еще глубже и поинтересоваться отдельными таблицами. В этом нам помогут два системных представления: `pg_stat_user_tables` и `pg_statio_user_tables`. Сначала рассмотрим первое:

```
test=# \d pg_stat_user_tables
```

Представление "pg_catalog.pg_stat_user_tables"			
Столбец	Тип	Правило сортировки	Допустимость NULL
<code>relid</code>	<code>oid</code>		
<code>schemaname</code>	<code>name</code>		

relname	name		
seq_scan	bigint		
seq_tup_read	bigint		
idx_scan	bigint		
idx_tup_fetch	bigint		
n_tup_ins	bigint		
n_tup_upd	bigint		
n_tup_del	bigint		
n_tup_hot_upd	bigint		
n_live_tup	bigint		
n_dead_tup	bigint		
n_mod_since_analyze	bigint		
last_vacuum	timestamp with time zone		
last_autovacuum	timestamp with time zone		
last_analyze	timestamp with time zone		
last_autoanalyze	timestamp with time zone		
vacuum_count	bigint		
autovacuum_count	bigint		
analyze_count	bigint		
autoanalyze_count	bigint		

На мой взгляд, `pg_stat_user_tables` – одно из важнейших и вместе с тем самых недопонимаемых или даже игнорируемых системных представлений. У меня такое ощущение, что многие люди смотрят на него, но не могут в полной мере осознать, что же они видят. А ведь при правильном использовании `pg_stat_user_tables` можно сравнить с божественным откровением.

Прежде чем переходить к интерпретации данных, важно понимать, что за поля нам показывают. Прежде всего для каждой таблицы имеется одна запись, в которой, в частности, присутствует количество последовательных просмотров таблицы (`seq_scan`). А поле `seq_tup_read` говорит, сколько кортежей система прочитала во время этих просмотров.



Запомните поле `seq_tup_read`; оно содержит важнейшую информацию, которая может помочь в решении проблем с производительностью.

Следующим в списке является поле `idx_scan`, которое показывает, сколько раз для доступа к этой таблице использовался индекс. Кроме того, PostgreSQL сообщает, сколько строк было прочитано в процессе просмотра по индексу. Далее мы видим несколько полей с префиксом `n_tup_`. Они говорят, сколько было выполнено операций вставки, обновления и удаления. Самое важное из них – `n_tup_hot_upd`. При выполнении команды `UPDATE` PostgreSQL должна скопировать строку, чтобы гарантировать правильную работу `ROLLBACK`. Горячее обновление (`HOT UPDATE`) хорошо тем, что новая строка остается в том же блоке, что исходная, – для производительности это благо. Большое количество горячих обновлений означает, что вы на правильном пути в ситуации, когда рабочая нагрузка включает много команд `UPDATE`. Невозможно сказать, каким должно быть идеальное соотношение между обычными и горячими обновлениями, все зависит от ситуации. Пользователи должны сами понять, при какой рабо-

чей нагрузке большое количество операций, выполняемых на месте, приносит ощутимый выигрыш. Общее правило таково: чем больше операций обновления в рабочей нагрузке, тем выгоднее иметь много горячих обновлений.

Наконец, имеется статистика работы VACUUM, которая не нуждается в пояснениях.

Интерпретации данных из представления pg_stat_user_tables

Читать эти данные интересно, но если вы не знаете, что с ними делать, то вся затея бессмысленна. Один из способов использования pg_stat_user_tables – выявление таблиц, нуждающихся в индексах. Правильное направление укажет следующий запрос, который верой и правдой служит мне много лет:

```
SELECT schemaname, relname, seq_scan, seq_tup_read,
       seq_tup_read / seq_scan AS avg, idx_scan
FROM pg_stat_user_tables
WHERE seq_scan > 0
ORDER BY seq_tup_read DESC
LIMIT 25;
```

Его идея – найти большие таблицы, которые часто просматриваются последовательно. Эти таблицы окажутся в начале списка и будут содержать упомогающие значения в поле seq_tup_read.



Просматривайте список сверху вниз и ищите дорогостоящие операции просмотра. Помните, что последовательный просмотр – не всегда зло. Они естественно возникают при резервном копировании, в аналитических запросах и других подобных вещах, где совершенно безвредны. Но если большие таблицы все время просматриваются последовательно, то ни о какой производительности не может быть и речи.

Этот запрос – чистое золото, он помогает найти таблицы с отсутствующими индексами. Практический опыт на протяжении почти двадцати лет снова и снова убеждал меня, что недостающие индексы – самая главная причина низкой производительности.

Определив, над какими таблицами, вероятно, нет нужных индексов, взгляните на то, как они кешируются. Для этого пригодится представление pg_statio_user_tables, которое содержит разнообразную информацию о вводе-выводе, в т. ч. о кешировании таблицы (heap_blks_), индексов (idx_blks_) и **методике хранения сверхбольших атрибутов** (The Oversized Attribute Storage Technique – **TOAST**). Наконец, можно получить дополнительные сведения о просмотрах по идентификатору кортежа, которые обычно не влияют на общую производительность системы.

```
test=# \d pg_statio_user_tables
```

```
Представление "pg_catalog.pg_statio_user_tables"
```

Столбец	Тип	Правило сортировки	Допустимость NULL
relid	oid		
schemaname	name		
relname	name		

heap_blks_read	bigint		
heap_blks_hit	bigint		
idx_blks_read	bigint		
idx_blks_hit	bigint		
toast_blks_read	bigint		
toast_blks_hit	bigint		
tidx_blks_read	bigint		
tidx_blks_hit	bigint		

Хотя представление `pg_statio_user_tables` содержит важную информацию, обычно `pg_stat_user_tables` дает больше действительно ценных сведений (например, о недостающих индексах).

Изучение индексов

Представление `pg_stat_user_tables` важно для поиска недостающих индексов, но иногда необходимо выявить индексы, которых на самом деле быть не должно. Недавно я ездил в командировку в Германию и обнаружил систему, содержащую в основном бесполезные индексы (74% занятого места на диске). В небольшой базе данных это не очень страшно, но в больших системах бесполезные индексы, занимающие сотни гигабайтов, наносят серьезный ущерб производительности.

К счастью, для выявления бесполезных индексов есть представление `pg_stat_user_indexes`:

```
test=# \d pg_stat_user_indexes
```

Представление "pg_catalog.pg_stat_user_indexes"

Столбец	Тип	Правило сортировки	Допустимость NULL
reloid	oid		
indexreloid	oid		
schemaname	name		
relname	name		
indexrelname	name		
idx_scan	bigint		
idx_tup_read	bigint		
idx_tup_fetch	bigint		

Это представление говорит, сколько раз использовался каждый индекс (`idx_scan`). При этом указывается, над какой таблицей из какой схемы этот индекс построен. Для обогащения содержащейся в представлении информации я предлагаю следующий SQL-запрос:

```
SELECT schemaname, relname, indexrelname, idx_scan,
       pg_size_pretty(pg_relation_size(indexreloid)) AS idx_size,
       pg_size_pretty(sum(pg_relation_size(indexreloid))
                     OVER (ORDER BY idx_scan, indexreloid)) AS total
FROM pg_stat_user_indexes
ORDER BY 6 ;
```

Он дает очень полезные сведения. Помимо информации о том, сколько раз использовался индекс, он сообщает, сколько места занимает каждый индекс.

И в шестом столбце место, занятое всеми индексами над данной таблицей, суммируется. Теперь можно еще раз подумать, а так ли нужны редко используемые индексы. Сформулировать общее правило удаления индекса трудно, тут не обойтись без ручных проверок.



Не стоит слепо удалять индексы. Иногда они не используются просто потому, что пользователи работают с приложением не так, как ожидалось. Если состав пользователей изменится (например, наймут новую секретаршу), то индекс может вновь стать очень ценным объектом.

Существует также представление `pg_statio_user_indexes`, содержащее информацию о кешировании индекса. Оно интересно, но обычно не дает возможности серьезно продвинуться в понимании системы.

Мониторинг фонового рабочего процесса

В этом разделе мы познакомимся со статистикой фонового процесса записи. Возможно, вам известно, что сервер обычно не записывает блоки данных напрямую. Это делает фоновый процесс записи или процесс контрольной точки.

Понять, как записываются данные, поможет представление `pg_stat_bgwriter`:

```
test=# \d pg_stat_bgwriter
```

Представление "pg_catalog.pg_stat_bgwriter"			
Столбец	Тип	Правило сортировки	Допустимость NULL
checkpoints_timed	bigint		
checkpoints_req	bigint		
checkpoint_write_time	double precision		
checkpoint_sync_time	double precision		
buffers_checkpoint	bigint		
buffers_clean	bigint		
maxwritten_clean	bigint		
buffers_backend	bigint		
buffers_backend_fsync	bigint		
buffers_alloc	bigint		
stats_reset	timestamp with time zone		

Прежде всего нужно обратить внимание на первые два столбца. Ниже в этой книге вы узнаете, что PostgreSQL регулярно записывает контрольные точки. Это необходимо, чтобы гарантировать попадание данных на физический диск. О том, что интервал между контрольными точками слишком мал, расскажет поле `checkpoint_req` (количество запрошенных контрольных точек). Слишком большая величина в нем может означать, что записывается очень много данных, что свидетельствует о высоком трафике. Кроме того, PostgreSQL расскажет, сколько времени потребовалось на запись данных во время контрольной точки и сколько времени ушло на синхронизацию. Поле `buffers_checkpoint` говорит, сколько буферов было записано во время контрольной точки, а поле `buffers_clean` – сколько их было записано фоновым процессом записи.

Но и это еще не все: поле `maxwritten_clean` говорит, сколько раз фоновый процесс записи останавливал сброс грязных страниц на диск, потому что записал слишком много буферов.

Наконец, имеются поля `buffers_backend` (сколько буферов записано самими обслуживающими процессами), `buffers_backend_fsync` (сколько раз обслуживающим процессам пришлось выполнять `fsync` самостоятельно) и `buffers_alloc` (количество выделенных буферов). Вообще говоря, нехорошо, когда обслуживающий процесс начинает сам записывать буферы на диск.

Мониторинг, архивация и потоковая репликация

В этом разделе мы рассмотрим некоторые средства, относящиеся к репликации и архивации журнала транзакций. Первым делом нужно обратиться к представлению `pg_stat_archiver`, которое расскажет о процессе архивации, занятом перемещением журнала транзакций (WAL) с главного сервера на устройство резервного хранения:

```
test=# \d pg_stat_archiver
```

Представление "pg_catalog.pg_stat_archiver"			
Столбец	Тип	Правило сортировки	Допустимость NULL
archived_count	bigint		
last_archived_wal	text		
last_archived_time	timestamp with time zone		
failed_count	bigint		
last_failed_wal	text		
last_failed_time	timestamp with time zone		
stats_reset	timestamp with time zone		

Прежде всего мы видим, сколько файлов журнала транзакций было архивировано (`archived_count`), а также какой файл был архивирован последним и когда (`last_archived_wal` и `last_achived_time`).

Конечно, знать количество WAL-файлов интересно, но не так уж важно. Лучше обратить внимание на поля `failed_count`, `last_failed_wal` и `last_failed_time`. Если при архивации журнала транзакций произошла ошибка, `last_failed_wal` сообщит имя последнего неархивированного файла, а поле `last_failed_time` – время ошибки. Рекомендуется следить за этими полями, потому что иначе архивация может остановиться, а вы об этом даже не узнаете.

Если в вашей системе работает потоковая репликация, то очень пригодятся следующие два представления. Первое, `pg_stat_replication`, содержит информацию о процессе потоковой репликации с главного сервера на резервный. Показывается по одной записи для каждого процесса отправки WAL. Если нет ни одной записи, значит, потоковая репликация журнала транзакций не проводится, а это, возможно, не то, чего вы хотели.

Вот как устроено представление `pg_stat_replication`:


```
test=# \d pg_stat_replication
```

Представление "pg_catalog.pg_stat_replication"			
Столбец	Тип	Правило сортировки	Допустимость NULL
pid	integer		
usesysid	oid		
username	name		
application_name	text		
client_addr	inet		
client_hostname	text		
client_port	integer		
backend_start	timestamp with time zone		
backend_xmin	xid		
state	text		
sent_lsn	pg_lsn		
write_lsn	pg_lsn		
flush_lsn	pg_lsn		
replay_lsn	pg_lsn		
write_lag	interval		
flush_lag	interval		
replay_lag	interval		
sync_priority	integer		
sync_state	text		

Здесь мы видим поле, содержащее имя пользователя, подключившегося посредством потоковой репликации. Далее идут имя приложения и сведения о соединении (*client_*). Также PostgreSQL сообщает, когда было открыто потоковое соединение. В производственной системе недавнее время открытия может означать сетевую проблему, а то и что-нибудь похуже (вопросы надежности и т. п.). Поле *state* показывает, в каком состоянии находится другая сторона потока.

Дополнительные сведения на тему репликации приведены в главе 10. Существуют поля, содержащие информацию о том, какая часть журнала транзакций была передана по сети (*sent_lsn*), сколько данных было отправлено ядру (*write_lsn*), сколько было сброшено на диск (*flush_lsn*) и сколько уже воспроизведено (*replay_lsn*). Наконец, указано состояние синхронизации. Начиная с версии PostgreSQL 10.0 имеются дополнительные поля вида **_lag*, содержащие задержки при выполнении операций главным и резервным серверами.

Если *pg_stat_replication* можно опрашивать на отправляющей стороне системы репликации, то *pg_stat_wal_receiver* – на принимающей стороне. Оно дает аналогичную информацию, которую можно извлечь на стороне реплики.

Вот как определено это представление:

```
test=# \d pg_stat_wal_receiver
```

Представление "pg_catalog.pg_stat_wal_receiver"			
Столбец	Тип	Правило сортировки	Допустимость NULL
pid	integer		
status	text		
receive_start_lsn	pg_lsn		

receive_start_tli	integer		
received_lsn	pg_lsn		
received_tli	integer		
last_msg_send_time	timestamp with time zone		
last_msg_receipt_time	timestamp with time zone		
latest_end_lsn	pg_lsn		
latest_end_time	timestamp with time zone		
slot_name	text		
sender_host	text		
sender_port	integer		
conninfo	text		

Прежде всего PostgreSQL сообщает идентификатор процесса-приемника WAL. Затем идет состояние соединения. Поле `receive_start_lsn` сообщает позицию в журнале транзакций на момент запуска приемника WAL, а поле `receive_start_tli` – состояние линии времени в этот момент. Рано или поздно может возникнуть желание узнать последнюю позицию в файле WAL и последнее состояние линии времени. Эти значения хранятся в полях `received_lsn` и `received_tli`.

Следующие два поля, `last_msg_send_time` и `last_msg_receipt_time`, содержат временные метки: время отправки последнего сообщения и время его получения.

Поле `latest_end_lsn` содержит последнюю позицию в журнале транзакций, которая была сообщена процессу отправки WAL в момент `latest_end_time`. Поле `slot_name` – запутанный способ представить информацию о соединении. В PostgreSQL 11 добавлены поля `sender_host`, `sender_port` и `conninfo`, содержащие сведения о сервере, к которому подключен приемник WAL.

Проверка SSL-подключений

Многие пользователи PostgreSQL используют SSL для шифрования подключений клиента к серверу. В недавних версиях PostgreSQL появилось представление `pg_stat_ssl`, содержащее сведения о таких подключениях:

```
test=# \d pg_stat_ssl
```

Представление "pg_catalog.pg_stat_ssl"			
Столбец	Тип	Правило сортировки	Допустимость NULL
pid	integer		
ssl	boolean		
version	text		
cipher	text		
bits	integer		
compression	boolean		
clientdn	text		

Каждый процесс представлен своим идентификатором. Если подключение зашифровано по протоколу SSL, то второе поле равно `true`. Третье и четвертое поля содержат версию протокола и шифр соответственно. Наконец, указана разрядность алгоритма шифрования, признак сжатия и отличительное имя (Distinguished Name DN), взятое из сертификата клиента.

Инспекция транзакций в режиме реального времени

До сих пор мы обсуждали статистические сведения о таблицах. Все они имеют общую цель: показать, что происходит в системе в целом. Но что, если разработчику интересно узнать о конкретной транзакции? На помощь приходит представление `pg_stat_xact_user_tables`. В нем нет данных обо всех транзакциях в системе, только о вашей текущей транзакции:

```
test=# \d pg_stat_xact_user_tables
        Представление "pg_catalog.pg_stat_xact_user_tables"
    Столбец      | Тип      | Правило сортировки | Допустимость NULL
-----+-----+-----+-----
 relid           | oid      |                     | 
 schemaname      | name     |                     | 
 relname         | name     |                     | 
 seq_scan        | bigint   |                     | 
 seq_tup_read     | bigint   |                     | 
 idx_scan        | bigint   |                     | 
 idx_tup_fetch   | bigint   |                     | 
 n_tup_ins       | bigint   |                     | 
 n_tup_upd       | bigint   |                     | 
 n_tup_del       | bigint   |                     | 
 n_tup_hot_upd   | bigint   |                     | 
```

Таким образом, разработчик может заглянуть внутрь транзакции еще до ее фиксации и узнать, не является ли она причиной проблем с производительностью. Это позволяет отделить данные в целом от того, что делает конкретное приложение.

Идеальный способ использования этого представления – добавить вызов функции в приложение до фиксации транзакции, чтобы посмотреть, что в ней сделано. Эти данные затем можно проинспектировать отдельно от общей рабочей нагрузки.

Мониторинг очистки

В версии PostgreSQL 9.6 сообщество добавило системное представление, которого давно ждали. В течение многих лет пользователи хотели иметь возможность следить за ходом процесса очистки, чтобы понять, долго ли осталось.

Для решения этой проблемы предназначено представление `pg_stat_progress_vacuum`:

```
test=# \d pg_stat_progress_vacuum
        Представление "pg_catalog.pg_stat_progress_vacuum"
    Столбец      | Тип      | Правило сортировки | Допустимость NULL
-----+-----+-----+-----
 pid            | integer  |                     | 
 datid          | oid      |                     | 
 datname        | name     |                     | 
 relid          | oid      |                     | 
 phase          | text     |                     | 
 heap_blks_total | bigint   |                     | 
```

heap_blks_scanned	bigint		
heap_blks_vacuumed	bigint		
index_vacuum_count	bigint		
max_dead_tuples	bigint		
num_dead_tuples	bigint		

Большая часть не нуждается в объяснении, поэтому я опущу детали. Есть лишь две вещи, о которых следует помнить. Во-первых, процесс очистки нелинейный – он то ускоряется, то замедляется. Во-вторых, очистка обычно происходит быстро, поэтому проследить за ее ходом не всегда удастся.

Использование *pg_stat_statements*

Вот мы и подошли к одному из самых важных представлений, которое позволяет выявить проблему с производительностью. Конечно, я говорю о *pg_stat_statements*. Оно служит для получения информации о запросах в системе и помогает выделить типы медленных запросов и узнать, как часто запросы выполняются.

Чтобы воспользоваться этим модулем, нужно выполнить следующие действия.

1. Добавить *pg_stat_statements* в список разделяемых библиотек (параметр *shared_preload_libraries*) в файле *postgresql.conf*.
2. Перезапустить сервер базы данных.
3. Выполнить команду *CREATE EXTENSION pg_stat_statements* в одной или нескольких базах данных.

Рассмотрим определение этого представления:

```
test=# \d pg_stat_statements
```

Представление "public.pg_stat_statements"			
Столбец	Тип	Правило сортировки	Допустимость NULL
userid	oid		
dbid	oid		
queryid	bigint		
query	text		
calls	bigint		
total_time	double precision		
min_time	double precision		
max_time	double precision		
mean_time	double precision		
stddev_time	double precision		
rows	bigint		
shared_blks_hit	bigint		
shared_blks_read	bigint		
shared_blks_dirtied	bigint		
shared_blks_written	bigint		
local_blks_hit	bigint		
local_blks_read	bigint		
local_blks_dirtied	bigint		
local_blks_written	bigint		

temp_blks_read	bigint			
temp_blks_written	bigint			
blk_read_time	double precision			
blk_write_time	double precision			

Представление `pg_stat_statements` дает просто фантастическую информацию. Для каждого пользователя в каждой базе данных оно содержит по одной строке на запрос. По умолчанию отслеживается 5000 запросов (это можно изменить, задав параметр `pg_stat_statements.max`).

Запросы отделены от параметров. PostgreSQL подставляет в запрос маркеры. Это позволяет агрегировать запросы, отличающиеся только параметрами. Так, запрос `SELECT ... FROM x WHERE y = 10` будет преобразован в `SELECT ... FROM x WHERE y = ?`.

Для каждого запроса PostgreSQL сообщает полное время выполнения, а также число вызовов. В недавних версиях добавлены поля `min_time`, `max_time`, `mean_time` и `stddev_time`. Особенно полезно стандартное отклонение, потому что оно показывает, является ли время выполнения запроса стабильным или флуктуирует. Нестабильность может возникать по разным причинам:

- если данные не полностью кешированы в памяти, то серверу приходится обращаться к диску, и это займет гораздо больше времени, чем для запросов, работающих с данными в памяти;
- план выполнения и результирующий набор могут зависеть от параметров;
- влияние могут оказывать конкурентное выполнение и блокировки.

PostgreSQL также сообщает о кешировании в процессе выполнения запроса. Поля `shared_blks_` говорят, сколько блоков прочитано из кеша (`_hit`), а сколько из операционной системы (`_read`). Если из операционной системы читается много блоков, то время выполнения запроса может флуктуировать.

Следующая группа полей относится к буферам. Локальные буферы – это блоки в памяти, выделенные конкретному подключению к базе данных.

Помимо всего прочего, PostgreSQL предоставляет информацию о вводе-выводе во временные файлы. Отметим, что временные файлы естественно создаются при построении большого индекса или выполнении другой DDL-команды, требующей создания большого объекта. Но в OLTP-системе временные файлы – обычно зло, т. к. замедляют работу всей системы вследствие потенциальной конкуренции за диск. Большой объем ввода-вывода во временные файлы может указывать на нежелательные вещи, из которых я отмечу три, на мой взгляд, самые главные:

- неправильное задание параметра `work_mem` (OLTP);
- неоптимальная настройка `maintenance_work_mem` (DDL-команды);
- запросы, которые вообще не следовало бы выполнять.

Наконец, два поля содержат информацию о хронометраже ввода-вывода. По умолчанию они пусты, поскольку измерение времени в некоторых системах может приводить к существенным накладным расходам. Поэтому параметр `track_io_timing` по умолчанию равен `false`, и, если эти данные вам нужны, не забудьте включить его.

Как только модуль активирован, PostgreSQL начинает собирать данные, и вы можете пользоваться этим представлением.

❑ Никогда не запускайте запрос `SELECT * FROM pg_stat_statements` в присутствии заказчика. Сколь раз я видел, как люди начинали тыкать пальцами в запросы и объяснять, кто, когда, зачем и почему выполнил тот или иной запрос. Если используете это представление, то всегда сортируйте вывод, чтобы относящаяся к делу информация была сразу видна.

Мы в компании Cybertec считаем, что следующий запрос очень полезен, когда нужно составить представление о происходящем на сервере:

```
test=# SELECT round((100 * total_time / sum(total_time) OVER ()):numeric, 2) percent,
              round(total_time::numeric, 2) AS total,
              calls,
              round(mean_time::numeric, 2) AS mean,
              substring(query, 1, 40)
```

```
FROM pg_stat_statements
ORDER BY total_time DESC
LIMIT 10;
```

percent	total	calls	mean	substring
54.47	111289.11	122161	0.91	UPDATE pgbench_branches SET bbalance = b
43.01	87879.25	122161	0.72	UPDATE pgbench_tellers SET tbalance = tb
1.46	2981.06	122161	0.02	UPDATE pgbench_accounts SET abalance = a
0.50	1019.83	122161	0.01	SELECT abalance FROM pgbench_accounts WH
0.42	856.22	122161	0.01	INSERT INTO pgbench_history (tid, bid, a
0.04	85.63	1	85.63	copy pgbench_accounts from stdin
0.02	44.11	1	44.11	vacuum analyze pgbench_accounts
0.02	42.86	122161	0.00	END;
0.02	34.08	122171	0.00	BEGIN;
0.01	22.46	1	22.46	alter table pgbench_accounts add primary

(10 строк)

Он показывает 10 запросов с наибольшим суммарным временем выполнения, вывода и долю времени в процентах. Кроме того, имеет смысл показать среднее время выполнения запроса, чтобы было понятно, действительно ли оно очень велико.

Пройдитесь по списку и найдите все запросы, среднее время работы которых кажется слишком большим.

Имейте в виду, что просматривать 1000 запросов обычно бессмысленно. В большинстве случаев причиной высокой нагрузки на систему являются первые несколько запросов.

❑ В примере выше я взял только первые 40 символов запроса, чтобы выдача уместилась на странице. Не надо так делать, если вы действительно хотите узнать, что происходит.

Напомню, что по умолчанию `pg_stat_statements` оставляет только 1024 байта запроса:

```
test=# SHOW track_activity_query_size;
track_activity_query_size
```

1024
(1 строка)

Подумайте о том, чтобы увеличить значение этого параметра, скажем, до 16 384. Если клиентами являются Java-приложения, написанные с применением Hibernate, то увеличение `track_activity_query_size` гарантирует, что запрос не будет обрезан раньше, чем начнется интересная часть.

Я хочу воспользоваться случаем и объяснить, насколько в действительности важно представление `pg_stat_statements`. Это самый простой способ выловить проблемы с производительностью. Журнал медленных запросов и рядом не стоял с `pg_stat_statements`, потому что он показывает только отдельные медленные запросы; из него мы не узнаем о проблемах, вызванных многочисленными запросами средней длительности. Поэтому рекомендую всегда включать этот модуль. Накладные расходы малы и никак не отразятся на общей производительности системы.

По умолчанию отслеживается 5000 типов запросов. Если приложения не слишком экзотические, то этого достаточно.

Чтобы сбросить данные, выполните команду:

```
test=# SELECT pg_stat_statements_reset();
pg_stat_statements_reset
-----
(1 строка)
```

Создание файлов журналов

Познакомившись с системными представлениями, которые предлагает PostgreSQL, мы можем перейти к конфигурированию протоколирования. По счастью, PostgreSQL предоставляет простой способ работы с журналами и помогает их правильно настроить.

Ведение журналов важно, поскольку дает информацию об ошибках и потенциальных проблемах в базе. Все необходимые параметры задаются в файле `postgresql.conf`.

Конфигурационный файл `postgresql.conf`

В этом разделе мы рассмотрим наиболее важные разделы файла `postgresql.conf`, относящиеся к протоколированию, и увидим, как использовать протоколирование во благо.

Но сначала я хочу сказать несколько слов о протоколировании в PostgreSQL вообще. При работе в системе Unix PostgreSQL по умолчанию пишет все, что должно попасть в журнал, в поток `stderr`. Но `stderr` – неподходящее место для журналов, поскольку вы обязательно захотите посмотреть содержимое журнала в какой-то момент времени. Поэтому рекомендую прочитать эту главу и настроить все под свои нужды.

Определение местонахождения журналов и порядка ротации

Посмотрим, что можно задать в файле `postgresql.conf`¹:

```
#-----
# СООБЩЕНИЯ ОБ ОШИБКАХ И ПРОТОКОЛИРОВАНИЕ
#-----

# - Куда записывать журнал -

log_destination = 'stderr'      # Допустимые значения - комбинации
                                # stderr, csvlog, syslog и eventlog
                                # в зависимости от платформы. Для csvlog
                                # необходимо включить logging_collector.

# Это используется при протоколировании на stderr:
logging_collector = off         # Разрешить перенаправление stderr
                                # и csvlog в файлы журналов. Должен быть равен
                                # on для csvlog.
                                # (После изменения необходима перезагрузка.)
```

Первый конфигурационный параметр определяет, как обрабатывать журнал. По умолчанию все записывается в `stderr` (в Unix). В Windows местом назначения по умолчанию является журнал событий `eventlog`. Можно также указать `csvlog` или `syslog`.

Если вы хотите, чтобы PostgreSQL писала журнал в файлы, то оставьте `stderr` и включите сборщик сообщений (`logging collector`). Тогда PostgreSQL будет создавать файлы журналов.

Возникает вопрос: как будут называться эти файлы, и где они будут храниться? В `postgresql.conf` на него есть ответ:

```
# Используются, только если logging_collector равен on:
log_directory = 'pg_log'
                                # в какой каталог записывать файлы журналов,
                                # путь может быть абсолютным или относительно PGDATA
log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log'
                                # шаблон имени файла журнала
                                # может включать спецификаторы, которые понимает strftime()
```

Параметр `log_directory` сообщает системе, где хранить журналы. Указывая абсолютный путь, вы задаете каталог явно. Если вы предпочитаете, чтобы журналы хранились вместе с данными PostgreSQL, укажите относительный путь. Преимущество такого решения – в том, что каталог данных становится автономным и может перемещаться как единое целое.

На следующем шаге мы определяем, как должны выглядеть имена журналов. PostgreSQL позволяет использовать все спецификаторы, которые понимает библиотечная функция `strftime`. Чтобы вы лучше понимали, насколько гибок этот механизм, я подсчитал, что на моей платформе `strftime` поддержи-

¹ В реальном файле тексты сообщений написаны по-английски. Здесь они переведены для удобства читателей. – *Прим. перев.*

вает 43 (!) спецификатора для задания имени файла. Возможно все, что обычно применяется.

Определившись с именами файлов, стоит задуматься об очистке. Для этого предлагаются такие параметры:

```
#log_truncate_on_rotation = off
#log_rotation_age = 1d
#log_rotation_size = 10MB
```

По умолчанию PostgreSQL пишет в один и тот же файл, пока он не окажется старше 1 дня или его размер не превысит 10 МБ. Дополнительный параметр `log_truncate_on_rotation` позволяет задать, нужно ли дописывать в конец существующего журнала или нет. Иногда задается циклическая схема именования журналов. Параметр `log_truncate_on_rotation` говорит, что делать, если файл уже существует: перезаписывать или дописывать в конец. Но если выбрана схема именования по умолчанию, то такого точно не произойдет.

Один из способов настроить авторотацию – использовать шаблон вида `postgresql_%a.log` в сочетании с параметром `log_truncate_on_rotation = on`. Спецификатор `%a` означает, что в имени файла будет фигурировать день недели. Тогда имя будет повторяться через каждые семь дней, т. е. файл будет храниться неделю, а затем перезаписываться. В этом случае размера файла 10 МБ может оказаться недостаточно. Быть может, стоит вообще отменить ограничение на размер.

Настройка syslog

Некоторые предпочитают писать журналы в `syslog`. PostgreSQL предлагает для этого следующие конфигурационные параметры:

```
# Используются при записи журналов в syslog:
#syslog_facility = 'LOCAL0'
#syslog_ident = 'postgres'
#syslog_sequence_numbers = on
#syslog_split_messages = on
```

Запись в `syslog` пользуется популярностью у системных администраторов. Настроить ее просто. Нужно лишь задать службу (facility) и идентификатор.

Протоколирование медленных запросов

Журнал можно использовать также для мониторинга отдельных медленных запросов. В старые добрые времена это был едва ли не единственный способ определять причины проблем с производительностью.

Как это работает? В файле `postgresql.conf` имеется параметр `log_min_duration_statement`. Если его значение больше нуля, то все запросы, выполнявшиеся дольше, записываются в журнал:

```
# log_min_duration_statement = -1
```

Многие смотрят на журнал медленных запросов как на источник высшей мудрости. Но я бы хотел предостеречь против этого. Есть много медленных за-

просов, пожирающих ресурсы процессора: построение индексов, экспорт данных, аналитические запросы и т. д.

Такие запросы не являются неожиданными, и во многих случаях вовсе не они – корень зла. Часто виновники – более короткие запросы, которых, однако, очень много. Например, 1000 запросов по 500 мс хуже двух запросов по 5 с. Поэтому во многих случаях журнал медленных запросов только сбивает с толку.

И все же он не совсем бесполезен, просто надо относиться к нему как к одному из источников информации, не единственному.

Что и как протоколировать

Итак, с базовыми настройками мы познакомились, теперь настало время решить, что протоколировать. По умолчанию протоколируются только ошибки. Но этого может оказаться недостаточно. В этом разделе мы узнаем, что можно протоколировать и как будет выглядеть строка журнала.

По умолчанию PostgreSQL не пишет в журнал информацию о контрольных точках. Следующий параметр позволит изменить эту ситуацию:

```
#log_checkpoints = off
```

То же самое относится к подключениям; PostgreSQL может протоколировать все факты установления и разрыва соединения:

```
#log_connections = off
#log_disconnections = off
```

В большинстве случаев протоколировать подключения не имеет смысла, потому что это замедляет работу системы. К аналитическим системам это относится в меньшей мере, но OLTP-системы могут серьезно пострадать.

Если вы хотите знать, сколько времени выполнялись команды, то можете включить следующий параметр:

```
#log_duration = off
```

Перейдем к одному из самых важных параметров. Пока что мы еще не задали формат сообщений, так что информация об ошибке будет иметь такой вид:

```
test=# SELECT 1 / 0;
ОШИБКА: деление на ноль
```

В журнал попадут слово ОШИБКА и сообщение об ошибке. До версии PostgreSQL 10.0 не было ни временной метки, ни имени пользователя – вообще ничего. Чтобы в журналах был хоть какой-то смысл, формат нужно было сразу же изменить. В PostgreSQL 10.0 значение по умолчанию стало более разумным, оно определяется параметром `log_line_prefix`:

```
#log_line_prefix = '%m [%p] '
# специальные значения:
# %a = имя приложения
# %u = имя пользователя
# %d = имя базы данных
```

```
# %g = удаленный сервер и порт
# %h = удаленный сервер
# %p = идентификатор процесса
# %t = временная метка без миллисекунд
# %m = временная метка с миллисекундами
# %n = временная метка с миллисекундами (как в Unix)
# %i = тег команды
# %e = состояние SQL
# %c = идентификатор сеанса
# %l = номер строки сеанса
# %s = временная метка начала сеанса
# %v = идентификатор виртуальной транзакции
# %x = идентификатор транзакции (0, если отсутствует)
# %q = ничего не выводит. Непользовательские процессы останавливаются в этой точке
# %% = '%'
```

Параметр `log_line_prefix` позволяет настроить формат строки под конкретные потребности. В общем случае лучше включать временную метку, иначе будет почти невозможно узнать, когда произошла неприятность. Лично я предпочитаю также включать имя пользователя, идентификатор транзакции и имя базы данных. Впрочем, вам решать, что именно вам нужно.

Иногда медлительность вызвана неправильным управлением блокировкой. Когда пользователи блокируют друг друга, ожидать высокой производительности не приходится, такие проблемы необходимо решать. Вообще говоря, выявлять проблемы, связанные с блокировкой, трудно. Помочь в этом может параметр `log_lock_waits`. Если блокировка удерживается дольше, чем величина `deadlock_timeout`, то в журнал записывается сообщение, при условии что этот параметр включен:

```
#log_lock_waits = off
```

Наконец, нужно сказать PostgreSQL, что протоколировать. До сих пор речь шла о медленных запросах, ошибках и т. п. Но параметр `log_statement` может принимать четыре значения:

```
#log_statement = 'none'
# none, ddl, mod, all
```

Значение `none` означает, что протоколируются только ошибки, `ddl` – что протоколируются также DDL-команды (`CREATE TABLE`, `ALTER TABLE` и т. д.), `mod` – что включаются также команды изменения данных, а `all` – что в журнал записываются все команды.



Имейте в виду, что в режиме `all` может выводиться очень много информации, замедляя работу системы. Чтобы дать представление о масштабе проблемы, я написал в своем блоге статью, с которой можно ознакомиться по адресу <https://www.cybertec-postgresql.com/en/logging-the-hidden-speedbrakes/>.

Если вы хотите получать более детальную информацию о репликации, то включите следующий параметр:

```
#log_replication_commands = off
```

Подробнее см. раздел документации <https://www.postgresql.org/docs/current/static/protocol-replication.html>¹.

Довольно часто снижение производительности вызвано вводом-выводом во временные файлы. Чтобы понять, какие запросы вызывают проблемы, можно задать следующие параметры:

```
#log_temp_files = -1
# Протоколировать временные файлы, размер которых
# в килобайтах больше или равен заданной величине.
# -1 отключить, 0 протоколировать все временные файлы
```

Если `pg_stat_statements` содержит агрегированную информацию, то `log_temp_files` разрешает протоколирование конкретных запросов, вызывающих проблемы. Обычно имеет смысл задавать не слишком большое значение. Какое именно, зависит от рабочей нагрузки, но начать вполне можно с 4 МБ.

По умолчанию PostgreSQL записывает журналы в поясное время сервера. Но если система распределена по всему миру, то будет разумно скорректировать часовой пояс, так чтобы можно было сравнивать записи в журналах:

```
log_timezone = 'Europe/Vienna'
```

Имейте в виду, что SQL-команды по-прежнему будут видеть локальное поясное время. Но если этот параметр задан, то временные метки в записях журналов будут относиться к другому часовому поясу.

РЕЗЮМЕ

Эта глава была посвящена статистике системы. Мы научились получать информацию от PostgreSQL и узнали, как использовать статистику с пользой. Были детально рассмотрены наиболее важные представления.

В главе 6 мы будем говорить об оптимизации запросов. Мы узнаем, как анализировать запросы и добиваться их оптимальности.

Вопросы

Как и в других главах, ответим на ключевые вопросы, связанные с только что изложенным материалом.

Какого рода статистику собирает PostgreSQL во время работы?

Разнообразную. Полный обзор можно найти в официальной документации по адресу <https://www.postgresql.org/docs/11/static/monitoring-stats.html>².

¹ На русском языке <https://postgrespro.ru/docs/postgresql/11/protocol-replication>. – Прим. перев.

² На русском языке <https://postgrespro.ru/docs/postgresql/11/monitoring-stats>. – Прим. перев.

Как легко выявить проблемы, связанные с производительностью?

Есть разные способы. Один из них – воспользоваться представлением `pg_stat_statements`. Другой – прибегнуть к `auto_explain` или просто к стандартным журналам PostgreSQL. Какой метод лучше, зависит от ваших потребностей. В статье по адресу <https://www.cybertec-postgresql.com/en/3-ways-to-detect-slowqueries-in-postgresql/> описано, как получить такого рода информацию от системы.

Как PostgreSQL записывает файлы журналов?

В PostgreSQL есть несколько способов создания журналов. Самый распространенный – обычные текстовые файлы. Но можно также использовать `syslog` или журнал событий в Windows (`eventlog`).

Оказывает ли протоколирование влияние на производительность?

В общем, да. Если сервер выполняет много небольших запросов и записывает миллионы строк в журнал, то производительность может пострадать.

Глава 6

Оптимизация запросов для достижения максимальной производительности

В главе 5 мы узнали о том, как получить статистику системы и воспользоваться ей. Вооружившись этими знаниями, мы поговорим о том, как добиться высокой производительности запросов. Будут рассмотрены следующие вопросы:

- внутреннее устройство оптимизатора;
- планы выполнения;
- секционирование данных;
- включение и выключение параметров оптимизатора;
- параметры, обеспечивающие высокую производительность;
- распараллеливание запросов;
- JIT-компиляция.

Прочитав эту главу, вы сможете писать более качественные и быстрые запросы. А если запрос недостаточно хорош, то вы сможете понять, в чем дело. Вы также сможете применить вновь полученные знания к секционированию данных.

Что делает оптимизатор

О производительности не стоит даже думать, не поняв, чем занимается оптимизатор запросов. Углубиться во внутреннее устройство имеет смысл хотя бы затем, чтобы уяснить, для чего предназначена база данных и как она достигает своих целей.

Оптимизация на примере

Чтобы продемонстрировать, как работает оптимизатор, я подготовил пример. Он верой и правдой служил мне многие годы для обучения PostgreSQL. Пусть имеются следующие три таблицы:

```
CREATE TABLE a (aid int, ...); -- 100 миллионов строк
CREATE TABLE b (bid int, ...); -- 200 миллионов строк
CREATE TABLE c (cid int, ...); -- 300 миллионов строк
```

И предположим, что они содержат миллионы, а то и сотни миллионов строк. Над ними построены индексы:

```
CREATE INDEX idx_a ON a (aid);
CREATE INDEX idx_b ON b (bid);
CREATE INDEX idx_c ON c (cid);
```

И наконец, имеется представление, соединяющее две таблицы:

```
CREATE VIEW v AS SELECT *
FROM a, b
WHERE aid = bid;
```

Теперь предположим, что пользователь хочет выполнить такой запрос:

```
SELECT *
FROM v, c
WHERE v.aid = c.cid AND cid = 4;
```

Как поступит оптимизатор? Какие варианты есть у планировщика?

Оценка вариантов соединения

У планировщика есть несколько вариантов, поэтому разберемся, что плохого может случиться, если выбрать тривиальный подход.

Предположим, что планировщик решает очертя голову броситься вперед и вычислить результат представления. Как лучше всего соединить 100 и 200 млн строк?

В этом разделе мы рассмотрим некоторые (не все) варианты соединения и покажем, что умеет делать PostgreSQL.

Вложенные циклы

Вложенные циклы – один из способов соединения двух таблиц. Принцип простой и описывается следующим псевдокодом:

```
for x in table1:
    for y in table2:
        if x.field == y.field
            вывести строку
        else
            продолжить
```

Вложенные циклы часто применяются, когда одна из соединяемых таблиц очень мала. В нашем примере пришлось бы выполнить 100 млн × 200 млн ите-

раций. Очевидно, что это никуда не годится, потому что время работы оказалось бы гигантским.

В общем случае временная сложность вложенного цикла равна $O(n^2)$, так что рассчитывать на эффективность можно, только если одна из таблиц очень мала. Поскольку в данном примере это не так, рассматривать этот вариант вычисления соединения не имеет смысла.

Соединение хешированием

Второй вариант – соединение хешированием. Для решения нашей мелкой проблемы можно применить следующую стратегию:

Соединение хешированием

Последовательный просмотр таблицы 1

Последовательный просмотр таблицы 2

Обе таблицы можно хешировать, после чего сравнить ключи и получить результат соединения. Проблема в том, что хеш-коды всех значений нужно где-то хранить.

Соединение слиянием

Наконец, существует соединение слиянием. Идея в том, чтобы использовать отсортированные списки. Если обе таблицы отсортированы, то система может сравнивать строки, последовательно продвигаясь от начала списков к концу, и возвращать те, для которых ключи совпали. Ниже приведен план:

Соединение слиянием

Отсортировать таблицу 1

Последовательный просмотр таблицы 1

Отсортировать таблицу 2

Последовательный просмотр таблицы 2

Чтобы соединить две таблицы, данные нужно представить в отсортированном виде. В большинстве случаев PostgreSQL просто сортирует данные. Но есть и другие способы соединения отсортированных данных, например воспользоваться индексом:

Соединение слиянием

Просмотр таблицы 1 по индексу

Просмотр таблицы 2 по индексу

На одной или на обеих сторонах соединения можно воспользоваться тем фактом, что данные были отсортированы на более низких уровнях плана выполнения запроса. Если производится доступ к базовой таблице, то очевидный способ – взять построенный над ней индекс, но только если выборка существенно меньше всей таблицы. В противном случае мы увеличим накладные расходы чуть ли не в два раза, т. к. должны будем сначала прочитать весь индекс, а потом всю таблицу. Если выборка составляет значительную часть таблицы, то последовательный просмотр эффективнее, особенно если доступ производится в порядке первичного ключа.

Плюс соединения слиянием в том, что так можно обработать большие объемы данных. А минус – в том, что данные должны быть в какой-то момент отсортированы или взяты из индекса.

Временная сложность сортировки составляет $O(n * \log(n))$. Поэтому перспектива сортировки 300 млн строк не вызывает энтузиазма.

i В версии PostgreSQL 10.0 все вышеперечисленные способы соединения доступны и в параллельных вариантах. Поэтому оптимизатор рассматривает не только стандартные способы соединения, но и решает, имеет ли смысл распараллеливать выполнение запроса.

Применение преобразований

Понятно, что очевидное решение (сначала вычислить представление) бесперспективно. Время выполнения вложенных циклов невообразимо велико. Для соединения хешированием нужно хешировать 300 млн строк, а для соединения слиянием – отсортировать эти миллионы строк. Все три варианта нас не устраивают. Выход – логически преобразовать запрос, сделав его быстрее. В этом разделе мы узнаем, как планировщик может ускорить запрос. Выполняется два шага.

Встраивание представления

Сначала оптимизатор встраивает представления. Вот что происходит:

```
SELECT *
FROM
(
  SELECT *
  FROM a, b
  WHERE aid = bid
) AS v, c
WHERE v.aid = c.cid AND cid = 4;
```

Представление встроено и преобразовано в подзапрос. И что это нам дает? Само по себе ничего. Но открывает дверь для дальнейшей оптимизации, которая в данном случае станет решающей.

Расправление подзапроса

Следующий шаг – «расправить» (flatten) подзапрос, т. е. включить его в главный запрос. Благодаря избавлению от подзапросов открываются дополнительные возможности оптимизации.

Вот как выглядит наш запрос после расправления подзапроса:

```
SELECT * FROM a, b, c WHERE a.aid = c.cid AND aid = bid AND cid = 4;
```

Это обычное соединение.

i Мы могли бы переписать этот SQL-код самостоятельно, но планировщик в любом случае выполняет такие преобразования. И теперь можно произвести дальнейшую оптимизацию.

Применение ограничений в виде равенства

Далее создаются ограничения в виде равенства. Идея в том, чтобы выявить дополнительные ограничения, условия соединения и фильтры. Сделаем глубокий вдох и рассмотрим следующее рассуждение: если $aid = cid$ и $aid = bid$, то $bid = cid$. Если $cid = 4$ и остальные равенства справедливы, то aid и bid тоже должны быть равны 4, что приводит к следующему запросу:

```
SELECT *
FROM a, b, c
WHERE a.aid = c.cid
      AND aid = bid
      AND cid = 4
      AND bid = cid
      AND aid = 4
      AND bid = 4
```

Важность этой оптимизации невозможно переоценить. Планировщик дал возможность использовать два дополнительных индекса – в исходном запросе такая возможность не просматривалась.

Возможность использовать индексы, построенные по всем трем столбцам, делает запрос гораздо дешевле. Оптимизатор теперь может выбрать две строки по индексу и использовать любой способ соединения.

Исчерпывающий поиск

Проделав эти формальные преобразования, PostgreSQL производит исчерпывающий поиск. Она пробует все возможные планы выполнения запроса и оставляет самый дешевый. PostgreSQL знает, какие есть индексы, и просто применяет стоимостную модель для нахождения оптимального плана.

В процессе исчерпывающего поиска PostgreSQL пытается также определить наилучший порядок соединения. В исходном запросе порядок был фиксирован: $A \rightarrow B$ и $A \rightarrow C$. Но благодаря ограничениям в виде равенства мы можем сначала соединить B с C , а затем присоединить к результату A . Планировщику открыты все возможности.

И что в итоге

Обсудив все оптимизации, мы можем посмотреть, какой план выбрала PostgreSQL:

```
tttest=# EXPLAIN SELECT * FROM v, c WHERE v.aid = c.cid AND cid = 4;
               QUERY PLAN
-----
Nested Loop (cost=1.71..17.78 rows=1 width=12)
-> Nested Loop (cost=1.14..9.18 rows=1 width=8)
    -> Index Only Scan using idx_a on a (cost=0.57..4.58 rows=1 width=4)
        Index Cond: (aid = 4)
    -> Index Only Scan using idx_b on b (cost=0.57..4.59 rows=1 width=4)
        Index Cond: (bid = 4)
-> Index Only Scan using idx_c on c (cost=0.57..8.59 rows=1 width=4)
```

Index Cond: (cid = 4)
(8 строк)

i Заметим, что планы, показанные в этой главе, необязательно в точности совпадают с тем, что вы видите. Возможны небольшие вариации в зависимости от того, сколько было загружено данных. Стоимость может также зависеть от физического расположения данных на диске (порядка следования). Имейте это в виду, когда будете прогонять примеры.

Как видим, PostgreSQL использует три индекса. Интересно также, что PostgreSQL решила соединять данные методом вложенных индексов. Это разумно, т. к. после просмотра по индексу данных почти и не осталось. Поэтому вложенные циклы будут вести себя очень эффективно.

Когда все идет намарку

Мы видели, что для нас может сделать PostgreSQL и как оптимизатор помогает ускорить выполнение запроса. PostgreSQL – создание умное, но предпочитает иметь дело с умными пользователями. Бывает так, что пользователь своими глупостями сводит на нет весь процесс оптимизации. Давайте удалим представление:

```
test=# DROP VIEW v;  
DROP VIEW
```

И создадим его заново. Обратите внимание на добавленную в конец фразу `OFFSET 0`:

```
test=# CREATE VIEW v AS SELECT *  
FROM a, b  
WHERE aid = bid  
OFFSET 0;  
CREATE VIEW
```

Логически это представление эквивалентно рассмотренному выше, но оптимизатор вынужден рассматривать его по-другому. Любое смещение, кроме 0, изменит результат, поэтому представление необходимо вычислять. Это сбивает весь процесс оптимизации.

✓ Сообщество PostgreSQL не стало оптимизировать случай наличия `OFFSET 0` в представлении. Просто считается, что нормальные люди так не делают. Мы взяли это в качестве примера, чтобы показать, как некоторые операции способны погубить производительность, и напомнить, что разработчикам следует понимать скрытый за кулисами процесс оптимизации.

Вот как выглядит новый план:

```
test=# EXPLAIN SELECT * FROM v, c WHERE v.aid = c.cid AND cid = 4;  
QUERY PLAN
```

```
-----  
Nested Loop (cost=120.71..7949879.40 rows=1 width=12)  
-> Subquery Scan on v  
    (cost=120.13..7949874.80 rows=1 width=8)
```

```

Filter: (v.aid = 4)
-> Merge Join (cost=120.13..6699874.80 rows=100000000 width=8)
    Merge Cond: (a.aid = b.bid)
    -> Index Only Scan using idx_a on a
        (cost=0.57..2596776.57 rows=100000000 width=4)
    -> Index Only Scan using idx_b on b
        (cost=0.57..5193532.33 rows=199999984 width=4)
-> Index Only Scan using idx_c on c
    (cost=0.57..4.59 rows=1 width=4)
    Index Cond: (cid = 4)
(9 строк)

```

Взгляните на предсказанные планировщиком стоимости. Если раньше они выражались двузначными числами, то теперь взлетели до небес. Очевидно, что производительность такого запроса никуда не годится¹.

Сворачивание констант

PostgreSQL умеет выполнять за кулисами много других оптимизаций, повышающих производительность. Одна из них – **сворачивание констант**, т. е. вычисление константных выражений во время компиляции:

```

test=# explain SELECT * FROM a WHERE aid = 3 + 1;
               QUERY PLAN
-----
Index Only Scan using idx_a on a
    (cost=0.57..4.58 rows=1 width=4)
    Index Cond: (aid = 4)
(2 строки)

```

Как видим, PostgreSQL ищет 4. Поскольку столбец `aid` проиндексирован, PostgreSQL производит просмотр по индексу. А поскольку в таблице всего один столбец, то PostgreSQL даже поняла, что просмотра одного лишь индекса достаточно для получения данных.

Что произойдет, если выражение находится в левой части?

```

test=# explain SELECT * FROM a WHERE aid - 1 = 3;
               QUERY PLAN
-----
Seq Scan on a (cost=0.00..1942478.48 rows=500000 width=4)
    Filter: ((aid - 1) = 3)
(2 строки)

```

В этом случае PostgreSQL не сможет подобрать подходящий индекс и вынуждена выбрать последовательный просмотр. Обратите внимание, что это план для одного процессорного ядра. Если таблица велика или PostgreSQL сконфигурирована по-другому, то может быть выбран план для нескольких ядер. Для простоты мы в этой главе рассматриваем только одноядерные планы.

¹ Для данного запроса это довольно очевидно, но в общем случае оценка стоимости ничего не говорит о реальной производительности. – *Прим. ред.*

Встраивание функций

Еще одна оптимизация, помогающая ускорить выполнение запросов, – **встраивание функций**. PostgreSQL умеет встраивать неизменяемые SQL-функции. Идея в том, чтобы уменьшить количество вызовов функций и тем ускорить работу.

Ниже приведен пример функции, допускающей встраивание:

```
test=# CREATE OR REPLACE FUNCTION ld(int)
      RETURNS numeric AS
$$
    SELECT log(2, $1);
$$
LANGUAGE 'sql' IMMUTABLE;
CREATE FUNCTION
```

Она вычисляет двоичный логарифм числа:

```
test=# SELECT ld(1024);
      ld
-----
 10.0000000000000000
(1 строка)
```

Для демонстрации заново создадим таблицу, уменьшив число строк, чтобы индекс строился быстрее:

```
test=# TRUNCATE a;
TRUNCATE TABLE
```

Добавим данные и построим индекс:

```
test=# INSERT INTO a SELECT * FROM generate_series(1, 10000);
INSERT 0 10000
test=# CREATE INDEX idx_ld ON a (ld(aid));
CREATE INDEX
```

Как и следовало ожидать, индекс, построенный по значению функции, работает, как любой другой индекс. Однако присмотримся внимательнее к условию поиска по индексу:

```
test=# EXPLAIN SELECT * FROM a WHERE ld(aid) = 10;
      QUERY PLAN
-----
Index Scan using idx_ld on a (cost=0.29..8.30 rows=1 width=4)
  Index Cond: (log('2'::numeric, (aid)::numeric) = '10'::numeric)
(2 строки)
```

Важное наблюдение состоит в том, что мы ищем в индексе функцию `log`, а не `ld`. Оптимизатор полностью избавился от вызова функции.

Логично, что тем самым открывается дверь для следующего запроса:

```
test=# EXPLAIN SELECT * FROM a WHERE log(2, aid) = 10;
      QUERY PLAN
```

```
-----
Index Scan using idx_ld on a (cost=0.29..8.30 rows=1 width=4)
  Index Cond: (log('2'::numeric, (aid)::numeric) = '10'::numeric)
(2 строки)
```

Устранение соединений

PostgreSQL предлагает также еще одну оптимизацию: **устранение соединений**. Идея в том, чтобы удалить соединение, если оно не нужно в запросе. Это бывает полезно, когда запросы генерируются каким-то ПО промежуточного уровня или системой объектно-реляционного отображения (ORM). Если соединение можно устранить, то работа, естественно, ускоряется, а накладные расходы снижаются.

Но как работает устранение соединений? Рассмотрим пример:

```
CREATE TABLE x (id int, PRIMARY KEY (id));
CREATE TABLE y (id int, PRIMARY KEY (id));
```

Сначала создадим две таблицы. Важно, чтобы по обе стороны условия соединения находились уникальные значения.

Теперь напомним простой запрос:

```
test=# EXPLAIN SELECT *
FROM x LEFT JOIN y ON (x.id = y.id)
WHERE x.id = 3;

                                QUERY PLAN
```

```
-----
Nested Loop Left Join (cost=0.31..16.36 rows=1 width=8)
  Join Filter: (x.id = y.id)
  -> Index Only Scan using x_pkey on x
      (cost=0.15..8.17 rows=1 width=4)
      Index Cond: (id = 3)
  -> Index Only Scan using y_pkey on y
      (cost=0.15..8.17 rows=1 width=4)
      Index Cond: (id = 3)
(6 строк)
```

Как видим, PostgreSQL соединяет эти таблицы непосредственно. Пока никаких сюрпризов. Но давайте немного модифицируем этот запрос: будем выбирать не все столбцы, а только столбцы из левой таблицы:

```
test=# EXPLAIN SELECT x.*
FROM x LEFT JOIN y ON (x.id = y.id)
WHERE x.id = 3;

                                QUERY PLAN
```

```
-----
Index Only Scan using x_pkey on x (cost=0.15..8.17 rows=1 width=4)
  Index Cond: (id = 3)
(2 строки)
```

PostgreSQL выбирает просмотр первой таблицы по индексу, а соединение вообще игнорирует. Это возможно и логически правильно по двум причинам:

- из правой таблицы столбцы не выбираются, поэтому их чтение нам ничего не даст;
- значения столбца в правой части соединения уникальны, поэтому операция соединения не может увеличить количество строк из-за наличия дубликатов.

Если соединение можно автоматически устранить, то выполнение запроса может ускориться на порядок. Прелесть в том, что для этого достаточно просто не выбирать столбцы, которые приложению все равно не нужны.

Ускорение операций над множествами

Операции над множествами позволяют комбинировать результаты нескольких запросов в один результирующий набор. К ним относятся UNION, INTERSECT и EXCEPT. PostgreSQL реализует их все и предлагает ряд важных оптимизаций.

Планировщик умеет переносить ограничения внутрь операций над множествами, что открывает дверь различным хитроумным оптимизациям. Для демонстрации рассмотрим следующий запрос:

```
test=# EXPLAIN SELECT *
FROM
(
    SELECT aid AS xid FROM a
    UNION ALL
    SELECT bid FROM b
) AS y
WHERE xid = 3;
```

QUERY PLAN

```
-----
Append (cost=0.29..12.89 rows=2 width=4)
-> Index Only Scan using idx_a on a
    (cost=0.29..8.30 rows=1 width=4)
    Index Cond: (aid = 3)
-> Index Only Scan using idx_b on b
    (cost=0.57..4.59 rows=1 width=4)
    Index Cond: (bid = 3)
(5 строк)
```

Здесь мы просто объединяем два отношения. Проблема в том, что единственное ограничение находится вне подзапроса. Однако PostgreSQL понимает, что фильтр можно перенести внутрь плана. Поэтому условие `xid = 3` налагается на `aid` и `bid`, что позволяет использовать индексы над обеими таблицами. Избавившись от последовательного просмотра обеих таблиц, мы сможем выполнить запрос гораздо быстрее.

Обратим внимание на различие между фразами UNION и UNION ALL. Фраза UNION слепо добавляет данные и возвращает результаты из обеих таблиц. А фраза UNION дополнительно устраняет дубликаты. Из следующего плана видно, как это происходит:

```
test=# EXPLAIN SELECT *
FROM
(
  SELECT aid AS xid FROM a
  UNION
  SELECT bid FROM b
) AS y
WHERE xid = 3;
```

QUERY PLAN

```
-----
Unique (cost=12.92..12.93 rows=2 width=4)
  -> Sort (cost=12.92..12.93 rows=2 width=4)
      Sort Key: a.aid
      -> Append (cost=0.29..12.91 rows=2 width=4)
          -> Index Only Scan using idx_a on a
              (cost=0.29..8.30 rows=1 width=4)
              Index Cond: (aid = 3)
          -> Index Only Scan using idx_b on b
              (cost=0.57..4.59 rows=1 width=4)
              Index Cond: (bid = 3)
```

(8 строк)

PostgreSQL была вынуждена добавить узел Sort над Append, чтобы впоследствии отфильтровать дубликаты.



Многие разработчики не подозревают о различии между фразами UNION и UNION ALL. Они жалуются на низкую производительность, не зная, что PostgreSQL отфильтровывает дубликаты, что особенно заметно в случае больших наборов данных.

РАЗБИРАЕМСЯ В ПЛАНАХ ВЫПОЛНЕНИЯ

Узнав о некоторых важных оптимизациях, применяемых в PostgreSQL, мы можем ближе познакомиться с планами выполнения. В этой книге мы уже не раз встречались с планами, но, чтобы задействовать их в полной мере, необходимо систематически подойти к вопросу их интерпретации. Именно этим мы сейчас и займемся.

Систематический подход к планам выполнения

Прежде всего необходимо понимать, что команда EXPLAIN может многое рассказать о плане, поэтому я настоятельно рекомендую использовать ее на всю катушку.

