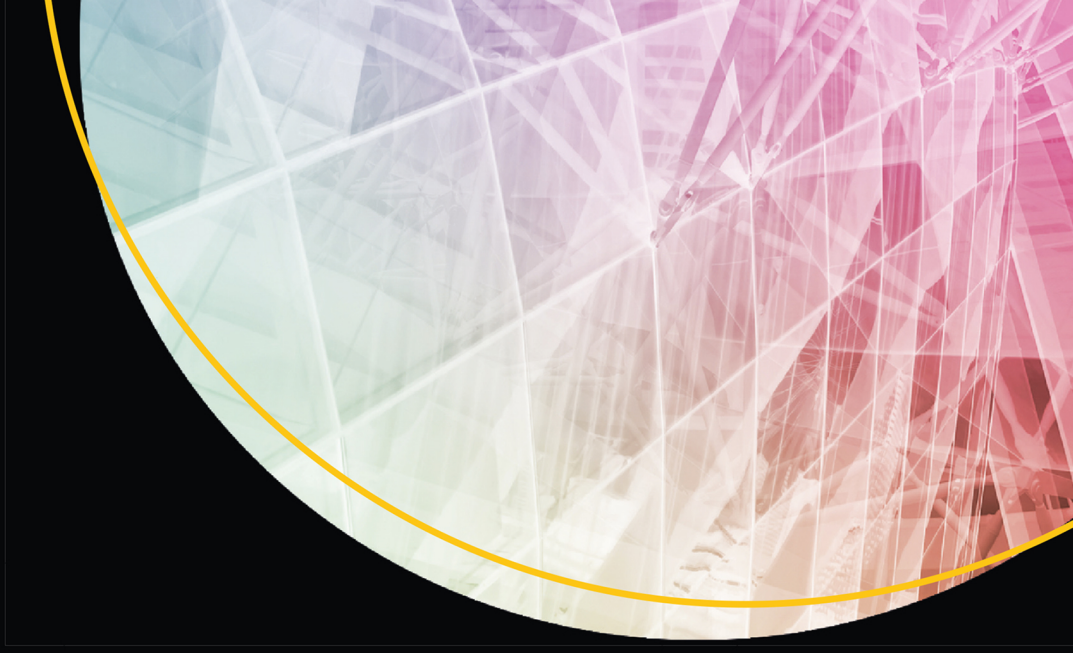


Оптимизация запросов в PostgreSQL

Домбровская Г.
Новиков Б.
Бейликова А.

ДМК Пресс
издательство

Генриэтта Домбровская
Борис Новиков
Анна Бейликова

Оптимизация запросов в PostgreSQL

PostgreSQL Query Optimization

The Ultimate Guide to Building Efficient Queries

Henrietta Dombrovskaya
Boris Novikov
Anna Bailliekova

Apress®

Оптимизация запросов в PostgreSQL

Полное руководство по созданию эффективных запросов

Генриэтта Домбровская
Борис Новиков
Анна Бейликова

ДМК Пресс
издательство
Москва, 2022

УДК 004.655
ББК **32.973.26-018.2**
Д66

Домбровская Г., Новиков Б., Бейликова А.
Д66 Оптимизация запросов в PostgreSQL / пер. с англ. Д. А. Беликова. – М.: ДМК Пресс, 2022. – 278 с.: ил.

ISBN 978-5-97060-963-7

Книга поможет вам писать запросы, которые выполняются быстро и вовремя доставляют результаты. Вы научитесь смотреть на процесс написания запроса с точки зрения механизма базы данных и начнете думать, как оптимизатор базы данных. Объясняется, как читать и понимать планы выполнения запросов, какие существуют методы воздействия на них с точки зрения оптимизации производительности, и показано, как эти методы используются вместе для создания эффективных приложений.

Издание предназначено разработчикам и администраторам баз данных, а также системным архитекторам, использующим PostgreSQL.

УДК 004.655
ББК 32.973.26-018.2

ISBN 978-1-4842-6884-1 (англ.)

ISBN 978-5-97060-963-7 (рус.)

Содержание

От издательства

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 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 укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Скачивание исходного кода примеров

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

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Apress очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Об авторах

Генриэтта Домбровская – исследователь и разработчик баз данных с более чем 35-летним академическим и производственным опытом. Она имеет докторскую степень в области компьютерных наук Санкт-Петербургского университета. В настоящее время она является заместителем директора по базам данных в Braviant Holdings, Чикаго, Иллинойс и активным членом сообщества PostgreSQL, часто выступает на конференциях PostgreSQL, а также является местным организатором группы пользователей PostgreSQL в Чикаго. Ее исследовательские интересы тесно связаны с практикой и сосредоточены на разработке эффективных взаимодействий между приложениями и базами данных. Лауреат премии «Технолог года» 2019 Технологической ассоциации штата Иллинойс.

Борис Новиков в настоящее время является профессором департамента информатики Национального исследовательского университета «Высшая школа экономики» в Санкт-Петербурге. Окончил механико-математический факультет Ленинградского университета. Проработал много лет в Санкт-Петербургском университете и перешел на свою нынешнюю должность в январе 2019 года. Его исследовательские интересы лежат в широкой области управления информацией и включают в себя аспекты проектирования, разработки и настройки баз данных, приложений и систем управления базами данных (СУБД). Также интересуется распределенными масштабируемыми системами для потоковой обработки и аналитики.

Анна Бейликова – старший инженер по обработке данных в компании Zendesk. Ранее она занималась созданием конвейеров ETL, ресурсов хранилищ данных и инструментов для ведения отчетности в качестве руководителя группы подразделения операций в компании Epic, а также занимала должности аналитика в различных политических кампаниях и в Greenberg Quinlan Rosner Research. Получила степень бакалавра с отличием в области политологии и информатики в колледже Нокс в Гейлсбурге, штат Иллинойс.

О техническом редакторе

Том Кинкейд – вице-президент по техническим операциям в компании EnterpriseDB. Том занимается разработкой, развертыванием и поддержкой систем баз данных и корпоративного программного обеспечения более 25 лет. До прихода в EnterpriseDB Том был генеральным менеджером 2ndQuadrant в Северной Америке, где курировал все аспекты динамичного и растущего бизнеса 2ndQuadrant для продуктов Postgres, обучения, поддержки и профессиональных услуг. Он работал напрямую с компаниями из всех отраслей и любого размера, помогая им успешно задействовать Postgres в своих критически важных операциях.

Ранее Том был вице-президентом по профессиональным услугам, а затем вице-президентом по продуктам и инжинирингу в EnterpriseDB, крупнейшей в мире компании, которая является поставщиком продуктов и услуг корпоративного класса на основе PostgreSQL.

Он курировал разработку и поставку обучающих решений Postgres, а также развертывание PostgreSQL как в финансовых учреждениях, входящих в список Fortune 500, так и на военных объектах по всему миру. Команды, которыми управлял Том, разработали важные функции, которые стали частью базы данных с открытым исходным кодом PostgreSQL. Он курировал разработку и успешную доставку продуктов высокой доступности для PostgreSQL и других баз данных.

Том также является основателем и организатором группы пользователей PostgreSQL в Бостоне.

Благодарности

Авторы хотят поблагодарить Джонатана Генника, Джилла Бальцано и всех сотрудников Apress за возможность поделиться своей точкой зрения.

Чэд Слотер и Джон Уолш были первыми читателями и предоставили бесценные отзывы. Алисса Ричи предоставила классы Java, чтобы показать код приложения, которое использовалось в качестве примера.

Вклад Тома Кинкейда как технического рецензента невозможно переоценить. Его внимательные, подробные и вдумчивые отзывы улучшили содержание, организацию и удобство использования текста. Благодаря Тому эта книга стала более точной, понятной и целостной. Ответственность за все оставшиеся вопросы, конечно же, ложится на авторов.

Генриэтта Домбровская хотела бы поблагодарить Чэда Слотера, системного архитектора из компании Enova International, и Джефа Йонжевича, руководителя ее команды, которые поверили в нее и позволили ей работать по-другому. Джефф Чаплевски, Алисса Ричи и Грег Нельсон часами, днями и неделями заставляли NORM работать с Java. Алисса и Джефф также внесли вклад в статьи, получившие международное признание за этот подход. Боб Сайдс из Braviant Holdings рискнул и позволил Генриэтте работать так, как никто раньше не делал, и доказать силу этого подхода.

Анна Бейликова хотела бы поблагодарить Энди Чиветтини за то, что он научил ее писать на сложные и технические темы доступным языком, а также за годы академического и профессионального наставничества и поддержки. Команда отдела операций в Epic стремится к постоянному совершенствованию; влияние членов команды ощущается каждый раз, когда она пишет SQL.

Наконец, Джон, Надя и Кира Бейликовы поддерживали ее в ходе написания этой книги. Анна им бесконечно благодарна.

Вступление

«Оптимизация» – достаточно широкий термин, охватывающий настройку производительности, личное улучшение и маркетинг через социальные сети, и неизменно вызывает большие надежды и ожидания читателей. Поэтому мы считаем благоразумным начать эту книгу не с введения в предмет обсуждения, а с того, почему эта книга существует и что остается за ее рамками, чтобы не разочаровывать читателей, которые могут ожидать от нее другого. Затем мы переходим к тому, о чем эта книга, о ее целевой аудитории, о том, что она охватывает, и о том, как извлечь из нее максимальную пользу.

Почему мы написали эту книгу

Как и многие авторы, мы написали эту книгу, потому что чувствовали, что не можем не написать ее. Мы сами и преподаватели, и практики; следовательно, мы видим, как и что изучают студенты и каких знаний им не хватает, когда они попадают на работу. Нам не нравится то, что мы видим, и надеемся, что данная книга поможет восполнить этот пробел.

Изучая управление данными, большинство студентов никогда не видят реальных промышленных баз данных, и, что еще более тревожно, их никогда не видели и многие из преподавателей. Отсутствие доступа к реальным системам влияет на всех студентов, изучающих информатику, но больше всего страдает образование будущих разработчиков баз данных и администраторов баз данных (DBA).

Используя небольшую обучающую базу данных, можно научиться писать синтаксически правильный SQL и даже написать запрос, который получает желаемый результат. Однако для обучения написанию эффективных запросов требуется набор данных промышленного размера. Более того, когда студент работает с набором данных, который легко помещается в оперативную память компьютера, и получает результат за миллисекунды независимо от сложности запроса, для него может быть неочевидно, что с производительностью могут возникнуть какие-то проблемы.

Помимо того что студенты незнакомы с реалистичными наборами данных, они часто используют не те СУБД, которые широко применяются на практике. Хотя предыдущее утверждение верно в отношении многих СУБД, в случае с PostgreSQL оно вызывает еще большее разочарование. PostgreSQL возникла в академической среде и поддерживается как проект с открытым исходным кодом, что делает ее идеальной базой данных для обучения реляционной теории и демонстрации внутреннего устройства систем баз данных. Однако пока очень немногие академические учреждения используют PostgreSQL для своих образовательных нужд.

В то время как PostgreSQL быстро развивается и становится все более мощным инструментом, все больше и больше компаний предпочитают ее проприетарным СУБД в попытке сократить расходы. Все больше и больше ИТ-менеджеров ищут сотрудников, знакомых с PostgreSQL. Все больше и больше потенциальных кандидатов учатся использовать PostgreSQL самостоятельно и упускают возможность получить от нее максимальную отдачу.

Мы надеемся, что эта книга поможет всем заинтересованным сторонам: кандидатам, менеджерам по найму, разработчикам баз данных и организациям, которые переходят на PostgreSQL для удовлетворения своих потребностей в данных.

О чем не говорится в этой книге

Многие считают, что оптимизация – это своего рода магия, которой обладает элитный круг волшебников. Они верят, что могут быть допущены в этот круг, если получат от старейшин ключи к священным знаниям, и тогда возможности их станут безграничны.

Поскольку мы знаем об этих заблуждениях, то хотим, чтобы все было прозрачно с самого начала. Ниже приводится список тем, которые часто обсуждаются в книгах по оптимизации, но которые не будут рассмотрены в этой книге:

- *оптимизация сервера* – потому что ее не требуется выполнять ежедневно;
- *большинство системных параметров* – потому что у разработчиков баз данных вряд ли будут привилегии изменять их;
- *распределенные системы* – потому что у нас недостаточно промышленного опыта работы с ними;
- *транзакции* – потому что их влияние на производительность очень ограничено;
- *новые и крутые функции* – потому что они меняются с каждым новым выпуском, а наша цель – охватить основы;
- *черная магия* (заклинания, ритуалы и т. д.) – потому что мы в них не разбираемся.

Существует множество книг, охватывающих все темы, перечисленные в предыдущем списке, за исключением, вероятно, черной магии, но эта книга не входит в их число. Вместо этого мы сосредоточимся на повседневных задачах, с которыми сталкиваются разработчики баз данных: невозможно дождаться открытия страницы приложения; клиент вылетает из приложения прямо перед страницей «контракт подписан»; вместо ключевого показателя эффективности продукта генеральный директор смотрит на песочные часы; и проблему нельзя решить приобретением дополнительного оборудования.

Все, что мы представляем в этой книге, было протестировано и реализовано в промышленном окружении, и хотя это и может показаться черной магией, мы объясним все улучшения производительности запросов (или отсутствие таких улучшений).

Целевая аудитория

В большинстве случаев книга об оптимизации рассматривается как книга для администраторов баз данных. Поскольку наша цель состоит в том, чтобы доказать, что оптимизация – это больше, чем просто создание индексов, мы надеемся, что книга будет полезна для более широкой аудитории.

Эта книга предназначена для ИТ-специалистов, работающих с PostgreSQL, которые хотят разрабатывать производительные и масштабируемые приложения. Она для всех, чья должность содержит слова «разработчик базы данных» или «администратор базы данных», и для серверных разработчиков, которые взаимодействуют с базой данных. Она также полезна для системных архитекторов, участвующих в общем проектировании систем приложений, работающих с базой данных PostgreSQL.

А как насчет составителей отчетов и специалистов по бизнес-аналитике? К сожалению, большие аналитические отчеты чаще всего считаются медленными по определению. Но если отчет написан без учета того, как он будет работать, время выполнения может составить не минуты или часы, а годы! Для большинства аналитических отчетов время выполнения можно значительно сократить, используя простые методы, описанные в этой книге.

Что узнают читатели

Из этой книги читатели узнают, как:

- определить цели оптимизации в системах OLTP (оперативная обработка транзакций) и OLAP (интерактивная аналитическая обработка);
- читать и понимать планы выполнения PostgreSQL;
- выбрать индексы, которые улучшат производительность запросов;
- оптимизировать полное сканирование таблиц;
- различать длинные и короткие запросы;
- выбрать подходящую технику оптимизации для каждого типа запроса;
- избегать подводных камней ORM-фреймворков.

В конце книги мы представляем *полный и окончательный алгоритм оптимизации*, который поможет разработчику базы данных в процессе создания наиболее эффективного запроса.

База данных Postgres Air

Примеры в этой книге построены на основе одной из баз данных виртуальной авиакомпании Postgres Air. Эта компания соединяет более 600 виртуальных направлений по всему миру, еженедельно предлагает около 32 000 прямых виртуальных рейсов, у нее более 100 000 виртуальных участников программы для часто летающих пассажиров и намного больше обычных

пассажиров. Флот авиакомпании состоит из виртуальных самолетов. Поскольку все полеты полностью виртуальны, компания не затронута пандемией COVID-19.

Обратите внимание, что все данные, представленные в этой базе, являются вымышленными и представлены только в иллюстративных целях. Хотя некоторые данные кажутся очень реалистичными (особенно описания аэропортов и самолетов), их нельзя использовать в качестве источников информации о реальных аэропортах или самолетах. Все номера телефонов, адреса электронной почты и имена сгенерированы.

Чтобы установить учебную базу данных в вашей локальной системе, откройте каталог postgres_air_dump по этой ссылке: https://drive.google.com/drive/folders/13F7M80Kf_somnjb-mTYAnh1hW1Y_g4kJ?usp=sharing.

Вы также можете использовать QR-код на рис. B.1.

Рис. B.1 ❖ QR-код для доступа к дампу базы данных

Этот общий каталог содержит дамп данных схемы postgres_air в трех форматах: формат каталога, формат pg_dump по умолчанию и сжатый формат SQL.

Общий размер каждого дампа составляет около 1,2 ГБ. Используйте формат каталога, если вы предпочитаете скачивать файлы меньшего размера (максимальный размер файла 419 МБайт). Используйте формат SQL, если хотите избежать предупреждений о владельце объектов.

Для восстановления из формата каталога и формата по умолчанию используйте утилиту `pg_restore` (www.postgresql.org/docs/12/app-pgrestore.html). Для восстановления из формата SQL разархивируйте файл и используйте psql.

Кроме того, после восстановления данных вам нужно будет запустить сценарий из листинга B.1 для создания нескольких индексов.

Мы будем использовать эту схему базы данных, чтобы иллюстрировать концепции и методы, описанные в книге. Вы также можете использовать эту схему, чтобы попрактиковаться в методах оптимизации.

Схема содержит данные, которые могут храниться в системе бронирования авиакомпаний. Мы предполагаем, что вы хотя бы один раз бронировали

рейс онлайн, поэтому структура данных должна быть вам понятна. Конечно, структура этой базы данных намного проще, чем структура любой реальной базы данных такого типа.

Листинг B.1 ❖ Начальный набор индексов

```
SET search_path TO postgres_air;
CREATE INDEX flight_departure_airport   ON flight (departure_airport);
CREATE INDEX flight_scheduled_departure ON flight (scheduled_departure);
CREATE INDEX flight_update_ts           ON flight (update_ts);
CREATE INDEX booking_leg_booking_id     ON booking_leg (booking_id);
CREATE INDEX booking_leg_update_ts      ON booking_leg (update_ts);
CREATE INDEX account_last_name          ON account (last_name);
```

Любой человек, бронирующий рейс, должен создать учетную запись, которая используется для входа и содержит имя и фамилию, а также контактную информацию. Мы также храним данные о часто летающих пассажирах, которые могут быть привязаны к учетной записи. Забронировать рейсы можно для нескольких пассажиров, которые могут иметь, а могут и не иметь учетные записи в системе. Каждое бронирование может включать в себя несколько перелетов (называемых также сегментами). Перед полетом каждому пассажиру выдается посадочный талон с номером места.

Диаграмма «сущность–связь» для этой базы данных представлена на рис. B.2:

- *airport* хранит информацию об аэропортах и содержит трехсимвольный код (IATA), название аэропорта, город, географическое положение и часовой пояс;
- *flight* хранит информацию о рейсах. В таблице хранятся номер рейса, аэропорты прилета и вылета, запланированное и фактическое время прибытия и отправления, код самолета и статус рейса;
- *account* хранит учетные данные для входа, имя и фамилию владельца учетной записи и, возможно, ссылку на членство в программе для часто летающих пассажиров; в каждой учетной записи потенциально может быть несколько номеров телефонов, которые хранятся в таблице *phone*;
- *frequency_flyer* хранит информацию о членстве в программе для часто летающих пассажиров;
- *booking* содержит информацию о забронированных полетах (возможно, для нескольких пассажиров), каждый полет может иметь несколько сегментов;
- *booking_leg* хранит отдельные сегменты бронирований;
- *passenger* хранит информацию о пассажирах, привязанную к каждому бронированию. Обратите внимание, что идентификатор пассажира уникален для одного бронирования; для любого другого бронирования у того же человека будет другой идентификатор;
- *aircraft* предоставляет описание самолета;
- наконец, в таблице *boarding_pass* хранится информация о выданных посадочных талонах.

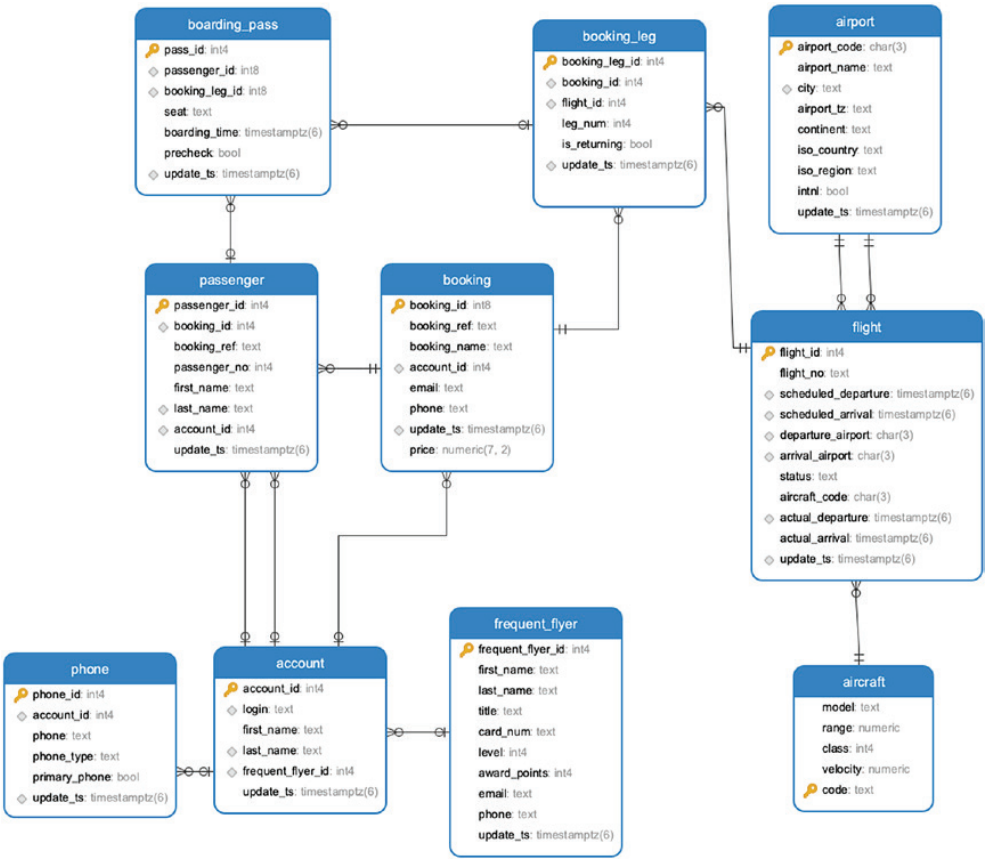


Рис. В.2 ❖ ER-диаграмма схемы бронирования

Глава 1

Зачем нужна оптимизация?

В этой главе рассказывается, почему оптимизация так важна для разработки баз данных. Вы узнаете о различиях между декларативными языками, такими как SQL, и, возможно, более знакомыми вам императивными языками, такими как Java, и о том, как эти различия влияют на стиль программирования. Мы также продемонстрируем, что оптимизация применяется не только к запросам, но и к проектированию баз данных и к архитектуре приложений.

Что подразумевается под оптимизацией?

В контексте данной книги оптимизация означает любое преобразование, улучшающее производительность системы. Это определение намеренно носит очень общий характер, поскольку мы хотим подчеркнуть, что оптимизация не является отдельным этапом разработки. Довольно часто разработчики баз данных сначала пытаются сделать так, чтобы «просто заработало», а уже потом переходят к оптимизации. Мы считаем такой подход непродуктивным. Написание запроса без представления о том, сколько времени потребуется для его выполнения, создает проблему, которой можно было бы избежать, правильно написав запрос с самого начала. Мы надеемся, что к тому времени, когда вы прочтете эту книгу, вы будете готовы рассматривать оптимизацию и разработку запросов как единый процесс.

Мы представим несколько конкретных техник; однако наиболее важно понимать, как движок базы данных обрабатывает запрос и как планировщик запросов решает, какой путь выполнения выбрать. Когда мы обучаем оптимизации на занятиях, то часто говорим: «Думайте как база данных!» Посмотрите на свой запрос с точки зрения движка базы данных и представьте, что он должен сделать, чтобы выполнить этот запрос; представьте, что вы, а не движок, должны выполнить запрос. Поразмыслив над объемом работы, вы можете избежать неоптимальных планов выполнения. Более подробно этот вопрос обсуждается в последующих главах.

Если вы достаточно долго будете «мыслить как база данных», это станет естественным способом мышления, и вы сразу сможете правильно писать запросы, часто без необходимости дальнейшей оптимизации.

Императивный и декларативный подходы: почему это сложно

Почему недостаточно написать инструкцию SQL, возвращающую правильный результат? Ведь так мы поступаем, когда пишем код приложения. Почему в SQL все иначе, и почему два запроса, дающих одинаковый результат, могут разительно отличаться по времени выполнения? Основной источник проблемы в том, что SQL – *декларативный язык*. Это означает, что когда мы пишем инструкцию SQL, то описываем результат, который хотим получить, но не указываем, *как* он должен быть получен. Напротив, в *императивном языке* мы указываем, *что* делать для получения желаемого результата, то есть записываем последовательность шагов, которые должны быть выполнены.

Как обсуждается в главе 2, *оптимизатор базы данных* выбирает лучший способ выполнить запрос. Что значит «лучший», определяется множеством различных факторов, таких как структура хранения, индексы и статистика.

Рассмотрим простой пример. Взгляните на запросы из листингов 1.1 и 1.2.

Листинг 1.1 ❖ Запрос для выбора рейса с оператором BETWEEN

```
SELECT flight_id,
       departure_airport,
       arrival_airport
  FROM flight
 WHERE scheduled_arrival BETWEEN '2020-10-14' AND '2020-10-15';
```

Листинг 1.2 ❖ Запрос для выбора рейса на определенную дату

```
SELECT flight_id,
       departure_airport,
       arrival_airport
  FROM flight
 WHERE scheduled_arrival::date = '2020-10-14';
```

Эти два запроса выглядят почти одинаково и должны давать одинаковые результаты. Тем не менее время выполнения будет разным, потому что работа, выполняемая движком базы данных, будет различаться. В главе 5 мы объясним, почему это происходит и как выбрать лучший запрос с точки зрения производительности.

Людям свойственно мыслить императивно. Обычно, когда мы думаем о выполнении задачи, то думаем о шагах, которые необходимо предпринять. Точно так же, когда мы думаем о сложном запросе, то думаем о последовательности условий, которые нужно применить для достижения желаемого

результата. Однако если мы заставим движок базы данных строго следовать этой последовательности, результат может оказаться неоптимальным.

Например, попробуем узнать, сколько часто летающих пассажиров с 4-м уровнем вылетают из Чикаго на День независимости. Если на первом этапе вы хотите выбрать всех часто летающих пассажиров с 4-м уровнем, то можно написать что-то вроде:

```
SELECT * FROM frequent_flyer WHERE level = 4
```

Затем можно выбрать номера их учетных записей:

```
SELECT * FROM account WHERE frequent_flyer_id IN (
    SELECT frequent_flyer_id FROM frequent_flyer WHERE level = 4
)
```

А потом, если вы хотите найти все бронирования, сделанные этими людьми, можно написать следующее:

```
WITH level4 AS (
    SELECT * FROM account WHERE frequent_flyer_id IN (
        SELECT frequent_flyer_id FROM frequent_flyer WHERE level = 4
    )
)
SELECT * FROM booking WHERE account_id IN (
    SELECT account_id FROM level4
)
```

Возможно, затем вы захотите узнать, какие из этих бронирований относятся к рейсам из Чикаго на 3 июля. Если вы продолжите строить запрос аналогичным образом, то следующим шагом будет код из листинга 1.3.

Листинг 1.3 ❖ Императивно построенный запрос

```
WITH bk AS (
    WITH level4 AS (
        SELECT * FROM account WHERE frequent_flyer_id IN (
            SELECT frequent_flyer_id FROM frequent_flyer WHERE level = 4
        )
    )
    SELECT * FROM booking WHERE account_id IN (
        SELECT account_id FROM level4
    )
)
SELECT * FROM bk WHERE bk.booking_id IN (
    SELECT booking_id FROM booking_leg
     WHERE leg_num=1
       AND is_returning IS false
       AND flight_id IN (
              SELECT flight_id FROM flight
               WHERE departure_airport IN ('ORD', 'MDW')
                 AND scheduled_departure::date = '2020-07-04'
          )
)
```

В конце можно подсчитать фактическое количество пассажиров. Это можно сделать с помощью запроса из листинга 1.4.

Листинг 1.4 ❖ Подсчет общего количества пассажиров

```
WITH bk_chi AS (
    WITH bk AS (
        WITH level4 AS (
            SELECT * FROM account WHERE frequent_flyer_id IN (
                SELECT frequent_flyer_id FROM frequent_flyer WHERE level = 4
            )
        )
        SELECT * FROM booking WHERE account_id IN (
            SELECT account_id FROM level4
        )
    )
    SELECT * FROM bk WHERE bk.booking_id IN (
        SELECT booking_id FROM booking_leg
         WHERE leg_num=1
           AND is_returning IS false
           AND flight_id IN (
                  SELECT flight_id FROM flight
                   WHERE departure_airport IN ('ORD', 'MDW')
                     AND scheduled_departure::date = '2020-07-04'
              )
    )
)
SELECT count(*) FROM passenger
WHERE booking_id IN (
    SELECT booking_id FROM bk_chi
)
```

При построенном таким образом запросе вы не даете планировщику запросов выбрать лучший путь выполнения, потому что последовательность действий жестко зашита в код. Хотя все строки написаны на декларативном языке, они императивны по своей природе.

Вместо этого, чтобы написать декларативный запрос, просто укажите, что вам нужно получить из базы данных, как показано в листинге 1.5.

Листинг 1.5 ❖ Декларативный запрос для расчета количества пассажиров

```
SELECT count(*)
  FROM booking bk
       JOIN booking_leg bl ON bk.booking_id = bl.booking_id
       JOIN flight f ON f.flight_id = bl.flight_id
       JOIN account a ON a.account_id = bk.account_id
       JOIN frequent_flyer ff ON ff.frequent_flyer_id = a.frequent_flyer_id
       JOIN passenger ps ON ps.booking_id = bk.booking_id
 WHERE level = 4
   AND leg_num = 1
   AND is_returning IS false
   AND departure_airport IN ('ORD', 'MDW')
   AND scheduled_departure BETWEEN '2020-07-04' AND '2020-07-05'
```

Таким образом, вы позволяете базе данных решить, какой порядок операций выбрать. Лучший порядок может отличаться в зависимости от распределения значений в соответствующих столбцах.

Эти запросы лучше выполнять после того, как будут построены все необходимые индексы в главе 5.

Цели оптимизации

До сих пор подразумевалось, что эффективный запрос – это запрос, который выполняется быстро. Однако это определение не является точным или полным. Даже если на мгновение мы сочтем сокращение времени выполнения единственной целью оптимизации, остается вопрос: какое время выполнения является «достаточно хорошим». Для ежемесячного финансового отчета крупной корпорации завершение в течение одного часа может быть отличным показателем. Для ежедневного маркетингового анализа минуты – отличное время выполнения. Для аналитической панели руководителя с дюжиной отчетов обновление в течение 10 секунд может быть хорошим достижением. Для функции, вызываемой из веб-приложения, даже сотня миллисекунд может оказаться недопустимо медленно.

Кроме того, для одного и того же запроса время выполнения может варьироваться в разное время дня или в зависимости от загрузки базы данных. В некоторых случаях нас может интересовать среднее время выполнения. Если у системы жесткий тайм-аут, нам может понадобиться измерить производительность, ограничив максимальное время исполнения. Есть также субъективная составляющая при измерении времени отклика. В конечном итоге компания заинтересована в удовлетворении потребностей пользователей; в большинстве случаев удовлетворенность пользователей зависит от времени отклика, но это также субъективная характеристика.

Однако помимо времени выполнения могут быть приняты во внимание и другие характеристики. Например, поставщик услуг может быть заинтересован в максимальном увеличении пропускной способности системы. Небольшой стартап может быть заинтересован в минимизации использования ресурсов без ущерба для времени отклика системы. Мы знаем одну компанию, которая увеличивала оперативную память, чтобы ускорить выполнение. Их целью было разместить в оперативной памяти всю базу данных. Некоторое время это помогало, пока база данных не превысила объем оперативной памяти всех доступных конфигураций.

Как определить цели оптимизации? Мы используем систему постановки целей SMART. Аббревиатура SMART означает:

- Specific – конкретность;
- Measurable – измеримость;
- Achievable или Attainable – достижимость;
- Result-based или Relevant – уместность;
- Time-bound – ограниченность во времени.

Большинство знают о целях SMART, которые применяются, когда речь идет о здоровье и фитнесе, но та же самая концепция прекрасно подходит и для оптимизации запросов. Примеры целей SMART представлены в табл. 1.1.

Таблица 1.1. Примеры целей SMART

Критерий	Плохой пример	Хороший пример
Конкретность	Все страницы должны отвечать быстро	Выполнение каждой функции должно быть завершено до заданного системой тайм-аута
Измеримость	Клиенты не должны ждать слишком долго, чтобы заполнить заявку	Время отклика страницы регистрации не должно превышать четырех секунд
Достижимость	Время ежедневного обновления данных в хранилище не должно увеличиваться	При росте объема исходных данных время ежедневного обновления данных должно увеличиваться не более чем логарифмически
Уместность	Каждое обновление отчета должно выполняться как можно быстрее	Время обновления для каждого отчета должно быть достаточно коротким, чтобы избежать ожидания блокировки
Ограниченность во времени	Оптимизируем столько отчетов, сколько можем	К концу месяца все финансовые отчеты должны выполняться менее чем за 30 секунд

Оптимизация процессов

Важно помнить, что база данных не существует в вакууме. Она является основой для нескольких, часто независимых приложений и систем. Все пользователи (внешние и внутренние) испытывают на себе именно общую производительность системы, и это то, что имеет для них значение.

На уровне организации цель состоит в том, чтобы добиться лучшей производительности всей системы. Это может быть время отклика или пропускная способность (что важно для поставщика услуг) либо (скорее всего) баланс того и другого. Никого не интересует оптимизация базы данных, которая не влияет на общую производительность.

Разработчики и администраторы баз данных часто склонны чрезмерно оптимизировать любой плохой запрос, который привлекает их внимание просто потому, что он плохой. При этом их работа нередко изолирована как от разработки приложений, так и от бизнес-аналитики. Это одна из причин, по которой усилия по оптимизации могут оказаться менее продуктивными, чем могли бы быть. SQL-запрос нельзя оптимизировать изолированно, вне контекста его назначения и окружения, в котором он выполняется.

Поскольку запросы можно писать не декларативно, первоначальная цель запроса может быть неочевидной. Выяснение того, что должно быть сделано с точки зрения бизнеса, – возможно, первый и самый важный шаг оптимизации. Более того, вопросы о цели отчета могут привести к выводу, что отчет вообще не нужен. Однажды вопросы о назначении наиболее длительных отчетов позволили нам сократить общий трафик на сервере отчетов на 40 %.

Оптимизация OLTP и OLAP

Есть много способов классификации баз данных, и разные классы баз данных могут отличаться как по критериям эффективности, так и по методам оптимизации. Два основных класса – это *OLTP* (оперативная обработка транзакций) и *OLAP* (интерактивная аналитическая обработка). OLTP-системы поддерживают приложения, а OLAP-системы – бизнес-аналитику и отчетность. На протяжении этой книги мы будем подчеркивать разные подходы к оптимизации OLTP и OLAP. Мы познакомим вас с понятиями *коротких* и *длинных* запросов, а также объясним, как их различать.

Подсказка Это не зависит от длины инструкции SQL.

В большинстве случаев в OLTP-системах оптимизируются короткие запросы, а в OLAP-системах – и короткие, и длинные запросы.

Проектирование базы данных и производительность

Мы уже упоминали, что нам не нравится концепция «сначала пиши, а потом оптимизируй» и что цель данной книги – помочь вам сразу же писать правильные запросы. Когда разработчику следует задуматься о производительности запроса, над которым он работает? Чем раньше, тем лучше. В идеале оптимизация начинается с требований. На практике это не всегда так, хотя сбор требований очень важен.

Говоря точнее, сбор требований позволяет спроектировать наиболее подходящую структуру базы данных, а ее структура может влиять на производительность.

Если вы администратор базы данных, то, скорее всего, время от времени вас будут просить проверить новые таблицы и представления, а это значит, что вам придется оценивать схему чужой базы данных. Если вы незнакомы с тем, что представляет собой новый проект, и не в курсе предназначения новых таблиц и представлений, вы вряд ли сможете определить, является ли предложенная структура оптимальной.

Единственное, что вы можете оценить, не вдаваясь в детали бизнес-требований, – нормализована ли база данных. Но даже это может быть неочевидно, если не знать специфики бизнеса.

Единственный способ оценить предлагаемую структуру базы данных – задать правильные вопросы. В том числе вопросы о том, какие реальные объекты представляют таблицы. Таким образом, оптимизация начинается со сбора требований. Чтобы проиллюстрировать это утверждение, рассмотрим следующий пример: нам нужно хранить учетные записи пользователей, и необходимо хранить телефонные номера каждого владельца записи. На рис. 1.1 и 1.2 показаны два возможных варианта.

Рис. 1.1 ❖ Вариант с одной таблицей

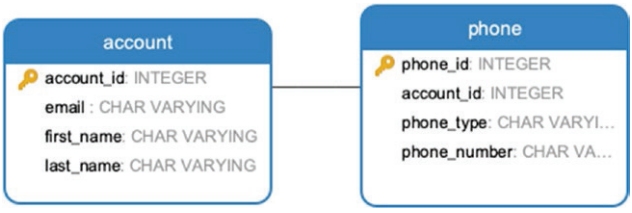


Рис. 1.2 ❖ Вариант с двумя таблицами

Какой из двух вариантов правильный? Это зависит от того, как будут использоваться данные. Если номера телефонов никогда не участвуют в критериях поиска и выбираются как часть учетной записи (для отображения на экране службы поддержки клиентов) или если в пользовательском интерфейсе есть поля, помеченные конкретными типами телефонов, то вариант с одной таблицей более уместен.

Но если мы собираемся искать по номеру телефона независимо от его типа, можно разместить все телефоны в отдельной таблице, и это сделает поиск более производительным.

Кроме того, пользователей часто просят указать, какой номер телефона является основным. В вариант с двумя таблицами легко добавить один логический атрибут `is_primary`, но в варианте с одной таблицей это не так просто. Дополнительные сложности могут возникнуть, если у пользователя нет стационарного или рабочего телефона, а такое случается часто. С другой стороны, у людей бывает несколько сотовых телефонов, или у них может быть виртуальный номер, например Google Voice, и может возникнуть желание записать этот номер в качестве основного, по которому с ними можно связаться. Все эти соображения говорят в пользу варианта с двумя таблицами.

Наконец, мы можем оценить частоту каждого варианта использования и критичность времени отклика в каждом случае.

Разработка приложений и производительность

Мы говорим о разработке приложений, а не только о разработке базы данных, поскольку запросы к базе данных не выполняются сами по себе – они яв-

ляются частью приложений. Традиционно именно оптимизация отдельных запросов рассматривается как просто «оптимизация», но мы будем смотреть на вещи шире.

Довольно часто, хотя каждый запрос к базе данных, выполняемый приложением, возвращает результат менее чем за десятую часть секунды, время отклика страницы приложения может составлять десятки секунд. С технической точки зрения оптимизация таких процессов – это не «оптимизация базы данных» в традиционном понимании, но разработчик базы данных может многое сделать, чтобы улучшить ситуацию. Мы рассмотрим соответствующие методы оптимизации в главах 10 и 13.

Другие этапы жизненного цикла

Жизненный цикл приложения не заканчивается после его выпуска в промышленное окружение, и оптимизация – это тоже непрерывный процесс. Хотя нашей целью должна быть долгосрочная оптимизация, трудно предсказать, как именно будет развиваться система. Полезно постоянно следить за производительностью системы, обращая внимание не только на время выполнения, но и на тенденции.

Запрос может быть очень производительным, и можно не заметить, что время выполнения начало увеличиваться, потому что оно по-прежнему находится в допустимых пределах и никакие автоматические системы мониторинга не выдают предупреждение.

Время выполнения запроса может измениться из-за увеличения объема данных, изменения распределения данных или увеличения частоты выполнения. Кроме того, в каждом новом выпуске PostgreSQL мы ожидаем увидеть новые индексы и другие улучшения, а некоторые из них могут оказаться настолько значительными, что подтолкнут нас к переписыванию исходных запросов.

Какой бы ни была причина изменения, нельзя считать, что какая-либо часть системы будет всегда оставаться оптимизированной.

Особенности PostgreSQL

Хотя принципы, описанные в предыдущем разделе, применимы к любой реляционной базе данных, PostgreSQL, как и любая другая база данных, имеет некоторые особенности, которые нужно учитывать. Если у вас уже есть опыт оптимизации других баз данных, может оказаться, что значительная часть ваших знаний неприменима. Не считайте это недостатком PostgreSQL; просто помните, что PostgreSQL многое делает иначе.

Возможно, самая важная особенность, о которой вам следует знать, – в PostgreSQL нет подсказок оптимизатору. Если вы ранее работали с такой базой данных, как Oracle, в которой есть возможность «подсказать» оптимизатору, то вы можете почувствовать себя беспомощным, столкнувшись с проблемой оптимизации запроса PostgreSQL. Однако есть и хорошие но-

вости: в PostgreSQL намеренно нет подсказок. Ключевая группа PostgreSQL верит в необходимость инвестировать в разработку планировщика запросов, способного выбирать самый подходящий путь выполнения без подсказок. В результате движок оптимизации PostgreSQL является одним из лучших среди как коммерческих систем, так и систем с открытым исходным кодом. Многие сильные разработчики баз данных перешли на Postgres из-за оптимизатора. Кроме того, исходный код Postgres был выбран в качестве основы для нескольких коммерческих баз данных отчасти из-за оптимизатора. В PostgreSQL еще более важно писать инструкции SQL декларативно, позволяя оптимизатору делать свою работу.

Еще одна особенность PostgreSQL, о которой следует знать, – это разница между выполнением параметризованных запросов и динамического SQL. Глава 12 посвящена использованию динамического SQL, которое часто упускают из виду.

В PostgreSQL очень важно знать о новых функциях и возможностях, которые появляются с каждым выпуском. В последнее время ежегодно добавляется более 180 функций, многие из которых связаны с оптимизацией. Мы не планируем рассматривать их все; более того, за тот период времени, который пройдет с момента написания этой главы до ее публикации, несомненно появится еще больше функций. PostgreSQL имеет невероятно богатый набор типов и индексов, и всегда стоит обращаться к последней версии документации, чтобы выяснить, была ли реализована нужная вам функция.

Подробнее об особенностях PostgreSQL мы поговорим позже.

Выводы

Написание запроса к базе данных отличается от написания кода приложения с использованием императивного языка. SQL – декларативный язык, а это означает, что мы указываем желаемый результат, но не указываем путь выполнения. Поскольку два запроса, дающих одинаковый результат, могут выполняться по-разному, используя разные ресурсы и занимая разное время, оптимизация и концепция «мыслить как база данных» являются основными составляющими разработки SQL.

Вместо того чтобы оптимизировать уже написанные запросы, наша цель – правильно писать запросы с самого начала. В идеале оптимизация начинается во время сбора требований и проектирования базы данных. Затем можно приступить к оптимизации отдельных запросов и структурированию вызовов базы данных из приложения. Но оптимизация на этом не заканчивается; чтобы система оставалась работоспособной, необходимо отслеживать производительность на протяжении всего жизненного цикла системы.

Глава 2

Теория: да, она нужна нам!

Чтобы писать эффективные запросы, разработчик базы данных должен понимать, как запросы обрабатываются движком базы данных. А для этого необходимы основы реляционной теории. Если слово «теория» звучит слишком сухо, можно назвать это «тайной жизнью запроса в базе данных». В этой главе мы рассмотрим эту «тайную жизнь», объяснив, что происходит с запросом между моментом, когда вы щелкаете мышью по кнопке **Выполнить** или нажимаете клавишу **Enter**, и моментом, когда вы видите набор результатов, возвращаемый базой данных.

Как обсуждалось в предыдущей главе, SQL-запрос указывает, какие результаты необходимы или что нужно изменить в базе данных, но не указывает, как именно достичь ожидаемых результатов. Работа движка базы данных состоит в том, чтобы преобразовать исходный SQL-запрос в исполняемый код и выполнить его. В этой главе рассматриваются операции, используемые движком базы данных при интерпретации SQL-запроса, и их теоретические основы.

Обзор обработки запросов

Чтобы получить результаты запроса, PostgreSQL выполняет следующие шаги:

- компилирует и преобразует инструкцию SQL в выражение, состоящее из логических операций высокого уровня, называемое *логический план*;
- оптимизирует логический план и превращает его в план выполнения;
- выполняет (интерпретирует) план и возвращает результаты.

Компиляция

Компиляция SQL-запроса аналогична компиляции кода, написанного на императивном языке. Исходный код анализируется, и генерируется внутреннее представление. Но компиляция инструкций SQL имеет два существенных отличия.

Во-первых, в императивном языке в исходный код обычно включаются определения идентификаторов, тогда как определения объектов, на которые ссылаются запросы SQL, в основном хранятся в базе данных. Следовательно, смысл запроса зависит от структуры базы данных: разные серверы баз данных могут интерпретировать один и тот же запрос по-разному.

Во-вторых, вывод компилятора императивного языка обычно представляет собой (почти) исполняемый код, например байт-код для виртуальной машины Java. Напротив, вывод компилятора запросов – это выражение, состоящее из высокоуровневых операций, которые остаются декларативными – они не дают никаких инструкций о том, как получить требуемый результат. На данном этапе указывается возможный порядок операций, но не способ их выполнения.

Оптимизация и выполнение

Инструкции по выполнению появляются на следующем этапе обработки запроса, при оптимизации. Оптимизатор выполняет два вида преобразований: заменяет логические операции на алгоритмы выполнения и, возможно, изменяет структуру логического выражения, меняя порядок выполнения логических операций.

Ни одно из этих преобразований не является простым; логическая операция может вычисляться с использованием разных алгоритмов, и оптимизатор пытается выбрать лучший. Один и тот же запрос может быть представлен несколькими эквивалентными выражениями, дающими один и тот же результат, но требующими существенно разного количества вычислительных ресурсов для выполнения. Оптимизатор пытается найти логический план и физические операции, которые минимизируют необходимые ресурсы, включая время выполнения. Для этого применяются сложные алгоритмы, которые не рассматриваются в данной книге. Однако мы расскажем, как оптимизатор оценивает количество ресурсов, необходимых для физических операций, и как эти ресурсы зависят от особенностей хранения данных.

Результатом работы оптимизатора является выражение, содержащее физические операции. Это выражение называется (физическим) планом выполнения. По данной причине оптимизатор PostgreSQL называют планировщиком запросов.

Наконец, план выполнения запроса интерпретируется движком выполнения запроса, который в сообществе PostgreSQL часто называют исполнителем, и вывод возвращается в клиентское приложение.

Давайте подробнее рассмотрим каждый шаг обработки запроса и используемые операции.

Реляционные, логические и физические операции

Чтобы подробнее выяснить, как движок базы данных понимает SQL, мы должны, наконец, обратиться к главной теме этой главы: теории. Многие

современные системы управления базами данных, включая PostgreSQL, называются реляционными, потому что они основаны на реляционной теории[1]. Несмотря на то что некоторым теория кажется сухой, непонятной или неуместной, понимание небольшой части реляционной теории – а именно реляционных операций – просто необходимо для того, чтобы овладеть оптимизацией. Выражаясь точнее, нам нужно будет понять, как реляционные операции соответствуют логическим операциям и языку, используемому в запросах. В предыдущем разделе были рассмотрены три этапа обработки запросов на высоком уровне; в этом разделе более подробно описывается каждый уровень, начиная с описания реляционных операций.

Некоторые читатели могут счесть представленный здесь материал элементарным и понятным, а другие могут воспринять его как ненужное усложнение. Пока просто верьте, что он закладывает основу для того, что будет дальше.

Реляционные операции

Центральная концепция реляционной теории – *отношение*. Для наших целей мы рассматриваем отношение в виде таблицы, хотя теоретики могут поспорить, что это исключает некоторые тонкие, но важные различия.

Любая реляционная операция принимает одно или несколько отношений в качестве аргументов и порождает еще одно отношение в качестве результата. Это результирующее отношение можно использовать как аргумент другой реляционной операции, порождая следующее отношение, которое, в свою очередь, тоже может стать аргументом. Таким образом, мы можем создавать сложные выражения и представлять сложные запросы.

Возможность построения сложных выражений делает набор реляционных операций (который называют *реляционной алгеброй*) мощным языком запросов.

Кроме того, выражения в реляционной алгебре могут использоваться для определения дополнительных операций.

Первые три операции, которые следует обсудить, – это *фильтрация*, *проекция* и *произведение*.

Фильтрацию (изображенную на рис. 2.1) часто называют селекцией, а в реляционной теории – ограничением. Мы предпочитаем использовать термин *фильтрация*, чтобы избежать путаницы с SQL-командой SELECT, а у термина *ограничение* слишком глубокие математические корни. Операция фильтрации принимает в качестве аргумента одно отношение и включает в результат все кортежи (или строки), удовлетворяющие условию фильтрации, например:

```
SELECT * FROM flight
 WHERE departure_airport = 'LAG'
   AND (arrival_airport = 'ORD' OR arrival_airport = 'MDW')
   AND scheduled_departure BETWEEN '2020-05-27' AND '2020-05-28'
```

---

[1] *К. Дж. Дейт*. Введение в системы баз данных; *Дж. Ульман*. Принципы систем баз данных. 2-е изд.

Рис. 2.1 ❖ Фильтр

Здесь мы начинаем с отношения `flight` и применяем ограничения на значения атрибутов `arrival_airport`, `departure_airport` и `scheduled_departure`. Результат представляет собой множество записей, то есть тоже отношение.

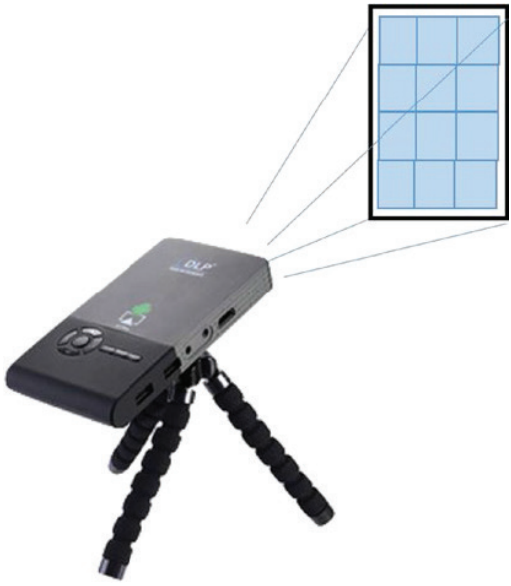


Рис. 2.2 ❖ Проекция

Проекция (представленная на рис. 2.2) также принимает одно отношение в качестве аргумента и удаляет некоторые атрибуты (столбцы). Реляционная проекция удаляет из результата все дубликаты, но проекция SQL этого не делает. Например, запрос

```
SELECT city, zip FROM address
```

при выполнении в PostgreSQL вернет столько строк, сколько записей в таблице `address`. Но если выполнить реляционную проекцию, то для каждого

почтового индекса останется одна запись. Чтобы добиться того же результата в PostgreSQL, нужно добавить к запросу ключевое слово `DISTINCT`:

```
SELECT DISTINCT city, zip FROM address
```

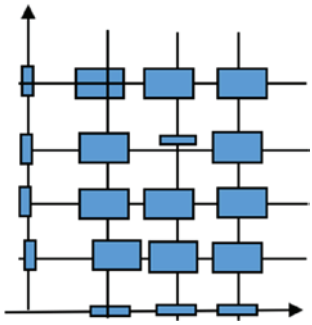


Рис. 2.3 ❖ Произведение

Произведение (также называемое декартовым произведением и изображенное на рис. 2.3) порождает множество всех пар строк из первого и второго аргументов. Очень трудно найти полезный пример произведения из реальной жизни, но давайте представим, что мы ищем все возможные рейсы, которые только могут существовать (из любого аэропорта мира в любой аэропорт). Операция произведения будет выглядеть так:

```
SELECT d.airport_code AS departure_airport,
       a.airport_code AS arrival_airport
  FROM airport a,
       airport d
```

Теперь, когда мы рассмотрели эти основные реляционные операции, вы, наверное, чувствуете себя обманутыми: где же соединения? Мы знаем, что соединения очень важны. Ответ на поверхности: соединение можно выразить как произведение, за которым следует фильтрация. С точки зрения реляционной теории соединение избыточно. Это прекрасный пример того, как работает декларативный язык; формальное определение – один (но не единственный) из способов получить результат соединения. Если мы вычислим декартово произведение двух таблиц, а затем применим фильтрацию, то получим желаемый результат. Но надеемся, что ни один из движков баз данных не делает так для больших наборов данных; на это могли бы уйти годы в буквальном смысле! В главе 3 мы обсудим, как реализовать соединение более эффективно, чем прямое вычисление на основе формального определения.

К реляционным операциям также относятся группировка, объединение, пересечение и разность множеств.

Последний элемент реляционной теории, который нужен для оптимизации, – *правила эквивалентности*. Реляционные операции удовлетворяют некоторым правилам эквивалентности, включая:

- *коммутативность* – `JOIN(R,S) = JOIN (S,R)`.
  Коммутативность означает, что порядок двух отношений не важен. Если у нас есть два отношения, `R` и `S`, то `R JOIN S` даст тот же результат, что и `S JOIN R`;
- *ассоциативность* – `JOIN(R, JOIN(S,T) = JOIN(JOIN(R,S), T)`.
  Ассоциативность означает, что если у нас есть три отношения, `R`, `S` и `T`, мы можем сначала выполнить `R JOIN S`, а затем присоединить `T` к результату, или сначала выполнить `S JOIN T`, а затем присоединить `R` к результату первого соединения, и результаты будут эквивалентны в обоих случаях;
- *дистрибутивность* – `JOIN(R, UNION(S, T)) = UNION(JOIN(R,S), JOIN(R, T))`.
  Дистрибутивность означает, что если мы соединим отношение с объединением двух других отношений, результат будет таким же, как если бы мы выполнили два соединения, `R JOIN S` и `R JOIN T`, по отдельности, а затем объединили результаты.

Правила эквивалентности, перечисленные выше, – всего лишь единичные примеры среди десятков других. Почему важно знать об этих правилах? Может оказаться, что в целях эффективности лучше выполнять операции не в том порядке, в котором они записаны. В последующих главах будет несколько примеров таких преобразований. Эквивалентность гарантирует, что запрос может быть представлен разными выражениями, и это дает оптимизатору простор для деятельности.

Логические операции

Набор логических операций, необходимых для представления SQL-запросов, включает в себя все реляционные операции, но с другой семантикой. Как отмечалось ранее, проекция SQL не удаляет дубликаты; для удаления дубликатов включена отдельная операция.

Другие дополнительные операции необходимы для представления конструкций SQL, которые нельзя выразить в реляционной теории, например левое, правое и полное внешние соединения дают результат, который не является отношением (но является таблицей SQL).

Многие правила эквивалентности также действительны и для логических операций. Для любого относительно сложного запроса оптимизатор может выбрать лучшее из огромного количества выражений.

Более подробную информацию о реляционной теории можно найти в ресурсах, приведенных в заключительных примечаниях.

Запросы как выражения: мыслить множествами

Написание декларативных запросов – непростая задача. Нам более привычны действия, нежели правила или условия. Мышление множествами[1]

[1] Joe Celko, *Joe Celko's Thinking in Sets: Auxiliary, Temporal, and Virtual Tables in SQL* (The

облегчает задачу: мы можем думать о действиях и операциях с таблицами, а не с отдельными объектами (или строками).

Все упомянутые ранее логические операции можно легко выразить на языке SQL. Эти операции принимают таблицы в качестве аргументов: и те, что хранятся в базе данных, и те, что являются результатом предыдущих операций.

Выражение PostgreSQL, записанное как SQL-запрос, будет обработано оптимизатором и, скорее всего, будет заменено другим эквивалентным выражением с использованием обсуждавшихся ранее правил эквивалентности.

Поскольку результатом любой реляционной операции является отношение, ее можно передать непосредственно следующей реляционной операции без необходимости промежуточного хранения. Некоторые разработчики баз данных предпочитают создавать временные таблицы для промежуточных результатов, но такая практика может привести к ненужным вычислительным затратам и помешать оптимизатору.

Говоря языком теории, в предыдущем абзаце говорится, что способность оптимизатора создавать эффективный план выполнения зависит от двух факторов:

- богатый набор эквивалентностей обеспечивает большое пространство эквивалентных выражений;
- реляционные операции не имеют побочных эффектов, таких как временные таблицы, то есть единственное, что они порождают, – это результат операции.

Операции и алгоритмы

Чтобы сделать запрос исполняемым, логические операции необходимо заменить физическими (которые также называют алгоритмами). В PostgreSQL эту замену выполняет планировщик, а общее время выполнения запроса зависит от того, какие выбраны алгоритмы и правильно ли они выбраны.

Когда мы переходим с логического уровня на физический, математические отношения превращаются в таблицы, которые хранятся в базе данных, и нам нужно определить способы получения данных из этих таблиц. Любые сохраненные данные должны извлекаться одним из *алгоритмов доступа к данным*, обсуждаемых в следующей главе. Обычно алгоритмы доступа к данным сочетаются с операциями, которые пользуются их результатами.

Более сложные логические операции, такие как соединение, объединение и группировка, можно реализовать несколькими альтернативными алгоритмами. Иногда сложная логическая операция заменяется несколькими физическими операциями.

Эти алгоритмы подробно обсуждаются в главе 3.

Morgan Kaufmann Series in Data Management Systems).

Выводы

Движок базы данных интерпретирует запросы SQL, преобразовывая их в логический план, трансформируя результаты, выбирая алгоритмы для реализации логического плана и, наконец, выполняя выбранные алгоритмы. Логические операции, используемые движком базы данных, основаны на операциях реляционной теории, и их понимание исключительно важно для умения мыслить как база данных.

Глава 3

Еще больше теории: алгоритмы

К настоящему времени внимательным читателям, не пропустившим ни одной главы, возможно, не терпится. Мы уже перешли к третьей главе – и до сих пор говорим о теории! Когда же мы будем писать код?

Очень скоро! В этой главе рассматривается последняя часть обработки запросов, и в конце у нас будет все необходимое для понимания планов выполнения.

Во второй главе рассказывается о реляционных операциях, и в ней говорится, что для выполнения запросов нужны физические операции, или алгоритмы. Сопоставить эти алгоритмы с логическими операциями непросто; иногда сложная логическая операция заменяется несколькими физическими операциями, или несколько логических операций объединяются в одну физическую.

В этой главе мы описываем эти алгоритмы, начиная с алгоритмов извлечения данных, а затем переходим к алгоритмам для более сложных операций.

Понимание этих алгоритмов позволит нам вернуться к планам выполнения и лучше понять их компоненты. Таким образом, мы будем всего в одном шаге от нашей цели, которая состоит в том, чтобы научиться настраивать запросы.

Стоимостные модели алгоритмов

В первой главе упоминалось несколько способов измерения производительности системы, включая время отклика, стоимость и удовлетворенность пользователей. Эти показатели являются внешними по отношению к базе данных, и хотя внешние показатели наиболее ценны, они не доступны оптимизатору запросов.

Вместо этого оптимизатор использует внутренние показатели, основанные на объеме вычислительных ресурсов, необходимых для выполнения запроса или отдельной физической операции в рамках плана. Наиболее важными

ресурсами являются те, что влияют на время выполнения, а именно циклы процессора и количество операций ввода-вывода (чтение и запись дисковых блоков). Другие ресурсы, такие как память или дисковое пространство, тоже косвенно влияют на время выполнения; например, количество доступной памяти будет влиять на соотношение циклов процессора и количество операций ввода-вывода. Распределение памяти контролируется параметрами сервера и здесь не рассматривается.

Эти два основных показателя, циклы процессора и количество операций ввода-вывода, напрямую не сопоставимы. Однако для сравнения планов выполнения запросов оптимизатор объединяет их в одну функцию стоимости: чем ниже стоимость, тем лучше план.

В течение нескольких десятилетий количество операций ввода-вывода было доминирующим компонентом стоимости, потому что вращающиеся жесткие диски работают на порядки медленнее, чем процессор. Но для современного оборудования это не обязательно так, поэтому оптимизатор должен быть настроен на использование правильного соотношения. Это также контролируется параметрами сервера.

Стоимостная модель физической операции оценивает ресурсы, необходимые для выполнения операции. Как правило, стоимость зависит от таблиц, указанных в качестве аргументов операции. Для представления стоимостных моделей мы будем использовать простые формулы со следующими обозначениями: для любой таблицы или отношения `R` символы `TR` и `BR` обозначают количество строк в таблице и количество дисковых блоков, занимаемых таблицей, соответственно. Дополнительные обозначения будут вводиться по мере необходимости.

В следующем разделе обсуждаются физические операции и рассматриваются алгоритмы и стоимостные модели для каждой из них. Поскольку относительная скорость процессора и внешнего хранилища может варьироваться в широких пределах, затраты на процессор и ввод-вывод рассматриваются отдельно. Мы не будем говорить про логические операции проекции и фильтрации, которые обсуждали в предыдущей главе. Обычно они объединяются с предшествующей им операцией, потому что могут применяться независимо к каждой строке, не завися от других строк в таблице аргумента.

Алгоритмы доступа к данным

Чтобы начать выполнение запроса, движок должен извлечь сохраненные данные. Этот раздел касается алгоритмов, используемых для чтения данных из объектов базы данных. На практике эти операции часто сочетаются с последующей операцией в плане выполнения запроса. Это выгодно в тех случаях, когда можно сэкономить время выполнения, избегая чтения, которое впоследствии будет отфильтровано.

Эффективность таких операций зависит от соотношения количества строк, составляющих результат операции, к общему количеству строк в сохранен-

ной таблице. Такое соотношение называется *селективностью* (*избирательностью)*. Выбор алгоритма для определенной операции чтения зависит от селективности фильтров, которые могут применяться одновременно.

Представление данных

Неудивительно, что данные хранятся в файлах на жестких дисках. Любой файл, используемый для объектов базы данных, делится на блоки одинаковой длины; по умолчанию PostgreSQL использует блоки по 8192 байта каждый. Блок – это единица, которая передается между жестким диском и оперативной памятью, а количество операций ввода-вывода, необходимое для выполнения любого доступа к данным, равно количеству блоков, которые читаются или записываются.

Объекты базы данных состоят из логических элементов (строк таблиц, индексных записей и т. д.). PostgreSQL выделяет место для этих элементов блоками. Несколько мелких элементов могут находиться в одном блоке; более крупные элементы могут быть распределены между несколькими блоками. Общая структура блока показана на рис. 3.1.

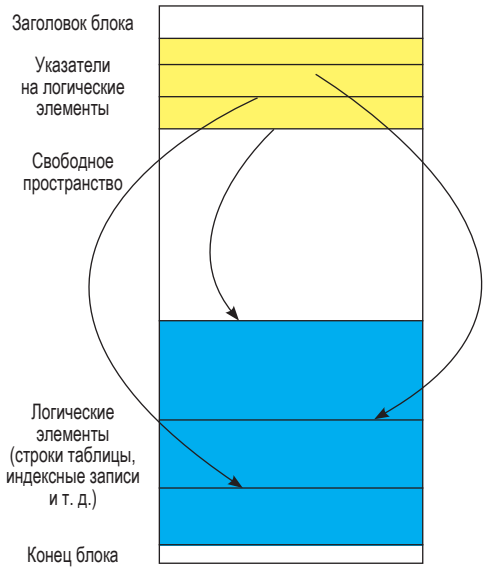


Рис. 3.1 ❖ Общая структура блока в PostgreSQL

Распределение элементов по блокам также зависит от типа объекта базы данных. Строки таблицы хранятся с использованием структуры данных, называемой кучей (heap): строка может быть вставлена в любой блок, в котором достаточно свободного места, без специального упорядочивания. Другие объекты (например, индексы) могут использовать блоки иначе.