Многие читатели, наверное, знают, что команда EXPLAIN ANALYZE выполняет запрос и возвращает план, включая информацию, полученную во время выполнения. Например:

```
test=# EXPLAIN ANALYZE SELECT *
FROM
(
  SELECT * FROM b
```



```

LIMIT 1000000
) AS b
ORDER BY cos(bid);

```

QUERY PLAN

```

-----
Sort (cost=146173.12..148673.12 rows=1000000)
  (actual time=837.049..1031.587 rows=1000000)
  Sort Key: (cos((b.bid)::double precision))
  Sort Method: external merge Disk: 25408kB
-> Subquery Scan on b
  (cost=0.00..29424.78 rows=1000000 width=12)
  (actual time=0.011..352.717 rows=1000000)
  -> Limit (cost=0.00..14424.78 rows=1000000)
    (actual time=0.008..169.784 rows=1000000)
    -> Seq Scan on b_b_1 (cost=0.00..2884955.84 rows=19999984 width=4)
      (actual time=0.008..85.710 rows=1000000)

Planning time: 0.064 ms
Execution time: 1159.919 ms
(8 строк)

```

Этот план выглядит пугающе, но не надо впадать в панику – мы разберем его шаг за шагом. Читать план следует изнутри наружу. В этом примере выполнение начинается с последовательного просмотра `b`. Здесь мы видим два блока информации: блок **cost** и блок **actual time**. Блок **cost** содержит оценки, а блок **actual time** – факты, т. е. истинное время выполнения. В данном случае последовательный просмотр занял 85.7 мс.



В вашей системе оценка стоимости может быть иной. Разница может быть обусловлена небольшими отличиями в собранной статистике. Сейчас нас интересует в основном, как читать план.

Данные, полученные в результате просмотра индекса, затем передаются узлу `Limit`, который отбрасывает лишние строки. Заметим, что на каждом этапе выполнения показывается количество строк. Как видим, PostgreSQL выбирает из таблицы только 1 млн строк, это гарантирует узел `Limit`. Однако это не бесплатно – время выполнения на этом шаге подскочило до 169 мс. Наконец, данные сортируются, и это занимает много времени. Главный урок состоит в том, что, глядя на план, мы можем понять, на что уходит время. В этом примере последовательный просмотр занимает некоторое время, но существенно уменьшить его мы не можем. Однако настоящий скачок наблюдается на этапе сортировки.

Конечно, процесс сортировки можно ускорить, но этим мы займемся позже.

Как сделать выдачу EXPLAIN подробнее

В PostgreSQL выдачу EXPLAIN можно сделать несколько подробнее. Чтобы извлечь из плана все до капли, попробуем включить параметры, как показано ниже:

```

test=# EXPLAIN (analyze, verbose, costs, timing, buffers)
       SELECT * FROM a ORDER BY random();
              QUERY PLAN

```

```

Sort (cost=834.39..859.39 rows=10000 width=12)
  (actual time=6.089..7.199 rows=10000 loops=1)
  Output: aid, (random())
  Sort Key: (random())
  Sort Method: quicksort Memory: 853kB
  Buffers: shared hit=45
  -> Seq Scan on public.a
    (cost=0.00..170.00 rows=10000 width=12)
    (actual time=0.012..2.625 rows=10000 loops=1)
    Output: aid, random()
    Buffers: shared hit=45
Planning time: 0.054 ms
Execution time: 7.992 ms
(10 строк)

```

Параметр `analyze true` означает, что нужно выполнить запрос, как показано выше; `verbose true` включает в план дополнительную информацию (о столбцах и иную); `costs true` показывает стоимость; `timing true` дает сведения о хронометраже на этапе выполнения, чтобы мы могли видеть, на что расходуется время. Наконец, параметр `buffers true` может дать очень ценную информацию. В примере выше он показывает, что для выполнения запроса пришлось обратиться к 45 буферам.

Выявление проблем

Располагая информацией из главы 5, мы можем указать на две потенциальные проблемы с производительностью, которые очень важны на практике.

Анализ времени выполнения

Изучая план, мы обязательно должны задать себе два вопроса:

- оправдано ли для данного запроса время выполнения, показанное командой `EXPLAIN ANALYZE`?
- если запрос выполняется медленно, то на что именно уходит время?

В нашем случае последовательный просмотр занял 2.625 мс. Сортировка закончилась спустя 7.199 мс, т. е. сама операция заняла примерно 4.5 мс – именно на нее потрачена большая часть времени.

Видя, на что тратится больше всего времени, мы можем понять, что происходит. Что предпринять, зависит от типа самой прожорливой операции. Дать общий совет невозможно, потому что проблемы могут возникать по самым разным причинам.

Анализ оценок

Но есть одна вещь, которую следует делать всегда: проверить, сильно ли расходятся оценка и фактическая величина. В некоторых случаях оптимизатор принимает плохие решения, потому что оценка по какой-то причине имеет мало общего с действительностью. Так бывает, например, когда статистика система устарела. Поэтому начинать надо с выполнения команды `ANALYZE`. Впрочем, за

актуальностью статистики обычно следит фоновый процесс автоочистки, так что следует подумать и о других причинах неправильной оценки. Рассмотрим следующий пример, для которого сначала создадим и заполним таблицу:

```
test=# CREATE TABLE t_estimate AS
SELECT * FROM generate_series(1, 10000) AS id;
SELECT 10000
```

Загрузив в таблицу 10 000 строк, соберем статистику для оптимизатора:

```
test=# ANALYZE t_estimate;
ANALYZE
```

Теперь посмотрим на оценки:

```
test=# EXPLAIN ANALYZE SELECT * FROM t_estimate WHERE cos(id) < 4;
               QUERY PLAN
```

```
-----
Seq Scan on t_estimate (cost=0.00..220.00 rows=3333 width=4)
    (actual time=0.010..4.006 rows=10000 loops=1)
    Filter: (cos((id)::double precision) < '4'::double precision)
    Planning time: 0.064 ms
    Execution time: 4.701 ms
(4 строки)
```

Во многих случаях PostgreSQL не может правильно обработать фразу WHERE, потому что располагает только статистикой по столбцам, но не по выражениям. Поэтому мы наблюдаем досадную недооценку количества строк, возвращенных WHERE.

Разумеется, бывает и так, что оценка количества строк завышена:

```
test=# EXPLAIN ANALYZE
SELECT *
FROM t_estimate
WHERE cos(id) > 4;
               QUERY PLAN
```

```
-----
Seq Scan on t_estimate (cost=0.00..220.00 rows=3333 width=4)
    (actual time=3.802..3.802 rows=0 loops=1)
    Filter: (cos((id)::double precision) > '4'::double precision)
    Rows Removed by Filter: 10000
    Planning time: 0.037 ms
    Execution time: 3.813 ms
(5 строк)
```

Если нечто подобное происходит глубоко внутри плана, то весь план может оказаться непригодным. Поэтому важно проверять, что оценки не слишком сильно расходятся с реальностью.

К счастью, есть способ обойти эту проблему:

```
test=# CREATE INDEX idx_cosine ON t_estimate (cos(id));
CREATE INDEX
```

Создание функционального индекса заставляет PostgreSQL хранить статистику для выражения:

```
test=# ANALYZE t_estimate;
ANALYZE
```

Мало того что это решение обеспечивает куда лучшую производительность, оно еще и исправляет статистику, даже если индекс не используется:

```
test=# EXPLAIN ANALYZE SELECT * FROM t_estimate WHERE cos(id) > 4;
                                QUERY PLAN
-----
Index Scan using idx_cosine on t_estimate
  (cost=0.29..8.30 rows=1 width=4)
  (actual time=0.002..0.002 rows=0 loops=1)
    Index Cond: (cos((id)::double precision) > '4'::double precision)
Planning time: 0.095 ms
Execution time: 0.011 ms
(4 строки)
```

Однако неверные оценки могут быть вызваны и причинами, не бросающимися в глаза. Проблема, которую часто недооценивают, называется **межстолбцовой корреляцией**. Рассмотрим простой пример, в котором участвуют два столбца:

- 20% людей любят кататься на лыжах;
- 20% людей живут в Африке.

Если мы захотим подсчитать количество лыжников в Африке, то, с точки зрения математики, получится $0.2 \times 0.2 = 4\%$ всего населения Земли. Но в Африке нет снега, а доходы населения малы. Поэтому истинный результат будет гораздо меньше. Два наблюдения не являются статистически независимыми. Из-за того, что PostgreSQL хранит статистику только по отдельным столбцам, часто получаются неудовлетворительные результаты.

Конечно, планировщик многое делает, чтобы предотвратить такое развитие событий. Но проблема тем не менее существует.

Начиная с версии PostgreSQL 10.0 мы можем собирать многомерную статистику, что позволяет раз и навсегда положить конец проблеме межстолбцовой корреляции.

Анализ использования буферов

Но сам план – не единственная причина возможных проблем. Зачастую опасности таятся на другом уровне. Память и кеширование могут приводить к нежелательному поведению, которое трудно понять, не имея опыта распознавания проблемы, описанной в этом разделе.

В следующем примере данные вставляются в таблицу в случайном порядке.

```
test=# CREATE TABLE t_random AS
        SELECT * FROM generate_series(1, 10000000) AS id ORDER BY random();
SELECT 10000000
test=# ANALYZE t_random
ANALYZE
```

Мы создали простую таблицу с 10 млн строк и собрали статистику для оптимизатора.

На следующем шаге выполним простой запрос, извлекающий небольшую часть записей:

```
test=# EXPLAIN (analyze true, buffers true, costs true, timing true)
      SELECT * FROM t_random WHERE id < 1000;
               QUERY PLAN
-----
Seq Scan on t_random (cost=0.00..169248.60 rows=1000 width=4)
    (actual time=1.068..685.410 rows=999 loops=1)
    Filter: (id < 1000)
    Rows Removed by Filter: 9999001
    Buffers: shared hit=2112 read=42136
Planning time: 0.035 ms
Execution time: 685.551 ms
(6 строк)
```

Прежде чем анализировать данные, выполните этот запрос дважды. Конечно, в данном случае имеет смысл использовать индекс. Но мы видим, что при выполнении запроса PostgreSQL нашла 2112 буферов в кеше, а 421 136 буферов получила от операционной системы. И тут возможны две вещи. Если нам повезет, то операционная система найдет данные в своем кеше, и запрос выполнится быстро. А если не повезет, то блоки придется читать с диска. Все вроде бы очевидно, но различие во времени выполнения будет огромным. Запрос, который полностью удовлетворяется из кеша, завершается в 100 раз быстрее запроса, вынужденного медленно собирать блоки из разных мест диска.

Попробуем пояснить эту проблему на простом примере. Пусть имеется система управления телефонами, в которой хранится 10 млрд строк (не такое уж необычное дело для крупных операторов). Данные изменяются очень быстро, а пользователи хотят их видеть. Понятно, что 10 млрд строк целиком в память не поместятся, поэтому данные часто будут читаться с диска.

Теперь напомним простой запрос, чтобы показать, как PostgreSQL ищет номер телефона:

```
SELECT * FROM data WHERE phone_number = '+12345678';
```

Даже если вы все время разговариваете по телефону, ваши данные будут разбросаны по всему диску. Если вы заканчиваете один разговор и сразу начинаете другой, то одновременно с вами точно так же поступают еще тысячи людей, поэтому шансы, что два ваших звонка окажутся в одном блоке, исчезающе малы. Просто представьте, что одновременно совершается 100 000 звонков. На диске данные о них будут распределены случайным образом. Если ваш номер телефона встречается часто, значит, для каждой строки придется прочитать с диска целый блок (в предположении, что коэффициент попадания в кеш очень мал). Предположим, что нужно вернуть 5000 строк. Считая, что нам при-

дется 5000 раз обратиться к диску, на это уйдет примерно $5000 \times 5 \text{ мс} = 25 \text{ с}$. Заметим, что время выполнения этого запроса вполне может варьироваться между несколькими секундами и, скажем, 30 с – все зависит от того, сколько данных было кешировано операционной системой или PostgreSQL.

Помните, что после каждого перезапуска сервера кэши PostgreSQL и файловой системы очищаются, так что отказ узла может стать причиной серьезных проблем.

Как исправить проблему частого чтения с диска

Возникает неизбежный вопрос: *как улучшить эту ситуацию?* Один из способов – выполнить команду CLUSTER:

```
test=# \h CLUSTER
Команда: CLUSTER
Описание: перегруппировать таблицу по индексу
Синтаксис:
CLUSTER [VERBOSE] имя_таблицы
[ USING имя_индекса ] CLUSTER [VERBOSE]
```

Команда CLUSTER переписывает таблицу в том порядке, в каком она обходится по индексу (типа btree). В случае рабочей нагрузки аналитического типа это может иметь смысл. Но в OLTP-системе для выполнения команды CLUSTER может не найтись времени, потому что на все время ее работы таблица блокируется.

СОЕДИНЕНИЯ: ОСМЫСЛЕНИЕ И ИСПРАВЛЕНИЕ

Соединения – вещь важная, все мы постоянно ими пользуемся. Поэтому соединения важны для достижения и поддержания высокой производительности. Мы рассмотрим эту тему, чтобы помочь вам конструировать соединения правильно.

Как соединять правильно

Прежде чем переходить к оптимизации, мы должны разобраться с типичными проблемами, сопутствующими соединениям, и понять, что должно вызывать тревогу.

Вот пример простой структуры для демонстрации соединений:

```
test=# CREATE TABLE a (aid int);
CREATE TABLE
test=# CREATE TABLE b (bid int);
CREATE TABLE
test=# INSERT INTO a VALUES (1), (2), (3);
INSERT 0 3
test=# INSERT INTO b VALUES (2), (3), (4);
INSERT 0 3
```

В следующем примере показано простое внешнее соединение:

```
test=# SELECT * FROM a LEFT JOIN b ON (aid = bid);
aid | bid
-----+-----
  1 | 
  2 |  2
  3 |  3
(3 строки)
```

Как видим, PostgreSQL берет все строки из левой таблицы и соответствующие им строки из правой таблицы.

А следующий пример многим может показаться неожиданным:

```
test=# SELECT * FROM a LEFT JOIN b ON (aid = bid AND bid = 2);
aid | bid
-----+-----
  1 | 
  2 |  2
  3 | 
(3 строки)
```

Нет, количество строк не уменьшилось – оно осталось прежним. Большинству людей кажется, что соединение должно содержать только одну строку, но это неправильно и может привести к скрытым ошибкам.

Рассмотрим следующий запрос, в котором выполняется простое соединение:

```
test=# SELECT avg(aid), avg(bid)
FROM a LEFT JOIN b
ON (aid = bid AND bid = 2);
      avg      |      avg
-----+-----
2.0000000000000000 | 2.0000000000000000
(1 строка)
```

Большинство считает, что среднее вычисляется по одной строке. Но, как уже сказано выше, это не так, и запросы, подобные этому, зачастую становятся причиной проблем с производительностью, т. к. по какой-то причине PostgreSQL не осуществляет поиск по индексу в левой таблице. Но сейчас нас интересует не производительность, а семантическая проблема. Раз за разом авторы запросов с внешним соединением не понимают, о чем они в действительности просят PostgreSQL. Поэтому лично я рекомендую сначала поинтересоваться семантической корректностью внешнего соединения, а затем уже разбираться в проблеме с производительностью, о которой сообщил заказчик. Не устаю повторять, насколько важно убедиться в том, что запрос делает именно то, чего от него хотели.

Обработка внешних соединений

Убедившись, что запрос семантически правилен, следует понять, что может сделать оптимизатор для ускорения внешних соединений. Во многих случаях

PostgreSQL может изменить порядок внутренних соединений и тем самым намного ускорить работу. Но в случае внешних соединений это не всегда возможно. Допустимы лишь немногие операции изменения порядка:

- $(A \text{ leftjoin } B \text{ on } (Pab)) \text{ innerjoin } C \text{ on } (Pac) =$
 $(A \text{ innerjoin } C \text{ on } (Pac)) \text{ leftjoin } B \text{ on } (Pab)$

Pac – это предикат, ссылающийся на A и C (очевидно, что в этом случае Pac не может ссылаться на B , иначе преобразование оказывается бессмысленным).

- $(A \text{ leftjoin } B \text{ on } (Pab)) \text{ leftjoin } C \text{ on } (Pac) =$
 $(A \text{ leftjoin } C \text{ on } (Pac)) \text{ leftjoin } B \text{ on } (Pab)$
- $(A \text{ leftjoin } B \text{ on } (Pab)) \text{ leftjoin } C \text{ on } (Pbc) =$
 $(A \text{ leftjoin } (B \text{ leftjoin } C \text{ on } (Pbc)) \text{ on } (Pab))$

Последнее правило имеет место, только если предикат Pbc должен вернуть `false` для всех строк B , содержащих `null` (т. е. Pbc строгий хотя бы для одного столбца B). Если Pbc не строгий, то первая форма может порождать строки, в которых столбец C содержит не `null`, а во второй форме эти записи будут содержать `null`.

Хотя порядок некоторых соединений можно изменить, в типичном запросе это не дает никакого выигрыша:

```
SELECT ...
FROM a LEFT JOIN b ON (aid = bid)
      LEFT JOIN c ON (bid = cid)
      LEFT JOIN d ON (cid = did)
...
```

Что нужно сделать, так это проверить, все ли внешние соединения необходимы. Зачастую пользователи пишут внешние соединения, которые в действительности не нужны. Иногда логика задачи такова, что в них вообще нет необходимости.

Параметр `join_collapse_limit`

В процессе выработки плана PostgreSQL пытается проверить все возможные порядки соединения. Часто это обходится довольно дорого, потому что перестановок много. Естественно, это замедляет процесс планирования.

Параметр `join_collapse_limit` позволяет разработчику обойти эту проблему и более явно указать, как должен обрабатываться запрос. Чтобы показать, в чем его смысл, рассмотрим простой пример:

```
SELECT * FROM tab1, tab2, tab3
WHERE tab1.id = tab2.id
      AND tab2.ref = tab3.id;
SELECT * FROM tab1 CROSS JOIN tab2
CROSS JOIN tab3
WHERE tab1.id = tab2.id
      AND tab2.ref = tab3.id;
SELECT * FROM tab1 JOIN (tab2 JOIN tab3
ON (tab2.ref = tab3.id))
ON (tab1.id = tab2.id);
```


По существу, все три запроса идентичны, и планировщик обрабатывает их одинаково. Первый запрос содержит неявные соединения, последний – только явные. Планировщик упорядочивает соединения, так чтобы добиться наименьшего времени выполнения. Вопрос: сколько явных соединений PostgreSQL будет планировать неявно? Именно это и определяет параметр `join_collapse_limit`. Для обычных запросов значение по умолчанию вполне приемлемо. Но если в запросе очень много соединений, то настройка этого параметра может заметно сократить время планирования, а это, в свою очередь, важно для обеспечения высокой пропускной способности.

Чтобы увидеть, как задание `join_collapse_limit` изменяет план, напишем такой простой запрос:

```
test=# EXPLAIN WITH x AS
(
    SELECT *
    FROM generate_series(1, 1000) AS id
)
SELECT *
FROM x AS a
JOIN x AS b ON (a.id = b.id)
JOIN x AS c ON (b.id = c.id)
JOIN x AS d ON (c.id = d.id)
JOIN x AS e ON (d.id = e.id)
JOIN x AS f ON (e.id = f.id);
```

Попробуйте выполнить его с разными значениями параметра и посмотрите, как изменяется план. К сожалению, план слишком длинный, поэтому я не могу привести его здесь и показать изменения.

ВКЛЮЧЕНИЕ И ВЫКЛЮЧЕНИЕ РЕЖИМОВ ОПТИМИЗАТОРА

Мы подробно обсудили самые важные оптимизации, выполняемые планировщиком. С годами в PostgreSQL было внесено немало усовершенствований. Но все-таки бывает, что планировщик сбивается с пути истинного и пользователю приходится наставлять его.

Для модификации планов PostgreSQL предлагает несколько параметров. Идея в том, чтобы пользователь мог сделать одни типы узлов плана более дорогими, чем другие. Что это означает на практике?

Вот простой план:

```
test=# EXPLAIN SELECT *
FROM generate_series(1, 100) AS a,
     generate_series(1, 100) AS b
WHERE a = b;
               QUERY PLAN
-----
Merge Join (cost=119.66..199.66 rows=5000 width=8)
  Merge Cond: (a.a = b.b)
```

```

-> Sort (cost=59.83..62.33 rows=1000 width=4)
    Sort Key: a.a
    -> Function Scan on generate_series a
        (cost=0.00..10.00 rows=1000 width=4)
-> Sort (cost=59.83..62.33 rows=1000 width=4)
    Sort Key: b.b
    -> Function Scan on generate_series b
        (cost=0.00..10.00 rows=1000 width=4)
(8 строк)

```

План показывает, что PostgreSQL получает данные от функции и сортирует оба результата. После этого производится соединение слиянием.

Но что, если соединение слиянием – не самый быстрый способ выполнить этот запрос? В PostgreSQL нет способа включить комментарии с указаниями планировщику, как в Oracle. Вместо этого мы должны сообщить, что некоторые операции считаются дорогими. Так, команда `SET enable_mergejoin TO off` делает слияние слишком дорогим:

```

test=# SET enable_mergejoin TO off;
SET
test=# EXPLAIN SELECT *
      FROM generate_series(1, 100) AS a,
           generate_series(1, 100) AS b
      WHERE a = b;
               QUERY PLAN
-----
Hash Join (cost=22.50..210.00 rows=5000 width=8)
  Hash Cond: (a.a = b.b)
    -> Function Scan on generate_series a
        (cost=0.00..10.00 rows=1000 width=4)
    -> Hash (cost=10.00..10.00 rows=1000 width=4)
        -> Function Scan on generate_series b
            (cost=0.00..10.00 rows=1000 width=4)
(5 строк)

```

Поскольку слияние слишком дорого, PostgreSQL решила попробовать соединение хешированием. Как видим, стоимость несколько возросла, но план все равно принимается, поскольку слияние объявлено нежелательным.

А что, если и соединение хешированием запретить?

```

test=# SET enable_hashjoin TO off;
SET
test=# EXPLAIN SELECT *
      FROM generate_series(1, 100) AS a,
           generate_series(1, 100) AS b
      WHERE a = b;
               QUERY PLAN
-----
Nested Loop (cost=0.01..22510.01 rows=5000 width=8)
  Join Filter: (a.a = b.b)
    -> Function Scan on generate_series a

```

```
(cost=0.00..10.00 rows=1000 width=4)
-> Function Scan on generate_series b
      (cost=0.00..10.00 rows=1000 width=4)
(4 строки)
```

PostgreSQL снова пытается выбрать еще какой-нибудь вариант и останавливается на вложенных циклах. Стоимость такого плана ошеломительно высока, но у планировщика нет выбора.

Ну а если запретить и вложенные циклы?

```
test=# SET enable_nestloop TO off;
SET
test=# EXPLAIN SELECT *
      FROM generate_series(1, 100) AS a,
           generate_series(1, 100) AS b
      WHERE a = b;
               QUERY PLAN
-----
Nested Loop (cost=10000000000.00..10000022510.00 rows=5000 width=8)
  Join Filter: (a.a = b.b)
    -> Function Scan on generate_series a
          (cost=0.00..10.00 rows=1000 width=4)
    -> Function Scan on generate_series b
          (cost=0.00..10.00 rows=1000 width=4)
(4 строки)
```

PostgreSQL все равно выполняет соединение методом вложенных циклов. Важно, что off не означает полного запрета операции, а лишь делает ее очень дорогой. Это существенно, потому что иначе запрос вообще нельзя было бы выполнить.

Ниже перечислены параметры, влияющие на работу планировщика:

- ☐ enable_bitmapscan = on
- ☐ enable_hashagg = on
- ☐ enable_hashjoin = on
- ☐ enable_indexscan = on
- ☐ enable_indexonlyscan = on
- ☐ enable_material = on
- ☐ enable_mergejoin = on
- ☐ enable_nestloop = on
- ☐ enable_seqscan = on
- ☐ enable_sort = on
- ☐ enable_tidscan = on

Хотя эти параметры, безусловно, полезны, изменять их следует с осторожностью и только для ускорения отдельных запросов, а не глобально. Выключение некоторого режима может очень быстро обернуться против вас и свести производительность на нет. Так что лучше семь раз отмерить, прежде чем один раз отрезать.

Генетическая оптимизация запросов

Результат планирования – ключ к достижению высокой производительности. Как показано в этой главе, планирование – далеко не тривиальный процесс, включающий различные сложные вычисления. Чем больше таблиц упоминается в запросе, тем сложнее планирование, поскольку планировщик должен рассматривать больше вариантов. Понятно, что время планирования возрастает. В какой-то момент планирование начинает занимать так много времени, что классический исчерпывающий поиск становится нецелесообразным. Более того, ошибки в оценках, допущенные во время планирования, в любом случае настолько велики, что теоретически оптимальный план необязательно дает наименьшее время выполнения.

В таких случаях на помощь может прийти **генетическая оптимизация запросов** (genetic query optimization – **GEQO**). Что это такое? Идея заимствована у природы и напоминает естественную эволюцию.

PostgreSQL подходит к этой задаче, как к задаче коммивояжера, и кодирует возможные порядки соединения строками целых чисел. Например, 4-1-3-2 означает: сначала соединить 4 с 1, затем с 3 и затем с 2. Числами представлены отношения.

Вначале генетический оптимизатор генерирует случайный набор планов. Затем эти планы анализируются. Плохие планы отбрасываются, а на основе генетических хороших планов генерируются новые. Теоретически вновь сгенерированные планы могут оказаться лучше. Этот процесс можно повторить столько раз, сколько необходимо. В итоге останется план, который, как ожидается, будет намного лучше случайного. Генетической оптимизацией управляет параметр `geqo`, как показано ниже:

```
test=# SHOW geqo;
      geqo
-----
on
(1 строка)

test=# SET geqo TO off;
SET
```

По умолчанию параметр `geqo` вступает в игру, когда превышен уровень сложности, задаваемый следующим параметром¹:

```
test=# SHOW geqo_threshold ;
      geqo_threshold
-----
12
(1 строка)
```

Если ваши запросы настолько сложны, что достигается такой уровень, то, безусловно, имеет смысл поэкспериментировать с этим параметром и посмотреть, как будут изменяться порождаемые планировщиком планы.

¹ Под «уровнем сложности» здесь понимается количество соединяемых таблиц. – *Прим. ред.*

Но, вообще говоря, я рекомендую воздерживаться от экспериментов с GEQO и сначала попытаться как-то исправить порядок соединений с помощью параметра `join_collapse_limit`. Заметим, что все запросы уникальны, так что стоит набираться опыта, экспериментируя и наблюдая, как ведет себя планировщик при разных условиях.

Если хотите посмотреть на действительно безумное соединение, послушайте презентацию, которую я проводил в Мадриде (<http://de.slideshare.net/hansjurgenschonig/postgresql-joining-1-million-tables>).

СЕКЦИОНИРОВАНИЕ ДАННЫХ

При размере блока 8 КБ PostgreSQL может хранить в одной таблице до 32 ТБ данных. Если откомпилировать PostgreSQL с размером блока 32 КБ, то эта величина возрастет до 128 ТБ. Но с такими большими таблицами неудобно работать, поэтому имеет смысл секционировать их, чтобы упростить, а иногда и немного ускорить обработку. Начиная с версии 10.0, PostgreSQL предлагает улучшенное секционирование, существенно упрощающее задачу определения секций.

В этом разделе мы рассмотрим как старые, так и новые средства, появившиеся в PostgreSQL 11.0.

Создание секций

Сначала рассмотрим устаревший метод секционирования данных. Знакомство с этой техникой важно для более глубокого понимания того, что PostgreSQL делает за кулисами.

Но прежде чем вдаваться в преимущества секционирования, я хочу показать, как создать секции. Все начинается с родительской таблицы, которая создается следующей командой:

```
test=# CREATE TABLE t_data (id serial, t date, payload text);
CREATE TABLE
```

В этом примере родительская таблица содержит три столбца. Столбец `date` будет использоваться для секционирования, но об этом чуть позже.

Имея родительскую таблицу, мы можем создать дочерние:

```
test=# CREATE TABLE t_data_2016 () INHERITS (t_data);
CREATE TABLE
test=# \d t_data_2016
```

```

      Таблица "public.t_data_2016"
      Модификаторы
-----+-----+-----
Столбец | Тип   | not null default nextval('t_data_id_seq'::regclass)
-----+-----+-----
id       | integer |
t        | date    |
payload  | text    |
Наследует: t_data
```

Таблица называется `t_data_2016` и наследует `t_data`. Никаких дополнительных столбцов в дочернюю таблицу мы добавлять не стали. Как видим, наследование означает, что столбцы родительской таблицы присутствуют в дочерней. Отметим также, что столбец `id` наследует от родительской таблицы последовательность, так что все потомки пользуются общей нумерацией.

Создадим еще несколько таблиц:

```
test=# CREATE TABLE t_data_2015 () INHERITS (t_data);
CREATE TABLE
test=# CREATE TABLE t_data_2014 () INHERITS (t_data);
CREATE TABLE
```

Пока что все таблицы одинаковы и просто наследуют родительской. Но вообще-то дочерние таблицы могут иметь больше столбцов, чем родительская. Добавить столбцы просто:

```
test=# CREATE TABLE t_data_2013 (special text) INHERITS (t_data);
CREATE TABLE
```

В данном случае добавлен столбец `special`. На родительскую таблицу он не влияет, а лишь обогащает дочернюю, позволяя хранить в ней больше данных.

Создав несколько таблиц, добавим, наконец, строку:

```
test=# INSERT INTO t_data_2015 (t, payload)
VALUES ('2015-05-04', 'some data');
INSERT 0 1
```

Самое важное здесь то, что с помощью родительской таблицы мы можем найти все данные в дочерних:

```
test=# SELECT * FROM t_data;
 id |      t      | payload
-----+-----+-----
  1 | 2015-05-04 | some data
(1 строка)
```

Запрос к родительской таблице позволяет просто и эффективно получить доступ к данным, вставленным во все дочерние.

Чтобы понять, как в PostgreSQL обрабатывается секционирование, взглянем на план выполнения запроса:

```
test=# EXPLAIN SELECT * FROM t_data;
          QUERY PLAN
-----
Append (cost=0.00..84.10 rows=4411 width=40)
-> Seq Scan on t_data (cost=0.00..0.00 rows=1 width=40)
-> Seq Scan on t_data_2016 (cost=0.00..22.00 rows=1200 width=40)
-> Seq Scan on t_data_2015 (cost=0.00..22.00 rows=1200 width=40)
-> Seq Scan on t_data_2014 (cost=0.00..22.00 rows=1200 width=40)
-> Seq Scan on t_data_2013 (cost=0.00..18.10 rows=810 width=40)
(6 строк)
```

На самом деле процесс очень простой. PostgreSQL объединяет все таблицы и показывает содержимое всех таблиц в данной секции и во всех секциях ниже нее. Отметим, что все таблицы независимы и лишь логически связаны посредством системного каталога.

Применение табличных ограничений

А что, если применить к таблице фильтры? Какой способ выполнения запроса оптимизатор сочтет наиболее эффективным? В следующем примере показано, как ведет себя планировщик PostgreSQL:

```
test=# EXPLAIN SELECT * FROM t_data WHERE t = '2016-01-04';
               QUERY PLAN
-----
Append (cost=0.00..95.12 rows=23 width=40)
  -> Seq Scan on t_data (cost=0.00..0.00 rows=1 width=40)
      Filter: (t = '2016-01-04'::date)
  -> Seq Scan on t_data_2016 (cost=0.00..25.00 rows=6 width=40)
      Filter: (t = '2016-01-04'::date)
  -> Seq Scan on t_data_2015 (cost=0.00..25.00 rows=6 width=40)
      Filter: (t = '2016-01-04'::date)
  -> Seq Scan on t_data_2014 (cost=0.00..25.00 rows=6 width=40)
      Filter: (t = '2016-01-04'::date)
  -> Seq Scan on t_data_2013 (cost=0.00..20.12 rows=4 width=40)
      Filter: (t = '2016-01-04'::date)
(11 строк)
```

PostgreSQL применяет фильтр ко всем секциям. Он не знает, что имя таблицы как-то связано с ее содержимым. С точки зрения базы данных, имена – это всего лишь имена, они не имеют никакого отношения к тому, что мы ищем. Это, конечно, правильно, потому что для иного поведения нет никакого математического обоснования.

Тогда возникает вопрос: как сказать базе данных, что в таблице `t_data_2016` хранятся только данные за 2016 год, в таблице `t_data_2015` – только данные за 2015 год и т. д.? Для этого предназначены табличные ограничения. Они сообщают PostgreSQL о содержимом таблицы и, стало быть, позволяют планировщику принимать более разумные решения. Эта возможность называется исключением по ограничению и во многих случаях значительно ускоряет выполнение запросов.

Ниже показано, как создаются табличные ограничения:

```
test=# ALTER TABLE t_data_2013 ADD CHECK (t < '2014-01-01');
ALTER TABLE
test=# ALTER TABLE t_data_2014 ADD CHECK (t >= '2014-01-01' AND t < '2015-01-01');
ALTER TABLE
test=# ALTER TABLE t_data_2015 ADD CHECK (t >= '2015-01-01' AND t < '2016-01-01');
ALTER TABLE
test=# ALTER TABLE t_data_2016 ADD CHECK (t >= '2016-01-01' AND t < '2017-01-01');
ALTER TABLE
```

Ограничение CHECK можно добавить к любой таблице.



PostgreSQL создает ограничение, только если каждая строка в таблице удовлетворяет этому ограничению. В отличие от MySQL, PostgreSQL относится к ограничениям серьезно и требует их выполнения во всех случаях.

В PostgreSQL ограничения на разные таблицы могут перекрываться – это не запрещено и в некоторых случаях имеет смысл. Но лучше, когда ограничения не перекрываются, поскольку тогда PostgreSQL может исключить из перебора больше таблиц.

Вот что происходит после добавления табличных ограничений:

```
test=# EXPLAIN SELECT * FROM t_data WHERE t = '2016-01-04';
               QUERY PLAN
-----
Append (cost=0.00..25.00 rows=7 width=40)
-> Seq Scan on t_data (cost=0.00..0.00 rows=1 width=40)
    Filter: (t = '2016-01-04'::date)
-> Seq Scan on t_data_2016 (cost=0.00..25.00 rows=6 width=40)
    Filter: (t = '2016-01-04'::date)
(5 строк)
```

Планировщик смог исключить из запроса многие таблицы, оставив только те, которые могут содержать данные. От более короткого и эффективного плана запрос, конечно, сильно выигрывает, особенно если исключенные таблицы велики.

Модификация наследуемой структуры

Время от времени структуры данных приходится модифицировать. Для этого служит команда ALTER TABLE. Вопрос в том, как модифицировать секционированные таблицы.

Достаточно добавить или удалить столбцы из родительской таблицы, и PostgreSQL автоматически распространит изменения на все дочерние:

```
test=# ALTER TABLE t_data ADD COLUMN x int;
ALTER TABLE
test=# \d t_data_2016
```

Столбец	Тип	Таблица "public.t_data_2016"	Модификаторы
id	integer	not null default	nextval('t_data_id_seq'::regclass)
t	date		
payload	text		
x	integer		

```
Ограничения-проверки:
    "t_data_2016_t_check"
    CHECK (t >= '2016-01-01'::date AND t < '2017-01-01'::date)
Наследует: t_data
```

Как видим, после добавления столбца в родительскую таблицу он автоматически был добавлен и в дочернюю.

Заметим, что это работает для столбцов и связанных с ними объектов. С индексами – совершенно другая история. В структуре наследования каждую таблицу нужно индексировать отдельно. Если построить индекс над родительской таблицей, то только над ней он и будет существовать – на дочерние таблицы он не распространяется. Индексирование тех же столбцов в дочерних таблицах – ваша задача, PostgreSQL не станет принимать за вас решения. Считать это особенностью или ограничением – дело вкуса. Можно сказать, что PostgreSQL позволяет гибко индексировать таблицы по отдельности, что потенциально может повысить эффективность. Но можно и возразить, что строить индексы по одному – лишняя работа.

Перемещение таблицы в наследуемую структуру и из нее

Пусть имеется наследуемая структура. Данные секционированы по дате, и мы хотим показывать пользователю данные за несколько последних лет. В какой-то момент возникает желание сделать часть данных недоступными пользователю, не удаляя их. Например, поместить в какой-то архив.

PostgreSQL предоставляет простые средства сделать это. Сначала создадим нового родителя:

```
test=# CREATE TABLE t_history (LIKE t_data);
CREATE TABLE
```

Ключевое слово LIKE позволяет создать таблицу с точно такой же структурой, как у t_data. Если вы забыли состав столбцов таблицы t_data, то это очень удобно, поскольку позволяет сэкономить время. Можно также добавить индексы, ограничения и значения по умолчанию.

Затем мы отделим таблицу от старого родителя и перенесем под новый:

```
test=# ALTER TABLE t_data_2013 NO INHERIT t_data;
ALTER TABLE
test=# ALTER TABLE t_data_2013 INHERIT t_history;
ALTER TABLE
```

Конечно, все это можно проделать в контексте одной транзакции, чтобы операция была атомарной.

Очистка данных

Одно из преимуществ секционированных таблиц – возможность быстро очистить данные. Предположим, что мы хотим удалить все данные за год. Если данные подходящим образом секционированы, то для этого достаточно простой команды DROP TABLE:

```
test=# DROP TABLE t_data_2014;
DROP TABLE
```

Как видим, удалить дочернюю таблицу легко. А как насчет родительской? Поскольку существуют зависимые от нее объекты, PostgreSQL, естественно, возражает, заботясь о том, чтобы не произошло ничего неожиданного:

```
test=# DROP TABLE t_data;
```

ОШИБКА: удалить объект таблица t_data нельзя, так как от него зависят другие объекты

ПОДРОБНОСТИ: значение по умолчанию, таблица t_data_2013 столбец id зависит от объекта последовательность t_data_id_seq

таблица t_data_2016 зависит от объекта таблица t_data

таблица t_data_2015 зависит от объекта таблица t_data

ПОДСКАЗКА: Для удаления зависимых объектов используйте DROP ... CASCADE.

Команда DROP TABLE предупреждает, что имеются зависимые объекты, и отказывается удалять таблицу. Чтобы все же заставить PostgreSQL удалить родительскую таблицу, нужно добавить слово CASCADE, и тогда будут удалены также дочерние таблицы. Ниже показана каскадная команда DROP TABLE:

```
test=# DROP TABLE t_data CASCADE;
```

ЗАМЕЧАНИЕ: удаление распространяется на ещё 3 объекта

ПОДРОБНОСТИ: удаление распространяется на объект значение по умолчанию, таблица t_data_2013 столбец id

удаление распространяется на объект таблица t_data_2016

удаление распространяется на объект таблица t_data_2015

```
DROP TABLE
```

Секционирование в PostgreSQL 11.0

Многое из добавленного в версиях PostgreSQL 10 и 11 призвано автоматизировать то, что в «старом мире» делалось вручную. Это относится к индексированию, к маршрутизации кортежей при вставке и т. д. Но давайте разберемся по порядку.

Уже долгие годы сообщество PostgreSQL работает над встроенным секционированием. Наконец, в PostgreSQL 10.0 была предложена первая реализация, вошедшая в ядро, которую мы рассмотрим в этой главе. Но в десятой версии секционирование все же требовало ручного труда, поэтому в PostgreSQL 11 многое было улучшено, чтобы упростить использование этой важной функциональности.

Чтобы показать, как работает этот механизм, я подготовил простой пример секционирования по диапазону:

```
CREATE TABLE data (
    payload integer
) PARTITION BY RANGE (payload);
```

```
CREATE TABLE negatives PARTITION OF data FOR VALUES FROM (MINVALUE) TO (0);
CREATE TABLE positives PARTITION OF data FOR VALUES FROM (0) TO (MAXVALUE);
```

Здесь одна секция содержит все отрицательные значения, а другая – все положительные. При создании родительской таблицы можно просто указать, как мы хотим секционировать данные.



В PostgreSQL 10.0 поддерживалось секционирование по диапазону и по списку. В PostgreSQL 11.0 появилось секционирование по хешу.

Имея родительскую таблицу, можно приступить к созданию секций. Для этого добавлена фраза PARTITION OF. В версии PostgreSQL 10 еще оставались ограни-

чения, самое важное из которых – невозможность переместить кортеж (строку) из одной секции в другую, например:

```
UPDATE data SET payload = -10 WHERE id = 5
```

К счастью, в версии PostgreSQL 11 это ограничение снято. Но имейте в виду, что перемещение данных между секциями – не лучшая идея.

Следующий важный момент связан с индексированием: в PostgreSQL 10 каждую таблицу (секцию) нужно было индексировать отдельно. В PostgreSQL 11 это уже не так. Давайте попробуем и посмотрим, что получится:

```
test=# CREATE INDEX idx_payload ON data (payload);
CREATE INDEX
```

```
test=# \d positives
```

```
Таблица "public.positives"
```

```
Столбец | Тип | Правило сортировки | Допустимость NULL | По умолчанию
```

```
-----+-----+-----+-----+-----
payload | integer | | |
```

```
Секция из: data FOR VALUES FROM (0) TO (MAXVALUE)
```

```
Индексы:
```

```
"positives_payload_idx" btree (payload)
```

Как видим, индекс над дочерней таблицей построен автоматически. Это важная особенность PostgreSQL 11, которую уже высоко оценили пользователи, перешедшие на эту версию.

Следующий момент – возможность создать секцию по умолчанию. Чтобы показать, как это работает, удалим одну из секций:

```
test=# DROP TABLE negatives;
DROP TABLE
```

Затем создадим секцию по умолчанию для таблицы data:

```
test=# CREATE TABLE p_def PARTITION OF data DEFAULT;
CREATE TABLE
```

Все данные, для которых не удалось определить подходящую секцию, отправляются в секцию по умолчанию. Это гарантирует, что мы никогда не забудем создать нужную секцию. Опыт доказал, что наличие секции по умолчанию заметно повышает надежность приложений.

Ниже показано, как вставить данные и где эти данные оказываются:

```
test=# INSERT INTO data VALUES (1), (-1);
```

```
INSERT 0 2
```

```
test=# SELECT * FROM data;
```

```
payload
```

```
-----
```

```
1
```

```
-1
```

```
(2 строки)
```

```
test=# SELECT * FROM positives;
```

```

payload
-----
1
(1 строка)

test=# SELECT * FROM p_def;
payload
-----
-1
(1 строка)

```

Как видим, при запросе к родительской таблице возвращаются все данные. А секции содержат части данных в соответствии с правилами секционирования.

НАСТРОЙКА ПАРАМЕТРОВ ДЛЯ ПОВЫШЕНИЯ ПРОИЗВОДИТЕЛЬНОСТИ ЗАПРОСОВ

Написание хороших запросов – первый шаг к достижению высокой производительности. Плохой запрос – плохая производительность. Хорошо написанный код даст максимальный выигрыш. А после того как код оптимизирован с логической и семантической точек зрения, можно подумать о финишных штрихах – ускорении за счет правильной настройки параметров памяти. В этом разделе мы узнаем, как PostgreSQL может распорядиться памятью вам на пользу. Чтобы работало только одно процессорное ядро, выполните следующую команду:

```

test=# SET max_parallel_workers_per_gather TO 0;
SET

```

Вот простой пример, демонстрирующий пользу параметров памяти:

```

test=# CREATE TABLE t_test (id serial, name text);
CREATE TABLE
test=# INSERT INTO t_test (name)
      SELECT 'hans' FROM generate_series(1, 100000);
INSERT 0 100000
test=# INSERT INTO t_test (name)
      SELECT 'paul' FROM generate_series(1, 100000);
INSERT 0 100000

```

Мы сначала вставили в таблицу миллион строк, содержащих имя hans, а затем миллион строк, содержащих имя paul. Всего получилось два миллиона уникальных идентификаторов, но при этом всего два разных имени.

Выполним простой запрос с настройками памяти по умолчанию:

```

test=# SELECT name, count(*) FROM t_test GROUP BY 1;
 name | count
-----+-----
 Hans | 100000

```

```
paul | 100000  
(2 строки)
```

Возвращено две строки, что и неудивительно. Важен не сам результат, а то, что PostgreSQL делает за кулисами:

```
test=# EXPLAIN ANALYZE SELECT name, count(*)  
      FROM t_test  
      GROUP BY 1;  
  
              QUERY PLAN  
-----  
HashAggregate (cost=4082.00..4082.01 rows=1 width=13)  
  (actual time=51.448..51.448 rows=2 loops=1)  
    Group Key: name  
    -> Seq Scan on t_test (cost=0.00..3082.00 rows=200000 width=5)  
        (actual time=0.007..14.150 rows=200000 loops=1)  
Planning time: 0.032 ms  
Execution time: 51.471 ms  
(5 строк)
```

PostgreSQL поняла, что количество групп очень мало. Поэтому она создает хеш, добавляет в него по одной записи на каждую группу и начинает подсчет. Из-за небольшого числа групп хеш совсем мал, и PostgreSQL может быстро завершить подсчет, инкрементируя счетчики для каждой группы.

А что, если группировать по идентификатору, а не по имени? Количество групп резко возрастает:

```
test=# EXPLAIN ANALYZE SELECT id, count(*) FROM t_test GROUP BY 1;  
              QUERY PLAN  
-----  
GroupAggregate (cost=23428.64..26928.64 rows=200000 width=12)  
  (actual time=97.128..154.205 rows=200000 loops=1)  
    Group Key: id  
    -> Sort (cost=23428.64..23928.64 rows=200000 width=4)  
        (actual time=97.120..113.017 rows=200000 loops=1)  
          Sort Key: id  
          Sort Method: external sort Disk: 2736kB  
          -> Seq Scan on t_test (cost=0.00..3082.00 rows=200000 width=4)  
              (actual time=0.017..19.469 rows=200000 loops=1)  
Planning time: 0.128 ms  
Execution time: 160.589 ms  
(8 строк)
```

PostgreSQL обнаруживает, что теперь групп гораздо больше, и тут же меняет стратегию. Проблема в том, что хеш, содержащий так много записей, не поместится в память:

```
test=# SHOW work_mem  
work_mem  
-----  
4MB  
(1 строка)
```

Параметр `work_mem` управляет размером хеша, используемого для обработки фразы `GROUP BY`. Поскольку записей слишком много, PostgreSQL должна выбрать стратегию, которая не требовала бы хранения их всех в памяти. Решение – отсортировать данные по идентификатору, а затем сгруппировать. Отсортировав данные, PostgreSQL сможет пробежаться по списку и сформировать одну группу за другой. Как только текущий идентификатор изменяется, можно вывести частичный результат и переходить к обработке следующей группы.

Чтобы ускорить выполнение запроса, мы можем динамически (или глобально) установить большее значение `work_mem`:

```
test=# SET work_mem TO '1 GB';
SET
```

Теперь планировщик снова выбирает быстрое агрегирование при помощи хеша:

```
test=# EXPLAIN ANALYZE SELECT id, count(*) FROM t_test GROUP BY 1;
               QUERY PLAN
-----
HashAggregate (cost=4082.00..6082.00 rows=200000 width=12)
  (actual time=76.967..118.926 rows=200000 loops=1)
    Group Key: id
    -> Seq Scan on t_test (cost=0.00..3082.00 rows=200000 width=4)
        (actual time=0.008..13.570 rows=200000 loops=1)
  Planning time: 0.073 ms
  Execution time: 126.456 ms
(5 строк)
```

PostgreSQL знает (или, по крайней мере, предполагает), что данные поместятся в память, поэтому переключается на более быстрый план. Как видим, время работы уменьшилось. Запрос выполняется не так быстро, как в случае группировки по имени, поскольку нужно вычислять гораздо больше значений в хеше, но в большинстве случаев мы все равно наблюдаем стабильный выигрыш.

Ускорение сортировки

Параметр `work_mem` ускоряет не только группировку. Он оказывает благотворное влияние и на сортировку – важнейший механизм, который лежит в основе любой СУБД.

Ниже показана простая операция с параметром по умолчанию 4 МБ:

```
test=# SET work_mem TO default;
SET
test=# EXPLAIN ANALYZE SELECT * FROM t_test ORDER BY name, id;
               QUERY PLAN
-----
Sort (cost=24111.14..24611.14 rows=200000 width=9)
  (actual time=219.298..235.008 rows=200000 loops=1)
    Sort Key: name, id
```

```

Sort Method: external sort Disk: 3712kB
-> Seq Scan on t_test (cost=0.00..3082.00 rows=200000 width=9)
    (actual time=0.006..13.807 rows=200000 loops=1)
Planning time: 0.064 ms
Execution time: 241.375 ms
(6 строк)

```

PostgreSQL понадобилось 13.8 мс, чтобы прочитать данные, и более 200 мс, чтобы отсортировать их. Из-за недостаточного объема памяти сортировку пришлось выполнять с помощью временных файлов. Метод внешней сортировки на диске (external sort) потребляет мало оперативной памяти, но должен хранить промежуточные данные на сравнительно медленном запоминающем устройстве, что, конечно, снижает производительность.

Если увеличить `work_mem`, то PostgreSQL сможет использовать больше памяти для сортировки:

```

test=# SET work_mem TO '1 GB';
SET
test=# EXPLAIN ANALYZE SELECT * FROM t_test ORDER BY name, id;
               QUERY PLAN
-----
Sort (cost=20691.64..21191.64 rows=200000 width=9)
    (actual time=36.481..47.899 rows=200000 loops=1)
    Sort Key: name, id
    Sort Method: quicksort Memory: 15520kB
    -> Seq Scan on t_test (cost=0.00..3082.00 rows=200000 width=9)
        (actual time=0.010..14.232 rows=200000 loops=1)
Planning time: 0.037 ms
Execution time: 55.520 ms
(6 строк)

```

Поскольку теперь памяти достаточно, база данных производит всю сортировку в памяти, и процесс стремительно ускоряется. Теперь сортировка занимает всего 33 мс, т. е. в семь раз меньше. Наличие дополнительной памяти ускоряет сортировку и работу всей системы.

До сих пор мы видели два механизма сортировки данных: external sort и quicksort. Но есть еще и третий: top-N heapsort (пирамидальная сортировка в памяти). Он используется, когда нужно получить только первые N строк:

```

test=# EXPLAIN ANALYZE SELECT * FROM t_test ORDER BY name, id LIMIT 10;
               QUERY PLAN
-----
Limit (cost=7403.93..7403.95 rows=10 width=9)
    (actual time=31.837..31.838 rows=10 loops=1)
    -> Sort (cost=7403.93..7903.93 rows=200000 width=9)
        (actual time=31.836..31.837 rows=10 loops=1)
        Sort Key: name, id
        Sort Method: top-N heapsort Memory: 25kB
        -> Seq Scan on t_test (cost=0.00..3082.00 rows=200000 width=9)
            (actual time=0.011..13.645 rows=200000 loops=1)

```

```
Planning time: 0.053 ms
Execution time: 31.856 ms
(7 строк )
```

Алгоритм работает молниеносно, весь запрос выполняется меньше чем за 30 мс. А сортировка занимает всего 18 мс, т. е. выполняется почти так же быстро, как чтение данных.

Отметим, что память размером `work_mem` выделяется на каждую операцию. Теоретически возможно, что при выполнении запроса потребуется выделять столько памяти более одного раза. Поэтому увеличивайте значение параметра с оглядкой.

Следует иметь в виду, что во многих книгах утверждается, что если задать слишком большое значение `work_mem` в OLTP-системе, то у сервера может кончиться память. Да, если 1000 человек одновременно сортируют 100 МБ, то проблемы с памятью могут возникнуть. Но неужели вы думаете, что диск справится с этим? Сомневаюсь. Решение тут одно – пересмотреть свой подход. Такого рода операций вообще не должно быть в OLTP-системе. Подумайте о том, чтобы построить правильные индексы, лучше написать запросы или изменить требования. В любом случае, конкурентная сортировка такого большого объема данных – плохая мысль. Остановитесь, прежде чем остановится ваше приложение.

Ускорение административных задач

Сортировкой и выделением памяти разнообразие операций не исчерпывается. Административные задачи, в частности `CREATE INDEX`, зависят не от параметра `work_mem`, а от параметра `maintenance_work_mem`. Смотрите:

```
test=# SET maintenance_work_mem TO '1 MB';
SET
test=# \timing
Секундомер включен.
test=# CREATE INDEX idx_id ON t_test (id);
CREATE INDEX
Время: 104,268 мс
```

Как видим, построение индекса над таблицей с двумя миллионами строк заняло примерно 100 мс, это очень медленно. Поэтому для ускорения сортировки (а именно этим занимается команда `CREATE INDEX`) увеличим параметр `maintenance_work_mem`:

```
test=# SET maintenance_work_mem TO '1 GB';
SET
test=# CREATE INDEX idx_id2 ON t_test (id);
CREATE INDEX
Время: 46,774 мс
```

Скорость удвоилась, потому что сортировка стала работать быстрее.

Есть и другие административные задачи, которые выигрывают от наличия большего объема памяти. Самые важные из них – команды `VACUUM` (ускоряется

очистка индексов) и ALTER TABLE. Правила задания maintenance_work_mem такие же, как для work_mem. Параметр действует для каждой операции, и динамически выделяется лишь столько памяти, сколько необходимо.

В PostgreSQL 11 в ядро базы данных добавлена новая возможность: теперь сервер может строить индекс типа btree параллельно, что сильно ускоряет индексирование больших таблиц. За настройку распараллеливания отвечает следующий параметр:

```
test=# SHOW max_parallel_maintenance_workers;
max_parallel_maintenance_workers
```

2

(1 строка)

Параметр max_parallel_maintenance_workers задает максимальное число параллельных процессов, выполняющих команду CREATE INDEX. Как и для любой параллельной операции, PostgreSQL решает, сколько исполнителей необходимо, в зависимости от размера таблиц. При индексировании больших таблиц выигрыш может быть очень существенным. В компании Cybertec я провел разностороннее тестирование и подвел итог в своем блоге: <https://www.cybertec-postgresql.com/en/postgresql-parallel-create-index-for-betterperformance/>.

РАСПАРАЛЛЕЛИВАНИЕ ЗАПРОСОВ

Начиная с версии 9.6 PostgreSQL поддерживает параллельное выполнение запросов. Поддержка распараллеливания постепенно совершенствовалась, и в версии 11 функциональность в очередной раз была расширена. В этом разделе мы посмотрим, как работает распараллеливание и что можно сделать для ускорения.

Прежде чем вдаваться в детали, подготовим тестовые данные:

```
test=# CREATE TABLE t_parallel AS
SELECT * FROM generate_series(1, 25000000) AS id;
SELECT 25000000
```

Теперь выполним наш первый параллельный запрос – просто подсчитаем количество строк и посмотрим, как выглядит план:

```
test=# EXPLAIN SELECT count(*) FROM t_parallel;
QUERY PLAN
-----
Finalize Aggregate (cost=258537.40..258537.41 rows=1 width=8)
-> Gather (cost=258537.19..258537.40 rows=2 width=8)
    Workers Planned: 2
    -> Partial Aggregate (cost=257537.19..257537.20 rows=1 width=8)
        -> Parallel Seq Scan on t_parallel (cost=0.00..228153.75 rows=11753375 width=0)
(5 строк)
```

Сначала PostgreSQL выполняет параллельный последовательный просмотр. Это означает, что PostgreSQL использует для обработки таблицы больше одно-

го процессора и создает частичные агрегаты. Задача узла сбора (Gather) – собрать данные от всех процессов и выполнить последний этап агрегирования. Таким образом, узел сбора завершает распараллеливание. Отметим, что распараллеливание (в текущей версии) не бывает вложенным, т. е. мы никогда не встретим один узел сбора внутри другого. В примере выше PostgreSQL запустила два процесса. Почему?

Рассмотрим следующий параметр:

```
test=# SHOW max_parallel_workers_per_gather;
max_parallel_workers_per_gather
```

2

(1 строка)

Параметр `max_parallel_workers_per_gather` ограничивает количество рабочих процессов двумя. Важно, что если таблица мала, то запросы к ней вообще не распараллеливаются. Размер таблицы должен быть не менее 8 МБ, что управляется следующим параметром:

```
test=# SHOW min_parallel_table_scan_size;
min_parallel_table_scan_size
```

8MB

(1 строка)

Правило таково: чтобы PostgreSQL добавила еще один рабочий процесс, размер таблицы должен стать втрое больше. Иными словами, чтобы над таблицей работало четыре дополнительных процесса, ее размер должен быть в 81 раз больше. Это имеет смысл, потому что из-за того, что размер базы увеличился в 100 раз, система хранения не стала в 100 раз быстрее. Поэтому количество ядер, которые можно задействовать с пользой, ограничено.

Однако наша таблица довольно велика:

```
test=# \d+
```

Схема	Имя	Список отношений			
		Тип	Владелец	Размер	Описание
public	t_parallel	table	hs	864 MB	

(1 строка)

В нашем примере `max_parallel_workers_per_gather` ограничивает число ядер. Если изменить этот параметр, то PostgreSQL задействует больше ядер:

```
test=# SET max_parallel_workers_per_gather TO 10;
```

```
SET
```

```
test=# EXPLAIN SELECT count(*) FROM t_parallel;
QUERY PLAN
```

```
-----
Finalize Aggregate (cost=174120.82..174120.83 rows=1 width=8)
-> Gather (cost=174120.30..174120.81 rows=5 width=8)
    Workers Planned: 5
-> Partial Aggregate (cost=173120.30..173120.31 rows=1 width=8)
```

```
-> Parallel Seq Scan on t_parallel (cost=0.00..160620.24 rows=5000024 width=0)
(5 строк)
```

В данном случае запущено всего 5 рабочих процессов (как и ожидалось).

Однако бывает, что мы хотим значительно увеличить количество ядер, обрабатывающих конкретную таблицу. Представьте себе, что база данных занимает 200 ГБ, что имеется 1 ТБ оперативной памяти и всего один пользователь. Этот пользователь мог бы задействовать все имеющиеся процессоры, никому не мешая. Для отмены рассмотренного выше правила утроения можно указать количество параллельных рабочих процессов в команде ALTER TABLE:

```
test=# ALTER TABLE t_parallel SET (parallel_workers = 9);
ALTER TABLE
```

Отметим, что параметр `max_parallel_workers_per_gather` по-прежнему остается в силе и является верхней границей.

Взглянув на план, мы убеждаемся, что заданное количество рабочих процессов учитывается:

```
test=# EXPLAIN SELECT count(*) FROM t_parallel;
               QUERY PLAN
-----
Finalize Aggregate (cost=146343.32..146343.33 rows=1 width=8)
-> Gather (cost=146342.39..146343.30 rows=9 width=8)
    Workers Planned: 9
-> Partial Aggregate (cost=145342.39..145342.40 rows=1 width=8)
-> Parallel Seq Scan on t_parallel (cost=0.00..138397.91 rows=2777791 width=0)
(5 строк)
```

Однако это не значит, что все ядра действительно используются:

```
test=# EXPLAIN analyze SELECT count(*) FROM t_parallel;
               QUERY PLAN
-----
Finalize Aggregate (cost=146343.32..146343.33 rows=1 width=8)
    (actual time=1209.441..1209.441 rows=1 loops=1)
-> Gather (cost=146342.39..146343.30 rows=9 width=8)
    (actual time=1209.432..1209.772 rows=8 loops=1)
    Workers Planned: 9
    Workers Launched: 7
-> Partial Aggregate (cost=145342.39..145342.40 rows=1 width=8)
    (actual time=1200.437..1200.438 rows=1 loops=8)
-> Parallel Seq Scan on t_parallel (cost=0.00..138397.91 ...)
    (actual time=0.038..0.174 rows=3125000 loops=8)
Planning Time: 0.091 ms
Execution Time: 1209.827 ms
(8 строк)
```

Как видим, запущено только семь процессов, хотя было запланировано девять. Почему? Потому что на решение влияют еще два параметра:

```
test=# SHOW max_worker_processes;
max_worker_processes
```

```

-----
                        8
(1 строка)
test=# SHOW max_parallel_workers;
max_parallel_workers
-----
                        8
(1 строка)

```

Первый параметр говорит PostgreSQL, сколько рабочих процессов вообще доступно. А `max_parallel_workers` говорит, сколько их доступно для распараллеливания запросов. Зачем эти два параметра нужны? Фоновые процессы используются не только в инфраструктуре распараллеливания запросов – они нужны и для других целей, поэтому и было решено ввести эти два параметра.

Мы в компании Cybertec (<https://www.cybertec-postgresql.com>) считаем, что в общем случае значение `max_worker_processes` должно быть равно количеству процессорных ядер на сервере. Больше обычно задавать не имеет смысла.

Что PostgreSQL умеет делать параллельно?