Полное (последовательное) сканирование

При полном сканировании движок базы данных последовательно считывает все строки в таблице и для каждой строки проверяет условие фильтрации. Чтобы оценить стоимость этого алгоритма, требуется более подробное описание, показанное псевдокодом в листинге 3.1.

Листинг 3.1 ❖ Псевдокод алгоритма доступа к данным полным сканированием

```
FOR each block IN a_table LOOP
      read block
      FOR each row IN block LOOP
            IF filter_condition (row)
                  THEN output (row)
            END IF
      END LOOP
END LOOP
```

Количество операций ввода-вывода равно `BR`; общее количество итераций внутреннего цикла равно `TR`. Нам также необходимо оценить стоимость операций, порождающих выходные строки. Она зависит от селективности (которая обозначается `S`) и равняется `S * TR`. Собрав все эти части воедино, мы можем вычислить стоимость полного сканирования:

```
c1 * BR + c2 * TR + c3 * S * TR,
```

где константы `c1`, `c2` и `c3` представляют характеристики аппаратного обеспечения.

Полностью просканировать можно любую таблицу; для этого не нужны дополнительные структуры данных. Остальные алгоритмы зависят от наличия индексов в таблице, как описано ниже.

Доступ к таблицам на основе индексов

Обратите внимание, что пока мы не перешли к физическим операциям, мы даже не упоминали алгоритмы доступа к данным. Нам не нужно «читать» отношения – это абстрактные объекты. Если следовать идее того, что отношения отображаются в таблицы, нет другого способа получить данные, кроме как прочитать всю таблицу в оперативную память. Как еще мы узнаем, какие значения содержатся в каких строках? Но реляционные базы данных не были бы таким мощным инструментом обработки данных, если бы на этом мы и остановились. Все реляционные базы данных, включая PostgreSQL, позволяют создавать дополнительные, избыточные структуры, значительно ускоряя доступ к данным по сравнению с простым последовательным чтением.

Эти дополнительные структуры называются индексами.

Как создаются индексы, мы рассмотрим позже; пока нам нужно знать два факта, касающихся индексов. Во-первых, индексы – «избыточные» объекты

базы данных; они не хранят никакой дополнительной информации, которую нельзя найти в исходной таблице.

Во-вторых, индексы предоставляют дополнительные пути доступа к данным; они позволяют определить, какие значения хранятся в строках таблицы, без необходимости чтения самой таблицы – так работает доступ на основе индексов. И как упоминалось ранее, это происходит полностью прозрачно для приложения.

Если условие (или условия) фильтрации поддерживается индексом в таблице, индекс можно использовать для доступа к данным из этой таблицы. Алгоритм извлекает список указателей на блоки, содержащие строки со значениями, удовлетворяющими условию фильтрации, и только эти блоки читаются из таблицы.

Чтобы получить строку таблицы по указателю, необходимо прочитать блок, содержащий эту строку. Основная структура данных таблицы – это *куча*, то есть строки хранятся неупорядоченно. Их порядок не гарантирован и не соответствует свойствам данных. Есть две отдельные физические операции, используемые PostgreSQL для получения строк с помощью индексов: индексное сканирование (`index scan`) и сканирование по битовой карте (`bitmap heap scan`). При индексном сканировании движок базы данных считывает одну за другой все записи индекса, которые удовлетворяют условию фильтрации, и в этом же порядке извлекает блоки. Поскольку базовая таблица представляет собой кучу, несколько записей индекса могут указывать на один и тот же блок. Чтобы избежать многократного чтения одного и того же блока, в PostgreSQL реализована операция сканирования по битовой карте, которая создает битовую карту блоков, содержащих необходимые строки. Потом все строки в этих блоках фильтруются. Преимущество реализации PostgreSQL состоит в том, что она упрощает использование нескольких индексов в одной и той же таблице в одном запросе, применяя логические операторы «и» и «или» к битовым картам блоков, порождаемым каждым индексом.

Стоимостная модель этого алгоритма намного сложнее. Неформально ее можно описать так: при малых значениях селективности, скорее всего, все строки, удовлетворяющие условиям фильтрации, будут располагаться в разных блоках, и, следовательно, стоимость будет пропорциональна количеству возвращаемых строк. Для больших значений селективности количество обрабатываемых блоков приближается к общему количеству блоков. В последнем случае стоимость становится выше, чем стоимость полного сканирования, поскольку для доступа к индексу необходимы ресурсы.

Сканирование только индекса

Операции доступа к данным не обязательно возвращают полные строки. Если некоторые столбцы не нужны для запроса, их можно опустить, как только строка пройдет условия фильтрации (если таковые имеются). Говоря более формально, это означает, что логическая проекция сочетается с доступом к данным. Такое сочетание особенно полезно, если индекс, используемый для фильтрации, содержит все столбцы, необходимые для запроса.

Алгоритм считывает данные из индекса и применяет оставшиеся условия фильтрации, если это необходимо. Обычно не нужно обращаться к табличным данным, но иногда необходимы дополнительные проверки – мы подробно рассмотрим эту тему в главе 5.

Стоимостная модель сканирования только индекса аналогична модели для доступа к таблице на основе индекса, за исключением того, что не нужно обращаться к данным таблицы. Для малых значений селективности стоимость примерно пропорциональна количеству возвращаемых строк. При больших значениях селективности алгоритм выполняет (почти) полный просмотр индекса. Стоимость просмотра индекса обычно ниже, чем стоимость полного просмотра таблицы, потому что индекс содержит меньше данных.

Сравнение алгоритмов доступа к данным

Выбор лучшего алгоритма доступа к данным в основном зависит от селективности запроса. Отношение стоимости к селективности для различных алгоритмов доступа к данным показано на рис. 3.2. Мы намеренно ограничиваемся качественным сравнением; все числа на этом графике опущены, поскольку они зависят от аппаратного обеспечения и размера таблицы.

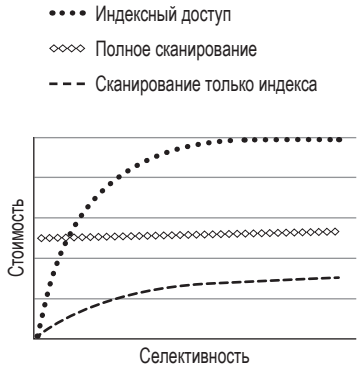


Рис. 3.2 ❖ Связь стоимости и селективности запросов для разных алгоритмов доступа к данным

Линия, соответствующая полному сканированию, прямая и почти горизонтальная: ее рост происходит из-за генерации выходных строк. Как правило, стоимость генерации незначительна по сравнению с другими затратами для этого алгоритма.

Линия, обозначающая стоимость доступа к таблице на основе индекса, начинается (почти) с нуля и быстро растет с ростом селективности. Рост замедляется при больших значениях селективности, где стоимость этого алгоритма значительно выше, чем стоимость полного сканирования.

Самым интересным моментом является пересечение двух линий: для меньших значений селективности предпочтительнее доступ на основе индексов, а полное сканирование лучше подходит для больших значений се-

лективности. Точное положение пересечения зависит от аппаратного обеспечения и может зависеть от размера таблицы. Для относительно медленных вращающихся дисков доступ на основе индексов предпочтительнее, только если селективность не превышает 2–5 %. Для твердотельных накопителей или виртуального окружения это значение может быть выше. На старых вращающихся дисках произвольный доступ к блокам может быть на порядок медленнее последовательного доступа, поэтому дополнительные накладные расходы на индексы при той же пропорции строк получаются выше.

Линия, соответствующая сканированию только индекса, располагается в самой нижней части графика, а это означает, что предпочтительно использовать этот алгоритм, если он применим (то есть все необходимые столбцы находятся в индексе).

Оптимизатор запросов оценивает селективность запроса и селективность, соответствующую точке пересечения для данной таблицы и данного индекса. Условие фильтрации запроса, показанного в листинге 3.2, выбирает значительную часть таблицы.

Листинг 3.2 ❖ Фильтрация по диапазону, выполняемая полным сканированием таблицы

```
SELECT flight_no, departure_airport, arrival_airport
  FROM flight
 WHERE scheduled_departure BETWEEN '2020-05-15' AND '2020-08-31';
```

В данном случае оптимизатор выбирает полное сканирование (*см.* рис. 3.3).

	QUERY PLAN text
1	Seq Scan on flight (cost=0.00..27058.64 rows=275729 width=12)
2	Filter: ((scheduled_departure >= '2020-05-15 00:00:00-05'::timestamp with time zone) AND (scheduled_departure <= '2020-08-...

Рис. 3.3 ❖ Последовательное сканирование

Однако меньший диапазон в том же запросе приводит к доступу к таблице на основе индекса. Запрос показан в листинге 3.3, а его план выполнения – на рис. 3.4.

Листинг 3.3 ❖ Фильтрация по диапазону с доступом к таблице на основе индекса

```
SELECT flight_no, departure_airport, arrival_airport
  FROM flight
 WHERE scheduled_departure BETWEEN '2020-08-12' AND '2020-08-13';
```

На самом деле работа оптимизатора запросов намного сложнее: условия фильтрации могут поддерживаться несколькими индексами с разными значениями селективности. Несколько индексов могут быть объединены для создания битовой карты, уменьшая число блоков, которые нужно просканировать. В результате количество вариантов, доступных оптимизатору, значительно превышает выбор из трех алгоритмов.

	QUERY PLAN text
1	Bitmap Heap Scan on flight (cost=83.27..9166.80 rows=3790 width=12)
2	Recheck Cond: ((scheduled_departure >= '2020-08-12 00:00:00-05'::timestamp w...
3	-> Bitmap Index Scan on flight_scheduled_departure (cost=0.00..82.33 rows=37...
4	Index Cond: ((scheduled_departure >= '2020-08-12 00:00:00-05'::timestamp w...

Рис. 3.4 ❖ Сканирование по битовой карте (доступ на основе индекса)

Таким образом, среди алгоритмов доступа к данным нет победителей и проигравших. Любой алгоритм может выиграть при определенных условиях. Далее, выбор алгоритма зависит от структур хранения и статистических свойств данных. База данных поддерживает для таблиц метаданные, называемые статистикой: информация о кардинальности столбца, разреженности и т. д. Обычно эта статистика неизвестна во время разработки приложения и может изменяться на протяжении жизненного цикла приложения. Следовательно, декларативный характер языка запросов важен для производительности системы. В частности, при изменении статистики таблицы или при корректировке других факторов стоимости для одного и того же запроса может быть выбран другой план выполнения.

Индексные структуры

Этот раздел начинается с абстрактного определения того, какую структуру хранения можно назвать индексом; кратко рассматриваются наиболее распространенные индексные структуры, такие как деревья и хеш-индексы, и затрагиваются некоторые особенности PostgreSQL.

Мы покажем, как оценить масштаб улучшений для разных типов индексов и как распознать случаи, когда использование индекса не дает никаких преимуществ в смысле производительности.

Что такое индекс?

Можно предположить, что любой человек, работающий с базами данных, знает, что такое индекс. Увы, удивительно много людей – разработчиков баз данных и составителей отчетов, а в некоторых случаях даже администраторов баз данных – используют индексы и даже создают их, лишь поверхностно представляя, что это за объекты и какова их структура. Во избежание недоразумений мы начнем с определения того, что мы подразумеваем под индексом.

Существует множество типов индексов, поэтому мы ориентируемся не на структурные свойства, а определяем индекс по способу его использования. Структура данных называется индексом, если:

- это избыточная структура данных;
- она невидима для приложения;
- она предназначена для ускорения выбора данных по определенным критериям.

Избыточность означает, что индекс можно удалить без потери данных и восстановить по информации, хранящейся где-то еще (конечно, в таблицах). Невидимость означает, что приложение не может узнать о наличии индекса или о его отсутствии. Иначе говоря, любой запрос дает те же результаты как с индексом, так и без него. И наконец, индекс создается в надежде (или с уверенностью), что это улучшит производительность конкретного запроса или (что еще лучше!) нескольких запросов.

Улучшение производительности не происходит бесплатно. Поскольку индекс избыточен, он должен обновляться при обновлении данных таблицы. Это приводит к накладным расходам для операций обновления, которыми иногда нельзя пренебречь. В частности, индексы PostgreSQL могут оказывать большое влияние на операцию очистки (`VACUUM`). Однако многие учебники по базам данных переоценивают эти расходы. Современные высокопроизводительные СУБД используют алгоритмы, которые снижают стоимость обновлений индексов, поэтому несколько индексов в таблице – обычное дело.

Хотя индексные структуры могут значительно различаться для разных типов индексов, ускорение всегда достигается за счет быстрой проверки некоторых условий фильтрации, указанных в запросе. Такие условия фильтрации устанавливают определенные ограничения на атрибуты таблицы. На рис. 3.5 показана структура наиболее распространенных индексов.

В правой части рисунка показана таблица, а в левой – индекс, который можно рассматривать как особый вид таблицы. Каждая строка индекса состоит из индексного ключа и указателя на строку таблицы. Значение ключа обычно совпадает со значением атрибута таблицы. В примере на рис. 3.5 в качестве значения используется код аэропорта; следовательно, данный индекс поддерживает поиск по коду аэропорта.

У столбца может быть одно и то же значение в нескольких строках таблицы. Если этот столбец индексирован, индекс должен содержать указатели на все строки, содержащие это значение индексного ключа. В PostgreSQL индекс содержит несколько записей, то есть ключ индекса повторяется для каждого указателя на строку таблицы.

Рисунок 3.5 объясняет, как добраться до соответствующей строки таблицы при обнаружении записи индекса; однако он не объясняет, почему строку индекса можно найти намного быстрее, чем строку таблицы. Действительно, это зависит от того, как структурирован индекс, и именно это мы обсуждаем в следующих подразделах.

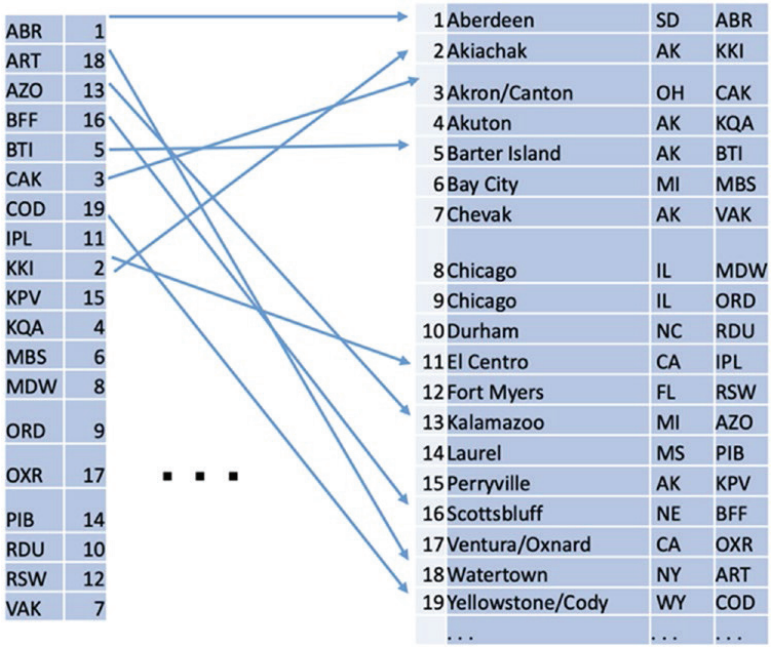


Рис. 3.5 ❖ Индексная структура

B-деревья

Самая распространенная индексная структура – это B-дерево. Структура B-дерева показана на рис. 3.6; коды аэропортов являются индексными ключами. Дерево состоит из иерархически организованных узлов, связанных с блоками, хранящимися на диске.

Листовые узлы (показанные на рис. 3.6 в нижней строке) содержат точно такие же записи индекса, как на рис. 3.5; эти записи состоят из индексного ключа и указателя на строку таблицы.

Нелистовые узлы (расположенные на всех уровнях, кроме нижнего) содержат записи, состоящие из наименьшего ключа (на рис. 3.5 это наименьшее буквенно-цифровое значение) в блоке, расположенном на следующем уровне, и указателя на этот блок. Все записи во всех блоках упорядочены, и в каждом блоке используется не менее половины доступного объема.

Любой поиск ключа `K` начинается с корневого узла B-дерева. Во время поиска по блоку будет найден самый большой ключ `P`, не превышающий `K`, и затем поиск продолжается в блоке, на который ссылается указатель, связанный с `P`. Поиск продолжается, пока мы не дойдем до листового узла, где указатели ссылаются на строки таблицы. Количество просмотренных при поиске узлов равно глубине дерева. Конечно, ключ `K` не обязательно хранится в индексе, но при поиске обнаруживается либо ключ, либо то место, где он мог быть расположен.

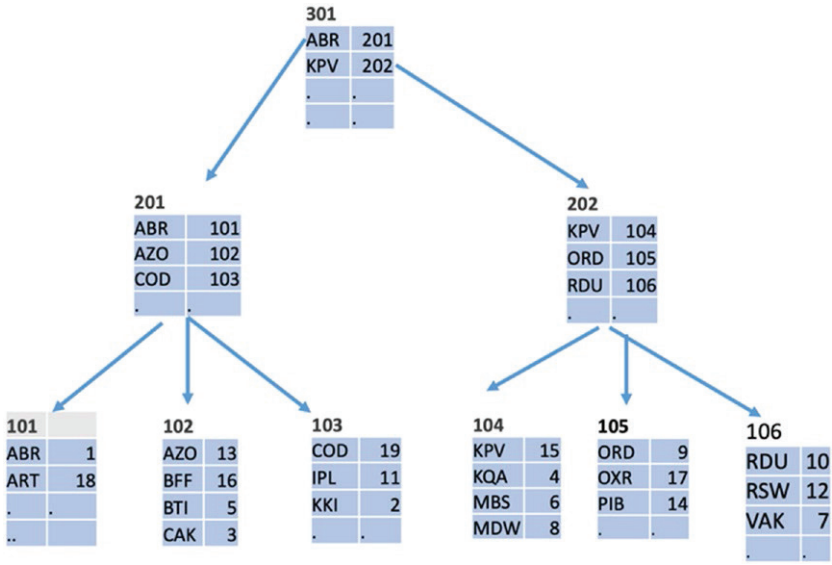


Рис. 3.6 ❖ Пример B-дерева

B-деревья также поддерживают поиск по диапазону (представляемый операцией `between` в SQL). Как только будет найдена нижняя граница диапазона, все индексные ключи диапазона можно получить последовательным сканированием листовых узлов до тех пор, пока не будет достигнута верхняя граница диапазона. Сканирование листовых узлов необходимо также для получения всех указателей, если индекс не является уникальным (то есть значение индексного ключа может соответствовать более чем одной строке).

Почему так часто используются B-деревья?

Из информатики мы знаем, что ни один алгоритм поиска не может найти индексный ключ среди `N` различных ключей быстрее, чем за время `log N` (выраженное в инструкциях процессора). Такая производительность достигается с помощью двоичного поиска по упорядоченному списку или с помощью двоичных деревьев. Однако стоимость обновлений (например, вставки новых ключей) может быть очень высока как для упорядоченных списков, так и для двоичных деревьев: вставка одной записи может привести к полной реструктуризации. Это делает обе структуры непригодными для хранения на диске.

Напротив, B-деревья можно изменять без значительных накладных расходов. Когда вы вставляете запись, реструктуризация ограничена одним блоком. Если емкость блока превышена, то он расщепляется на два блока, и обновление распространяется на верхние уровни. Даже в худшем случае количество измененных блоков не будет превышать глубины дерева.

Чтобы оценить стоимость поиска в B-дереве, нужно вычислить его глубину. Если каждый блок содержит `f` указателей, то количество блоков на каждом уровне в `f` раз больше, чем в предыдущем. Следовательно, глубина дерева, содержащего `N` записей, равна `log N / log f`. Эта формула дает количество обращений к диску, необходимое для поиска по одному ключу. Количество инструкций процессора для каждого блока ограничено, и обычно внутри блока используется двоичный поиск. Следовательно, стоимость ресурсов процессора лишь ненамного хуже теоретически возможного лучшего варианта. Размер блока в PostgreSQL составляет 8 Кбайт. В такой блок помещаются десятки индексных записей; следовательно, индекс с шестью-семью уровнями может вместить миллиарды записей.

В PostgreSQL индекс на основе B-дерева можно создать для любого порядкового типа данных; то есть такого, что из любых двух различных значений этого типа одно значение меньше другого. В эту категорию входят и пользовательские типы.

Битовые карты

Битовая карта – это вспомогательная структура данных, которая используется внутри PostgreSQL с разными целями. Битовые карты можно рассматривать как своего рода индекс: они созданы для облегчения доступа к другим структурам данных, содержащим несколько блоков. Обычно битовые карты используются для компактного представления свойств табличных данных.

Чаще всего битовая карта содержит по одному биту для каждого блока (8192 байта). Значение бита равно 1, если блок обладает нужным свойством, и 0, если нет. На рис. 3.7 показано, как битовые карты используются для доступа к данным по нескольким индексам.

Номер блока	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Индекс 1	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	1
Индекс 2	0	0	1	0	0	1	0	0	0	1	0	0	1	0	0	1
Логическое «и» ⇓																
	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1

Рис. 3.7 ❖ Использование битовых карт для доступа к таблицам по нескольким индексам

Движок базы данных начинает со сканирования двух индексов и построения для каждого из них битовой карты, которая указывает на блоки с подходящими табличными строками. Эти битовые карты показаны на рисунке в строках, обозначенных «Индекс 1» и «Индекс 2». Как только битовые карты будут созданы, движок выполняет побитовую операцию «и», чтобы найти блоки, содержащие подходящие значения для обоих критериев отбора. Наконец, сканируются блоки данных, соответствующие единицам в полученной

битовой карте. Таким образом, к блокам, которые удовлетворяют только одному из двух критериев, не придется обращаться.

Обратите внимание, что подходящие значения могут находиться в разных строках одного блока. Битовая карта гарантирует, что соответствующие строки не будут пропущены, но не гарантирует, что все просканированные блоки содержат подходящие строки.

Битовые карты очень компактны; однако для очень больших таблиц они могут занимать много блоков.

Другие виды индексов

PostgreSQL предлагает множество индексных структур, поддерживающих разные типы данных и разные классы условий поиска.

Хеш-индекс использует хеш-функцию для вычисления адреса индексного блока, содержащего индексный ключ. Для условия равенства этот тип индекса работает быстрее B-дерева, однако он совершенно бесполезен для запросов по диапазону. Стоимостная оценка поиска по хеш-индексу не зависит от размера индекса (в отличие от логарифмической зависимости для B-деревьев).

R-дерево поддерживает поиск по пространственным данным. Индексный ключ для R-дерева всегда представляет собой прямоугольник в многомерном пространстве. Поиск возвращает все объекты, имеющие непустое пересечение с прямоугольником запроса. Структура R-дерева похожа на структуру B-дерева; однако расщепление переполненных узлов устроено намного сложнее. R-деревья эффективны для небольшого числа измерений (обычно от двух до трех).

Другие типы индексов, доступные в PostgreSQL, полезны для полнотекстового поиска, поиска в очень больших таблицах и многого другого. Дополнительная информация по этим темам представлена в главе 14. Любой из этих индексов можно относительно легко настроить для пользовательских типов данных. Однако в этой книге мы не обсуждаем индексы по пользовательским типам.

Сочетание отношений

Настоящая мощь реляционной теории и баз данных SQL основана на сочетании данных из нескольких таблиц.

В этом разделе описываются алгоритмы операций, сочетающие данные, в том числе декартово произведение, соединение, объединение, пересечение и даже группировка. Удивительно, но большинство этих операций можно реализовать с помощью практически идентичных алгоритмов. По этой причине мы обсуждаем алгоритмы, а не операции, которые они реализуют. Мы будем использовать имена `R` и `S` для входных таблиц при описании этих алгоритмов.

Вложенные циклы

Первый алгоритм предназначен для получения декартова произведения, то есть множества всех пар строк из входных таблиц. Самый простой способ вычислить произведение – перебрать строки таблицы R и для каждой строки из R перебрать строки таблицы S. Псевдокод этого простого алгоритма представлен в листинге 3.4, а графическое представление алгоритма показано на рис. 3.8.

Листинг 3.4 ❖ Псевдокод для вложенных циклов

```
FOR row1 IN table1 LOOP
      FOR row2 IN table2 LOOP
            output (row)
      END LOOP
END LOOP
```

Время, необходимое для этого простого алгоритма, пропорционально произведению размеров таблиц ввода: `rows(R)*rows(S)`.

Интересный теоретический факт состоит в том, что любой алгоритм, вычисляющий декартово произведение, не может работать лучше; то есть стоимость любого алгоритма будет пропорциональна произведению размеров его входов или выше. Конечно, некоторые вариации этого алгоритма могут работать лучше, чем другие, но стоимость остается пропорциональной произведению.

Небольшие модификации алгоритма вложенного цикла позволяют вычислить практически любую логическую операцию, сочетающую данные из двух таблиц. Псевдокод из листинга 3.5 реализует соединение.

Листинг 3.5 ❖ Алгоритм вложенного цикла для соединения

```
FOR row1 IN table1 LOOP
      FOR row2 IN table2 LOOP
            IF match(row1,row2) THEN
                  output (row)
            END IF
      END LOOP
END LOOP
```

Обратите внимание, что соединение вложенными циклами является простой реализацией абстрактного определения соединения как декартова произведения, за которым следует фильтр. Поскольку вложенный цикл обрабатывает все пары входных строк, стоимость остается той же, хотя количество выходных строк меньше, чем в случае декартова произведения.

На практике одна или обе входные таблицы являются хранимыми таблицами, а не результатом предыдущей операции. В таком случае алгоритм соединения можно комбинировать с алгоритмом доступа к данным.

Хотя стоимость обработки остается прежней, варианты алгоритма вложенного цикла в сочетании с полным сканированием выполняют вложенные

циклы по блокам входных таблиц и еще один уровень вложенных циклов по строкам, содержащимся в этих блоках. Более сложные алгоритмы минимизируют количество обращений к диску, загружая несколько блоков первой таблицы (внешний цикл) и обрабатывая все строки этих блоков за один проход по таблице S.

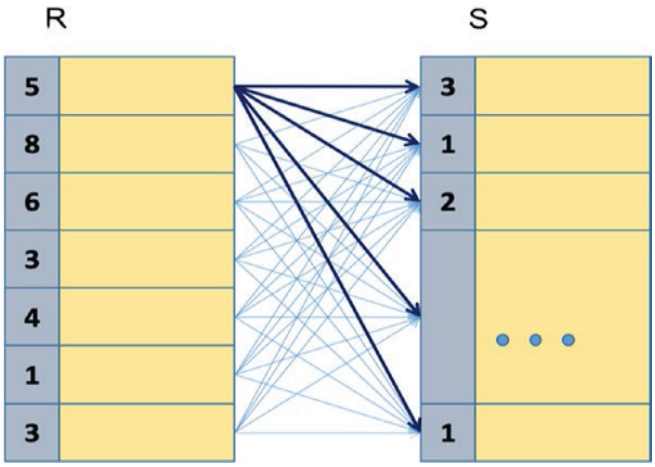


Рис. 3.8 ❖ Алгоритм вложенного цикла

Вышеупомянутые алгоритмы могут работать с любыми условиями соединения. Тем не менее большинство соединений, которые нужны на практике, являются естественными (natural): их условие соединения требует, чтобы некоторые атрибуты таблицы R были равны соответствующим атрибутам таблицы S.

Алгоритм соединения вложенными циклами также можно комбинировать с индексным доступом к данным, если в таблице S есть индекс по атрибутам, используемым в условии соединения. Для естественных соединений внутренний цикл такого алгоритма сокращается до нескольких строк таблицы S для каждой строки таблицы R. Внутренний цикл может даже полностью исчезнуть, если индекс по таблице S уникален, например атрибут соединения таблицы S является его первичным ключом.

Алгоритм вложенного цикла с индексным доступом обычно предпочтителен, если количество строк в таблице R тоже мало. Однако доступ на основе индекса становится неэффективным, когда количество строк, подлежащих обработке, увеличивается, как описано в главе 2.

Можно формально доказать, что для декартова произведения и соединений с произвольными условиями не существует более производительного алгоритма, чем вложенные циклы. Однако важный вопрос заключается в том, имеется ли более подходящий алгоритм для каждого конкретного типа условий соединения. В следующем разделе показано, что для естественных соединений такой алгоритм есть.

Алгоритмы на основе хеширования

Результат естественного соединения состоит из пар строк из таблиц R и S, которые имеют равные значения атрибутов, по которым выполняется соединение. Идея алгоритма соединения хешированием проста: если значения равны, то и хеш-значения тоже равны.

Алгоритм разбивает обе входные таблицы по корзинам в соответствии со значениями хеш-функции, а затем независимо соединяет строки в каждой корзине. Схема этого алгоритма показана на рис. 3.9.

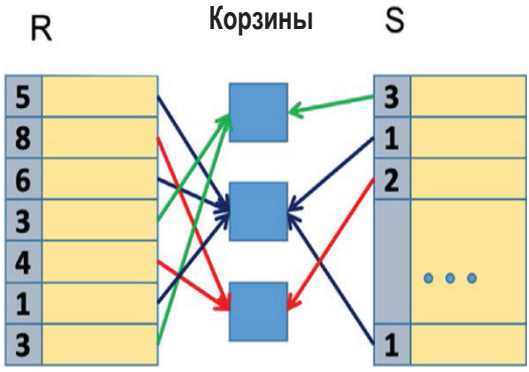


Рис. 3.9 ❖ Алгоритм соединения хешированием

Базовая версия алгоритма соединения хешированием включает две фазы:

1) на этапе *построения* все кортежи таблицы R сохраняются в корзинах согласно значениям хеш-функции;
2) на этапе *проверки* каждая строка таблицы S направляется в соответствующую ей корзину. Если подходящие строки таблицы R находятся в этой корзине, порождаются выходные строки.

Самый простой способ найти подходящие строки в корзине – использовать вложенные циклы (фактически нужно перебирать все строки в корзине для каждой строки таблицы S).

Две фазы алгоритма на основе хеширования показаны как отдельные физические операции в плане выполнения.

Стоимость соединения хешированием приблизительно можно оценить по следующей формуле, где JA – атрибут, по которому выполняется соединение:

```
cost(hash,R,S) = size(R) + size(S) + size(R)*size(S)/size(JA)
```

Первое и второе слагаемые в этой формуле приблизительно равны стоимости одного прохода по всем строкам таблиц R и S. Последнее слагаемое обозначает размер порождаемого результата соединения. Конечно, стоимость вывода одинакова для всех алгоритмов соединения, но нам не нужно было включать его в оценку стоимости алгоритма вложенного цикла, потому что она меньше, чем стоимость вложенных циклов.

Эта формула показывает, что алгоритм на основе хеширования значительно лучше вложенных циклов подходит для больших таблиц и большого количества различных значений атрибута соединения. Например, если атрибут соединения в одной из входных таблиц уникален, то последнее слагаемое всего лишь будет равно размеру другой таблицы.

Базовый алгоритм соединения хешированием работает, если все корзины, созданные на этапе построения хеш-таблицы, помещаются в оперативную память. Другой вариант, называемый гибридным соединением хешированием, соединяет таблицы, которые не могут поместиться в оперативную память. Гибридное соединение хешированием разделяет обе таблицы так, чтобы отдельные разделы помещались в память, а затем выполняет базовый алгоритм для каждой пары соответствующих разделов. Стоимость гибридного соединения хешированием выше, потому что разделы временно хранятся на жестком диске и обе таблицы сканируются дважды. Однако стоимость по-прежнему пропорциональна сумме размеров таблиц, а не их произведению.