Как уже отмечалось, поддержка распараллеливания постепенно улучшалась, начиная с версии PostgreSQL 9.6. В каждой версии добавлялось что-то новое. Теперь параллельно могут выполняться следующие операции:

- параллельный последовательный просмотр;
- параллельный просмотр по индексу (только для индексов типа `btree`);
- параллельный просмотр таблицы по битовой карте;
- параллельное соединение (все типы соединений);
- параллельное индексирование.

В PostgreSQL 11 добавлено параллельное построение индексов. Обычные операции сортировки пока еще распараллелены не полностью, поэтому создавать параллельно можно только индексы типа `btree`. Для управления степенью параллелизма служит следующий параметр:

```

test=# SHOW max_parallel_maintenance_workers;
max_parallel_maintenance_workers
-----
                        2
(1 строка)

```

Правила распараллеливания такие же, как для обычных операций.

Об ускорении построения индекса вы можете прочитать в моем блоге по адресу <https://www.cybertec-postgresql.com/en/postgresql-parallel-create-index-for-better-performance/>.

Распараллеливание на практике

Познакомившись с основами распараллеливания, посмотрим, как это выглядит в реальности. Возьмем следующий запрос:

```
test=# EXPLAIN SELECT * FROM t_parallel;  
               QUERY PLAN  
-----  
Seq Scan on t_parallel (cost=0.00..360621.20 rows=25000120 width=4)  
(1 строка)
```

Почему PostgreSQL не использует распараллеливание? Таблица достаточно велика, рабочие процессы доступны, так в чем же дело? А в том, что межпроцессное взаимодействие обходится очень дорого. Если бы PostgreSQL должна была передавать строки от одного процесса другому, то запрос мог бы выполняться даже дольше, чем в однопроцессном режиме. Оптимизатор пользуется стоимостными параметрами, чтобы штрафовать за межпроцессное взаимодействие.

```
#parallel_tuple_cost = 0.1
```

При каждой передаче кортежа между процессами к стоимости вычислений прибавляется 0.1 балла. Чтобы показать, как PostgreSQL стала бы выполнять запрос параллельно, если бы мы ее к этому принудили, я включил следующий пример:

```
test=# SET force_parallel_mode TO on;  
SET  
test=# EXPLAIN SELECT * FROM t_parallel;  
               QUERY PLAN  
-----  
Gather (cost=1000.00..2861633.20 rows=25000120 width=4)  
  Workers Planned: 1  
  Single Copy: true  
    -> Seq Scan on t_parallel (cost=0.00..360621.20 rows=25000120 width=4)  
(4 строки)
```

Как видим, стоимость оказалась выше, чем в однопроцессном режиме. Это важный практический вопрос, потому что многие пользователи недоумевают, почему PostgreSQL задействует всего одно ядро.

В реальных приложениях важно также понимать, что увеличение числа ядер автоматически не приводит к большему быстродействию. Нахождение оптимального уровня параллелизма – тонкий процесс поиска компромисса.

ВВЕДЕНИЕ В JIT-КОМПИЛЯЦИЮ

JIT-компиляция – одна из самых злободневных тем в PostgreSQL 11. Это было героическое предприятие, и первые результаты выглядят многообещающе. Но начнем с основ: что вообще такое JIT-компиляция? При выполнении запроса PostgreSQL должна многое выяснять во время работы. При компиляции самой программы PostgreSQL не знает, какой запрос вы запустите следующим, поэтому должна быть готова к любому сценарию.

Ядро системы универсально, что означает, что оно может справиться с любой работой. Но ведь мы хотим максимально быстро выполнить конкретный

запрос, а не какой-то обобщенный. Итог: на этапе выполнения системе известно гораздо больше о том, что предстоит сделать, чем на этапе компиляции PostgreSQL. Поэтому если JIT-компиляция включена, PostgreSQL исследует запрос и, если он занимает много времени, динамически (своевременно – Just in Time) создает высоко оптимизированный код.

Настройка JIT

JIT-компиляцию необходимо задавать на этапе компиляции программы PostgreSQL. Для этого служат следующие параметры утилиты `configure`:

```
--with-llvm build with LLVM based JIT support
...
LLVM_CONFIG path to llvm-config command
```

В некоторых дистрибутивах Linux имеется дополнительный пакет, содержащий поддержку JIT. Если вы хотите пользоваться JIT-компиляцией, не забудьте его установить.

Если JIT-компиляция доступна, то в вашем распоряжении имеются следующие параметры для ее настройки:

<code>#jit = on</code>	# разрешить JIT-компиляцию
<code>#jit_provider = 'llvmjit'</code>	# какую реализацию JIT использовать
<code>#jit_above_cost = 100000</code>	# выполнять JIT-компиляцию, если стоимость запроса больше указанной; -1 – выключить
<code>#jit_optimize_above_cost = 500000</code>	# оптимизировать JIT-компилированный код, если стоимость запроса больше указанной; -1 – выключить
<code>#jit_inline_above_cost = 500000</code>	# пытаться встраивать операторы и функции, если стоимость запроса больше указанной; -1 – выключить

Параметр `jit_above_cost` означает, что JIT-компиляция рассматривается, только если ожидаемая стоимость не меньше 100 000. Почему это существенно? Если запрос недостаточно сложный, то накладные расходы на компиляцию могут быть значительно выше потенциального выигрыша. Есть еще два параметра. По-настоящему глубокие оптимизации предпринимаются, только если ожидаемая стоимость запроса не менее 500 000. В этом случае также производится попытка встраивать операторы и функции.

Пока что PostgreSQL поддерживает только реализацию движка JIT-компиляции, написанную с применением LLVM. В будущем, возможно, появятся и другие реализации. Впрочем, LLVM прекрасно справляется и охватывает большинство профессиональных платформ.

Выполнение запросов

Чтобы показать, как работает JIT-компиляция, рассмотрим простой пример. Начнем с создания большой таблицы, поскольку JIT-компиляция полезна, только если операция достаточно сложная. Для начала 50 млн строк должно хватить. Заполним таблицу, как показано ниже:

```
jit=# CREATE TABLE t_jit AS
      SELECT (random()*10000)::int AS x, (random()*100000)::int AS y,
             (random()*1000000)::int AS z
      FROM generate_series(1, 50000000) AS id;
SELECT 50000000
jit=# VACUUM ANALYZE t_jit;
VACUUM
```

Мы воспользовались функцией `random` для генерации случайных данных. Чтобы увидеть, как работает JIT, и упростить чтение планов выполнения, следует выключить распараллеливание запросов. JIT-компиляция работает и для параллельных запросов, но планы выполнения становятся значительно длиннее:

```
jit=# SET max_parallel_workers_per_gather TO 0;
SET
jit=# SET jit TO off;
SET
jit=# EXPLAIN (analyze, verbose) SELECT avg(z+y-pi()), avg(y-pi()), max(x/pi())
FROM t_jit
WHERE ((y+z))>((y-x)*0.000001);
               QUERY PLAN
-----
Aggregate (cost=1936901.68..1936901.69 rows=1 width=24)
  (actual time=20617.425..20617.425 rows=1 loops=1)
    Output: avg((((z + y))::double precision - '3.14159265358979'::double precision)),
             avg(((y)::double precision - '3.14159265358979'::double precision)),
             max(((x)::double precision / '3.14159265358979'::double precision))
    -> Seq Scan on public.t_jit (cost=0.00..1520244.00 rows=16666307 width=12)
        (actual time=0.061..15322.555 rows=50000000 loops=1)
          Output: x, y, z
          Filter: (((t_jit.y + t_jit.z))::numeric > (((t_jit.y - t_jit.x))::numeric *
0.000001))
    Planning Time: 0.078 ms
    Execution Time: 20617.473 ms
(7 строк)
```

В данном случае запрос работал 20 с.

i Я выполнил `VACUUM` с целью установить все биты подсказок и т. п., чтобы сделать честным сравнение между обычным и JIT-компилированным запросом.

Повторим тест, включив JIT-компиляцию:

```
jit=# SET jit TO on;
SET
jit=# explain (analyze, verbose) SELECT avg(z+y-pi()), avg(y-pi()), max(x/pi())
FROM t_jit
WHERE ((y+z))>((y-x)*0.000001);
               QUERY PLAN
-----
Aggregate (cost=1936901.68..1936901.69 rows=1 width=24)
  (actual time=17585.788..17585.789 rows=1 loops=1)
```

```

Output: avg((((z + y)::double precision - '3.14159265358979'::double precision)),
        avg(((y)::double precision - '3.14159265358979'::double precision)),
        max(((x)::double precision / '3.14159265358979'::double precision))
-> Seq Scan on public.t_jit (cost=0.00..1520244.00 rows=16666307 width=12)
    (actual time=81.991..13396.227 rows=50000000 loops=1)
    Output: x, y, z
    Filter: (((t_jit.y + t_jit.z)::numeric > (((t_jit.y - t_jit.x)::numeric *
0.000001)))
Planning Time: 0.135 ms
JIT:
  Functions: 5
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 2.942 ms, Inlining 15.717 ms, Optimization 40.806 ms, Emission 25.233 ms,
          Total 84.698 ms
Execution Time: 17588.851 ms
(11 строк)

```

Теперь запрос выполнен на 10% быстрее, чем раньше, что само по себе значительное достижение. В некоторых случаях выигрыш оказывается еще больше. Но имейте в виду, что на перекомпиляцию кода тратятся дополнительные ресурсы, так что это выгодно не для каждого запроса.

РЕЗЮМЕ

В этой главе мы обсудили ряд оптимизаций выполнения запросов. Мы узнали об оптимизаторе и различных внутренних оптимизациях: сворачивании констант, встраивании представлений, соединениях и т. п. Все они повышают производительность и заметно ускоряют работу.

Следующая глава будет посвящена хранимым процедурам. Мы увидим, что PostgreSQL предлагает для работы с пользовательским кодом.

Глава 7

Написание хранимых процедур

В главе 6 мы многое узнали об оптимизаторе и оптимизациях. А сейчас расскажем о хранимых процедурах и о том, как их эффективно использовать. Вы узнаете, из каких частей состоит хранимая процедура, какие доступны языки программирования и как можно ускорить работу. Дополнительно мы поговорим о некоторых продвинутых средствах языка PL/pgSQL.

В этой главе рассматриваются следующие вопросы:

- правильный выбор языка;
- различие между процедурами и функциями;
- как выполняются хранимые процедуры;
- дополнительные средства PL/pgSQL;
- создание расширений;
- оптимизация для повышения производительности;
- конфигурирование параметров функций.

ЯЗЫКИ ХРАНИМЫХ ПРОЦЕДУР

В части хранимых процедур и функций PostgreSQL существенно отличается от других СУБД, в большинстве из которых для написания серверного кода используется вполне определенный язык программирования. Microsoft SQL Server предлагает Transact-SQL, а Oracle – PL/SQL. PostgreSQL не навязывает язык, а позволяет вам выбирать тот, который вы лучше знаете и больше любите.

Причина такой гибкости PostgreSQL весьма интересна и в историческом плане. Много лет назад один из самых известных разработчиков PostgreSQL, Ян Вика (Jan Wieck), написавший множество изменений, когда система была еще совсем юной, высказал идею использовать TCL в качестве серверного языка программирования. Проблема была в том, что никто не хотел работать с TCL и включать это добро в ядро СУБД. Но решение нашлось – сделать языковой интерфейс настолько гибким, чтобы с PostgreSQL можно было легко интегрировать любой язык. Тогда-то и родилась команда CREATE LANGUAGE. Вот ее синтаксис:

```
test=# \h CREATE LANGUAGE
```

Команда: CREATE LANGUAGE

Описание: создать процедурный язык

Синтаксис:

```
CREATE [ OR REPLACE ] [ PROCEDURAL ] LANGUAGE имя
```

```
CREATE [ OR REPLACE ] [ TRUSTED ] [ PROCEDURAL ] LANGUAGE имя
```

```
    HANDLER обработчик_вызова
```

```
    [ INLINE обработчик_внедренного_кода ] [ VALIDATOR функция_проверки ]
```

В настоящее время функции и хранимые процедуры можно писать на самых разных языках. Гибкость PostgreSQL многократно окупилась, и теперь у нас есть богатый выбор языков программирования.

Как именно PostgreSQL справляется с языками? Взглянув на синтаксис команды CREATE LANGUAGE, мы можем выделить несколько ключевых слов.

- HANDLER: эта функция является прокладкой между PostgreSQL и внешним языком. Она отвечает за отображение внутренних структур данных PostgreSQL на структуры, необходимые языку, и помогает передавать код.
- VALIDATOR: это полицейский. Если он присутствует, то отвечает за извещение пользователей о синтаксических ошибках. Многие языки умеют производить грамматический разбор кода до его выполнения. PostgreSQL может этим воспользоваться и сказать, правильно ли написана функция, еще в момент ее создания. К сожалению, так устроены не все языки, поэтому в некоторых случаях проблемы проявляются уже на этапе выполнения.
- INLINE: если этот обработчик присутствует, то PostgreSQL может выполнять анонимные блоки кода.

Фундаментальные основы – хранимые процедуры и функции

Прежде чем переходить к анатомии хранимой процедуры, поговорим о функциях и процедурах вообще. Говоря о функциях, мы традиционно употребляем термин «хранимая процедура». Поэтому так важно понимать, в чем разница между функцией и процедурой.

Функция – часть обычной команды SQL, ей не разрешено начинать или фиксировать транзакцию. Например:

```
SELECT func(id) FROM large_table;
```

Предположим, что func(id) вызывается 50 млн раз. Что должно было бы произойти, если бы в ней использовалась команда COMMIT? Невозможно в середине запроса завершить транзакцию и начать новую. Это разрушило бы всю концепцию транзакционной целостности, непротиворечивости и т. д.

Напротив, процедура может управлять транзакциями и даже последовательно запускать несколько транзакций. Но ее нельзя вызывать из команды SELECT, а только с помощью команды CALL, синтаксис которой описан ниже:

```
test=# \h CALL
```

Команда: CALL

Описание: вызвать процедуру

Синтаксис:

CALL имя ([аргумент] [, ...])

Таким образом, между функциями и процедурами имеется фундаментальное различие. В интернете не всегда употребляется правильная терминология. Но знать об этом важном различии необходимо. Функции существовали в PostgreSQL с самого начала. Но концепция процедуры, описанная в этом разделе, появилась только в версии PostgreSQL 11. В этой главе мы рассмотрим то и другое.

Анатомия функции

Прежде чем вдаваться в детали конкретного языка, рассмотрим анатомию типичной функции. Для демонстрации возьмем функцию, которая просто складывает два числа:

```
test=# CREATE OR REPLACE FUNCTION mysum(int, int)
      RETURNS int AS
      '
        SELECT $1 + $2;
      ' LANGUAGE 'sql';
CREATE FUNCTION
```

Сразу отметим, что эта функция написана на SQL. PostgreSQL должна знать, какой язык мы используем, поэтому его нужно указывать в определении функции.



Заметим, что код функции передается PostgreSQL в виде строки, заключенной в одиночные кавычки. Это важно, потому что таким образом функция становится черным ящиком для механизма исполнения.

В других СУБД код функции – не строка, а неотъемлемая часть команды. Вот этот простой слой абстрагирования и является основой гибкости диспетчера функций в PostgreSQL.

Внутри строки можно использовать практически все, что предлагает выбранный язык программирования.

В этом примере мы складываем два числа, переданных функции, – два целых числа. Важно отметить, что PostgreSQL предоставляет механизм перегрузки функций. То есть функции `mysum(int, int)` и `mysum(int8, int8)` различны. Перегрузка функций – вещь удобная, но нужно внимательно следить за тем, чтобы случайно не создать слишком много функций из-за того, что список параметров время от времени меняется. Всегда удаляйте функции, ставшие ненужными.



Команда `CREATE OR REPLACE FUNCTION` не модифицирует список параметров. Поэтому ей можно пользоваться, только если сигнатура функции остается неизменной. В противном случае команда либо завершится с ошибкой, либо создаст новую функцию.

Давайте выполним эту функцию:

```
test=# SELECT mysum(10, 20);
mysum
-----
    30
(1 строка)
```

Получилось 30, что и неудивительно. Далее рассмотрим еще одну важную тему: заключение в кавычки.

Долларовые кавычки

Передача кода PostgreSQL в виде строки – очень гибкое решение. Но использование одиночных кавычек может вызывать проблемы. Во многих языках программирования этот символ часто встречается. И чтобы использовать такие кавычки, приходится экранировать их при передаче строки PostgreSQL. Много лет эта процедура была стандартной. Но, к счастью, те времена прошли, и появились новые способы передавать код. Один из них – долларовые кавычки:

```
test=# CREATE OR REPLACE FUNCTION mysum(int, int)
RETURNS int AS
$$
    SELECT $1 + $2;
$$ LANGUAGE 'sql';
CREATE FUNCTION
```

Начало и конец строки обозначаются не одиночной кавычкой, а последовательностью \$\$. В настоящее время есть два языка, в которых \$\$ имеет специальное значение. В Perl и в скриптах на языке bash \$\$ обозначает идентификатор процесса. Чтобы преодолеть это мелкое препятствие, мы можем использовать для отметки начала и конца строки практически любую последовательность символов, начинающуюся с \$, например:

```
test=# CREATE OR REPLACE FUNCTION mysum(int, int) RETURNS int AS
$body$
    SELECT $1 + $2;
$body$ LANGUAGE 'sql';
CREATE FUNCTION
```

Благодаря такой гибкости проблема кавычек решена раз и навсегда. Коль скоро начальная и конечная строки совпадают, проблем не возникнет.

Анонимные блоки кода

Выше мы написали простейшую функцию и научились выполнять код. Но PostgreSQL позволяет выполнять не только полноценные функции, но и анонимные блоки кода, которые нужны только один раз. Это особенно полезно при решении административных задач. Анонимный блок кода не принимает параметров и не хранится в базе данных, поскольку не имеет имени.

Вот простой пример анонимного блока кода:

```
test=# DO
$$
  BEGIN
    RAISE NOTICE 'current time: %', now();
  END;
$$ LANGUAGE 'plpgsql';
NOTICE: current time: 2016-12-12 15:25:50.678922+01
CONTEXT: PL/pgSQL function inline_code_block line 3 at RAISE
DO
```

В этом примере код всего лишь выдает сообщение. Как и раньше, необходимо указывать, на каком языке блок кода написан. Строка, передаваемая PostgreSQL, заключена в долларские кавычки.

Функции и транзакции

Как вы знаете, все, что делает PostgreSQL от имени пользователя, совершается в контексте транзакции. Разумеется, это относится и к функциям. Функция всегда является частью объемлющей транзакции. Она не автономна – так же, как оператор или любая другая операция. Например:

```
test=# SELECT now(), mysum(id, id) FROM generate_series(1, 3) AS id;
           now                | mysum
-----+-----
2017-10-12 15:54:32.287027+01 |      2
2017-10-12 15:54:32.287027+01 |      4
2017-10-12 15:54:32.287027+01 |      6
(3 строки)
```

Все три вызова функции происходят в одной транзакции. Отсюда следует, что внутри функции не может быть команд управления транзакциями. В самом деле, что случилось бы, если бы во втором вызове транзакция была бы зафиксирована? Нет, такое просто не может работать.

Однако в Oracle имеется механизм автономных транзакций. Идея в том, что даже если транзакция откатывается, некоторые ее части, возможно, необходимы и должны быть сохранены. Приведем классический пример.

1. Начать функцию, которая изменяет секретные данные.
2. Добавить в журнал запись о том, что кто-то изменил секретные данные.
3. Зафиксировать эту журнальную запись, но откатить изменение данных.
4. Сохранить информацию о том, что была попытка изменить данные.

Для решения подобных проблем и нужны автономные транзакции, позволяющие выполнять фиксацию независимо от главной транзакции. В данном случае запись в таблице журнала остается, даже если само изменение будет откатоено.

В PostgreSQL 11.0 автономные транзакции отсутствуют. Однако по сети уже гуляют заплатки, в которых эта возможность реализована. Будем ждать, когда она появится в ядре.

Чтобы показать, как это, скорее всего, будет работать, приведем фрагмент кода, основанный на имеющихся изменениях:

```
...
AS
$$
DECLARE
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    FOR i IN 0..9 LOOP
        START TRANSACTION;
        INSERT INTO test1 VALUES (i);
        IF i % 2 = 0 THEN
            COMMIT;
        ELSE
            ROLLBACK;
        END IF;
    END LOOP;
    RETURN 42;
END;
$$;
...
```

Здесь мы на лету решаем, зафиксировать или откатить автономную транзакцию.

ЯЗЫКИ ХРАНИМЫХ ПРОЦЕДУР

Как уже было сказано, PostgreSQL позволяет писать функции и хранимые процедуры на разных языках. В состав ядра включены следующие языки:

- SQL;
- PL/pgSQL;
- PL/Perl и PL/PerlU;
- PL/Python;
- PL/Tcl и PL/TclU.

SQL – очевидный выбор для написания функций, по возможности именно его и следует использовать, поскольку он дает максимальную свободу оптимизатору. Но если нужно написать чуть более сложный код, то стоит прибегнуть к PL/pgSQL.

PL/pgSQL обладает более развитыми средствами управления и многими другими возможностями. В этой главе мы рассмотрим некоторые малоизвестные продвинутые возможности PL/pgSQL, но имейте в виду, что это все же не полное руководство по языку.

В ядре находится код, позволяющий записать серверные функции на языке Perl. Логика такая же, как в остальных случаях: код передается в виде строки и выполняется интерпретатором Perl. Но помните, что сама PostgreSQL не понимает Perl, она лишь передает код внешнему интерпретатору.

Вы, вероятно, обратили внимание, что есть два варианта Perl и TCL: «надежный» (PL/Perl и PL/TCL) и «ненадежный» (PL/PerlU и PL/TCLU). Разница между ними очень существенна. В PostgreSQL язык имеет доступ к самому соединению с базой данных. Поэтому на нем можно написать вредоносный код. Чтобы избежать проблем с безопасностью, предложена концепция надежных языков. В надежное подмножество входят самые базовые средства языка. Запрещается делать следующее:

- включать библиотеки;
- открывать сетевые сокеты;
- выполнять любые системные вызовы, в т. ч. открытие файлов.

Perl предлагает так называемый осторожный режим (taint mode), с помощью которого этот механизм и реализован в PostgreSQL. В этом режиме Perl сам налагает на себя ограничения и выдает ошибку, если возможно нарушение безопасности. В ненадежном режиме можно делать все, поэтому запускать код в таком режиме разрешено только суперпользователю.

Если вы хотите выполнять как надежный, так и ненадежный код, то активируйте оба языка: `plperl` и `plperlU` (соответственно `pltcl` и `pltclU`).

В настоящее время Python имеется только в ненадежной ипостаси, поэтому администратор должен быть особенно осторожен, т. к. функция, написанная на Python, способна обойти все механизмы безопасности, встроенные в PostgreSQL.

Итак, приступим к самой животрепещущей теме этой главы.

Введение в PL/pgSQL

В этом разделе мы познакомимся с некоторыми продвинутыми возможностями языка PL/pgSQL, важными для написания правильного и эффективного кода.



Отметим, что это не введение в программирование для начинающих и не общий учебник по PL/pgSQL.

Экранирование

Один из самых важных вопросов программирования баз данных – экранирование. Неправильно экранируя специальные символы, вы можете стать жертвой внедрения SQL-кода и открыть неприемлемые бреши в системе безопасности.

Что такое внедрение SQL-кода? Рассмотрим пример:

```
CREATE FUNCTION broken(text) RETURNS void AS
$$
DECLARE
    v_sql text;
BEGIN
    v_sql := 'SELECT schemaname
              FROM pg_tables
              WHERE tablename = ''' || $1 || ''';
    RAISE NOTICE 'v_sql: %', v_sql;
RETURN;
```

```
END;
$$ LANGUAGE 'plpgsql';
```

Здесь SQL-код формируется конкатенацией нескольких строк с помощью оператора || без оглядки на безопасность. Если пользователь подставляет правильное имя таблицы, то ничего страшного не произойдет, например:

```
SELECT broken('t_test');
```

Однако надо быть готовым к тому, что кто-то попытается взломать вашу систему. Рассмотрим такой пример:

```
SELECT broken(''; DROP TABLE t_test; ');
```

Теперь мы начинаем понимать, в чем проблема. В листинге ниже показано классическое внедрение SQL-кода:

```
ЗАМЕЧАНИЕ: v_sql: SELECT schemaname FROM pg_tables
WHERE tablename = ''; DROP TABLE t_test; '
КОНТЕКСТ: функция PL/pgSQL broken(text), строка 6, оператор RAISE
broken
-----
(1 строка)
```

Мы хотели все лишь выбрать данные, а в результате удалили таблицу – кому это понравится? Очевидно, что безопасность приложения не должна зависеть от передаваемых параметров.

Чтобы предотвратить внедрение SQL-кода, PostgreSQL предлагает различные функции, их обязательно нужно использовать, чтобы не подвергать приложение опасности:

```
test=# SELECT quote_literal(E'o'reilly'), quote_ident(E'o'reilly');
quote_literal | quote_ident
-----+-----
'o'reilly'    | "o'reilly"
(1 строка)
```

Функция `quote_literal` экранирует строку, так что ничего плохого точно не случится. Она заключает строку в кавычки и экранирует потенциально проблематичные символы внутри строки. Это делает невозможным несанкционированное завершение одной строки и начало другой.

Вторая функция, `quote_ident`, служит для правильного заключения в кавычки имен объектов. Обратите внимание, что используются двойные кавычки – именно так следует поступать с именами таблиц. В примере ниже показано, как используются сложные имена:

```
test=# CREATE TABLE "Some stupid name" ("ID" int);
CREATE TABLE
test=# \d "Some stupid name"
Таблица "public.Some stupid name"
Столбец | Тип | Модификаторы
-----+-----+-----
ID | integer |
```


Обычно имена таблиц в PostgreSQL записываются строчными буквами. Но при использовании двойных кавычек имена объектов могут содержать и заглавные буквы. Вообще говоря, делать так не рекомендуется, потому что придется употреблять двойные кавычки все время, а это не очень удобно.

Теперь посмотрим, как обрабатываются значения NULL. В листинге ниже показано, как обращается с NULL функция `quote_literal`:

```
test=# SELECT quote_literal(NULL);
quote_literal
-----
(1 строка)
```

Если передать `quote_literal` значение NULL, то она просто вернет NULL. В этом случае заботиться об экранировании не надо.

PostgreSQL предлагает и другие функции специально для работы с NULL:

```
test=# SELECT quote_nullable(123), quote_nullable(NULL);
quote_nullable | quote_nullable
-----+-----
'123'          | NULL
(1 строка)
```

PL/pgSQL можно использовать не только для экранирования строк и имен объектов, но и для форматирования и подготовки целых запросов. Прелесть в том, что функция `format` позволяет включать параметры в команду, например:

```
CREATE FUNCTION simple_format() RETURNS text AS
$$
DECLARE
    v_string text;
    v_result text;
BEGIN
    v_string := format('SELECT schemaname|| ' ' . ' || tablename FROM pg_tables
    WHERE %I = $1 AND %I = $2', 'schemaname', 'tablename');
    EXECUTE v_string USING 'public', 't_test' INTO v_result;
    RAISE NOTICE 'result: %', v_result;
    RETURN v_string;
END;
$$ LANGUAGE 'plpgsql';
```

Функции `format` передаются имена полей. А во фразе `USING` команды `EXECUTE` задаются параметры запроса, который затем и выполняется. И снова внедрение SQL-кода успешно предотвращается.

Вот что произойдет при вызове этой простой функции:

```
test=# SELECT simple_format ();
NOTICE: result: public .t_test
simple_format
-----
SELECT schemaname|| ' ' . ' || tablename  +
FROM pg_tables                            +
WHERE schemaname = $1 AND tablename = $2
(1 строка)
```

Как видим, в отладочном сообщении правильно напечатано имя таблицы, включающее схему, и весь запрос в целом.

Управление областями видимости

Разобравшись с экранированием и основами безопасности (внедрением SQL-кода), поговорим еще об одной важной теме – областях видимости.

Как и в большинстве популярных языков программирования, использование переменных в PL/pgSQL зависит от контекста. Переменные определяются в команде DECLARE. Однако PL/pgSQL допускает вложенные команды DECLARE:

```
CREATE FUNCTION scope_test () RETURNS int AS
$$
DECLARE
    i int := 0;
BEGIN
    RAISE NOTICE 'i1: %', i;
    DECLARE
        i int;
        BEGIN
            RAISE NOTICE 'i2: %', i;
        END;
    RETURN i;
END;
$$ LANGUAGE 'plpgsql';
```

В команде DECLARE определена переменная *i*, и ей присвоено значение. Затем переменная *i* отображается. Конечно, будет выведено 0. Затем начинается вторая команда DECLARE. В ней переменная *i* определена еще раз, но значение не присвоено, поэтому оно будет равно NULL. Далее PostgreSQL отображает внутреннюю переменную *i*. Во что получается:

```
test=# SELECT scope_test();
ЗАМЕЧАНИЕ: i1: 0
ЗАМЕЧАНИЕ: i2: <NULL>
scope_test
-----
0
(1 строка)
```

Как и следовало ожидать, печатаются значения 0 и NULL. PostgreSQL допускает использование разнообразных трюков. Но настоятельно рекомендуется делать код простым и понятным читателю.

Обработка ошибок

Обработка ошибок в программе или в модуле – важная составная часть языка программирования. Все когда-то ломается, поэтому ошибки следует обрабатывать надлежащим образом, как подобает профессионалу. В PL/pgSQL для обработки ошибок служат блоки EXCEPTION. Идея в том, что если в блоке BEGIN произойдет что-то плохое, то блок EXCEPTION позаботится об этом и решит проблему. Как и во многих других языках, например в Java, можно по-разному реагировать на различные типы ошибок.

В следующем примере ошибка может возникнуть из-за деления на ноль. Наша задача – перехватить ее и соответственно отреагировать:

```
CREATE FUNCTION error_test1(int, int) RETURNS int AS
$$
BEGIN
    RAISE NOTICE 'отладочное сообщение: % / %', $1, $2;
    BEGIN
        RETURN $1 / $2;
    EXCEPTION
        WHEN division_by_zero THEN
            RAISE NOTICE 'обнаружено деление на ноль: %', sqlerrm;
        WHEN others THEN
            RAISE NOTICE 'какая-то другая ошибка: %', sqlerrm;
    END;
    RAISE NOTICE 'все ошибки обработаны';
    RETURN 0;
END;
$$ LANGUAGE 'plpgsql';
```

Очевидно, что блок BEGIN может возбудить ошибку из-за деления на ноль. Однако в блоке EXCEPTION будет перехвачена как эта ошибка, так и другие потенциальные ошибки.

Технически это напоминает точку сохранения, потому ошибка не приводит к отмене всей транзакции. Мини-откату подвергается только блок, в котором возникла ошибка.

В переменной sqlerrm хранится текст сообщения об ошибке. Выполним код:

```
test=# SELECT error_test1(9, 0);
ЗАМЕЧАНИЕ: отладочное сообщение: 9 / 0
ЗАМЕЧАНИЕ: обнаружено деление на ноль: деление на ноль
ЗАМЕЧАНИЕ: все ошибки обработаны
error_test1
-----
          0
(1 строка)
```

В блоке EXCEPTION PostgreSQL перехватывает исключение и показывает сообщение. Она даже любезно показывает строку, в которой произошла ошибка. Это заметно упрощает отладку и исправление ошибочного кода.

В некоторых случаях имеет смысл возбудить исключение самостоятельно. Это совсем просто:

```
RAISE unique_violation
USING MESSAGE = 'Повторяющийся идентификатор пользователя: ' || user_id;
```

Но вообще-то PostgreSQL предлагает много predefined кодов ошибок. Полный перечень приведен на странице <https://www.postgresql.org/docs/11/static/errcodes-appendix.html>¹.

¹ На русском языке <https://postgrespro.ru/docs/postgresql/11/errcodes-appendix>. – Прим. перев.

Команда *GET DIAGNOSTICS*

Многие пользователи, ранее работавшие с Oracle, знакомы с командой *GET DIAGNOSTICS*, которая позволяет узнать, что происходит в системе. Синтаксис может показаться странным тем, кто привык к современному коду, но это не умаляет ее ценности в качестве средства улучшения приложения.

На мой взгляд, команду *GET DIAGNOSTICS* стоит использовать для решения двух задач:

- получения счетчика строк;
- получения информации о контексте и трассировки стека вызовов.

Счетчик строк мы проверяем постоянно. А информация о контексте полезна для отладки. В примере ниже показано, как используется *GET DIAGNOSTICS*:

```
CREATE FUNCTION get_diag() RETURNS int AS
$$
DECLARE
    rc int;
    _sqlstate text;
    _message text;
    _context text;
BEGIN
    EXECUTE 'SELECT * FROM generate_series(1, 10)';
    GET DIAGNOSTICS rc = ROW_COUNT;
    RAISE NOTICE 'счетчик строк: %', rc;
    SELECT rc / 0;
    EXCEPTION
        WHEN OTHERS THEN
            GET STACKED DIAGNOSTICS
                _sqlstate = returned_sqlstate,
                _message = message_text,
                _context = pg_exception_context;
            RAISE NOTICE 'sqlstate: %, message: %, context: [%]%',
                _sqlstate,
                _message,
                replace( _context, E'\n', ' <- ' );
    RETURN rc;
END;
$$ LANGUAGE 'plpgsql';
```

Сразу после объявления переменных мы выполняем SQL-команду и запрашиваем у *GET DIAGNOSTICS* счетчик строк, который затем выводим в отладочном сообщении. Затем функция намеренно вызывает ошибку, после чего мы запрашиваем у *GET DIAGNOSTICS* информацию и отображаем ее.

Вот что происходит в результате вызова функции *get_diag*:

```
test=# SELECT get_diag();
```

ЗАМЕЧАНИЕ: счетчик строк: 10

КОНТЕКСТ: функция PL/pgSQL *get_diag()*, строка 10, оператор *RAISE*

ЗАМЕЧАНИЕ: sqlstate: 22012, message: деление на ноль, context: [SQL-оператор "SELECT rc / 0" <- функция PL/pgSQL *get_diag()*, строка 11, оператор SQL-оператор]

КОНТЕКСТ: функция PL/pgSQL *get_diag()*, строка 18, оператор *RAISE*

```
get_diag
-----
      10
(1 строка)
```

Как видим, команда GET DIAGNOSTICS дает подробную информацию о событии.

Использование курсора для получения данных порциями

В процессе выполнения SQL-команды база данных вычисляет результат и отправляет его приложению. Отправив результат клиенту, приложение может продолжать работу. Проблема в том, что делать, если результирующий набор так велик, что не помещается в память. Что, если база вернула 10 млрд строк? Клиентское приложение не может обработать столько данных сразу, да и не должно. Решение – использовать курсор. Идея курсора заключается в том, что данные генерируются по мере необходимости (в момент вызова команды FETCH). Поэтому приложение может приступить к потреблению данных, пока база их еще генерирует. Ко всему прочему, объем памяти, необходимой для выполнения операции, оказывается гораздо меньше.

В PL/pgSQL курсоры также играют важную роль. Когда мы в цикле проходим по результирующему набору, PostgreSQL автоматически использует курсор. В итоге потребление памяти приложениями значительно снижается, и маловероятно, что из-за большого объема данных приложению не хватит памяти. Ниже приведен упрощенный пример использования курсора внутри функции:

```
CREATE OR REPLACE FUNCTION c(int)
  RETURNS setof text AS
$$
DECLARE
  v_rec record;
BEGIN
  FOR v_rec IN SELECT tablename FROM pg_tables LIMIT $1
  LOOP
    RETURN NEXT v_rec.tablename;
  END LOOP;
  RETURN;
END;
$$ LANGUAGE 'plpgsql';
```

Этот код интересен по двум причинам. Во-первых, это **функция, возвращающая множество** (Set Returning Function – **SRF**). Она порождает целый столбец, а не одну строку. Для этого мы дополняем указание возвращаемого типа данных ключевым словом `setof`. Команда `RETURN NEXT` продолжает строить результирующий набор, пока не будет достигнут конец данных. Команда `RETURN` говорит, что мы сделали все, что хотели, и желаем выйти из функции.

Во-вторых, обход запроса в цикле автоматически приводит к созданию внутреннего курсора. Иными словами, не нужно бояться, что внезапно кончится память. PostgreSQL оптимизирует запрос так, чтобы первые 10% данных (значение параметра `cursor_tuple_fraction`) вернуть как можно быстрее.

Вот что вернет этот запрос:

```
test=# SELECT * FROM c(3);
      c
-----
 t_test
pg_statistic
pg_type
(3 строки)
```

В этом примере печатается просто перечень случайных таблиц. Если в вашей системе результат отличается, то не пугайтесь – так и должно быть.

То, что вы сейчас видели, – на мой взгляд, самый употребительный способ использования неявных курсоров в PL/pgSQL. В примере ниже показан устаревший механизм, который, возможно, знаком читателям, имеющим опыт работы с Oracle:

```
CREATE OR REPLACE FUNCTION d(int)
  RETURNS setof text AS
$$
DECLARE
  v_cur refcursor;
  v_data text;
BEGIN
  OPEN v_cur FOR SELECT tablename FROM pg_tables LIMIT $1;
  WHILE true LOOP
    FETCH v_cur INTO v_data;
    IF FOUND THEN
      RETURN NEXT v_data;
    ELSE
      RETURN;
    END IF;
  END LOOP;
END;
$$ LANGUAGE 'plpgsql';
```

Здесь мы явно объявляем и открываем курсор. Затем в цикле данные явно выбираются и возвращаются пользователю. Оба подхода в точности эквивалентны. Какой выбрать, зависит от пристрастий разработчика.

У вас сложилось ощущение, что вы еще не все знаете о курсорах? Так оно и есть: существует третий вариант, делающий ровно то же самое:

```
CREATE OR REPLACE FUNCTION e(int)
  RETURNS setof text AS
$$
DECLARE
  v_cur CURSOR (param1 int) FOR SELECT tablename FROM pg_tables LIMIT param1;
  v_data text;
BEGIN
  OPEN v_cur ($1);
  WHILE true LOOP
```

```
    FETCH v_cur INTO v_data;
    IF FOUND THEN
        RETURN NEXT v_data;
    ELSE
        RETURN;
    END IF;
END LOOP;
END;
$$ LANGUAGE 'plpgsql';
```

В этом случае курсору передается целочисленный параметр, совпадающий с тем, что был передан функции (\$1).

Иногда курсор не используется в самой функции, а возвращается вызывающей программе. В таком случае возвращаемое значение должно иметь тип `refcursor`:

```
CREATE OR REPLACE FUNCTION cursor_test(c refcursor)
    RETURNS refcursor AS
$$
BEGIN
    OPEN c FOR SELECT * FROM generate_series(1, 10) AS id;
    RETURN c;
END;
$$ LANGUAGE plpgsql;
```

Логика простая. Функции передается имя курсора, а она открывает и возвращает курсор. Прелесть в том, что связанный с курсором запрос можно создать и откомпилировать динамически.

Приложение может производить выборку из возвращенного курсора точно так же, как и раньше, например:

```
test=# BEGIN;
BEGIN
test=# SELECT cursor_test('mytest');
   cursor_test
-----
mytest
(1 строка)

test=# FETCH NEXT FROM mytest;
   id
----
   1
(1 строка)

test=# FETCH NEXT FROM mytest;
   id
----
   2
(1 строка)
```

Отметим, что это работает только внутри блока транзакции.

В этом разделе мы узнали, что курсоры порождают данные только при чтении из них. Это справедливо для большинства запросов. Но в приведенном выше примере есть подвох – если используется функция, возвращающая множество, то результат должен быть материализован целиком. Он создается не на лету, а сразу. Дело в том, что SQL должен иметь возможность повторно просмотреть отношение. Это легко в случае обычной таблицы, но не для функции. Поэтому результат SRF-функции всегда вычисляется и материализуется, что делает использование курсора в примере выше абсолютно бесполезным. Иными словами, при написании функций необходима осмотрительность, иногда опасность прячется в технических деталях.

Использование составных типов

В большинстве других СУБД хранимые процедуры используются только в сочетании с примитивными типами данных: целочисленными, вещественными, строковыми и т. д. PostgreSQL в этом отношении сильно отличается. Разрешается использовать любой доступный тип: примитивный, составной или пользовательский. Нет абсолютно никаких ограничений на типы. Для раскрытия всей мощи PostgreSQL составные типы очень важны и зачастую применяются в расширениях, которые можно найти в интернете.

В примере ниже показано, как составной тип передается функции, используется внутри нее и возвращается наружу.

```
CREATE TYPE my_cool_type AS (s text, t text);

CREATE FUNCTION f(my_cool_type)
  RETURNS my_cool_type AS
$$
DECLARE
  v_row my_cool_type;
BEGIN
  RAISE NOTICE 'schema: (%) / table: (%)', $1.s, $1.t;

  SELECT schemaname, tablename INTO v_row FROM pg_tables
  WHERE tablename = trim($1.t) AND schemaname = trim($1.s)
  LIMIT 1;

  RETURN v_row;
END;
$$ LANGUAGE 'plpgsql';
```

Главное, на что следует обратить внимание, – синтаксис `$1.field_name` для доступа к данным составного типа. Вернуть составной тип тоже нетрудно. Нужно лишь динамически построить значение составного типа и вернуть его, как любое другое. Можно даже использовать массивы и еще более сложные структуры.

Ниже показано, что возвращает PostgreSQL:

```
test=# SELECT (f).s, (f).t
        FROM f ('(public', 't_test')::my_cool_type);
ЗАМЕЧАНИЕ: schema: (public) / table: ( t_test)
   s   |   t
```



```
-----+-----
public | t_test
(1 строка)
```

Написание триггеров на PL/pgSQL

Серверный код особенно часто применяется, когда требуется отреагировать на события, происходящие в базе данных. Триггер позволяет вызвать функцию, если над таблицей выполняется команда INSERT, UPDATE, DELETE или TRUNCATE. Вызываемая триггером функция может модифицировать изменившиеся данные или просто выполнить необходимую операцию.

В PostgreSQL возможности триггеров с годами расширялись, и вот что мы имеем теперь:

```
test=# \h CREATE TRIGGER
Команда: CREATE TRIGGER
Описание: создать триггер
Синтаксис:
CREATE [ CONSTRAINT ] TRIGGER имя { BEFORE | AFTER | INSTEAD OF } { событие [ OR ... ] }
ON имя_таблицы
[ FROM ссылающаяся_таблица ]
[ NOT DEFERRABLE | [ DEFERRABLE ] [ INITIALLY IMMEDIATE | INITIALLY DEFERRED ] ]
[ REFERENCING { { OLD | NEW } TABLE [ AS ] имя_переходного_отношения } [...] ]
[ FOR [ EACH ] { ROW | STATEMENT } ]
[ WHEN ( условие ) ]
EXECUTE PROCEDURE имя_функции ( аргументы )
```

где допустимое событие:

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

Сразу отметим, что триггер всегда срабатывает для таблицы или представления и вызывает некоторую функцию. У триггера есть имя и момент срабатывания: до или после события. В PostgreSQL над одной таблицей можно определить сколько угодно триггеров. Для твердых приверженцев PostgreSQL в этом нет ничего удивительного, но хочу отметить, что во многих дорогих коммерческих СУБД, которые все еще используются в мире, это не так.

Если над таблицей определено несколько триггеров, то, согласно правилу, введенному много лет назад в версии PostgreSQL 7.3, они срабатывают в алфавитном порядке. Сначала срабатывают триггеры BEFORE, затем PostgreSQL выполняет операцию над строкой, для которой сработал триггер, а потом срабатывают триггеры AFTER. Иными словами, порядок выполнения триггеров детерминирован, а их количество практически не ограничено.

Триггер может модифицировать данные до указанного в команде изменения. В общем случае это хороший способ проверить данные и завершить операцию с ошибкой, если нарушены какие-то пользовательские ограниче-

ния. В примере ниже показан триггер, который срабатывает при выполнении команды INSERT и изменяет данные, добавленные в таблицу:

```
CREATE TABLE t_sensor (
    id serial,
    ts timestamp,
    temperature numeric
);
```

В нашей таблице имеется всего три столбца. Задача – вызывать функцию при вставке каждой строки:

```
CREATE OR REPLACE FUNCTION trig_func()
    RETURNS trigger AS
$$
BEGIN
    IF NEW.temperature < -273
    THEN
        NEW.temperature := 0;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE 'plpgsql';
```

Как уже было сказано, триггер всегда вызывает функцию, что позволяет удобно абстрагировать код. Важно, что триггерная функция обязана вернуть значение типа trigger. Чтобы получить доступ к строке, которую мы собираемся вставить, нужно обратиться к переменной NEW.



Триггеры INSERT и UPDATE всегда предоставляют переменную NEW. Триггеры UPDATE и DELETE предоставляют переменную OLD. Эти переменные содержат модифицируемую строку.

В нашем примере код проверяет, не слишком ли мала температура. Если это так, то значение неправильное, поэтому оно динамически корректируется. Чтобы модифицированную строку можно было использовать, мы просто возвращаем NEW. Если вслед за первым триггером вызывается второй, то новая функция увидит уже модифицированную строку.

На следующем шаге создадим сам триггер командой CREATE TRIGGER:

```
CREATE TRIGGER sensor_trig
    BEFORE INSERT ON t_sensor
    FOR EACH ROW
    EXECUTE PROCEDURE trig_func();
```

Вот каков эффект этого триггера:

```
test=# INSERT INTO t_sensor (ts, temperature)
VALUES ('2017-05-04 14:43', -300) RETURNING *;
 id |          ts          | temperature
-----+-----+-----
  1 | 2017-05-04 14:43:00 |           0
(1 строка)

INSERT 0 1
```

Как видим, значение было правильно скорректировано. В таблицу добавлена запись, в которой температура равна 0.

Триггер очень много знает о себе самом. Ему доступны переменные, позволяющие писать более изощренный код, обеспечивающий лучшее абстрагирование.

Сначала удалим только что созданный триггер:

```
test=# DROP TRIGGER sensor_trig ON t_sensor;
DROP TRIGGER
```

Затем напомним новую функцию и пересоздадим триггер:

```
CREATE OR REPLACE FUNCTION trig_demo()
  RETURNS trigger AS
$$
BEGIN
  RAISE NOTICE 'TG_NAME: %', TG_NAME;
  RAISE NOTICE 'TG_RELNAME: %', TG_RELNAME;
  RAISE NOTICE 'TG_TABLE_SCHEMA: %', TG_TABLE_SCHEMA;
  RAISE NOTICE 'TG_TABLE_NAME: %', TG_TABLE_NAME;
  RAISE NOTICE 'TG_WHEN: %', TG_WHEN;
  RAISE NOTICE 'TG_LEVEL: %', TG_LEVEL;
  RAISE NOTICE 'TG_OP: %', TG_OP;
  RAISE NOTICE 'TG_NARGS: %', TG_NARGS;
  -- RAISE NOTICE 'TG_ARGV: %', TG_ARGV;
  RETURN NEW;
END;
$$ LANGUAGE 'plpgsql';

CREATE TRIGGER sensor_trig
  BEFORE INSERT ON t_sensor
  FOR EACH ROW
  EXECUTE PROCEDURE trig_demo();
```

Все эти переменные предопределены и по умолчанию доступны. Наш код просто печатает их значения:

```
test=# INSERT INTO t_sensor (ts, temperature)
  VALUES ('2017-05-04 14:43', -300) RETURNING *;
ЗАМЕЧАНИЕ: TG_NAME: demo_trigger
ЗАМЕЧАНИЕ: TG_RELNAME: t_sensor
ЗАМЕЧАНИЕ: TG_TABLE_SCHEMA: public
ЗАМЕЧАНИЕ: TG_TABLE_NAME: t_sensor
ЗАМЕЧАНИЕ: TG_WHEN: BEFORE
ЗАМЕЧАНИЕ: TG_LEVEL: ROW
ЗАМЕЧАНИЕ: TG_OP: INSERT
ЗАМЕЧАНИЕ: TG_NARGS: 0
```

id	ts	temperature
2	2017-05-04 14:43:00	-300

(1 строка)

```
INSERT 0 1
```

Как видим, триггер знает свое имя, таблицу, для которой сработал, и многое другое. Если нужно выполнить схожие действия для разных таблиц, то мы можем написать одну функцию, воспользовавшись этими переменными, чтобы избежать дублирования кода.

До сих пор мы говорили о простых триггерах уровня строки, которые срабатывают по одному разу для каждой затронутой строки. Но в версии PostgreSQL 10.0 были добавлены новые возможности. Триггеры уровня команды существовали довольно давно. Однако раньше было невозможно получить доступ к данным, измененным триггером. В PostgreSQL 10.0 это исправлено, и теперь мы можем пользоваться переходными таблицами, содержащими все сделанные изменения.

Ниже приведен полный код, демонстрирующий использование переходной таблицы:

```
CREATE OR REPLACE FUNCTION transition_trigger()
  RETURNS TRIGGER AS $$
DECLARE
  v_record record;
BEGIN
  IF (TG_OP = 'INSERT') THEN
    RAISE NOTICE 'новые данные: ';
    FOR v_record IN SELECT * FROM new_table
    LOOP
      RAISE NOTICE '%', v_record;
    END LOOP;
  ELSE
    RAISE NOTICE 'старые данные: ';
    FOR v_record IN SELECT * FROM old_table
    LOOP
      RAISE NOTICE '%', v_record;
    END LOOP;
  END IF;
  RETURN NULL; -- результат игнорируется, поскольку это триггер AFTER
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER transition_test_trigger_ins
  AFTER INSERT ON t_sensor
  REFERENCING NEW TABLE AS new_table
  FOR EACH STATEMENT EXECUTE PROCEDURE transition_trigger();

CREATE TRIGGER transition_test_trigger_del
  AFTER DELETE ON t_sensor
  REFERENCING OLD TABLE AS old_table
  FOR EACH STATEMENT EXECUTE PROCEDURE transition_trigger();
```

В этом случае нам нужно два определения триггера, потому что в одно определение уместить все невозможно. Внутри триггерной функции переходную таблицу использовать легко – точно так же, как обычную.

Протестируем код этого триггера, для чего вставим и удалим данные:

```
INSERT INTO t_sensor
  SELECT *, now(), random() * 20
  FROM generate_series(1, 5);
DELETE FROM t_sensor;
```

В этом примере код просто выдает сообщение NOTICE для каждой записи в переходной таблице:

```
ЗАМЕЧАНИЕ: новые данные:
ЗАМЕЧАНИЕ: (1,"2017-10-04 15:47:14.129151",10.4552665632218)
ЗАМЕЧАНИЕ: (2,"2017-10-04 15:47:14.129151",12.8670312650502)
ЗАМЕЧАНИЕ: (3,"2017-10-04 15:47:14.129151",14.3934494629502)
ЗАМЕЧАНИЕ: (4,"2017-10-04 15:47:14.129151",4.35718866065145)
ЗАМЕЧАНИЕ: (5,"2017-10-04 15:47:14.129151",10.9121138229966)

INSERT 0 5
ЗАМЕЧАНИЕ: старые данные:
ЗАМЕЧАНИЕ: (1,"2017-10-04 15:47:14.129151",10.4552665632218)
ЗАМЕЧАНИЕ: (2,"2017-10-04 15:47:14.129151",12.8670312650502)
ЗАМЕЧАНИЕ: (3,"2017-10-04 15:47:14.129151",14.3934494629502)
ЗАМЕЧАНИЕ: (4,"2017-10-04 15:47:14.129151",4.35718866065145)
ЗАМЕЧАНИЕ: (5,"2017-10-04 15:47:14.129151",10.9121138229966)
DELETE 5
```

Помните, что переходные таблицы, содержащие миллиарды строк, – не лучшая идея. PostgreSQL, конечно, масштабируется, но в какой-то момент становится очевидно, что все имеет свою цену в терминах производительности.

Создание хранимых процедур на PL/pgSQL

Теперь разберемся, как писать процедуры. В этом разделе мы научимся создавать хранимые процедуры, появившиеся в версии PostgreSQL 11. Для создания процедур предназначена команда CREATE PROCEDURE. Синтаксически она очень похожа на CREATE FUNCTION. Но есть и мелкие различия:

```
test=# \h CREATE PROCEDURE
Команда: CREATE PROCEDURE
Описание: создать процедуру
Синтаксис:
CREATE [ OR REPLACE ] PROCEDURE
  имя ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
  [ { DEFAULT | = } выражение_по_умолчанию ]
  [ , ... ] ] )
{ LANGUAGE имя_языка
  | TRANSFORM { FOR TYPE имя_типа } [ , ... ]
  | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
  | SET параметр_конфигурации { TO значение | = значение | FROM CURRENT }
  | AS 'определение'
  | AS 'объектный_файл', 'объектный_символ'
} ...
```

В примере ниже показана хранимая процедура, выполняющая две транзакции. Первая транзакция фиксируется и создает две таблицы, вторая откатывается:

```
test=# CREATE PROCEDURE test_proc()
      LANGUAGE plpgsql
AS $$
BEGIN
    CREATE TABLE a (aid int);
    CREATE TABLE b (bid int);
    COMMIT;
    CREATE TABLE c (cid int);
    ROLLBACK;
END;
$$;
CREATE PROCEDURE
```

Как было сказано выше, процедура может явно управлять транзакциями. Назначение процедуры – запускать пакетные задания и другие операции, которые трудно реализовать с помощью функции.

Для запуска процедуры нужно выполнить команду CALL:

```
test=# CALL test_proc();
CALL
```

Первые две таблицы созданы, третья не создана из-за отката внутри процедуры:

```
test=# \d
      Список отношений
Схема | Имя | Тип | Владелец
-----+-----+-----+-----
public | a   | таблица | hs
public | b   | таблица | hs
(2 строки)
```

Процедуры – одна из самых важных возможностей, добавленных в PostgreSQL 11, они значительно повысят эффективность разработки ПО.

Введение в PL/Perl

Можно еще много говорить о PL/pgSQL. Но поскольку невозможно охватить все в одной книге, пора переходить к следующему процедурному языку. PL/Perl многими считается идеальным языком для манипуляций со строками. Как вы, наверное, знаете, язык Perl славится своими средствами для работы со строками и потому, несмотря на почтенный возраст, остается популярным.

Подключить PL/Perl можно двумя способами:

```
test=# CREATE EXTENSION plperl;
CREATE EXTENSION
test=# CREATE EXTENSION plperl;
CREATE EXTENSION
```

Можно развернуть надежный или ненадежный Perl. Если вам нужно то и другое, развертывайте оба расширения.

Чтобы показать, как работает PL/Perl, я написал функцию, которая разбирает адрес электронной почты и возвращает true или false.

```
test=# CREATE OR REPLACE FUNCTION verify_email(text)
      RETURNS boolean AS
$$
if ($_[0] =~ /^[a-z0-9.]+@[a-z0-9.-]+$/ )
{
    return true;
}
return false;
$$ LANGUAGE 'plperl';
CREATE FUNCTION
```

Функции передается параметр text. Внутри функции все входные параметры доступны с помощью специальной переменной \$_. В данном случае выполняется регулярное выражение и возвращается значение true или false.

Функция вызывается как любая другая процедура, написанная на любом другом языке:

```
test=# SELECT verify_email('hs@cybertec.at');
verify_email
-----
t
(1 строка)

test=# SELECT verify_email('totally wrong');
verify_email
-----
f
(1 строка)
```

Использование PL/Perl для абстрагирования типа данных

Как уже было сказано в этой главе, функции в PostgreSQL довольно универсальны и используются в разных контекстах. Если вы хотите использовать функции для повышения качества данных, то в вашем распоряжении команда CREATE DOMAIN:

```
test=# \h CREATE DOMAIN
Команда: CREATE DOMAIN
Описание: создать домен
Синтаксис:
CREATE DOMAIN имя [ AS ] тип_данных
[ COLLATE правило_сортировки ]
[ DEFAULT выражение ]
[ ограничение [ ... ] ]
```

где ограничение:

```
[ CONSTRAINT имя_ограничения ]
{ NOT NULL | NULL | CHECK (выражение) }
```

В следующем примере функция на PL/Perl используется для создания домена email, который может выступать в роли типа данных.

```
test=# CREATE DOMAIN email AS text
        CHECK (verify_email(VALUE) = true);
CREATE DOMAIN
```

Доменные функции ведут себя как обычный тип данных:

```
test=# CREATE TABLE t_email (id serial, data email);
CREATE TABLE
```

Perl-функция гарантирует, что строка, нарушающая ограничения, не попадет в базу данных:

```
test=# INSERT INTO t_email (data)
        VALUES ('somewhere@example.com');
INSERT 0 1
test=# INSERT INTO t_email (data)
        VALUES ('somewhere_wrong_example.com');
ОШИБКА: значение из домена email нарушает проверочное ограничение "email_check"
```

Perl, возможно, и является хорошим инструментом для работы со строками, но, как обычно, вам решать, хотите ли вы, чтобы такой код выполнялся внутри самой базы данных.

Выбор между PL/Perl и PL/PerlU

До сих пор мы всего лишь сравнивали строку с регулярным выражением, так что никаких угроз безопасности в коде на Perl не возникало. Но что, если попытаться совершить нечто недопустимое внутри Perl-функции? Тогда PL/Perl просто выдаст ошибку:

```
test=# CREATE OR REPLACE FUNCTION test_security()
        RETURNS boolean AS
$$
use strict;
my $fp = open("/etc/password", "r");

return false;
$$ LANGUAGE 'plperl';
ОШИБКА: 'open' trapped by operation mask at line
КОНТЕКСТ: компиляция функции PL/Perl "test_security"
```

PL/Perl ругается уже в момент создания функции. Сообщение об ошибке появляется мгновенно.

Если вы все же хотите выполнять ненадежный код на Perl, то должны использовать PL/PerlU:

```
test=# CREATE OR REPLACE FUNCTION first_line()
        RETURNS text AS
$$
open(my $fh, '<:encoding(UTF-8)', "/etc/passwd")
    or elog(NOTICE, "Could not open file '$filename' $!");
```



```
my $row = <$fh>;
close($fh);

return $row;
$$ LANGUAGE 'plperl';
CREATE FUNCTION
```

Процедура не изменилась. Она по-прежнему возвращает строку. Но теперь ей разрешено делать все, что угодно. Единственное отличие – то, что теперь функция помечена как написанная на языке plperl.

Результат не вызывает удивления:

```
test=# SELECT first_line();
           first_line
-----
root:x:0:0:root:/root:/bin/bash+
(1 строка)
```

Использование интерфейса SPI

Иногда процедура, написанная на Perl, должна обращаться к базе данных. Напомним, что функция выполняется в контексте соединения с базой. Поэтому нет смысла создавать новое соединение. Для обращения к базе сервер PostgreSQL предоставляет интерфейс SPI к внутренним механизмам, написанный на C. Все процедурные языки, применяемые для написания серверного кода, пользуются этим интерфейсом. PL/Perl – не исключение, и в этом разделе мы покажем, как использовать Perl-обертку вокруг интерфейса SPI.

Типичная задача – выполнить SQL-команду и получить количество выбранных строк. Именно это делает функция spi_exec_query. Первый ее параметр – сам запрос, второй параметр – количество подлежащих выборке строк. Для простоты я решил выбрать все. Код приведен в следующем примере:

```
test=# CREATE OR REPLACE FUNCTION spi_sample(int)
RETURNS void AS
$$
my $rv = spi_exec_query(" SELECT * FROM generate_series(1, $_[0])", $_[0]);
elog(NOTICE, "выбрано строк: " . $rv->{processed});
elog(NOTICE, "состояние: " . $rv->{status});

return;
$$ LANGUAGE 'plperl';
```

SPI выполняет запрос и отображает количество строк. Важно, что все языки для написания хранимых процедур предоставляют средства протоколирования. В случае PL/Perl эта функция называется elog и принимает два параметра. Первый описывает важность сообщения (INFO, NOTICE, WARNING, ERROR и т. д.), второй – само сообщение.

Ниже показано, что возвращает запрос:

```
test=# SELECT spi_sample(9);
ЗАМЕЧАНИЕ:  выбрано строк: 9
ЗАМЕЧАНИЕ:  состояние: SPI_OK_SELECT
```

```
spi_sample
-----
```

(1 строка)

Применение интерфейса SPI в функциях, возвращающих множество

Часто мы хотим не просто выполнить SQL-команду и забыть о ней. В большинстве случаев процедура обходит результирующий набор в цикле и что-то с ним делает. В следующем примере показано, как обойти результат запроса. Заодно я решил сделать пример более содержательным и продемонстрировать функцию, возвращающую составной тип данных. Работать с составными типами в Perl очень легко – можно просто поместить данные в хеш и вернуть его.