Сортировка слиянием

Еще один алгоритм для естественных соединений, который называют сортировкой слиянием, показан на рис. 3.10.

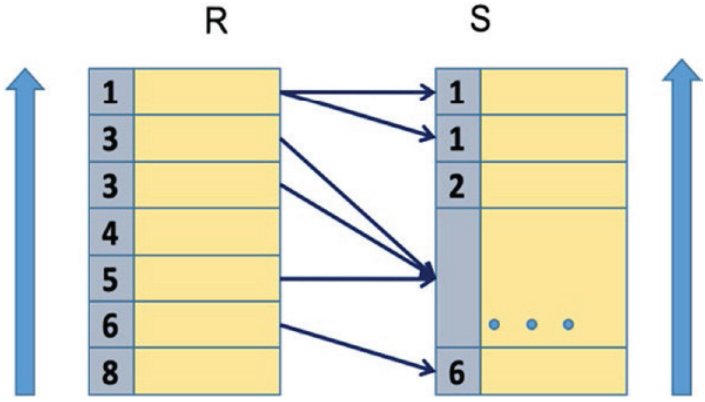


Рис. 3.10 ❖ Алгоритм сортировки слиянием

На первом этапе алгоритма обе входные таблицы сортируются в порядке возрастания по атрибутам соединения.

После того как таблицы правильно упорядочены, на этапе слияния обе таблицы сканируются один раз и для каждого значения атрибута соединения вычисляется декартово произведение строк, содержащих это значение. Обратите внимание, что это произведение является необходимой частью результата соединения. Новые строки с таким же значением атрибута не могут появиться в оставшейся части входных данных, потому что таблицы упорядочены.

Стоимость фазы слияния можно выразить той же формулой, что и для соединения хешированием, то есть она пропорциональна сумме размеров входных и выходных данных. Фактическая стоимость несколько ниже, потому что нет необходимости в этапе построения хеш-таблицы.

Стоимость сортировки можно оценить следующей формулой:

```
size(R)*log(size(R)) + size(S)*log(size(S))
```

Алгоритм сортировки слиянием особенно эффективен, если одна или обе входные таблицы уже отсортированы. Это может произойти в серии соединений по одним и тем же атрибутам.

Сравнение алгоритмов

Как и в случае с алгоритмами доступа к данным, здесь нет заведомых победителей или проигравших. Любой из алгоритмов может оказаться лучше в зависимости от обстоятельств. Алгоритм вложенного цикла более универсален и лучше всего подходит для небольших соединений с использованием индексов; сортировка слиянием и хеширование более эффективны для больших таблиц, если они применимы.

Выводы

Рассмотрев стоимостные модели алгоритмов, алгоритмы доступа к данным, назначение и структуру индексов и алгоритмы для более сложных операций вроде соединения, мы наконец-то набрали достаточно строительных блоков, чтобы взяться за результат работы планировщика запросов – план выполнения.

В следующей главе рассказывается, как читать и понимать планы выполнения и улучшать их.

Глава 4

Планы выполнения

Наконец-то пришло время взглянуть на планы выполнения. Прежде чем мы начнем, вспомним изученные теоретические основы. В главе 3 объясняется, как логические операции сопоставляются с физическим исполнением, и рассказывается об извлечении данных и более сложных операциях.

Понимание этих алгоритмов позволит нам интерпретировать планы выполнения и лучше понять их компоненты.

Собираем все вместе: как оптимизатор создает план выполнения

Результатом работы оптимизатора PostgreSQL является *план выполнения*. В то время как запрос определяет, *что* нужно сделать, план выполнения определяет, *как* выполнять операции SQL.

Задача оптимизатора – построить наилучший возможный физический план, реализующий определенный логический план. Это непростой процесс: иногда сложная логическая операция заменяется несколькими физическими операциями, или несколько логических операций объединяются в одну физическую.

Для построения плана оптимизатор использует *правила преобразования*, *эвристики* и *алгоритмы оптимизации* на основе стоимости. *Правило* преобразует план в другой план с меньшей стоимостью. Например, фильтрация и проекция уменьшают размер набора данных и, следовательно, должны выполняться как можно раньше; правило может переупорядочить операции, чтобы фильтрация и проекция выполнялись раньше. *Алгоритм оптимизации* выбирает план с самой низкой оценкой стоимости. Однако количество возможных планов (*пространство планов*) для запроса, содержащего несколько операций, огромно – оно слишком велико, чтобы алгоритм мог рассмотреть все возможные варианты. Ведь время, потраченное на выбор правильного алгоритма, добавляется к общему времени выполнения запроса. *Эвристики* используются для уменьшения количества планов, рассматриваемых оптимизатором.

Чтение планов выполнения

Перефразируя Элвиса: поменьше абстракций, побольше действия. Мы уже готовы увидеть настоящие планы выполнения. Запрос из листинга 4.1 выбирает все рейсы, которые вылетели из аэропорта Джона Кеннеди (JFK) 14 августа 2020 г. и прибыли в аэропорт О'Хара (ORD). Для каждого рейса рассчитывается общее количество пассажиров.

Листинг 4.1 ❖ Запрос количества пассажиров на конкретных рейсах

```
SELECT f.flight_no,
       f.actual_departure,
       count(passenger_id) passengers
  FROM flight f
       JOIN booking_leg bl ON bl.flight_id = f.flight_id
       JOIN passenger p ON p.booking_id=bl.booking_id
 WHERE f.departure_airport = 'JFK'
   AND f.arrival_airport = 'ORD'
   AND f.actual_departure BETWEEN '2020-08-14' AND '2020-08-15'
GROUP BY f.flight_id, f.actual_departure;
```

Логический план этого запроса показан в листинге 4.2.

Листинг 4.2 ❖ Логический план запроса из листинга 4.1

```
project f.flight_no, f.actual_departure, count(p.passenger_id)[] (
   group [f.flight_no, f.actual_departure] (
      filter [f.departure_airport = 'JFK'] (
         filter [f.arrival_airport = 'ORD'] (
            filter [f.actual_departure >= '2020-08-14'](
               filter [f.actual_departure <= '2020-08-15' ] (
                  join [bl.flight_id = f.flight_id] (
                     access (flights f),
                     join(bl.booking_id=p.booking_id (
                        access (booking_leg bl),
                        access (passenger p)
                     ))))))))
```

Логический план показывает, какие логические операции следует выполнить, но не предоставляет подробной информации о том, как они будут выполняться. Планировщик создает для этого запроса план выполнения, показанный на рис. 4.1.

Чтобы получить план выполнения запроса, выполняется команда `EXPLAIN`. Она принимает любую грамматически правильную инструкцию SQL в качестве параметра и возвращает ее план выполнения.

Мы рекомендуем запускать примеры кода, приведенные в этой книге, и изучать планы выполнения. Однако учтите: выбор правильного плана выполнения – недетерминированный процесс. Планы, создаваемые вашей локальной базой данных, могут слегка отличаться от планов, показанных в этой книге; даже когда планы идентичны, время выполнения может отличаться в зависимости от аппаратного обеспечения и конфигурации.

	QUERY PLAN text
1	GroupAggregate (cost=762870.42..762872.26 rows=1 width=24)
2	Group Key: f.flight_id
3	-> Sort (cost=762870.42..762871.03 rows=244 width=20)
4	Sort Key: f.flight_id
5	-> Hash Join (cost=366897.02..762860.74 rows=244 width=20)
6	Hash Cond: (p.booking_id = bl.booking_id)
7	-> Seq Scan on passenger p (cost=0.00..334784.93 rows=16313693 width=8)
8	-> Hash (cost=366896.70..366896.70 rows=26 width=20)
9	-> Hash Join (cost=9419.14..366896.70 rows=26 width=20)
10	Hash Cond: (bl.flight_id = f.flight_id)
11	-> Seq Scan on booking_leg bl (cost=0.00..310506.66 rows=17893566 width=8)
12	-> Hash (cost=9419.12..9419.12 rows=1 width=16)
13	-> Bitmap Heap Scan on flight f (cost=194.99..9419.12 rows=1 width=16)
14	Recheck Cond: (departure_airport = 'JFK'::bpchar)
15	Filter: ((actual_departure >= '2020-08-14 00:00:00-05'::timestamp with time zone) AND (actual_departure <= '2020-08-15 00:0...
16	-> Bitmap Index Scan on flight_departure_airport (cost=0.00..194.99 rows=10475 width=0)
17	Index Cond: (departure_airport = 'JFK'::bpchar)

Рис. 4.1 ❖ План выполнения

Надеемся, что если вы посмотрите на рис. 4.1, то ценность предыдущих глав станет очевидной – каждая строка плана представляет собой операцию, описанную ранее, поэтому ясно, что происходит за кулисами. Обратите внимание, что, помимо названий алгоритмов, каждая строка плана выполнения включает в себя несколько загадочных цифр в скобках. Все встанет на свои места, если вспомнить главу 3, в которой обсуждалось, как рассчитывается стоимость различных алгоритмов.

А именно план содержит оценки стоимости, ожидаемое количество выходных строк и ожидаемый средний размер («ширину») строк. Все эти значения рассчитываются исходя из статистики базы данных. Значения стоимости включают в себя совокупную стоимость всех предыдущих операций. Для каждой операции есть две оценки стоимости: первая показывает стоимость, необходимую для создания первой строки вывода, в то время как вторая оценивает полную стоимость получения всего результата. Позже в этой главе мы объясним, как оцениваются затраты. Оценки количества и ширины строк вывода необходимы для оценки стоимости вышестоящих операций.

Важно подчеркнуть, что все эти цифры являются приблизительными. Фактические значения, получаемые во время выполнения, могут отличаться от расчетных. Если вы подозреваете, что оптимизатор выбрал план, который не является оптимальным, вам может потребоваться взглянуть на эти оценки. Обычно для хранимых таблиц ошибка небольшая, но она неизбежно увеличивается после каждой следующей операции.

План выполнения представлен в виде дерева физических операций. В этом дереве узлы представляют операции, а стрелки указывают на операнды. На рис. 4.1 может быть непросто разглядеть дерево. Есть несколько инструментов, в том числе pgAdmin, которые могут генерировать графическое пред-

ставление плана выполнения. На рис. 4.2 показано, как это может выглядеть. Фактически на этом рисунке изображен план выполнения для листинга 4.4, который мы обсудим позже в данной главе.

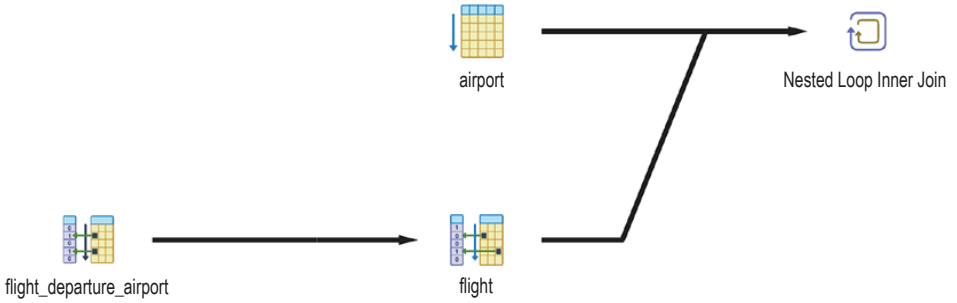


Рис. 4.2 ❖ Графическое представление простого плана выполнения (листинг 4.4)

Для более сложных запросов графическое представление плана выполнения может быть менее полезным – взгляните на представление плана выполнения для листинга 4.1 на рис. 4.3.

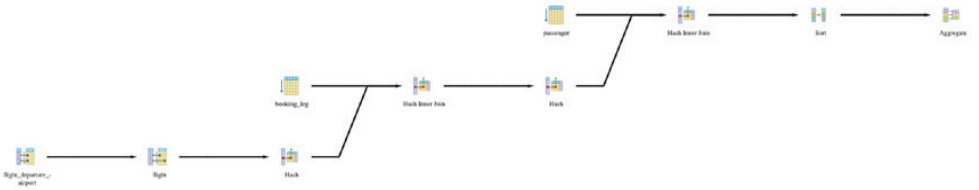


Рис. 4.3 ❖ Графическое представление плана выполнения для листинга 4.1

В таких случаях более компактное текстовое представление, как на рис. 4.4, оказывается полезнее.

Теперь вернемся к фактическому выводу команды `EXPLAIN`, изображенному на рис. 4.1. Он показывает каждый узел дерева в отдельной строке, начинающейся с символа `->`, а глубина узла представлена отступом. Поддеревья помещаются после своего родительского узла. Некоторые операции представлены двумя строками.

Выполнение плана начинается с листьев и заканчивается у корня. Это значит, что операция, которая выполняется первой, находится в строке с наибольшим отступом. Конечно, план может содержать несколько листовых узлов, которые выполняются независимо. Как только операция создает выходную строку, эта строка передается следующей операции. Таким образом, не нужно хранить промежуточные результаты между операциями.

На рис. 4.1 выполнение начинается с последней строки, а таблица `flight` читается с помощью индекса по столбцу `departure_airport`. Поскольку к таблице применено несколько фильтров, а индексом поддерживается только одно из условий фильтрации, PostgreSQL выполняет операцию `index bitmap`

scan (описанную в главе 2). Движок обращается к индексу и составляет битовую карту блоков, которые могут содержать необходимые записи. Затем он считывает фактические блоки из базы данных с помощью операции bitmap heap scan, и для каждой записи, извлеченной из базы данных, повторно проверяет (recheck), что найденные в индексе строки являются текущими, и применяет фильтрацию (filter) для дополнительных условий по столбцам arrival_airport и schedule_departure, для которых у нас нет индексов.

#	Node
1.	→ Aggregate
2.	→ Sort
3.	→ Hash Inner Join Hash Cond: (p.booking_id = bl.booking_id)
4.	→ Seq Scan on passenger as p
5.	→ Hash
6.	→ Hash Inner Join Hash Cond: (bl.flight_id = f.flight_id)
7.	→ Seq Scan on booking_leg as bl
8.	→ Hash
9.	→ Bitmap Heap Scan on flight as f Filter: ((scheduled_departure >= '2020-08-14 00:00:00-05'::timestamp with time zone) AND (scheduled_departure <= '2020-08-15 00:00:00-05'::timestamp with time zone) AND (arrival_airport = 'ORD'::bpchar)) Recheck Cond: (departure_airport = 'JFK'::bpchar)
10.	→ Bitmap Index Scan using flight_departure_airport Index Cond: (departure_airport = 'JFK'::bpchar)

Рис. 4.4 ❖ Альтернативное текстовое представление того же плана выполнения

Результат соединяется с таблицей booking_leg. PostgreSQL использует последовательное сканирование для доступа к этой таблице и алгоритм соединения хешированием по условию bl.flight_id = f.flight_id.

Затем таблица passenger читается последовательным сканированием (поскольку на ней нет никаких индексов), и снова используется алгоритм соединения хешированием по условию p.booking_id = bl.booking_id.

Последняя выполняемая операция – группировка и вычисление агрегатной функции sum. После сортировки выясняется, что критериям поиска удовлетворяет только один рейс. Таким образом, нет необходимости использовать какие-либо алгоритмы группировки, и выполняется подсчет всех пассажиров на найденном рейсе.

В следующем разделе рассматривается, что еще можно почерпнуть из плана выполнения и почему это важно.

Планы выполнения

Часто, когда мы объясняем, как читать планы выполнения, слушателей поражает размер плана даже для относительно простого запроса, а ведь в плане выполнения более сложного запроса могут быть сотни строк. Чтобы прочитать даже такой простой план, как на рис. 4.1, может потребоваться некото-

рое время. Иногда, хотя каждая строчка плана сама по себе понятна, остается вопрос: «Вот у меня медленный запрос, и вы говорите, чтобы я посмотрел план выполнения. Но там сотни строк, и что мне с этим делать? С чего начать?»

Хорошая новость заключается в том, что в большинстве случаев вам не нужно читать весь план, чтобы понять, что именно замедляет выполнение. В этом разделе мы подробнее познакомимся с интерпретацией планов выполнения.

Что происходит во время оптимизации?

Как упоминалось в главе 2, оптимизатор выполняет два вида преобразований: заменяет логические операции соответствующими физическими алгоритмами выполнения и (возможно) изменяет структуру логического выражения, изменяя порядок, в котором выполняются логические операции.

Первый шаг – это переписывание запроса. На этом этапе оптимизатор PostgreSQL улучшает код, раскрывая подзапросы, заменяя представления их текстовым видом и т. д. Важно помнить, что этот шаг происходит всегда. Учебники по SQL, рассказывая о концепции представления, часто говорят, что «представления можно использовать как таблицы», и это вводит в заблуждение. В большинстве случаев представления заменяются их исходным кодом. Тем не менее «в большинстве случаев» не значит «всегда». В главе 7 обсуждаются представления и то, как оптимизатор обрабатывает их, а также связанные с ними потенциальные проблемы производительности.

Следующим шагом после переписывания запроса выполняется то, что мы обычно и называем оптимизацией. Она включает:

- определение возможных порядков операций;
- определение возможных алгоритмов выполнения для каждой операции;
- сравнение стоимости разных планов;
- выбор оптимального плана выполнения.

Многие разработчики SQL предполагают, что PostgreSQL выполняет запросы, обращаясь к таблицам в том порядке, в котором они указаны в предложении `FROM`, и соединяя их в том же самом порядке.

Однако в большинстве случаев порядок соединений *не* сохраняется. В следующих главах мы более подробно обсудим, что влияет на порядок операций, а пока посмотрим, как оценить план выполнения.

Почему планов выполнения так много?

Мы несколько раз отмечали, что одна инструкция SQL может выполняться разными способами, используя разные планы исполнения. На самом деле могут быть сотни, тысячи или даже миллионы возможных способов выполнения одной команды! Эта глава дает представление о том, откуда берутся эти цифры. Планы могут отличаться:

- порядком операций;
- алгоритмами, используемыми для соединений и других операций (такими как вложенные циклы или соединение хешированием);
- методами получения данных (такими как индексный доступ или полное сканирование).

Говоря формально, оптимизатор находит лучший план, вычисляя стоимости всех возможных планов, а затем сравнивая их. Но поскольку мы знаем, что для выполнения каждого соединения есть три основных алгоритма, даже простой запрос из трех таблиц порождает девять возможных планов выполнения; учитывая 12 возможных порядков соединения, это дает 108 возможных планов (3 × 3 × 12 = 108). А если мы учтем все возможные методы извлечения данных для каждой таблицы, то количество планов увеличится до нескольких тысяч.

К счастью, PostgreSQL не проверяет все возможные планы.

Алгоритм оптимизации основан на принципе оптимальности: поддерево оптимального плана оптимально для соответствующего подзапроса. План можно рассматривать как композицию из нескольких составных частей или поддеревьев. Поддерево плана – это план, который включает в себя некоторую операцию исходного плана в качестве корневого узла и все ее дочерние узлы, то есть все операции, которые являются аргументами операции, выбранной в качестве корня поддерева. Оптимизатор строит оптимальный план, начиная с самых маленьких поддеревьев (то есть с доступа к отдельным таблицам), и постепенно создает более сложные поддеревья, включающие больше операций, выполняя на каждом шаге лишь несколько сравнений стоимости. Алгоритм является исчерпывающим: он строит оптимальный план, несмотря на то что значительная часть возможных планов не рассматривается.

Например, в предыдущем примере, как только оптимизатор выберет правильный алгоритм извлечения данных для одной из трех таблиц, он не будет рассматривать планы, которые не используют этот оптимальный алгоритм.

Тем не менее количество рассмотренных планов может быть огромным. Эвристики исключают те части пространства планов, которые вряд ли будут содержать оптимальные планы, уменьшая тем самым количество проверяемых вариантов. Хотя это и помогает оптимизатору быстрее выбирать план выполнения, но может отрицательно сказаться на производительности: существует риск, что самый подходящий план выполнения будет случайно исключен перед сравнением стоимостей.

Хотя эвристики и могут исключить оптимальный план, алгоритм строит лучший план из оставшихся.

Теперь рассмотрим подробнее, как рассчитываются стоимости.

Как рассчитываются стоимости выполнения?

В главе 3 мы обсудили способы измерения производительности алгоритмов баз данных. Мы рассказали о внутренних показателях и установили, что стоимости алгоритмов измеряются количеством операций ввода-вывода и циклов процессора. Теперь мы применим эту теорию на практике.

Стоимость каждого плана выполнения зависит от:

- формул стоимости алгоритмов, используемых в плане;
- статистических данных по таблицам и индексам, включая распределение значений;
- системных настроек (параметров и предпочтений), таких как `join_collapse_limit` или `cpu_index_tuple_cost`.

В главе 3 приведены формулы для расчета стоимости каждого алгоритма. Каждая из них зависит от размера обрабатываемых таблиц, а также от ожидаемого размера результирующего множества. И наконец, пользователи могут изменить стоимость по умолчанию для отдельных операций с помощью системных настроек. Выбором оптимального плана можно неявно управлять, изменяя параметры оптимизатора, которые используются при оценке стоимости. Таким образом, при расчете стоимости выполнения планов учитываются все три пункта.

Это нелогично; часто у разработчиков SQL есть подсознательные ожидания, что существует один «наилучший план» и, более того, он одинаков для всех «похожих» запросов. Однако из-за перечисленных факторов оптимизатор может создавать разные планы выполнения для почти идентичных SQL-запросов или даже для одного и того же запроса. Как такое может быть? Оптимизатор выбирает план с лучшей оценкой стоимости. Однако может быть несколько планов с почти одинаковыми стоимостями. Оценка стоимости зависит от статистики базы данных, собранной на основе случайной выборки. Статистика, собранная вчера, может незначительно отличаться от сегодняшней статистики. Из-за этих небольших изменений план, который был лучшим вчера, сегодня может занять второе место. Конечно, статистика также может измениться в результате операций вставки, обновления и удаления.

Рассмотрим несколько примеров. В листингах 4.3 и 4.4 представлены два запроса, которые почти идентичны. Единственная разница состоит в значении, по которому происходит фильтрация. Однако планы выполнения, представленные на рис. 4.5 и 4.6, заметно отличаются.

	QUERY PLAN text
1	Hash Join (cost=20.09..25467.60 rows=144636 width=12)
2	Hash Cond: (f.departure_airport = a.airport_code)
3	-> Seq Scan on flight f (cost=0.00..23642.76 rows=683176 width=16)
4	-> Hash (cost=18.33..18.33 rows=141 width=4)
5	-> Seq Scan on airport a (cost=0.00..18.33 rows=141 width=4)
6	Filter: (iso_country = 'US'::text)

Рис. 4.5 ❖ План выполнения для листинга 4.3

Чем вызвана эта разница? Рисунок 4.7 дает подсказку: первый запрос выбирает значительную часть всех аэропортов, и использование индекса не улучшит производительность. Второй запрос, напротив, выберет только один аэропорт, и в этом случае доступ на основе индекса будет более эффективным.

Листинг 4.3 ❖ Простой запрос с одним условием

```
SELECT flight_id, scheduled_departure
  FROM flight f
       JOIN airport a ON departure_airport = airport_code
                      AND iso_country = 'US'
```

Листинг 4.4 ❖ Тот же запрос, что и в листинге 4.3, но с другим значением

```
SELECT flight_id, scheduled_departure
  FROM flight f
       JOIN airport a ON departure_airport = airport_code
                      AND iso_country = 'CZ'
```

	QUERY PLAN text
1	Nested Loop (cost=40.40..3347.29 rows=1026 width=12)
2	-> Seq Scan on airport a (cost=0.00..18.33 rows=1 width=4)
3	Filter: (iso_country = 'CZ'::text)
4	-> Bitmap Heap Scan on flight f (cost=40.40..3318.67 rows=1029 width=16)
5	Recheck Cond: (departure_airport = a.airport_code)
6	-> Bitmap Index Scan on flight_departure_airport (cost=0.00..40.14 rows=1029 width=0)
7	Index Cond: (departure_airport = a.airport_code)

Рис. 4.6 ❖ План выполнения для листинга 4.4

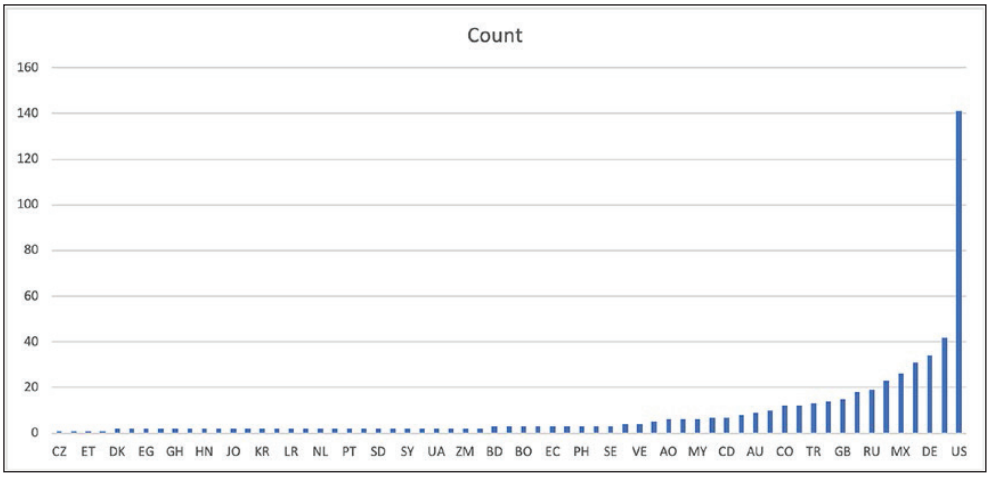


Рис. 4.7 ❖ Гистограмма распределения значений

Почему оптимизатор может ошибаться?

Но как мы можем быть уверены, что выбранный оптимизатором план действительно наилучший? И можно ли вообще найти наилучший план вы-

полнения? Мы потратили довольно много времени на объяснение того, что оптимизатор выполняет свою работу наилучшим образом, если мы не будем ему мешать. Но если это так, то о чем же остальная часть книги? Реальность такова, что ни один оптимизатор не идеален, даже планировщик запросов PostgreSQL.

Во-первых, хотя алгоритм оптимизации математически верен, то есть находит план с лучшей оценкой стоимости, эти оценки по сути своей неточны. Простые формулы, объясненные в главе 3, действительны только для равномерного распределения данных, но равномерное распределение редко встречается в реальных базах данных. На самом деле оптимизаторы используют более сложные формулы, но они также являются несовершенными приближениями к реальности. Как сказал Джордж Бокс: «Все модели неверны, но некоторые из них полезны».

Во-вторых, системы баз данных, включая PostgreSQL, собирают подробную статистику хранимых данных (обычно в виде гистограмм). Гистограммы значительно улучшают оценки селективности. Но, к сожалению, их нельзя использовать для промежуточных результатов. Ошибки при оценке промежуточных результатов – основная причина, из-за которой у оптимизатора может не получиться построить оптимальный план.

В-третьих, оптимальный план может быть исключен эвристиками, или запрос может быть слишком сложен для точного алгоритма оптимизации. В последнем случае используются алгоритмы приблизительной оптимизации.

Во всех этих ситуациях требуется вмешательство человека, и книга как раз об этом. Теперь, когда мы знаем, что происходит во время оптимизации, мы можем исправить проблему, если что-то работает не так, как надо.

Несмотря на все описанные возможности ошибиться, в большинстве случаев оптимизаторы работают хорошо. Однако мы наблюдаем за поведением всей системы и, следовательно, нам доступно больше информации, чем оптимизатору. Мы можем использовать эти дополнительные знания, чтобы помочь оптимизатору работать еще лучше.

Выводы

В этой главе мы рассмотрели планы выполнения: как они создаются, как их читать и понимать. Мы также узнали об оптимизации на основе стоимости и факторах, которые влияют на стоимость планов выполнения.

Хотя стоимостные оптимизаторы обычно хорошо справляются со своей задачей, иногда им нужна помощь, и теперь у нас есть все необходимое, чтобы ее предоставить. В следующих главах будет рассмотрено несколько примеров запросов, требующих вмешательства человека для повышения производительности.

Глава 5

Короткие запросы и индексы

В главе 4 подробно рассказывается о планах выполнения. Теперь перейдем к тому, что делать после того, как команда `EXPLAIN` показала план выполнения. С чего начать, если наша цель – улучшить план выполнения запроса?

Первым делом надо определить, является запрос коротким или длинным. Эта глава посвящена оптимизации коротких запросов. Вы научитесь различать короткие запросы, узнаете, какой метод оптимизации использовать и почему индексы критически важны для этого типа запросов. Мы также обсудим различные виды индексов, которые есть в PostgreSQL, и в каких ситуациях они применимы.

Прежде чем продолжить, создадим несколько дополнительных индексов:

```
SET search_path TO postgres_air;
CREATE INDEX flight_arrival_airport       ON flight (arrival_airport);
CREATE INDEX booking_leg_flight_id        ON booking_leg (flight_id);
CREATE INDEX flight_actual_departure      ON flight (actual_departure);
CREATE INDEX boarding_pass_booking_leg_id ON boarding_pass (booking_leg_id);
```

Какие запросы считаются короткими?

Термин *короткий запрос* уже появлялся несколько раз без формального определения. Что такое короткий запрос? Во-первых, тип запроса не имеет ничего общего с длиной SQL-запроса. Взгляните на два запроса, показанных в листингах 5.1 и 5.2. Листинг 5.1 содержит всего четыре строки кода, но в нем представлен *длинный запрос*. Листинг 5.2 содержит гораздо больше строк, но это короткий запрос.

Листинг 5.1 ❖ Пример длинного запроса

```
SELECT d.airport_code AS departure_airport,
       a.airport_code AS arrival_airport
  FROM airport a,
       airport d
```

Листинг 5.2 ❖ Пример короткого запроса

```
SELECT f.flight_no,
       f.scheduled_departure,
       boarding_time,
       p.last_name,
       p.first_name,
       bp.update_ts as pass_issued,
       ff.level
  FROM flight f
       JOIN booking_leg bl ON bl.flight_id = f.flight_id
       JOIN passenger p ON p.booking_id = bl.booking_id
       JOIN account a ON a.account_id = p.account_id
       JOIN boarding_pass bp ON bp.passenger_id = p.passenger_id
       LEFT OUTER JOIN frequent_flyer ff ON ff.frequent_flyer_id = a.frequent_flyer_id
 WHERE f.departure_airport = 'JFK'
   AND f.arrival_airport = 'ORD'
   AND f.scheduled_departure BETWEEN '2020-08-05' AND '2020-08-07'
```

Во-вторых, тип запроса не определяется размером результирующего множества. Запрос из листинга 5.3 выдает только одну строку; однако это длинный запрос.

Листинг 5.3 ❖ Длинный запрос, выводящий одну строку

```
SELECT avg(flight_length),
       avg (passengers)
  FROM ( SELECT flight_no,
                scheduled_arrival - scheduled_departure AS flight_length,
                count(passenger_id) passengers
           FROM flight f
                JOIN booking_leg bl ON bl.flight_id = f.flight_id
                JOIN passenger p ON p.booking_id = bl.booking_id
          GROUP BY 1,2 ) a
```

Так что же такое короткий запрос?

Запрос является коротким, когда количество строк, необходимых для получения результата, невелико независимо от того, насколько велики задействованные таблицы. Короткие запросы могут считывать все строки из маленьких таблиц, но лишь небольшой процент строк из больших таблиц.

Насколько мал «небольшой процент»? Неудивительно, что это зависит от параметров системы, специфики приложения, фактических размеров таблиц и, возможно, других факторов. Во всяком случае, обычно менее 10 %. Позже в данной главе мы покажем, как определить эту границу.