Функция `return_next` постепенно строит результирующий набор и завершается предложением `return`.

В данном случае таблица заполняется случайными значениями:

```
CREATE TYPE random_type AS (a float8, b float8);

CREATE OR REPLACE FUNCTION spi_srf_perl(int)
    RETURNS setof random_type AS
$$
my $rv = spi_query("SELECT random() AS a, random() AS b
                    FROM generate_series(1, $_[0])");
while (defined (my $row = spi_fetchrow($rv)))
{
    elog(NOTICE, "данные: " . $row->{a} . " / " . $row->{b});
    return_next({a => $row->{a}, b => $row->{b}});
}
return;
$$ LANGUAGE 'plperl';
CREATE FUNCTION
```

Сначала мы вызываем функцию `spi_query` и в цикле обходим результирующий набор, вызывая функцию `spi_fetchrow`. На каждой итерации цикла в хеш помещается значение составного типа.

Как и следовало ожидать, функция возвращает множество случайных значений:

```
test=# SELECT * FROM spi_srf_perl(3);
ЗАМЕЧАНИЕ: данные: 0.154673356097192 / 0.278830723837018
КОНТЕКСТ: функция PL/Perl "spi_srf_perl"
ЗАМЕЧАНИЕ: данные: 0.615888888947666 / 0.632620786316693
КОНТЕКСТ: функция PL/Perl "spi_srf_perl"
ЗАМЕЧАНИЕ: данные: 0.910436692181975 / 0.753427186980844
КОНТЕКСТ: функция PL/Perl "spi_srf_perl"
   a_col      |      b_col
-----+-----
0.154673356097192 | 0.278830723837018
0.615888888947666 | 0.632620786316693
0.910436692181975 | 0.753427186980844
(3 строки)
```

Помните, что результат функции, возвращающей множество, материализуется и целиком хранится в памяти.

Экранирование в PL/Perl и вспомогательные функции

До сих пор мы работали только с целыми числами, поэтому вопрос о внедрении SQL-кода или о специальных именах таблиц не стоял. Но вообще-то в PL/Perl имеются следующие функции:

- `quote_literal`: экранирует литеральную строку;
- `quote_nullable`: экранирует строку;
- `quote_ident`: экранирует идентификаторы SQL (имена объектов и т. п.);
- `decode_bytea`: декодирует байтовый массив PostgreSQL;
- `encode_bytea`: кодирует данные, преобразуя их в байтовый массив;
- `encode_literal_array`: кодирует массив литералов;
- `encode_typed_literal`: преобразует переменную Perl в значение, тип которого указан во втором аргументе, и возвращает строковое представление этого значения;
- `encode_array_constructor`: возвращает содержимое массива в виде строки в формате конструктора массива;
- `looks_like_number`: возвращает true, если строка выглядит как число;
- `is_array_ref`: возвращает true, если аргумент выглядит как ссылка на массив.

Эти функции доступны всегда, для обращения к ним не нужно загружать никакую библиотеку.

Сохранение данных между вызовами функций

Иногда необходимо сохранить данные между вызовами, и в инфраструктуре есть соответствующие средства. В Perl хеш можно использовать для хранения произвольных данных. Рассмотрим следующий пример:

```
CREATE FUNCTION perl_shared(text) RETURNS int AS
$$
if ( !defined $_SHARED[$_[0]] )
{
    $_SHARED[$_[0]] = 0;
}
else
{
    $_SHARED[$_[0]]++;
}
return $_SHARED[$_[0]];
$$ LANGUAGE 'plperl';
```

Переменная `$_SHARED` инициализируется нулем, если оказывается, что переданного функции ключа еще нет в хеше. В противном случае счетчик увеличивается на 1, так что на выходе мы получаем:

```
test=# SELECT perl_shared('some_key') FROM generate_series(1, 3);
perl_shared
```

```
-----
0
1
2
(3 строки)
```

В более сложных случаях разработчик не знает, в каком порядке будут вызываться функции, и об этом важно помнить.

Создание триггеров на Perl

На любом входящем в ядро PostgreSQL процедурном языке можно писать триггеры, в т. ч. на Perl. Поскольку размер этой главы ограничен, я решил не включать пример триггера на Perl, а отослать читателя на страницу официальной документации по адресу <https://www.postgresql.org/docs/current/static/plperl-triggers.html>¹.

По существу, триггеры на Perl пишутся так же, как на PL/pgSQL. Все predefined переменные присутствуют, а к возвращаемым значениям применимы общие для всех процедурных языков правила.

Введение в PL/Python

Если вы незнакомы с Perl, то, быть может, вам подойдет PL/Python. Язык Python уже давно является частью инфраструктуры PostgreSQL и, стало быть, хорошо протестирован.

Говоря о PL/Python, следует помнить, что он существует только в ненадежной версии, и это важно с точки зрения безопасности.

Для подключения PL/Python выполните следующую команду из командной строки, в ней test – имя базы данных, в которой вы хотите использовать PL/Python:

```
createlang plpythonu test
```

После этого можно писать код на PL/Python.

Разумеется, вместо этого можно воспользоваться командой CREATE LANGUAGE. Имейте в виду, что для работы с серверными языками необходимы также поддерживающие их пакеты PostgreSQL (postgresql-plpython-\$(VERSIONNUMBER) и т. д.).

Написание простого кода на PL/Python

В этом разделе мы научимся писать процедуры на Python на следующем простом примере: в Австрии, если вы посещаете клиента на машине, то можете вычесть 42 евроцента на километр из суммы, на которую начисляется подоходный налог. Наша функция должна будет получить количество километров и вернуть сумму, вычитаемую из налогооблагаемого дохода.

```
CREATE OR REPLACE FUNCTION calculate_deduction(km float)
  RETURNS numeric AS
$$
```

¹ На русском языке <https://postgrespro.ru/docs/postgresql/11/plperl-triggers>. – Прим. перев.

```
if km <= 0:
    elog(ERROR, 'неправильное количество километров')
else:
    return km * 0.42
$$ LANGUAGE 'plpythonu';
```

Функция проверяет, что передано положительное число. Если это так, то она вычисляет и возвращает результат. Передача функции серверу PostgreSQL осуществляется так же, как для Perl и PL/pgSQL.

Использование интерфейса SPI

Как и все процедурные языки, PL/Python предоставляет доступ к интерфейсу SPI. В примере ниже показано суммирование чисел.

```
CREATE FUNCTION add_numbers(rows_desired integer)
    RETURNS integer AS
$$
mysum = 0
cursor = plpy.cursor("SELECT * FROM
    generate_series(1, %d) AS id" % (rows_desired))
while True:
    rows = cursor.fetch(rows_desired)
    if not rows:
        break
    for row in rows:
        mysum += row['id']
return mysum
$$ LANGUAGE 'plpythonu';
```

Если будете выполнять этот пример, то наберите код, начинающийся словом `cursor`, в одной строке. В Python отступы – часть синтаксиса, поэтому разбивать строку в произвольном месте нельзя.

Создав курсор, мы можем обойти его в цикле и вычислить сумму чисел. Ссылаться на столбец можно по его имени.

Вызов функции дает ожидаемый результат:

```
test=# SELECT add_numbers(10);
 add_numbers
-----
          55
(1 строка)
```

Для просмотра результирующего набора, возвращенного SQL-командой, PL/Python предлагает различные функции. Все они являются обертками вокруг того, что предлагает интерфейс SPI, написанный на C.

Следующая функция позволяет подробно исследовать результат:

```
CREATE OR REPLACE FUNCTION result_diag(rows_desired integer)
    RETURNS integer AS
$$
rv = plpy.execute("SELECT *
```

```

FROM generate_series(1, %d) AS id" % (rows_desired))
plpy.notice(rv.nrows())
plpy.notice(rv.status())
plpy.notice(rv.colnames())
plpy.notice(rv.coltypes())
plpy.notice(rv.coltypmods())
plpy.notice(rv.str())
return 0
$$ LANGUAGE 'plpythonu';

```

Функция `nrows()` выводит число строк. Функция `status()` сообщает, были ли ошибки. Функция `colnames()` возвращает список столбцов. Функция `coltypes()` возвращает объектные идентификаторы типов данных в результирующем наборе. Так, 23 – это внутренний номер типа `integer`, как следует из распечатки ниже:

```

test=# SELECT typname FROM pg_type WHERE oid = 23;
 typname
-----
 int4
(1 строка)

```

Что касается `typmod`, то это переменная часть типа, как, например, 20 в случае типа `varchar(20)`.

Наконец, функция `rv.str()` возвращает всю эту информацию в виде одной строки для отладки. Вызов функции `result_diag()` печатает следующий результат:

```

test=# SELECT result_diag(3);
ЗАМЕЧАНИЕ:  3
ЗАМЕЧАНИЕ:  5
ЗАМЕЧАНИЕ:  ['id']
ЗАМЕЧАНИЕ:  [23]
ЗАМЕЧАНИЕ:  [-1]
ЗАМЕЧАНИЕ:  <PlyResult status=5 nrows=3 rows=[{'id': 1}, {'id': 2}, {'id': 3}]>
 result_diag
-----
          0
(1 строка)

```

Интерфейс SPI содержит еще много функций для выполнения SQL-команд.

Обработка ошибок

Ошибки встречаются, и их нужно обрабатывать. Конечно, в Python есть для этого средства. Например:

```

CREATE OR REPLACE FUNCTION trial_error()
  RETURNS text AS
$$
try:
    rv = plpy.execute("SELECT surely_a_syntax_error")
except plpy.SPIError:
    return "мы перехватили ошибку"
else:

```

```
return "все хорошо"
$$ LANGUAGE 'plpythonu';
```

Мы можем использовать обычный блок try/except и проверить код ошибки `plpy.SPIError`. После обработки ошибки функция может вернуться нормально, не отменяя текущую транзакцию:

```
test=# SELECT trial_error();
      trial_error
```

```
-----
мы перехватили ошибку
(1 строка)
```

Напомним, что PL/Python имеет доступ ко всем внутренним механизмам PostgreSQL. Поэтому процедура может получать самые разнообразные ошибки, например:

```
except spiexceptions.DivisionByZero:
    return "обнаружено деление на ноль"
except spiexceptions.UniqueViolation:
    return "обнаружено нарушение ограничения уникальности"
except plpy.SPIError, e:
    return "прочие ошибки, SQLSTATE %s" % e.sqlstate
```

Перехватывать ошибки в Python совсем нетрудно, а в результате ваши функции не будут завершаться аварийно¹.

УЛУЧШЕНИЕ ФУНКЦИЙ

Выше мы видели, как писать на разных языках простые функции и триггеры. Разумеется, PostgreSQL поддерживает гораздо больше языков. Из наиболее известных отметим PL/R (R – это развитый пакет статистических программ) и PL/v8 (основан на движке JavaScript, разработанном Google). Но эти языки выходят за рамки данной главы (несмотря на всю их полезность).

В этом разделе мы займемся повышением производительности функций. Ускорить обработку можно несколькими способами:

- уменьшение числа вызовов;
- использование кешированных планов выполнения;
- передача указаний оптимизатору.

В этой главе мы обсудим все три.

Уменьшение числа вызовов функций

В большинстве случаев производительность низкая из-за того, что функции вызываются слишком часто. Лично я считаю и не устаю подчеркивать, что слишком частые вызовы – главная причина неудовлетворительной произво-

¹ Не стоит воспринимать этот совет буквально. В необработанных исключениях нет ничего плохого: ошибку надо перехватывать в том месте, в котором её можно обработать осмысленно. – *Прим. ред.*

длительности. При создании функции можно указать одну из трех категорий изменчивости: *volatile* (изменчивая), *stable* (стабильная) и *immutable* (неизменяемая). Приведем пример:

```
test=# SELECT random(), random();
          random          |          random
-----+-----
 0.276252629235387 | 0.710661871358752
(1 строка)

test=# SELECT now(), now();
          now          |          now
-----+-----
2016-12-16 12:57:17.135751+01 | 2016-12-16 12:57:17.135751+01
(1 строка)

test=# SELECT pi();
          pi
-----
 3.14159265358979
(1 строка)
```

С изменчивой функцией оптимизатор ничего не может сделать. Она должна вызываться снова и снова. Изменчивые функции могут быть также причиной, из-за которой не используется индекс. По умолчанию любая функция считается изменчивой. Стабильная функция возвращает одни и те же данные в рамках одного оператора SQL. Оптимизатор может удалять ее вызовы. Функция *now()* – пример стабильной функции, внутри одной транзакции она возвращает одни и те же данные.

Неизменяемые функции – золотой стандарт, потому что они допускают большинство оптимизаций, а объясняется это тем, что при одних и тех же входных данных они всегда возвращают одинаковый результат. Первый шаг на пути к оптимизации функции – правильно указать ее категорию изменчивости, добавив в конец определения ключевое слово *volatile*, *stable* или *immutable*.

Использование кешированных планов

В PostgreSQL выполнение запроса состоит из четырех этапов.

1. **Анализатор:** проверяет синтаксис.
2. **Система перезаписывания:** применяет правила.
3. **Оптимизатор/планировщик:** оптимизирует запрос.
4. **Исполнитель:** выполняет план, составленный планировщиком.

Если запрос короткий, то первые три этапа занимают больше времени, чем само выполнение. Поэтому имеет смысл кешировать планы выполнения. В PL/pgSQL кеширование планов производится автоматически, нам об этом думать не надо. PL/Perl и PL/Python оставляют это на наше усмотрение.

Интерфейс SPI предлагает функции для создания и выполнения подготовленных запросов, поэтому у программиста есть выбор, подготавливать запрос или нет. Если запрос длинный, то лучше не заниматься его подготовкой. А ко-

роткие запросы обычно имеет смысл подготавливать, чтобы сократить внутренние накладные расходы.

Назначение стоимости функции

С точки зрения оптимизатора, функция ведет себя как оператор. PostgreSQL обращается со стоимостями функций так же, как если бы это был стандартный оператор. Проблема в том, что сложение двух чисел обычно дешевле, чем вычисление пересечения береговых линий с помощью какой-нибудь функции PostGIS. Но оптимизатор-то не знает, какая функция дешевая, а какая – дорогая.

К счастью, мы можем сообщить оптимизатору сравнительные оценки стоимости функций:

```
test=# \h CREATE FUNCTION
Команда: CREATE FUNCTION
Описание: создать функцию
Синтаксис:
CREATE [ OR REPLACE ] FUNCTION
...
| COST стоимость_выполнения
| ROWS строк_в_результате
...
```

Параметр `COST` говорит, насколько ваша функция дороже стандартного оператора. Это число, кратное конфигурационному параметру `cpu_operator_cost`, а не статическое значение. По умолчанию значение равно 100, если только функция не написана на C.

Теперь о параметре `ROWS`. По умолчанию PostgreSQL предполагает, что SRF-функция возвращает 1000 строк, потому что система не может точно сказать, сколько строк будет возвращено. Параметр `ROWS` позволяет сообщить PostgreSQL ожидаемое количество строк.

Использование функций для разных целей

В PostgreSQL хранимые процедуры можно использовать практически для всего. В этой главе мы уже встречались с командой `CREATE DOMAIN`, но точно так же можно создавать свои операторы, приведения типов и даже правила сортировки.

В этом разделе мы продемонстрируем создание и использование простого приведения типа. Для этого служит команда `CREATE CAST`, имеющая такой синтаксис:

```
test=# \h CREATE CAST
Команда: CREATE CAST
Описание: создать приведение
Синтаксис:
CREATE CAST (исходный_тип AS целевой_тип)
  WITH FUNCTION имя_функции [ (тип_аргумента [, ...]) ]
  [ AS ASSIGNMENT | AS IMPLICIT ]
CREATE CAST (исходный_тип AS целевой_тип)
  WITHOUT FUNCTION
```

```
[ AS ASSIGNMENT | AS IMPLICIT ]
```

```
CREATE CAST (исходный_тип AS целевой_тип)
  WITH INOUT
  [ AS ASSIGNMENT | AS IMPLICIT ]
```

Пользоваться этим очень легко. Мы просто говорим PostgreSQL, какая процедура умеет приводить один тип к другому.

В стандартном дистрибутиве PostgreSQL невозможно приведение IP-адреса к булеву значению. Вот такое приведение мы и создадим. Сначала нужно определить хранимую процедуру:

```
CREATE FUNCTION inet_to_boolean(inet)
  RETURNS boolean AS
  $$
  BEGIN
    RETURN true;
  END;
  $$ LANGUAGE 'plpgsql';
```

Для простоты пусть функция возвращает true. Но, в принципе, можно написать произвольный код преобразования на любом языке.

Дальше уже можно определить приведение типа:

```
CREATE CAST (inet AS boolean)
  WITH FUNCTION inet_to_boolean(inet) AS IMPLICIT;
```

Прежде всего мы сообщаем PostgreSQL, что хотим привести тип `inet` к типу `boolean`. Затем указываем функцию, которая это делает, и сообщаем, что предпочли бы неявное приведение.

Все, теперь можно тестировать:

```
test=# SELECT '192.168.0.34'::inet::boolean;
      bool
-----
      t
(1 строка)
```

Точно так же можно определить правило сортировки – нужно лишь указать соответствующую хранимую процедуру:

```
test=# \h CREATE COLLATION
Команда: CREATE COLLATION
Описание: создать правило сортировки
Синтаксис:
CREATE COLLATION [ IF NOT EXISTS ] имя (
  [ LOCALE = локаль, ]
  [ LC_COLLATE = категория_сортировки, ]
  [ LC_CTYPE = категория_типов_символов, ]
  [ PROVIDER = провайдер, ]
  [ VERSION = версия ]
)
CREATE COLLATION [ IF NOT EXISTS ] имя FROM существующее_правило
```

РЕЗЮМЕ

В этой главе мы научились писать хранимые процедуры. После теоретического введения мы сосредоточились на некоторых избранных возможностях PL/pgSQL. Кроме того, мы узнали, как использовать PL/Perl и PL/Python, два важных языка, входящих в дистрибутив PostgreSQL. Конечно, доступных языков гораздо больше. Но размеры книги не позволяют рассмотреть их все подробно. Дополнительные сведения можно найти на странице https://wiki.postgresql.org/wiki/PL_Matrix.

Глава 8 посвящена безопасности в PostgreSQL. Мы узнаем, как управлять пользователями и правами. Попутно мы поговорим о сетевой безопасности.

Вопросы

В чем разница между функцией и хранимой процедурой?

Слова «функция» и «процедура» часто считают синонимами. Но на самом деле между ними есть различие. В процедуре можно выполнять несколько транзакций, поэтому она не может быть частью команды SELECT, а должна вызываться командой CALL. Напротив, у функции возможности управления транзакцией ограничены, и она может быть частью команд SELECT, INSERT, UPDATE, DELETE.

В чем разница между надежным и ненадежным языком?

Надежный язык ограничивает доступ программиста к системным вызовам. Поэтому такие языки считаются безопасными. С другой стороны, на ненадежном языке можно программировать любые операции, поэтому он доступен только суперпользователям, что предотвратит бреши в системе безопасности.

Функции – это хорошо или плохо?

На этот вопрос ответить затруднительно. Не бывает в мире абсолютного добра или зла. То же относится и к функциям. Они могут ускорить достижение цели, но в случае злоупотребления способны нанести вред. Как и все остальное, функции следует использовать разумно.

Какие серверные языки имеются в PostgreSQL?

Исторически PostgreSQL проявляла большую гибкость в подходах к серверному программированию. В ядро включена поддержка SQL, PL/pgSQL, PL/Perl, PL/TCL и PL/Python. Но есть много других языков, которые можно установить из различных источников и использовать себе во благо. К наиболее популярным относятся PL/V8 (JavaScript) и PL/R¹.

Что такое триггер?

Триггер позволяет автоматически выполнять функцию, когда изменяются строки. Существуют разные типы триггеров: уровня строки, срабатывающие

¹ Для полноты картины отметим и язык C. – Прим. ред.

для каждой строки; уровня команды, срабатывающие для команды в целом; и событийные, срабатывающие при выполнении DLL-команд.

На каких языках можно писать функции?

Обычно функции пишутся на SQL, PL/pgSQL, PL/Perl, PL/TCL и PL/Python. При необходимости можно добавить и другие языки. Все зависит от конкретной ситуации, так что экспериментируйте, сколько душе угодно.

А вообще, стоит использовать функции?

Трудно сказать. Все зависит от того, к чему вы стремитесь. На мой взгляд, перемещать алгоритмы ближе к данным – идея здравая. Но есть люди, у которых на этот счет совершенно иное мнение. Я думаю, что «на вкус и на цвет товарища нет», поэтому изучите ситуацию, поймите, чего вы хотите достичь, и решайте сами, что для вас лучше.

Какой язык самый быстрый?

Зависит от того, что вы делаете. Думаю, что на этот вопрос нет и никогда не будет однозначного ответа. Нужно проверять в конкретной ситуации.

Глава 8

Безопасность в PostgreSQL

Глава 7 была посвящена хранимым процедурам и написанию серверного кода. После знакомства со многими важными темами пора обратить взоры на безопасность в PostgreSQL. Мы узнаем, как обеспечить безопасность сервера и настроить права доступа.

В этой главе рассматриваются следующие вопросы:

- настройка доступа из сети;
- управление аутентификацией;
- пользователи и роли;
- настройка безопасности базы данных;
- управление схемами, таблицами и столбцами;
- безопасность на уровне строк.

Прочитав эту главу, вы сможете профессионально настраивать безопасность в PostgreSQL.

УПРАВЛЕНИЕ СЕТЕВОЙ БЕЗОПАСНОСТЬЮ

Прежде чем переходить к практическим примерам, я хочу дать краткий обзор различных уровней безопасности, которые станут предметом этой главы. О них важно помнить, выстраивая систематический подход к организации безопасности.

Вот какую модель я держу в голове:

- **адреса привязки:** параметр `listen_addresses` в файле `postgresql.conf`;
- **управление доступом на основе имени узла:** файл `pg_hba.conf`;
- **права на уровне экземпляра:** пользователи, роли, создание базы данных, вход в систему и репликация;
- **права на уровне базы данных:** подключение, создание схем и т. д.;
- **права на уровне схемы:** использование схемы и создание объектов внутри схемы;
- **права на уровне таблицы:** выборка, вставка, обновление и т. д.;

- **права на уровне столбца:** разрешение или запрет доступа к столбцам;
- **безопасность на уровне строк:** ограничение доступа к строкам.

Прежде чем прочитать значение, PostgreSQL проверяет, достаточно ли у пользователя прав на каждом уровне. Вся цепочка прав должна быть настроена правильно.

Подключения и адреса привязки

Один из первых этапов настройки сервера PostgreSQL – определение удаленного доступа. По умолчанию PostgreSQL не принимает удаленные подключения. Сервер даже не отвергает такие попытки, поскольку он попросту не прослушивает порт. Если мы попытаемся установить соединение, то получим сообщение от операционной системы, т. к. PostgreSQL тут вообще ни при чем.

В предположении, что используется конфигурация по умолчанию для адреса 192.168.0.123, вот что мы увидим:

```
iMac:~ hs$ telnet 192.168.0.123 5432
Trying 192.168.0.123...
telnet: connect to address 192.168.0.123: Connection refused
telnet: Unable to connect to remote host
```

Telnet пытается подключиться к порту 5432 и получает немедленный отказ от удаленного компьютера. Извне создается впечатление, что сервер PostgreSQL вообще не запущен.

Ключ к успеху находится в файле `postgresql.conf`¹:

- Параметры подключения -

```
#listen_addresses = 'localhost'      # какие IP-адреса прослушивать;
# список адресов через запятую;
# по умолчанию 'localhost'; '*' означает все
# (после изменения необходимо перезапустить сервер)
```

Параметр `listen_addresses` говорит PostgreSQL, какие адреса прослушивать. На техническом жаргоне они называются адресами привязки. Разберемся, что это означает.

Предположим, что компьютер оснащен четырьмя сетевыми картами. Мы можем прослушивать, к примеру, три из соответствующих им IP-адресов. PostgreSQL принимает во внимание запросы к трем картам и игнорирует четвертую. Порт на этом IP-адресе попросту закрыт.



Мы должны включить в список `listen_addresses` IP-адрес нашего сервера, а не IP-адреса клиентов.

Если положить `listen_addresses` равным `*`, то PostgreSQL будет прослушивать все IP-адреса, назначенные компьютеру.

¹ Комментарии в файле написаны по-английски. Для удобства читателя они переведены. – *Прим. перев.*

- ✓ Имейте в виду, что после изменения `listen_addresses` необходимо перезапустить службу PostgreSQL. Динамически изменить этот параметр нельзя.

К управлению подключениями относится еще ряд не менее важных параметров:

```
#port = 5432
# (после изменения необходимо перезапустить сервер)
max_connections = 100
# (после изменения необходимо перезапустить сервер)
# Примечание: увеличение max_connections обходится примерно
# в 400 байтов разделяемой памяти на каждое подключение
# плюс память под блокировки (см. max_locks_per_transaction).
#superuser_reserved_connections = 3
# (после изменения необходимо перезапустить сервер)
#unix_socket_directories = '/tmp' # список каталогов через запятую
# (после изменения необходимо перезапустить сервер)
#unix_socket_group = ''
# (после изменения необходимо перезапустить сервер)
#unix_socket_permissions = 0777 # восьмеричное число должно начинаться с 0
# (после изменения необходимо перезапустить сервер)
```

Прежде всего PostgreSQL прослушивает единственный TCP-порт – по умолчанию 5432. При поступлении запроса порождается новый процесс для обслуживания нового подключения. По умолчанию разрешено не более 100 подключений, причем три из них зарезервированы для суперпользователей, так что для обычных пользователей остается 97 подключений.

- i После любого изменения параметров, относящихся к подключениям, необходимо перезапускать сервер. Одна из причин в том, что разделяемая память выделяется статически, и ее размер нельзя изменить на лету.

Число подключений и производительность

Меня часто спрашивают, оказывает ли максимальное количество подключений влияние на производительность. Небольшое, поскольку контекстные переключения в любом случае сопряжены с издержками. А вот что важно, так это количество открытых снимков (snapshot). Чем больше открыто снимков, тем больше накладные расходы на стороне сервера. В общем, можете увеличивать `max_connections` без опаски.

- ✓ Если вас интересуют реальные данные, почитайте мою старую статью по адресу https://www.cybertec-postgresql.com/max_connections-performance-impacts/.

Жизнь в мире без TCP

Иногда мы не хотим использовать сеть. Так часто бывает, когда база данных используется только для работы локального приложения. Быть может, база данных PostgreSQL входит в комплект поставки приложения, а быть может, вы просто не рискуете использовать сеть. В таком случае вам нужны Unix-сокеты –

не нуждающееся в сети средство коммуникации. Приложение может подключиться к Unix-сокету локально, не раскрывая ничего внешнему миру.

Но для работы с Unix-сокетом необходимо указать каталог. По умолчанию PostgreSQL использует каталог /tmp. Но если на одной машине работает несколько серверов баз данных, то у каждого должен быть свой каталог.

Помимо безопасности, существуют и другие причины не пользоваться сетью. Одна из них – производительность. Unix-сокеты гораздо быстрее закольцованного устройства (127.0.0.1). Если вам это кажется удивительным, то вы не одиноки. Так или иначе, издержки сетевого подключения не следует недооценивать, если выполняются только очень маленькие запросы.

Для иллюстрации реального положения вещей я подготовил простенький тест производительности. Файл script.sql содержит скрипт, который выбирает одно число. Проще не придумаешь.

```
[hs@linuxpc ~]$ cat /tmp/script.sql
SELECT 1
```

Далее мы можем воспользоваться программой pgbench для многократного выполнения SQL-скрипта. Параметр -f задает имя скрипта, а -c 10 означает, что мы хотим создать 10 одновременных подключений, каждое из которых активно 5 с (-T 5). Тест запускается от имени пользователя postgres и работает с базой данных postgres, которая по умолчанию существует. Все приведенные ниже примеры работают в UNIX-системах, производных от дистрибутива RHEL. В системах на основе Debian пути другие.

```
[hs@linuxpc ~]$ pgbench -f /tmp/script.sql -c 10 -T 5 -U postgres postgres 2>/dev/null
transaction type: /tmp/script.sql
scaling factor: 1
query mode: simple
number of clients: 10
number of threads: 1
duration: 5 s
number of transactions actually processed: 871407
latency average = 0.057 ms
tps = 174278.158426 (including connections establishing)
tps = 174377.935625 (excluding connections establishing)
```

Как видим, программе pgbench не передается имя узла, поэтому она подключается к локальному Unix-сокету и выполняет скрипт с максимально возможной скоростью. На моем четырехъядерном компьютере с процессором Intel система достигала производительности 174 000 транзакций в секунду.

А что, если добавить параметр -h localhost? Мы увидим такую картину:

```
[hs@linuxpc ~]$ pgbench -f /tmp/script.sql -h localhost -c 10 -T 5 -U postgres
postgres 2>/dev/null
transaction type: /tmp/script.sql
scaling factor: 1
query mode: simple
number of clients: 10
number of threads: 1
```



```
duration: 5 s
number of transactions actually processed: 535251
latency average = 0.093 ms
tps = 107000.872598 (including connections establishing)
tps = 107046.943632 (excluding connections establishing)
```

Производительность, как камень, пошла ко дну и составила всего 107 000 транзакций в секунду. Разницу, очевидно, следует отнести на счет сетевых издержек.

❏ Задав параметр `-j` (количество потоков в `pgbench`), мы сможем выжать из системы еще немного транзакций. Но в моем примере общая картина при этом не изменилась. В других ситуациях может измениться, поскольку если процессор недостаточно мощный, то `pgbench` может оказаться узким местом.

Как видим, сеть может создавать проблемы не только для безопасности, но и для производительности.

Файл `pg_hba.conf`

Задав адреса привязки, мы можем перейти к следующему шагу. Из файла `pg_hba.conf` PostgreSQL узнает, как аутентифицировать пользователей, обращающихся к базе. В этом файле могут присутствовать записи следующего вида:

```
# local    БАЗА ПОЛЬЗОВАТЕЛЬ МЕТОД [ДОП.]
# host     БАЗА ПОЛЬЗОВАТЕЛЬ АДРЕС МЕТОД [ДОП.]
# hostssl  БАЗА ПОЛЬЗОВАТЕЛЬ АДРЕС МЕТОД [ДОП.]
# hostnossl БАЗА ПОЛЬЗОВАТЕЛЬ АДРЕС МЕТОД [ДОП.]
```

Ниже описано, что означает каждая запись:

- `local`: используется для настройки локальных подключений через Unix-сокеты;
- `host`: используется для настройки подключений по сети с использованием и без использования SSL;
- `hostssl`: относится только к SSL-подключениям. Для этого сервер должен быть откомпилирован с поддержкой SSL. Для готовых пакетов PostgreSQL так оно и есть. Кроме того, в файле `postgresql.conf` должен быть включен параметр `ssl = on`;
- `hostnossl`: относится к подключениям без использования SSL.

Ниже показан пример файла `pg_hba.conf`:

```
# ТИП БАЗА ПОЛЬЗОВАТЕЛЬ АДРЕС МЕТОД
# "local" - только для подключений через Unix-сокеты
local all all trust
# локальные соединения по IPv4:
host all all 127.0.0.1/32 trust
# локальные соединения по IPv6:
host all all ::1/128 trust
```

Мы видим три простых правила. Запись `local` означает, что все пользователи, обращающиеся к любой базе данных через Unix-сокеты, считаются заслуживающими доверия. Метод `trust` означает, что пользователь входит в систему

без пароля. Остальные два правила означают то же самое для подключений через localhost – по IPv4-адресу 127.0.0.1 или по IPv6-адресу ::1/128.

Поскольку подключение без пароля – очевидно, не лучший выбор для удаленного доступа, PostgreSQL предлагает различные методы аутентификации, которые задаются в файле `pg_hba.conf`. Ниже приведен их перечень:

- **trust:** аутентификация без пароля. Указанный пользователь должен существовать в PostgreSQL;
- **reject:** в подключении будет отказано;
- **md5 и password:** подключение с паролем; md5 означает, что пароль передается в зашифрованном виде, password – что в открытом виде, чего вообще нельзя допускать в современных системах. Метод md5 уже не считается безопасным. В PostgreSQL 10 и более поздних версиях следует использовать метод `scram-sha-256`;
- **scram-sha-256:** пришел на смену md5, обеспечивает улучшенную безопасность;
- **GSS и SSPI:** используется метод аутентификации GSSAPI или SSPI. Доступно только для подключений по протоколу TCP/IP и применяется для обеспечения единой точки входа;
- **ident:** получает имя пользователя в клиентской операционной системе путем обращения к серверу ident на стороне клиента и сравнивает с запрошенным именем пользователя базы данных;
- **peer:** предположим, что мы вошли в Unix под именем *abc*. Если указан метод peer, то в PostgreSQL мы тоже можем войти только как *abc*¹. При попытке указать другое имя пользователя в подключении будет отказано. Прелесть в том, что *abc* для аутентификации не нужен пароль, а идея заключается в том, что в базу данных в Unix-системе может войти только администратор базы данных, а не всякий, кто имеет учетную запись. Этот метод работает только для локальных подключений;
- **PAM:** используется **подключаемый модуль аутентификации (PAM)**. Это особенно важно, если вы собираетесь использовать метод аутентификации, отсутствующий в дистрибутиве PostgreSQL. Чтобы использовать метод PAM, создайте в своей Linux-системе файл `/etc/pam.d/postgresql` и укажите PAM-модули, которые планируете использовать, в конфигурационном файле. С помощью PAM можно аутентифицировать пользователей, например по каталогу Active Directory;
- **LDAP:** метод аутентификации по протоколу **LDAP (облегченный протокол доступа к каталогам)**. Отметим, что PostgreSQL запрашивает у LDAP только аутентификацию; если пользователь прописан в LDAP, но отсутствует в PostgreSQL, то в доступе будет отказано. Также отметим, что PostgreSQL должна знать, где находится сервер LDAP. Эта информа-

¹ Можно войти и под другим именем, если указать сопоставление имен. – Прим. ред.

ция должна храниться в файле `pg_hba.conf`, как описано в официальной документации по адресу <https://www.postgresql.org/docs/current/static/auth-ldap.html>¹;

- **RADIUS:** протокол **remote authentication dial-in user service (RADIUS)** – средства организации единой точки входа. Параметры также задаются в конфигурационном файле;
- **cert:** в этом методе аутентификации используются клиентские сертификаты SSL, поэтому он работает только для SSL-подключений. Преимущество в том, что не нужно передавать пароль. Атрибут сертификата `CN` сравнивается с запрошенным именем пользователя базы данных; если они совпадают, то вход разрешается. Для перекодирования имен пользователей можно использовать таблицу соответствия.

Правила записываются последовательно. Порядок имеет значение, как видно из следующего примера:

```
host all all 192.168.1.0/24 scram-sha-256
host all all 192.168.1.54/32 reject
```

Просматривая файл `pg_hba.conf`, PostgreSQL использует первое подходящее правило. Запрос, пришедший с адреса `192.168.1.54`, удовлетворяет первому правилу, а до второго даже дело не доходит. Следовательно, пользователь с адресом `192.168.1.54` сможет зайти в систему, если укажет правильное имя и пароль, а второе правило не имеет смысла. Если требуется исключить этот IP-адрес, то правила нужно поменять местами.

SSL-подключения

PostgreSQL позволяет шифровать обмен данными между клиентом и сервером. Шифрование очень полезно, особенно если данные передаются на большое расстояние. SSL предлагает простой и безопасный способ гарантировать, что посторонний не сможет вас подслушать. В этом разделе мы покажем, как настраивать SSL.

Прежде всего нужно задать значение `on` параметра `ssl` в файле `postgresql.conf`, который сервер читает на этапе запуска. Далее следует поместить сертификаты SSL в каталог, на который указывает переменная окружения `$PGDATA`. Если вы хотите поместить сертификаты в какой-то другой каталог, измените следующие параметры:

```
#ssl_cert_file = 'server.crt' # (после изменения необходимо перезапустить сервер)
#ssl_key_file = 'server.key'  # (после изменения необходимо перезапустить сервер)
#ssl_ca_file = ''             # (после изменения необходимо перезапустить сервер)
#ssl_crl_file = ''            # (после изменения необходимо перезапустить сервер)
```

Если вы готовы использовать самоподписанные сертификаты, выполните следующую команду:

```
openssl req -new -text -out server.req
```

¹ На русском языке <https://postgrespro.ru/docs/postgresql/11/auth-ldap>. – Прим. перев.

Ответьте на вопросы, которые задаст OpenSSL. В качестве общего имени (common name) укажите локальное имя узла. Пароль можете не задавать. В результате будет сгенерирован ключ, защищенный парольной фразой, которая должна быть не короче четырех символов.

Чтобы удалить парольную фразу (это необходимо, если вы собираетесь запускать сервер в автоматическом режиме), выполните следующие команды:

```
openssl rsa -in privkey.pem -out server.key
rm privkey.pem
```

Чтобы разблокировать ключ, введите старую парольную фразу. Далее, чтобы сделать сертификат самоподписанным и скопировать его и ключ туда, где их сможет найти сервер, выполните команду:

```
openssl req -x509 -in server.req -text -key server.key -out server.crt
```

Затем задайте права доступа к файлам, как показано ниже:

```
chmod og-rwx server.key
```

Записав все необходимое в файл postgresql.conf, мы можем подключиться к серверу по SSL. Чтобы убедиться в том, что SSL действительно используется, воспользуемся представлением pg_stat_ssl. Она сообщит обо всех подключениях и, в частности, о том, используется SSL или нет. Кроме того, выдается важная информация о применяемом шифровании:

```
test=# \d pg_stat_ssl
Представление "pg_catalog.pg_stat_ssl"
  Столбец   | Тип      | Модификаторы
```

-----+	-----+	-----+
pid	integer	
ssl	boolean	
version	text	
cipher	text	
bits	integer	
compression	boolean	
clientdn	text	

Если поле ssl равно true, то PostgreSQL ведет себя ожидаемым образом:

```
postgres=# SELECT * FROM pg_stat_ssl;
-[ RECORD 1 ]-----
pid      | 20075
ssl       | t
version   | TLSv1.2
cipher    | ECDHE-RSA-AES256-GCM-SHA384
bits      | 256
compression | f
clientdn  |
```

Безопасность на уровне экземпляра

Пока что мы описали настройку адресов привязки и аутентификацию по диапазонам IP-адресов. Все это части конфигурации, относящиеся к сети.

На следующем этапе мы обратимся к правам на уровне экземпляра. Главное, о чем нужно помнить, – что пользователи в PostgreSQL определены на уровне экземпляра. Созданный пользователь виден не в одной, а во всех базах данных. У него могут быть права доступа только к одной базе, но существует он все равно на уровне экземпляра.

Тем, кто раньше не работал с PostgreSQL, следует иметь в виду еще одну вещь: пользователи и роли – одно и то же. У команд CREATE ROLE и CREATE USER разные значения по умолчанию (собственно, единственное различие состоит в том, что у роли по умолчанию нет атрибута LOGIN), но по существу пользователи и роли – одна и та же концепция. Поэтому и синтаксис команд CREATE ROLE и CREATE USER одинаков:

```
test=# \h CREATE USER
```

Команда: CREATE USER

Описание: создать роль в базе данных

Синтаксис:

```
CREATE USER имя [ WITH ] параметр [ ... ] ]
```

где допустимые параметры:

```
SUPERUSER | NOSUPERUSER
| CREATEDB | NOCREATEDB
| CREATEROLE | NOCREATEROLE
| INHERIT | NOINHERIT
| LOGIN | NOLOGIN
| REPLICATION | NOREPLICATION
| BYPASSRLS | NOBYPASSRLS
| CONNECTION LIMIT предел_подключений
| [ ENCRYPTED ] PASSWORD 'пароль'
| VALID UNTIL 'timestamp'
| IN ROLE имя_роли [, ...]
| IN GROUP имя_роли [, ...]
| ROLE имя_роли [, ...]
| ADMIN имя_роли [, ...]
| USER имя_роли [, ...]
| SYSID uid
```

Прежде всего отметим, что бывают обычные пользователи и суперпользователи. Если задан атрибут SUPERUSER, то ограничения, налагаемые на обычного пользователя, не действуют. Суперпользователь может удалять любые объекты (базы данных и т. д.), какие захочет.

Важный момент: для создания базы данных у пользователя должны быть права на уровне экземпляра.



Отметим, что пользователь, создавший базу данных, автоматически становится ее владельцем.

Общее правило таково: создатель объекта является его владельцем (если явно не указано противное, как можно сделать в команде `CREATE DATABASE`). Владелец объекта может его удалить.

i Фраза `CREATOROLE/NOCREATOROLE` определяет, разрешено ли пользователю создавать других пользователей и роли.

Следующая важная фраза – `INHERIT/NOINHERIT`. Если `INHERIT` присутствует (по умолчанию это так), то пользователь может наследовать права от какого-то другого пользователя. Наследование позволяет использовать роли – удобный способ абстрагирования прав. Например, можно создать роль `bookkeeper` (бухгалтер), которой будут наследовать многие другие роли. Смысл в том, что мы только один раз говорим PostgreSQL, что разрешено делать роли `bookkeeper`, даже если в бухгалтерии работает много людей.

Фраза `LOGIN/NOLOGIN` определяет, разрешено ли роли подключаться к экземпляру.

i Отметим, что одной лишь фразы `LOGIN` недостаточно для подключения к базе данных. Для этого нужны дополнительные права.

Экземпляр – это, по существу, портал ко всем находящимся внутри него базам данных. Вернемся к нашему примеру: роль `bookkeeper` можно пометить атрибутом `NOLOGIN`, потому что мы хотим, чтобы каждый пользователь заходил в систему под своим настоящим именем. Бухгалтеры (скажем, Джо и Джейн) будут помечены атрибутом `LOGIN`, но наследуют все права от роли `bookkeeper`. Подобная структура гарантирует, что у всех бухгалтеров будут одинаковые права, хотя заходить они будут под разными именами.

Если мы планируем эксплуатировать PostgreSQL в режиме потоковой репликации, то можем производить потоковую обработку журнала транзакций от имени суперпользователя. Но с точки зрения безопасности это нежелательно. Поэтому PostgreSQL разрешает предоставлять право репликации обычному пользователю, который и будет отвечать за потоковую обработку. Очень часто для этой цели создают специального пользователя.

Как мы увидим ниже в этой главе, PostgreSQL обеспечивает безопасность на уровне строк. Смысл ее в том, что некоторые строки можно сделать невидимыми пользователю. Если пользователю явно разрешено обходить этот механизм, то следует установить атрибут `BYPASSRLS`. По умолчанию предполагается режим `NOBYPASSRLS`.

Иногда имеет смысл ограничить количество соединений, доступных пользователю. Для этого предназначена фраза `CONNECTION LIMIT`. Отметим, что общее число соединений никогда не может быть больше заданного в файле `postgresql.conf` (параметр `max_connections`). Но для некоторых пользователей можно задать меньшее значение.

Зачастую мы заранее знаем, что некий пользователь скоро покинет организацию. Фраза `VALID UNTIL` позволяет автоматически блокировать его учетную запись по достижении указанной даты.

Во фразе `IN ROLE` перечисляются существующие роли, в которые новая роль будет добавлена в качестве члена. Это позволяет избежать дополнительных шагов. Синонимом является фраза `IN GROUP`.

Во фразе `ROLE` перечисляются роли, которые будут добавлены в новую роль.

Фраза `ADMIN` – то же, что `ROLE`, но с добавлением `WITH ADMIN OPTION`.

Наконец, фраза `SYSID` позволяет задать конкретный идентификатор пользователя (по аналогии с тем, что некоторые администраторы Unix делают на уровне операционной системы).

Создание и модификация пользователей

После этого теоретического введения самое время заняться созданием пользователей и посмотреть, как все это выглядит на практике:

```
test=# CREATE ROLE bookkeeper NOLOGIN;
CREATE ROLE
test=# CREATE ROLE joe LOGIN;
CREATE ROLE
test=# GRANT bookkeeper TO joe;
GRANT ROLE
```

Сначала мы создаем роль `bookkeeper`. Поскольку мы не хотим, чтобы кто-то входил от имени пользователя `bookkeeper`, эта роль помечена признаком `NOLOGIN`.



Отметим также, что `NOLOGIN` подразумевается по умолчанию в случае, когда используется команда `CREATE ROLE`. В команде `CREATE USER` по умолчанию подразумевается режим `LOGIN`.

Затем создается роль `joe` с атрибутом `LOGIN`. И наконец, роли `joe` передаются все права роли `bookkeeper`, так что теперь она может делать все, что разрешено `bookkeeper`'у.

Создав пользователей, проверим, что получилось:

```
[hs@zenbook ~]$ psql test -U bookkeeper
psql: ВАЖНО: для роли "bookkeeper" вход запрещен
```

Как и следовало ожидать, роли `bookkeeper` запрещено входить в систему. А что, если попробовать войти в систему от имени `joe`?

```
[hs@zenbook ~]$ psql test -U joe
...
test=>
```

Так и должно быть. Заметим, однако, что приглашение изменилось – так PostgreSQL показывает, что вошедший – не суперпользователь.

Иногда описание пользователя нужно модифицировать, например изменить пароль. В PostgreSQL пользователи могут сами менять свои пароли, например:

```
test=> ALTER ROLE joe PASSWORD 'abc';
ALTER ROLE
test=> SELECT current_user;
current_user
-----
joe
(1 строка)
```

Команда ALTER ROLE (или ALTER USER) позволяет изменить большую часть параметров, заданных в момент создания пользователя. Но этим управление пользователями не исчерпывается. Зачастую мы хотим ассоциировать с пользователем специальные параметры. ALTER USER предоставляет такую возможность:

```
ALTER ROLE { указание_роли | ALL } [ IN DATABASE имя_базы_данных ]
    SET параметр_конфигурации { TO | = } { значение | DEFAULT }
ALTER ROLE { указание_роли | ALL } [ IN DATABASE имя_базы_данных ]
    SET параметр_конфигурации FROM CURRENT
ALTER ROLE { указание_роли | ALL }
    RESET параметр_конфигурации
ALTER ROLE { указание_роли | ALL } RESET ALL
```

Синтаксис довольно простой. Чтобы вы поняли, зачем это нужно, я приведу пример из реальной жизни. Предположим, что Джо живет на острове Маврикий. Он хочет работать в системе в своем поясное время, даже если сервер базы данных находится в Европе.

```
test=> ALTER ROLE joe SET TimeZone = 'UTC-4';
ALTER ROLE

test=> SELECT now();
now
-----
2017-01-09 20:36:48.571584+01
(1 строка)

test=> \q
[hs@zenbook ~]$ psql test -U joe
...
test=> SELECT now();
now
-----
2017-01-09 23:36:53.357845+04
(1 строка)
```

Команда ALTER ROLE модифицирует описание пользователя. Когда joe снова войдет в систему, будет установлено его поясное время.



Часовой пояс изменяется не сразу, а только при повторном входе или после выполнения команды `SET ... TO DEFAULT`.

Точно так же можно изменить некоторые параметры памяти, которые рассматривались в книге ранее, например `work_mem`.

Задание безопасности на уровне базы данных

Задав пользователей на уровне экземпляра, мы можем пойти дальше и посмотреть, что можно сделать на уровне базы данных. Сразу возникает вопрос: мы разрешили Джо заходить на экземпляр сервера, но кто или что позволило Джо подключаться к конкретной базе данных? Может быть, мы не хотим, чтобы Джо мог подключаться к любой из имеющихся баз данных. Ограничение доступа к определенным базам – это именно то, что делается на этом уровне.

Для баз данных команда `GRANT` позволяет задать следующие права:

```
GRANT { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...]
        | ALL [ PRIVILEGES ] }
ON DATABASE имя_базы_данных [, ...]
TO указание_роли [, ...] [ WITH GRANT OPTION ]
```

Два основных права на уровне базы данных заслуживают особого внимания:

- **CREATE**: позволяет создавать схемы внутри базы данных. Отметим, что **CREATE** не разрешает создавать таблицы, это право распространяется только на схемы. В PostgreSQL любая таблица принадлежит какой-то схеме, поэтому, чтобы создать таблицу, нужно сначала получить доступ к схеме;
- **CONNECT**: позволяет подключаться к базе данных.

Теперь вопрос: никто явно не выдавал право **CONNECT** роли `joe`, так каким же образом он его получил? Ответ таков: имеется такая штука – `public`, аналогичная тому, что в Unix называется «миром» (`world`). Если что-то разрешено всему миру, то разрешено и `joe`, являющемуся частью мира.

Важно, что `public` не является ролью в том смысле, что ее нельзя удалить или переименовать. Можно просто считать, что это эквивалент «всех пользователей системы».

Поэтому если мы не хотим, чтобы любой пользователь мог в любой момент подключиться к любой базе данных, мы должны отозвать право **CONNECT** у `public`. Для этого следует войти от имени суперпользователя и исправить положение:

```
[hs@zenbook ~]$ psql test -U postgres
...
test=# REVOKE ALL ON DATABASE test FROM public;
REVOKE
test=# \q
[hs@zenbook ~]$ psql test -U joe
psql: ВАЖНО: доступ к базе "test" запрещен
ПОДРОБНОСТИ: Пользователь не имеет привилегии CONNECT.
```

Как видим, роль `joe` больше не может подключиться. Сейчас к базе `test` имеют доступ только суперпользователи.

Вообще говоря, рекомендуется отзывать права доступа в базе данных postgres еще до создания других баз. Смысл в том, что тогда отозванные права не будут предоставляться во вновь создаваемых базах¹. Если кому-то потребуется доступ к определенной базе, то его можно будет предоставить явно. Если мы пожелаем разрешить joe подключаться к базе данных test, то нужно будет выполнить такие команды от имени суперпользователя:

```
[hs@zenbook ~]$ psql test -U postgres
...
test=# GRANT CONNECT ON DATABASE test TO bookkeeper;
GRANT
test=# \q
[hs@zenbook ~]$ psql test -U joe
...
test=>
```

У нас есть два варианта действий:

- дать право самому пользователю joe, так что только он и сможет подключиться;
- дать право роли bookkeeper. Напомним, что пользователь joe наследует все права роли bookkeeper, поэтому если мы хотим, чтобы все бухгалтеры могли подключаться к этой базе, то идея предоставить права роли bookkeeper выглядит привлекательно.

Предоставление прав роли bookkeeper безопасно, потому что ей самой вообще запрещено подключаться к экземпляру, т. е. она является лишь источником прав.

Задание прав на уровне схемы

Закончив настройку на уровне базы данных, мы можем перейти к уровню схемы. Но сначала выполним простенький тест:

```
test=> CREATE DATABASE test;
ОШИБКА: нет прав на создание базы данных
test=> CREATE USER xy;
ОШИБКА: нет прав для создания роли
test=> CREATE SCHEMA sales;
ОШИБКА: нет доступа к базе данных test
```

Как видим, у joe сегодня плохой день – ему не разрешено ничего, кроме подключения к базе данных.

Впрочем, есть одно исключение, которое многих несказанно удивляет:

```
test=> CREATE TABLE t_broken (id int);
CREATE TABLE
test=> \d
      Список отношений
  Схема |   Имя   | Тип  | Владелец
-----+-----+-----+-----
 public | t_broken | table | joe
(1 строка)
```

¹ Это не так: при создании новой базы данных public автоматически получает к ней доступ в любом случае. – Прим. ред.

По умолчанию квазироли `public` разрешено работать со схемой `public`, которая присутствует всегда. Если вы серьезно относитесь к безопасности своей базы, то эту проблему нужно решить. Иначе обычные пользователи замусорят схему `public` всякими таблицами, от чего может пострадать вся система. Кроме того, помните, что если пользователю разрешено создавать объект, то он становится его владельцем. Владение означает, что создателю автоматически предоставляются все права, в т. ч. и право удалить объект.

Чтобы отозвать эти права у `public`, выполните следующую команду от имени суперпользователя:

```
test=# REVOKE ALL ON SCHEMA public FROM public;
REVOKE
```

Теперь никто не сможет создавать объекты в схеме `public` без явного разрешения. Что и доказывает следующая распечатка:

```
[hs@zenbook ~]$ psql test -U joe
...
test=> CREATE TABLE t_data (id int);
ОШИБКА: схема для создания объектов не выбрана
СТРОКА 1: CREATE TABLE t_data (id int);
```

Как видим, команда не выполнена. Интересно, как сформулировано сообщение об ошибке: PostgreSQL не знает, куда поместить таблицу. По умолчанию она пытается поместить таблицу в одну из следующих схем:

```
test=> SHOW search_path ;
search_path
-----
"$user", public
(1 строка)
```

Поскольку схемы `joe` не существует, PostgreSQL пробует схему `public`. Но на создание таблицы в ней нет прав, поэтому СУБД жалуется, что не знает, где создать таблицу.

Но стоит явно указать префикс таблицы, как ситуация мгновенно меняется:

```
test=> CREATE TABLE public.t_data (id int);
ОШИБКА: нет доступа к схеме public
СТРОКА 1: CREATE TABLE public.t_data (id int);
```

Вот теперь мы получили ожидаемое сообщение. PostgreSQL запрещает доступ к схеме `public`.

Следующий вопрос: какие права можно задать на уровне схемы, чтобы пользователь `joe` мог хоть что-нибудь делать?

```
GRANT { { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
ON SCHEMA имя_схемы [, ...]
TO указание_роли [, ...] [ WITH GRANT OPTION ]
```

`CREATE` означает, что пользователь может создавать в схеме объекты. `USAGE` означает, что пользователю разрешено войти в схему. Но это еще не значит,

что пользователь сможет что-то из находящего в этой схеме использовать; соответствующие права еще не предоставлены. Пока что пользователю разрешено только просматривать эту схему в системном каталоге.

Чтобы пользователь joe мог получить доступ к созданной ранее таблице, необходимо выполнить следующую команду (от имени суперпользователя):

```
test=# GRANT USAGE ON SCHEMA public TO bookkeeper;
GRANT
```

Теперь joe может читать таблицу¹:

```
[hs@zenbook ~]$ psql test -U joe
test=> SELECT count(*) FROM t_broken;
 count
-----
      0
(1 строка)
```

Пользователь joe может также добавлять и модифицировать строки, потому что является владельцем этой таблицы. Но хотя joe уже может многое, он еще не всемогущ. Рассмотрим следующую команду:

```
test=> ALTER TABLE t_broken RENAME TO t_useful;
ОШИБКА: нет доступа к схеме public
```

Приглядимся к сообщению об ошибке. Система жалуется на отсутствие доступа к схеме, а не к самой таблице (напомним, joe владеет этой таблицей). Поэтому и устранять проблему нужно на уровне схемы, а не таблицы. Выполните следующую команду от имени суперпользователя:

```
test=# GRANT CREATE ON SCHEMA public TO bookkeeper;
GRANT
```

Теперь joe может изменить имя таблицы:

```
[hs@zenbook ~]$ psql test -U joe
test=> ALTER TABLE t_broken RENAME TO t_useful;
ALTER TABLE
```

Описанное выше действие необходимо, если вы собираетесь использовать DDL-команды. В своей работе по предоставлению поддержки пользователям PostgreSQL я несколько раз встречался с ситуациями, когда это оказывалось проблемой.

Работа с таблицами

Итак, мы разобрались с адресами привязки, сетевой аутентификацией, пользователями, базами данных и схемами. Теперь, наконец, мы добрались до уровня таблиц. Ниже показано, какие можно предоставить права доступа к таблице:

¹ Выше в тексте таблица `t_broken` не создавалась, поэтому эта и следующие команды закончатся ошибкой. Следует либо создать таблицу `t_broken`, либо заменить имя на `t_data`. – Прим. перев.

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE
        | REFERENCES | TRIGGER }
        [, ...] | ALL [ PRIVILEGES ] }
ON { [ TABLE ] имя_таблицы [, ...]
     | ALL TABLES IN SCHEMA имя_схемы [, ...] }
TO указание_роли [, ...] [ WITH GRANT OPTION ]
```

Рассмотрим эти права поочередно.

- SELECT: разрешено читать таблицу.
- INSERT: разрешено добавлять строки в таблицу (включает также команду COPY и другие, а не только команду INSERT). Отметим, что разрешение вставлять строки еще не означает, что таблицу разрешено читать. Чтобы вставленные строки можно было прочитать, необходимы оба права, SELECT и INSERT.
- UPDATE: разрешено модифицировать содержимое таблицы.
- DELETE: разрешено удалять строки из таблицы.
- TRUNCATE: разрешено использовать команду TRUNCATE. Заметим, что DELETE и TRUNCATE – два разных права, потому что команда TRUNCATE блокирует всю таблицу, чего команда DELETE не делает (даже в отсутствие условия WHERE).
- REFERENCES: разрешено создавать внешние ключи. Это право необходимо иметь для обоих столбцов: того, который ссылается, и того, на который он ссылается. В противном случае при попытке создать ключ возникнет ошибка.
- TRIGGER: разрешено создание триггеров.



Команда GRANT хороша еще и тем, что позволяет сразу выдать права на все таблицы в схеме, что существенно упрощает процедуру настройки прав.

В команде GRANT может быть также фраза WITH GRANT OPTION. Идея в том, чтобы позволить обычным пользователям предоставлять права другим, что несколько снижает нагрузку на администратора. Представьте систему, в которой нужно выдавать права сотням пользователей, – это занятие способно поглотить все время, поэтому администратор может переложить часть ответственности на некоторых доверенных пользователей.

Задание прав на уровне столбцов

Иногда не всем пользователям разрешено просматривать все данные. Рассмотрим банк. Некоторым сотрудникам разрешено видеть всю информацию о банковском счете, тогда как остальным – лишь ее подмножество. На практике не всем разрешено видеть остаток на счете или процентную ставку по кредиту.

Другой пример – пользователям разрешено видеть профили других людей, но не их фотографии и прочие персональные данные. Вопрос: как можно воспользоваться безопасностью на уровне столбцов?

Для демонстрации добавим столбец в существующую таблицу, принадлежащую роли joe:

```
test=> ALTER TABLE t_useful ADD COLUMN name text;
ALTER TABLE
```

Теперь таблица содержит два столбца. Цель примера – сделать так, чтобы пользователь мог видеть только один из этих столбцов:

```
test=> \d t_useful
        Таблица "public.t_useful"
  Столбец |  Тип  | Модификаторы
-----+-----+-----
   id     | integer |
   name   | text    |
```

От имени суперпользователя создадим нового пользователя и предоставим ему доступ к схеме, содержащей таблицу:

```
test=# CREATE ROLE paul LOGIN;
CREATE ROLE
test=# GRANT CONNECT ON DATABASE test TO paul;
GRANT
test=# GRANT USAGE ON SCHEMA public TO paul;
GRANT
```

Не забудьте дать новому пользователю право `CONNECT`, потому что выше в этой главе это право было у `public` отозвано. Без явного предоставления `paul` даже не сможет добраться до таблицы.

Дадим роли `paul` право `SELECT`:

```
test=# GRANT SELECT (id) ON t_useful TO paul;
GRANT
```

Этого уже достаточно. Теперь можно подключиться к базе от имени пользователя `paul` и прочитать столбец:

```
[hs@zenbook ~]$ psql test -U paul
...
test=> SELECT id FROM t_useful;
 id
----
(0 строк)
```

При использовании прав доступа к столбцам важно иметь в виду, что про `SELECT *` следует забыть, т. к. эта конструкция работать не будет:

```
test=> SELECT * FROM t_useful;
ОШИБКА: нет доступа к таблице t_useful
```

Символ `*` по-прежнему означает «все столбцы», но получить доступ ко всем столбцам нельзя, поэтому и возникает ошибка.

Задание привилегий по умолчанию

Итак, мы научились настраивать различные аспекты безопасности. Но что, если в систему будут добавлены новые таблицы? Утомительно и рискованно обрабатывать их поодиночке, устанавливая нужные права. Вот было бы хоро-

шо, если бы все это происходило автоматически! Именно для этого предназначена команда ALTER DEFAULT PRIVILEGES. Смысл ее в том, чтобы предоставить пользователям средства автоматически задавать желательные права сразу после появления объекта на свет. Тогда будет попросту невозможно забыть про установку прав.

Ниже показана первая часть синтаксического описания:

```
postgres=# \h ALTER DEFAULT PRIVILEGES
Команда: ALTER DEFAULT PRIVILEGES
Описание: определить права доступа по умолчанию
Синтаксис:
ALTER DEFAULT PRIVILEGES
  [ FOR { ROLE | USER } целевая_роль [, ...] ]
  [ IN SCHEMA имя_схемы [, ...] ]
  предложение_grant_или_revoke
```

где предложение_grant_или_revoke может быть следующим:

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
[, ...] | ALL [ PRIVILEGES ] }
ON TABLES
TO { [ GROUP ] имя_роли | PUBLIC } [, ...] [ WITH GRANT OPTION ]
...
```

Синтаксис такой же, как в команде GRANT, поэтому сложностей при использовании не возникает. Для демонстрации я подготовил простой пример: если таблицу создает joe, то paul автоматически сможет ей пользоваться:

```
test=# ALTER DEFAULT PRIVILEGES FOR ROLE joe
      IN SCHEMA public GRANT ALL ON TABLES TO paul;
ALTER DEFAULT PRIVILEGES
```

Подключимся от имени joe и создадим таблицу:

```
[hs@zenbook ~]$ psql test -U joe
...
test=> CREATE TABLE t_user (id serial, name text, passwd text);
CREATE TABLE
```

Теперь, подключившись от имени роли paul, мы убедимся, что таблице назначен нужный набор прав:

```
[hs@zenbook ~]$ psql test -U paul
...
test=> SELECT * FROM t_user;
 id | name | passwd
-----+-----+-----
(0 строк)
```

БЕЗОПАСНОСТЬ НА УРОВНЕ СТРОК

До сих пор мы рассматривали таблицу как единое целое. Если таблица содержала миллион строк, то мы могли выбрать их все. Всякий, кто имеет право

читать таблицу, может прочитать ее целиком. Но во многих случаях этого недостаточно и требуется, чтобы пользователю было разрешено читать не все строки.

Рассмотрим пример из реальной жизни. Бухгалтерия ведет учет деятельности многих сотрудников. Таблица с налоговыми ставками должна быть видна всем, поскольку всегда применяются одни и те же ставки. Но что касается отдельных операций, каждому человеку разрешено видеть только свои операции. А не должен видеть данные В. Кроме того, руководитель подразделения должен иметь возможность видеть все данные, относящиеся к подведомственной ему части компании.

Безопасность на уровне строк предназначена для решения именно таких задач и позволяет без труда строить многоарендные системы. Для задания прав в этом случае применяются политики, создаваемые командой CREATE POLICY:

```
test=# \h CREATE POLICY
Команда: CREATE POLICY
Описание: создать новую политику защиты на уровне строк для таблицы
Синтаксис:
CREATE POLICY имя ON имя_таблицы
[ AS { PERMISSIVE | RESTRICTIVE } ]
[ FOR { ALL | SELECT | INSERT | UPDATE | DELETE } ]
[ TO { имя_роли | PUBLIC | CURRENT_USER | SESSION_USER } [, ...] ]
[ USING ( выражение_using ) ]
[ WITH CHECK ( выражение_check ) ]
```

Чтобы понять, как записывается политика, войдем сначала от имени суперпользователя и создадим таблицу, содержащую несколько записей:

```
test=# CREATE TABLE t_person (gender text, name text);
CREATE TABLE
test=# INSERT INTO t_person
VALUES ('male', 'joe'),
      ('male', 'paul'),
      ('female', 'sarah'),
      (NULL, 'R2-D2');
INSERT 0 4
```

Теперь предоставим доступ к ней роли joe:

```
test=# GRANT ALL ON t_person TO joe;
GRANT
```

Пока что все идет как обычно – joe может читать все таблицу, поскольку безопасность на уровне строк не активирована. А теперь посмотрим, что случится, если включить режим ROW LEVEL SECURITY для этой таблицы:

```
test=# ALTER TABLE t_person ENABLE ROW LEVEL SECURITY;
ALTER TABLE
```

По умолчанию действует политика запрета доступа ко всему, поэтому joe увидит пустую таблицу:


```
test=> SELECT * FROM t_person;
gender | name
-----+-----
(0 строк)
```

Такая политика по умолчанию не лишена смысла, потому что заставляет пользователей явно задать разрешения.

После того как безопасность на уровне строк активирована, суперпользователь может создавать политики:

```
test=# CREATE POLICY joe_pol_1
ON t_person
FOR SELECT TO joe
USING (gender = 'male');
CREATE POLICY
```

Если теперь зайти от имени joe и выбрать все данные, то мы увидим только две строки:

```
test=> SELECT * FROM t_person;
gender | name
-----+-----
male   | joe
male   | paul
(2 строки)
```

Рассмотрим только что созданную политику более внимательно. Прежде всего у политики есть имя. Кроме того, она связана с таблицей и разрешает некоторые операции (в данном случае SELECT). Затем идет фраза USING. Она определяет, что разрешено видеть роли joe. Таким образом, фраза USING задает фильтр, принудительно присоединяемый к каждому запросу и ограничивающий множество строк, видимых пользователю.

Отметим еще один важный момент: если политик несколько, то PostgreSQL соединяет их связкой OR. То есть по умолчанию чем больше политик, тем больше данных вы сможете увидеть. В версии PostgreSQL 9.6 так было всегда. Но в версии PostgreSQL 10.0 пользователь может явно указать одну из связок OR или AND:

PERMISSIVE | RESTRICTIVE

По умолчанию подразумевается режим PERMISSIVE, т. е. выбирается связка OR. Если же задан режим RESTRICTIVE, то фильтры соединяются связкой AND.

Предположим теперь, что joe разрешено видеть также роботов. Для решения этой задачи у нас есть два пути. Первый – изменить существующую политику командой ALTER POLICY:

```
postgres=# \h ALTER POLICY
Команда: ALTER POLICY
Описание: изменить определение политики защиты на уровне строк
Синтаксис:
ALTER POLICY имя ON имя_таблицы RENAME TO новое_имя
```

```
ALTER POLICY имя ON имя_таблицы
[ TO { имя_роли | PUBLIC | CURRENT_USER | SESSION_USER } [, ...] ]
[ USING ( выражение_using ) ]
[ WITH CHECK ( выражение_check ) ]
```

Второй путь – создать еще одну политику:

```
test=# CREATE POLICY joe_pol_2
ON t_person
FOR SELECT TO joe
USING (gender IS NULL);
CREATE POLICY
```

Как было сказано выше, обе политики соединяются связкой OR, если не задано ключевое слово RESTRICTIVE. Поэтому теперь PostgreSQL вернет не две, а три строки:

```
test=> SELECT * FROM t_person;
gender | name
-----+-----
male   | joe
male   | paul
        | R2-D2
(3 строки)
```

В состав результата теперь входит также роль R2-D2, поскольку она соответствует второй политике. Чтобы показать, как PostgreSQL выполняет этот запрос, я включил план выполнения:

```
test=> EXPLAIN SELECT * FROM t_person;
               QUERY PLAN
-----
Seq Scan on t_person (cost=0.00..21.00 rows=9 width=64)
  Filter: ((gender IS NULL) OR (gender = 'male'::text))
(2 строки)
```

Как видим, обе фразы USING превратились в принудительно добавленные фильтры¹. Заметим, что в синтаксическом описании есть две разные фразы:

- USING: задает фильтрацию уже существующих строк и применяется к командам SELECT, UPDATE и т. п.;
- CHECK: применяется к новым строкам, создаваемым командами INSERT, UPDATE и т. п.

Вот что произойдет при попытке вставить строку:

```
test=> INSERT INTO t_person VALUES ('male', 'kaarel');
ОШИБКА: новая строка нарушает политику защиты на уровне строк для таблицы "t_person"
```

Поскольку для команды INSERT нет политики, мы естественно получаем ошибку. Ниже приведена политика, разрешающая вставку:

¹ Применение политик не сводится к простому добавлению фильтров: условия политик проверяются раньше обычных условий, чтобы предотвратить утечку защищаемых данных через пользовательские функции. – *Прим. ред.*

```
test=# CREATE POLICY joe_pol_3
        ON t_person
        FOR INSERT TO joe
        WITH CHECK (gender IN ('male', 'female'));
CREATE POLICY
```

Роли joe разрешено добавлять в таблицу записи о мужчинах и женщинах:

```
test=> INSERT INTO t_person VALUES ('female', 'maria');
INSERT 0 1
```

Но есть подвох, который иллюстрируется следующим примером:

```
test=> INSERT INTO t_person VALUES ('female', 'maria') RETURNING *;
ОШИБКА: новая строка нарушает политику защиты на уровне строк для таблицы "t_person"
```

Напомним, что определена только политика для выбора записей о мужчинах. Проблема в том, что эта команда возвращает запись о женщине, что запрещено.

Фраза RETURNING * будет работать, только если вставлена запись о мужчине:

```
test=> INSERT INTO t_person VALUES ('male', 'max') RETURNING *;
gender | name
-----+-----
male   | max
(1 строка)
INSERT 0 1
```

Если такое поведение нежелательно, то нужно написать политику с подходящей фразой USING.

ПРОСМОТР ПРАВ

Иногда необходимо узнать, кто какими правами наделен. Администраторам необходимо знать, кто что может делать. К сожалению, это непростая процедура, требующая кое-каких знаний. Вообще-то, я предпочитаю пользоваться командной строкой. Но в случае системы прав имеет смысл прибегнуть к графическому интерфейсу.

Прежде чем продемонстрировать чтение прав в PostgreSQL, назначим права роли joe, чтобы на следующем шаге их можно было просмотреть:

```
test=# GRANT ALL ON t_person TO joe;
GRANT
```

Для получения информации о правах пригодится команда \z в psql:

```
test=# \x
Расширенный вывод включен.
test=# \z t_person
Права доступа
-[ RECORD 1 ]-----
Схема          | public
Имя            | t_person
```

Тип	table	
Права доступа	postgres=arwdDxt/postgres	+
	joe=arwdDxt/postgres	
Права для столбцов		
Политики	joe_pol_1 (r):	+
	(u): (gender = 'male'::text)	+
	to: joe	+
	joe_pol_2 (r):	+
	(u): (gender IS NULL)	+
	to: joe	+
	joe_pol_3 (a):	+
	(c): (gender = ANY (ARRAY['male'::text, 'female'::text]))+	
	to: joe	

Мы видим все политики, а также сведения о правах доступа. К сожалению, эти сокращения трудно читать, и у меня такое чувство, что не все администраторы их понимают. В этом примере роль joe получила от PostgreSQL права arwdDxt. Что это означает?

- а: сокращение от «appends», относится к команде INSERT.
- r: сокращение от «reads», относится к команде SELECT.
- w: сокращение от «writes», относится к команде UPDATE.
- d: сокращение от «deletes», относится к команде DELETE.
- D: относится к команде TRUNCATE (когда это сокращение вводилось, буква t уже была занята).
- x: относится к ссылкам (внешним ключам).
- t: относится к триггерам.

Если вы не можете запомнить эти коды, есть второй способ получить информацию в более удобочитаемом виде:

```
test=# SELECT * FROM aclexplode('{joe=arwdDxt/postgres}');
 grantor | grantee | privilege_type | is_grantable
```

10		18481	INSERT	f
10		18481	SELECT	f
10		18481	UPDATE	f
10		18481	DELETE	f
10		18481	TRUNCATE	f
10		18481	REFERENCES	f
10		18481	TRIGGER	f

(7 строк)

Как видим, набор прав возвращен в виде простой таблицы, которая существенно облегчает жизнь.

ПЕРЕДАЧА ОБЪЕКТОВ И УДАЛЕНИЕ ПОЛЬЗОВАТЕЛЕЙ

Бывает, что из системы нужно удалить пользователей. Неудивительно, что для этой цели предназначены команды DROP ROLE и DROP USER:

```
test=# \h DROP ROLE
```

Команда: DROP ROLE

Описание: удалить роль пользователя БД

Синтаксис:

```
DROP ROLE [ IF EXISTS ] имя [, ...]
```

Попробуем.

```
test=# DROP ROLE joe;
```

ОШИБКА: роль "joe" нельзя удалить, потому что есть зависящие от нее объекты

ПОДРОБНОСТИ: субъект политики joe_pol_3 для таблицы t_person

субъект политики joe_pol_2 для таблицы t_person

субъект политики joe_pol_1 для таблицы t_person

права доступа к таблице t_person

владелец таблицы t_user

владелец последовательности t_user_id_seq

владелец прав по умолчанию на новые отношения, принадлежащие роли joe в схеме public

владелец таблицы t_useful

PostgreSQL выдает сообщения об ошибках, потому что пользователя можно удалить, только после того как у него отображено все. И в этом есть смысл: представьте, что некто владеет таблицей. Что PostgreSQL с этой таблицей делать? Кому-то же она должна принадлежать.

Чтобы передать таблицы от одного пользователя другому, воспользуемся командой REASSIGN:

```
test=# \h REASSIGN
```

Команда: REASSIGN OWNED

Описание: сменить владельца объектов базы данных, принадлежащих заданной роли

Синтаксис:

```
REASSIGN OWNED BY { старая_роль | CURRENT_USER | SESSION_USER } [, ...]
```

```
TO { новая_роль | CURRENT_USER | SESSION_USER }
```

Синтаксис снова очень простой, облегчающий процедуру передачи. Например:

```
test=# REASSIGN OWNED BY joe TO postgres;
```

```
REASSIGN OWNED
```

Попробуем снова удалить роль joe:

```
test=# DROP ROLE joe;
```

ОШИБКА: роль "joe" нельзя удалить, потому что есть зависящие от нее объекты

ПОДРОБНОСТИ: субъект политики joe_pol_3 для таблицы t_person

субъект политики joe_pol_2 для таблицы t_person

субъект политики joe_pol_1 для таблицы t_person

права доступа к таблице t_person

владелец прав по умолчанию на новые отношения, принадлежащие роли joe в схеме public

Как видим, перечень проблем значительно сократился. Нам нужно разрешить их поочередно, а затем удалить роль. Ни о каком коротком пути мне неизвестно. Повысить эффективность можно только одним способом – выдавать как можно меньше прав конкретным людям. Постарайтесь сосредоточить их в ролях, которым могут наследовать многие пользователи.

РЕЗЮМЕ

Безопасность базы данных – обширная тема, и на 30 страницах вряд ли можно охватить все аспекты безопасности в PostgreSQL. Мы даже не касались таких вещей, как SELinux, безопасность вызывающего и определившего и многое другое. Но мы узнали о большинстве типичных проблем, с которыми сталкиваются разработчики и администраторы баз данных. Мы также рассказали, как избежать основных ловушек и сделать системы более безопасными.

В главе 9 мы будем изучать резервное копирование и восстановление.

Вопросы

Как настроить сетевой доступ к PostgreSQL?

Существует два уровня настройки сетевого доступа. В файле `postgresql.conf` (параметр `listen_addresses`) задаются адреса привязки и количество удаленных подключений к базе данных. В файле `pg_hba.conf` настраивается аутентификация сетевых соединений. Для разных диапазонов IP-адресов клиента могут задаваться разные правила.

Что такое пользователь и роль?

Разницы между пользователем и ролью почти нет. Для роли по умолчанию предполагается режим `NOLOGIN`, а для пользователя это не так. Во всех остальных отношениях роль и пользователь – одно и то же.

Как изменить пароль?

Очень просто – воспользуйтесь командой `ALTER USER`:

```
test=# ALTER USER hs PASSWORD 'abc';
ALTER ROLE
```

Имейте в виду, что пароли необязательно хранятся в PostgreSQL. Если вы используете аутентификацию через LDAP или еще какой-то внешний метод, то пароль на стороне LDAP не изменится.

Что такое безопасность на уровне строк?

Это возможность ограничить доступ пользователей к содержимому таблицы. Например, пользователю `joe` может быть разрешено видеть только записи о женщинах, а пользователю `jane` – только записи о мужчинах. Таким образом, безопасность на уровне строк сводится к фильтру, принудительно применяемому к таблице с целью ограничения множества строк, доступных пользователю.

Глава 9

Резервное копирование и восстановление

В главе 8 мы узнали о том, как обеспечить безопасность PostgreSQL в типичных ситуациях. Эта глава посвящена резервному копированию и восстановлению. Выполнять резервное копирование следует регулярно, и каждый администратор должен внимательно следить за этой жизненно важной деятельностью. По счастью, PostgreSQL предоставляет простые средства создания резервных копий.

В этой главе рассматриваются следующие вопросы:

- программа `pg_dump`;
- частичная выгрузка данных;
- восстановление из резервной копии;
- распараллеливание;
- сохранение глобальных данных.

Прочитав эту главу, вы сможете правильно настроить механизмы резервного копирования.

Простая выгрузка

Существует два основных метода резервного копирования:

- логические копии (выгрузка SQL-скрипта, представляющего данные);
- трансляция журналов транзакций.

Идея трансляции журналов заключается в том, чтобы в двоичном виде архивировать изменения, произошедшие в базе. Большинство пользователей считает, что трансляция журналов транзакций – единственный реальный способ резервного копирования. Но, на мой взгляд, это не всегда так.

Многие пользуются программой `pg_dump`, чтобы выгрузить текстовое представление данных. Интересно, что `pg_dump` – это самый старый способ резервного копирования, существовавший с момента создания проекта PostgreSQL (трансляция журналов транзакций была добавлена гораздо позже). Каждый администратор PostgreSQL рано или поздно сталкивается с `pg_dump`, поэтому важно знать, как она работает и что делает.

Запуск pg_dump

Для начала создадим простую текстовую выгрузку:

```
[hs@linuxpc ~]$ pg_dump test > /tmp/dump.sql
```

Это самая примитивная резервная копия, которую только можно представить. Программа `pg_dump` заходит на локальный экземпляр базы данных, подключается к базе `test` и начинает выбирать все данные, которые затем отправляются на стандартный вывод и перенаправляются в файл. Прелесть в том, что стандартный вывод предоставляет всю гибкость системы Unix. Данные легко можно сжать, добавив соответствующую программу в конвейер, и вообще сделать с ними все, что угодно.

Иногда требуется запустить `pg_dump` от имени другого пользователя. Все клиентские программы PostgreSQL поддерживают единый набор параметров для задания информации о пользователе. Чтобы просто указать пользователя, воспользуйтесь флагом `-U`:

```
[hs@linuxpc ~]$ pg_dump -U whatever_powerful_user test > /tmp/dump.sql
```

Ниже перечислены параметры, присутствующие во всех клиентских программах PostgreSQL:

...

Параметры подключения:

```
-d, --dbname=БД      имя базы данных для выгрузки
-h, --host=ИМЯ       имя сервера базы данных или каталог сокетов
-p, --port=ПОРТ      номер порта сервера БД
-U, --username=ИМЯ   имя пользователя баз данных
-W, --no-password    не запрашивать пароль
-W, --password       запрашивать пароль всегда (обычно не требуется)
--role=ROLENAME      выполнить SET ROLE перед выгрузкой
...
```

Вы сообщаете необходимую информацию `pg_dump`, и если у вас достаточно прав, то PostgreSQL выбирает данные. Важно понимать, как программа работает на самом деле. Она подключается к базе данных и начинает большую транзакцию с уровнем изоляции `REPEATABLE READ`, которая читает все данные. Напомним, что в этом случае PostgreSQL создает непротиворечивый снимок данных, который не изменяется на всем протяжении транзакции. Иными словами, копия будет согласованной – ни одно ограничение внешнего ключа не будет нарушено. На выходе получится снимок, отражающий состояние данных на момент начала выгрузки. Согласованность – это ключевое слово. Отсюда следует, что никакие изменения данных, произведенные во время выгрузки, не попадут в копию.



Программа `pg_dump` просто читает все подряд, поэтому никаких специальных разрешений на выгрузку не нужно. Если вы можете читать, то можете также создать резервную копию.

Отметим также, что по умолчанию выгрузка производится в текстовом формате. Это означает, что можно спокойно выгрузить данные в ОС Solaris и перенести их на машину с другой архитектурой процессора. В случае двоичных копий такое, очевидно, невозможно, потому что формат данных на диске зависит от архитектуры процессора.

Задание пароля и информации о подключении

Обратите внимание, что среди параметров подключения, перечисленных в предыдущем разделе, нет пароля. Можно затребовать приглашение для ввода пароля, но нельзя задать его в командной строке `pg_dump`. Причина в том, что пароль присутствовал бы в таблице процессов, видимой другим людям. Тогда возникает вопрос: если в находящемся на сервере файле `pg_hba.conf` указано, что пароль необходим, то как передать его клиентской программе?

Есть разные способы. Вот некоторые из них:

- в переменной окружения;
- в файле `.pgpass`;
- в файле службы.

Далее мы рассмотрим все три.

Использование переменных окружения

Переменные окружения – один из способов передать программе различные параметры. Если некоторая информация не передана `pg_dump` явно, то программа будет искать ее в предопределенных переменных окружения. Список всех таких переменных приведен на странице <https://www.postgresql.org/docs/11/static/libpq-envvars.html>¹.

Ниже перечислены некоторые переменные окружения, которые обычно используются для резервного копирования:

- `PGHOST`: определяет, к какому серверу подключаться;
- `PGPORT`: определяет номер используемого TCP-порта;
- `PGUSER`: задает имя пользователя;
- `PGPASSWORD`: задает пароль;
- `PGDATABASE`: определяет имя базы данных, к которой нужно подключиться.

Преимущество этих переменных в том, что пароль не будет появляться в таблице процессов. Но это еще не все. Рассмотрим пример:

```
psql -U ... -h ... -p ... -d ...
```

Представьте, что вы системный администратор. Захочется вам набирать эту длинную строку по многу раз каждый день? Если вы всегда работаете с одним сервером, то просто установите переменные окружения и вводите только строку `psql`. Ниже показано, как это сделать:

```
[hs@linuxpc ~]$ export PGHOST=localhost
[hs@linuxpc ~]$ export PGUSER=hs
```

¹ На русском языке <https://postgrespro.ru/docs/postgresql/11/libpq-envvars>. – Прим. перев.

```
[hs@linuxpc ~]$ export PGPASSWORD=abc
[hs@linuxpc ~]$ export PGPORT=5432
[hs@linuxpc ~]$ export PGDATABASE=test
[hs@linuxpc ~]$ psql
psql (11.0)
Введите "help", чтобы получить справку.
```

Как видим, параметры командной строки больше не нужны.



Все приложения, основанные на стандартной клиентской библиотеке PostgreSQL, написанной на C (libpq), понимают эти переменные окружения, поэтому их можно использовать не только в psql и pg_dump, но и во многих других приложениях.

Использование файлов .pgpass

Очень часто информацию о подключении хранят в файлах .pgpass. Идея проста: поместите в свой домашний каталог файл .pgpass, содержащий все параметры соединения. Формат очень простой:

имя_сервера:порт:база_данных:имя_пользователя:пароль

Например:

192.168.0.45:5432:mydb:xy:abc

PostgreSQL предлагает дополнительную функциональность, разрешая поля, содержащие звездочку *, например:

::*:xy:abc

Здесь звездочки означают, что на любом сервере, при любом номере порта, для любой базы данных пользователь xy будет использовать пароль abc. PostgreSQL будет использовать файл .pgpass, только если для него заданы правильные права доступа:

```
chmod 0600 ~/.pgpass
```

Файл .pgpass можно использовать и в Windows. В этом случае путь к нему должен иметь вид %APPDATA%\postgresql\pgpass.conf.

Использование файла службы

Но .pgpass – не единственный файл, позволяющий решить задачу. Можно также воспользоваться файлом службы. Если вы собираетесь все время подключаться к одним и тем же серверам, то можете создать файл .pg_service.conf, записав в него всю необходимую информацию.

Ниже приведен пример файла .pg_service.conf:

```
Mac:~ hs$ cat .pg_service.conf
# пример службы
[hansservice]
host=localhost
port=5432
dbname=test
user=hs
```

```
password=abc
[paulservice]
host=192.168.0.45
port=5432
dbname=xyz
user=paul
password=cde
```

Для подключения к любой из перечисленных служб просто установите переменную окружения:

```
iMac:~ hs$ export PGSERVICE=hansservice
```

Теперь можно подключаться, не передавая параметры `psql`:

```
iMac:~ hs$ psql
psql (11.0)
Введите "help", чтобы получить справку.
test=#
```

Можно вместо этого сделать так:

```
psql service=hansservice
```

Извлечение подмножества данных

До сих пор мы говорили о том, как выгрузить базу данных целиком. Однако не всегда это необходимо. Иногда требуется извлечь лишь подмножество таблиц или схем. К счастью, `pg_dump` поможет и в этом – с помощью целого ряда флагов:

- `-a`: выгружать только сами данные, но не структуру базы;
- `-s`: выгружать только структуру базы, но не сами данные;
- `-n`: выгружать только определенные схемы;
- `-N`: выгружать все, кроме определенных схем;
- `-t`: выгружать только определенные таблицы;
- `-T`: выгружать все, кроме определенных таблиц (это полезно, если мы хотим исключить таблицы журналов и т. п.).

Частичная выгрузка также позволяет значительно сократить время копирования.

ФОРМАТЫ РЕЗЕРВНОЙ КОПИИ

До сих пор мы рассматривали использование `pg_dump` для создания текстовых файлов. Проблема в том, что текстовый файл можно восстановить только целиком. Если мы сохранили копию всей базы, то и восстановить сможем только всю базу. Но обычно нам нужно не это. Поэтому PostgreSQL предлагает другие формы, обладающие дополнительной функциональностью.

В настоящее время поддерживается четыре формата:

```
-F, --format=c|d|t|p формат выводимых данных (пользовательский, каталог, tag,
                        текстовый(по умолчанию))
```

Текстовый формат мы уже видели. Существует также пользовательский формат. Это сжатая копия с включенным оглавлением. Есть два способа создать копию в пользовательском формате:

```
[hs@linuxpc ~]$ pg_dump -Fc test > /tmp/dump.fc
[hs@linuxpc ~]$ pg_dump -Fc test -f /tmp/dump.fc
```

Помимо оглавления, у сжатой копии есть еще одно достоинство – она гораздо меньше. Как правило, копия в пользовательском формате примерно на 90% меньше размера исходных файлов базы данных. Конечно, все зависит от количества индексов, но для большинства приложений эта грубая оценка верна.

Создав копию, мы можем просмотреть содержимое файла:

```
[hs@linuxpc ~]$ pg_restore --list /tmp/dump.fc
;
; Archive created at 2018-11-04 15:44:56 CET
; dbname: test
; TOC Entries: 18
; Compression: -1
; Dump Version: 1.12-0
; Format: CUSTOM
; Integer: 4 bytes
; Offset: 8 bytes
; Dumped from database version: 11.0
; Dumped by pg_dump version: 11.0
;
; Selected TOC Entries:
;
3103; 1262 16384 DATABASE - test hs
3; 2615 2200 SCHEMA - public hs
3104; 0 0 COMMENT - SCHEMA public hs
1; 3079 13350 EXTENSION - plpgsql
3105; 0 0 COMMENT - EXTENSION plpgsql
187; 1259 16391 TABLE public t_test hs
...
```

Команда `pg_restore --list` возвращает оглавление резервной копии.

Пользовательский формат хорош тем, что размер копии уменьшается. Но это еще не все: команда с флагом `-Fd` создает копию в формате каталога, т. е. вместо одного файла мы получаем каталог, содержащий несколько файлов:

```
[hs@linuxpc ~]$ mkdir /tmp/backup
[hs@linuxpc ~]$ pg_dump -Fd test -f /tmp/backup/
[hs@linuxpc ~]$ cd /tmp/backup/
[hs@linuxpc backup]$ ls -lh
total 86M
-rw-rw-r--. 1 hs hs 85M Jan 4 15:54 3095.dat.gz
-rw-rw-r--. 1 hs hs 107 Jan 4 15:54 3096.dat.gz
-rw-rw-r--. 1 hs hs 740K Jan 4 15:54 3097.dat.gz
-rw-rw-r--. 1 hs hs 39 Jan 4 15:54 3098.dat.gz
-rw-rw-r--. 1 hs hs 4.3K Jan 4 15:54 toc.dat
```

Одно из преимуществ такого режима заключается в том, что для создания копии можно задействовать несколько ядер, тогда как копия в текстовом или пользовательском формате создается одним ядром. В примере ниже показано, как сказать `pg_dump`, что нужно использовать четыре ядра (задания):

```
[hs@linuxpc backup]$ rm -rf *
[hs@linuxpc backup]$ pg_dump -Fd test -f /tmp/backup/ -j 4
```



Чем больше объектов в базе данных, тем больше вероятность ускорения процесса.

ВОССТАНОВЛЕНИЕ ИЗ РЕЗЕРВНОЙ КОПИИ

Наличие резервной копии не имело бы смысла, если бы не могли восстановить из нее базу данных. К счастью, это легко сделать. Если копия создавалась в текстовом формате, то нужно просто выполнить SQL-сценарий:

```
psql your_db < your_file.sql
```

Для восстановления из копии в пользовательском формате или в формате каталога служит команда `pg_restore`. У нее есть много дополнительных возможностей, например восстановление части базы данных и т. д. Но чаще всего мы восстанавливаем базу целиком. В следующем примере мы создаем пустую базу данных и восстанавливаем в нее копию в пользовательском формате:

```
[hs@linuxpc backup]$ createdb new_db
[hs@linuxpc backup]$ pg_restore -d new_db -j 4 /tmp/dump.fc
```

Отметим, что `pg_restore` добавляет данные в существующую базу. Если база не пуста, то `pg_restore` может выдавать сообщения об ошибках, но при этом продолжит работу.

Флаг `-j` позволяет запустить несколько процессов. В данном случае для восстановления задействовано четыре ядра, но смысл в этом есть только тогда, когда восстановлению подлежит более одной таблицы.



Если копия создана в формате каталога, то нужно передать имя каталога, а не файла.

С точки зрения производительности, резервные копии хороши, если мы работаем с данными небольшого или среднего объема. У них два основных недостатка:

- мы получаем мгновенный снимок, т. е. все изменения, произведенные с момента начала копирования, теряются;
- восстановление выгруженных данных происходит медленно, по сравнению с двоичными копиями, потому что необходимо перестраивать все индексы.

Создание двоичных копий мы рассмотрим в главе 10.

СОХРАНЕНИЕ ГЛОБАЛЬНЫХ ДАННЫХ

Выше мы рассмотрели программы `pg_dump` и `pg_restore`, применяемые в процессе резервного копирования. Отметим, что `pg_dump` выгружает только одну базу данных. Если мы хотим создать резервную копию всего экземпляра, то нужно воспользоваться программой `pg_dumpall` или выгружать базы по отдельности. Для начала посмотрим, как работает `pg_dumpall`:

```
pg_dumpall > /tmp/all.sql
```

Программа `pg_dumpall` поочередно подключается к каждой базе и отправляет данные на стандартный выход, так что мы можем обработать их средствами Unix. Но, в отличие от `pg_dump`, у `pg_dumpall` есть недостатки. Она не поддерживает ни пользовательский формат, ни формат каталога, поэтому воспользоваться наличием нескольких ядер невозможно – все копирование производится одним процессом.

Однако у `pg_dumpall` есть в запасе кое-что еще. При создании обычной копии базы данных сохраняются все права, но не команды `CREATE USER`. Эти *глобальные данные* не включаются в обычную копию, их может извлечь только `pg_dumpall`.

Если нужны только глобальные данные, запустите `pg_dumpall` с флагом `-g`:

```
pg_dumpall -g > /tmp/globals.sql
```

В большинстве случаев для создания резервной копии экземпляра можно объединить `pg_dumpall -g` с копированием в пользовательском формате или формате каталога. Простой скрипт мог бы выглядеть следующим образом:

```
#!/bin/sh
BACKUP_DIR=/tmp/
pg_dumpall -g > $BACKUP_DIR/globals.sql
for x in $(psql -c "SELECT datname FROM pg_database
  WHERE datname NOT IN ('postgres', 'template0', 'template1')" postgres -A -t)
do
pg_dump -Fc $x > $BACKUP_DIR/$x.fc done
```

Этот скрипт сначала выгружает глобальные данные, а затем перебирает в цикле все базы данных, выгружая их в пользовательском формате.

РЕЗЮМЕ

В этой главе мы узнали о создании резервных копий. Пока что мы не говорили о двоичных копиях, но уже умеем получать от сервера копии в текстовом формате и восстанавливать из них данные самым простым способом.

В главе 10 мы расскажем о трансляции журналов транзакций, потоковой репликации и двоичных резервных копиях. Мы также узнаем, как использовать включенные в PostgreSQL инструменты для репликации экземпляров.

Вопросы

Всегда ли следует создавать резервные копии?

Если база данных относительно мала, то резервное копирование, безусловно, имеет смысл. Но если база огромна (> XXX ГБ), то создать резервную копию не всегда практически возможно и нужно использовать другие средства (архивацию журналов предзаписи – WAL). Следует также помнить, что резервное копирование дает лишь мгновенный снимок данных и не обеспечивает восстановление на определенный момент времени. Поэтому его лучше рассматривать как дополнительное средство, а не замену архивации WAL.

Почему файлы резервных копий такие маленькие?

Сжатая резервная копия обычно примерно в 10 раз меньше исходной базы данных. Причина в том, что в базе хранятся индексы, а в копии – только сами данные и определения индексов. Кроме того, PostgreSQL хранит дополнительные метаданные, например заголовки кортежей, которые тоже занимают место.

Нужно ли копировать также глобальные данные?

Да, безусловно.

Безопасен ли файл .pgpass?

Да. Если права доступа установлены правильно, то файл .pgpass совершенно безопасен. К тому же посудите сами: компьютер, с которого запускается резервное копирование другой базы данных, должен располагать всей информацией, необходимой для подключения к базе. И не важно, в каком формате эта информация хранится.

Глава 10

Резервное копирование и репликация

В главе 9 мы многое узнали о резервном копировании и восстановлении – теме, неразрывно связанной с администрированием. Но пока мы рассмотрели только логические резервные копии, а сейчас исправим это упущение.

Эта глава посвящена журналу транзакций в PostgreSQL и тому, как правильно его настроить, повысив тем самым безопасность. Мы рассмотрим следующие вопросы:

- что такое журнал транзакций и зачем он нужен;
- восстановление на определенный момент времени;
- настройка потоковой репликации;
- конфликты репликации;
- мониторинг репликации;
- синхронная и асинхронная репликация;
- что такое линии времени;
- логическая репликация;
- создание подписок и публикаций.

Прочитав эту главу, вы сможете настраивать архивацию и репликацию журнала транзакций. Имейте в виду, что эта глава – не полное руководство по репликации, а лишь краткое введение. Полное изложение всех аспектов репликации занимает около 500 страниц. Для сравнения заметим, что в книге «PostgreSQL Replication», вышедшей в издательстве Packt Publishing, почти 400 страниц.

В этой главе мы изложим основные факты в более компактной форме.

Что такое журнал транзакций

Любая современная СУБД содержит средства, гарантирующие, что система сможет продолжить функционирование после аварии в результате сбоя или отключения электропитания. Это справедливо как для файловых систем, так и для систем управления базами данных.

PostgreSQL также предоставляет средства, благодаря которым авария не приводит к повреждению данных и нарушению целостности. Гарантируется, что после выключения питания система сможет восстановиться и продолжить работу.

Такую защиту обеспечивает **журнал предзаписи** (Write Ahead Log – **WAL**), он же **журнал транзакций** (xlog). Идея в том, чтобы производить запись не сразу в файл данных, а сначала в журнал. Почему это так важно? Допустим, что мы записываем какие-то данные:

```
INSERT INTO data ... VALUES ('12345678');
```

Предположим, что запись производится сразу в файл данных. Если что-то случится в середине операции, то файл данных окажется поврежденным. Дело может кончиться наполовину записанными строками, столбцами без индексных указателей, отсутствующей информацией о фиксации и т. д. Поскольку оборудование не гарантирует атомарной записи больших блоков данных, необходимо найти более надежный способ. Проблему можно решить, если писать в журнал, а не в файл.



В PostgreSQL журнал транзакций состоит из записей.

Одна операция записи может содержать несколько разных записей, каждая имеет контрольную сумму, и все они связаны в цепочку. В одной транзакции могут присутствовать записи, касающиеся В-дерева, индекса, диспетчера хранения, фиксации и многого другого. С каждым типом объектов связаны свои записи WAL, гарантирующие, что объект сможет пережить аварию. Когда PostgreSQL запускается после аварии, она исправляет файлы данных, исходя из содержимого журнала транзакций, предотвращая тем самым перманентное повреждение.

Знакомство с журналом транзакций

В PostgreSQL журнал предзаписи обычно находится в подкаталоге `pg_wal` каталога `data`, если противное не указано при запуске `initdb`. В старых версиях PostgreSQL каталог журналов предзаписи назывался `pg_xlog`, но с выходом PostgreSQL 10.0 был переименован.

Причина в том, что пользователи часто удаляли содержимое каталога `pg_xlog`, что, конечно, вело к серьезным последствиям и, возможно, повреждению базы данных. Поэтому сообщество решилось на беспрецедентный шаг – переименовать каталог, являющийся частью экземпляра PostgreSQL, в надежде, что новое имя будет выглядеть достаточно загадочно, чтобы никто не рискнул удалить содержимое.

В следующей распечатке показано, как выглядит каталог `pg_wal`:

```
[postgres@zenbook pg_wal]$ pwd
/var/lib/pgsql/11/data/pg_wal
[postgres@zenbook pg_wal]$ ls -l
total 688132
-rw-----. 1 postgres postgres 16777216 Jan 19 07:58 000000010000000000000000CD
```

```
-rw-----. 1 postgres postgres 16777216 Jan 13 17:04 000000010000000000000000CE
-rw-----. 1 postgres postgres 16777216 Jan 13 17:04 000000010000000000000000CF
-rw-----. 1 postgres postgres 16777216 Jan 13 17:04 000000010000000000000000D0
-rw-----. 1 postgres postgres 16777216 Jan 13 17:04 000000010000000000000000D1
-rw-----. 1 postgres postgres 16777216 Jan 13 17:04 000000010000000000000000D2
```

Мы видим, что каждый файл журнала транзакций занимает 16 МБ, а его имя состоит из 24 шестнадцатеричных цифр. За CF следует D0. Все файлы одинакового размера.

i Заметим, что в PostgreSQL количество файлов журнала транзакций не связано с размером транзакции. Даже очень небольшой набор файлов вполне справляется с транзакцией размером несколько терабайтов.

Традиционно журнал предзаписи состоял из файлов размером 16 МБ. Но теперь размер сегмента WAL можно задавать с помощью программы `initdb`. Иногда это ускоряет работу. Ниже показано, как сделать размер файла WAL равным 32 МБ:

```
initdb -D /pgdata --wal-segsize=32
```

Контрольные точки

Как уже отмечалось, любое изменение записывается в журнал транзакций в двоичном формате (оно не содержит никаких команд SQL). Проблема в том, что сервер не может писать в журнал бесконечно, поскольку со временем он занимает все больше и больше места. Следовательно, в какой-то момент журнал транзакций нужно использовать повторно. Для этого предназначена **контрольная точка**, которая создается автоматически в фоновом режиме.

Смысл в том, что записываемые данные сначала попадают в журнал транзакций. Но со временем измененные («грязные») буферы сбрасываются на диск, т. е. записываются в файлы данных фоновым процессом записи или в процессе создания контрольной точки. Как только все грязные буферы сброшены на диск, журнал транзакций можно удалять.

i Умоляю вас, никогда не удаляйте журнал транзакций вручную. Иначе в случае аварии сервер не сможет запуститься, а если вас беспокоит место на диске, то оно так или иначе будет освобождено по мере поступления новых транзакций. Пожалуйста, не прикасайтесь к журналу транзакций. PostgreSQL сама позаботится о нем, а ваше вмешательство только повредит.

Оптимизация журнала транзакций

Контрольные точки создаются автоматически, когда сервер сочтет нужным. Однако в файле `postgresql.conf` есть несколько конфигурационных параметров, влияющих на этот процесс:

```
#checkpoint_timeout = 5min # диапазон 30s-1d
#max_wal_size = 1GB
#min_wal_size = 80MB
```

Есть две причины начать создание контрольной точки: истекло отведенное время или занято все зарезервированное под журналы место на диске.

Максимальное время между двумя контрольными точками задается параметром `checkpoint_timeout`. Место для хранения журналов транзакций может изменяться от `min_wal_size` до `max_wal_size`. PostgreSQL автоматически начинает создание контрольной точки, так чтобы фактически занятое место не выходило за эти пределы.

i Величина `max_wal_size` – нестрогий предел, иногда (при высокой нагрузке) PostgreSQL может временно занять чуть больше места. Иными словами, если журнал транзакций хранится на отдельном диске, то имеет смысл зарезервировать под него место с небольшим избытком¹.

Как настроить журнал транзакций в версиях PostgreSQL 9.6 и 10.0? В версии 9.6 были внесены некоторые изменения в механизм фоновой записи и контрольных точек. В прежних версиях существовали ситуации, когда для повышения производительности имело смысл задавать меньшее время между контрольными точками. Начиная с версии 9.6 положение изменилось, и теперь больший интервал всегда выгоднее, потому что на уровне базы данных и ОС возможны многочисленные оптимизации, ускоряющие работу. Самая главная из них – сортировка блоков перед записью, что существенно снижает объем произвольного ввода-вывода для вращающихся дисков.

Но это еще не все. Чем больше интервал между контрольными точками, тем меньше размер создаваемых файлов WAL. Да, да, я не оговорился – чем меньше интервал, тем меньше размер.

Причина очень простая. Любой блок, впервые измененный после записи контрольной точки, должен быть отправлен в WAL целиком. Если блок изменяется часто, то в журнал записываются лишь изменения. Чем больше интервал между контрольными точками, тем меньше количество записей полных блоков, что ведет к сокращению общего размера журнала транзакций. Разница может быть весьма существенной, как показано в моей статье по адресу <https://www.cybertec-postgresql.com/checkpoint-distance-and-amount-of-wal/>.

PostgreSQL также позволяет указать, должны ли контрольные точки быть короткими и интенсивными или размазанными по более длительному периоду. По умолчанию подразумевается значение 0.5, означающее, что контрольную точку нужно создавать так, чтобы процесс закончился посередине между текущей и следующей контрольной точкой:

```
#checkpoint_completion_target = 0.5
```

Если увеличить это значение, то контрольная точка будет более растянутой во времени. Во многих случаях увеличенное значение оказывается выгоднее,

¹ Параметр `max_wal_size` задает не общий размер журнальных файлов, а размер между контрольными точками. Если контрольная точка растянута по времени (см. ниже), то общий размер может оказаться в два (а в более старых версиях PostgreSQL – в три) раза больше. – *Прим. ред.*

т. к. позволяет сгладить пики ввода-вывода, неизбежные в случае интенсивного процесса.

АРХИВАЦИЯ И ВОССТАНОВЛЕНИЕ ЖУРНАЛА ТРАНЗАКЦИЙ

После краткого введения в механизм работы журнала транзакций перейдем к процессу архивации журнала. Как мы только что видели, журнал транзакций содержит последовательность двоичных изменений, произведенных в системе хранения. Так почему бы не использовать его для репликации экземпляра базы данных и целого ряда других вещей, например архивации?

Настройка архивации

Прежде всего мы опишем, как настроить стандартное **восстановление на определенный момент времени** (Point-In-Time Recovery – PITR). У PITR есть несколько преимуществ, по сравнению с обычной выгрузкой данных.

- Теряется меньше данных, потому что открывается возможность восстанавливать данные на определенный момент времени, а не на момент начала резервного копирования.
- Восстановление производится быстрее, потому что индексы не нужно создавать с нуля. Они просто копируются и сразу готовы к работе.

Настроить PITR легко. Нужно лишь внести несколько изменений в файл `postgresql.conf`:

```
wal_level = replica # в старых версиях значение называлось "hot_standby"
max_wal_senders = 10 # не менее 2
```

Параметр `wal_level` говорит, что сервер должен записывать в журнал транзакций достаточно информации для PITR. Если в качестве `wal_level` задать значение `minimal` (которое подразумевалось по умолчанию вплоть до версии PostgreSQL 9.6), то журнал будет содержать лишь информацию, необходимую для восстановления одного узла, – этого мало для репликации. В PostgreSQL 10.0 значение по умолчанию установлено правильно и отпала необходимость изменять большую часть параметров.

Параметр `max_wal_senders` позволяет потоком передавать журнал транзакций с сервера. Для создания начальной резервной копии мы можем воспользоваться программой `pg_basebackup` вместо традиционного копирования файлов. Достоинство `pg_basebackup` – в простоте использования. В версии 10.0 начальное значение установлено так, что в 90 % случаев никаких изменений не требуется.

Идея потоковой репликации заключается в том, чтобы скопировать создаваемый журнал транзакций в безопасное место. Существует два способа транспортировки журнала:

- программа `pg_recvwal` (вплоть до версии 9.6 она называлась `pg_recvexlog`);
- использование файловой системы как средства архивации.

В этом разделе мы рассмотрим второй вариант. В процессе нормальной работы PostgreSQL продолжает писать в файлы WAL. Если параметр `archive_mode` в файле `postgresql.conf` равен `on`, то для каждого файла автоматически вызывается программа, заданная в параметре `archive_command`.

Ниже приведен пример фрагмента конфигурационного файла. Предварительно создадим каталог для хранения файлов журнала транзакций:

```
mkdir /archive
chown postgres.postgres archive
```

Затем изменим параметры в файле `postgresql.conf`:

```
archive_mode = on
archive_command = 'cp %p /archive/%f'
```

Для запуска архивации необходимо перезапустить сервер, но сначала изменим файл `pg_hba.conf`, чтобы свести время простоя к минимуму.



В параметр `archive_command` можно записать произвольную команду.

Многие используют для транспортировки файлов WAL в безопасное место команды `gsync`, `scp` и другие. Если скрипт возвращает 0, то PostgreSQL считает, что файл успешно архивирован, в противном случае будет пытаться архивировать его еще раз. Это необходимо, потому что сервер должен быть уверен, что ни один файл не потерялся. Для восстановления необходимо иметь все файлы, отсутствие хотя бы одного делает восстановление невозможным.

Конфигурирование файла `pg_hba.conf`

Сконфигурировав файл `postgresql.conf`, перейдем к файлу `pg_hba.conf`. Это необходимо, только если вы собираетесь использовать программу `pg_basebackup` – современное средство создания резервных копий базы данных.

Параметры, задаваемые в файле `pg_hba.conf`, уже были описаны в главе 8, посвященной безопасности в PostgreSQL. Нужно только помнить об одном важном нюансе:

```
# Разрешить подключения для репликации с localhost от имени пользователя
# с привилегией replication.
local replication postgres trust
host replication postgres 127.0.0.1/32 trust
host replication postgres ::1/128 trust
```

Мы можем определить стандартные правила в файле `pg_hba.conf`. Но важно, что во втором столбце находится слово `replication`. Обычных правил недостаточно – необходимо явно добавить право на репликацию. Причем это необязательно должен быть суперпользователь, можно создать специального пользователя, которому будет разрешено только подключаться и выполнять репликацию.

Начиная с версии PostgreSQL 10 система заранее сконфигурирована, как описано в этом разделе. Локальная репликация работает без какой-либо дополнительной настройки, а для удаленной надо указать IP-адреса в файле `pg_hba.conf`.

Вот теперь можно перезапускать сервер PostgreSQL.

Создание базовой резервной копии

Итак, архивировать файлы WAL мы научились, настало время создать первую резервную копию. Имея ее и последующие файлы WAL, мы сможем восстановить состояние базы данных на любой момент времени.

Для создания начальной копии воспользуемся командой `pg_basebackup`:

```
pg_basebackup -D /some_target_dir
-h localhost
--checkpoint=fast
--wal-method=stream
```

У этой команды четыре параметра.

- `-D`: где должна находиться базовая резервная копия? PostgreSQL требует, чтобы это был пустой каталог. По завершении копирования в нем окажется копия каталога `data` на сервере.
- `-h`: IP-адрес или имя главного сервера (источника), который мы собираемся скопировать.
- `--checkpoint=fast`: обычно `pg_basebackup` ждет, пока главный сервер не создаст контрольную точку, поскольку процесс восстановления должен с чего-то начинаться. Контрольная точка гарантирует, что записаны все данные вплоть до определенного момента, поэтому PostgreSQL может безопасно начать восстановление с этого состояния. В принципе, для этого параметр `--checkpoint=fast` не нужен. Но тогда `pg_basebackup` может отложить начало копирования, если, к примеру, интервал между контрольными точками составляет час, а зачем нам такая задержка?
- `--wal-method=stream`: по умолчанию `pg_basebackup` подключается к главному серверу и начинает копировать файлы. Но имейте в виду, что пока происходит копирование, эти файлы могут модифицироваться. Поэтому данные в резервной копии могут оказаться несогласованными. Эту несогласованность можно устранить в процессе восстановления с помощью журнала транзакций. Но сама резервная копия необязательно согласована. Параметр `--wal-method=stream` позволяет создать согласованную резервную копию, из которой можно восстановить базу данных, не воспроизводя журнал транзакций. Это удобно, если нам нужно всего лишь клонировать экземпляр, а восстановление на определенный момент времени ни к чему. К счастью, режим `--wal-method=stream` в версии PostgreSQL 10.0 подразумевается по умолчанию. Но в версии 9.6 и более ранних рекомендуется использовать его прежний аналог `--xlog-method=stream`.

Уменьшение полосы пропускания, занятой резервным копированием

Программа `pg_basebackup` стремится закончить работу как можно скорее. Если сеть быстрая, то она может копировать сотни мегабайт в секунду. Но если подсистема ввода-вывода не справляется, то `pg_basebackup` способна поглотить все ресурсы, и тогда запросы пользователей будут выполняться очень медленно.

Для контроля над максимальной скоростью передачи у `pg_basebackup` имеется параметр:

```
-g, --max-rate=RATE
    максимальная скорость передачи данных с сервера
    (по умолчанию в КБ/с, но можно указать суффикс "k" или "M")
```

Создавая базовую резервную копию, следите за тем, чтобы дисковая подсистема на сервере справлялась с нагрузкой. Настройка скорости передачи может оказать в этом помощь.

Отображение табличных пространств

Если структура каталогов в конечной системе такая же, как в исходной, программу `pg_basebackup` можно запускать без дополнительных параметров. В противном случае мы можем отобразить структуру файловой системы сервера на нужную нам структуру:

```
-T, --tablespace-mapping=OLDDIR=NEWDIR
    переместить табличное пространство из каталога OLDDIR в каталог NEWDIR/
```



Если ваша система мала, то разумно хранить все данные в одном табличном пространстве. Это справедливо, когда ввод-вывод не является проблемой (быть может, потому что размер данных составляет всего несколько гигабайт).

Использование различных форматов

Программа `pg_basebackup` поддерживает различные форматы. По умолчанию все данные помещаются в пустой каталог. Программа подключается к серверу-источнику и копирует данные в указанный каталог.

Проблема в том, что `pg_basebackup` создает много файлов, что неудобно, если мы собираемся поместить резервную копию во внешнюю систему архивирования типа Tivoli или ей подобную. Ниже показано, какие форматы вывода поддерживает `pg_basebackup`:

```
-F, --format=p|t формат вывода (простой (по умолчанию), tar)
```

Чтобы создать один файл, следует задать флаг `-F=t`. По умолчанию при этом создается файл `base.tar`, с которым потом проще управляться. Очевидный недостаток этого подхода в том, что перед восстановлением архив придется распаковать.

Проверка результата архивации журнала транзакций

Прежде чем переходить к восстановлению, имеет смысл проверить, правильно ли произведена архивация и будет ли она работать, как ожидается. Для этого достаточно простой команды `ls`:


```
[hs@zenbook archive]$ ls -l
total 212996
-rw----- 1 hs hs 16777216 Jan 30 09:04 00000001000000000000000001
-rw----- 1 hs hs 16777216 Jan 30 09:04 00000001000000000000000002
-rw----- 1 hs hs      302 Jan 30 09:04 00000001000000000000000002.00000028.backup
-rw----- 1 hs hs 16777216 Jan 30 09:20 00000001000000000000000003
-rw----- 1 hs hs 16777216 Jan 30 09:20 00000001000000000000000004
-rw----- 1 hs hs 16777216 Jan 30 09:20 00000001000000000000000005
-rw----- 1 hs hs 16777216 Jan 30 09:20 00000001000000000000000006
...
```

Помимо простой проверки файлов, полезно также следующее системное представление:

```
test=# \d pg_stat_archiver
Представление "pg_catalog.pg_stat_archiver"
  Столбец      | Тип              | Модификаторы
-----+-----+-----
archived_count | bigint           |
last_archived_wal | text            |
last_archived_time | timestamp with time zone |
failed_count    | bigint           |
last_failed_wal  | text            |
last_failed_time | timestamp with time zone |
stats_reset     | timestamp with time zone |
```

Представление `pg_stat_archiver` очень полезно, когда нужно выяснить, не остановилась ли по какой-то причине архивация, и если да, то когда это произошло. Оно сообщит о количестве уже архивированных файлов (`archived_count`). Также видно, какой файл был архивирован последним и когда это случилось. Наконец, если произошла какая-то ошибка, то `pg_stat_archiver` покажет это. К сожалению, ни кода ошибки, ни сообщения о ней в этом представлении нет, но сообщения команды архивирования `archive_command` нетрудно протоколировать самостоятельно.

Обратим внимание еще на один момент. Выше уже было описано, как посмотреть, какие файлы архивированы. Но это не все. Программа `pg_basebackup` создает также файл с расширением `.backup`. Это небольшой файл, содержащий только сведения о самой резервной копии, он чисто информационный и в процессе восстановления не используется. Однако он весьма полезен. Когда позднее мы начнем воспроизводить журнал транзакций, все файлы WAL, которые старше `backup`-файла, можно удалить. В нашем примере `backup`-файл называется `00000001000000000000000002.00000028.backup`. Это значит, что процесс восстановления начинается где-то в файле `...0002` (с позиции `...28`). И следовательно, все файлы, которые старше `...0002`, можно удалить, поскольку они уже не пригодятся. Имейте в виду, что может храниться более одной базовой резервной копии, я сейчас говорю только о текущей.

Воспроизведение журнала транзакций

Подведем итог сделанному до сих пор. Мы задали параметры в файле `postgresql.conf` (`wal_level`, `max_wal_senders`, `archive_mode` и `archive_command`) и прописали

параметры для `pg_basebackup` в файле `pg_hba.conf`. Затем мы перезапустили базу данных и создали базовую резервную копию.

Отметим, что в процессе создания базовой резервной копии база может работать без ограничений. Нам пришлось только ненадолго приостановить ее на время перезапуска, чтобы параметры `max_wal_sender` и `wal_level` вступили в силу.

Ну а потом в процессе работы системы может произойти авария, и нам придется восстанавливать базу. Мы можем выполнить процедуру PITR, чтобы восстановить столько данных, сколько возможно. Первым делом мы должны поместить базовую резервную копию в нужное место.



Я рекомендую предварительно сохранить старый кластер баз данных. Даже если он поврежден, компании, осуществляющей техническую поддержку, он может понадобиться для установления причины аварии. Впоследствии, когда все снова заработает, его можно будет удалить.

Для этого нужно будет выполнить примерно такую команду:

```
cd /some_target_dir
cp -Rv * /data
```

Здесь предполагается, что данные восстановленного сервера будут находиться в каталоге `/data`. Прежде чем копировать базовую копию, убедитесь, что этот каталог пуст.

На следующем шаге нужно создать файл `recovery.conf`. Он будет содержать всю информацию о процессе воспроизведения: местонахождение архива файлов WAL, время, на которое мы хотим восстановить данные, и т. д. Ниже приведен пример файла `recovery.conf`:

```
restore_command = 'cp /archive/%f %p'
recovery_target_time = '2019-04-05 15:43:12'
```

Поместив файл `recovery.conf` в каталог `$PGDATA`, мы можем запустить сервер. При этом будет печататься информация вида:

```
server starting
LOG: database system was interrupted; last known up
at 2017-01-30 09:04:07 CET
LOG: starting point-in-time recovery to 2019-04-05 15:43:12+02
LOG: restored log file "00000001000000000000000002" from archive
LOG: redo starts at 0/2000028
LOG: consistent recovery state reached at 0/20000F8
LOG: restored log file "00000001000000000000000003" from archive
LOG: restored log file "00000001000000000000000004" from archive
LOG: restored log file "00000001000000000000000005" from archive
...
LOG: restored log file "0000000100000000000000000E" from archive
cp: cannot stat '/archive/0000000100000000000000000F': No such file or directory
LOG: redo done at 0/E7BF710
LOG: last completed transaction was at log time
2017-01-30 09:20:47.249497+01
```

```

LOG: restored log file "0000000100000000000000E" from archive
cp: cannot stat '/archive/00000002.history': No such file or directory
LOG: selected new timeline ID: 2
cp: cannot stat '/archive/00000001.history': No such file or directory
LOG: archive recovery complete
LOG: MultiXact member wraparound protections are now enabled
LOG: database system is ready to accept connections
LOG: autovacuum launcher started

```

После того как сервер запустится, нужно поискать в журнале сообщения, показывающие, что восстановление произошло успешно. Первое из них – `consistent recovery state reached`. Оно означает, что PostgreSQL сможет воспроизвести часть журнала транзакций и привести базу данных в работоспособное состояние.

Затем PostgreSQL начинает поочередно копировать и воспроизводить файлы. Обратите внимание, что в файле `recovery.conf` мы попросили довести базу данных до некоторого момента в 2019 году. Но этот текст был написан в 2017 году, поэтому файлов WAL за 2019 год, конечно, нет. PostgreSQL сообщает об этом и выводит время последней зафиксированной транзакции.

Разумеется, это всего лишь демонстрация, на практике мы, скорее всего, указали бы дату в прошлом, до которой можно безопасно восстановиться. Я просто хотел показать, что задавать дату в будущем тоже можно, только приготовьтесь увидеть сообщение об ошибке.

По завершении восстановления файл `recovery.conf` переименовывается в `recovery.done`, чтобы было понятно, что задача выполнена. Все серверные процессы запущены, и экземпляр базы данных готов к работе.

Нахождение нужной временной метки

До сих пор мы предполагали, что знаем, на какой момент времени хотим восстановиться, или собираемся воспроизвести весь журнал транзакций, чтобы минимизировать потерю данных. Но что, если мы не хотим воспроизводить все? И не знаем точный момент времени? На практике это самая распространенная ситуация. Как-то один из наших разработчиков утром потерял данные, а от нас требовалось сделать, чтобы все снова стало хорошо. Но вот вопрос: утром-то утром, но когда именно? Если восстановление закончилось, то заново запустить его не так-то просто. После восстановления база окажется в состоянии, соответствующем указанному моменту времени, и снова воспроизвести журнал транзакций мы уже не сможем.

Однако мы можем приостановить восстановление без перевода базы в новое состояние, проверить, что оказалось в базе, и продолжить.

Сделать это просто. Сначала проверьте, что параметр `hot_standby` в файле `postgresql.conf` равен `on`. Это означает, что базу данных можно читать в процессе восстановления. Затем еще до запуска воспроизведения измените файл `recovery.conf`:

```
recovery_target_action = 'pause'
```

Параметр `recovery_target_action` может принимать различные значения. Если задано значение `pause`, то PostgreSQL приостановит процесс в указанное время и даст возможность проверить, что уже воспроизведено. Потом можно скорректировать время и попробовать снова. Другие возможные значения: `promote` и `shutdown`.

Есть и иной способ приостановить воспроизведение журнала транзакций. Его можно использовать при восстановлении на определенный момент времени, но обычно он применяется в сочетании с потоковой репликацией. Вот что можно сделать во время воспроизведения файлов WAL:

```
postgres=# \x
Расширенный вывод включен.
postgres=# \df *pause*
Список функций
-[ RECORD 1 ]-----+-----
Схема          | pg_catalog
Имя            | pg_is_wal_replay_paused
Тип данных результата | boolean
Типы данных аргументов | 
Тип           | обычная
-[ RECORD 2 ]-----+-----
Схема          | pg_catalog
Имя            | pg_wal_replay_paused
Тип данных результата | void
Типы данных аргументов | 
Тип           | обычная

postgres=# \df *resume*
Список функций
-[ RECORD 1 ]-----+-----
Схема          | pg_catalog
Имя            | pg_wal_replay_resume
Тип данных результата | void
Типы данных аргументов | 
Тип           | обычная
```

Мы можем выполнить команду `SELECT pg_wal_replay_pause();`, чтобы остановить воспроизведение журнала транзакций до момента выполнения команды `SELECT pg_wal_replay_resume();`.