Напротив, результат длинного запроса зависит от значительной части строк в большой таблице или нескольких таблицах.

Наша таксономия запросов похожа на общепринятое различие между запросами OLTP и OLAP. Все запросы OLTP короткие. Однако многим современ-

ным приложениям требуются запросы, возвращающие сотни строк, и они тоже могут быть короткими.

Почему листинг 5.1 представляет длинный запрос? Потому что для получения результата требуются все строки в таблице аэропортов. Почему запрос из листинга 5.2 – короткий? Потому что ему нужны данные всего по паре рейсов из примерно 200 000. Почему запрос из листинга 5.3 не является коротким? Потому что для расчета результатов необходимы данные каждого бронирования в системе.

Когда мы оптимизируем короткий запрос, мы знаем, что в конечном итоге мы выбираем относительно небольшое количество записей. Это означает, что цель оптимизации – уменьшить размер результирующего множества как можно раньше. Если на первых этапах выполнения запроса применяется самый строгий критерий фильтрации, дальнейшие сортировки, группировки и даже соединения будут менее затратными. В плане выполнения не должно быть сканирований больших таблиц. Однако для небольших таблиц полное сканирование может применяться, как показано на рис. 3.2 в главе 3.

Выбор критериев фильтрации

Может показаться несложным убедиться, что в первую очередь применяются самые строгие критерии фильтрации; однако не всегда все так однозначно. Признаемся: эта глава называется «Короткие запросы и индексы» не просто так: нельзя быстро выбрать подмножество записей из таблицы, если нет индекса, поддерживающего соответствующий поиск. Вот почему для ускорения коротких запросов требуются индексы. Если для поддержки сильно ограничительного условия нет индекса, его, по всей видимости, нужно создать.

Селективность индексов

В главе 3 была представлена концепция *селективности запроса*. Ту же концепцию можно применить и к индексам: чем меньшее количество записей соответствует одному значению индекса, тем ниже значение его селективности. Нам не нужно создавать индексы с высокой селективностью; как было показано в главе 3, для извлечения данных на основе индекса в этом случае потребуется *больше* времени, чем при последовательном сканировании. Поскольку оптимизатор PostgreSQL заранее определяет стоимость каждого метода доступа, этот индекс никогда не будет использоваться, и производительность не пострадает. Однако нежелательно добавлять объект базы данных, который требует места для хранения и дополнительное время для обновления, но не дает никаких преимуществ.

В таблице базы данных может быть несколько индексов по разным столбцам, каждый со своей селективностью. Лучшая производительность для короткого запроса достигается, когда используются наиболее ограничительные индексы (то есть индексы с самой низкой селективностью).

Посмотрим на запрос из листинга 5.4. Можете ли вы сказать, какой критерий фильтрации самый ограничительный?

Листинг 5.4 ❖ Селективность индексов

```
SELECT * FROM flight
 WHERE departure_airport = 'LAX'
   AND update_ts BETWEEN '2020-08-16' AND '2020-08-18'
   AND status = 'Delayed'
   AND scheduled_departure BETWEEN '2020-08-16' AND '2020-08-18'
```

Статус задержанного рейса (Delayed) может быть самым ограничительным, поскольку в идеале в любой день гораздо больше рейсов, вылетающих по расписанию.

В нашей учебной базе данных расписание рейсов составлено на шесть месяцев, поэтому диапазон из двух дней, возможно, не сильно ограничит выборку. С другой стороны, обычно расписание рейсов публикуется заранее, так что рейсы, обновленные непосредственно перед датой запланированного вылета, скорее всего, задержены или отменены.

Еще один фактор, который можно принять во внимание, – популярность аэропорта. Международный аэропорт Лос-Анджелес (LAX) является популярным, и для листинга 5.1 ограничение по `update_ts` будет более ограничительным, чем по `departure_airport`. Однако если мы изменим фильтрацию по `departure_airport` на аэропорт Фукуока (FUK), критерий аэропорта будет более ограничительным, чем выбор на основе `update_ts`.

Если все критерии поиска проиндексированы, поводов для беспокойства нет; скоро мы рассмотрим возможность совместной работы нескольких индексов. Но если самый строгий критерий не проиндексирован, то план выполнения может быть неоптимальным и, вероятно, потребуется дополнительный индекс.

Уникальные индексы и ограничения

Чем лучше (ниже) селективность индекса, тем быстрее выполняется поиск. Таким образом, наиболее эффективные индексы – уникальные.

Индекс *уникален*, если каждому индексированному значению соответствует ровно одна строка в таблице.

Есть несколько разных способов создать уникальный индекс. Во-первых, PostgreSQL автоматически создает уникальный индекс для поддержки каждого первичного ключа или ограничения уникальности в таблице.

В чем разница между первичным ключом и уникальным ограничением? Среди разработчиков SQL распространено заблуждение, согласно которому первичный ключ должен быть возрастающим числовым значением и каждая таблица «должна» иметь первичный ключ. Хотя часто бывает полезно

иметь числовой первичный ключ с автоувеличением (*суррогатный ключ*), первичный ключ не обязан быть числовым и, более того, не обязан определяться только для одного атрибута. Можно определить первичный ключ и как комбинацию нескольких атрибутов; он просто должен удовлетворять двум условиям: комбинация должна быть уникальна (`UNIQUE`) и определена (`NOT NULL`) для всех участвующих атрибутов. Отличие уникальных ограничений в том, что они допускают неопределенные значения.

Таблица может иметь один первичный ключ (хотя это и не обязательно) и сколько угодно ограничений уникальности. Любое ограничение уникальности, не допускающее неопределенных значений, может быть выбрано в качестве первичного ключа для таблицы; таким образом, нельзя формально определить лучшего кандидата на роль первичного ключа. Например, у таблицы `booking` есть первичный ключ `booking_id` и уникальный ключ `booking_ref` – см. листинг 5.5.

Листинг 5.5 ❖ Первичный ключ и ограничение уникальности

```
ALTER TABLE booking
    ADD CONSTRAINT booking_pkey PRIMARY KEY (booking_id);

ALTER TABLE booking
    ADD CONSTRAINT booking_booking_ref_key UNIQUE (booking_ref);
```

Поскольку атрибут `booking_ref` не допускает неопределенных значений, в качестве первичного ключа можно выбрать либо `booking_id`, либо `booking_ref`.

Как показано на ER-диаграмме в главе 1, столбец `frequency_flyer_id` в таблице `account` может имеет неопределенные значения, а также является уникальным:

```
ALTER TABLE account
    ADD CONSTRAINT account_freq_flyer_unq_key UNIQUE (frequent_flyer_id);
```

Уникальный индекс можно создать и без формального определения ограничения уникальности. Нужно только добавить ключевое слово `unique`:

```
CREATE UNIQUE INDEX account_freq_flyer ON account (frequent_flyer_id);
```

Если мы создаем такой индекс уже после того, как данные вставлены в таблицу, команда `CREATE UNIQUE INDEX` проверит уникальность значений, и если будут обнаружены какие-либо дубликаты, индекс не будет создан. Для любых последующих вставок и обновлений уникальность новых значений также будет проверяться.

А как насчет внешних ключей? Создают ли они автоматически какие-либо индексы? Существует распространенное заблуждение, что наличие внешнего ключа обязательно подразумевает наличие индекса в дочерней таблице. Это не так.

Внешний ключ – это ограничение ссылочной целостности; таблица, в которой определен внешний ключ, называется дочерней, а таблица, на которую он ссылается, – родительской. Ограничение внешнего ключа гарантирует,

что для каждого определенного (не NULL) значения из дочерней таблицы в родительской таблице есть совпадающее с ним уникальное значение.

Например, для таблицы flight существует ограничение внешнего ключа, которое гарантирует, что каждый аэропорт прибытия соответствует существующему коду аэропорта:

```
ALTER TABLE flight
    ADD CONSTRAINT arrival_airport_fk FOREIGN KEY (arrival_airport)
    REFERENCES airport (airport_code);
```

Данное ограничение не создает индекс автоматически; если поиск по аэропорту прибытия работает медленно, индекс должен быть создан явно:

```
CREATE INDEX flight_arrival_airport ON flight (arrival_airport);
```

В главе 3 упоминалось, что уникальные индексы делают вложенные циклы эффективными. Если вы обратитесь к рис. 3.7, то поймете, что происходит при наличии индекса.

Алгоритм соединения вложенным циклом также можно сочетать с доступом к данным на основе индекса, если у таблицы S есть индекс по атрибутам, используемым в условии соединения. Для естественных соединений внутренний цикл алгоритма с индексным доступом сокращается до перебора нескольких строк таблицы S для каждой строки из таблицы R. Внутренний цикл может даже полностью исчезнуть, если индекс по таблице S уникален, например если атрибут соединения является первичным ключом.

Часто это истолковывается неверно, и считается, что вложенные циклы всегда эффективны, когда соединение выполняется по связке первичного и внешнего ключей. Однако, как упоминалось ранее, это верно, только если столбец в дочерней таблице – внешний ключ – проиндексирован.

Всегда ли следует создавать индекс по столбцу с ограничением внешнего ключа? Не всегда. Это имеет смысл только в том случае, если количество различных значений достаточно большое. Помните, что индексы с высокой селективностью вряд ли пригодятся. Например, у таблицы flight есть ограничение внешнего ключа по aircraft_code_id:

```
ALTER TABLE flight
    ADD CONSTRAINT aircraft_code_fk FOREIGN KEY (aircraft_code)
    REFERENCES aircraft (code);
```

Это ограничение внешнего ключа необходимо, потому что каждому рейсу должен быть назначен существующий самолет. Чтобы поддерживать ограничение внешнего ключа, в таблицу aircraft был добавлен первичный ключ. Однако в этой таблице всего 12 строк. Следовательно, нет необходимости создавать индекс по столбцу aircraft_code таблицы flight. У этого столбца только 12 различных значений, поэтому индекс по этому столбцу использоваться не будет.

Чтобы проиллюстрировать это утверждение, посмотрим на запрос из листинга 5.6. Этот запрос выбирает все рейсы между аэропортами Джона Кен-

неди (JFK) и О'Хара (ORD) с 14 по 16 августа 2020 г. Для каждого рейса мы выбираем его номер, дату вылета, модель самолета и количество пассажиров.

Листинг 5.6 ❖ Соединение по первичному и внешнему ключам без индекса

```
SELECT f.flight_no,
       f.scheduled_departure,
       model,
       count(passenger_id) passengers
  FROM flight f
       JOIN booking_leg bl ON bl.flight_id = f.flight_id
       JOIN passenger p ON p.booking_id=bl.booking_id
       JOIN aircraft ac ON ac.code=f.aircraft_code
 WHERE f.departure_airport = 'JFK'
   AND f.arrival_airport = 'ORD'
   AND f.scheduled_departure BETWEEN '2020-08-14' AND '2020-08-16'
 GROUP BY 1,2,3
```

План выполнения этого запроса показан на рис. 5.1, и он довольно внушительный.

	QUERY PLAN text
1	GroupAggregate (cost=398855.05..398885.50 rows=1200 width=52)
2	Group Key: f.flight_no, f.scheduled_departure, ac.model
3	-> Sort (cost=398855.05..398858.74 rows=1476 width=48)
4	Sort Key: f.flight_no, f.scheduled_departure, ac.model
5	-> Hash Join (cost=2801.33..398777.36 rows=1476 width=48)
6	Hash Cond: (p.booking_id = bl.booking_id)
7	-> Seq Scan on passenger p (cost=0.00..334784.93 rows=16313693 width=8)
8	-> Hash (cost=2799.37..2799.37 rows=157 width=48)
9	-> Nested Loop (cost=335.03..2799.37 rows=157 width=48)
10	-> Hash Join (cost=334.46..704.08 rows=6 width=48)
11	Hash Cond: ((f.aircraft_code)::text = ac.code)
12	-> Bitmap Heap Scan on flight f (cost=333.19..702.53 rows=100 width=20)
13	Recheck Cond: ((scheduled_departure >= '2020-08-14 00:00:00-05'::timestamp with time zone) AND (scheduled_departure <= '2020-08-16 00:00:00-05'::timestamp with time zon
14	-> BitmapAnd (cost=333.19..333.19 rows=100 width=0)
15	-> Bitmap Index Scan on flight_scheduled_departure (cost=0.00..137.91 rows=6548 width=0)
16	Index Cond: ((scheduled_departure >= '2020-08-14 00:00:00-05'::timestamp with time zone) AND (scheduled_departure <= '2020-08-16 00:00:00-05'::timestamp with time
17	-> Bitmap Index Scan on flight_departure_airport (cost=0.00..194.99 rows=10475 width=0)
18	Index Cond: (departure_airport = 'JFK'::bpchar)
19	-> Hash (cost=1.12..1.12 rows=12 width=64)
20	-> Seq Scan on aircraft ac (cost=0.00..1.12 rows=12 width=64)
21	-> Index Scan using booking_leg_flight_id on booking_leg bl (cost=0.56..348.33 rows=89 width=8)
22	Index Cond: (flight_id = f.flight_id)

Рис. 5.1 ❖ План с последовательным сканированием небольшой таблицы

Вот единственная часть этого плана, которая нас сейчас интересует:

```
Hash  (cost=1.12..1.12 rows=12 width=64)
   -> Seq Scan on aircraft ac  (cost=0.00..1.12 rows=12
```

Оптимизатор PostgreSQL обращается к статистике и может определить, что размер таблицы `aircraft` мал, и индексный доступ не будет эффективным.

Индексы и неравенства

В главе 3 описана структура B-деревьев, рассказывается, как они устроены и как используются для поиска. Далее следует демонстрация их применения на практике.

Предыдущий раздел относится к простым B-деревьям. Как было отмечено в главе 3, они могут поддерживать поиск по отношениям равенства, «больше», «меньше» и `between`: все это виды поиска, требующие сравнения и упорядочения. Большинство поисковых условий в OLTP-системах попадают в эту категорию, но есть и нетривиальные случаи, когда критерии поиска более сложные.

Индексы и преобразования столбцов

Преобразование столбца происходит, когда в критерии поиска участвует не сам столбец, а как-либо измененное его значение. Например, `lower(last_name)` (преобразование фамилии в нижний регистр) и `update_ts::date` (приведение временной метки с часовым поясом к дате) являются преобразованиями столбцов.

Как преобразования столбцов влияют на использование индекса? Попросту говоря, B-деревья по преобразованному атрибуту использовать нельзя. Вы помните из главы 3, как строится B-дерево и как выполняется поиск по нему: в каждом узле значение атрибута сравнивается со значением в узле. Преобразованное значение нигде не записывается, поэтому его не с чем сравнивать. Пусть имеется индекс по фамилии:

```
CREATE INDEX account_last_name ON account (last_name);
```

Следующий запрос не сможет воспользоваться этим индексом:

```
SELECT * FROM account WHERE lower(last_name) = 'daniels';
```

Как решить эту проблему? Поиск такого типа может потребоваться, потому что пассажиры могут вводить свои фамилии в разных регистрах. Если вы полагаете, что достаточно охватить наиболее распространенные случаи, то можно изменить критерий поиска следующим образом:

```
SELECT * FROM account
WHERE last_name = 'daniels'
   OR last_name = 'Daniels'
   OR last_name = 'DANIELS'
```

План выполнения этого запроса показан на рис. 5.2.

Однако лучше было бы создать (дополнительный) *функциональный* индекс:

```
CREATE INDEX account_last_name_lower ON account (lower(last_name));
```

При создании функционального индекса PostgreSQL применяет функцию к значениям столбца (или столбцов), а затем помещает эти значения

в B-дерево. Подобно обычному B-дереву, где узлы содержат значения столбца, в функциональном индексе узлы содержат значения функции. В нашем случае это функция `lower`. После создания индекса запрос № 1 в листинге 5.7 вместо последовательного сканирования сможет использовать новый индекс. Соответствующий план выполнения показан на рис. 5.3.

	QUERY PLAN text
1	Bitmap Heap Scan on account (cost=14.45..506.31 rows=143 width=53)
2	Recheck Cond: ((last_name = 'daniels'::text) OR (last_name = 'Daniels'::text) OR (last_name = 'DANIELS'::text))
3	-> BitmapOr (cost=14.45..14.45 rows=143 width=0)
4	-> Bitmap Index Scan on account_last_name (cost=0.00..4.78 rows=48 width=0)
5	Index Cond: (last_name = 'daniels'::text)
6	-> Bitmap Index Scan on account_last_name (cost=0.00..4.78 rows=48 width=0)
7	Index Cond: (last_name = 'Daniels'::text)
8	-> Bitmap Index Scan on account_last_name (cost=0.00..4.78 rows=48 width=0)
9	Index Cond: (last_name = 'DANIELS'::text)

Рис. 5.2 ❖ План выполнения с перечислением условий

Листинг 5.7 ❖ В разных условиях поиска используются разные индексы

```
--- № 1
SELECT * FROM account WHERE lower(last_name) = 'daniels';
--- № 2
SELECT * FROM account WHERE last_name = 'Daniels';
--- № 3
SELECT * FROM account WHERE last_name = 'daniels';
--- № 4
SELECT * FROM account WHERE lower(last_name) = 'Daniels';
```

Обратите внимание, что индекс по столбцу `last_name` по-прежнему необходим, если мы хотим, чтобы регистрозависимый поиск поддерживался индексом (например, запрос № 2). Кроме того, если таблица `accounts` содержит запись с фамилией Daniels и запись с фамилией DANIELS, то запрос № 1 вернет оба варианта, запрос № 2 вернет только первую запись, а запросы № 3 и № 4 не вернут ни одну, ни другую.

	QUERY PLAN text
1	Bitmap Heap Scan on account (cost=5.25..382.20 rows=107 width=53)
2	Recheck Cond: (lower(last_name) = 'daniels'::text)
3	-> Bitmap Index Scan on account_last_name_lower_pattern (cost=0.00..5.22 rows=107 width=0)
4	Index Cond: (lower(last_name) = 'daniels'::text)

Рис. 5.3 ❖ План, использующий функциональный индекс

Иногда дополнительный индекс не требуется.

Должен ли функциональный индекс создаваться каждый раз, когда нам нужно выполнять поиск по преобразованному столбцу? Не обязательно. Однако важно распознать это преобразование; оно может быть незаметным.

Например, посмотрите на эту команду:

```
SELECT * FROM flight
WHERE scheduled_departure::date BETWEEN '2020-08-17' AND '2020-08-18'
```

На первый взгляд кажется, что мы используем столбец `schedule_departure` в качестве критерия фильтрации, и поскольку по этому столбцу есть индекс, он должен применяться. Однако план на рис. 5.4 переключается на последовательное сканирование.

	QUERY PLAN text
1	Seq Scan on flight (cost=0.00..30474.52 rows=3416 width=71)
2	Filter: (((scheduled_departure)::date >= '2020-08-17'::date) AND ((scheduled_departure)::date <= '2020-08-18'::date))

Рис. 5.4 ❖ План, в котором индекс не используется из-за преобразования столбца

Почему PostgreSQL не использует индекс? Потому что приведение значения типа `timestamp` к дате является преобразованием столбца.

Нужен ли дополнительный функциональный индекс по выражению `schedule_departure::date`? Не обязательно. Этот критерий фильтрации означает, что мы хотим выбрать рейсы, вылет которых запланирован на две указанные даты, независимо от времени суток. Это значит, что рейс может вылететь в любое время с полуночи 17 августа 2020 г. до полуночи 20 августа 2020 г. Для того чтобы работал существующий индекс, критерий фильтрации можно изменить:

```
SELECT * FROM flight
WHERE scheduled_departure BETWEEN '2020-08-17' AND '2020-08-19'
```

На рис. 5.5 показано, как изменяется план выполнения.

	QUERY PLAN text
1	Bitmap Heap Scan on flight (cost=167.20..13857.42 rows=7685 width=71)
2	Recheck Cond: ((scheduled_departure >= '2020-08-17 00:00:00-05'::timestamp with time zone) AND (scheduled_departure <= '2020-08-1...
3	-> Bitmap Index Scan on flight_scheduled_departure (cost=0.00..165.28 rows=7685 width=0)
4	Index Cond: ((scheduled_departure >= '2020-08-17 00:00:00-05'::timestamp with time zone) AND (scheduled_departure <= '2020-08-1...

Рис. 5.5 ❖ План выполнения, использующий индекс

В планах выполнения видно, что стоимость плана с доступом на основе индекса в два с лишним раза меньше, чем при последовательном сканировании (13857,42 против 30474,52). Что более важно, время выполнения подтверждает это наблюдение: 0,5 секунды для доступа на основе индекса против 1,5 секунды для последовательного сканирования.

Обратите пристальное внимание на этот пример. Когда вы читаете о нем в книге, предыдущий абзац кажется очевидным. Однако многие разработчики SQL и составители отчетов продолжают использовать похожие условия поиска. Один из частых случаев – поиск изменений, внесенных в таблицу за сегодняшний день. В девяноста пяти процентах случаев это условие формулируют как `update_ts::date = CURRENT_DATE`, фактически лишаясь индекса по столбцу `update_ts`. Чтобы воспользоваться индексом, критерий следует записать как

```
update_ts >= CURRENT_DATE,
```

или, если значения могут быть в будущем, условие должно быть написано так:

```
update_ts >= CURRENT_DATE AND update_ts < CURRENT_DATE + 1.
```

Рассмотрим еще один пример, когда преобразование столбца часто остается незаметным. Допустим, сегодня 17 августа 2020 г. Мы ищем рейсы, которые вылетели или запланированы на сегодня. Мы знаем, что для рейсов, которые еще не вылетели, столбец `actual_departure` может иметь неопределенное значение.

Функция `coalesce` в PostgreSQL позволяет нам использовать другое значение, когда первый аргумент неопределен. Таким образом, `coalesce(actual_departure, schedule_departure)` вернет `actual_departure`, если это значение отлично от `NULL`, и `schedule_departure` в противном случае. Оба столбца `schedule_departure` и `actual_departure` проиндексированы, и вы можете ожидать, что эти индексы будут использоваться. Например, посмотрите на план выполнения следующей инструкции SQL, представленный на рис. 5.6:

```
SELECT * FROM flight
WHERE coalesce(actual_departure, scheduled_departure)
    BETWEEN '2020-08-17' AND '2020-08-18'
```

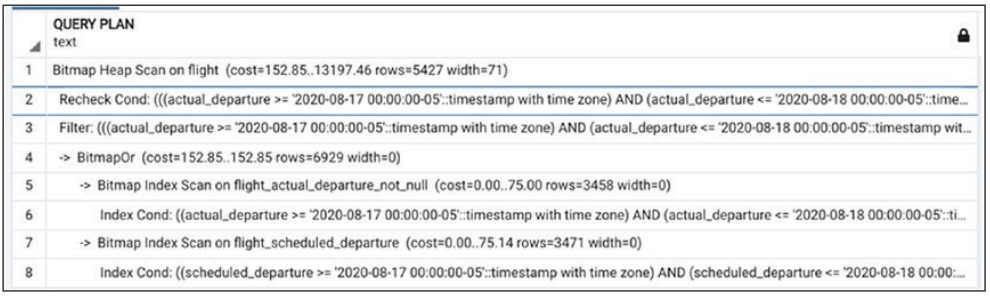
	QUERY PLAN text
1	Seq Scan on flight (cost=0.00..27058.64 rows=3416 width=71)
2	Filter: ((COALESCE(actual_departure, scheduled_departure) >= '2020-08-17 00:00:00-05'::timestamp with time zone) AND (COALESCE(actual_depa...

Рис. 5.6 ❖ План с последовательным сканированием при наличии индексов

Почему не используются индексы? Потому что `coalesce` – это функция, которая изменяет значения столбца. Должны ли мы создать еще один функциональный индекс? Это можно сделать, но на самом деле не нужно. Вместо этого перепишем оператор так, как показано в листинге 5.8, что приведет к плану выполнения, показанному на рис. 5.7.

Листинг 5.8 ❖ Запрос, использующий оба индекса

```
SELECT * FROM flight
WHERE (actual_departure BETWEEN '2020-08-17' AND '2020-08-18')
   OR (actual_departure IS NULL
       AND scheduled_departure BETWEEN '2020-08-17' AND '2020-08-18')
```



	QUERY PLAN text
1	Bitmap Heap Scan on flight (cost=152.85..13197.46 rows=5427 width=71)
2	Recheck Cond: (((actual_departure >= '2020-08-17 00:00:00-05'::timestamp with time zone) AND (actual_departure <= '2020-08-18 00:00:00-05'::time...
3	Filter: (((actual_departure >= '2020-08-17 00:00:00-05'::timestamp with time zone) AND (actual_departure <= '2020-08-18 00:00:00-05'::timestamp wit...
4	-> BitmapOr (cost=152.85..152.85 rows=6929 width=0)
5	-> Bitmap Index Scan on flight_actual_departure_not_null (cost=0.00..75.00 rows=3458 width=0)
6	Index Cond: ((actual_departure >= '2020-08-17 00:00:00-05'::timestamp with time zone) AND (actual_departure <= '2020-08-18 00:00:00-05'::ti...
7	-> Bitmap Index Scan on flight_scheduled_departure (cost=0.00..75.14 rows=3471 width=0)
8	Index Cond: ((scheduled_departure >= '2020-08-17 00:00:00-05'::timestamp with time zone) AND (scheduled_departure <= '2020-08-18 00:00:...

Рис. 5.7 ❖ План выполнения запроса из листинга 5.8

Индексы и оператор LIKE

Еще одна группа условий поиска, которые не являются простым сравнением константы со значениями столбца, – это поиск с использованием оператора `like`. Например, запрос

```
SELECT *
  FROM account
 WHERE lower(last_name) like 'johns%';
```

возвращает все учетные записи, в которых фамилия начинается с «johns». В схеме `postgres_air` список возвращаемых фамилий выглядит следующим образом:

```
"Johnson"
"Johns"
"johns"
"Johnston"
"JOHNSTON"
"JOHNS"
"JOHNSON"
"johnston"
"johnson"
```

Единственная проблема с этим запросом заключается в том, что он не использует функциональный индекс, который мы создали в предыдущем разделе, потому что B-деревья не поддерживают поиск с оператором `like`. Если мы проверим план выполнения для этого запроса, то увидим последовательное сканирование таблицы учетных записей.

Как решить эту проблему и избежать полного сканирования?

Одно из возможных решений – переписать запрос, заменив `like` двумя условиями:

```
SELECT *
  FROM account
 WHERE (lower(last_name) >= 'johns' AND lower(last_name) < 'johnt')
```

План выполнения этого запроса представлен на рис. 5.8, и мы видим, что это он использует существующий индекс.

Лучшим решением было бы создать индекс для *поиска по шаблону*:

```
CREATE INDEX account_last_name_lower_pattern
    ON account (lower(last_name) text_pattern_ops);
```

	QUERY PLAN text
1	Bitmap Heap Scan on account (cost=83.20..3905.42 rows=2222 width=53)
2	Recheck Cond: ((lower(last_name) >= 'johns'::text) AND (lower(last_name) < 'johnt'::text))
3	-> Bitmap Index Scan on account_last_name_lower (cost=0.00..82.64 rows=2222 width=0)
4	Index Cond: ((lower(last_name) >= 'johns'::text) AND (lower(last_name) < 'johnt'::text))

Рис. 5.8 ❖ План переписанного запроса, использующего индекс

Зачем нужен этот индекс? Потому что сравнение текстовых значений зависит от *локали*: набора правил, касающихся порядка символов, форматирования и прочих особенностей, которые различаются в зависимости от языка и страны. Можно подумать, что порядок, принятый в американском английском, является универсальным, но это не так. Единственная локаль, которая позволила бы нам использовать B-дерево, – это стандартная локаль «C», используемая по умолчанию. В этой локали допустимы только символы ASCII.

Чтобы узнать, какая локаль была определена при создании базы данных, необходимо выполнить команду

```
SHOW LC_COLLATE;
```

И если вы проживаете в Соединенных Штатах, то велика вероятность, что вы увидите

```
"en_US.UTF-8"
```

Созданный индекс будет использоваться запросами с оператором `like`. Новый план выполнения для нашего исходного запроса представлен на рис. 5.9, и видно, что он использует новый индекс.

	QUERY PLAN text
1	Bitmap Heap Scan on account (cost=83.61..3894.73 rows=3889 width=53)
2	Filter: (lower(last_name) ~~ 'johns%'::text)
3	-> Bitmap Index Scan on account_last_name_lower_pattern (cost=0.00..82.64 rows=2222 width=0)
4	Index Cond: ((lower(last_name) ~>=~ 'johns'::text) AND (lower(last_name) ~<~ 'johnt'::text))

Рис. 5.9 ❖ План выполнения с индексом для поиска по шаблону

Использование нескольких индексов

На рис. 5.7 показан план выполнения, в котором используются два индекса в одной таблице `flight`. В главе 3 в основном обсуждался индексный доступ в случае единственного индекса. Что происходит, когда доступно несколько индексов? Каким образом PostgreSQL использует их эффективно?

Ответ заключается в слове `bitmap`, как видно из плана выполнения. Создание карт в памяти позволяет оптимизатору использовать несколько индексов для одной таблицы для ускорения доступа к данным. Посмотрите на запрос с тремя критериями фильтрации для одной таблицы. Все критерии поддерживаются индексами.

Листинг 5.9 ❖ Запрос с тремя фильтрами для одной таблицы

```
SELECT scheduled_departure,
       scheduled_arrival
  FROM flight
 WHERE departure_airport = 'ORD'
   AND arrival_airport = 'JFK'
   AND scheduled_departure BETWEEN '2020-07-03' AND '2020-07-04';
```

План выполнения этого запроса показан на рис. 5.10.

	QUERY PLAN text
1	Bitmap Heap Scan on flight (cost=458.15..673.57 rows=1 width=16)
2	Recheck Cond: ((scheduled_departure >= '2020-07-03 00:00:00-05'::timestamp with time zone) AND (scheduled_depart...
3	Filter: (departure_airport = 'ORD'::bpchar)
4	-> BitmapAnd (cost=458.15..458.15 rows=56 width=0)
5	-> Bitmap Index Scan on flight_scheduled_departure (cost=0.00..83.52 rows=3909 width=0)
6	Index Cond: ((scheduled_departure >= '2020-07-03 00:00:00-05'::timestamp with time zone) AND (scheduled_de...
7	-> Bitmap Index Scan on flight_arrival_airport (cost=0.00..374.38 rows=9861 width=0)
8	Index Cond: (arrival_airport = 'JFK'::bpchar)

Рис. 5.10 ❖ План выполнения с несколькими индексными сканированиями для одной таблицы

Postgres может использовать результаты поиска по нескольким индексам, создавая в оперативной памяти битовую карту блоков с подходящими записями, а затем применяя к ним операторы «или» либо «и». В результате остаются только блоки, удовлетворяющие всем критериям поиска, и PostgreSQL считывает все записи в этих блоках, чтобы перепроверить условия поиска. Блоки будут сканироваться в физическом порядке, поэтому упорядочение на основе индексов будет потеряно.

Обратите внимание, что в плане выполнения, показанном на рис. 5.10, используются только два индекса из трех. Это связано с тем, что после применения логического оператора «и» к результатам поиска по двум индексам

остается только 64 строки данных, и в этом случае быстрее прочитать их все и отфильтровать лишние, чем сканировать еще один индекс, извлекая более 12 000 записей.

Использование операторов «и» и «или» для нескольких битовых карт – очень эффективный механизм применения нескольких фильтров. Но не единственный. В следующем разделе мы обсудим еще один вариант – создание составных индексов.

Составные индексы

До сих пор все показанные индексы относились к отдельным столбцам. В этом разделе обсуждаются индексы, созданные по нескольким столбцам, и рассматриваются их преимущества.

Как работают составные индексы?

Вернемся к запросу из листинга 5.9. Результат применения трех критериев поиска к таблице `flight` можно вычислить с использованием нескольких индексов. Еще один вариант – создать составной индекс по всем трем столбцам:

```
CREATE INDEX flight_depart_arr_sched_dep
    ON flight (departure_airport,
               arrival_airport,
               scheduled_departure)
```

План выполнения с этим индексом показан на рис. 5.11.

	QUERY PLAN text
1	Index Scan using flight_depart_arr_sched_dep on flight (cost=0.42..8.45 rows=1 width=16)
2	Index Cond: ((departure_airport = 'ORD'::bpchar) AND (arrival_airport = 'JFK'::bpchar) AND (scheduled_departure >= '...

Рис. 5.11 ❖ План, использующий составной индекс

Новый составной индекс будет поддерживать поиск по столбцу `departure_airport`, по столбцам `departure_airport` и `arrival_airport` и также по `departure_airport`, `arrival_airport` и `schedule_departure` вместе. Однако он не поддерживает поиск по `arrival_airport` или `scheduled_departure`.

Запрос

```
SELECT departure_airport,
       scheduled_arrival,
       scheduled_departure
  FROM flight
 WHERE arrival_airport = 'JFK'
   AND scheduled_departure BETWEEN '2020-07-03' AND '2020-07-04'
```

будет выполняться по плану, представленному на рис. 5.12.

	QUERY PLAN text
1	Bitmap Heap Scan on flight (cost=458.18..673.46 rows=56 width=20)
2	Recheck Cond: ((scheduled_departure >= '2020-07-03 00:00:00-05'::timestamp with time zone) AND (sc...
3	-> BitmapAnd (cost=458.18..458.18 rows=56 width=0)
4	-> Bitmap Index Scan on flight_scheduled_departure (cost=0.00..83.52 rows=3909 width=0)
5	Index Cond: ((scheduled_departure >= '2020-07-03 00:00:00-05'::timestamp with time zone) AND...
6	-> Bitmap Index Scan on flight_arrival_airport (cost=0.00..374.38 rows=9861 width=0)
7	Index Cond: (arrival_airport = 'JFK'::bpchar)

Рис. 5.12 ❖ Составной индекс не используется

С другой стороны, запрос

```
SELECT scheduled_departure ,
       scheduled_arrival
  FROM flight
 WHERE departure_airport = 'ORD'
   AND arrival_airport = 'JFK'
   AND scheduled_arrival BETWEEN '2020-07-03' AND '2020-07-04'
```

будет использовать составной индекс, но только для первых двух столбцов, как показано на рис. 5.13.

	QUERY PLAN text
1	Bitmap Heap Scan on flight (cost=6.38..723.75 rows=1 width=16)
2	Recheck Cond: ((departure_airport = 'ORD'::bpchar) AND (arrival_airport = 'JFK'::bpchar))
3	Filter: ((scheduled_arrival >= '2020-07-03 00:00:00-05'::timestamp with time zone) AND (scheduled_arrival <= '...
4	-> Bitmap Index Scan on flight_depart_arr_sched_dep (cost=0.00..6.38 rows=195 width=0)
5	Index Cond: ((departure_airport = 'ORD'::bpchar) AND (arrival_airport = 'JFK'::bpchar))

Рис. 5.13 ❖ План, использующий составной индекс для первых двух столбцов

Как правило, индекс по столбцам (X, Y, Z) будет использоваться для поиска по X, XY, XYZ и даже XZ, но не только по Y и не по YZ. Таким образом, при создании составного индекса недостаточно решить, какие столбцы в него включить; необходимо также учитывать их порядок.

Зачем создавать составные индексы? В конце концов, предыдущий раздел продемонстрировал, что и при использовании нескольких обычных индексов все будет работать нормально. Есть две основные причины: меньшая селективность и дополнительный источник данных.

Меньшая селективность

Помните, что чем ниже селективность, тем быстрее идет поиск, и когда мы оптимизируем короткие запросы, наша цель – избежать чтения большого количества строк на каждом шаге (даже если позже мы сможем отфильтровать их). Иногда ни одно из отдельных значений столбцов не ограничивает достаточно выборку, и только определенная комбинация значений делает запрос коротким.

В примере из предыдущего раздела 12 922 рейса вылетают из аэропорта О'Хара и 10 530 рейсов прибывают в аэропорт Джона Кеннеди. Однако вылетают из аэропорта О'Хара и приземляются в аэропорту Джона Кеннеди всего 184 рейса.

Использование индексов для получения данных

Когда все столбцы из запроса включены в составной индекс, их можно получить без доступа к таблице. Этот метод получения данных называется *сканированием только индекса*.

Все планы выполнения в предыдущем разделе читают табличные записи после того, как эти записи найдены с помощью сканирования индекса, потому что нам требовались еще и значения из столбцов, которые не были включены в индекс.

Создадим еще один составной индекс и включим в него еще один столбец:

```
CREATE INDEX flight_depart_arr_sched_dep_sched_arr
    ON flight (departure_airport,
               arrival_airport,
               scheduled_departure,
               scheduled_arrival );
```

План выполнения запроса тут же изменится на сканирование только индекса, как показано на рис. 5.14.

	QUERY PLAN text
1	Index Only Scan using flight_depart_arr_sched_dep_inc_sched_arr on flight (cost=0.42..8.45 rows=1 width=16)
2	Index Cond: ((departure_airport = 'ORD'::bpchar) AND (arrival_airport = 'JFK'::bpchar) AND (scheduled_departure >= '...

Рис. 5.14 ❖ План со сканированием только индекса

Обратите внимание, что поиск производился по первым трем столбцам индекса. Что, если бы условие поиска не включало первый столбец индекса? Например:

```
SELECT departure_airport,
       scheduled_departure,
       scheduled_arrival
```

```
  FROM flight
 WHERE arrival_airport = 'JFK'
   AND scheduled_departure BETWEEN '2020-07-03' AND '2020-07-04'
```

План выполнения в этом случае вернулся бы к использованию нескольких индексов с операторами «и» и «или», как показано на рис. 5.15.

	QUERY PLAN text
1	Bitmap Heap Scan on flight (cost=458.18..673.46 rows=56 width=20)
2	Recheck Cond: ((scheduled_departure >= '2020-07-03 00:00:00-05'::timestamp with time zone) AND (scheduled_depart...
3	-> BitmapAnd (cost=458.18..458.18 rows=56 width=0)
4	-> Bitmap Index Scan on flight_scheduled_departure (cost=0.00..83.52 rows=3909 width=0)
5	Index Cond: ((scheduled_departure >= '2020-07-03 00:00:00-05'::timestamp with time zone) AND (scheduled_de...
6	-> Bitmap Index Scan on flight_arrival_airport (cost=0.00..374.38 rows=9861 width=0)
7	Index Cond: (arrival_airport = 'JFK'::bpchar)

Рис. 5.15 ❖ Составной индекс не используется, если условие не включает первый столбец индекса

Покрывающие индексы

Покрывающие индексы появились в PostgreSQL 11. Эти индексы можно рассматривать как развитие идеи сканирования только индекса. Они специально разработаны для включения в индекс столбцов, необходимых для частых запросов определенного вида.

В предыдущем разделе к индексу был добавлен столбец `schedule_arrival` исключительно для того, чтобы избежать лишнего обращения к таблице. Он не был предназначен для поиска по нему. В таком случае вместо обычного можно использовать покрывающий индекс:

```
CREATE INDEX flight_depart_arr_sched_dep_inc_sched_arr
    ON flight (departure_airport,
               arrival_airport,
               scheduled_departure)
    INCLUDE (scheduled_arrival);
```

План выполнения запроса

```
SELECT departure_airport,
       scheduled_departure,
       scheduled_arrival
  FROM flight
 WHERE arrival_airport = 'JFK'
   AND departure_airport ='ORD'
   AND scheduled_departure BETWEEN '2020-07-03' AND '2020-07-04'
```

показан на рис. 5.16.

	QUERY PLAN text
1	Index Only Scan using flight_depart_arr_sched_dep_inc_sched_arr on flight (cost=0.42..8.45 rows=1 width=16)
2	Index Cond: ((departure_airport = 'ORD'::bpchar) AND (arrival_airport = 'JFK'::bpchar) AND (scheduled_departure >= '...

Рис. 5.16 ❖ План со сканированием только покрывающего индекса

В таких случаях нет большой разницы между включением дополнительного столбца в индекс по сравнению с созданием покрывающего индекса. Однако если вместе с индексированными значениями нужно хранить больше столбцов (или столбцы будут шире), покрывающий индекс, вероятно, будет более компактным.

Избыточные критерии отбора

Иногда, если логика фильтрации сложна и включает атрибуты из нескольких таблиц, необходимо предоставить дополнительные избыточные фильтры, чтобы движок базы данных использовал определенные индексы или уменьшил размер соединеняемых наборов строк. Такая практика называется использованием избыточных критериев отбора. Идея состоит в том, чтобы использовать этот дополнительный фильтр для предварительного выбора небольшого подмножества записей из большой таблицы.

Для некоторых из таких сложных критериев PostgreSQL может переписать запрос автоматически.

Например, условия фильтрации в запросе в листинге 5.10 объединяют значения атрибутов из таблиц `flight` и `passenger`. В ранних версиях PostgreSQL движок не мог начать фильтрацию до соединения всех таблиц, потому что оператор «и» применяется к столбцам разных таблиц.

Листинг 5.10 ❖ Запрос с условиями по двум таблицам

```
SELECT last_name,
       first_name,
       seat
  FROM boarding_pass bp
       JOIN booking_leg bl USING (booking_leg_id)
       JOIN flight f USING (flight_id)
       JOIN booking b USING(booking_id)
       JOIN passenger p USING (passenger_id)
 WHERE (
               departure_airport = 'JFK'
           AND scheduled_departure BETWEEN '2020-07-10' AND '2020-07-11'
           AND last_name = 'JOHNSON'
       ) OR (
               departure_airport = 'EDW'
           AND scheduled_departure BETWEEN '2020-07-13' AND '2020-07-14'
           AND last_name = 'JOHNSTON'
       )
```

Однако теперь оптимизатор может выполнить перезапись сложного запроса, как показано на рис. 5.17.

	QUERY PLAN text
1	Nested Loop (cost=294.48..79687.87 rows=14 width=15)
2	-> Nested Loop (cost=294.05..79663.39 rows=14 width=19)
3	Join Filter: (((f.departure_airport = 'JFK'::bpchar) AND (f.scheduled_departure >= '2020-07-10 00:00:00-05'::timestamp with time zone) AND (f.scheduled_departure <= '2020-
4	-> Nested Loop (cost=293.61..72443.63 rows=2332 width=27)
5	-> Nested Loop (cost=293.05..22798.08 rows=1650 width=20)
6	-> Bitmap Heap Scan on flight f (cost=292.49..529.86 rows=63 width=16)
7	Recheck Cond: ((((scheduled_departure >= '2020-07-10 00:00:00-05'::timestamp with time zone) AND (scheduled_departure <= '2020-07-11 00:00:00-05'::timestam
8	-> BitmapOr (cost=292.49..292.49 rows=63 width=0)
9	-> BitmapAnd (cost=279.32..279.32 rows=61 width=0)
10	-> Bitmap Index Scan on flight_scheduled_departure (cost=0.00..84.05 rows=3963 width=0)
11	Index Cond: ((scheduled_departure >= '2020-07-10 00:00:00-05'::timestamp with time zone) AND (scheduled_departure <= '2020-07-11 00:00:00-05'::tim
12	-> Bitmap Index Scan on flight_departure_airport (cost=0.00..194.99 rows=10475 width=0)
13	Index Cond: (departure_airport = 'JFK'::bpchar)
14	-> Bitmap Index Scan on flight_depart_arr_sched_dep_inc_sched_arr (cost=0.00..13.15 rows=2 width=0)
15	Index Cond: ((departure_airport = 'EDW'::bpchar) AND (scheduled_departure >= '2020-07-13 00:00:00-05'::timestamp with time zone) AND (scheduled_depa
16	-> Index Scan using booking_leg_flight_id on booking_leg bl (cost=0.56..352.57 rows=89 width=12)
17	Index Cond: (flight_id = f.flight_id)
18	-> Index Scan using boarding_pass_booking_leg_id on boarding_pass bp (cost=0.56..29.86 rows=23 width=19)
19	Index Cond: (booking_leg_id = bl.booking_leg_id)
20	-> Index Scan using passenger_pkey on passenger p (cost=0.43..3.07 rows=1 width=16)
21	Index Cond: (passenger_id = bp.passenger_id)
22	Filter: ((last_name = 'JOHNSON'::text) OR (last_name = 'JOHNSTON'::text))
23	-> Index Only Scan using booking_pkey on booking b (cost=0.43..1.75 rows=1 width=8)
24	Index Cond: (booking_id = bl.booking_id)

Рис. 5.17 ❖ План выполнения переписанного запроса с условиями по двум таблицам

Обратите внимание на строки с 8 по 15. PostgreSQL переписывает логическое выражение и выбирает все записи из таблицы `flight`, которые могут понадобиться для обоих условий, соединенных оператором `OR`.

В таких случаях нужно просто не мешать PostgreSQL делать свою работу.

Однако встречаются запросы, которые без вмешательства человека будут выполняться бесконечно. Посмотрите на запрос из листинга 5.11. Этот запрос ищет рейсы с задержкой вылета больше, чем на час (которых не должно быть много). Для всех этих задержанных рейсов запрос выбирает посадочные талоны, выданные после запланированного времени вылета.

Листинг 5.11 ❖ Короткий запрос с труднооптимизируемой фильтрацией

```
SELECT bp.update_ts boarding_pass_issued,
       scheduled_departure,
       actual_departure,
       status
  FROM flight f
       JOIN booking_leg bl USING (flight_id)
       JOIN boarding_pass bp USING (booking_leg_id)
 WHERE bp.update_ts > scheduled_departure + interval '30 minutes'
   AND f.update_ts >= scheduled_departure - interval '1 hour'
```

Возможно, этот пример может показаться надуманным, но он смоделирован на основе производственных отчетов об исключительных ситуациях. Во многих компаниях существуют те или иные отчеты об исключениях для выявления ненормального поведения системы. Важно, что по определению вывод таких отчетов должен быть небольшим. Чтобы быть полезными, отчеты об исключениях должны сообщать об условиях, которые возникают относительно редко, – иначе это будут уже обычные бизнес-отчеты.

Описанная ситуация с поздней выдачей посадочных талонов, безусловно, ненормальна, и таких случаев не должно быть много. Однако план выполнения, показанный на рис. 5.18, предусматривает полное сканирование больших таблиц и соединение хешированием, несмотря на наличие подходящих индексов по всем задействованным таблицам.

Что же пошло не так?

	QUERY PLAN text
1	Hash Join (cost=822656.15..2204168.07 rows=2810405 width=162)
2	Hash Cond: (bp.booking_leg_id = bl.booking_leg_id)
3	Join Filter: (bp.update_ts > (f.scheduled_departure + '00:30:00'::interval))
4	-> Seq Scan on boarding_pass bp (cost=0.00..513696.84 rows=25293684 width=40)
5	-> Hash (cost=654903.74..654903.74 rows=5964513 width=99)
6	-> Hash Join (cost=32574.20..654903.74 rows=5964513 width=99)
7	Hash Cond: (bl.flight_id = f.flight_id)
8	-> Seq Scan on booking_leg bl (cost=0.00..328049.64 rows=17893564 width=32)
9	-> Hash (cost=27058.64..27058.64 rows=227725 width=71)
10	-> Seq Scan on flight f (cost=0.00..27058.64 rows=227725 width=71)
11	Filter: (update_ts >= (scheduled_departure - '01:00:00'::interval))

Рис. 5.18 ❖ Неоптимальный план выполнения короткого запроса

Вернемся к определению короткого запроса. Вначале оно казалось очень ясным, но теперь ситуация осложняется. Напомним, что запрос является коротким, если ему требуется небольшое количество строк для вычисления результатов. Действительно, в данном случае количество нужных нам строк невелико, но их непросто найти. Здесь необходимо пояснение: дело не только в том, что для короткого запроса требуется небольшое количество строк, но и в том, что каждая промежуточная операция также должна давать немного строк. Если запрос с тремя соединениями короткий, но результат первого соединения получился огромным – значит, что-то не так с планом выполнения.

Как обсуждалось ранее, единственный способ прочитать небольшое количество строк из таблицы – использовать индекс. Однако у нас нет индексов, которые поддерживали бы условия фильтрации в запросе из листинга 5.11.

Более того, создать такой индекс невозможно, поскольку критерии выбора из одной таблицы зависят от значений из другой таблицы. В данном случае нет возможности сделать фильтрацию до соединения, что и приводит к полному сканированию и соединениям хешированием.

Как улучшить этот запрос? Ответ не связан напрямую с SQL. В главе 1 сказано, что оптимизация базы данных начинается со сбора требований, и это тот самый случай, когда лучший путь к оптимизации лежит через сбор точных требований.

Обратите внимание, что в исходном запросе область поиска – это все рейсы с начала времен или, по крайней мере, за весь период, охваченный базой данных. Однако это отчет об исключительных ситуациях, который, скорее всего, просматривается регулярно, и, вероятно, заказчика этого отчета интересуют только недавние случаи с момента последней проверки. Более ранние исключения уже появлялись в предыдущих отчетах и, вероятно, уже отработаны. Следующий шаг – связаться с заказчиком этого отчета и спросить, будет ли соответствовать потребностям отчет, включающий только самые последние исключения.

Если ответ – да, то избыточный критерий отбора, который мы только что получили от бизнеса, можно применить к запросу. Также нам понадобится еще один индекс:

```
CREATE INDEX boarding_pass_update_ts ON boarding_pass (update_ts);
```

В листинге 5.12 показан измененный запрос исключений за два дня.

Листинг 5.12 ❖ Запрос с добавленными избыточными критериями отбора

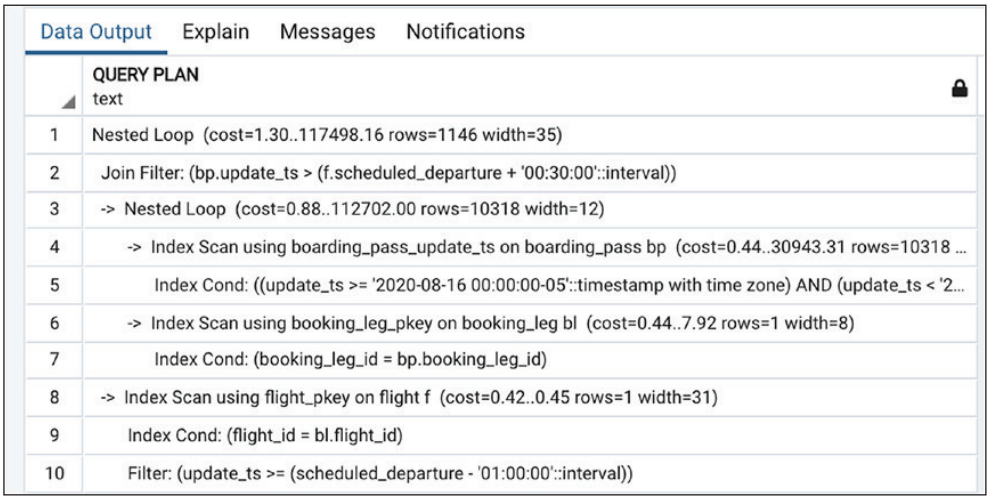
```
SELECT bp.update_ts boarding_pass_issued,
       scheduled_departure,
       actual_departure,
       status
  FROM flight f
       JOIN booking_leg bl USING (flight_id)
       JOIN boarding_pass bp USING (booking_leg_id)
 WHERE bp.update_ts > scheduled_departure + interval '30 minutes'
   AND f.update_ts >= scheduled_departure - interval '1 hour'
   AND bp.update_ts >= '2020-08-16' AND bp.update_ts < '2020-08-20'
```

Теперь сначала будет выполнен поиск по времени, как показано в плане выполнения на рис. 5.19.

Время выполнения этого запроса составляет менее 200 миллисекунд; время выполнения исходного запроса составляло 2 минуты 44 секунды.

Частичные индексы

Частичные индексы – одна из лучших возможностей PostgreSQL. Как следует из названия, частичный индекс строится на подмножестве таблицы, определяемом предложением `WHERE` команды `CREATE INDEX`.



Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Nested Loop (cost=1.30..117498.16 rows=1146 width=35)
2	Join Filter: (bp.update_ts > (f.scheduled_departure + '00:30:00'::interval))
3	-> Nested Loop (cost=0.88..112702.00 rows=10318 width=12)
4	-> Index Scan using boarding_pass_update_ts on boarding_pass bp (cost=0.44..30943.31 rows=10318 ...
5	Index Cond: ((update_ts >= '2020-08-16 00:00:00-05'::timestamp with time zone) AND (update_ts < '2...
6	-> Index Scan using booking_leg_pkey on booking_leg bl (cost=0.44..7.92 rows=1 width=8)
7	Index Cond: (booking_leg_id = bp.booking_leg_id)
8	-> Index Scan using flight_pkey on flight f (cost=0.42..0.45 rows=1 width=31)
9	Index Cond: (flight_id = bl.flight_id)
10	Filter: (update_ts >= (scheduled_departure - '01:00:00'::interval))

Рис. 5.19 ❖ План выполнения с избыточным критерием отбора

Например, для рейсов, запланированных на будущее, столбец `actual_departure` имеет неопределенное значение. Чтобы улучшить поиск по `actual_departure`, можно создать индекс только для рейсов с фактическим значением отправления, отличным от `NULL`:

```
CREATE INDEX flight_actual_departure_not_null ON flight (actual_departure)
    WHERE actual_departure IS NOT NULL
```

В данном конкретном случае разница во времени выполнения не будет значительной, потому что таблица `flight` не очень большая, и при текущем распределении данных только у половины рейсов отсутствует фактическая дата вылета. Однако если значения в столбце распределены неравномерно, то использование частичного индекса может дать большое преимущество.

Например, столбец `status` в таблице `flight` имеет только три возможных значения: On schedule (по расписанию), Delayed (отложен) и Canceled (отменен). Эти значения распределены неравномерно; рейсов со статусом On schedule значительно больше. Создавать индекс по столбцу статуса было бы непрактично из-за очень высокой селективности этого столбца. Однако было бы неплохо иметь возможность быстро отфильтровывать отмененные рейсы, тем более что в отличие от реальной жизни отмененных рейсов в схеме postgres_air не так много.

Мы создадим частичный индекс:

```
CREATE INDEX flight_canceled ON flight (flight_id)
    WHERE status = 'Canceled';
```

Этот индекс будет использоваться во всех запросах, в которых мы выбираем отмененные рейсы, независимо от любых других условий фильтрации, например:

```
SELECT * FROM flight
 WHERE scheduled_departure WHERE '2020-08-15' AND '2020-08-18'
   AND status = 'Canceled'
```

План выполнения этого запроса показан на рис. 5.20.

	QUERY PLAN text
1	Bitmap Heap Scan on flight (cost=229.66..237.63 rows=2 width=71)
2	Recheck Cond: ((status = 'Canceled'::text) AND (scheduled_departure >= '2020-08-15 00:00:00-05'::timestamp...
3	-> BitmapAnd (cost=229.66..229.66 rows=2 width=0)
4	-> Bitmap Index Scan on flight_canceled (cost=0.00..8.71 rows=114 width=0)
5	-> Bitmap Index Scan on flight_scheduled_departure (cost=0.00..220.69 rows=10427 width=0)
6	Index Cond: ((scheduled_departure >= '2020-08-15 00:00:00-05'::timestamp with time zone) AND (sche...

Рис. 5.20 ❖ Использование частичного индекса

Использование частичного индекса сокращает время выполнения с 0,72 секунды до 0,16 секунды.

Индексы и порядок соединений

Как упоминалось ранее, цель оптимизации коротких запросов – избежать больших промежуточных результатов. Это означает, что самые ограничительные критерии фильтрации должны применяться первыми. И затем после каждой операции соединения результат должен оставаться небольшим.

Количество строк, полученных в результате соединения, может быть небольшим либо из-за ограничений на соединяемые таблицы (небольшое количество записей в аргументах соединения), либо из-за полусоединения (один аргумент существенно ограничивает размер результата).

В большинстве случаев планировщик запросов выбирает правильный порядок соединений, если только не навязать ему неправильный порядок принудительно.

Начнем с создания еще нескольких индексов:

```
CREATE INDEX account_login ON account (login);
CREATE INDEX account_login_lower_pattern
    ON account (lower(login) text_pattern_ops);
CREATE INDEX passenger_last_name ON passenger (last_name);
CREATE INDEX boarding_pass_passenger_id ON boarding_pass (passenger_id);
CREATE INDEX passenger_last_name_lower_pattern
    ON passenger (lower(last_name) text_pattern_ops);
CREATE INDEX passenger_booking_id ON passenger (booking_id);
CREATE INDEX booking_account_id ON booking (account_id);
```

Теперь рассмотрим пример из листинга 5.13.

Листинг 5.13 ❖ Пример порядка соединений

```
SELECT b.account_id,
       a.login,
       p.last_name,
       p.first_name
  FROM passenger p
       JOIN booking b USING(booking_id)
       JOIN account a ON a.account_id = b.account_id
 WHERE lower(p.last_name) = 'smith'
   AND lower(login) LIKE 'smith%'
```

План выполнения этого запроса показан на рис. 5.21. Обратите внимание, что хотя первая таблица в тексте запроса – `passenger` и что первый критерий фильтрации применяется к этой же таблице, выполнение начинается с таблицы `account`.

	QUERY PLAN text
1	Nested Loop (cost=2902.68..162958.28 rows=408 width=40)
2	-> Hash Join (cost=2902.24..141003.09 rows=28223 width=36)
3	Hash Cond: (b.account_id = a.account_id)
4	-> Seq Scan on booking b (cost=0.00..123287.16 rows=5643216 width=12)
5	-> Hash (cost=2886.15..2886.15 rows=1287 width=28)
6	-> Bitmap Heap Scan on account a (cost=45.61..2886.15 rows=1287 width=28)
7	Filter: (lower(login) ~~ 'smith%'::text)
8	-> Bitmap Index Scan on account_login_lower_pattern (cost=0.00..45.29 rows=1287 width=0)
9	Index Cond: ((lower(login) ~>=~ 'smith'::text) AND (lower(login) ~<~ 'smiti'::text))
10	-> Index Scan using passenger_booking_id on passenger p (cost=0.43..0.77 rows=1 width=16)
11	Index Cond: (booking_id = b.booking_id)
12	Filter: (lower(last_name) = 'smith'::text)

Рис. 5.21 ❖ Порядок соединений: при схожей селективности выполнение начинается с меньшей таблицы

Причина в том, что таблица `account` содержит значительно меньше записей, чем таблица `passenger`, и хотя селективность обоих фильтров примерно одинакова, индекс по таблице `account` даст меньше записей.

Однако план выполнения значительно меняется, когда выбираются пассажиры с необычной фамилией, обладающей очень низкой селективностью. План выполнения на рис. 5.22 показывает, что в этом случае начать выполнение с таблицы `passenger` будет более ограничительным.

В листинге 5.14 используется похожий запрос, но вместо соединения с таблицей `passenger` выполняется соединение с таблицей `frequent_flyer`, которая составляет примерно половину размера таблицы `account`. Конечно, чтобы иметь возможность вести поиск по этой таблице, требуется еще два индекса:

```
CREATE INDEX frequent_fl_last_name_lower_pattern
    ON frequent_flyer (lower(last_name) text_pattern_ops);
CREATE INDEX frequent_fl_last_name_lower ON frequent_flyer (lower(last_name));
```

	QUERY PLAN text
1	Nested Loop (cost=134.09..80896.94 rows=137 width=41)
2	-> Nested Loop (cost=133.67..77872.90 rows=6797 width=16)
3	-> Bitmap Heap Scan on passenger p (cost=133.24..22958.25 rows=6797 width=16)
4	Recheck Cond: (lower(last_name) = 'foryth'::text)
5	-> Bitmap Index Scan on passenger_last_name_lower_pattern (cost=0.00..131.54 rows=6797 wid...
6	Index Cond: (lower(last_name) = 'foryth'::text)
7	-> Index Scan using booking_pkey on booking b (cost=0.43..8.08 rows=1 width=12)
8	Index Cond: (booking_id = p.booking_id)
9	-> Index Scan using account_pkey on account a (cost=0.42..0.44 rows=1 width=29)
10	Index Cond: (account_id = b.account_id)
11	Filter: (lower(login) ~~ 'smith%'::text)

Рис. 5.22 ❖ Другая селективность требует иного порядка соединений

В данном случае выполнение начнется с таблицы `frequent_flyer`, как показано на рис. 5.23.

Листинг 5.14 ❖ Запрос, выбирающий количество бронирований для каждого часто летающего пассажира

```
SELECT a.account_id,
       a.login,
       f.last_name,
       f.first_name,
       count(*) AS num_bookings
  FROM frequent_flyer f
       JOIN account a USING(frequent_flyer_id)
       JOIN booking b USING(account_id)
 WHERE lower(f.last_name) = 'smith'
   AND lower(login) LIKE 'smith%'
 GROUP BY 1,2,3,4
```

	QUERY PLAN text
1	GroupAggregate (cost=4388.12..4400.94 rows=570 width=49)
2	Group Key: a.account_id, f.last_name, f.first_name
3	-> Sort (cost=4388.12..4389.54 rows=570 width=41)
4	Sort Key: a.account_id, f.last_name, f.first_name
5	-> Nested Loop (cost=1389.66..4362.03 rows=570 width=41)
6	-> Hash Join (cost=1389.23..4200.87 rows=26 width=41)
7	Hash Cond: (a.frequent_flyer_id = f.frequent_flyer_id)
8	-> Bitmap Heap Scan on account a (cost=87.45..2885.45 rows=5199 width=33)
9	Filter: (lower(login) ~~ 'smith%'::text)
10	-> Bitmap Index Scan on account_login_lower_pattern (cost=0.00..86.15 rows=2573 width=0)
11	Index Cond: ((lower(login) ~>=~ 'smith'::text) AND (lower(login) ~<~ 'smiti'::text))
12	-> Hash (cost=1293.75..1293.75 rows=642 width=16)
13	-> Bitmap Heap Scan on frequent_flyer f (cost=13.39..1293.75 rows=642 width=16)
14	Recheck Cond: (lower(last_name) = 'smith'::text)
15	-> Bitmap Index Scan on frequent_fl_last_name_lower (cost=0.00..13.23 rows=642 width=0)
16	Index Cond: (lower(last_name) = 'smith'::text)
17	-> Index Only Scan using booking_account_id on booking b (cost=0.43..5.35 rows=85 width=4)
18	Index Cond: (account_id = a.account_id)

Рис. 5.23 ❖ План выполнения запроса из листинга 5.14

Когда индексы не используются

До сих пор в этой главе рассказывалось, как индексы используются в запросах. В данном разделе мы рассмотрим ситуации, когда индексы *не используются*. А именно здесь обсуждаются две ситуации: как в некоторых случаях запретить PostgreSQL использовать индексы и что делать, когда индекс не используется, а мы считаем, что его нужно использовать.

Избегаем использования индекса

Почему может понадобиться избегать использования индекса? Часто разработчики баз данных считают, что использование индексов улучшает производительность любого запроса. Каждый из нас может вспомнить ситуации, когда нас просили «построить какие-нибудь индексы, чтобы запрос выполнялся быстрее». Однако индекс не всегда нужен и в некоторых случаях может только мешать. Два примера из этой главы (рис. 5.1 и 5.6) показывают планы выполнения с последовательным чтением, которые тем не менее довольно эффективны.

Две основные причины, по которым мы можем захотеть избежать использования индексов, заключаются в следующем:

- небольшая таблица полностью считывается в оперативную память;
- для выполнения запроса нам нужна большая часть строк таблицы.