Задача в том, чтобы выяснить, какая часть журнала транзакций уже воспроизведена, и, если необходимо, продолжить. Но имейте в виду: если сервер уже переведен в новое состояние, то продолжить воспроизведение журнала мы не сможем.

Итак, узнать, до какого момента следует восстанавливать данные, довольно трудно. Поэтому PostgreSQL предоставляет помощь в этом деле. Рассмотрим такой пример из реальной жизни: ежедневно в полночь мы запускаем процесс, который завершается неизвестно когда. Задача – восстановить данные точно на момент завершения этого процесса. Но как узнать, когда процесс

завершился? Обычно это нелегко. Но почему бы не включить в журнал транзакций специальный маркер:

```
postgres=# SELECT pg_create_restore_point('my_daily_process_ended');
pg_create_restore_point
-----
1F/E574A7B8
(1 строка)
```

Если наш процесс выполнит эту команду сразу после завершения, то мы сможем воспользоваться этой меткой и восстановиться точно на этот момент. Для этого нужно добавить в файл `recovery.conf` такую строку:

```
recovery_target_name = 'my_daily_process_ended'
```

Если использовать эту директиву вместо `recovery_target_time`, то воспроизведение дойдет точно до момента завершения ночного процесса.

Разумеется, воспроизвести журнал можно и до идентификатора конкретной транзакции. Но в жизни идентификатор транзакции редко бывает известен администратору, так что практической пользы в этом немного.

Очистка архива журналов транзакций

До сих пор мы только писали в архив и не задумывались о том, когда его очищать, чтобы освободить место в файловой системе. PostgreSQL не может сделать эту работу за нас, потому что не знает, захотим ли мы когда-нибудь использовать архив. Так что бремя очистки архива лежит на администраторе.

Предположим, что мы хотим стереть старый, уже не нужный журнал транзакций. Быть может, мы храним несколько базовых резервных копий и стираем журналы, которые уже не понадобятся для восстановления из старых копий.

В таком случае нам поможет командная утилита `pg_archivecleanup`. Ей нужно передать каталог архива и имя backup-файла, а она позаботится об удалении файлов с диска. Благодаря этому инструменту мы можем не думать, какие журналы транзакций сохранять, а какие можно стереть.

```
[hs@asus ~]$ pg_archivecleanup --help
pg_archivecleanup удаляет старые файлы WAL из архивов PostgreSQL.
```

Порядок вызова:

```
pg_archivecleanup [OPTION]... ARCHIVELOCATION OLDESTKEPTWALFILE
```

Параметры:

- d выводить отладочные сообщения (режим подробной информации)
- n пробный прогон, показывать имена файлов, которые будут удалены
- V, --version вывести информацию версии и выйти
- x EXT стереть только файлы с указанным расширением
- ?, --help вывести это сообщение и выйти

Для использования в качестве `archive_cleanup_command` в файле `recovery.conf`, когда `standby_mode = on`:

```
archive_cleanup_command = 'pg_archivecleanup [OPTION]... ARCHIVELOCATION %r'
```

например:

```
archive_cleanup_command = 'pg_archivecleanup /mnt/server/archiverdir %r'
```

Или для использования в качестве автономного средства очистки архивов:

Например:

```
pg_archivecleanup /mnt/server/archiverdir
000000010000000000000010.00000020.backup
```

Эта программа имеется на всех платформах, и работать с ней легко.

НАСТРОЙКА АСИНХРОННОЙ РЕПЛИКАЦИИ

Познакомившись с архивацией журнала транзакций и восстановлением на определенный момент времени, перейдем к одному из самых популярных на сегодняшний день средств PostgreSQL: потоковой репликации. Идея проста: после создания начальной базовой копии резервный сервер подключается к главному, забирает журнал транзакций в режиме реального времени и применяет его. Воспроизведение журнала транзакций становится не однократной операцией, а непрерывным процессом, который работает на протяжении всего времени существования кластера.

Базовая настройка

В этом разделе мы узнаем, как легко и быстро настроить асинхронную репликацию. Нашей целью будет конфигурирование системы с двумя узлами.

Собственно, основная часть работы уже была проделана, когда мы настраивали архивацию журнала транзакций. Но все же рассмотрим всю процедуру от начала до конца, поскольку мы не можем предполагать, что трансляция WAL действительно настроена правильно.

Прежде всего изменим следующие параметры в файле `postgresql.conf`:

```
wal_level = replica
max_wal_senders = 10 # или любое значение >= 2
hot_standby = on      # это уже усложнение
```

i Для некоторых параметров в PostgreSQL 10.0 уже заданы хорошие значения по умолчанию.

Как и раньше, параметр `wal_level` задан так, чтобы PostgreSQL порождала достаточно файлов журналов, чтобы резервный сервер был постоянно занят. Затем нужно выбрать значение `max_wal_senders`. В момент запуска резервного сервера и при создании базовой резервной копии процесс-отправитель WAL контактирует с процессом-получателем WAL на стороне клиента. Параметр `max_wal_senders` позволяет PostgreSQL создать достаточно процессов для обслуживания клиентов.

✓ Теоретически достаточно всего одного процесса-отправителя WAL. Но это неудобно. Для базовой резервной копии с параметром `--wal-method=stream` уже нужно два процесса-отправителя. Если вы хотите, чтобы одновременно работал резервный сервер и создавалась базовая резервная копия, то требуется уже три процесса. Поэтому позвольте PostgreSQL заранее создать достаточно процессов, чтобы избежать лишних перезапусков.

Следующий параметр – `hot_standby`. Главный сервер игнорирует параметр `hot_standby`. Его единственное назначение – разрешить чтение с резервного сервера во время воспроизведения журнала транзакций. Тогда какое нам до него дело? Вспомните, что программа `pg_basebackup` клонирует сервер целиком, включая конфигурацию. Это означает, что если этот параметр установлен на главном сервере, то он перейдет на все резервные при создании копии.

Далее перейдем к файлу `pg_hba.conf`: мы должны разрешить резервному серверу выполнять репликацию, добавив правила – те же самые, что при настройке PITR.

После этого нужно перезапустить сервер базы данных.

Затем можно запускать на резервном сервере программу `pg_basebackup`. Но сначала убедитесь, что каталог `/target` пуст. Если вы пользуетесь RPM-пакетами, то сначала остановите потенциально работающий экземпляр и удалите все из каталога (например, `/var/lib/pgsql/data`).

```
pg_basebackup -D /target
-h master.example.com
--checkpoint=fast
--wal-method=stream -R
```

Вместо `/target` подставьте свой конечный каталог, а вместо `master.example.com` – IP-адрес или доменное имя главного сервера. Параметр `--checkpoint=fast` означает, что контрольную точку следует создать немедленно. Благодаря параметру `--wal-method=stream` открываются два потока. Через один поток копируются данные, а через другой – журнал транзакций, который создается, пока работает программа `pg_basebackup`.

И наконец, флаг `-R`:

```
-R, --write-recovery-conf # записать recovery.conf после создания резервной копии
```

Это очень полезная возможность. Программа `pg_basebackup` умеет автоматически создавать конфигурацию резервного сервера, включая различные параметры в файл `recovery.conf`:

```
standby_mode = on
primary_conninfo = ' ... '
```

Первый параметр говорит, что PostgreSQL должна постоянно оставаться в режиме воспроизведения журнала транзакций: если весь журнал уже воспроизведен, то сервер должен ждать поступления следующего файла WAL. Второй параметр сообщает, где находится главный сервер. Это стандартная информация о подключении к базе данных.



Один резервный сервер может подключаться к другому для потоковой передачи журналов транзакций. Можно настроить каскадную репликацию, сняв базовую резервную копию с резервного сервера. В этом контексте *главный* означает просто сервер-источник.

После запуска программы `pg_basebackup` уже можно запускать службы. Сначала проверим, что на главном сервере работает *процесс-отправитель WAL*:

```
[hs@linuxpc ~]$ ps ax | grep sender
17873 ? Ss 0:00 postgres: wal sender process ah ::1(57596) streaming 1F/E9000060
```

Если это так, то на резервном сервере будет работать *процесс-приемник WAL*:

```
17872 ? Ss 0:00 postgres: wal receiver process streaming 1F/E9000060
```

Если эти процессы присутствуют, значит, мы на правильном пути, и репликация работает, как ожидается. Обе стороны взаимодействуют друг с другом, и журнал транзакций передается от главного сервера резервному.

Повышение уровня безопасности

До сих пор потоковая репликация осуществлялась от имени суперпользователя. Но разрешать доступ от имени суперпользователя с удаленного узла не стоит. По счастью, PostgreSQL позволяет создать пользователя, которому разрешено только потреблять поток журнала транзакций и больше ничего.

```
test=# CREATE USER repl LOGIN REPLICATION;
CREATE ROLE
```

Дав пользователю право `replication`, мы наделяем его возможностью производить потоковую репликацию – все остальное запрещено.

Настоятельно рекомендуется не использовать учетную запись суперпользователя для потоковой репликации. Просто измените имя пользователя в файле `recovery.conf`. Тем самым вы существенно повысите уровень безопасности.

Остановка и возобновление репликации

Будучи запущена, потоковая репликация работает, не требуя особого внимания администратора. Но в некоторых случаях бывает необходимо остановить репликацию, с тем чтобы возобновить ее позже. Зачем это может понадобиться?

Рассмотрим следующую ситуацию: вы отвечаете за настройку системы с главным и резервным серверами, на которой работает какая-то дурацкая CMS (система управления контентом) или сомнительный форум. Требуется перейти с ужасной версии CMS 1.0 на кошмарную CMS 2.0. В базе данных производятся некоторые изменения, которые мгновенно реплицируются на резервный сервер. А что, если при переходе на новую версию что-то пойдет не так? Тогда ошибка сразу же появится на обоих узлах.

Чтобы избежать мгновенного распространения изменений, мы можем остановить репликацию и возобновить, когда понадобится. В рассмотренном примере нужно будет сделать следующее.

1. Остановить репликацию.
2. Произвести обновление ПО на главном сервере.
3. Проверить, что приложение по-прежнему работает. Если это так, возобновить репликацию, иначе сделать главным сервером реплику, на которой по-прежнему находятся старые данные.

Этот механизм позволяет защитить данные, т. к. мы можем вернуться к состоянию, существовавшему перед возникновением проблемы. Ниже в этой главе мы покажем, как поменять ролями резервный и главный серверы.

Теперь вопрос: как остановить репликацию? Для этого нужно выполнить следующую команду на резервном сервере:

```
test=# SELECT pg_wal_replay_pause();
```

Эта команда останавливает репликацию. Журнал транзакций по-прежнему передается с главного сервера на резервный – остановлен только процесс воспроизведения. Данные по-прежнему защищены, поскольку их копия находится на резервном сервере. В случае аварии данные не будут потеряны.

Помните, что процесс воспроизведения следует останавливать именно на резервном сервере. В противном случае PostgreSQL выдаст сообщение об ошибке:

ОШИБКА: восстановление не выполняется

ПОДСКАЗКА: функции управления восстановлением можно использовать только в процессе восстановления.

Когда настанет время возобновить репликацию, на резервном сервере нужно будет выполнить такую команду:

```
SELECT pg_wal_replay_resume();
```

PostgreSQL снова начнет воспроизводить журнал транзакций.

Проверка состояния репликации для обеспечения доступности

Одна из основных задач администратора – следить, чтобы репликация не останавливалась. Если репликация прекратится, то в случае аварии главного сервера возможна потеря данных.

К счастью, PostgreSQL предоставляет системные представления, которые позволяют подробно изучить, что происходит. Одно из них называется `pg_stat_replication`:

```
test=# \d pg_stat_replication
```

Представление "pg_catalog.pg_stat_replication"			
Столбец	Тип	Правило сортировки	Допустимость NULL
pid	integer		
usesysid	oid		
username	name		
application_name	text		
client_addr	inet		
client_hostname	text		
client_port	integer		
backend_start	timestamp with time zone		
backend_xmin	xid		
state	text		
sent_lsn	pg_lsn		
write_lsn	pg_lsn		

flush_lsn	pg_lsn		
replay_lsn	pg_lsn		
write_lag	interval		
flush_lag	interval		
replay_lag	interval		
sync_priority	integer		
sync_state	text		

Это представление содержит информацию об отправителе. Я не хочу употреблять здесь термин «главный сервер», потому один резервный сервер может быть соединен с другим, тоже резервным. Можно построить дерево серверов, и тогда главный сервер будет знать только о тех резервных, которые подключены к нему напрямую.

Первое, что мы видим, – это идентификатор процесса-отправителя WAL. Он поможет найти процесс, если что-то пойдет не так. Впрочем, это редкость. Далее идет имя пользователя, которое резервный сервер использовал для подключения к отправителю. Поля вида `client_*` показывают, где в сети находится резервный сервер. Поле `backend_start` говорит, когда резервный сервер начал потоковую репликацию отправителя.

Далее мы видим загадочное поле `backend_xmin`. Предположим, что работает система в конфигурации главный–резервный. Резервный сервер можно попросить, чтобы он сообщил главному свой идентификатор транзакции. Смысл в том, чтобы задержать очистку на главном сервере, так чтобы данные не брались из транзакции, выполняемой на резервном.

Поле `state` информирует о состоянии сервера. Если с системой все в порядке, то поле будет содержать значение `streaming`. В противном случае необходимо изучить проблему детально.

Следующие четыре поля очень важны. Поле `sent_lsn`, которое раньше называлось `sent_location`, показывает, какая часть журнала транзакций уже передана другой стороне, т. е. подтверждена приемником. После того как данные журнала транзакций приняты из сети, они передаются ОС. Поле `write_lsn`, бывшее `write_location`, говорит, какая часть журнала уже дошла до ОС. Поле `flush_lsn`, бывшее `flush_location`, говорит, какая часть журнала уже сброшена на диск.

И наконец, поле `replay_lsn`, которое раньше называлось `replay_location`. Тот факт, что журнал транзакций записан на диск резервного сервера, еще не означает, что PostgreSQL уже воспроизвела этот журнал, т. е. сделала находящиеся в нем изменения видимыми пользователям. Предположим, что репликация приостановлена. Данные по-прежнему передаются резервному серверу, но пока не воспроизводятся. Поле `replay_lsn` как раз и говорит, какая часть данных видима.

В версии PostgreSQL 10.0 в представление `pg_stat_replication` добавлены новые поля; поля вида `*_lag` показывают, насколько резервный сервер отстает от главного.

i Тип этих полей `interval`, чтобы была наглядно видна разница во времени.

Наконец, PostgreSQL сообщает, является ли репликация синхронной или асинхронной.

Если вы по-прежнему работаете с версией PostgreSQL 9.6, то можете подсчитать отставание принимающего сервера от передающего в байтах. Полей `*_lag` в версии 9.6 еще нет, но знать расхождение в байтах иногда полезно. Вот как это делается:

```
SELECT client_addr, pg_current_wal_location() - sent_location AS diff
FROM pg_stat_replication;
```

Функция `pg_current_wal_location()`, выполняемая на главном сервере, возвращает текущую позицию в журнале транзакций. В PostgreSQL 9.6 имеется специальный тип данных `pg_lsn` для таких позиций. Для этого типа определено несколько операторов, которыми мы здесь воспользовались, чтобы вычесть позицию на резервном сервере из позиции на главном сервере. Таким образом, запрос возвращает расхождение между двумя серверами в байтах (задержку репликации).

i В таком виде команда работает только в PostgreSQL 10. В предыдущих версиях функция называлась `pg_current_xlog_location()`.

Если представление `pg_stat_replication` содержит информацию о передающей стороне, то `pg_stat_wal_receiver` дает аналогичную информацию о принимающей стороне:

```
test=# \d pg_stat_wal_receiver
```

Представление "pg_catalog.pg_stat_wal_receiver"				
Столбец	Тип	Правило сортировки	Допустимость NULL	
pid	integer			
status	text			
receive_start_lsn	pg_lsn			
receive_start_tli	integer			
received_lsn	pg_lsn			
received_tli	integer			
last_msg_send_time	timestamp with time zone			
last_msg_receipt_time	timestamp with time zone			
latest_end_lsn	pg_lsn			
latest_end_time	timestamp with time zone			
slot_name	text			
sender_host	text			
sender_port	integer			
conninfo	text			

Помимо идентификатора процесса-приемника WAL, PostgreSQL сообщает состояние этого процесса. Поле `receive_start_lsn` содержит позицию в журнале

транзакций, начиная с которой осуществляется прием, а поле `receive_start_tli` – линию времени, которая использовалась, когда был запущен приемник.

В поле `received_lsn` хранится позиция в журнале транзакций, до которой данные уже приняты и сброшены на диск. Затем идет информация о времени, slots репликации и подключении.

Вообще говоря, многие считают, что читать представление `pg_stat_replication` проще, чем `pg_stat_wal_receiver`, и большинство инструментальных средств работает именно с `pg_stat_replication`.

Обработка отказов и линии времени

Один раз настроенная пара главный–резервный обычно бесперебойно работает в течение очень долгого времени. Но все когда-то ломается, поэтому важно понимать, как заменить вышедший из строя сервер резервным.

PostgreSQL упрощает обработку отказов и переход резервного сервера на роль главного. Нужно лишь вызвать программу `pg_ctl`, которая повысит статус реплики:

```
pg_ctl -D data_dir promote
```

Резервный сервер мгновенно отключается от главного и сам становится главным. Напомним, что в момент повышения статуса резервный сервер мог обслуживать тысячи подключений, по которым производилось чтение. И все эти подключения будут автоматически преобразованы, так что станет возможна и запись – даже переподключаться не придется.

Повышая статус сервера, PostgreSQL увеличивает на единицу линию времени. Для совершенно нового сервера линия времени равна 1. Если резервный сервер является копией главного, то его линия времени будет такой же, как у главного, т. е. оба сервера будут находиться на линии времени 1. После повышения статуса резервный сервер становится независимым главным и переходит на линию времени 2.

Линии времени особенно важны при восстановлении на определенный момент времени. Предположим, что базовая резервная копия создана примерно в полночь. В 12:00 дня резервный сервер стал главным. В 3:00 дня происходит сбой, и мы хотим восстановить состояние на момент 2:00 дня. Мы воспроизведем журнал транзакций, созданный после базовой резервной копии, и затем перейдем на поток WAL от главного сервера, поскольку эти два узла разошлись в 12:00 дня.

Об изменении линии времени можно также судить по именам файлов журнала транзакций. Вот как выглядит файл WAL на линии времени 1:

```
000000001000000000000000F5
```

После перехода на линию времени 2 имя файла становится таким:

```
000000002000000000000000F5
```

Как видим, файлы WAL, принадлежащие разным линиям времени, теоретически могут находиться в одном архивном каталоге.

Управление конфликтами

Мы уже много узнали о репликации. Далее мы познакомимся с конфликтами репликации. Главный вопрос: как вообще может случиться конфликт?

Рассмотрим пример.

Главный	Резервный
	BEGIN;
	SELECT ... FROM tab WHERE ...
	... работает ...
DROP TABLE tab;	... имеет место конфликт ...
	... транзакции разрешено работать еще 30 с ...
	... конфликт разрешается, или запрос снимается до истечения тайм-аута ...

Проблема здесь в том, что главный сервер не знает, что на резервном выполняется транзакция. Поэтому команда DROP TABLE не задерживается до окончания чтения, как было бы, если бы обе транзакции работали на одном сервере. Поскольку серверы разные, то команда DROP TABLE отрабатывает нормально, и посредством журнала транзакций на резервный сервер попадает запрос с требованием удалить соответствующие файлы данных с диска. Теперь резервный сервер сталкивается с дилеммой: если удалить таблицу с диска, то команду SELECT нужно прерывать, а если ждать, пока SELECT завершится, и задержать воспроизведение журнала, то можно безнадежно отстать от главного сервера.

Идеальным решением является компромисс, который управляется конфигурационным параметром:

```
max_standby_streaming_delay = 30s
# максимальная задержка до отмены запросов чтения
# в системе с потоковой репликацией WAL;
```

Идея в том, чтобы подождать 30 с, прежде чем отменять запрос на резервном сервере. В зависимости от приложения можно задавать этот параметр более или менее агрессивно. Отметим, что 30 с относятся ко всему потоку репликации, а не к одному запросу. Может случиться, что конкретный запрос будет снят гораздо раньше, потому что система уже долго ждет завершения какого-то другого запроса.

Команда DROP TABLE – очевидный пример конфликта, но далеко не единственный. Вот еще один:

```
BEGIN;
...
DELETE FROM tab WHERE id < 10000;
COMMIT;
...
VACUUM tab;
```

Снова предположим, что на резервном сервере выполняется длительная команда SELECT. Понятно, что команда DELETE не составляет проблемы, потому

что она только помечает строку как удаленную, а не удаляет ее на самом деле. COMMIT тоже не проблема, потому что она просто помечает транзакцию как завершенную. Физически строка все еще на месте.

Проблемы начинаются, когда в игру вступает команда VACUUM. Вот она-то уничтожает строку на диске. И разумеется, эти изменения попадают в журнал транзакций и рано или поздно доберутся до резервного сервера, у которого возникнут неприятности.

Чтобы предотвратить появление типичных проблем, характерных для стандартной рабочей нагрузки типа OLTP, был введен следующий конфигурационный параметр:

```
hot_standby_feedback = off
# резервный сервер будет отправлять информацию,
# чтобы предотвратить конфликты при выполнении запросов
```

Если этот параметр равен on, то резервный сервер периодически отправляет главному идентификатор самой старой транзакции. Тогда процесс очистки будет знать, что где-то в системе имеется более старая транзакция, и сдвинет возраст очистки на более поздний момент, когда строки будет безопасно стирать. По сути дела, параметр hot_standby_feedback приводит к такому же эффекту, как выполнение долгой транзакции на главном сервере.

По умолчанию параметр hot_standby_feedback равен off. Почему? Причина весьма основательная: если этот режим выключен, то резервный сервер никак не влияет на главный. Поточковая передача журнала транзакций не потребляет много ресурсов процессора, так что потоковая репликация эффективна и обходится недорого. Но если резервный сервер (который мы, возможно, даже не контролируем) слишком долго удерживает транзакции открытыми, то на главном сервере будет иметь место разбухание таблиц из-за отложенной очистки. В конфигурации по умолчанию это считается менее желательным, чем уменьшение числа конфликтов.

Если задать hot_standby_feedback = on, то обычно удастся предотвратить 99 % типичных для OLTP конфликтов, что особенно важно, если длительность транзакций превышает пару миллисекунд.

Повышение надежности репликации

Как мы видели в этой главе, настроить репликацию просто. Но бывают нетипичные случаи, приводящие к эксплуатационным сложностям. Один из них связан со сроком хранения журналов транзакций.

Рассмотрим следующий сценарий.

1. Создана базовая резервная копия.
2. После ее создания ничего не происходило в течение часа.
3. Затем запущен резервный сервер.

Главному серверу нет дела до существования резервного. Поэтому журнала транзакций, необходимого резервному серверу, может уже и не быть, поскольку

ку он удален процессом создания контрольной точки. Теперь для запуска резервного сервера необходима повторная синхронизация. Понятно, что в случае базы размером несколько терабайтов это серьезная проблема.

Потенциальное решение дает параметр `wal_keep_segments`:

`wal_keep_segments = 0` # измеряется в сегментах журнала длиной 16 МБ; 0 - выключить

По умолчанию PostgreSQL хранит достаточно сегментов журнала транзакций, чтобы восстановиться после неожиданной аварии, но не больше. Параметр `wal_keep_segments` позволяет сказать серверу, что нужно хранить больше данных, чтобы резервный сервер мог догнать главный, даже если сильно отстал.

Важно иметь в виду, что сервер может отстать не только потому, что он слишком медленный или слишком занятый. Зачастую запаздывание возникает из-за медленной сети. Предположим, что создается индекс над таблицей размера 1 ТБ; PostgreSQL сортирует данные, после чего уже построенный индекс помещается в журнал транзакций. А теперь представьте, что произойдет при попытке передать сотни мегабайт журнала по сети с пропускной способностью всего 1 Гб/с или около того. В результате может быть потеряно много гигабайт данных, и это произойдет в течение нескольких секунд. Поэтому параметр `wal_keep_segments` следует задавать в расчете не на типичную, а на максимальную задержку, еще устраивающую администратора (быть может, с некоторым запасом для пущей безопасности).

Задание достаточно большого значения `wal_keep_segments` более чем оправдано, и я рекомендую позаботиться о том, чтобы данных всегда хватало.

Альтернативное решение проблемы преждевременного удаления журналов транзакций дают слоты репликации, которые мы рассмотрим далее.

ПЕРЕХОД НА СИНХРОННУЮ РЕПЛИКАЦИЮ

Выше мы довольно подробно рассмотрели асинхронную репликацию. Но асинхронность означает, что фиксация транзакции на резервном сервере может происходить после ее фиксации на главном. Если главный сервер выйдет из строя, то данные, не дошедшие до резервного, могут быть потеряны, даже при настроенной репликации.

Проблему решает синхронная репликация – в этом случае зафиксированная транзакция должна попасть на диск хотя бы на одной реплике, чтобы транзакция считалась зафиксированной на главном сервере. Поэтому синхронная репликация значительно снижает вероятность потери данных.

В PostgreSQL настроить синхронную репликацию просто. Нужно сделать всего две вещи:

- изменить параметр `synchronous_standby_names` в файле `postgresql.conf` на главном сервере;
- добавить параметр `application_name` в строку `primary_conninfo` в файле `recovery.conf` на реплике.

Начнем с файла `postgresql.conf` на главном сервере:

```
synchronous_standby_names = ''
# резервные серверы, принимающие участие в синхронной репликации
# список синхронных резервных серверов, каждый из которых
# идентифицируется значением application_name в свойствах
# подключения; '*' = все
```

Если этот параметр задать равным '*', то все узлы будут рассматриваться как кандидаты на синхронную репликацию. Но на практике обычно указывают всего несколько узлов, например:

```
synchronous_standby_names = 'slave2, slave1, slave3'
```

Теперь нужно изменить файл `recovery.conf`, добавив `application_name`:

```
primary_conninfo = '... application_name=slave2'
```

С этого момента реплика будет подключаться к главному серверу под именем `slave2`. Главный сервер проверит свою конфигурацию и выяснит, что `slave2` первый в списке, поэтому является резервным сервером. Поэтому PostgreSQL будет считать фиксацию на главном сервере успешной, только если резервный сервер подтвердит, что зафиксировал ее.

Предположим теперь, что `slave2` по какой-то причине вышел из строя. Тогда PostgreSQL проверит остальные два сервера, перечисленных в списке `synchronous_standby_names`. Но что, если не осталось ни одного работающего сервера?

В таком случае PostgreSQL будет ждать фиксации вечно. Да-да, именно так – PostgreSQL не зафиксирует транзакцию, если нет хотя бы двух работоспособных узлов. Напомним, что мы просили PostgreSQL сохранять данные, по крайней мере, на двух узлах, и если мы не сможем обеспечить достаточное количество серверов в любой момент времени, то это наша вина. На практике это означает, что для синхронной репликации лучше всего иметь три узла – один главный и два резервных, – поскольку всегда есть шанс потерять один узел.

Раз уж мы заговорили об отказах серверов, то отметим еще один момент: если синхронный партнер выйдет из строя в процессе фиксации, то PostgreSQL будет ждать его возвращения. Или же может завершить синхронную фиксацию на каком-то другом потенциальном партнере. Конечный пользователь может даже не заметить, что партнер по синхронной репликации сменился.

Иногда хранить данные только на двух узлах недостаточно – для пущей безопасности мы можем потребовать, чтобы количество резервных узлов было больше. Для этого начиная с версии PostgreSQL 9.6 следует использовать такой синтаксис:

```
synchronous_standby_names =
'4(slave1, slave2, slave3, slave4, slave5, slave6)'
```

Это означает, что данные должны быть сохранены хотя бы на четырех из шести указанных серверов, только тогда главный сервер будет считать транзакцию успешно зафиксированной.

Разумеется, за все это надо платить – скорость работы снижается по мере добавления синхронных реплик. Бесплатных завтраков не бывает. Впрочем, PostgreSQL предлагает два способа держать производительность под контролем. Мы обсудим их в следующем разделе.

В версии PostgreSQL 10.0 добавлена новая функциональность:

```
[FIRST] число_синхронных ( имя_резервного [, ...] )
ANY число_синхронных ( имя_резервного [, ...] )
имя_резервного [, ...]
```

Добавлены ключевые слова ANY и FIRST. FIRST позволяет задавать приоритеты серверов, а ANY повышает гибкость в части фиксации синхронной транзакции.

Настройка долговечности

Как мы видели, данные можно реплицировать синхронно или асинхронно. Но это не глобальный режим. Для повышения производительности PostgreSQL допускает очень гибкие конфигурации. Конечно, никто не мешает реплицировать все синхронно либо асинхронно, но во многих случаях можно подойти к делу не столь прямолинейно. Для этого предназначен параметр `synchronous_commit`.

В предположении, что задана синхронная репликация, параметр `application_name` в файле `recovery.conf` и параметр `synchronous_standby_names` в файле `postgresql.conf`, параметр `synchronous_commit` открывает следующие возможности:

- `off`: это, по существу, асинхронная репликация. Журнал транзакций на главном сервере не сбрасывается на диск мгновенно, и главный сервер не ждет, пока резервный запишет данные на свой диск. Если главный сервер выйдет из строя, часть данных может быть потеряна (за время, в три раза превышающее `wal_writer_delay`);
- `local`: журнал транзакций сбрасывается на диск главного сервера в процессе фиксации. Но главный сервер не ждет резервного (асинхронная репликация);
- `remote_write`: в этом режиме PostgreSQL уже производит синхронную репликацию. Однако данные на диске сохраняет только главный сервер. Резервному достаточно передать данные операционной системе. Идея в том, чтобы не дожидаться сброса на второй диск и тем ускорить работу. Крайне маловероятно, что обе системы хранения выйдут из строя одновременно. Поэтому риск потери данных близок к нулю;
- `on`: в этом случае транзакция считается успешной, если и главный, и резервные серверы сбросили данные на диск. Приложение не получит подтверждения команды COMMIT, если данные не сохранены на двух серверах (или большем числе – в зависимости от конфигурации);
- `remote_apply`: режим `on` гарантирует сохранение данных на двух узлах, но не гарантирует возможности немедленно балансировать нагрузку. Тот факт, что данные сброшены на диск, еще не означает, что пользо-

ватель может их увидеть. Например, в случае конфликта резервный сервер остановит воспроизведение транзакции, но по-прежнему будет принимать журнал транзакций и записывать его на диск. Короче говоря, может случиться так, что данные сброшены на диск резервного сервера, хотя пользователь их еще не видит. Эту проблему исправляет режим `remote_apply`. Он требует, чтобы данные были видны на реплике, так что при следующем запросе на чтение на резервном сервере уже будут видны изменения, произведенные на главном сервере. Понятно, что режим `remote_apply` – самый медленный способ репликации данных, поскольку требуется, чтобы резервный сервер сделал изменения видимыми конечному пользователю.

В PostgreSQL параметр `synchronous_commit` не является глобальным. Как и многие другие параметры, его можно задавать на разных уровнях, например на уровне базы данных:

```
test=# ALTER DATABASE test SET synchronous_commit TO off;
ALTER DATABASE
```

Иногда определенный режим репликации следует задавать лишь для одной базы данных. Можно также задать синхронную репликацию только в сеансе определенного пользователя. И наконец, можно указать, как следует реплицировать одну конкретную транзакцию.

Рассмотрим, к примеру, два сценария:

- при записи в журнальную таблицу фиксация должна быть асинхронной, чтобы ускорить работу;
- при сохранении данных о платеже кредитной картой мы хотим обеспечить максимальную безопасность, поэтому транзакция должна быть синхронной.

Как видим, в одной и той же базе данных требования могут быть разными – в зависимости от того, какие данные изменяются. Поэтому изменение режима репликации на уровне транзакции весьма полезно для повышения быстродействия.

СЛОТЫ РЕПЛИКАЦИИ

Познакомившись с синхронной репликацией и динамическим изменением долговечности, обратимся к понятию слота репликации.

Рассмотрим пример: имеются главный и резервный серверы. На главном сервере выполняется большая транзакция, но скорость сетевого соединения недостаточна для доставки всех данных вовремя. В какой-то момент (во время создания контрольной точки) главный сервер удаляет журнал транзакций. Если резервный сервер отстал слишком далеко, то необходима повторная синхронизация. Как мы видели, параметр `wal_keep_segments` позволяет уменьшить риск ошибки репликации. Но возникает вопрос: каково оптимальное значение этого параметра? Понятно, что чем больше, тем лучше, но где остановиться?

Эту проблему решают слоты репликации. Если используется слот репликации, то главный сервер может стереть журнал транзакций только тогда, когда его обработали все реплики. Плюс в том, что резервный сервер никогда не сможет отстать настолько, что понадобится повторная синхронизация.

Но предположим, что мы остановили реплику, не уведомив главный сервер. Тогда главный сервер будет хранить журнал транзакций на своем диске вечно, и в конце концов диск заполнится, что приведет к остановке системы.

Чтобы уменьшить этот риск, слоты репликации следует использовать только в сочетании с налаженным мониторингом и системой оповещения. Необходимо следить за открытыми слотами репликации, которые могут вызвать проблемы или уже вообще не используются.

В PostgreSQL есть два типа слотов репликации: физические и логические. Физические слоты используются для стандартной потоковой репликации. Они гарантируют, что данные не будут уничтожены слишком рано. Логические слоты решают ту же задачу, но применяются для логического декодирования. Смысл логического декодирования заключается в том, чтобы дать пользователю возможность присоединиться к журналу транзакций и с помощью подключаемого модуля декодировать его. Таким образом, логический слот – это что-то вроде команды `tail -f` для экземпляров баз данных. Он позволяет узнавать об изменениях, произведенных в базе данных (и, следовательно, попавших в журнал транзакций), и представлять их в любом формате. Зачастую логический слот репликации используется для выполнения логической репликации.

Работа с физическими слотами репликации

Для использования слотов репликации в файл `postgresql.conf` нужно внести следующие изменения:

```
wal_level = logical
max_replication_slots = 5 # или любое другое число
```

Для физических слотов значение `logical` необязательно, достаточно и значения `replica`. Но для логических слотов необходимо более требовательное значение `wal_level`. Параметр `max_replication_slots` следует изменить, если вы работаете с версией PostgreSQL 9.6 или более ранней. В PostgreSQL 10.0 уже установлено более подходящее значение по умолчанию. В общем, задайте такое число, которое отвечает вашим целям. Я рекомендую задавать число слотов с запасом, чтобы можно было подключить дополнительных потребителей, не перезапуская сервер.

После перезапуска можно создать слот:

```
test=# \x
Расширенный вывод включен.
postgres=# \df *create*physical*slot*
Список функций
-[ RECORD 1 ]-----...
```

Схема	pg_catalog
Имя	pg_create_physical_replication_slot
Тип данных результата	record
Типы данных аргументов	slot_name name, immediately_reserve boolean DEFAULT false, temporary boolean DEFAULT false, OUT slot_name name, OUT lsn pg_lsn
Тип	func

Функция `pg_create_physical_replication_slot` создает слот. Ее можно вызывать с одним или двумя параметрами. Если задано только имя слота, то слот активируется при первом подключении клиента потоковой репликации. Если второй параметр задан и равен `true`, то слот сразу же начинает заботиться о сохранении журнала транзакций:

```
test=# SELECT * FROM pg_create_physical_replication_slot('some_slot_name', true);
      slot_name      | lsn
-----+-----
some_slot_name | 0/EF8AD1D8
(1 строка)
```

Чтобы получить список активных слотов на главном сервере, выполните следующую команду SQL:

```
test=# \x
Расширенный вывод включен.
test=# SELECT * FROM pg_replication_slots;
-[ RECORD 1 ]-----+-----
slot_name          | some_slot_name
plugin              |
slot_type           | physical
datoid              |
database            |
temporary           | f
active              | f
active_pid          |
xmin                |
catalog_xmin         |
restart_lsn          | 0/1653398
confirmed_flush_lsn |
```

Это представление может много рассказать о слоте: его тип, позиции в журнале транзакции и т. д.

Чтобы воспользоваться слотом, мы должны лишь добавить его в файл `recovery.conf`:

```
primary_slot_name = 'some_slot_name'
```

После перезапуска потоковой репликации слот начнет защищать ее. Если слот больше не нужен, его можно удалить:

```
test=# \df *drop*slot*
Список функций
```

```

-[ RECORD 1 ]-----+-----
Схема                | pg_catalog
Имя                  | pg_drop_replication_slot
Тип данных результата | void
Типы данных аргументов | name
Тип                  | normal

```

С точки зрения удаления, логический слот ничем не отличается от физического. Нужно лишь передать имя слота функции.



Запрещено использовать удаленный слот. В противном случае PostgreSQL выдаст ошибку.

Работа с логическими слотами репликации

Логические слоты репликации играют важнейшую роль в логической репликации. Из-за недостатка места невозможно осветить в этой главе все аспекты логической репликации. Но я все же хочу объяснить некоторые базовые концепции логического декодирования, а стало быть, и логической репликации.

Функция создания логического слота репликации принимает два параметра: имя слота и имя подключаемого модуля для декодирования журнала транзакций:

```

test=# SELECT *
        FROM pg_create_logical_replication_slot('logical_slot', 'test_decoding');
 slot_name |      lsn
-----+-----
logical_slot | 0/EF8AD4B0
(1 строка)

```

Для проверки существования слота можно воспользоваться той же командой, что и раньше. Чтобы понять, что в действительности делает слот, напишем небольшой тест:

```

test=# CREATE TABLE t_demo (id int, name text, payload text);
CREATE TABLE
test=# BEGIN;
BEGIN
test=# INSERT INTO t_demo
VALUES (1, 'hans', 'some data');
INSERT 0 1
test=# INSERT INTO t_demo VALUES (2, 'paul', 'some more data');
INSERT 0 1
test=# COMMIT;
COMMIT
test=# INSERT INTO t_demo VALUES (3, 'joe', 'less data');
INSERT 0 1

```

Как видим, выполнено две транзакции. Произведенные этими транзакциями изменения можно получить из слота:

```

test=# SELECT pg_logical_slot_get_changes('logical_slot', NULL, NULL);
           pg_logical_slot_get_changes

```

```

-----
(0/EF8AF5B0,606546,"BEGIN 606546")
(0/EF8CCCA0,606546,"COMMIT 606546")
(0/EF8CCCD8,606547,"BEGIN 606547")
(0/EF8CCCD8,606547,"table public.t_demo: INSERT: id[integer]:1
 name[text]:'hans' payload[text]:'some data'")
(0/EF8CCD60,606547,"table public.t_demo: INSERT: id[integer]:2
 name[text]:'paul' payload[text]:'some more data'")
(0/EF8CCDE0,606547,"COMMIT 606547")
(0/EF8CCE18,606548,"BEGIN 606548")
(0/EF8CCE18,606548,"table public.t_demo: INSERT: id[integer]:3
 name[text]:'joe' payload[text]:'less data'")
(0/EF8CCE98,606548,"COMMIT 606548")
(9 строк)

```

Формат зависит от заданного ранее подключаемого модуля. Для PostgreSQL есть несколько таких модулей, например `wal2json`.



По умолчанию логический поток содержит значения, а не просто функции. Это значения, которые оказались в полях базовых таблиц.

Также имейте в виду, что после того как данные один раз прочитаны, слот больше не возвращает их:

```

test=# SELECT pg_logical_slot_get_changes('logical_slot', NULL, NULL);
pg_logical_slot_get_changes
-----
(0 строк)

```

При втором обращении результирующий набор оказался пуст. На случай, если мы захотим выбрать данные несколько раз, PostgreSQL предлагает функцию `pg_logical_slot_peek_changes`. Она работает так же, как `pg_logical_slot_get_changes`, но не удаляет данные из слота.

Конечно, SQL – не единственный способ получить данные из журнала транзакций. Имеется также командная утилита `pg_recvlogical`. Это некий аналог команды `tail -f` для всего экземпляра, она получает поток данных в режиме реального времени:

```

[hs@zenbook ~]$ pg_recvlogical -S logical_slot -P test_decoding
-d test -U postgres --start -f -

```

В данном случае команда подключается к базе данных `test` и потребляет данные из слота `logical_slot`. Флаг `-f` означает, что поток направляется на `stdout`. Давайте удалим часть данных:

```

test=# DELETE FROM t_demo WHERE id < random()*10;
DELETE 3

```

Изменения попадают в журнал транзакций. Но по умолчанию базе данных интересно только, как будет выглядеть таблица после удаления. Она знает, какие блоки должны быть изменены и все такое, но не знает, что в них было раньше:

```
BEGIN 606549
table public.t_demo: DELETE: (no-tuple-data)
table public.t_demo: DELETE: (no-tuple-data)
table public.t_demo: DELETE: (no-tuple-data)
COMMIT 606549
```

Таким образом, вывод мало что нам дает. Чтобы исправить ситуацию, выполним такую команду:

```
test=# ALTER TABLE t_demo REPLICA IDENTITY FULL;
ALTER TABLE
```

Если в таблицу вставить и снова удалить данные, то поток журнала транзакций будет выглядеть следующим образом:

```
BEGIN 606558
table public.t_demo: DELETE: id[integer]:1 name[text]:'hans'
payload[text]:'some data'
table public.t_demo: DELETE: id[integer]:2 name[text]:'paul'
payload[text]:'some more data'
table public.t_demo: DELETE: id[integer]:3 name[text]:'joe'
payload[text]:'less data'
COMMIT 606558
```

Вот теперь мы видим все изменения.

Примеры применения логических слотов

Слоты репликации применяются для разных целей. Самый простой вариант – получить данные от сервера в желаемом формате и использовать для аудита, отладки или просто для мониторинга работы экземпляра.

Следующий шаг – взять поток изменений и использовать его для репликации. Такие решения, как **двусторонняя репликация** (Bi-Directional Replication – **BDR**), всецело основаны на логическом декодировании, поскольку двоичной репликации с несколькими главными серверами не существует.

Наконец, упомянем переход на новую версию без простоя. Напомним, что двоичную репликацию журнала транзакций нельзя использовать для реплицирования между разными версиями PostgreSQL. Поэтому в будущих версиях PostgreSQL будет поддерживаться программа `pglogical`, которая делает возможным смену версии без остановки сервера.

ИСПОЛЬЗОВАНИЕ КОМАНД CREATE PUBLICATION И CREATE SUBSCRIPTION

В версию 10.0 сообщество PostgreSQL включило две новые команды: `CREATE PUBLICATION` и `CREATE SUBSCRIPTION`. Их можно использовать для логической репликации, которая позволяет избирательно реплицировать данные и производить переход на новую версию почти без простоя. Выше мы рассмотрели двоичную репликацию и репликацию журнала транзакций. Но иногда мы хотим репли-

цировать не всю базу данных, а всего лишь пару-тройку таблиц. Именно для этого и может пригодиться логическая репликация.

Прежде всего изменим значение параметра `wal_level` в файле `postgresql.conf` на `logical` и перезапустим сервер:

```
wal_level = logical
```

Затем создадим простую таблицу:

```
test=# CREATE TABLE t_test (a int, b int);
CREATE TABLE
```

Таблица с такой же структурой должна существовать во второй базе данных. Автоматически PostgreSQL ее не создаст.

```
test=# CREATE DATABASE repl;
CREATE DATABASE
```

```
repl=# CREATE TABLE t_test (a int, b int);
CREATE TABLE
```

Наша цель – опубликовать содержимое таблицы `t_test` в базе данных `test` для заинтересованной стороны. Для публикации изменений PostgreSQL предлагает команду `CREATE PUBLICATION`:

```
test=# \h CREATE PUBLICATION
Команда: CREATE PUBLICATION
Описание: создать публикацию
Синтаксис:
CREATE PUBLICATION имя
    [ FOR TABLE [ ONLY ] имя_таблицы [ * ] [, ...]
    | FOR ALL TABLES ]
    [ WITH ( параметр_публикации [= значение] [, ...] ) ]
```

Синтаксис простой. Нужно только задать имя публикации и список реплицируемых таблиц:

```
test=# CREATE PUBLICATION pub1 FOR TABLE t_test;
CREATE PUBLICATION
```

На следующем шаге можно создавать подписку. Синтаксис снова незамысловатый:

```
test=# \h CREATE SUBSCRIPTION
Команда: CREATE SUBSCRIPTION
Описание: создать подписку
Синтаксис:
CREATE SUBSCRIPTION имя_подписки
    CONNECTION 'строка_подключения'
    PUBLICATION имя_публикации [, ...]
    [ WITH ( параметр_подписки [= значение] [, ...] ) ]
```

Вообще-то, непосредственное создание подписки не вызывает никаких проблем. Но если мы собираемся делать все это в пределах одного экземпляра

для репликации базы данных `test` в базу данных `repl`, то необходимо вручную создать слот репликации. Иначе команда `CREATE SUBSCRIPTION` никогда не завершится:

```
test=# SELECT pg_create_logical_replication_slot('sub1', 'pgoutput');
pg_create_logical_replication_slot
-----
(sub1,0/27E2B2D0)
(1 строка)
```

В данном случае мы создали в главной базе данных слот `sub1`. Далее мы должны подключиться к целевой базе данных и выполнить такую команду:

```
repl=# CREATE SUBSCRIPTION sub1
      CONNECTION 'host=localhost dbname=test user=postgres'
      PUBLICATION pub1
      WITH (create_slot = false);
CREATE SUBSCRIPTION
```

Конечно, вы должны подставить в эту команду собственные параметры подключения. После этого PostgreSQL синхронизирует данные, и все будет готово к работе.

i Заметим, что часть `create_slot = false` указана только потому, что база данных `test` находится в том же экземпляре, что и `repl`. Если бы базы находились в разных экземплярах, то не пришлось бы ни создавать слот вручную, ни указывать `create_slot = false`.

РЕЗЮМЕ

В этой главе мы узнали о наиболее важных чертах репликации в PostgreSQL, в частности о потоковой репликации и конфликтах репликации. Затем мы познакомились с восстановлением на определенный момент времени и со слотами репликации. Заметим, что для полного изложения темы репликации понадобилось бы порядка 400 страниц, но мы все же рассказали о самых главных вещах, которые должен знать любой администратор.

В главе 11 речь пойдет о полезных расширениях PostgreSQL. Мы расскажем о расширениях, широко используемых в отрасли и наделяющих PostgreSQL еще большей функциональностью.

Вопросы

В чем цель логической репликации?

В случае двоичной репликации на главном и резервном серверах должна работать одна и та же основная версия PostgreSQL. Иными словами, перейти с версии PostgreSQL 10 на версию PostgreSQL 11, реплицируя журнал транзакций потоком, невозможно. Логическая репликация помогает закрыть этот пробел. Кроме того, логическую репликацию можно применить для избирательной репликации таблиц.

Как синхронная репликация влияет на производительность?

Синхронная репликация медленнее асинхронной. Особенно это заметно в случае длительных транзакций. Трудно точно оценить снижение производительности, поскольку это существенно зависит от сетевой задержки. Чем медленнее сеть, тем медленнее будет работать синхронная репликация.

Почему не пользоваться только синхронной репликацией?

Если хотите добиться высокого быстродействия и доступности системы, то старайтесь использовать синхронную репликацию, только когда это абсолютно необходимо. В большинстве случаев асинхронной репликации вполне достаточно.

Глава 11

Полезные расширения

Глава 10 была посвящена репликации, трансляции журнала транзакций и логическому декодированию. Рассмотрев темы, интересные главным образом администратору, мы теперь обратимся к более широкому контексту. В мире PostgreSQL многое делается с помощью расширений. Достоинство их в том, что функциональность можно расширить, не раздувая ядро PostgreSQL. Пользователи могут сами выбрать, какие расширения (иногда конкурирующие между собой) им больше подходят. Философия PostgreSQL – сохранять ядро компактным, простым для сопровождения и готовым к будущему.

В этой главе мы обсудим некоторые наиболее распространенные расширения PostgreSQL. Сразу хочу предупредить, что рассматриваются только расширения, которые лично я считаю полезными. В наши дни модули так разплодились, что описать их все сколько-нибудь подробно попросту невозможно. Новые модули публикуются каждый день, и иногда даже профессионалу трудно оставаться в курсе событий. Рекомендую заглянуть на сайт PGXN (<https://pgxn.org/>), где находится множество расширений PostgreSQL.

В этой главе будут рассмотрены следующие вопросы:

- как работают расширения;
- подборка модулей contrib;
- краткое знакомство с модулями, относящимися к ГИС;
- другие полезные расширения.

Отметим, что упомянуты только самые важные расширения.

КАК РАБОТАЮТ РАСШИРЕНИЯ

Прежде чем рассказывать об имеющихся расширениях, было бы полезно понять, как они вообще устроены и работают.

Сначала синтаксис:

test=# \h CREATE EXTENSION

Команда: CREATE EXTENSION

Описание: установить расширение

Синтаксис:

```
CREATE EXTENSION [ IF NOT EXISTS ] имя_расширения  
[ WITH ] [ SCHEMA имя_схемы ]
```

```
[ VERSION версия ]
[ FROM старая_версия ]
[ CASCADE ]
```

Чтобы развернуть расширение, просто выполните команду `CREATE EXTENSION`. Она проверит наличие расширения и загрузит его в базу данных. Отметим, что расширение загружается в конкретную базу, а не в экземпляр в целом.

При загрузке расширения можно указать схему. Многие расширения можно перемещать, так чтобы пользователь мог выбрать схему, которая ему удобнее. Далее можно указать конкретную версию расширения. Зачастую мы не хотим устанавливать последнюю версию, поскольку на стороне клиента может работать устаревшее ПО. В таком случае удобно иметь возможность загрузить любую из версий, имеющихся в системе.

О фразе `FROM старая_версия` стоит поговорить подробнее. Когда-то давно PostgreSQL не поддерживала расширений, поэтому в сети все еще можно найти россыпи не оформленного в виде пакета кода. Эта фраза заставляет команду `CREATE EXTENSION` выполнить альтернативный скрипт установки, который конструирует расширение из существующих объектов, а не создает новые. В этом случае во фразе `SCHEMA` должна быть указана схема, содержащая эти уже существующие объекты. Пользуйтесь этой возможностью, только когда в системе имеются старые модули.

И наконец, фраза `CASCADE`. Бывает так, что одно расширение зависит от других. При наличии фразы `CASCADE` будут автоматически установлены все необходимые пакеты. Например:

```
test=# CREATE EXTENSION earthdistance;
ОШИБКА: требуемое расширение "cube" не установлено
ПОДСКАЗКА: Выполните CREATE EXTENSION ... CASCADE, чтобы установить также требуемые расширения.
```

Модуль `earthdistance` вычисляет расстояние по большой окружности. Как вы, наверное, знаете, кратчайшее расстояние между двумя точками на поверхности Земли – не прямая; пилот должен постоянно корректировать курс, чтобы найти самый быстрый маршрут перелета из одной точки в другую. Но нам важно, что расширение `earthdistance` зависит от расширения `cube`, позволяющего выполнять математические операции на сфере.

Чтобы автоматически установить зависимость, мы можем добавить фразу `CASCADE`:

```
test=# CREATE EXTENSION earthdistance CASCADE;
ЗАМЕЧАНИЕ: установка требуемого расширения "cube"
CREATE EXTENSION
```

Теперь установлены оба расширения.

Проверка доступных расширений

PostgreSQL предлагает различные способы узнать, какие расширения имеются в системе и какие из них установлены. Один из них – системное представле-

ние `pg_available_extensions`. Оно содержит список всех доступных расширений, включающий имя, номер версии по умолчанию и номер установленной версии. Для удобства пользователя приведено также краткое описание расширения.

В следующей распечатке показаны две строки из представления `pg_available_extensions`:

```
test=# \x
Расширенный вывод включен.
test=# SELECT * FROM pg_available_extensions LIMIT 2;
-[ RECORD 1 ]-----+-----
name           | unaccent
default_version | 1.1
installed_version |
comment        | text search dictionary that removes accents
-[ RECORD 2 ]-----+-----
name           | tsm_system_time
default_version | 1.0
installed_version |
comment        | TABLESAMPLE method which accepts time in milliseconds as a limit
```

В моей базе данных установлены расширения `earthdistance` и `plpgsql`. Расширение `plpgsql` устанавливается по умолчанию, а `earthdistance` было добавлено мной. Это представление полезно тем, что позволяет легко узнать, что уже установлено, а что можно установить.

В некоторых случаях предлагается несколько версий расширения. Чтобы получить дополнительные сведения о версиях, можно воспользоваться следующим представлением:

```
test=# \d pg_available_extension_versions
Представление "pg_catalog.pg_available_extension_versions"
  Столбец   | Тип      | Модификаторы
-----+-----+-----
name       | name     |
version    | text     |
installed  | boolean  |
superuser  | boolean  |
relocatable | boolean  |
schema     | name     |
requires   | name[]   |
comment    | text     |
```

Оно содержит более подробную информацию, как видно из следующей распечатки:

```
test=# SELECT * FROM pg_available_extension_versions LIMIT 1;
-[ RECORD 1 ]-----+-----
name          | earthdistance
version       | 1.1
installed     | t
superuser     | t
relocatable   | t
```

schema	
requires	{cube}
comment	calculate great-circle distances on the surface of the Earth

PostgreSQL также сообщает, можно ли перемещать расширение, в какую схему оно установлено и от каких расширений зависит. Имеется еще поле `comment`, которое было рассмотрено выше.

Спрашивается, откуда PostgreSQL получает всю информацию о расширениях, имеющихся в системе? В предположении, что вы брали PostgreSQL 11.0 из официального RPM-репозитория PostgreSQL, в каталоге `/usr/pgsql-11/share/extension` будет находиться ряд файлов:

```
...
-bash-4.3$ ls -l citext*
ls -l citext*
-rw-r--r--. 1 hs hs 1028 Sep 11 19:53 citext--1.0--1.1.sql
-rw-r--r--. 1 hs hs 2748 Sep 11 19:53 citext--1.1--1.2.sql
-rw-r--r--. 1 hs hs 307 Sep 11 19:53 citext--1.2--1.3.sql
-rw-r--r--. 1 hs hs 668 Sep 11 19:53 citext--1.3--1.4.sql
-rw-r--r--. 1 hs hs 2284 Sep 11 19:53 citext--1.4--1.5.sql
-rw-r--r--. 1 hs hs 13466 Sep 11 19:53 citext--1.4.sql
-rw-r--r--. 1 hs hs 158 Sep 11 19:53 citext.control
-rw-r--r--. 1 hs hs 9781 Sep 11 19:53 citext--unpackaged--1.0.sql
...
```

У расширения `citext` (текст, нечувствительный к регистру) версия по умолчанию имеет номер 1.4, поэтому имеется файл `citext--1.4.sql`. Кроме того, имеются файлы для перехода с одной версии на другую ($1.0 \rightarrow 1.1$, $1.1 \rightarrow 1.2$ и т. д.), а также файл с расширением `.control`:

```
-bash-4.3$ cat citext.control
# citext extension
comment = 'data type for case-insensitive character strings'
default_version = '1.4'
module_pathname = '$libdir/citext'
relocatable = true
```

Этот файл содержит метаданные расширения. В первой записи находится комментарий. Именно он показывается в рассмотренных ранее системных представлениях. Когда мы обращаемся к этим представлениям, PostgreSQL заходит в этот каталог и читает все присутствующие в нем `control`-файлы. Помимо комментария, `control`-файл содержит номер версии по умолчанию и путь к исполняемому файлам.

При установке типичного расширения из RPM-репозитория файлы копируются в каталог `$libdir`, совпадающий с подкаталогом `lib` установочного каталога PostgreSQL. Но если вы написали расширение сами, то можете поместить его в другое место.

Последний параметр сообщает PostgreSQL, может ли расширение находиться в любой схеме или только в предопределенной.

Наконец, в каталоге имеется файл, имя которого содержит строку `unpackaged`. Приведем его фрагмент:

```
...
ALTER EXTENSION citext ADD type citext;
ALTER EXTENSION citext ADD function citextin(cstring);
ALTER EXTENSION citext ADD function citextout(citext);
ALTER EXTENSION citext ADD function citextrecv(internal);
...
```

Этот файл преобразует существующий код в расширение. Он, таким образом, важен для того, чтобы собрать имеющиеся в базе объекты вместе.

ИСПОЛЬЗОВАНИЕ МОДУЛЕЙ ИЗ ПОДБОРКИ CONTRIB

Познакомившись с тем, что такое расширения вообще, взглянем на некоторые важные конкретные расширения, представленные в каталоге contrib. Я рекомендую всегда устанавливать эти модули вместе с PostgreSQL, поскольку они содержат расширения, способные сделать вашу жизнь существенно проще.

В следующем разделе дается обзор тех из них, которые я считаю наиболее интересными и полезными по той или иной причине (для отладки, для оптимизации производительности и т. п.).

Модуль adminpack

Модуль adminpack дает администраторам способ обратиться к файловой системе, не настраивая доступа по SSH. Для этого пакет содержит несколько функций.

Чтобы загрузить модуль в базу данных, выполните команду:

```
test=# CREATE EXTENSION adminpack;
CREATE EXTENSION
```

Одно из самых интересных средств модуля adminpack – возможность просматривать файлы журналов. Функция pg_logdir_ls возвращает список log-файлов из каталога log¹:

```
test=# SELECT * FROM pg_catalog.pg_logdir_ls() AS (a timestamp, b text);
ОШИБКА: параметр log_filename должен быть равен 'postgresql-%Y-%m-%d_%H%M%S.log'
```

Здесь говорится, что параметр log_filename должен быть выставлен в соответствии с требованиями модуля adminpack. Если вы устанавливали дистрибутив из RPM-архива, находящегося в репозиториях PostgreSQL, то параметр log_filename определен как postgresql-%a, и это нужно изменить, чтобы исправить ошибку.

После этого возвращается список имен файлов журналов:

```
test=# SELECT * FROM pg_catalog.pg_logdir_ls() AS (a timestamp, b text);
      a      |      b
-----+-----
2017-03-03 16:32:58 | pg_log/postgresql-2017-03-03_163258.log
(1 строка)
```

¹ В PostgreSQL 10 в ядре появилась аналогичная функция pg_ls_logdir. – *Прим. ред.*

Можно также определить размер файла на диске, например:

```
test=# SELECT b, pg_catalog.pg_file_length(b)
        FROM pg_catalog.pg_logdir_ls() AS (a timestamp, b text);
           b                | pg_file_length
-----+-----
pg_log/postgresql-2017-03-03_163258.log | 1525
(1 строка)
```

Модуль предоставляет и другие функции:

```
test=# SELECT proname FROM pg_proc WHERE proname ~ 'pg_file_*';
   proname
-----
pg_file_write
pg_file_rename
pg_file_unlink
pg_file_read
pg_file_length
(5 строк)
```

Как видим, мы получаем возможность читать, записывать, переименовывать и удалять файлы.



Разумеется, все эти функции может вызывать только суперпользователь.

Применение фильтра Блума

Начиная с версии PostgreSQL 9.6 стало возможно с помощью расширений динамически добавлять полнофункциональные типы индексов с поддержкой транзакционности. Для этого используется новая команда `CREATE ACCESS METHOD` в сочетании с некоторыми дополнительными средствами.

Расширение `bloom` предоставляет в распоряжение пользователей PostgreSQL фильтр Блума, который позволяет эффективно уменьшать объем данных на ранних стадиях. Идея фильтра Блума заключается в том, чтобы вычислить битовую маску данных и сравнить ее с запросом. Иногда фильтр Блума возвращает ложноположительные результаты, но все равно объем данных значительно сокращается.

Это особенно полезно, когда таблица содержит сотни столбцов и миллионы строк. С помощью В-деревьев проиндексировать сотни столбцов невозможно, поэтому фильтр Блума является подходящей альтернативой, т. к. позволяет проиндексировать сразу все.