Есть ли способ избежать использования существующих индексов? В большинстве случаев оптимизатор достаточно умен, чтобы понять, когда следует, а когда не следует использовать индексы. Но в редких случаях, когда планировщик ошибается, мы можем изменить критерии фильтрации. В начале этой главы говорилось, что преобразования столбцов могут блокировать использование индексов. Это представлялось как нечто негативное, но преобразование столбцов можно использовать и для повышения производительности, когда нужно предотвратить использование индекса.

Столбец числового типа можно изменить, добавив к его значению ноль. Например, условие `attr1 + 0 = p_value` не даст использовать индекс для столбца `attr1`. Для любого типа данных функция `coalesce` будет блокировать использование индексов, поэтому, предполагая, что `attr2` не допускает неопределенных значений, условие можно изменить и написать что-то вроде `coalesce(t1.attr2, '0') = coalesce(t2.attr2, '0')`.

Почему PostgreSQL игнорирует мой индекс?

Иногда случается неприятность: подходящий индекс создан, но по какой-то причине PostgreSQL его не использует. В этот момент разработчику базы данных может сильно не хватать подсказок оптимизатору, если он имеет опыт работы с системами, которые поддерживают такие подсказки. Однако в большинстве случаев для разочарования нет причин. Оптимизатор PostgreSQL – один из лучших и в большинстве случаев поступает правильно. Так что, скорее всего, существует веская причина не использовать индекс, и ее можно найти, изучив план выполнения.

Рассмотрим пример. Здесь, а также в некоторых примерах из последующих глав используются крупные таблицы, которые не включены в дистрибутив `postgres_air` из-за своего размера. Эти таблицы необходимы, чтобы проиллюстрировать случаи, которые происходят в реальной жизни и с которыми вы можете столкнуться. Здесь используется таблица `boarding_pass_large`, у которой та же структура, что и у таблицы `boarding_pass`, но она содержит в три раза больше строк – свыше 75 000 000 посадочных талонов. Чтобы создать крупную таблицу для экспериментов, вы можете вставить каждую строку таблицы `boarding_pass` по три раза.

В базе данных `postgres_air` текущая дата – 17 августа 2020 г. Запросим выборку из ста пассажиров, которые зарегистрировались за последнюю неделю:

```
SELECT *
  FROM boarding_pass_large
 WHERE update_ts::date BETWEEN '2020-08-10' AND '2020-08-17'
 LIMIT 100
```

Как и следовало ожидать, план выполнения, представленный на рис. 5.24, показывает последовательное сканирование.

	QUERY PLAN text
1	Limit (cost=0.00..606.19 rows=100 width=40)
2	-> Seq Scan on boarding_pass_large (cost=0.00..2299882.44 rows=379402 width=40)
3	Filter: (((update_ts)::date >= '2020-08-10'::date) AND ((update_ts)::date <= '2020-08-17'::date))

Рис. 5.24 ❖ Последовательное сканирование по причине преобразования столбца

Мы рассказали, как избежать этой проблемы, так что вместо приведения типов используем интервал:

```
SELECT *
  FROM boarding_pass_large
 WHERE update_ts BETWEEN '2020-08-10' AND '2020-08-18'
 LIMIT 100
```

Однако проверка показывает, что план выполнения по-прежнему использует последовательное сканирование!

Почему избавление от преобразования столбца не привело к использованию индекса? Ответ на рис. 5.25: это сочетание относительно высокой селективности индекса для большой таблицы и наличие оператора `LIMIT`. Планировщик запросов вычисляет, что при указанном условии фильтрации будет выбрано более 700 000 строк, которые, напомним, могут потребовать вдвое больше дисковых операций ввода-вывода. Так как требуется всего 100 строк и порядок не указан, быстрее использовать последовательное сканирование таблицы. Так больше шансов, что сто записей, удовлетворяющих этому критерию, будут найдены быстрее.

	QUERY PLAN text
1	Limit (cost=0.00..267.12 rows=100 width=40)
2	-> Seq Scan on boarding_pass_large (cost=0.00..1920480.08 rows=718957 width=40)
3	Filter: ((update_ts >= '2020-08-10 00:00:00-05'::timestamp with time zone) AND (update_ts <= '2020-08-18 00:00:00-05'::timest...

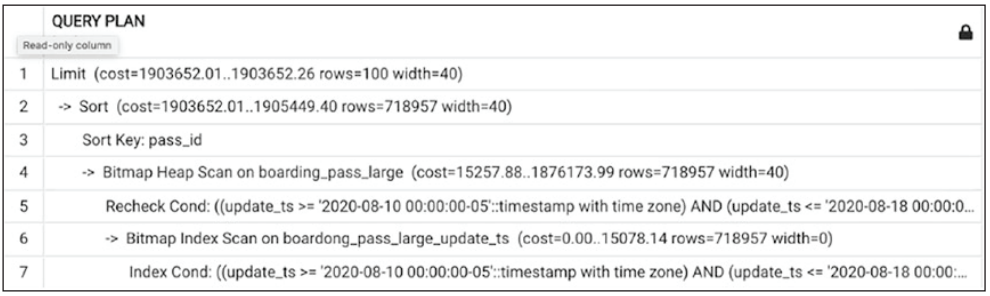
Рис. 5.25 ❖ Последовательное сканирование из-за высокой селективности индекса

Ситуация была бы иной, если бы эти сто записей нужно было выбрать в определенном порядке. Если требуется сортировка не по индексированному атрибуту, PostgreSQL потребуется выбрать *все* записи, прежде чем он сможет понять, какие из них идут первыми.

Изменим команду `SELECT`, чтобы добавить упорядочивание:

```
SELECT *
  FROM boarding_pass_large
 WHERE update_ts::date BETWEEN '2020-08-10' AND '2020-08-17'
 ORDER BY 1
 LIMIT 100
```

Теперь план выполнения (показанный на рис. 5.26) выглядит совершенно иначе.



	QUERY PLAN
	Read-only column
1	Limit (cost=1903652.01..1903652.26 rows=100 width=40)
2	-> Sort (cost=1903652.01..1905449.40 rows=718957 width=40)
3	Sort Key: pass_id
4	-> Bitmap Heap Scan on boarding_pass_large (cost=15257.88..1876173.99 rows=718957 width=40)
5	Recheck Cond: ((update_ts >= '2020-08-10 00:00:00-05'::timestamp with time zone) AND (update_ts <= '2020-08-18 00:00:0...
6	-> Bitmap Index Scan on boardong_pass_large_update_ts (cost=0.00..15078.14 rows=718957 width=0)
7	Index Cond: ((update_ts >= '2020-08-10 00:00:00-05'::timestamp with time zone) AND (update_ts <= '2020-08-18 00:00:...

Рис. 5.26 ❖ План выполнения с сортировкой

Запрос с последовательным сканированием выполнялся в течение 140 миллисекунд; запрос с принудительным индексным доступом – в течение 620 миллисекунд. Сравнение времен выполнения показывает, что последовательное сканирование в данном случае действительно более эффективно.

Не мешайте PostgreSQL делать свое дело

В этом разделе мы используем несколько модификаций запросов из предыдущих разделов, чтобы проиллюстрировать, как PostgreSQL изменяет планы выполнения на основе статистики данных.

Мы надеемся, что к настоящему времени вы получили убедительные доказательства того, что в большинстве случаев оптимизатор выполняет свою работу правильно, и обычно наша цель – не ограничивать гибкость и дать возможность сделать правильный выбор.

Вернемся к запросу по большой таблице из предыдущего раздела:

```
SELECT *
  FROM boarding_pass_large
 WHERE update_ts BETWEEN '2020-08-10' AND '2020-08-18'
 LIMIT 100
```

План выполнения этого запроса показан на рис. 5.25. Оптимизатор PostgreSQL выбрал последовательное сканирование, потому что интервал в семь дней слишком велик, чтобы получить какие-либо преимущества от индексного доступа. Теперь давайте сократим временной интервал:

```
SELECT *
  FROM boarding_pass_large
 WHERE update_ts BETWEEN '2020-08-15' AND '2020-08-18'
 LIMIT 100
```

План выполнения этого запроса на рис. 5.27 показывает, что используется доступ на основе индекса.

	QUERY PLAN text
1	Limit (cost=0.57..342.11 rows=100 width=40)
2	-> Index Scan using boardong_pass_large_update_ts on boarding_pass_large (cost=0.57..797482.15 rows=233492 width=40)
3	Index Cond: ((update_ts >= '2020-08-15 00:00:00-05'::timestamp with time zone) AND (update_ts <= '2020-08-18 00:00:00-05'::timestamp with time ...

Рис. 5.27 ❖ План использует индексный доступ для меньшего временного интервала

Продолжая проверять разные интервалы, мы увидим, что в данном случае план меняется на восьми днях. Если интервал начинается после 10 августа, план выполнения покажет использование индекса.

Что еще более интересно, если из запроса удалить предложение `LIMIT 100`, план выполнения покажет индексный доступ, но если мы увеличим интервал еще на один день, то план выполнения переключится на последовательное сканирование даже без `LIMIT 100`, как показано в планах выполнения на рис. 5.28 и 5.29.

	QUERY PLAN text
1	Bitmap Heap Scan on boarding_pass_large (cost=15257.88..1876173.99 rows=718957 width=40)
2	Recheck Cond: ((update_ts >= '2020-08-10 00:00:00-05'::timestamp with time zone) AND (update_ts <= '2020-08-18 00:00:00-05'::t...
3	-> Bitmap Index Scan on boardong_pass_large_update_ts (cost=0.00..15078.14 rows=718957 width=0)
4	Index Cond: ((update_ts >= '2020-08-10 00:00:00-05'::timestamp with time zone) AND (update_ts <= '2020-08-18 00:00:00-05'::...

Рис. 5.28 ❖ План выполнения с индексным сканированием

	QUERY PLAN text
1	Seq Scan on boarding_pass_large (cost=0.00..1920480.08 rows=983784 width=40)
2	Filter: ((update_ts >= '2020-08-09 00:00:00-05'::timestamp with time zone) AND (update_ts <= '2020-08-18 00:00:00-05'::timestam...

Рис. 5.29 ❖ План выполнения с последовательным сканированием

Рассмотрим еще один пример – запрос из листинга 5.13. Мы заметили, что в зависимости от селективности фамилии пассажира порядок соединений (и применение индексов) изменяется. На самом деле, экспериментируя с разными фамилиями, можно определить селективность, при которой меняется план выполнения (около 200 вхождений).

Наконец, рассмотрим одну относительно простую инструкцию SQL. Три запроса из листинга 5.15 идентичны, за исключением значений фильтрации для каждого критерия поиска. В первом запросе аэропорт вылета имеет высокую селективность, а имя пассажира – низкую. Во втором запросе оба значения очень селективны. В последнем запросе аэропорт вылета имеет низкую селективность. Планы, представленные на рис. 5.30, 5.31 и 5.32, различаются алгоритмами соединения, порядком соединений и используемыми индексами.

Листинг 5.15 ❖ Запрос с тремя различными наборами параметров

```
-- № 1

SELECT p.last_name,
       p.first_name
  FROM passenger p
       JOIN boarding_pass bp USING (passenger_id)
       JOIN booking_leg bl USING (booking_leg_id)
       JOIN flight USING(flight_id)
 WHERE departure_airport = 'LAX'
   AND lower(last_name) = 'clark'

-- № 2

SELECT p.last_name,
       p.first_name
  FROM passenger p
       JOIN boarding_pass bp USING (passenger_id)
       JOIN booking_leg bl USING (booking_leg_id)
       JOIN flight USING(flight_id)
 WHERE departure_airport = 'LAX'
   AND lower(last_name) = 'smith'

-- № 3

SELECT p.last_name,
       p.first_name
  FROM passenger p
       JOIN boarding_pass bp USING (passenger_id)
       JOIN booking_leg bl USING (booking_leg_id)
       JOIN flight USING(flight_id)
 WHERE departure_airport = 'FUK'
   AND lower(last_name) = 'smith'
```

Как создать правильные индексы?

В начале этой главы в схеме `postgres_air` был определен минимальный набор индексов. Практически каждый раз, когда мы хотели улучшить производительность запросов, мы предлагали создать еще один индекс. Все эти индексы действительно помогли сократить время выполнения. Но мы еще не обсуждали, необходимо ли дополнительное обоснование для создания нового индекса.

Создавать или не создавать

Двадцать лет назад мы были более осторожны при принятии решения о добавлении еще одного индекса. Во-первых, индексы занимают дополнительное место в базе данных. Во-вторых, операции вставки и обновления замед-

	QUERY PLAN text
1	Hash Join (cost=9678.47..165905.98 rows=198 width=12)
2	Hash Cond: (bl.flight_id = flight.flight_id)
3	-> Nested Loop (cost=134.45..156334.18 rows=10582 width=16)
4	-> Nested Loop (cost=134.02..138080.14 rows=10582 width=20)
5	-> Bitmap Heap Scan on passenger p (cost=133.45..23039.81 rows=6825 width=16)
6	Recheck Cond: (lower(last_name) = 'clark'::text)
7	-> Bitmap Index Scan on passenger_last_name_lower_pattern (cost=0.00..131.75 rows=6825 width=0)
8	Index Cond: (lower(last_name) = 'clark'::text)
9	-> Index Scan using boarding_pass_passenger_id on boarding_pass bp (cost=0.56..16.82 rows=4 width=16)
10	Index Cond: (passenger_id = p.passenger_id)
11	-> Index Scan using booking_leg_pkey on booking_leg bl (cost=0.44..1.73 rows=1 width=8)
12	Index Cond: (booking_leg_id = bp.booking_leg_id)
13	-> Hash (cost=9384.33..9384.33 rows=12775 width=4)
14	-> Bitmap Heap Scan on flight (cost=243.43..9384.33 rows=12775 width=4)
15	Recheck Cond: (departure_airport = 'LAX'::bpchar)
16	-> Bitmap Index Scan on flight_departure_airport (cost=0.00..240.24 rows=12775 width=0)
17	Index Cond: (departure_airport = 'LAX'::bpchar)

Рис. 5.30 ❖ План выполнения запроса № 1

	QUERY PLAN text
1	Hash Join (cost=9678.47..165905.98 rows=198 width=12)
2	Hash Cond: (bl.flight_id = flight.flight_id)
3	-> Nested Loop (cost=134.45..156334.18 rows=10582 width=16)
4	-> Nested Loop (cost=134.02..138080.14 rows=10582 width=20)
5	-> Bitmap Heap Scan on passenger p (cost=133.45..23039.81 rows=6825 width=16)
6	Recheck Cond: (lower(last_name) = ' smith'::text)
7	-> Bitmap Index Scan on passenger_last_name_lower_pattern (cost=0.00..131.75 rows=6825 width=0)
8	Index Cond: (lower(last_name) = ' smith'::text)
9	-> Index Scan using boarding_pass_passenger_id on boarding_pass bp (cost=0.56..16.82 rows=4 width=16)
10	Index Cond: (passenger_id = p.passenger_id)
11	-> Index Scan using booking_leg_pkey on booking_leg bl (cost=0.44..1.73 rows=1 width=8)
12	Index Cond: (booking_leg_id = bp.booking_leg_id)
13	-> Hash (cost=9384.33..9384.33 rows=12775 width=4)
14	-> Bitmap Heap Scan on flight (cost=243.43..9384.33 rows=12775 width=4)
15	Recheck Cond: (departure_airport = 'LAX'::bpchar)
16	-> Bitmap Index Scan on flight_departure_airport (cost=0.00..240.24 rows=12775 width=0)
17	Index Cond: (departure_airport = 'LAX'::bpchar)

Рис. 5.31 ❖ План выполнения запроса № 2

	QUERY PLAN text
1	Nested Loop (cost=12.91..217066.10 rows=360 width=12)
2	-> Nested Loop (cost=12.47..191823.09 rows=13958 width=8)
3	-> Nested Loop (cost=11.91..34548.25 rows=9874 width=4)
4	-> Bitmap Heap Scan on flight (cost=11.35..1341.38 rows=377 width=4)
5	Recheck Cond: (departure_airport = 'FUK'::bpchar)
6	-> Bitmap Index Scan on flight_depart_arr_sched_dep_inc_sched_arr (cost=0.00..11.25 row...
7	Index Cond: (departure_airport = 'FUK'::bpchar)
8	-> Index Scan using booking_leg_flight_id on booking_leg bl (cost=0.56..87.17 rows=91 width=8)
9	Index Cond: (flight_id = flight.flight_id)
10	-> Index Scan using boarding_pass_booking_leg_id on boarding_pass bp (cost=0.56..15.70 rows=...
11	Index Cond: (booking_leg_id = bl.booking_leg_id)
12	-> Index Scan using passenger_pkey on passenger p (cost=0.43..1.81 rows=1 width=16)
13	Index Cond: (passenger_id = bp.passenger_id)
14	Filter: (lower(last_name) = 'smith'::text)

Рис. 5.32 ❖ План выполнения запроса № 3

ляются, когда вместе с самой записью приходится обновлять еще и много индексов. Раньше обычно рекомендовалось удалить все индексы в таблице перед массовой загрузкой, а затем создать их снова. Некоторые учебники по базам данных по-прежнему предлагают задумываться о количестве индексов в таблице.

Но времена изменились. При нынешнем аппаратном и программном обеспечении ситуация выглядит иначе. Дисковое хранилище подешевело, диски стали быстрее, и в целом быстрый отклик более ценен, чем экономия места на диске. Двадцать лет назад настораживало, если размер индексов в таблице превышал размер самой таблицы. В наши дни это норма для OLTP-систем. Но все же остается вопрос: когда пора остановиться?

Какие индексы нужны?

Сложно дать какие-либо общие рекомендации относительно того, какие индексы нужны. В OLTP-системах время отклика обычно имеет решающее значение, и любой короткий запрос должен поддерживаться индексами.

Мы рекомендуем создавать частичные и покрывающие индексы, когда это имеет смысл. Частичные индексы меньше обычных и с большей вероятностью поместятся в оперативной памяти. Покрывающие индексы экономят обращения к таблице, позволяя движку выполнять бóльшую часть обработки в оперативной памяти.

Дополнительное время, необходимое для вставки и обновления, обычно менее важно, чем быстрый отклик. Однако за этим временем надо следить

и переосмысливать количество индексов, если обнаруживается замедление. Обычно замедление вызывается индексами, поддерживающими уникальные и первичные ключи, внешними ключами, которые ссылаются на поля уникальных и первичных ключей, а также триггерами для операций вставки или обновления. В каждом случае вам нужно будет оценить важность целостности данных по сравнению со скоростью обновлений.

В интернете доступен ряд запросов, которые вычисляют общий размер индексов в каждой таблице, и большинство инструментов мониторинга предупредят вас о чрезмерном росте.

Какие индексы не нужны?

Хотя обычно нас не волнует место на диске, занимаемое индексами, мы не хотим создавать бесполезные объекты базы данных. Системное представление `pg_stat_all_indexes` показывает статистику использования индекса (общее количество сканирований, количество прочитанных индексных строк и количество выбранных табличных строк) с момента последнего сброса статистики.

Обратите внимание, что некоторые индексы первичного ключа *никогда* не используются для извлечения данных; однако они жизненно важны для целостности данных и не должны удаляться.

Индексы и масштабируемость коротких запросов

В этом разделе мы обсудим, как оптимизировать короткие запросы, чтобы они оставались эффективными при увеличении объема данных.

В главе 1 мы упоминали, что с переходом запроса в промышленную эксплуатацию оптимизация не прекращается. Нужно продолжать отслеживать эффективность и заранее выявлять изменения в динамике производительности.

Для коротких запросов такой мониторинг производительности жизненно важен, поскольку поведение запроса может резко измениться с ростом объема данных, особенно когда скорость роста для разных таблиц различается.

Когда запрос поддерживается индексами, есть по крайней мере некоторая уверенность в его масштабируемости, потому что количество обращений к индексу растет только логарифмически относительно роста таблицы. Но если размер таблицы растет быстро, индекс может перестать помещаться в оперативную память или может вытесняться индексами конкурирующих запросов. Если такое случится, то время выполнения может резко увеличиться.

Возможно, что вначале запрос работает быстро без всяких индексов, и мы можем не знать наверняка, какие индексы понадобятся в будущем. Также возможно, что вначале условие для частичного индекса было очень ограни-

чительным и индексный доступ был быстрым, но со временем появлялось все больше и больше записей, удовлетворяющих этому условию, и индекс стал менее эффективным.

Говоря кратко, хотя мы стремимся обеспечить масштабируемость коротких запросов и их хорошую производительность даже при росте объема данных, нельзя предполагать, что какой-то запрос будет оптимизирован «раз и навсегда». Мы должны постоянно следить за объемом данных, распределением значений и прочими характеристиками, которые могут повлиять на производительность.

Выводы

В этой главе были рассмотрены короткие запросы и способы их оптимизации. Основная цель оптимизации коротких запросов – в первую очередь применить максимально ограничительные критерии поиска и гарантировать, что все промежуточные результаты остаются небольшими. Мы обсудили роль индексов для коротких запросов и показали, как определить, какие индексы необходимо создать для поддержки конкретных запросов.

Также в этой главе на различных примерах было показано, как читать планы выполнения и понимать порядок соединений и фильтраций. Мы также обсудили различные виды индексов, которые есть в PostgreSQL, и возможности их использования. Более сложные типы индексов будут подробно рассмотрены в главе 14.

Глава 6

Длинные запросы и полное сканирование

В главе 5 мы обсуждали короткие запросы и выяснили, как их распознать и какие стратегии и техники оптимизации можно с ними использовать. Мы также рассказали о важности индексов для коротких запросов, о наиболее часто используемых типах индексов и их применении. Глава 5 дала также возможность попрактиковаться в чтении планов выполнения и, надеемся, обрести уверенность в оптимизаторе PostgreSQL.

В этой главе рассматриваются длинные запросы. Некоторые запросы просто не могут выполняться за долю секунды, как бы хорошо они ни были написаны. Это не значит, что их нельзя оптимизировать. Многие практики считают, что поскольку аналитические отчеты не имеют строгих требований ко времени отклика, не важно, насколько быстро они работают. Иной раз разработчики не прилагают усилий, чтобы отчеты завершались в разумные сроки, оправдывая это тем, что запрос выполняется только раз в день, раз в неделю или раз в месяц.

Это опасная практика. Если пренебречь производительностью отчета, время выполнения может легко вырасти с минут до часов и даже больше. Мы наблюдали отчеты, которые выполнялись по шесть дней! И когда ситуация становится настолько серьезной, исправить ее в сжатые сроки непросто. Часто при разработке аналитического отчета используются исходные данные небольшого объема, и все работает хорошо. Задача разработчиков SQL – проверять планы выполнения, даже если в настоящий момент запросы выполняются нормально, и принимать упреждающие меры, чтобы предотвратить ухудшение производительности в будущем.

Какие запросы считаются длинными?

В главе 5 представлено формальное определение короткого запроса. Логично предположить, что все некороткие запросы являются длинными. Это верно, но определение, основанное на отрицании, непросто осознать и применять на практике.

Два примера длинных запросов из главы 5 (листинги 5.1 и 5.3) скопированы сюда, в листинги 6.1 и 6.2 соответственно. Первый из двух запросов – это длинный запрос, возвращающий много строк – все возможные комбинации аэропортов прибытия и отправления. Второй выводит только одну строку со средней протяженностью рейсов и общим количеством пассажиров на всех рейсах в схеме postgres_air – но все равно является длинным запросом.

Листинг 6.1 ❖ Длинный запрос, возвращающий множество строк

```
SELECT d.airport_code AS departure_airport
       a.airport_code AS arrival_airport
  FROM airport a,
       airport d
 WHERE a.airport_code <> d.airport_code
```

Листинг 6.2 ❖ Длинный запрос, возвращающий одну строку

```
SELECT avg(flight_length),
       avg (passengers)
  FROM ( SELECT flight_no,
                scheduled_arrival - scheduled_departure AS flight_length,
                count(passenger_id) passengers
           FROM flight f
                JOIN booking_leg bl ON bl.flight_id = f.flight_id
                JOIN passenger p ON p.booking_id = bl.booking_id
         GROUP BY 1,2 ) a
```

Так что же такое длинный запрос?

Запрос считается *длинным*, если селективность запроса высока по крайней мере для одной из больших таблиц; то есть результат, даже если он невелик, определяется почти всеми строками.

Каковы цели оптимизации длинных запросов? В этой главе опровергается распространенное заблуждение, что невозможно значительно улучшить производительность длинного запроса. Каждый из нас может поделиться опытом повышения производительности длинных запросов на *несколько сотен порядков*. Такие улучшения становятся возможными при применении двух стратегий оптимизации:

1) избегать многократных сканирований таблиц;
2) уменьшать размер результата на как можно более ранней стадии.

В оставшейся части главы подробно рассматриваются эти стратегии и описываются несколько методов достижения цели.

Длинные запросы и полное сканирование

В главе 5 говорилось, что короткие запросы требуют наличия индексов по столбцам, включенным в критерии поиска. Для длинных запросов все наобо-

рот: индексы не нужны, а если таблицы проиндексированы, надо убедиться, что индексы не используются.

Почему для длинных запросов предпочтительно полное сканирование таблицы? Как показано на рис. 3.1, когда количество необходимых строк достаточно велико, для индексного доступа потребуется больше операций ввода-вывода. Какой процент или количество записей является «достаточно большим», варьируется и зависит от множества разных факторов. К настоящему времени вас не должно удивлять, что в большинстве случаев PostgreSQL вычисляет этот процент верно.

В главе 5 похожие слова были сказаны и о коротких запросах. Но «достаточно велико» труднее оценить, чем «достаточно мало». Оценка этой верхней границы меняется по мере развития и улучшения аппаратного обеспечения, дисков и процессоров. По этой причине в книге мы стараемся не указывать конкретные числа пороговых значений, которые со временем обязательно изменятся. Чтобы привести показательные примеры для этой главы, мы создали несколько таблиц с сотнями миллионов строк данных. Они слишком велики, чтобы включать их в дистрибутив `postgres_air`. Однако мы не удивимся, если через пару лет для некоторых примеров не хватит и этого размера.

Длинные запросы и соединения хешированием

В большинстве примеров этой главы используется алгоритм соединения хешированием, и именно его мы надеемся увидеть в плане выполнения длинного запроса. Почему соединение хешированием в данном случае предпочтительнее? В главе 3 мы вычислили стоимость алгоритмов вложенного цикла и соединения хешированием.

Для вложенного цикла стоимость соединения таблиц `R` и `S` составляет

```
cost(nl,R,S) = size(R)*size(S) + size(R)*size(S)/size(JA)
```

Для соединения хешированием:

```
cost(hash,R,S) = size(R) + size(S) + size(R)*size(S)/size(JA)
```

Здесь `JA` обозначает количество различных значений атрибута соединения. Как уже упоминалось в главе 3, слагаемое, соответствующее размеру результирующего множества, нужно добавить к стоимости обоих алгоритмов. Но для алгоритма вложенного цикла это значение значительно меньше, чем стоимость самого соединения. Для длинных запросов размер таблиц `R` и `S` больше (потому на них не накладываются значительные ограничения), поэтому стоимость вложенных циклов значительно превышает стоимость соединения хешированием.

Если у нас есть таблица `R` с 1 000 000 строк и таблица `S` с 2 000 000 строк, а `JA` имеет 100 000 различных значений, стоимость алгоритма вложенного цикла будет равна 2 000 020 000 000, а стоимость алгоритма соединения хешированием составит 23 000 000.

Соединения хешированием работают лучше всего, когда первый аргумент помещается в оперативную память. Размер доступной памяти можно настроить с помощью параметров сервера.

В некоторых случаях используется алгоритм соединения слиянием, как показано, например, на рис. 6.10 далее в этой главе. В главе 3 мы упоминали, что соединение слиянием может быть более эффективным, если по крайней мере одна из таблиц была предварительно отсортирована. В этом примере, поскольку выбираются уникальные значения, фактически выполняется сортировка.

Подводя итоги главы 5 и этой главы, в большинстве случаев индексный доступ работает хорошо с алгоритмом вложенного цикла (и наоборот), а последовательное сканирование хорошо работает с соединением хешированием.

Почему в PostgreSQL нет подсказок оптимизатору? Есть ли способ принудительно использовать определенный алгоритм соединения? Как уже неоднократно упоминалось, лучшее, что мы можем сделать, – не ограничивать оптимизатор при написании инструкций SQL.

Длинные запросы и порядок соединений

Порядок соединений для небольших запросов обсуждался в главе 5. Для коротких запросов желаемый порядок соединения – тот, при котором в первую очередь используются индексы с более низкой селективностью.

Поскольку мы не ожидаем, что индексы будут использоваться в длинных запросах, имеет ли значение порядок соединений? Возможно, вы удивитесь, но да. Большие таблицы могут существенно отличаться по размеру. Кроме того, на практике при выборе «почти всех записей» слово «почти» может означать и 30 %, и 100 %. Даже когда индексы не используются, порядок соединений имеет значение, потому что важно, чтобы промежуточные наборы данных были как можно меньше.

Наиболее ограничительные соединения (то есть соединения, которые сильнее всего уменьшают количество результирующих строк) должны выполняться первыми.

Чаще всего оптимизатор выбирает правильный порядок; однако разработчик должен убедиться, что оптимизатор сделал верный выбор.

Что такое полусоединение?

Часто наиболее ограничительным соединением в запросе является *полусоединение*. Давайте остановимся, чтобы дать формальное определение.

Полусоединение двух таблиц `R` и `S` возвращает строки из таблицы `R`, для которых есть хотя бы одна строка из таблицы `S` с совпадающими значениями в соединяемых столбцах.

Полусоединение не является дополнительной операцией SQL; мы не пишем что-то вроде `SELECT a.* FROM a SEMI JOIN b`. Полусоединение – особый вид соединения, удовлетворяющий двум условиям. Во-первых, в результирующем множестве появляются только столбцы из первой таблицы. Во-вторых, строки из первой таблицы не дублируются, если во второй таблице для них есть несколько соответствий. Чаще всего полусоединение вообще не предполагает ключевое слово `JOIN`. Первый и наиболее распространенный способ получить полусоединение представлен в листинге 6.3. Этот запрос находит всю информацию о рейсах с по крайней мере одним бронированием.

Листинг 6.3 ❖ Использование полусоединения с помощью ключевого слова EXISTS

```
SELECT * FROM flight f WHERE EXISTS
  (SELECT flight_id FROM booking_leg WHERE flight_id = f.flight_id)
```

Этот запрос использует неявное соединение с таблицей `booking_leg` для фильтрации записей таблицы `flight`. Другими словами, вместо указания значений для фильтрации мы используем значения столбцов из другой таблицы.

Эквивалентный запрос, показывающий еще один способ получить полусоединение, представлен в листинге 6.4.

Листинг 6.4 ❖ Использование полусоединения с помощью ключевого слова IN

```
SELECT * FROM flight WHERE flight_id IN
  (SELECT flight_id FROM booking_leg)
```

Как эти запросы могут содержать соединения, если ни один из них не использует ключевое слово `JOIN`? Ответ находится в плане выполнения, который идентичен для обоих запросов и показан на рис. 6.1.

	QUERY PLAN text
1	Nested Loop Semi Join (cost=0.56..438436.84 rows=197629 width=71)
2	-> Seq Scan on flight f (cost=0.00..23642.76 rows=683176 width=71)
3	-> Index Only Scan using booking_leg_flight_id on booking_leg (cost=0.56..3.32 rows=91 width=4)
4	Index Cond: (flight_id = f.flight_id)

Рис. 6.1 ❖ План выполнения для полусоединения

В этом плане мы видим `SEMI JOIN`, хотя ключевое слово `JOIN` не использовалось в самом запросе.

Хотя два этих способа написания запросов с полусоединениями одинаковы по смыслу, в PostgreSQL только первый из них гарантирует наличие `SEMI JOIN` в плане выполнения. Для запросов из листингов 6.3 и 6.4 планы совпадают, но в других случаях оптимизатор может переписать запрос с использованием обычного соединения. Это решение зависит от кардинальности соединения двух таблиц и от селективности фильтра.