Чтобы понять, как это работает, установим расширение:

```
test=# CREATE EXTENSION bloom;
CREATE EXTENSION
```

Далее создадим таблицу с несколькими столбцами:

```
test=# CREATE TABLE t_bloom
(
```

```
id serial,
col1 int4 DEFAULT random() * 1000,
col2 int4 DEFAULT random() * 1000,
col3 int4 DEFAULT random() * 1000,
col4 int4 DEFAULT random() * 1000,
col5 int4 DEFAULT random() * 1000,
col6 int4 DEFAULT random() * 1000,
col7 int4 DEFAULT random() * 1000,
col8 int4 DEFAULT random() * 1000,
col9 int4 DEFAULT random() * 1000
);
CREATE TABLE
```

Для простоты у каждого столбца есть значение по умолчанию, так что заполнить таблицу позволяет следующая простая команда SELECT:

```
test=# INSERT INTO t_bloom (id)
SELECT * FROM generate_series(1, 1000000);
INSERT 0 1000000
```

Этот запрос добавляет в таблицу миллион строк. Теперь проиндексируем таблицу:

```
test=# CREATE INDEX idx_bloom ON t_bloom
USING bloom(col1, col2, col3, col4, col5, col6, col7, col8, col9);
CREATE INDEX
```

Обратите внимание, что индекс содержит все девять столбцов. В отличие от В-дерева, порядок столбцов роли не играет.

Созданная таблица без индексов занимает примерно 65 МБ. Индекс добавляет еще 15 МБ:

```
test=# \di+ idx_bloom
```

Список отношений						
Схема	Имя	Тип	Владелец	Таблица	Размер	Описание
public	idx_bloom	index	hs	t_bloom	15 MB	

(1 строка)

Прелесть фильтра Блума в том, что он позволяет искать по любой комбинации столбцов:

```
test=# explain SELECT count(*)
FROM t_bloom
WHERE col4 = 454 AND col3 = 354 AND col9 = 423;
QUERY PLAN
-----
Aggregate (cost=20352.02..20352.03 rows=1 width=8)
-> Bitmap Heap Scan on t_bloom
    (cost=20348.00..20352.02 rows=1 width=0)
    Recheck Cond: ((col3 = 354) AND (col4 = 454) AND (col9 = 423))
    -> Bitmap Index Scan on idx_bloom
        (cost=0.00..20348.00 rows=1 width=0)
        Index Cond: ((col3 = 354) AND (col4 = 454) AND (col9 = 423))
(5 строк)
```


То, что мы сейчас видели, выглядит поразительно. Возникает естественный вопрос: а почему мы не пользуемся фильтром Блума всегда? Причина простая – чтобы воспользоваться фильтром Блума, сервер должен целиком прочитывать его в память. В случае В-дерева (и не только) это необязательно.

В будущем, вероятно, будут добавлены новые типы индексов, так что PostgreSQL можно будет использовать для решения еще большего числа задач. Желаящие больше узнать о фильтрах Блума могут ознакомиться со статьей в нашем блоге по адресу <https://www.cybertec-postgresql.com/en/trying-out-postgres-bloom-indexes/>.

Установка btree_gist и btree_gin

В PostgreSQL существует концепция классов операторов, которую мы обсуждали в главе 3, посвященной индексам.

В каталоге contrib есть два расширения (btree_gist и btree_gin), которые наделяют индексы типа GiST и GIN функциональностью В-дерева. Для чего это нужно? GiST-индексы поддерживают разнообразные свойства, отсутствующие у В-деревьев. Одно из них – поиск **К ближайших соседей** (K-Nearest Neighbor – **KNN**).

Ну и что? – спросите вы. Представьте, что требуется найти данные, добавленные вчера в районе полудня. Так когда же это было? Иногда трудно определить границы, например если кто-то ищет товар стоимостью около 70 евро. На помощь может прийти KNN.

Создадим тестовую таблицу:

```
test=# CREATE TABLE t_test (id int);
CREATE TABLE
```

Поместим в нее простые данные:

```
test=# INSERT INTO t_test SELECT * FROM generate_series(1, 100000);
INSERT 0 100000
```

Добавим расширение:

```
test=# CREATE EXTENSION btree_gist;
CREATE EXTENSION
```

Построить по столбцу GiST-индекс просто – нужно лишь добавить фразу USING gist. Заметим, что создать GiST-индекс по столбцу типа integer можно, только если установлено расширение btree_gist. В противном случае PostgreSQL сообщит, что нет подходящего класса операторов:

```
test=# CREATE INDEX idx_id ON t_test USING gist(id);
CREATE INDEX
```

Имея индекс, можно упорядочить данные по расстоянию:

```
test=# SELECT *
FROM t_test
ORDER BY id <-> 100
```

```
LIMIT 6;
id
-----
100
101
99
102
98
97
(6 строк)
```

Как видим, первая строка дает точное соответствие. В последующих строках точность уменьшается – чем дальше, тем хуже. Запрос возвращает фиксированное число строк.

Посмотрим на план выполнения – это очень важно:

```
test=# EXPLAIN SELECT *
      FROM t_test
      ORDER BY id <-> 100
      LIMIT 6;

                                QUERY PLAN
-----
Limit (cost=0.28..0.64 rows=6 width=8)
  -> Index Only Scan using idx_id on t_test
      (cost=0.28..5968.28 rows=100000 width=8)
    Order By: (id <-> 100)
(3 строки)
```

Мы видим, что PostgreSQL выбирает просмотр по индексу, что значительно ускоряет выполнение запроса.

В будущих версиях PostgreSQL B-деревья, скорее всего, будут поддерживать поиск KNN. Дополнение, содержащее эту возможность, уже включено в список рассылки для разработчиков. Быть может, в конечном итоге оно войдет и в ядро. Поддержка KNN B-деревьями позволила бы реже строить GiST-индексы по столбцам, имеющим стандартные типы данных.

Dblink – пора расстаться

Желание связывать разные базы данных существовало много лет. Но на рубеже столетий обертки внешних данных даже не просматривались на горизонте, как, впрочем, и традиционная реализация связи между базами данных. Примерно в это время разработчик из Калифорнии (Джо Конвей) начал работу по связыванию баз данных, включив в PostgreSQL концепцию dblink. Расширение dblink верой и правдой служило людям на протяжении многих лет, но теперь уже не является последним словом.

Поэтому мы рекомендуем отказываться от dblink и переходить на более современную реализацию спецификации SQL/MED, которая определяет, как следует интегрировать внешние данные с реляционной базой. Расширение `postgres_fdw` построено в соответствии с SQL/MED и предлагает не только под-

ключение к другой базе данных PostgreSQL, но и практически к любому источнику данных.

Доступ к файлам с помощью file_fdw

Иногда возникает необходимость прочитать файл с диска и представить его PostgreSQL в виде таблицы. Именно для этого служит расширение file_fdw.

Модуль устанавливается как обычно:

```
CREATE EXTENSION file_fdw;
```

Далее мы создадим виртуальный сервер:

```
CREATE SERVER file_server
FOREIGN DATA WRAPPER file_fdw;
```

Сервер file_server основан на обертке внешних (сторонних) данных file_fdw, которая наделяет PostgreSQL возможностью доступа к файлам.

Чтобы представить файл в виде таблицы, выполним такую команду:

```
CREATE FOREIGN TABLE t_passwd
(
    username text,
    passwd text,
    uid int,
    gid int,
    gecos text,
    dir text,
    shell text
) SERVER file_server
OPTIONS (format 'text', filename '/etc/passwd', header 'false', delimiter ':');
```

В этом примере читается файл /etc/passwd. Должны быть перечислены все его поля и для каждого указан подходящий тип. Дополнительная информация передается модулю во фразе OPTIONS. В данном случае PostgreSQL необходимо знать тип файла (текстовый), путь к нему и символ-разделитель. Также можно сказать, имеется ли в файле заголовок. Если да, то первая строка пропускается. Это важно при чтении файлов в формате CSV.

Создав таблицу, мы можем прочитать данные:

```
SELECT * FROM t_passwd;
```

PostgreSQL, естественно, возвращает содержимое файла /etc/passwd:

```
test=# \x
Расширенный вывод включен.
test=# SELECT * FROM t_passwd LIMIT 1;
-[ RECORD 1 ]-----
username | root
passwd   | x
uid      | 0
gid      | 0
```

```
gecos    | root
dir      | /root
shell    | /bin/bash
```

В плане выполнения присутствует шаг «Foreign Scan» (просмотр внешних данных) для извлечения данных из файла:

```
test=# EXPLAIN (verbose true, analyze true) SELECT * FROM t_passwd;
               QUERY PLAN
```

```
-----
Foreign Scan on public.t_passwd (cost=0.00..2.80 rows=18 width=168)
  (actual time=0.022..0.072 rows=61 loops=1)
    Output: username, passwd, uid, gid, gecos, dir, shell
    Foreign File: /etc/passwd
    Foreign File Size: 3484
Planning time: 0.058 ms
Execution time: 0.138 ms
(6 строк)
```

План выполнения также сообщает размер файла и прочие сведения. Раз уж мы говорим о планировщике, уместно сделать одно замечание. PostgreSQL читает даже статистические данные о файле. Планировщик определяет размер файла и назначает файлу такую же стоимость, какую назначил бы обычной таблице такого размера.

Анализ хранилища с помощью pageinspect

Столкнувшись с повреждением хранилища или иной проблемой, которая может быть вызвана дефектными блоками в таблице, можно призвать на помощь расширение `pageinspect`. Сначала установим расширение:

```
test=# CREATE EXTENSION pageinspect;
CREATE EXTENSION
```

Модуль `pageinspect` позволяет изучить таблицу на двоичном уровне. При работе с этим модулем прежде всего нужно выбрать блок:

```
test=# SELECT * FROM get_raw_page('pg_class', 0);
...
```

Этот вызов возвращает один блок – первый блок системной таблицы `pg_class` (вы, конечно, можете взять любую другую таблицу).

Затем получим заголовок страницы:

```
test=# \x
Расширенный вывод включен.
test=# SELECT * FROM page_header(get_raw_page('pg_class', 0));
-[ RECORD 1 ]-----
lsn          | 1/35CAE5B8
checksum     | 0
flags       | 1
lower       | 240
```

```
upper      | 1288
special    | 8192
pagesize   | 8192
version     | 4
prune_xid  | 606562
```

Уже в нем содержится много информации о странице. Но если этого недостаточно, то можно вызвать функцию `hear_page_items`, которая возвращает по одной строке на кортеж:

```
test=# SELECT * FROM heap_page_items(get_raw_page('pg_class', 0))
LIMIT 1;
-[ RECORD 1 ]---
lp           | 1
lp_off       | 49
lp_flags     | 2
lp_len       | 0
t_xmin       |
t_xmax       |
t_field3     |
t_ctid       |
t_infomask2  |
t_infomask   |
t_hoff       |
t_bits       |
t_oid        |
t_data       | ...
```

Можно также разбить данные на кортежи:

```
test=# SELECT tuple_data_split('pg_class'::regclass,  
                                t_data, t_infomask, t_infomask2, t_bits)  
        FROM heap_page_items(get_raw_page('pg_class', 0))  
        LIMIT 2;  
-[ RECORD 1 ]-----+-----  
tuple_data_split |  
-[ RECORD 2 ]-----+-----  
tuple_data_split |  
{ "\\x6100000000000000000000000000000000000000000000000000000000000000"  
0000000000000000000000000000000000000000000000000000000000000000", "\\x98080000", "  
\\x50ac0c00", "\\x00000000", "\\x01400000", "\\x00000000", "\\x4eac0c00", "\\x00  
000000", "\\xbb010000", "\\x0050c347", "\\x00000000", "\\x00000000", "\\x01", "\\  
x00", "\\x70", "\\x72", "\\x0100", "\\x0000", "\\x00", "\\x00", "\\x00", "\\x00", "\\  
x00", "\\x00", "\\x00", "\\x01", "\\x64", "\\xc3400900", "\\x01000000", NULL, NULL  
}
```

Для интерпретации этих данных необходимо понимать формат хранения данных на диске в PostgreSQL, иначе это выглядит абракадаброй.

Расширение `pageinspect` предоставляет функции для всех методов доступа (к таблицам, к индексам и т. д.), позволяя очень подробно проанализировать хранилище.

Анализ кеша с помощью pg_buffercache

После краткого знакомства с расширением `pageinspect` перейдем к расширению `pg_buffercache`, которое позволяет заглянуть внутрь кеша ввода-вывода:

```
test=# CREATE EXTENSION pg_buffercache;
CREATE EXTENSION
```

В этом расширении имеется представление, содержащее несколько полей:

```
test=# \d pg_buffercache
Представление "public.pg_buffercache"
  Столбец      | Тип      | Модификаторы
-----+-----+-----
bufferid       | integer  |
relfilenode    | oid      |
reltablespace  | oid      |
reldatabase    | oid      |
relforknumber  | smallint |
relblocknumber | bigint   |
isdirty        | boolean  |
usagescount    | smallint |
pinning_backends | integer  |
```

Поле `bufferid` – числовой идентификатор буфера. Поле `relfilenode` указывает на файл на диске. Если мы хотим узнать, какой таблице он принадлежит, то должны обратиться к таблице `pg_class`, которая также содержит поле `relfilenode`. Далее следуют поля `reldatabase` и `reltablespace`. Отметим, что все три поля имеют тип `oid`, поэтому для получения полезной информации нужно произвести соединение с системными таблицами.

Поле `relforknumber` говорит, какая часть таблицы кеширована. Это может быть куча, карта свободного места или какая-то другая компонента, например карта видимости. В будущем наверняка появятся новые типы слоев отношения.

Следующее поле, `relblocknumber`, говорит, какой блок кеширован в данном буфере. Наконец, поле `isdirty` показывает, был ли модифицирован блок, поле `usagescount` – сколько раз к нему обращались, а поле `pinning_backends` – количество обслуживающих процессов, закрепивших этот буфер.

Для осмысленного использования расширения `pg_buffercache` необходима дополнительная информация. Чтобы узнать, в какой базе данных кеширование применяется наиболее интенсивно, поможет следующий запрос:

```
test=# SELECT datname,
       count(*),
       count(*) FILTER (WHERE isdirty = true) AS dirty
FROM pg_buffercache AS b, pg_database AS d
WHERE d.oid = b.reldatabase
GROUP BY ROLLUP (1);
 datname | count | dirty
-----+-----+-----
abc      |    132 |      1
postgres |     30 |      0
```

```
test      | 11975 | 53
          | 12137 | 54
(4 строки)
```

В данном случае мы соединили `pg_buffercache` с таблицей `pg_database`. Соединение производится со столбцом `oid`, что может показаться неочевидным начинающим пользователям PostgreSQL.

Иногда нужно узнать, какие блоки конкретной базы данных кешированы:

```
test=# SELECT relname,
        relkind,
        count(*),
        count(*) FILTER (WHERE isdirty = true) AS dirty
FROM pg_buffercache AS b, pg_database AS d, pg_class AS c
WHERE d.oid = b.reldatabase
      AND c.relfilenode = b.relfilenode
      AND datname = 'test'
GROUP BY 1, 2
ORDER BY 3 DESC
LIMIT 7;
```

relname	relkind	count	dirty
t_bloom	r	8338	0
idx_bloom	i	1962	0
idx_id	i	549	0
t_test	r	445	0
pg_statistic	r	90	0
pg_depend	r	60	0
pg_depend_reference_index	i	34	0

(7 строк)

В этом примере мы выбрали базу данных и произвели соединение с отношением `pg_class`, содержащим список объектов. Особенно интересен столбец `relkind`: буква `r` означает таблицу (relation), а буква `i` – индекс (index).

Шифрование данных с помощью `pgcrypto`

`Pgcrypto` – одно из самых мощных расширений в каталоге `contrib`. Первоначально оно было написано одним из системных администраторов Skype и предлагает бесчисленные функции для шифрования и дешифрирования данных.

Имеются функции как для симметричного, так и для асимметричного шифрования. Поскольку функций очень много, рекомендую обратиться к документации по адресу <https://www.postgresql.org/docs/current/static/pgcrypto.html>¹.

Из-за нехватки места мы не можем вдаваться в детали расширения `pgcrypto`.

Прогрев кеша с помощью `pg_prewarm`

В процессе нормальной работы PostgreSQL стремится кешировать важные данные. Параметр `shared_buffers` определяет размер кеша PostgreSQL. Проблема в том, что после перезапуска сервера весь кеш теряется. Конечно, операцион-

¹ На русском языке <https://postgrespro.ru/docs/postgresql/11/pgcrypto>. – Прим. перев.

ная система еще может хранить в своем кеше какие-то данные, уменьшающие время ожидания диска, но, как правило, этого недостаточно. Решение предлагает расширение `pg_prewarm`.

```
test=# CREATE EXTENSION pg_prewarm;
CREATE EXTENSION
```

Это расширение содержит функцию, позволяющую явно *прогреть* кеш при необходимости:

```
test=# \x
Расширенный вывод включен.
test=# \df *prewa*
Список функций
-[ RECORD 1 ]-----+-----
Схема          | public
Имя            | autoprewarm_dump_now
Тип данных результата | bigint
Типы данных аргументов |
Тип            | func
-[ RECORD 2 ]-----+-----
Схема          | public
Имя            | autoprewarm_start_worker
Тип данных результата | void
Типы данных аргументов |
Тип            | func
-[ RECORD 3 ]-----+-----
Схема          | public
Имя            | pg_prewarm
Тип данных результата | bigint
Типы данных аргументов | regclass, mode text DEFAULT 'buffer'::text,
                        | fork text DEFAULT 'main'::text,
                        | first_block bigint DEFAULT NULL::bigint,
                        | last_block bigint DEFAULT NULL::bigint
Тип            | func
```

Самый простой и распространенный способ работы с `pg_prewarm` – попросить его кешировать объект целиком:

```
test=# SELECT pg_prewarm('t_test');
pg_prewarm
-----
         443
(1 строка)
```

Заметим, что если таблица очень велика и не помещается в кеш целиком, то будут кешированы только ее части, что в большинстве случаев хорошо.

Функция возвращает количество обработанных 8-килобайтных блоков.

Если мы не хотим кешировать все блоки объекта, то можем задать диапазон блоков. В примере ниже кешируются блоки с номерами от 10 до 30 из слоя `main`:

```
test=# SELECT pg_prewarm('t_test', 'buffer', 'main', 10, 30);
pg_prewarm
-----
```


21

(1 строка)

Видно, что кеширован 21 блок.

Анализ производительности с помощью pg_stat_statements

Расширение pg_stat_statements – важнейшее из всех находящихся в каталоге contrib. Его обязательно следует активировать, поскольку оно дает ценнейшие данные о производительности. Без pg_stat_statements очень трудно понять причины проблем в этой области.

В силу его важности расширение pg_stat_statements было рассмотрено в этой книге выше.

Анализ хранилища с помощью pgstattuple

Бывает, что таблицы в PostgreSQL начинают неудержимо расти. Технически это называется **разбуханием таблицы** (table bloat). Спрашивается, какие таблицы разбухли и в какой мере их рост объясняется разбуханием? На эти вопросы отвечает расширение pgstattuple.

```
test=# CREATE EXTENSION pgstattuple;
CREATE EXTENSION
```

Функция, предоставляемая этим расширением, возвращает строку составного типа. Поэтому, чтобы получить понятный результат, ее надо употреблять во фразе FROM:

```
test=# \x
Расширенный вывод включен.
test=# SELECT * FROM pgstattuple('t_test');
-[ RECORD 1 ]
```

```
-----+-----
table_len      | 3629056
tuple_count    | 100000
tuple_len      | 2800000
tuple_percent  | 77.16
dead_tuple_count | 0
dead_tuple_len | 0
dead_tuple_percent | 0
free_space     | 16652
free_percent   | 0.46
```

Таблица, использованная в этом примере, находится в приличном состоянии; ее размер равен 3.6 МБ, и мертвых строк в ней нет. Свободное пространство тоже не слишком велико. Если доступ к таблице замедляется из-за разбухания, значит, число мертвых строк и объем свободного пространства вышли из-под контроля. В небольших дозах то и другое безвредно, но если таблица состоит преимущественно из мертвых строк и свободного пространства, то нужны решительные меры.

Расширение `pgstattuple` предлагает также функцию для анализа индексов:

```
test=# CREATE INDEX idx_id ON t_test (id);
CREATE INDEX
```

Функция `pgstattindex` возвращает подробную информацию об индексе:

```
test=# SELECT * FROM pgstattindex('idx_id');
-[ RECORD 1 ]
```

```
-----+-----
version          | 2
tree_level       | 1
index_size       | 2260992
root_block_no    | 3
internal_pages   | 1
leaf_pages       | 274
empty_pages      | 0
deleted_pages    | 0
avg_leaf_density | 89.83
leaf_fragmentation | 0
```

Наш индекс довольно плотный (89.83%). Это хороший признак. По умолчанию коэффициент заполнения индекса (`FILLFACTOR`) равен 90%, так что значение, близкое к 90%, говорит, что индекс в отличном состоянии.

Иногда мы хотим проанализировать не одну таблицу, а все таблицы в базе данных или в какой-то схеме. Как это сделать? Обычно список подлежащих обработке объектов указывается во фразе `FROM`. Однако в нашем примере во фразе `FROM` уже находится функция, и как нам заставить PostgreSQL просмотреть список таблиц? Ответ дает латеральное соединение.

Имейте в виду, что `pgstattuple` должна прочитать объект целиком. Если база данных велика, то на это может уйти много времени. Поэтому рекомендую сохранять результаты запроса, чтобы их можно было всесторонне проанализировать, не выполняя запрос снова и снова.

Нечеткий поиск с помощью `pg_trgm`

Расширение `pg_trgm` позволяет выполнять нечеткий поиск. Мы уже рассматривали его в главе 3.

Подключение к удаленному серверу с помощью `postgres_fdw`

Не всегда данные находятся в одном месте. Гораздо чаще они распределены по всей инфраструктуре, и бывает так, что данные из разных мест необходимо объединить. Решение дает обертка внешних (сторонних) данных, реализованная в соответствии со стандартом SQL/MED.

В этом разделе мы обсудим расширение `postgres_fdw`, которое позволяет выбирать данные из источника данных PostgreSQL. Первым делом нужно создать обертку внешних данных:

```
test=# \h CREATE FOREIGN DATA WRAPPER
Команда: CREATE FOREIGN DATA WRAPPER
```

Описание: создать обёртку сторонних данных

Синтаксис:

```
CREATE FOREIGN DATA WRAPPER имя
  [ HANDLER функция_обработчик | NO HANDLER ]
  [ VALIDATOR функция_проверки | NO VALIDATOR ]
  OPTIONS ( параметр 'значение' [, ... ] ) ]
```

Команда CREATE FOREIGN DATA WRAPPER становится доступна после установки расширения стандартным образом:

```
test=# CREATE EXTENSION postgres_fdw;
CREATE EXTENSION
```

Следующий шаг – определение виртуального сервера, который будет указывать на реальный сервер, откуда PostgreSQL должна брать данные. В конечном итоге PostgreSQL должна будет сконструировать полную строку соединения, и сведения о местонахождении сервера – главное, о чем она должна знать. Информацию о пользователе мы добавим позже, а пока зададим адрес, номер порта и т. п. Для этого служит команда CREATE SERVER с таким синтаксисом:

```
test=# \h CREATE SERVER
Команда: CREATE SERVER
Описание: создать сторонний сервер
Синтаксис:
CREATE SERVER [ IF NOT EXISTS ] имя_сервера name [ TYPE тип_сервера' ]
  [ VERSION 'версия_сервера' ]
  FOREIGN DATA WRAPPER имя_обёртки_сторонних_данных
  [ OPTIONS ( параметр 'значение' [, ... ] ) ]
```

Для демонстрации создадим вторую базу данных на той же машине и определим внешний сервер:

```
[hs@zenbook~]$ createdb customer
[hs@zenbook~]$ psql customer
customer=# CREATE TABLE t_customer (id int, name text);
CREATE TABLE
```

```
customer=# CREATE TABLE t_company (
  country text,
  name text,
  active text
);
CREATE TABLE
```

```
customer=# \d
          Список отношений
  Схема |      Имя      | Тип  | Владелец
-----+-----+-----+-----
public | t_company     | table |
hs public | t_customer   | table | hs
(2 строки)
```

Теперь следует добавить сервер в базу данных test:

```
test=# CREATE SERVER customer_server
        FOREIGN DATA WRAPPER postgres_fdw
        OPTIONS (host 'localhost', dbname 'customer', port '5432');
CREATE SERVER
```

Вся важная информация задается во фразе `OPTIONS`. Это увеличивает гибкость, потому что различных оберток внешних данных много, и у каждой из них свои параметры.

Определив сервер, перейдем к сопоставлению пользователей. Мы работаем с двумя серверами, и вовсе необязательно, что состав пользователей на обоих одинаков. Поэтому необходимо сопоставить пользователю на одном сервере пользователя на другом:

```
test=# \h CREATE USER MAPPING
Команда: CREATE USER MAPPING
Описание: создать сопоставление пользователя для стороннего сервера
Синтаксис:
CREATE USER MAPPING FOR { имя_пользователя | USER | CURRENT_USER | PUBLIC }
        SERVER имя_сервера
        [ OPTIONS ( параметр 'значение' [ , ... ] ) ]
```

Синтаксис простой, поэтому написать нужную команду легко:

```
test=# CREATE USER MAPPING
        FOR CURRENT_USER SERVER customer_server
        OPTIONS (user 'hs', password 'abc');
CREATE USER MAPPING
```

И снова вся важная информация находится во фразе `OPTIONS`, поскольку в разных обертках состав параметров может быть различен. В данном случае задаются просто данные локального пользователя.

Подготовив необходимую инфраструктуру, мы можем приступить к созданию внешних таблиц. Синтаксически создание внешней таблицы мало чем отличается от создания обычной локальной таблицы. Мы должны перечислить все столбцы, указав их типы:

```
test=# CREATE FOREIGN TABLE f_customer (id int, name text)
        SERVER customer_server
        OPTIONS (schema_name 'public', table_name 't_customer');
CREATE FOREIGN TABLE
```

Столбцы описываются так же, как в команде `CREATE TABLE`. Но внешняя таблица указывает на таблицу на удаленном сервере. Имя схемы и имя удаленной таблицы задаются во фразе `OPTIONS`.

Созданную таблицу можно использовать:

```
test=# SELECT * FROM f_customer
 id | name
-----+-----
(0 строк)
```

Чтобы понять, что при этом делает PostgreSQL, выполним команду EXPLAIN с параметром analyze:

```
test=# EXPLAIN (analyze true, verbose true)
SELECT * FROM f_customer
               QUERY PLAN
-----
Foreign Scan on public.f_customer
  (cost=100.00..150.95 rows=1365 width=36)
  (actual time=0.221..0.221 rows=0 loops=1)
    Output: id, name
    Remote SQL: SELECT id, name FROM public.t_customer
Planning time: 0.067 ms
Execution time: 0.451 ms
(5 строк)
```

Самая важная часть здесь – Remote SQL. Обертка внешних данных отправляет запрос другой стороне и выбирает минимально возможное число строк, потому что все ограничения, какие возможно, отрабатываются на удаленном сервере. Это относится к условиям фильтрации, соединениям и даже агрегатным функциям (в версии PostgreSQL 10.0).

Конечно, команда CREATE FOREIGN TABLE – вещь замечательная, но перечислять столбцы каждой таблицы снова и снова утомительно. Можно вместо этого воспользоваться командой IMPORT, которая позволяет легко и быстро импортировать в локальную базу всю схему целиком, создав внешние таблицы:

```
test=# \h IMPORT
Команда: IMPORT FOREIGN SCHEMA
Описание: импортировать определения таблиц со стороннего сервера
Синтаксис:
IMPORT FOREIGN SCHEMA удалённая_схема
  [ { LIMIT TO | EXCEPT } ( имя_таблицы [, ...] ) ]
  FROM SERVER имя_сервера
  INTO локальная_схема
  [ OPTIONS ( параметр 'значение' [, ...] ) ]
```

IMPORT дает возможность связывать целые наборы таблиц. При этом уменьшается риск опечаток и разного рода ошибок, потому что вся информация берется непосредственно из удаленного источника данных. Вот как это работает:

```
test=# IMPORT FOREIGN SCHEMA public
      FROM SERVER customer_server INTO public;
IMPORT FOREIGN SCHEMA
```

Мы связали все таблицы, которые ранее были созданы в схеме public. Легко видеть, что все удаленные таблицы теперь доступны:

```
test=# \det
      Список удаленных таблиц
  Схема | Таблица | Сервер
-----+-----+-----
public | f_customer | customer_server
```

```
public | t_company | customer_server
public | t_customer | customer_server
(3 строки)
```

Обработка ошибок и опечаток

Создавать внешние таблицы нетрудно, но иногда случаются ошибки, а то и просто изменяются пароли пользователей. Для решения таких проблем PostgreSQL располагает двумя командами: ALTER SERVER и ALTER USER MAPPING.

ALTER SERVER позволяет изменить определение сервера:

```
test=# \h ALTER SERVER
Команда: ALTER SERVER
Описание: изменить определение стороннего сервера
Синтаксис:
ALTER SERVER имя [ VERSION 'новая_версия' ]
      [ OPTIONS ( [ ADD | SET | DROP ] параметр ['значение'] [, ... ] ) ]
ALTER SERVER имя OWNER TO { новый_владелец | CURRENT_USER | SESSION_USER }
ALTER SERVER имя RENAME TO новое_имя
```

Эту команду можно использовать для добавления и удаления параметров сервера – очень удобно, если мы что-то забыли.

Для изменения информации о пользователе служит команда ALTER USER MAPPING:

```
test=# \h ALTER USER MAPPING
Команда: ALTER USER MAPPING
Описание: изменить сопоставление пользователей
Синтаксис:
ALTER USER MAPPING FOR { имя_пользователя | USER | CURRENT_USER | SESSION_USER | PUBLIC }
      SERVER имя_сервера
      OPTIONS ( [ ADD | SET | DROP ] параметр ['значение'] [, ... ] )
```

Интерфейс SQL/MED регулярно совершенствуется, и на момент написания этой книги велась работа над добавлением новых возможностей. В будущем в ядро будут внесены дополнительные оптимизации, что сделает SQL/MED хорошим инструментом повышения масштабируемости.

ДРУГИЕ ПОЛЕЗНЫЕ РАСШИРЕНИЯ

Все описанные выше расширения входят в состав пакета contrib, поставляемого вместе с исходным кодом PostgreSQL. Но это не единственные расширения, доступные сообществу PostgreSQL. Есть много других пакетов, реализующих самую разную функциональность.

К сожалению, эта глава слишком коротка, чтобы рассказать обо всем имеющемся богатстве. Количество модулей растет с каждым днем, и рассмотреть их все невозможно. Поэтому остановлюсь только на тех, которые считаю самыми важными.

PostGIS (<http://postgis.net/>) – интерфейс между базой данных и геоинформа-

ционной системой (ГИС). Он принят повсеместно и является стандартом де-факто в мире реляционных СУБД с открытым исходным кодом. Это в высшей степени профессиональный и мощный продукт.

Если вас интересует геопространственная маршрутизация, то обратите внимание на модуль `pgRouting`. Он содержит различные алгоритмы нахождения наилучшего пути между точками и работает на основе PostgreSQL.

В этой главе мы рассказали о расширении `postgres_fdw`, которое позволяет подключаться к другой базе данных PostgreSQL. Но оберток внешних данных куда больше. Одна из самых известных среди профессионалов – расширение `oracle_fdw`. Оно позволяет интегрироваться с Oracle и получать данные из удаленной базы так же, как `postgres_fdw`.

Иногда нас интересует тестирование стабильности инфраструктуры. Для этого предназначено расширение `pg_crash` (https://github.com/cybertec-postgresql/pg_crash). Этот модуль, который постоянно приводит к краху базы данных, – прекрасное решение для тестирования и отладки пулов подключений, обладающее возможностью заново подключать приложение к «упавшей» базе. `pg_crash` периодически «взрывает бомбу» – разрывает сеансы работы с базой данных или затирает память. Идеально для длительного тестирования.

РЕЗЮМЕ

В этой главе мы узнали о наиболее многообещающих модулях, поставляемых в составе стандартного дистрибутива PostgreSQL. Они весьма разнообразны по функциональности: от подключения к базе данных до нечувствительного к регистру текста и анализа внутреннего состояния сервера.

Познакомившись с расширениями, мы в следующей главе поговорим о том, как искать и устранять неполадки в PostgreSQL.

Глава 12

Поиск и устранение неполадок

В главе 11 мы узнали о некоторых полезных и широко распространенных расширениях, которые значительно расширяют возможности вашей системы. Теперь мы расскажем о систематическом подходе к поиску и устранению неполадок в PostgreSQL.

В этой главе рассматриваются следующие вопросы:

- первоначальное изучение незнакомой базы данных;
- сбор сведений;
- выявление узких мест;
- устранение повреждений хранилища;
- анализ поврежденных реплик.

Имейте в виду, что в базе данных возможны разнообразные неполадки, поэтому очень важен профессиональный мониторинг.

ПЕРВОНАЧАЛЬНОЕ ИЗУЧЕНИЕ НЕЗНАКОМОЙ БАЗЫ ДАННЫХ

Приступая к администрированию крупномасштабной системы, вы не всегда знаете, что именно делает система. Когда управлять приходится сотнями систем, невозможно знать, что происходит на каждой из них.

Суть поиска неполадок можно выразить одним словом: **данные**. Если данных недостаточно, то ничего не исправишь. Поэтому первым делом нужно настроить средство мониторинга, например pgwatch2 (<https://www.cybertec-postgresql.com/en/products/pgwatch2/>), которое позволяет заглянуть внутрь сервера баз данных.

Если из отчетов видно, что ситуация заслуживает пристального внимания, значит, инструмент оказался полезным средством организации работы.

АНАЛИЗ РЕЗУЛЬТАТОВ PG_STAT_ACTIVITY

Первым делом рекомендуется проверить представление pg_stat_activity и получить ответы на следующие вопросы:

- сколько запросов одновременно выполняется в системе?
- верно ли, что в столбце query все время отображаются похожие запросы?
- имеются ли запросы, которые выполняются долго?
- ожидает ли какой-нибудь запрос освобождения блокировки?
- нет ли подключений с подозрительных адресов?

Представление `pg_stat_activity` дает общую картину происходящего в системе, поэтому его и нужно проверять первым. Конечно, графический инструмент мониторинга показал бы состояние дел лучше, но в конечном итоге все сводится к выполняемым запросам. Поэтому хороший обзор системы, который дает `pg_stat_activity`, исключительно важен для выявления проблем.

Чтобы упростить вам жизнь, я подготовил несколько запросов, которые считаю полезными для скорейшего нахождения потенциальных проблем.

Опрос `pg_stat_activity`

Следующий запрос показывает, сколько запросов к базе данных выполняется в данный момент:

```
test=# SELECT datname,
        count(*) AS open,
        count(*) FILTER (WHERE state = 'active') AS active,
        count(*) FILTER (WHERE state = 'idle') AS idle,
        count(*) FILTER (WHERE state = 'idle in transaction') AS idle_in_trans
        FROM pg_stat_activity
        WHERE backend_type = 'client backend'
        GROUP BY ROLLUP(1);
```

datname	open	active	idle	idle_in_trans
test	2	1	0	1

(2 строки)

Чтобы показать как можно больше информации на одном экране, использованы частичные агрегаты. Мы видим активные, неактивные и неактивные, но находящиеся внутри транзакции запросы. Если количество неактивных запросов внутри транзакции велико, то нужно разобраться, как долго эти транзакции открыты:

```
test=# SELECT pid, xact_start, now() - xact_start AS duration
        FROM pg_stat_activity
        WHERE state LIKE '%transaction%'
        ORDER BY 3 DESC;
```

pid	xact_start	duration
19758	2017-11-26 20:27:08.168554+01	22:12:10.194363

(1 строка)

Показанная выше транзакция открыта более 22 часов. Вопрос: как случилось, что транзакция открыта в течение столь длительного времени? Как правило, такие долгие транзакции должны вызывать подозрение и несут серь-

езную опасность. В чем же их опасность? Выше в этой книге мы видели, что команда VACUUM может очистить только те мертвые строки, которые уже не видны ни одной транзакции. Но если транзакция остается открытой много часов или даже дней, то VACUUM не может сделать ничего полезного, что ведет к разбуханию таблицы.

Поэтому настоятельно рекомендуется наблюдать за длительными транзакциями и в случае необходимости отменять их. Начиная с версии 9.6 в PostgreSQL появилась функция **«слишком старый снимок»**, которая позволяет прекращать длительные транзакции, если снимок существует слишком долго.

Кроме того, полезно проверять, нет ли в системе долго выполняющихся запросов:

```
test=# SELECT now() - query_start AS duration, datname, query
        FROM pg_stat_activity
        WHERE state = 'active'
        ORDER BY 1 DESC;
 duration | datname | query
-----+-----+-----
00:00:38.814526 | dev | SELECT pg_sleep(10000);
00:00:00 | test | SELECT now() - query_start AS duration,
        datname, query
        FROM pg_stat_activity
        WHERE state = 'active'
        ORDER BY 1 DESC;
```

(2 строки)

В данном случае мы выбрали все активные запросы и вычислили, сколько времени каждый из них пребывает в этом состоянии. Часто наверху оказываются похожие запросы, это дает ценную информацию о том, что происходит в системе.

Команды, генерируемые Hibernate

Многие системы объектно-реляционного отображения (ORM), в т. ч. Hibernate, генерируют безумно длинные команды SQL. Беда в том, что pg_stat_activity показывает только первые 1024 байта запроса, а остальные отсекает. В случае длинных запросов, сгенерированных Hibernate, самые интересные части (в частности, фраза FROM) как раз и отсекаются. Для решения проблемы нужно изменить конфигурационный параметр в файле postgresql.conf:

```
test=# SHOW track_activity_query_size;
 track_activity_query_size
-----
1024
```

(1 строка)

Если увеличить значение (скажем, до 32 768) и перезапустить PostgreSQL, то мы будем видеть гораздо более длинные запросы, и диагностировать проблемы станет проще.

Как выяснить, откуда пришел запрос

В представлении `pg_stat_activity` есть несколько полей, позволяющих узнать, откуда пришел запрос:

<code>client_addr</code>	<code>inet</code>	
<code>client_hostname</code>	<code>text</code>	
<code>client_port</code>	<code>integer</code>	

Эти поля содержат IP-адреса и доменные имена (если их можно определить). Но что, если все приложения отправляют запросы с одного и того же IP-адреса, поскольку работают на одном сервере приложений? Тогда будет очень трудно понять, от какого приложения поступил конкретный запрос.

Решить проблему можно, попросив разработчиков установить параметр `application_name`:

```
test=# SHOW application_name
application_name
-----
psql
(1 строка)

test=# SET application_name TO 'some_name';
SET

test=# SHOW application_name
application_name
-----
some_name
(1 строка)
```

Если они пойдут навстречу, то значение `application_name` появится в системном представлении, так что узнать источник запроса станет гораздо проще. Параметр `application_name` можно также задать в строке подключения.

Выявление медленных запросов

После изучения `pg_stat_activity` стоит поинтересоваться медленными, занимающими много времени запросами. Есть два основных подхода к этой проблеме:

- искать отдельные медленные запросы в журнале;
- искать типы медленных запросов.

Нахождение отдельных медленных запросов – классический подход к оптимизации производительности. Если задать в параметре `log_min_duration_statement` пороговое значение, то PostgreSQL будет помещать в журнал информацию о каждом запросе, который выполнялся дольше. По умолчанию протоколирование медленных запросов выключено:

```
test=# SHOW log_min_duration_statement;
log_min_duration_statement
-----
-1
(1 строка)
```

Но выставить разумное значение очень даже имеет смысл. Конечно, это значение зависит от характера рабочей нагрузки в вашей системе.

Часто порог меняется от базы к базе. Поэтому можно установить этот параметр для каждой базы отдельно:

```
test=# ALTER DATABASE test SET log_min_duration_statement TO 10000;
ALTER DATABASE
```

При использовании журнала медленных запросов следует помнить об одном важном факторе – много быстрых запросов создает более высокую нагрузку, чем горстка медленных. Разумеется, знать об отдельных медленных запросах полезно, но иногда не они являются проблемой. Рассмотрим пример: в системе выполняется миллион запросов по 500 мс каждый, а также несколько аналитических запросов, занимающих по несколько минут. Очевидно, что настоящая проблема так и не отразится в журнале медленных запросов, а все операции экспорта данных, создания индексов и массовой загрузки (избежать которых все равно невозможно) будут протоколироваться и указывать в неверном направлении.

Поэтому лично я рекомендую использовать журнал медленных запросов, но с осторожностью. Важно понимать, что именно мы измеряем.

На мой взгляд, гораздо полезнее активно использовать представление `pg_stat_statements`. Оно содержит агрегированную информацию, а не сведения об отдельных запросах. Это представление уже обсуждалось выше, но переоценить его значимость невозможно.

Анализ отдельных запросов

Иногда медленные запросы удастся идентифицировать, но мы все равно не понимаем, что происходит. Следующий естественный шаг – проанализировать план выполнения запроса. Выявить операции, из-за которых растет время выполнения, довольно просто. Предлагаю следующую последовательность действий:

- посмотрите, в каком месте плана тратится больше всего времени;
- проверьте, все ли необходимые индексы построены (отсутствие индексов – одна из основных причин низкой производительности);
- выполните команду `EXPLAIN (buffers true, analyze true и т. д.)`, чтобы узнать, не потребляет ли запрос слишком много буферов;
- включите параметр `track_io_timing`, чтобы понять, связана проблема с вводом-выводом или с процессором (особенно внимательно смотрите, нет ли ввода-вывода с произвольной выборкой);
- посмотрите, нет ли неправильных оценок, и, если есть, постарайтесь исправить ситуацию;
- поищите слишком часто выполняемые хранимые процедуры;
- изучите, нельзя ли пометить некоторые из них как `STABLE` или `IMMUTABLE`.

Заметим, что в представлении `pg_stat_statements` не учитывается время синтаксического разбора запросов, поэтому если запросы очень длинные, то показанный результат может быть не совсем точным.

Углубленный анализ с помощью perf

В большинстве случаев описанные действия помогут найти проблемы быстро и эффективно. Но иногда информации, полученной от базы данных, может быть недостаточно.

Программа perf – это инструмент анализа для Linux, который позволяет узнать, какие написанные на C функции вызывают системные проблемы. Обычно perf не устанавливается по умолчанию, но я рекомендую установить ее. Для использования perf зайдите на сервер от имени root и выполните команду:

```
perf top
```

Экран обновляется раз в несколько секунд, так что вы будете видеть, что происходит, в динамике. В распечатке ниже показана типичная картина для базы, в которой производится только чтение:

```
Samples: 164K of event 'cycles:ppp', Event count (approx.): 109789128766
Overhead Shared Object Symbol
3.10% postgres[.] AllocSetAlloc
1.99% postgres[.] SearchCatCache
1.51% postgres[.] base_yyparse
1.42% postgres[.] hash_search_with_hash_value
1.27% libc-2.22.so[.] vfprintf
1.13% libc-2.22.so[.] _int_malloc
0.87% postgres[.] palloc
0.74% postgres[.] MemoryContextAllocZeroAligned
0.66% libc-2.22.so[.] __strcmp_sse2_unaligned
0.66% [kernel][k] _raw_spin_lock_irqsave
0.66% postgres[.] _bt_compare
0.63% [kernel][k] __fget_light
0.62% libc-2.22.so[.] strlen
```

В данном случае ни одна функция не потребляет слишком много времени, т. е. система работает нормально.

Но так бывает не всегда. Существует весьма распространенная проблема **состязания за спинлок** (циклическую блокировку). В чем ее суть? Спинлоки (<https://ru.wikipedia.org/wiki/Spinlock>) используются ядром PostgreSQL для синхронизации, например, доступа к буферу. Эта возможность предоставляется современными процессорами, чтобы избежать обращения к операционной системе ради мелких операций (например, увеличения счетчика на единицу). Вообще говоря, это очень удобно, но в некоторых особых случаях спинлоки начинают сходить с ума. О наличии состязания за спинлок свидетельствуют следующие симптомы:

- очень высокая занятость процессора;
- невыносимо низкая производительность (запросы, выполнение которых обычно занимает считанные миллисекунды, теперь работают несколько секунд);
- ввода-вывода почти нет, потому что процессор занят обслуживанием спинлоков.

Часто состязание за спинлок возникает внезапно. Система работала нормально, и вдруг ни с того ни с сего загрузка подскакивает, а пропускная способность падает, как камень на дно. Команда `perf top` показывает, что система проводит почти все время в функции `s_lock`. Если вы наблюдаете такую картину, попробуйте изменить следующий параметр:

`huge_pages = try # on, off или try`

Установите его значение равным `off`, а не `try`, или вообще выключите очень большие страницы на уровне операционной системы. Есть подозрение, что в одних версиях ядра такие проблемы возникают чаще, чем в других. Особенно плохо в этом отношении, похоже, ведут себя ядра серии Red Hat 2.6.32 (обратите внимание на слово *похоже*).

Программа `perf` полезна также при использовании PostGIS. Если больше всего времени потребляют функции, относящиеся к ГИС (из какой-то поддерживающей библиотеки), то проблема, вероятно, связана не с неправильной настройкой PostgreSQL, а просто с тем, что выполняются действительно дорогостоящие операции.

Анализ журнала

Если система ведет себя подозрительно, то имеет смысл заглянуть в журнал и посмотреть, что происходит. Отметим, что записи в журнале различаются по важности. В PostgreSQL серьезность сообщения варьируется от `DEBUG` (отладочное) до `PANIC` (авария).

Для администратора особый интерес представляют сообщения трех уровней:

- ERROR
- FATAL
- PANIC

Сообщения типа `ERROR` относятся к синтаксическим ошибкам, проблемам с правами доступа и т. д. Такие сообщения присутствуют всегда. Важно лишь, насколько часто возникают ошибки определенного вида. Очевидно, что миллионы синтаксических ошибок – ситуация, которой не должно быть на производственном сервере базы данных.

Сообщения типа `FATAL` серьезнее, чем `ERROR`. Они возникают, например, когда системе не удалось выделить разделяемую память или обнаружено неожиданное состояние приемника журнала транзакций. Иными словами, такие сообщения свидетельствуют о наличии ошибки, которую нельзя игнорировать.

И наконец, сообщения типа `PANIC`. Если вам встретилось такое сообщение, значит, все действительно плохо. Классические примеры `PANIC` – повреждение таблицы блокировок или слишком большое количество созданных семафоров. Это приводит к остановке сервера.

Анализ наличия индексов

Выполнив первые три шага, мы должны взглянуть на производительность в общем плане. На протяжении всей книги я не устаю повторять, что отсутствие индексов может стать причиной никуда не годной производительности. Поэтому всякий раз, как мы сталкиваемся с медленной системой, следует проверить, все ли необходимые индексы построены, и, если нет, достроить недостающие.

Обычно заказчики просят нас оптимизировать уровень RAID, настроить ядро или заняться еще каким-то нетривиальным делом. Но на практике все часто сводится к горстке отсутствующих индексов. На мой взгляд, всегда стоит потратить какое-то время, чтобы проверить, все ли нужные индексы на месте. Это нетрудно и не отнимает много времени, поэтому должно быть сделано вне зависимости от характера проблем с производительностью.

Вот мой любимый запрос, который может подсказать, что какой-то индекс отсутствует:

```
SELECT schemaname, relname, seq_scan, seq_tup_read,
       idx_scan, seq_tup_read / seq_scan AS avg
FROM pg_stat_user_tables
WHERE seq_scan > 0
ORDER BY seq_tup_read DESC
LIMIT 20;
```

Поищите большие таблицы, которые часто просматриваются последовательно (большая величина `avg`). Они обычно оказываются в верхних строчках результата.

Анализ памяти и ввода-вывода

Разобравшись с отсутствующими индексами, мы можем проанализировать потребление памяти и характер ввода-вывода. Для этого рекомендуется включить параметр `track_io_timing`. Если он равен `on`, то PostgreSQL будет собирать и показывать информацию об ожидании диска.

Нередко заказчики спрашивают, ускорится ли работа базы, если добавить диски. Можно гадать о том, что случится, но, вообще говоря, лучше измерить. Включение режима `track_io_timing` поможет собрать данные и понять, что происходит в действительности.

PostgreSQL представляет сведения об ожидании диска разными способами. Один из них – представление `pg_stat_database`:

```
test=# \d pg_stat_database
```

Представление "pg_catalog.pg_stat_database"

Столбец	Тип	Модификаторы
<code>datid</code>	<code>oid</code>	
<code>datname</code>	<code>name</code>	

```

...
conflicts      | bigint          |
temp_files     | bigint          |
temp_bytes     | bigint          |
...
blk_read_time  | double precision|
blk_write_time | double precision|

```

Обратите внимание на два последних поля: `blk_read_time` и `blk_write_time`. Они говорят, сколько времени PostgreSQL потратила на ожидание ответа от операционной системы. Подчеркнем, что измеряется не само время ожидания диска, а время, понадобившееся операционной системе, чтобы вернуть данные.

Если операционная система нашла данные в кеше, то это время будет сравнительно мало. Если же ей пришлось выполнять реальный ввод-вывод с произвольной выборкой, то получение одного-единственного блока может занять несколько миллисекунд.

Часто значения `blk_read_time` и `blk_write_time` велики, когда велики значения `temp_files` и `temp_bytes`. Во многих случаях это свидетельствует о неправильной установке параметров `work_mem` или `maintenance_work_mem`. Запомните: если PostgreSQL не может сделать что-то в памяти, она вынуждена обращаться к диску. Признаком этого является рост значения `temp_files`. Если временные файлы (`temp_files`) существуют, значит, есть все основания для ожидания диска.

Глобальное представление на уровне базы данных, конечно, интересно, но оно не дает полной информации о настоящем источнике проблемы. Зачастую виновниками является всего несколько запросов. Найти их можно с помощью представления `pg_stat_statements`:

```

test=# \d pg_stat_statements
        Представление "public.pg_stat_statements"
        Столбец          | Тип          | Модификаторы
-----+-----+-----
...
query                   | text         |
calls                   | bigint       |
total_time              | double precision|
...
temp_blks_read          | bigint       |
temp_blks_written       | bigint       |
blk_read_time           | double precision|
blk_write_time          | double precision|

```

Оно позволяет для каждого запроса понять, было ожидание диска или нет. Особенно интересно отношение поля `blk_time` к `total_time`. В общем случае запрос, для которого ожидание диска превышает 30%, можно считать сильно ограниченным скоростью ввода-вывода.

Проверив системные таблицы PostgreSQL, можно поинтересоваться, о чем говорит команда `vmstat`, являющаяся частью Linux. Альтернативой является команда `iostat`:


```
[hs@zenbook ~]$ vmstat 2
procs -----memory----- --swap-- -----io----- -system-- -----cpu-----
 r  b   swpd   free   buff   cache    si   so    bi   bo    in   cs   us   sy   id   wa   st
 0  0  367088 199488    96 2320388    0    2   83   96   106 156 16   6  78   0   0
 0  0  367088 198140    96 2320504    0    0    0   10  595 2624  3   1  96   0   0
 0  0  367088 191448    96 2320964    0    0    0    8  920 2957  8   2  90   0   0
```

При работе с базой данных следует обращать внимание на три поля: `bi`, `bo` и `wa`. Поле `bi` содержит количество прочитанных блоков – 1000 означает 1 МБ/с. Поле `bo` – это количество записанных блоков. В некотором роде `bi` и `bo` определяют чистую пропускную способность. Сами по себе они безвредны. Проблему составляет высокое значение `wa`¹. Если `bi` и `bo` малы, а `wa` при этом велико, значит, диск может являться узким местом, что, как правило, связано с большим количеством операций ввода-вывода с произвольной выборкой. Чем больше `wa`, тем медленнее выполняются запросы, поскольку сервер должен ждать ответа от диска.

i Высокая чистая пропускная способность – это, конечно, хорошо, но она может также являться индикатором проблемы. Если в OLTP-системе наблюдается высокая пропускная способность, значит, недостаточно оперативной памяти или отсутствуют какие-то индексы, вследствие чего PostgreSQL вынуждена читать слишком много данных. Помните, что все в мире взаимосвязано и данные не следует рассматривать в отрыве от всего остального.

О КОНКРЕТНЫХ ОШИБОЧНЫХ СИТУАЦИЯХ

Представив общие рекомендации по поиску неполадок в базе данных, мы далее обсудим некоторые распространенные ошибки, встречающиеся при работе с PostgreSQL.

Повреждение clog

В PostgreSQL имеется буфер состояния фиксации (сейчас он называется `pg_xact`, а раньше назывался `pg_clog`). В нем хранится состояние всех транзакций в системе, чтобы PostgreSQL могла понять, видна некоторая строка или нет. Вообще говоря, транзакция может находиться в одном из четырех состояний:

```
#define TRANSACTION_STATUS_IN_PROGRESS    0x00
#define TRANSACTION_STATUS_COMMITTED      0x01
#define TRANSACTION_STATUS_ABORTED        0x02
#define TRANSACTION_STATUS_SUB_COMMITTED  0x03
```

Для clog имеется отдельный каталог внутри экземпляра базы данных (`pg_xact`).

В прошлом пользователи, бывало, жаловались на **повреждение clog**, оно может быть вызвано сбойными дисками или ошибками в PostgreSQL, которые с годами были исправлены. Поврежденный буфер состояния фиксации – вещь

¹ Это значение показывает, сколько процентов времени процессора было потрачено на ожидание дисковых операций. – *Прим. ред.*

крайне неприятная, потому что, несмотря на наличие всех данных, PostgreSQL не может определить, действительны еще строки или уже нет. Так что такое повреждение можно назвать полной катастрофой.

Как администратор может определить, что буфер состояния фиксаций поврежден? Обычно об этом свидетельствует сообщение вида:

ОШИБКА: не удалось получить состояние транзакции 118831

Если PostgreSQL не может получить доступ к состоянию транзакции, то проблема налицо. Вопрос в том, как ее исправить. Честно говоря, никак – мы можем только попытаться спасти столько данных, сколько удастся.

Буфер состояния фиксаций содержит по два бита на каждую транзакцию, т. е. один байт представляет четыре транзакции, а один блок – 32 768 транзакций. Вычислив, в каком блоке находится информация о конкретной транзакции, мы можем подделать буфер состояния фиксаций:

```
dd if=/dev/zero of=<местоположение каталога data>/pg_clog/0001 bs=256K count=1
```

Здесь команда `dd` используется, чтобы изменить буфер состояния фиксаций и записать в состояние транзакции требуемое значение. Но возникает ключевой вопрос: каким должно быть состояние транзакции? Да любым – оно все равно берется с потолка, потому что мы не знаем, как на самом деле завершилась транзакция.

Однако имеет смысл указывать, что транзакция зафиксирована, т. к. при этом будет потеряно меньше данных. Впрочем, какой результат считать менее разрушительным, зависит от конкретных данных и рабочей нагрузки.

Если уж приходится идти на такой шаг, то постарайтесь хотя бы подделывать `clog` не больше, чем необходимо. Помните, что вы вручную выставляете состояние транзакций, что совсем бесполезно для движка базы данных.

После подделки `clog` следует как можно быстрее создать резервную копию и пересоздать экземпляр базы данных. Системе больше нельзя доверять, поэтому мы должны поскорее извлечь из нее данные. Имейте в виду, что хранящиеся в системе данные могут быть противоречивы и неправильны, поэтому необходимо будет тщательно проверить качество того, что удалось спасти.

Что означают сообщения о контрольной точке

Контрольные точки необходимы для обеспечения целостности данных и производительности. Чем больше интервал между контрольными точками, тем обычно выше производительность. Конфигурация PostgreSQL по умолчанию довольно консервативна, поэтому контрольные точки создаются сравнительно быстро. Если в базе одновременно изменяется много данных, то PostgreSQL может сообщить, что контрольные точки создаются слишком часто. Тогда в журнале появятся записи вида:

СООБЩЕНИЕ: контрольные точки происходят слишком часто (через 2 сек.)

СООБЩЕНИЕ: контрольные точки происходят слишком часто (через 3 сек.)

При выполнении интенсивных операций записи вследствие выгрузки/восстановления или другой большой операции PostgreSQL может заметить, что значения конфигурационных параметров слишком малы. Тогда она и помещает в журнал показанные выше сообщения.

Если встретится такое сообщение, то в интересах производительности настоятельно рекомендуется увеличить интервал между контрольными точками, установив гораздо более высокое значение параметра `max_wal_size` (в прежних версиях он назывался `checkpoint_segments`). В последних версиях PostgreSQL конфигурация по умолчанию стала гораздо лучше, чем раньше, но все равно слишком частое создание контрольных точек вполне возможно.

Увидев такое сообщение, не следует впадать в панику. Слишком частое создание контрольных точек не опасно, просто оно ведет к снижению производительности. Запись происходит гораздо медленнее, чем могла бы, но данным ничто не угрожает. Если увеличить интервал между контрольными точками, то сообщения перестанут появляться, а база данных станет работать быстрее.

Что делать с поврежденными страницами данных

PostgreSQL – очень стабильная СУБД. Она надежно защищает данные, что было доказано на протяжении многих лет эксплуатации. Но PostgreSQL зависит от оборудования и корректной работы файловой системы. Если хранилище отказывает, то отказывает и PostgreSQL – и тут мало что можно сделать, кроме добавления реплик для повышения отказоустойчивости.

Время от времени файловая система или диск выходит из строя. Но обычно это не означает, что все пропало, – просто оказались поврежденными несколько блоков. Недавно мы столкнулись с этим в виртуальной среде. Некоторые виртуальные машины по умолчанию не производят сброса на диск, поэтому PostgreSQL не может гарантировать, что данные записаны на устройство. Такое поведение может стать причиной случайных ошибок, которые трудно предсказать.

Если блок перестает читаться, то может возникнуть сообщение вида:

```
"не удалось прочитать блок %u в файле "%s": %m"
```

Выполнение запроса при этом завершается с ошибкой.

По счастью, в PostgreSQL есть средство борьбы с такими ситуациями:

```
test=# SET zero_damaged_pages TO on;
SET
test=# SHOW zero_damaged_pages;
zero_damaged_pages
-----
on
(1 строка)
```

Параметр `zero_damaged_pages` определяет способ реагирования на повреждение страниц. Вместо выдачи ошибки PostgreSQL просто заполняет блок нулями.

Отметим, что это по необходимости ведет к потере данных. Но ведь данные и так уже были повреждены или потеряны, так что это не более чем способ обработки повреждения, вызванного сбоями в системе хранения.

Я рекомендую относиться к параметру `zero_damaged_pages` с осторожностью – вы должны понимать, что делаете, устанавливая этот режим.

Беззаботное управление подключениями

В PostgreSQL каждое подключение к базе данных обслуживается отдельным процессом. Все они синхронизированы и обращаются к общей разделяемой памяти (технически в большинстве случаев это проецируемая память, но сейчас это не важно). В разделяемой памяти хранятся кеш ввода-вывода, список активных подключений, блокировки и многое другое, необходимое для правильной работы системы.

При закрытии подключения все относящиеся к нему данные удаляются из разделяемой памяти, и система остается в корректном состоянии. Но что, если подключение по какой-то причине завершится аварийно?

Главный процесс `postmaster` обнаружит отсутствие какого-то дочернего процесса. После этого все остальные подключения закрываются и начинается процесс наката журналов. Зачем? Вполне может случиться, что процесс «грохнулся», когда изменял область разделяемой памяти, и оставил ее в несогласованном состоянии. Поэтому `postmaster` вмешивается, прежде чем повреждение успеет распространиться по всей системе. Вся память очищается, и каждый клиент должен будет подключиться заново.

С точки зрения пользователя, это выглядит, как будто PostgreSQL «грохнулась» и перезапустилась, хотя на самом деле все не так. Поскольку процесс не может отреагировать на собственное «падение» (нарушение защиты памяти или некоторые другие сигналы), то очистка всего абсолютно необходима для защиты данных.

То же самое произойдет, если вы снимете процесс, обслуживающий подключение к базе данных, командой `kill -9`. Процесс по определению не может перехватить сигнал 9, поэтому должен вмешаться `postmaster`.

Борьба с разбуханием таблиц

Разбухание таблиц – одна из важнейших проблем при работе с PostgreSQL. Столкнувшись с низкой производительностью, всегда имеет смысл посмотреть, нет ли объектов, занимающих намного больше места, чем ожидается.

Как узнать, что имеет место разбухание таблицы? Проверить представление `pg_stat_user_tables`:

```
test=# \d pg_stat_user_tables
Представление "pg_catalog.pg_stat_user_tables"
  Столбец      | Тип      | Модификаторы
-----+-----+-----
 relid         | oid      |
```

schemaname	name	
relname	name	
...		
n_live_tup	bigint	
n_dead_tup	bigint	

Поля `n_live_tup` и `n_dead_tup` показывают, что происходит. Можно также воспользоваться модулем `pgstattuple`, описанным в предыдущей главе.

Что делать, если таблица сильно разбухла? Основное лечение – команда `VACUUM FULL`. Беда в том, что эта команда блокирует всю таблицу. Если таблица велика, то возникает проблема, т. к. пользователи не смогут писать в нее.



Если вы работаете с версией PostgreSQL не ниже 9.6, то можете воспользоваться программой `pg_squeeze`. Она реорганизует таблицу без блокировки (https://www.cybertec-postgresql.com/en/products/pg_squeeze/). Это особенно полезно для очень больших таблиц.

РЕЗЮМЕ

В этой главе мы изучили систематический подход к обнаружению и устранению наиболее распространенных неполадок, с которыми сталкиваются пользователи PostgreSQL. Мы рассмотрели некоторые важные системные таблицы, а также факторы, определяющие удачный или неудачный исход операции.

В последней главе мы займемся вопросом о переходе на PostgreSQL. Возможно, вы работаете с Oracle или какой-то другой СУБД, но подумываете о PostgreSQL. В главе 13 мы расскажем, как осуществить этот план.

Вопросы

Почему СУБД не администрирует себя сама?

Пользователь всегда знает больше, чем база данных. Ко всему прочему, у пользователя или администратора имеется доступ к обширной внешней информации об операционной системе, оборудовании, способах использования и т. д. Движок базы данных не может понять, осмыслен запрос пользователя или нет – он просто не знает, какова его цель. Поэтому у администраторов и разработчиков всегда имеется преимущество над СУБД, и потому они необходимы (и, скорее всего, так будет всегда).

Часто ли в PostgreSQL случаются повреждения?

Нет. Моя компания обслуживает тысячи компаний. Но нам очень редко приходилось встречаться с повреждением базы данных, а если и приходилось, то почти всегда оно было вызвано аппаратными причинами.

Нуждается ли PostgreSQL в постоянном присмотре?

Обычно нет, разве что вы эксплуатируете базу в неоптимальном режиме. Вообще говоря, PostgreSQL многое делает самостоятельно, например автоматически выполняет очистку с помощью команды `VACUUM`.

Глава 13

Переход на PostgreSQL

В главе 12 мы видели, как диагностировать и устранять типичные неполадки, встречающиеся при работе с PostgreSQL. Здесь важен систематический подход, который мы и постарались описать.

Последняя глава книги посвящена переходу с других СУБД на PostgreSQL. Многим читателям досаждают высокая стоимость лицензии на использование коммерческой СУБД. Я хочу указать таким пользователям выход и объяснить, как можно перенести данные из коммерческой системы в PostgreSQL. Переход на PostgreSQL оправдан не только с финансовой точки зрения, но и если вам нужны передовые возможности и повышенная гибкость. PostgreSQL есть что предложить, и с каждым днем добавляются новые возможности. Это относится и к разнообразию средств для перехода на PostgreSQL. Инструменты становятся все лучше, а разработчики постоянно работают над совершенствованием своих изделий.

В этой главе рассматриваются следующие вопросы:

- перенос команд SQL в PostgreSQL;
- переход с Oracle на PostgreSQL;
- переход с MySQL на PostgreSQL.

ПЕРЕНОС КОМАНД SQL В POSTGRESQL

При переходе на PostgreSQL с другой СУБД следует проанализировать функциональные возможности, предлагаемые обеими СУБД. Перенести сами данные и структуру обычно не составляет труда. Но вот с переписыванием SQL-кода дело обстоит иначе. Поэтому я включил целый раздел, в котором рассматриваются различные передовые средства SQL и их доступность в современных СУБД.

Латеральные соединения

В SQL латеральное соединение можно рассматривать как своеобразный цикл. Оно позволяет параметризовать соединение и использовать все находящееся внутри фразы LATERAL несколько раз. Например:

```
test=# SELECT *  
      FROM generate_series(1, 4) AS x,
```

```

        LATERAL (SELECT array_agg(y)
                  FROM generate_series(1, x) AS y
                  ) AS z;
x | array_agg
---+-----
1 | {1}
2 | {1,2}
3 | {1,2,3}
4 | {1,2,3,4}
(4 строки)

```

Фраза LATERAL выполняется для каждого x и имитирует цикл.

Поддержка латеральных соединений

Ниже описано, какие СУБД поддерживают латеральные соединения, а какие – нет:

- **MariaDB**: не поддерживает;
- **MySQL**: не поддерживает;
- **PostgreSQL**: поддерживает начиная с версии PostgreSQL 9.3;
- **SQLite**: не поддерживает;
- **Db2 LUW**: поддерживает начиная с версии 9.1 (2005);
- **Oracle**: поддерживает начиная с версии 12c;
- **Microsoft SQL Server**: поддерживает начиная с версии 2005 года, но используется другой синтаксис.

Наборы группирования

Наборы группирования очень полезны, когда нужно выполнить несколько агрегатных функций в одном запросе. Их использование ускоряет агрегирование, поскольку не требуется обрабатывать данные несколько раз.

Приведем пример:

```

test=# SELECT x % 2, array_agg(x)
        FROM generate_series(1, 4) AS x
        GROUP BY ROLLUP (1);
?column? | array_agg
---+-----
0 | {2,4}
1 | {1,3}
    | {2,4,1,3}
(3 строки)

```

Помимо ROLLUP, PostgreSQL поддерживает фразы CUBE и GROUPING SETS.

Поддержка наборов группирования

Ниже описано, какие СУБД поддерживают наборы группирования, а какие – нет:

- **MariaDB**: поддерживает только ROLLUP начиная с версии 5.1 (неполная поддержка);

- **MySQL:** поддерживает только ROLLUP начиная с версии 5.0 (неполная поддержка);
- **PostgreSQL:** поддерживает начиная с версии PostgreSQL 9.5;
- **SQLite:** не поддерживает;
- **Db2 LUW:** поддерживает, по крайней мере, с 1999 года;
- **Oracle:** поддерживает начиная с версии 9iR1 (приблизительно с 2000 года);
- **Microsoft SQL Server:** поддерживает начиная с версии 2008 года.

Фраза WITH – общие табличные выражения

Общие табличные выражения (CTE) – изящный способ выполнить запрос внутри SQL-команды, но только один раз. PostgreSQL выполняет все фразы WITH и позволяет воспользоваться их результатами в основном запросе.

Ниже приведен простой пример:

```
test=# WITH x AS (SELECT avg(id)
                  FROM generate_series(1, 10) AS id)
      SELECT *, y - (SELECT avg FROM x) AS diff
      FROM generate_series(1, 10) AS y
      WHERE y > (SELECT avg FROM x);
 y | diff
-----+-----
 6 | 0.5000000000000000
 7 | 1.5000000000000000
 8 | 2.5000000000000000
 9 | 3.5000000000000000
10 | 4.5000000000000000
(5 строк)
```

Здесь в общем табличном выражении WITH вычисляется среднее временного ряда, сгенерированного функцией `generate_series`. Вычисленное значение `x` можно использовать как таблицу в любом месте следующего далее запроса. В данном случае `x` используется дважды.

Поддержка общих табличных выражений

Ниже описано, какие СУБД поддерживают фразу WITH, а какие – нет:

- **MariaDB:** не поддерживает;
- **MySQL:** не поддерживает;
- **PostgreSQL:** поддерживает начиная с версии PostgreSQL 8.4;
- **SQLite:** поддерживает начиная с версии 3.8.3;
- **Db2 LUW:** поддерживает начиная с версии 8 (2000 год);
- **Oracle:** поддерживает начиная с версии 9iR2;
- **Microsoft SQL Server:** поддерживает начиная с версии 2005 года.



Отметим, что в PostgreSQL общие табличные выражения поддерживают даже операции записи (команды INSERT, UPDATE и DELETE). Я не знаю никакой другой СУБД, которая умела бы это делать.

Фраза WITH RECURSIVE

Есть две разновидности фразы WITH:

- стандартные CTE;
- CTE, позволяющие выполнять рекурсивные запросы в SQL.

Простые CTE были описаны в предыдущем разделе, а сейчас мы рассмотрим рекурсивную форму.

Поддержка фразы WITH RECURSIVE

Ниже описано, какие СУБД поддерживают фразу WITH RECURSIVE, а какие – нет:

- **MariaDB**: не поддерживает;
- **MySQL**: не поддерживает;
- **PostgreSQL**: поддерживает начиная с версии PostgreSQL 8.4;
- **SQLite**: поддерживает начиная с версии 3.8.3;
- **Db2 LUW**: поддерживает начиная с версии 7 (2000 год);
- **Oracle**: поддерживает начиная с версии 11gR2 (в Oracle принято использовать фразу CONNECT BY, а не WITH RECURSIVE);
- **Microsoft SQL Server**: поддерживает начиная с версии 2005 года.

Фраза FILTER

В стандарте SQL фраза FILTER присутствует начиная с 2003 года. Но лишь немногие СУБД поддерживают этот в высшей степени полезный элемент.

Ниже приведен пример:

```
test=# SELECT count(*),
              count(*) FILTER (WHERE id < 5),
              count(*) FILTER (WHERE id > 2)
        FROM generate_series(1, 10) AS id;
 count | count | count 
-----+-----+-----
    10 |     4 |     8 
(1 строка)
```

Фраза FILTER полезна, если условие невозможно поместить в обычную фразу WHERE, например потому, что требуется вычислить несколько агрегатов.

Отметим, что того же эффекта можно достичь с помощью более длинной конструкции:

```
SELECT sum(CASE WHEN .. THEN 1 ELSE 0 END) AS whatever FROM some_table;
```

Поддержка фразы FILTER

Ниже описано, какие СУБД поддерживают фразу FILTER, а какие – нет:

- **MariaDB**: не поддерживает;
- **MySQL**: не поддерживает;
- **PostgreSQL**: поддерживает начиная с версии PostgreSQL 9.4;
- **SQLite**: не поддерживает;
- **Db2 LUW**: не поддерживает;

- **Oracle:** не поддерживает;
- **Microsoft SQL Server:** не поддерживает.

Оконные функции

Оконные функции и их применение для решения аналитических задач подробно обсуждались в этой книге. Поэтому сразу перейдем к вопросу о совместимости.

Поддержка оконных функций

Ниже описано, какие СУБД поддерживают оконные функции, а какие – нет:

- **MariaDB:** не поддерживает;
- **MySQL:** не поддерживает;
- **PostgreSQL:** поддерживает начиная с версии PostgreSQL 8.4;
- **SQLite:** не поддерживает;
- **Db2 LUW:** поддерживает начиная с версии 7;
- **Oracle:** поддерживает начиная с версии 8i;
- **Microsoft SQL Server:** поддерживает начиная с версии 2005 года.

i Аналитические средства поддерживают также некоторые другие СУБД, в т. ч. Hive, Impala, Spark и Nuodb.

Упорядоченные наборы – фраза WITHIN GROUP

Упорядоченные наборы появились в PostgreSQL недавно. Разница между упорядоченным набором и обычным агрегатом состоит в том, что для упорядоченного набора имеет значение, каким образом данные подаются агрегатной функции. Предположим, что мы хотим выявить тренд в данных – порядок данных в этом случае важен.

В простом примере ниже вычисляется медианное значение:

```
test=# SELECT id % 2,
        percentile_disc(0.5) WITHIN GROUP (ORDER BY id)
        FROM generate_series(1, 123) AS id
        GROUP BY 1;
?column? | percentile_disc
-----+-----
0 | 62
1 | 61
(2 строки)
```

Медиану можно найти, только если входные данные отсортированы.

Поддержка фразы WITHIN GROUP

Ниже описано, какие СУБД поддерживают фразу WITHIN GROUP, а какие – нет:

- **MariaDB:** не поддерживает;
- **MySQL:** не поддерживает;
- **PostgreSQL:** поддерживает начиная с версии PostgreSQL 9.4;

- **SQLite**: не поддерживает;
- **Db2 LUW**: не поддерживает;
- **Oracle**: поддерживает начиная с версии 9iR1
- **Microsoft SQL Server**: поддерживает, но запрос следует переформулировать с использованием оконной функции.

Фраза TABLESAMPLE

Извлечение выборки из таблицы долгое время было сильной стороной коммерческих СУБД. Традиционные СУБД предлагали эту возможность в течение многих лет. Но теперь их монополия разрушена. Начиная с версии PostgreSQL 9.5 у нас также есть решение проблемы выборки.

Ниже показано, как оно работает. Сначала создадим таблицу, содержащую миллион строк:

```
test=# CREATE TABLE t_test (id int);
CREATE TABLE
test=# INSERT INTO t_test
SELECT * FROM generate_series(1, 1000000);
INSERT 0 1000000
```

Затем выполним тесты:

```
test=# SELECT count(*), avg(id)
FROM t_test TABLESAMPLE BERNOULLI (1);
 count | avg
-----+-----
  9802 | 502453.220873291165
(1 строка)
```

```
test=# SELECT count(*), avg(id)
FROM t_test TABLESAMPLE BERNOULLI (1);
 count | avg
-----+-----
 10082 | 497514.321959928586
(1 строка)
```

Мы выполнили один и тот же тест дважды, и в каждом случае выбрали из таблицы 1 % строк. В обоих случаях среднее значение близко к 5 млн, так что со статистической точки зрения результат хороший.

Поддержка фразы TABLESAMPLE

Ниже описано, какие СУБД поддерживают фразу TABLESAMPLE, а какие – нет:

- **MariaDB**: не поддерживает;
- **MySQL**: не поддерживает;
- **PostgreSQL**: поддерживает начиная с версии PostgreSQL 9.5;
- **SQLite**: не поддерживает;
- **Db2 LUW**: поддерживает начиная с версии 8.2;
- **Oracle**: поддерживает начиная с версии 8;
- **Microsoft SQL Server**: поддерживает начиная с версии 2005 года.

Ограничение выборки и смещение

Ограничение выборки в SQL – это грустная история. Если коротко, то все СУБД поддерживают эту возможность, но по-разному. Хотя в стандарте SQL четко описано, как ограничивать выборку, ни один поставщик не реализовал полной поддержки стандартного синтаксиса. Правильно было бы писать так:

```
test=# SELECT * FROM t_test FETCH FIRST 3 ROWS ONLY;
 id
----
  1
  2
  3
(3 строки )
```

Если вы никогда не встречали такой синтаксис, не расстраивайтесь – вы не одиноки.

Поддержка фразы *FETCH FIRST*

Ниже описано, какие СУБД поддерживают фразу *FETCH FIRST*, а какие – нет:

- **MariaDB**: поддерживает начиная с версии 5.1 (обычно используется `limit/offset`);
- **MySQL**: поддерживает начиная с версии 3.19.3 (обычно используется `limit/offset`);
- **PostgreSQL**: поддерживает начиная с версии 8.4 (обычно используется `limit/offset`);
- **SQLite**: поддерживает начиная с версии 2.1.0;
- **Db2 LUW**: поддерживает начиная с версии 7;
- **Oracle**: поддерживает начиная с версии 12c (используется подзапрос с функцией `row_num`);
- **Microsoft SQL Server**: поддерживает начиная с версии 2012 года (традиционно используется `top-N`).

Как видим, ограничить размер выборки – нетривиальная задача, и если вы будете переносить коммерческую базу данных на PostgreSQL, то, скорее всего, столкнетесь с каким-нибудь нестандартным синтаксисом.

Фраза *OFFSET*

Фраза *OFFSET* аналогична фразе *FETCH FIRST*. Она проста, но не поддерживается всеми. Дело обстоит не так плохо, как с фразой *FETCH FIRST*, но проблема тем не менее существует.

Поддержка фразы *OFFSET*

Ниже описано, какие СУБД поддерживают фразу *OFFSET*, а какие – нет:

- **MariaDB**: поддерживает начиная с версии 5.1;
- **MySQL**: поддерживает начиная с версии 4.0.6;
- **PostgreSQL**: поддерживает начиная с версии PostgreSQL 6.5;

- **SQLite**: поддерживает начиная с версии 2.1.0;
- **Db2 LUW**: поддерживает начиная с версии 11.1;
- **Oracle**: поддерживает начиная с версии 12c;
- **Microsoft SQL Server**: поддерживает начиная с версии 2012 года.

Темпоральные таблицы

В некоторых СУБД темпоральные таблицы используются для версионирования. К сожалению, в дистрибутиве PostgreSQL нет версионирования. Поэтому если вы переходите с Db2 или Oracle, то для переноса требуемой функциональности в PostgreSQL придется поработать. Обычно внести небольшие изменения в код для PostgreSQL нетрудно. Но все же потребуется ручная работа – копированием и вставкой уже не обойдешься.

Поддержка темпоральных таблиц

Ниже описано, какие СУБД поддерживают темпоральные таблицы, а какие – нет:

- **MariaDB**: не поддерживает;
- **MySQL**: не поддерживает;
- **PostgreSQL**: не поддерживает;
- **SQLite**: не поддерживает;
- **Db2 LUW**: поддерживает начиная с версии 10.1;
- **Oracle**: поддерживает начиная с версии 12cR1;
- **Microsoft SQL Server**: поддерживает начиная с версии 2016 года.

Сопоставление с образцом во временных рядах

В последнем известном мне стандарте SQL (SQL 2016) имеется средство для поиска соответствий во временных рядах. Пока что только компания Oracle реализовала эту функциональность в последней версии своей СУБД.

В настоящий момент ее примеру больше никто не последовал. Чтобы смоделировать этот последний писк технологической моды в PostgreSQL, придется использовать оконные функции и подзапросы. В Oracle сопоставление с образцом во временных рядах – довольно мощное средство; чтобы повторить его в PostgreSQL, одного какого-то типа запросов не хватит.

ПЕРЕХОД С ORACLE НА POSTGRESQL

Выше мы показали, как перенести в PostgreSQL наиболее важные из переносимых средств SQL. А теперь посмотрим, как выполнить переход конкретно с СУБД Oracle.

В настоящее время переход с Oracle на PostgreSQL стал весьма популярен из-за новой политики лицензирования и ведения бизнеса Oracle. Во всех странах люди начинают расставаться с Oracle и переходить на PostgreSQL.

Использование расширения `oracle_fdw` для переноса данных

Лично я для переноса данных с Oracle в PostgreSQL предпочитаю написанное Лауренцем Албе (Lorenz Albe) расширение `oracle_fdw` (https://github.com/laurenz/oracle_fdw). Это **обертка внешних данных** (FDW), позволяющая представить таблицу Oracle как таблицу PostgreSQL. Расширение `oracle_fdw` – одна из самых развитых оберток, оно надежно как скала, хорошо документировано, бесплатно и с открытым исходным кодом.

Для установки `oracle_fdw` нужно будет установить клиентскую библиотеку Oracle. К счастью, имеются готовые RPM-пакеты (<http://www.oracle.com/technetwork/topics/linuxx86-64soft-092277.html>). Понадобится драйвер OCI для взаимодействия с Oracle. Помимо готовых клиентских драйверов Oracle, нужен RPM-пакет, содержащий само расширение `oracle_fdw`, он предоставляется сообществом. Если вы работаете с ОС, в которой нет менеджера пакетов RPM, то сможете откомпилировать расширение самостоятельно, что, безусловно, возможно, но несколько утомительно.

Установив расширение, активируйте его:

```
test=# CREATE EXTENSION oracle_fdw;
```

Команда `CREATE EXTENSION` загружает расширение в текущую базу данных. На следующем шаге необходимо создать сервер и установить соответствие между пользователями PostgreSQL и Oracle:

```
test=# CREATE SERVER oraserver FOREIGN DATA WRAPPER oracle_fdw
      OPTIONS (dbserver '//dbserver.example.com/ORADB');
test=# CREATE USER MAPPING FOR postgres SERVER oradb
      OPTIONS (user 'orauser', password 'orapass');
```

Теперь можно приступать к получению данных. Я предпочитаю использовать команду `IMPORT FOREIGN SCHEMA`, которая импортирует определения данных. Она создает внешнюю таблицу для каждой таблицы в удаленной схеме, после чего можно легко читать данные из Oracle.

Самый простой способ воспользоваться импортом схемы – создать отдельные схемы на стороне PostgreSQL, куда будут импортироваться таблицы. В последнем разделе этой главы показано, как это делается в случае переноса данных из MySQL/MariaDB. Имейте в виду, что команда `IMPORT FOREIGN SCHEMA` – часть стандарта SQL/MED, поэтому процесс одинаков что для MySQL/MariaDB, что для Oracle. Это относится практически к любой обертке внешних данных, которая поддерживает команду `IMPORT FOREIGN SCHEMA`.

Хотя расширение `oracle_fdw` делает за нас большую часть работы, все же имеет смысл полюбопытствовать, как сопоставляются типы данных. В Oracle и PostgreSQL набор типов данных различен, а их сопоставление можно либо поручить `oracle_fdw`, либо выполнить самостоятельно. В таблице ниже показано соответствие типов данных: слева – типы Oracle, справа – типы PostgreSQL.

Типы Oracle	Типы PostgreSQL
CHAR	char, varchar, text
NCHAR	char, varchar, text
VARCHAR	char, varchar, text
VARCHAR2	char, varchar, text
NVARCHAR2	char, varchar, text
CLOB	char, varchar, text
LONG	char, varchar, text
RAW	uuid и bytea
BLOB	bytea
BFILE	bytea (только для чтения)
LONG RAW	bytea
NUMBER	numeric, float4, float8, char, varchar, text
NUMBER(n,m) с $m \leq 0$	numeric, float4, float8, int2, int4, int8, boolean, char, varchar, text
FLOAT	numeric, float4, float8, char, varchar, text
BINARY_FLOAT	numeric, float4, float8, char, varchar, text
BINARY_DOUBLE	numeric, float4, float8, char, varchar, text
DATE	date, timestamp, timestamptz, char, varchar, text
TIMESTAMP	date, timestamp, timestamptz, char, varchar, text
TIMESTAMP WITH TIME ZONE	date, timestamp, timestamptz, char, varchar, text
TIMESTAMP WITH LOCAL TIME ZONE	date, timestamp, timestamptz, char, varchar, text
INTERVAL YEAR TO MONTH	interval, char, varchar, text
INTERVAL DAY TO SECOND	interval, char, varchar, text
MDSYS.SDO_GEOMETRY	geometry

Если вы собираетесь использовать геометрические данные, то не забудьте установить расширение PostGIS.

Недостаток расширения `oracle_fdw` заключается в том, что оно не может перенести процедуры. Хранимые процедуры – вещь специфическая, для их переноса необходимо вмешательство человека.

Использование `ora2pg` для перехода с Oracle

Люди переходили с Oracle на PostgreSQL задолго до появления оберток внешних данных. Высокая стоимость лицензии давно вызывает недовольство пользователей, поэтому уже много лет переход на PostgreSQL представляется естественным решением.

Альтернатива расширению `oracle_fdw` – программа `ora2pg` – существует много лет, ее можно скачать с сайта <https://github.com/darold/ora2pg>. Этот скрипт, написанный на Perl, пережил уже много версий.

Перечень возможностей `ora2pg` впечатляет:

- перенос всей схемы базы данных, включая таблицы, представления, последовательности и индексы (ограничения уникальности, первичного ключа, внешнего ключа и проверочные);

- перенос прав пользователей и групп;
- перенос секционированных таблиц;
- способность экспортировать predefined функции, триггеры, процедуры, пакеты и тела пакетов;
- полный или частичный (с указанием фразы WHERE) перенос данных;
- полноценная поддержка BLOB-объектов Oracle с помощью типа данных PostgreSQL bytea;
- способность экспортировать представления Oracle в виде таблиц PostgreSQL;
- способность экспортировать пользовательские типы данных Oracle;
- базовое автоматическое преобразование PL/SQL-кода в PL/pgSQL-код. Отметим, что полностью автоматическое преобразование всего кода все же невозможно, но значительная его часть преобразуется автоматически;
- способность экспортировать таблицы Oracle в виде внешних (FDW) таблиц;
- способность экспортировать материализованные представления;
- формирование подробных отчетов о содержимом базы данных Oracle;
- оценка сложности процесса переноса базы данных Oracle;
- оценка стоимости переноса PL/SQL-кода;
- способность генерировать XML-файлы для интерпретатора данных Pentaho (Kettle);
- способность экспортировать данные Oracle Locator и пространственные геометрические данные в PostGIS;
- способность экспортировать ссылки на базы данных в виде обертки внешних данных Oracle FDW;
- способность экспортировать синонимы в виде представлений;
- способность экспортировать каталог в виде внешней таблицы или каталога для расширения external_file;
- способность распределять перечень задач на SQL между несколькими подключениями к PostgreSQL;
- способность вычислять разницу (diff) между базами данных Oracle и PostgreSQL для проверки.

На первый взгляд, может показаться, что работать с ora2pg трудно. Но в действительности это гораздо проще, чем кажется. Программа запускается следующим образом:

```
/usr/local/bin/ora2pg -c /some_path/new_ora2pg.conf
```

Для работы ora2pg необходим конфигурационный файл, содержащий всю информацию, необходимую для управления процессом. По умолчанию этот файл можно рассматривать как отправную точку для большинства задач переноса. В терминологии ora2pg процесс переноса называется проектом.

Конфигурация описывает весь проект. Во время работы ora2pg создает каталоги, в которые помещаются все данные, экспортированные из Oracle:


```
ora2pg --project_base /app/migration/ --init_project test_project
Creating project test_project.
/app/migration/test_project/
  schema/
    dblinks/
    directories/
    functions/
    grants/
    mviews/
    packages/
    partitions/
    procedures/
    sequences/
    synonyms/
    tables/
    tablespaces/
    triggers/
    types/
    views/
  sources/
    functions/
    mviews/
    packages/
    partitions/
    procedures/
    triggers/
    types/
    views/
  data/
  config/
  reports/
```

Generating generic configuration file

Creating script export_schema.sh to automate all exports.

Creating script import_all.sh to automate all imports.

Как видим, программа генерирует скрипты, которые затем можно выполнить. Результирующие данные без труда импортируются в PostgreSQL. Будьте готовы внести некоторые правки в процедуры. Не все поддается автоматическому переносу, так что ручное вмешательство – вещь ожидаемая.

Распространенные подводные камни

Существует ряд простейших синтаксических элементов, которые работают в Oracle, но могут не работать в PostgreSQL. В этом разделе перечислены самые важные из таких подводных камней. Разумеется, список далеко не полон, но он указывает в правильном направлении.

В Oracle можно встретить такую команду:

```
DELETE mytable;
```

В PostgreSQL эта конструкция неправильна, т. к. в команде DELETE необходимо ключевое слово FROM. Впрочем, это легко поправить.

Следующая проблема:

```
SELECT sysdate FROM dual;
```

В PostgreSQL нет ни функции `sysdate`, ни таблицы `dual`. С `dual` разобраться легко, достаточно создать представление, содержащее одну строку. В Oracle таблица `dual` устроена следующим образом:

```
SQL> desc dual
Name                               Null?    Type
-----
DUMMY                               VARCHA2(1)

SQL> select * from dual;
D
-
X
```

В PostgreSQL того же эффекта можно добиться, создав представление:

```
CREATE VIEW dual AS SELECT 'X' AS dummy;
```

Проблему с функцией `sysdate` тоже легко решить, заменив ее функцией `clock_timestamp()`.

Еще одна типичная проблема – отсутствие некоторых типов данных, например `VARCHAR2`, и некоторых специальных функций, которые поддерживает только Oracle. Для разрешения этих трудностей можно установить расширение `orafce`, которое предоставляет большую часть недостающей функциональности. Для получения дополнительных сведений зайдите на сайт <https://github.com/orafce/orafce>. Это расширение существует уже много лет, и ему можно доверять.

Недавнее исследование (проведенное NTT) показало, что расширение `orafce` позволяет выполнить в PostgreSQL 73 % SQL-кода, написанного для Oracle, без какой-либо модификации.

Одна из самых известных ловушек – синтаксис внешних соединений в Oracle. Рассмотрим пример:

```
SELECT employee_id, manager_id
FROM employees
WHERE employees.manager_id(+) = employees.employee_id;
```

PostgreSQL не поддерживает и никогда не будет поддерживать такой синтаксис. Поэтому указанную команду нужно переписать как положено. Знак `+` в этом контексте – сугубая специфика Oracle, от которой следует избавиться.

ora_migrator – быстрая миграция Oracle в PostgreSQL

В силу специфики своей работы мне не раз доводилось осуществлять переход из Oracle в PostgreSQL. Но иногда возникали проблемы с поиском инструментария, точно отвечающего моим целям. Мне так и не удалось найти идеальный инструмент, который мог бы быстро переносить простые базы данных, не пы-

таясь браться за сложные вещи (с которыми, возможно, не справился бы). Тогда я решил написать инструмент сам. Вместе с сотрудниками мы разработали программу на основе `oracle_fdw`, которая обладает инфраструктурой для определения типов данных на стороне PostgreSQL и умеет загружать данные быстро и эффективно. Ко всему прочему, `oracle_fdw` умеет не только подключаться к таблице, но и преобразовывать результат запроса в таблицу PostgreSQL. Поэтому `oracle_fdw` – прекрасная основа для разработки средства переноса.

Как работает ora_migrator?

Естественно возникает вопрос: как работает `ora_migrator`? Разберемся. Прежде всего `ora_migrator` подключается к системному каталогу Oracle и копирует данные в промежуточную схему в базе данных PostgreSQL. Это ключ к успеху, поскольку нам нет нужды разбирать весь код, написанный на Oracle SQL, и пытаться конвертировать его. Обращаясь к системным таблицам напрямую, мы можем собрать все необходимое на стороне PostgreSQL самостоятельно, что гораздо проще и надежнее. В расширении `oracle_fdw` есть все необходимые для этого механизмы, так что нам нужно только написать свой код поверх него.

Затем определения данных копируются из Oracle в промежуточную схему PostgreSQL. Идея в том, чтобы позднее можно было изменить структуру таблицы и типы данных. `oracle_fdw` отлично справляется с определением структуры данных в PostgreSQL, но не может знать наверняка, что нам нужно. Поэтому лучше выполнить этот промежуточный шаг перед окончательным переносом данных в PostgreSQL.

На нашем сайте устройство `ora_migrator` описано во всех подробностях: https://www.cybertec-postgresql.com/en/ora_migrator-moving-from-oracle-to-postgresql-even-faster/.



Это открытый код, который можно использовать бесплатно.

ПЕРЕХОД ИЗ MySQL или MARIADB НА POSTGRESQL

Итак, мы уже усвоили несколько ценных уроков о переходе из Oracle в PostgreSQL. Переход из СУБД MySQL или MariaDB на PostgreSQL производится довольно легко, по сравнению с Oracle или Informix. Но у Informix и Oracle есть одна общая черта: они соблюдают ограничения CHECK и корректно работают с типами данных. Вообще, мы можем без опаски предполагать, что данные в этих коммерческих системах корректные, не нарушают базовых правил целостности и не идут вразрез со здравым смыслом.

Следующий наш кандидат не таков. Многие из того, что вы знаете о коммерческих базах данных, к MySQL неприменимо. Фраза NOT NULL мало что значит для MySQL (если только вы явно не установите строгий режим). В Oracle, Informix, Db2 и всех остальных известных мне СУБД NOT NULL – закон, который нельзя нарушать ни при каких обстоятельствах. MySQL по умолчанию не относит-

ся к этим ограничениям сколько-нибудь серьезно (хотя будем справедливы, в последних версиях это изменилось – строгий режим теперь по умолчанию включен, но во многих старых базах данных по-прежнему действуют старые умолчания). В процессе переноса это создает серьезные проблемы. Что делать с данными, которые, с технической точки зрения, некорректны? Если в столбце с ограничением NOT NULL вдруг обнаруживается куча значений NULL, как мы должны их обрабатывать? Нет, MySQL не вставляет NULL-значения в столбцы вида NOT NULL. Она поступает иначе – вставляет пустую строку или ноль в зависимости от типа данных. Выглядит все это крайне неприятно.

Обработка данных в MySQL и MariaDB

Как вы, наверное, заметили, когда речь заходит о базах данных, я далек от беспристрастности. Но я не собираюсь по этой причине грубо охаивать MySQL и MariaDB. Наша настоящая цель – понять, почему MySQL и MariaDB вызывают такие сложности. У моей пристрастности есть основания, и я хочу объяснить, в чем дело. Все те вещи, с которыми нам предстоит столкнуться, пугают и имеют серьезные последствия для процесса переноса в целом. Я уже сказал, что MySQL – это нечто особенное, и теперь попытаюсь обосновать свою точку зрения.

Повторю, что в примерах ниже мы предполагаем, что в той версии MySQL/MariaDB, с которой мы работаем, строгий режим выключен. Так оно и было, когда я писал первый вариант этой главы (во времена PostgreSQL 9.6). Когда вышла версия PostgreSQL 10.0, строгий режим был уже по умолчанию включен, так что большая часть сказанного ниже относится к более старым версиям MySQL/MariaDB.

Для начала создадим простую таблицу:

```
MariaDB [test]> CREATE TABLE data (
    id integer NOT NULL,
    data numeric(4, 2)
);
Query OK, 0 rows affected (0.02 sec)

MariaDB [test]> INSERT INTO data VALUES (1, 1234.5678);
Query OK, 1 row affected, 1 warning (0.01 sec)
```

Пока что ничего особенного. Мы создали таблицу с двумя столбцами. Первый столбец явно описан как NOT NULL, а второй, по идее, должен содержать числовое значение, содержащее не более четырех цифр. И наконец, мы добавили ничем не примечательную строку. Вы видите, где заложена бомба, готовая вот-вот взорваться? Скорее всего, нет. Но давайте посмотрим на следующую распечатку:

```
MariaDB [test]> SELECT * FROM data;
+-----+
| id | data |
+-----+
```

```
| 1 | 99.99 |
+-----+
1 row in set (0.00 sec)
```

Если мне не изменяет память, мы пытались добавить восьмизначное число, и это вообще не должно было работать. Однако MariaDB просто взяла и изменила мои данные. Конечно, выдано предупреждение, но какая разница, если содержимое таблицы никак не соотносится с тем, что было вставлено на самом деле?

Попробуем проделать то же самое в PostgreSQL:

```
test=# CREATE TABLE data
(
    id integer NOT NULL,
    data numeric(4, 2)
);
CREATE TABLE
test=# INSERT INTO data VALUES (1, 1234.5678);
ОШИБКА: переполнение поля numeric
ПОДРОБНОСТИ: поле с точностью 4, порядком 2 должно округляться до
абсолютного значения меньше, чем 10^2.
```

Таблица создана, как и раньше, но кардинальное отличие от MariaDB/MySQL состоит в том, что PostgreSQL выдает ошибку при попытке вставить в таблицу недопустимое значение. Какой смысл четко указывать, чего мы хотим, если база данных плюет на это? Представьте, что вы выиграли в лотерею, а система не выдала пару миллионов долларов, потому что решила, что так для вас будет лучше!

Всю свою жизнь я сражаюсь с коммерческими СУБД, но никогда не встречал ничего подобного ни в какой дорогущей коммерческой системе (Oracle, Db2, Microsoft SQL Server и т. д.). У них, конечно, есть свои проблемы, но к данным они относятся бережно.

Изменение определения столбцов

Посмотрим, что произойдет, если мы попробуем изменить определение таблицы:

```
MariaDB [test]> ALTER TABLE data MODIFY data numeric(3, 2);
Query OK, 1 row affected, 1 warning (0.06 sec)
Records: 1 Duplicates: 0 Warnings: 1
```

Вот она, проблема:

```
MariaDB [test]> SELECT * FROM data;
+-----+
| id | data |
+-----+
| 1  | 9.99 |
+-----+
1 row in set (0.00 sec)
```

Как видите, данные снова модифицированы. Этого не должно было быть изначально, но случилось опять. Мы еще раз могли бы потерять деньги или еще какие-то ценности, потому что MySQL стремится быть умнее нас.

А вот как это выглядит в PostgreSQL:

```
test=# INSERT INTO data VALUES (1, 34.5678);
INSERT 0 1
test=# SELECT * FROM data;
 id | data
-----+-----
  1 | 34.57
(1 строка)
```

Теперь изменим определение столбца:

```
test=# ALTER TABLE data ALTER COLUMN data TYPE numeric(3, 2);
ОШИБКА: переполнение поля numeric
ПОДРОБНОСТИ: поле с точностью 3, порядком 2 должно округляться до
абсолютного значения меньше, чем 10^1.
```

И снова PostgreSQL выдает ошибку, не позволяя нам изгадить данные. И такого же поведения мы ожидаем от любой авторитетной СУБД. Правило простое: PostgreSQL и другие базы не позволяют нам уничтожить собственные данные.

Однако кое-что PostgreSQL позволяет сделать:

```
test=# ALTER TABLE data
ALTER COLUMN data TYPE numeric(3, 2) USING (data / 10);
ALTER TABLE
```

Мы можем явно сказать системе, что делать. В данном случае мы ясно указали, что значение столбца нужно разделить на 10. Разработчик вправе задать явные правила, которые следует применить к данным. PostgreSQL не пытается быть умнее всех – и не без причины:

```
test=# SELECT * FROM data;
 id | data
-----+-----
  1 | 3.46
(1 строка)
```

Данные изменены точно так, как мы и хотели.

Обработка значений null

Мы не хотим превращать эту главу в жалобы на то, что MariaDB – плохая СУБД, но этот последний пример, на мой взгляд, очень важен.

```
MariaDB [test]> UPDATE data SET id = NULL WHERE id = 1;
Query OK, 1 row affected, 1 warning (0.01 sec)
Rows matched: 1 Changed: 1 Warnings: 1
```

Столбец id явно помечен как NOT NULL:

```
MariaDB [test]> SELECT * FROM data;
+-----+
| id | data |
+-----+
| 0  | 9.99 |
+-----+
1 row in set (0.00 sec)
```

Очевидно, MySQL и MariaDB полагают, что null и ноль – одно и то же. Попробую проиллюстрировать эту проблему с помощью простой аналогии: знать, что в бумажнике ничего нет, и не знать, сколько в нем денег, – совсем не одно и то же. Когда я пишу эти строки, я не знаю, сколько у меня с собой денег (null = unknown), но я абсолютно уверен, что больше нуля (я точно знаю, что их хватит, чтобы заправить мою ласточку по пути домой из аэропорта, что было бы затруднительно без гроша в кармане).

Вот еще более пугающий сценарий:

```
MariaDB [test]> DESCRIBE data;
+-----+-----+-----+-----+-----+-----+
| Field | Type          | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| id    | int(11)       | NO   |     | NULL    |       |
| data  | decimal(3,2)  | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

MariaDB помнит, что столбец был помечен как NOT NULL, но в очередной раз изменяет данные, которые вы задали.

Ждите проблем

Основная проблема заключается в тех неприятностях, которые грозят нам при переносе данных в PostgreSQL. Представьте, что мы хотим перенести данные, и на стороне PostgreSQL имеется ограничение NOT NULL. Мы знаем, что MySQL на него наплевать:

```
MariaDB [test]> SELECT
    CAST('2014-02-99 10:00:00' AS datetime) AS x,
    CAST('2014-02-09 10:00:00' AS datetime) AS y;
+-----+-----+
| x    | y    |
+-----+-----+
| NULL | 2014-02-09 10:00:00 |
+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

PostgreSQL, безусловно, отвергнет дату 99 февраля (и это правильно), но и NULL не сможет принять, если это явно запрещено (и это тоже правильно). Вам придется как-то исправить данные, чтобы они удовлетворяли правилам модели, которые введены не без причины. И к этому не следует относиться с легким сердцем, потому что вы собираетесь изменить данные, чего в принципе делать не следует.

Перенос данных и схемы

После того как я объяснил, почему переход на PostgreSQL – здравая мысль, и обрисовал некоторые наиболее серьезные трудности, настало время рассказать о нескольких способах избавиться, наконец, от MySQL/MariaDB.

Использование *pg_chameleon*

Один из способов перейти с MySQL/MariaDB на PostgreSQL – воспользоваться программой Федерико Камполи (Federico Campoli) *pg_chameleon*, которую можно скачать с GitHub (https://github.com/the4thdoctor/pg_chameleon). Она специально разработана для переноса данных в PostgreSQL и выполняет за нас кучу работы, в т. ч. преобразование схемы.

В этой программе можно выделить четыре основных шага.

1. *pg_chameleon* читает схемы и данные из MySQL и создает схему в PostgreSQL.
2. Программа запоминает в PostgreSQL основную информацию о подключении к MySQL.
3. В PostgreSQL создаются первичные ключи и индексы.
4. Данные копируются из MySQL/MariaDB в PostgreSQL.

Программа *pg_chameleon* поддерживает основные DDL-команды, например: CREATE, DROP, ALTER TABLE и DROP PRIMARY KEY. Однако в силу самой природы MySQL/MariaDB она не может поддерживать все DDL-команды и ограничивается только самыми важными средствами.

Но это еще не все. Я не раз подчеркивал, что данные не всегда такие, какими должны быть. Подход *pg_chameleon* к этой проблеме простой: она отбрасывает некорректные данные, но сохраняет их в таблице *sch_chameleon.t_discarded_rows*. Конечно, это неидеальное, но – с учетом низкого качества входных данных – единственное разумное решение, которое приходит мне на ум. Пусть разработчик сам решает, что делать с некорректными строками; *pg_chameleon* не может и не должна принимать решения о том, как починить то, что сломано кем-то другим.

В разработку этого инструмента в последнее время вложено немало труда. Поэтому я искренне рекомендую заглянуть на его страницу в GitHub и проштудировать документацию. Добавление новых возможностей и исправление ошибок происходят прямо сейчас – пока вы читаете этот абзац. Недостаток места не позволяет мне изложить все подробности.



Хранимые процедуры, триггеры и прочие специальные объекты можно обработать только вручную. В *pg_chameleon* нет средств для их автоматической обработки.

Использование оберток внешних данных

Для перехода с MySQL/MariaDB на PostgreSQL есть не один способ. Альтернативой *pg_chameleon* является обертка внешних данных (FDW), которая позволяет быстро импортировать в PostgreSQL схему и данные. Возможность соединить MySQL и PostgreSQL существует довольно давно, поэтому FDW определенно заслуживает внимания.

В общем, расширение `mysql_fdw` работает, как любая другая обертка. По сравнению с другими, менее известными FDW, `mysql_fdw` весьма развито и предлагает следующие возможности:

- запись в базу MySQL/MariaDB;
- пул подключений;
- перенос фразы `WHERE` на удаленный сервер (это означает, что фильтры, применяемые к таблице, в действительности выполняются в удаленной базе для повышения производительности);
- ограничение состава столбцов (из удаленной базы выбираются только необходимые столбцы; в прежних версиях выбирались все столбцы, что приводило к увеличению сетевого трафика);
- подготовка команд на удаленном сервере.

Для использования расширения `mysql_fdw` необходимо выполнить команду `IMPORT FOREIGN SCHEMA`, которая позволяет впоследствии перенести данные в PostgreSQL. К счастью, в Unix-системах это довольно просто сделать. Рассмотрим последовательность шагов подробно.

Первым делом скачиваем код с GitHub:

```
git clone https://github.com/EnterpriseDB/mysql_fdw.git
```

Затем компилируем FDW. В вашей системе пути могут отличаться от указанных ниже. Я предположил, что MySQL и PostgreSQL установлены в каталог `/usr/local`, тогда как у вас все может быть по-другому:

```
$ export PATH=/usr/local/pgsql/bin/:$PATH
$ export PATH=/usr/local/mysql/bin/:$PATH
$ make USE_PGXS=1
$ make USE_PGXS=1 install
```

После того как код откомпилирован, FDW можно включить в базу данных:

```
CREATE EXTENSION mysql_fdw;
```

Следующий шаг – создать описание сервера, с которого мы хотим перенести данные:

```
CREATE SERVER migrate_me_server
  FOREIGN DATA WRAPPER mysql_fdw
  OPTIONS (host 'host.example.com', port '3306');
```

Затем создаем отображение пользователей:

```
CREATE USER MAPPING FOR postgres
  SERVER migrate_me_server
  OPTIONS (username 'joe', password 'public');
```

И наконец, приступаем к переносу. Сначала нужно импортировать схему. Я рекомендую создать специальную схему для связанных таблиц:

```
CREATE SCHEMA migration_schema;
```

В команде `IMPORT FOREIGN SCHEMA` мы сможем указать эту схему в качестве целевой – той, где будут храниться все внешние таблицы. А по завершении переноса мы ее просто удалим.

По завершении команды `IMPORT FOREIGN SCHEMA` можно создать реальные таблицы. Проще всего сделать это с помощью ключевого слова `LIKE`, которое можно указать в команде `CREATE TABLE`. Оно позволяет создать локальную таблицу в PostgreSQL с такой же структурой, как у внешней. Например:

```
CREATE TABLE t_customer  
(LIKE migration_schema.t_customer);
```

Теперь можно приступить к переносу данных:

```
INSERT INTO t_customer  
SELECT * FROM migration_schema.t_customer
```

Именно на этом шаге мы можем исправить данные, исключить некорректные строки или произвести минимальную обработку данных. Учитывая низкое качество исходных данных, полезно применить ограничения после первой попытки переноса. Такой подход может оказаться менее болезненным.

После импорта данных можно добавить ограничения, индексы и прочее. На этом шаге могут обнаружиться все те неприятные сюрпризы, о которых я говорил выше, поэтому не стоит ожидать, что данные будут непогрешимы. В случае MySQL миграция может оказаться довольно сложным процессом.

РЕЗЮМЕ

В этой главе мы узнали о переходе на PostgreSQL. Миграция – важная тема, поскольку все больше народу поглядывают в сторону PostgreSQL.

В версии PostgreSQL 11 появилось много новых возможностей, в т. ч. встроенное секционирование. В будущем мы увидим прогресс во всех частях PostgreSQL, а особенно в тех, которые обеспечивают более высокий уровень масштабирования и более быстрое выполнение запросов. Нам еще предстоит увидеть, что припасло для нас будущее.

Предметный указатель

A

adminpack модуль, [293](#)
ALTER TABLE команда, [23](#)
autovacuum, [41](#)

B

BRIN-индексы, [77](#)
btree_gin, установка, [296](#)
btree_gist, установка, [296](#)
B-деревья, свойства, [63](#)
 добавление данных во время
 индексирования, [66](#)
 комбинированные индексы, [63](#)
 создание класса операторов, [68](#)
 уменьшение занятого места
 на диске, [65](#)
 функциональные индексы, [64](#)

C

clog, повреждение, [320](#)
contrib, модули
 adminpack, [293](#)
 bloom, [294](#)
 btree_gin, [296](#)
 btree_gist, [296](#)
 dblink, [297](#)
 file_fdw, доступ к файлам, [298](#)
 pageinspect, анализ хранилища, [299](#)
 pg_buffercache, анализ кеша, [301](#)
 pgcrypto, шифрование, [302](#)
 pg_prewarm, прогрев кеша, [302](#)
 pg_stat_statements, анализ
 производительности, [304](#)
 pgstattuple, анализ хранилища, [304](#)
 pg_trgm, нечеткий поиск, [305](#)
 postgres_fdw, подключение
 к удаленному серверу, [305](#)
CREATE PUBLICATION, команда, [285](#)
CREATE SUBSCRIPTION, команда, [285](#)

D

dblink, [297](#)

E

EXPLAIN, команда, [50](#)

F

FETCH FIRST, фраза, [331](#)
file_fdw, расширение, [298](#)
FILTER, фраза, [94](#), [328](#)
FOR KEY SHARE, [36](#)
FOR NO KEY UPDATE, [36](#)
FOR SHARE, [33](#), [36](#)
FOR UPDATE, [33](#)

G

GET DIAGNOSTICS, команда, [195](#)
GIN-индексы, [76](#)
 расширение, [76](#)
GiST-индексы, [73](#)
 расширение, [75](#)

K

K ближайших соседей поиск, [296](#)

L

LDAP (облегченный протокол доступа
к каталогам), [225](#)
LIKE, предикат, ускорение запросов, [82](#)

M

MySQL и MariaDB, переход на
PostgreSQL, [338](#)

O

OFFSET, фраза, поддержка, [331](#)
ora2pg, [334](#)
oracle_fdw, расширение, [333](#)
Oracle, переход на PostgreSQL, [332](#)
 ora2pg, [334](#)
 ora_migrator, [337](#)

подводные камни, 336
 расширение oracle_fdw, 333
 orafce, расширение, 337
 ora_migrator, 337

Р

pageinspect, расширение, 299
 perf, 316
 pg_buffercache, расширение, 301
 pg_chameleon, программа, 343
 pg_crash, расширение, 310
 pgcrypto, расширение, 302
 pg_dump, задание информации
 о подключении, 247
 pg_hba.conf, файл, 224
 pg_prewarm, расширение, 302
 pg_squeeze, расширение, 43, 324
 pg_stat_activity, расширение
 выяснение источника запроса, 314
 и команды, сгенерированные
 Hibernate, 313
 опрос, 312
 pg_stat_statements, представление, 17,
 131, 304
 pgstattuple, расширение, 304
 pg_trgm, расширение, 80
 pgwatch2, программа, 311
 PL/Perl
 абстрагирование типа данных, 206
 и PL/PerlU, 207
 интерфейс SPI, 208
 написание триггеров, 211
 общие сведения, 205
 сохранение данных между вызовами
 функций, 210
 функции, 210
 экранирование, 210
 PL/pgSQL
 GET DIAGNOSTICS, команда, 195
 курсоры, 196
 написание триггеров, 200
 обработка ошибок, 193
 создание хранимых процедур, 204
 составные типы данных, 199
 управление областями видимости, 193
 экранирование, 190

PL/Python

интерфейс SPI, 212
 написание кода, 211
 обработка ошибок, 213
 общие сведения, 211

PostGIS, 310

postgres_fdw, расширение, 305
 PostgreSQL 11.0, нововведения, 16
 postgresql.conf

медленные запросы, 136
 местонахождение журналов и
 порядок ротации, 135
 настройка syslog, 136
 протоколирование, 137

PostgreSQL, системные представления

pg_stat_archiver, архивация, 127
 pg_stat_bgwriter, мониторинг
 фонового рабочего процесса, 126
 pg_stat_replication, потоковая
 репликация, 127
 pg_stat_user_indexes, изучение
 индексов, 125
 pg_stat_user_tables, 124
 pg_stat_wal_receiver, потоковая
 репликация, 128
 общие сведения, 117

R

RADIUS протокол
 аутентификации, 226

S

SAVEPOINT, 28
 SP-GiST-индексы, 77
 SSI-транзакции, 38

T

TABLESAMPLE фраза, 330
 поддержка, 330

V

VACUUM

зацикливание идентификаторов
 транзакций, 42
 наблюдение за работой, 43
 настройка, 41

VACUUM FULL, 43

W

WITHIN GROUP, фраза, поддержка, 329
WITH RECURSIVE, фраза,
поддержка, 328
WITH, фраза, поддержка, 327

X

xlog, 256

A

Административные задачи,
ускорение, 175
Администрирование базы данных, 17
 задание размера сегментов WAL, 17
 модуль pg_stat_statements, 17
Адреса привязки, 221
Аналитические средства, 97
Анонимные блоки кода, 187
Асинхронная репликация
 возобновление, 270
 линии времени, 274
 настройка, 268
 остановка, 270
 отработка отказов, 274
 повышение надежности, 276
 повышение уровня безопасности, 270
 проверка состояния, 271
 управление конфликтами, 275

Б

Базовая резервная копия, 261
 отображение табличных
 пространств, 262
 проверка результата архивации
 журнала транзакций, 262
 различные форматы, 262
 уменьшение полосы
 пропускания, 262
Безопасность
 на уровне базы данных, 232
 на уровне экземпляра, 230
Блокировка, 30
 предотвращение типичных
 ошибок, 31
 различные режимы, 32
 явная, 31
Блума фильтр, 79, 294

В

Ввода-вывода анализ, 318
Взаимоблокировки, 38
Внешние соединения, 158
Восстановление из резервной
копии, 252
Восстановление на определенный
момент времени (PITR), 259
Временные ряды, сопоставление
с образцом, 332
Встраивание функций, 148
Встроенные оконные функции
 first_value(), 107
 lag(), 105
 last_value(), 108
 lead(), 106
 nth_value(), 108
 ntile(), 104
 rank(), 103
 row_number(), 109
Выгрузка данных, 246
 извлечение подмножества
 данных, 250
 команда pg_dump, 247
Вызовы функций
 использование кешированных
 планов, 215
 назначение стоимости
 функции, 216
 уменьшение числа вызовов, 214

Г

Генетическая оптимизация
запросов, 163
Геоинформационные системы
(ГИС), 310
Гипотетические агрегаты, 97
 написание, 114
Глобальные данные, 253

Д

Двусторонняя репликация, 285
Долларовые кавычки, 187

Ж

Журнал предзаписи (WAL), 256

Журнал транзакций
 воспроизведение, 263
 контрольные точки, 257
 настройка архивации, 259
 нахождение временной метки, 265
 общие сведения, 255
 оптимизация, 257
 очистка архива, 267
 просмотр, 256
 создание базовой резервной копии, 261
 файл pg_hba.conf, 260
 Журналы, создание, 134

И

Индексирование,
 усовершенствования, 18
 параллельное построение индексов, 19
 покрывающие индексы, 19
 статистика по выражению, 18

Индексы, типы, 73

BRIN-индексы, 77

GIN-индексы, 76

GiST-индексы, 73

SP-GiST-индексы, 77

добавление новых, 79

хеш-индексы, 73

Исчерпывающий поиск, 145

К

Карта свободного пространства, 41

Кластеризованные таблицы, 59

просмотр только индекса, 62

Комбинированные индексы, 63

Контрольная точка, 257

сообщения, 321

Л

Латеральные соединения, 325

поддержка, 326

Логические слоты репликации,

примеры применения, 285

М

Медленные запросы, 314

анализ с помощью perf, 316

Межстолбцовая корреляция, 155

Методика хранения сверхбольших атрибутов (TOAST), 124

Методы аутентификации, 225

Многоверсионное управление конкурентностью (MVCC), 30

Н

Наборы группирования, 90

изучение производительности, 93

поддержка, 326

сочетание с фразой FILTER, 94

Нечеткий поиск, 80

расширение pg_trgm, 80

регулярные выражения, 83

ускорение запросов с предикатом

LIKE, 82

О

Обертка внешних данных, 333, 343

Общие табличные выражения (CTE), 327

Ограничения в виде равенства, 145

Оконные функции

абстрагирование окон, 102

встроенные, 103

использование, 97

поддержка, 329

разбиение данных, 98

скользящие окна, 100

упорядочение данных внутри окна, 99

Оперативная обработка транзакций (OLTP), 24, 37

Операторы, классы

для В-дерева, создание, 68

общие сведения, 67

создание, 71

тестирование пользовательских классов, 72

Операторы исключения, 87

Оптимизация

варианты соединения, оценка, 142

включение и выключение

режимов, 160

встраивание функций, 148

генетическая, 163

исчерпывающий поиск, 145

на примере, 142

- применение ограничений в виде равенства, 145
- применение преобразований, 144
- сворачивание констант, 147
- ускорение теоретико-множественных операций, 150
- устранение соединений, 149

Отличительное имя, 129

Очистка, управление, 41

П

Параллельное построение индексов, 19

Пароли

- задание пароля, 248
- использование переменных окружения, 248
- использование файла службы, 249
- использование файлов .pgpass, 249

Передача объектов, 243

Переменные окружения, 248

Планы выполнения

- команда EXPLAIN, 152
- общие сведения, 151
- систематический подход, 151

Планы выполнения, проблемы

- анализ времени выполнения, 153
- анализ использования буферов, 155
- анализ оценок, 153
- частое чтение с диска, 157

Поврежденные страницы данных, 322

Подключаемый модуль

аутентификации (PAM), 225

Покрывающие индексы, 19, 63

Полнотекстовый поиск

- GIN-индексы, 85
- общие сведения, 83
- операторы исключения, 87
- отладка поиска, 86
- сбор статистики по словам, 87
- сравнение строк, 84

Пользовательские агрегаты

- гипотетические, 114
- повышение эффективности, 113
- поддержка параллельных запросов, 112
- создание, 109

Пользовательские классы операторов, тестирование, 72

Преобразования

- встраивание представления, 144
- применение, 144
- распавление подзапроса, 144

Просмотр по битовой карте, 56

Просмотр только индекса, 62

Р

Разбухание таблиц, 304, 323

Распараллеливание запросов, 176

Расширения, 289

- получение перечня доступных, 290

Регулярные выражения, 83

Рекомендательные блокировки, 40

С

Сворачивание констант, 147

Секционирование данных, 164

- в PostgreSQL 11.0, 169

- в версии PostgreSQL 11, 21

- модификация наследуемой структуры, 167

- очистка данных, 168

- перемещение таблицы, 168

- применение табличных

- ограничений, 166

- создание секций, 164

Сетевая безопасность

- адреса привязки, 221

- на уровне баз данных, 232

- на уровне столбцов, 237

- на уровне строк, 238

- на уровне схемы, 233

- на уровне таблиц, 235

- на уровне экземпляра, 228

- подключение, 221

- управление, 220

- файл pg_hba.conf, 224

Синхронная репликация, 277

- настройка долговечности, 279

Системные представления

- SSL-подключения, 129

- динамическая проверка выполняемых запросов, 118

инспекция транзакций, 130
 использование `pg_stat_statements`, 131
 мониторинг очистки, 130
 получение информации о базах данных, 120
 Скользящие окна, 100
 Слоты репликации, 280
 логические, 283
 физические, 281
 Соединение, методы, 142
 вложенные циклы, 142
 соединение слиянием, 143
 соединение хешированием, 143
 устранение соединений, 149
 Соединение хешированием, 143
 Соединения
 `join_collapse_limit` параметр, 159
 обработка внешних соединений, 158
 осмысление и исправление, 157
 Сортировка результатов, 55
 использование нескольких индексов одновременно, 56
 Спинлок, состязание за, 316
 Статистика работы системы, 117
 Стоимостная модель, 49, 52

Т

Темпоральные таблицы, поддержка, 332
 Типичные ошибки, 320
 беззаботное управление подключениями, 323
 повреждение `clog`, 320
 поврежденные страницы данных, 322
 разбухание таблиц, 323
 сообщения о контрольной точке, 321
 Транзакции, 25
 использование `SAVEPOINT`, 28
 обработка ошибок, 27
 ограничение длительности, 46
 транзакционные DDL-команды, 29

уровни изоляции, 36

У

Узел сбора, 51
 Упорядоченные наборы, 95
 Управление кешем, 20
 Управление подключениями без использования TCP, 222
 общие сведения, 221
 производительность, 222

Ф

Функции
 анатомия, 186
 анонимные блоки кода, 187
 долларовые кавычки, 187
 использование, 216
 и хранимые процедуры, 185
 улучшение, 214
 Функции, возвращающие множество, 196
 Функциональные индексы, 64

Х

Хеш-индексы, 73
 Хранилище
 анализ с помощью `pageinspect`, 299
 анализ с помощью `pgstattuple`, 304
 оптимизация, 41
 Хранимые процедуры, 22
 и функции, 185
 языки, 184
 PL/Perl, 205
 PL/pgSQL, 190
 PL/Python, 211

Ш

Шифрование данных с помощью `pgcrypto`, 302

Э

Экранирование, 190

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.a-planet.ru**.

Оптовые закупки: тел. **(499) 782-38-89**.

Электронный адрес: **books@alians-kniga.ru**.

Ганс-Юрген Шениг

PostgreSQL 11 **Мастерство разработки**

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com

Технический редактор *Рогов Е. В.*
Перевод *Слинкин А. А.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.

Гарнитура «PT Serif». Печать офсетная.

Усл. печ. л. 28,6. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**