Полусоединения и порядок соединений

Поскольку полусоединение может значительно уменьшить размер набора результатов и, по определению, никогда не увеличит его, оно часто является наиболее ограничительным соединением в запросе, и как указывалось ранее, именно наиболее ограничительное соединение должно выполняться в первую очередь.

Полусоединения никогда не увеличивают размер набора результатов; проверьте, выгодно ли применить их в первую очередь.

Конечно, это возможно, лишь если условие полусоединения применяется к столбцам одной из таблиц. В тех случаях, когда условие полусоединения ссылается на несколько таблиц, эти таблицы нужно соединить до применения полусоединения.

Рассмотрим пример из листинга 6.5, где показаны бронирования рейсов, которые отправляются из аэропортов, находящихся в США.

Листинг 6.5 ❖ Порядок соединений при наличии полусоединения

```
SELECT departure_airport,
       booking_id,
       is_returning
  FROM booking_leg bl
       JOIN flight f USING (flight_id)
 WHERE departure_airport IN
       (SELECT airport_code FROM airport WHERE iso_country = 'US')
```

На рис. 6.2 показан план выполнения этого запроса.

	QUERY PLAN text
1	Nested Loop (cost=20.65..636661.37 rows=3788277 width=9)
2	-> Hash Join (cost=20.09..25467.60 rows=144636 width=8)
3	Hash Cond: (f.departure_airport = airport.airport_code)
4	-> Seq Scan on flight f (cost=0.00..23642.76 rows=683176 width=8)
5	-> Hash (cost=18.33..18.33 rows=141 width=4)
6	-> Seq Scan on airport (cost=0.00..18.33 rows=141 width=4)
7	Filter: (iso_country = 'US'::text)
8	-> Index Scan using booking_leg_flight_id on booking_leg bl (cost=0.56..3.32 rows=91 width=9)
9	Index Cond: (flight_id = f.flight_id)

Рис. 6.2 ❖ План выполнения для запроса из листинга 6.5

Этот план выполнения показывает не операцию полусоединения, а соединение хешированием, поскольку здесь нет дубликатов, которые нужно удалить. Однако это именно логическое полусоединение, и оно наиболее ограничительно, поэтому выполняется первым. Также стоит сделать небольшое отступление, чтобы отметить последовательное сканирование таблицы `airport`. Последовательное сканирование используется, потому что по полю `iso_country` нет индекса. Давайте создадим этот индекс и посмотрим, приведет ли он к ускорению.

```
CREATE INDEX airport_iso_country ON airport (iso_country);
```

Если этот индекс существует, то планировщик запросов будет использовать его, как показано на рис. 6.3. Однако время выполнения в этом случае будет таким же или хуже, чем время при последовательном сканировании, потому что этот индекс недостаточно избирательный. Пока удалим его.

	QUERY PLAN text
1	Nested Loop (cost=19.46..636660.17 rows=3788277 width=9)
2	-> Hash Join (cost=18.89..25466.40 rows=144636 width=8)
3	Hash Cond: (f.departure_airport = airport.airport_code)
4	-> Seq Scan on flight f (cost=0.00..23642.76 rows=683176 width=8)
5	-> Hash (cost=17.13..17.13 rows=141 width=4)
6	-> Bitmap Heap Scan on airport (cost=5.37..17.13 rows=141 width=4)
7	Recheck Cond: (iso_country = 'US'::text)
8	-> Bitmap Index Scan on airport_iso_country (cost=0.00..5.33 rows=141 width=0)
9	Index Cond: (iso_country = 'US'::text)
10	-> Index Scan using booking_leg_flight_id on booking_leg bl (cost=0.56..3.32 rows=91 width...
11	Index Cond: (flight_id = f.flight_id)

Рис. 6.3 ❖ План выполнения с индексным сканированием

Подробнее о порядке соединений

Посмотрим на более сложный пример длинного запроса с несколькими полусоединениями в листинге 6.6. Этот запрос, как и предыдущий, находит бронирования рейсов с вылетом из США, но ограничен бронированиями, обновленными с 1 июля 2020 г. Поскольку у нас нет индекса по столбцу `update_ts` таблицы `booking`, давайте создадим его и посмотрим, будет ли он использоваться:

```
CREATE INDEX booking_update_ts ON booking (update_ts);
```

Листинг 6.6 ❖ Два полусоединения в одном длинном запросе

```
SELECT departure_airport, booking_id, is_returning
  FROM booking_leg bl
       JOIN flight f USING (flight_id)
  WHERE departure_airport IN
        (SELECT airport_code FROM airport WHERE iso_country = 'US')
    AND bl.booking_id IN
        (SELECT booking_id FROM booking WHERE update_ts > '2020-07-01')
```

План выполнения на рис. 6.4 показывает, что полусоединение по `airport.iso_country` выполняется первым. Как и в предыдущем запросе, мы используем ключевое слово `IN`, но оптимизатор использует `JOIN`, а не `SEMI JOIN`, потому что нет необходимости удалять дубликаты.

	QUERY PLAN text
1	Hash Join (cost=236218.96..927427.92 rows=1311781 width=9)
2	Hash Cond: (bl.booking_id = booking.booking_id)
3	-> Nested Loop (cost=20.65..636661.37 rows=3788277 width=9)
4	-> Hash Join (cost=20.09..25467.60 rows=144636 width=8)
5	Hash Cond: (f.departure_airport = airport.airport_code)
6	-> Seq Scan on flight f (cost=0.00..23642.76 rows=683176 width=8)
7	-> Hash (cost=18.33..18.33 rows=141 width=4)
8	-> Seq Scan on airport (cost=0.00..18.33 rows=141 width=4)
9	Filter: (iso_country = 'US'::text)
10	-> Index Scan using booking_leg_flight_id on booking_leg bl (cost=0.56..3.32 rows=91 width=9)
11	Index Cond: (flight_id = f.flight_id)
12	-> Hash (cost=204162.39..204162.39 rows=1952634 width=8)
13	-> Seq Scan on booking (cost=0.00..204162.39 rows=1952634 width=8)
14	Filter: (update_ts > '2020-07-01 00:00:00-05'::timestamp with time zone)

Рис. 6.4 ❖ План выполнения с двумя полусоединениями

Следует отметить три момента в этом плане выполнения. Во-первых, хотя для получения промежуточных результатов используется доступ на основе индекса и мы видим, что в данном случае применяется алгоритм соединения вложенными циклами, финальное соединение основано на хешировании, потому что используется значительная часть обоих наборов данных. Во-вторых, полусоединение использует последовательное сканирование таблицы. И даже хотя таким образом мы читаем все строки из таблицы `airport`, результирующее множество получается меньше, чем при объединении рейсов с сегментами бронирования и последующей фильтрацией по местоположению аэропорта. Спасибо оптимизатору, который выбирает наиболее ограничительное полусоединение.

Наконец, хотя по столбцу `update_ts` таблицы `booking` есть индекс, он не используется, потому что условие `update_ts > '2020-07-01'` охватывает почти половину строк таблицы.

Однако если мы изменим в этом запросе (показанном выше в листинге 6.6) критерии фильтрации и уменьшим интервал до `update_ts > '2020-08-01'`, план

выполнения радикально изменится – см. рис. 6.5. В новом плане выполнения видно, что фильтр по `update_ts` теперь более ограничительный, а оптимизатор отдает предпочтение индексному доступу.

Действительно ли индексный доступ к таблице `booking` лучше в данном случае? Мы можем сравнить, запретив индексный доступ с помощью преобразования столбца `update_ts` и переписав фильтр следующим образом: `coalesce(update_ts, '2020-08-03') > '2020-08-02'`.

	QUERY PLAN text
1	Hash Join (cost=236849.96..818170.46 rows=389071 width=9)
2	Hash Cond: (bl.flight_id = f.flight_id)
3	-> Hash Join (cost=209009.41..761034.69 rows=1837738 width=9)
4	Hash Cond: (bl.booking_id = booking.booking_id)
5	-> Seq Scan on booking_leg bl (cost=0.00..328049.64 rows=17893564 width=9)
6	-> Hash (cost=199507.09..199507.09 rows=579146 width=8)
7	-> Bitmap Heap Scan on booking (cost=10848.81..199507.09 rows=579146 width=8)
8	Recheck Cond: (update_ts > '2020-08-01 00:00:00-05'::timestamp with time zone)
9	-> Bitmap Index Scan on booking_update_ts (cost=0.00..10704.03 rows=579146 width=0)
10	Index Cond: (update_ts > '2020-08-01 00:00:00-05'::timestamp with time zone)
11	-> Hash (cost=25467.60..25467.60 rows=144636 width=8)
12	-> Hash Join (cost=20.09..25467.60 rows=144636 width=8)
13	Hash Cond: (f.departure_airport = airport.airport_code)
14	-> Seq Scan on flight f (cost=0.00..23642.76 rows=683176 width=8)
15	-> Hash (cost=18.33..18.33 rows=141 width=4)
16	-> Seq Scan on airport (cost=0.00..18.33 rows=141 width=4)
17	Filter: (iso_country = 'US'::text)

Рис. 6.5 ❖ План выполнения с двумя полусоединениями с разной селективностью

Как видно на рис. 6.6, эти действия приводят к последовательному сканированию. И действительно, блокировка индекса и принудительное последовательное сканирование работают лучше, чем индексный доступ на больших временных интервалах. По мере сокращения временного интервала у индексного доступа появляется преимущество. Дата 2020-08-01 оказывается переломным моментом; для всех дат, начиная с 2020-08-02, индексный доступ будет работать лучше.

Что такое антисоединение?

Как ясно из самого названия, `ANTI JOIN` – это противоположность `SEMI JOIN`. Формально:

> Антисоединение двух таблиц `R` и `S` возвращает строки из таблицы `R`, для которых в таблице `S` нет строк с совпадающим значением в столбце соединения.

	QUERY PLAN text
1	Hash Join (cost=262841.74..1061063.14 rows=1258140 width=9)
2	Hash Cond: (bl.flight_id = f.flight_id)
3	-> Hash Join (cost=235001.19..939755.08 rows=5942704 width=9)
4	Hash Cond: (bl.booking_id = booking.booking_id)
5	-> Seq Scan on booking_leg bl (cost=0.00..476508.08 rows=17828108 width=9)
6	-> Hash (cost=204162.39..204162.39 rows=1879664 width=8)
7	-> Seq Scan on booking (cost=0.00..204162.39 rows=1879664 width=8)
8	Filter: (COALESCE(update_ts, '2020-08-02 00:00:00-05'::timestamp with time zone) > '2020-08-01 00:00:00-05'::timestamp with time zone)
9	-> Hash (cost=25467.60..25467.60 rows=144636 width=8)
10	-> Hash Join (cost=20.09..25467.60 rows=144636 width=8)
11	Hash Cond: (f.departure_airport = airport.airport_code)
12	-> Seq Scan on flight f (cost=0.00..23642.76 rows=683176 width=8)
13	-> Hash (cost=18.33..18.33 rows=141 width=4)
14	-> Seq Scan on airport (cost=0.00..18.33 rows=141 width=4)
15	Filter: (iso_country = 'US'::text)

Рис. 6.6 ❖ Принудительное полное сканирование

Как и в случае с полусоединением, не существует команды `ANTI JOIN`. Вместо этого запрос с антисоединением можно записать двумя разными способами, представленными в листингах 6.7 и 6.8. Они запрашивают рейсы, на которые нет бронирований.

Листинг 6.7 ❖ Использование антисоединения с помощью NOT EXISTS

```
SELECT * FROM flight f WHERE NOT EXISTS
  (SELECT flight_id FROM booking_leg WHERE flight_id = f.flight_id)
```

Листинг 6.8 ❖ Использование антисоединения с помощью NOT IN

```
SELECT * FROM flight WHERE flight_id NOT IN
  (SELECT flight_id FROM booking_leg)
```

Как и в случае с полусоединениями, хотя оба способа написания запроса с антисоединением семантически эквивалентны, в PostgreSQL только версия с `NOT EXISTS` гарантирует наличие антисоединения в плане исполнения. На рис. 6.7 и 6.8 показан план выполнения для листингов 6.7 и 6.8 соответственно. В этом конкретном случае оба запроса будут выполнены примерно за одно и то же время, а план с антисоединением лишь немного быстрее. Не существует общих указаний относительно того, какой синтаксис для антисоединения лучше. Разработчикам следует опробовать оба способа, чтобы увидеть, какой из них будет работать лучше в их случае.

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	Nested Loop Anti Join (cost=0.56..429448.81 rows=543600 width=71)
2	-> Seq Scan on flight f (cost=0.00..23642.76 rows=683176 width=71)
3	-> Index Only Scan using booking_leg_flight_id on booking_leg (cost=0.56..3.48 rows=128 width=4)
4	Index Cond: (flight_id = f.flight_id)

Рис. 6.7 ❖ План выполнения с антисоединением

	QUERY PLAN text
1	Seq Scan on flight (cost=0.00..232232347044.02 rows=341588 width=71)
2	Filter: (NOT (SubPlan 1))
3	SubPlan 1
4	-> Materialize (cost=0.00..635290.62 rows=17828108 width=4)
5	-> Seq Scan on booking_leg (cost=0.00..476508.08 rows=17828108 width=4)

Рис. 6.8 ❖ План выполнения без антисоединения

Полу- и антисоединения с использованием оператора JOIN

На этом этапе внимательный читатель может задаться вопросом, почему нельзя использовать явное соединение и указать именно то, что нужно. Зачем использовать операторы `EXISTS` и `IN`? Отвечаем: это возможно и в некоторых случаях действительно лучше, чем использовать полусоединения. Нужно только аккуратно построить логически эквивалентный запрос.

Запросы в листингах 6.3 и 6.4 семантически эквивалентны, но запрос в листинге 6.9 не эквивалентен им. Напомним, что запросы в листингах 6.3 и 6.4 возвращают информацию о рейсах, на которые есть хотя бы одно бронирование.

Листинг 6.9 ❖ Соединение возвращает дубликаты

```
SELECT f.*
  FROM flight f
       JOIN booking_leg bl USING (flight_id)
```

В отличие от них, запрос в листинге 6.9 вернет для каждого рейса столько строк, сколько найдется бронирований с соответствующим `flight_id`. Чтобы вернуть только по одной записи на рейс, как в исходном варианте, запрос необходимо переписать, как показано в листинге 6.10.

Листинг 6.10 ❖ Запрос с соединением, возвращающий по одной строке на рейс

```
SELECT *
  FROM flight f
       JOIN (SELECT DISTINCT flight_id FROM booking_leg) bl
       USING (flight_id)
```

План выполнения этого запроса показан на рис. 6.9, и он не содержит полусоединение.

	QUERY PLAN text
1	Hash Join (cost=40189.02..828769.06 rows=139576 width=71)
2	Hash Cond: (booking_leg.flight_id = f.flight_id)
3	-> Unique (cost=0.56..777720.45 rows=139576 width=4)
4	-> Index Only Scan using booking_leg_flight_id on booking_leg (cost=0.56..733150.18 rows=17828108 width=4)
5	-> Hash (cost=23642.76..23642.76 rows=683176 width=71)
6	-> Seq Scan on flight f (cost=0.00..23642.76 rows=683176 width=71)

Рис. 6.9 ❖ План выполнения для запроса из листинга 6.10

Из плана выполнения не очевидно, будет ли этот запрос выполняться быстрее или медленнее, чем запрос с полусоединением. На практике он выполняется более чем в два раза быстрее, чем запрос из листинга 6.3.

Если вам нужны только идентификаторы рейсов, на которые имеется бронирование, достаточно выполнить запрос из листинга 6.11.

Листинг 6.11 ❖ Запрос с соединением, возвращающий только flight_id

```
SELECT flight_id
  FROM flight f
       JOIN (SELECT DISTINCT flight_id FROM booking_leg) bl USING (flight_id)
```

План выполнения этого запроса, показанный на рис. 6.10, значительно отличается от плана на рис. 6.9, а выполнение идет еще быстрее.

	QUERY PLAN text
1	Merge Join (cost=0.99..819305.18 rows=139576 width=4)
2	Merge Cond: (f.flight_id = booking_leg.flight_id)
3	-> Index Only Scan using flight_pkey on flight f (cost=0.42..36736.33 rows=683176 width=4)
4	-> Unique (cost=0.56..777720.45 rows=139576 width=4)
5	-> Index Only Scan using booking_leg_flight_id on booking_leg (cost=0.56..733150.18 rows=17828108 width=4)

Рис. 6.10 ❖ План выполнения с соединением слиянием

А что насчет антисоединений? Антисоединение не может создавать дубликаты, а это означает, что можно использовать `OUTER JOIN` с последующим

отбором неопределенных значений. Таким образом, запрос из листинга 6.7 эквивалентен запросу из листинга 6.12.

Листинг 6.12 ❖ Внешнее соединение с отбором неопределенных значений

```
SELECT f.flight_id
  FROM flight f
LEFT OUTER JOIN booking_leg bl USING (flight_id)
 WHERE bl.flight_id IS NULL
```

План выполнения этого запроса использует антисоединение – см. рис. 6.11.

	QUERY PLAN text
1	Nested Loop Anti Join (cost=0.56..429448.81 rows=543600 width=4)
2	-> Seq Scan on flight f (cost=0.00..23642.76 rows=683176 width=4)
3	-> Index Only Scan using booking_leg_flight_id on booking_leg bl (cost=0.56..3.48 rows=128 width=4)
4	Index Cond: (flight_id = f.flight_id)

Рис. 6.11 ❖ План выполнения для запроса из листинга 6.12.

Оптимизатор распознает эту конструкцию и переписывает ее на антисоединение. Такое поведение оптимизатора является стабильным, и на него можно положиться.

Когда необходимо указывать порядок соединения?

До сих пор оптимизатор выбирал лучший порядок соединения без какого-либо вмешательства со стороны разработчика SQL, но так бывает не всегда.

Длинные запросы более вероятны в системах OLAP. Другими словами, длинный запрос, скорее всего, представляет собой аналитический отчет, соединяющий некоторое количество таблиц. Это количество, как может подтвердить любой, кто работал с системами OLAP, бывает довольно внушительным. Когда количество таблиц, участвующих в запросе, становится слишком большим, оптимизатор больше не пытается найти лучший порядок соединения из всех возможных. Хотя рассмотрение параметров системы выходит за рамки этой книги, стоит упомянуть об одном из них: `join_collapse_limit`.

Данный параметр ограничивает количество таблиц в соединении, которые будут обрабатываться стоимостным оптимизатором. Значение этого параметра по умолчанию равно 8. Это означает, что если количество таблиц в соединении меньше или равно восьми, оптимизатор построит планы-кандидаты, сравнит их и выберет лучший. Но если количество таблиц больше или равно девяти, он просто выполнит соединения в том порядке, в котором таблицы перечислены в запросе.

Почему бы не задать для этого параметра максимально возможное значение? Официального верхнего предела для этого параметра не существует, можно использовать любое целое число вплоть до максимума, равного 2 147 483 647. Однако чем выше будет значение, которое вы зададите для этого параметра, тем больше времени будет потрачено на выбор лучшего плана. Число возможных планов, которые следует учитывать, равно факториалу от количества таблиц. Таким образом, для 8 таблиц нужно сравнить сорок тысяч планов. Если увеличить количество таблиц до 10, придется сравнить уже три миллиона планов. Очевидно, что количество растет и дальше – если задать для параметра значение 20, общее количество планов уже выйдет за пределы типа integer. Один из авторов однажды наблюдал, как специалист по обработке данных изменил значение этого параметра на 30, чтобы обработать запрос с тридцатью соединениями. Последствия оказались плачевными – запрос «повис», и даже команда EXPLAIN не могла вернуть результат.

С параметром легко экспериментировать, задавая значение локально на уровне сеанса, поэтому выполните

```
SET join_collapse_limit = 10
```

и проверьте время выполнения команды EXPLAIN.

Кроме того, вспомните, что статистика таблицы недоступна для промежуточных результатов, а это может привести к тому, что оптимизатор выберет неоптимальный порядок соединения. Если разработчик SQL знает лучший порядок соединений, можно принудительно установить этот порядок, задав для join_collapse_limit значение 1. В этом случае оптимизатор сгенерирует план, в котором соединения будут выполняться в том порядке, в каком они указаны в команде SELECT.

Принудительно установить определенный порядок соединения можно, задав для параметра join_collapse_limit значение 1.

Например, если выполняется команда из листинга 6.13 (то есть EXPLAIN для запроса из листинга 6.6), план выполнения на рис. 6.12 показывает, что соединения выполняются точно в том порядке, в котором они перечислены, а индекс по update_ts не используется (что в этом случае отрицательно сказывается на производительности).

Листинг 6.13 ❖ Отключение стоимостной оптимизации

```
SET join_collapse_limit=1;
EXPLAIN
  SELECT departure_airport, booking_id, is_returning
    FROM booking_leg bl
         JOIN flight f USING (flight_id)
   WHERE departure_airport IN
         (SELECT airport_code FROM airport WHERE iso_country = 'US')
     AND bl.booking_id IN
         (SELECT booking_id FROM booking WHERE update_ts > '2020-08-01')
```

	QUERY PLAN text
1	Hash Join (cost=266193.25..897630.55 rows=400551 width=9)
2	Hash Cond: (bl.booking_id = booking.booking_id)
3	-> Hash Join (cost=26627.55..608785.61 rows=3788277 width=9)
4	Hash Cond: (f.departure_airport = airport.airport_code)
5	-> Hash Join (cost=26607.46..561496.02 rows=17893566 width=9)
6	Hash Cond: (bl.flight_id = f.flight_id)
7	-> Seq Scan on booking_leg bl (cost=0.00..310506.66 rows=17893566 width=9)
8	-> Hash (cost=15398.76..15398.76 rows=683176 width=8)
9	-> Seq Scan on flight f (cost=0.00..15398.76 rows=683176 width=8)
10	-> Hash (cost=18.33..18.33 rows=141 width=4)
11	-> Seq Scan on airport (cost=0.00..18.33 rows=141 width=4)
12	Filter: (iso_country = 'US'::text)
13	-> Hash (cost=229744.28..229744.28 rows=598594 width=8)
14	-> Seq Scan on booking (cost=0.00..229744.28 rows=598594 width=8)
15	Filter: (update_ts > '2020-08-01 00:00:00-05'::timestamp with time zone)

Рис. 6.12 ❖ План выполнения с отключенной оптимизацией на основе затрат

Еще один способ установить определенный порядок соединений – использовать *общие табличные выражения*, которые мы обсудим в главе 7.

Группировка: сначала фильтруем, затем группируем

В главе 5 мы упоминали, что для коротких запросов группировка не занимает много времени. Для длинных запросов такой подход к группировке может очень сильно повлиять на производительность. Неоптимальные решения относительно момента выполнения группировки часто становятся основной причиной медленного выполнения запросов.

В листинге 6.14 показан запрос, который для каждого рейса, имеющего бронирования, вычисляет среднюю стоимость полета и общее количество пассажиров.

Для вычисления этих значений только для одного рейса обычным антипаттерном является запрос из листинга 6.15.

В этом запросе мы выбираем данные для одного рейса из вложенной команды SELECT. Более ранние версии PostgreSQL не могли эффективно обрабатывать такие конструкции. Движок базы данных сначала выполнял внутреннюю команду SELECT с группировкой и только затем выбирал строку, со-

ответствующую конкретному рейсу. Чтобы запрос выполнялся эффективно, его нужно было написать, как показано в листинге 6.16.

Листинг 6.14 ❖ Средняя цена билета и общее количество пассажиров

```
SELECT bl.flight_id,
       departure_airport,
       (avg(price))::numeric (7,2) AS avg_price,
       count(DISTINCT passenger_id) AS num_passengers
  FROM booking b
       JOIN booking_leg bl USING (booking_id)
       JOIN flight f USING (flight_id)
       JOIN passenger p USING (booking_id)
 GROUP BY 1,2
```

Листинг 6.15 ❖ Средняя цена билета и общее количество пассажиров на одном рейсе

```
SELECT * FROM (
        SELECT bl.flight_id,
               departure_airport,
               (avg(price))::numeric (7,2) AS avg_price,
               count(DISTINCT passenger_id) AS num_passengers
          FROM booking b
               JOIN booking_leg bl USING (booking_id)
               JOIN flight f USING (flight_id)
               JOIN passenger p USING (booking_id)
         GROUP BY 1,2
       ) a
 WHERE flight_id = 222183
```

Листинг 6.16 ❖ Перемещение условия на уровень GROUP BY

```
SELECT bl.flight_id,
       departure_airport,
       (avg(price))::numeric (7,2) AS avg_price,
       count(DISTINCT passenger_id) AS num_passengers
  FROM booking b
       JOIN booking_leg bl USING (booking_id)
       JOIN flight f USING (flight_id)
       JOIN passenger p USING (booking_id)
 WHERE flight_id = 222183
 GROUP BY 1,2
```

Но теперь благодаря постоянным улучшениям оптимизатора оба запроса будут выполняться по плану, показанному на рис. 6.13. Этот план использует индексный доступ, и время выполнения запроса составляет около 2 секунд.

Фильтрация всех столбцов, используемых в предложении GROUP BY, должна выполняться на уровне группировки.

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	GroupAggregate (cost=1201.39..1205.93 rows=89 width=30)
2	Group Key: bl.flight_id, f.departure_airport
3	-> Sort (cost=1201.39..1202.03 rows=256 width=18)
4	Sort Key: f.departure_airport
5	-> Nested Loop (cost=6.54..1191.15 rows=256 width=18)
6	-> Nested Loop (cost=6.11..1116.80 rows=89 width=26)
7	-> Index Scan using flight_pkey on flight f (cost=0.42..8.44 rows=1 width=8)
8	Index Cond: (flight_id = 222183)
9	-> Nested Loop (cost=5.68..1107.47 rows=89 width=22)
10	-> Bitmap Heap Scan on booking_leg bl (cost=5.25..355.42 rows=89 width=8)
11	Recheck Cond: (flight_id = 222183)
12	-> Bitmap Index Scan on booking_leg_flight_id (cost=0.00..5.23 rows=89 width=0)
13	Index Cond: (flight_id = 222183)
14	-> Index Scan using booking_pkey on booking b (cost=0.43..8.45 rows=1 width=14)
15	Index Cond: (booking_id = bl.booking_id)
16	-> Index Scan using passenger_booking_id on passenger p (cost=0.43..0.75 rows=9 width=8)
17	Index Cond: (booking_id = b.booking_id)

Рис. 6.13 ❖ План выполнения для одного рейса

В текущей версии PostgreSQL оптимизатор сам позаботится об этой перезаписи, но ее может потребоваться сделать вручную в более старых версиях.

Рассмотрим другой пример. В листинге 6.17 вычисляются те же значения (средняя цена и количество пассажиров) для всех рейсов, вылетающих из аэропорта О'Хара (ORD).

Листинг 6.17 ❖ Выбор нескольких рейсов

```
SELECT flight_id,
       avg_price,
       num_passengers
  FROM (
         SELECT bl.flight_id,
                departure_airport,
                (avg(price))::numeric (7,2) AS avg_price,
                count(DISTINCT passenger_id) AS num_passengers
           FROM booking b
                JOIN booking_leg bl USING (booking_id)
                JOIN flight f USING (flight_id)
                JOIN passenger p USING (booking_id)
          GROUP BY 1,2
       )a
 WHERE departure_airport = 'ORD'
```

План выполнения этого запроса представлен на рис. 6.14. Выполнение запроса занимает около полутора минут. Это большой запрос, и большинство соединений выполняются с использованием алгоритма соединения хешированием. Важно то, что условие для `departure_airport` применяется первым, перед группировкой.

	QUERY PLAN text
1	Subquery Scan on a (cost=899526.09..935621.26 rows=962538 width=26)
2	-> GroupAggregate (cost=899526.09..925995.88 rows=962538 width=30)
3	Group Key: bl.flight_id, f.departure_airport
4	-> Sort (cost=899526.09..901932.43 rows=962538 width=18)
5	Sort Key: bl.flight_id
6	-> Nested Loop (cost=9545.27..784126.73 rows=962538 width=18)
7	Join Filter: (b.booking_id = p.booking_id)
8	-> Nested Loop (cost=9544.84..532480.61 rows=334023 width=26)
9	-> Hash Join (cost=9544.41..367021.97 rows=334023 width=12)
10	Hash Cond: (bl.flight_id = f.flight_id)
11	-> Seq Scan on booking_leg bl (cost=0.00..310506.66 rows=17893566 width=8)
12	-> Hash (cost=9384.99..9384.99 rows=12753 width=8)
13	-> Bitmap Heap Scan on flight f (cost=243.26..9384.99 rows=12753 width=8)
14	Recheck Cond: (departure_airport = 'ORD'::bpchar)
15	-> Bitmap Index Scan on flight_departure_airport (cost=0.00..240.07 rows=12753 width=0)
16	Index Cond: (departure_airport = 'ORD'::bpchar)
17	-> Index Scan using booking_pkey on booking b (cost=0.43..0.50 rows=1 width=14)
18	Index Cond: (booking_id = bl.booking_id)
19	-> Index Scan using passenger_booking_id on passenger p (cost=0.43..0.64 rows=9 width=8)
20	Index Cond: (booking_id = bl.booking_id)

Рис. 6.14 ❖ План выполнения для запроса из листинга 6.17

Однако более сложные условия фильтрации нельзя переместить на уровень группировки. В листинге 6.18 вычисляется та же статистика, но список `flight_id` не передается напрямую, а выбирается из таблицы `booking_leg`.

План выполнения (рис. 6.15) показывает, что сначала выполняется группировка, а затем результат группировки фильтруется. Это означает, что сначала выполняются расчеты для всех рейсов в системе, а затем из них выбирается нужное подмножество. Общее время выполнения этого запроса составляет 10 минут.

Запрос из листинга 6.18 – пример того, что мы называем *пессимизацией* – использование приемов, гарантированно замедляющих выполнение запросов. Легко понять, почему этот запрос написан именно так. Сначала разработчик базы данных выясняет, *как* выполнять определенные вычисления или как выбирать определенные значения, а затем применяет к результату фильтр. Таким образом, он ограничивает оптимизатор определенным порядком операций, который в данном случае не является оптимальным.

Листинг 6.18 ❖ Условие нельзя переместить на уровень группировки

```
SELECT a.flight_id,
       a.avg_price,
       a.num_passengers
  FROM (
         SELECT bl.flight_id,
                departure_airport,
                (avg(price))::numeric (7,2) AS avg_price,
                count(DISTINCT passenger_id) AS num_passengers
           FROM booking b
                JOIN booking_leg bl USING (booking_id)
                JOIN flight f USING (flight_id)
                JOIN passenger p USING (booking_id)
          GROUP BY 1,2
       ) a
 WHERE flight_id IN (
         SELECT flight_id
           FROM flight
          WHERE scheduled_departure BETWEEN '07-03-2020' AND '07-05-2020'
       )
```

	QUERY PLAN text
2	Merge Cond: (bl.flight_id = flight.flight_id)
3	-> GroupAggregate (cost=12664527.22..14069895.72 rows=51104309 width=30)
4	Group Key: bl.flight_id, f.departure_airport
5	-> Sort (cost=12664527.22..12792287.99 rows=51104309 width=18)
6	Sort Key: bl.flight_id, f.departure_airport
7	-> Hash Join (cost=1130162.26..2977281.94 rows=51104309 width=18)
8	Hash Cond: (bl.booking_id = b.booking_id)
9	-> Hash Join (cost=34851.46..700111.61 rows=17828108 width=12)
10	Hash Cond: (bl.flight_id = f.flight_id)
11	-> Seq Scan on booking_leg bl (cost=0.00..476508.08 rows=17828108 width=8)
12	-> Hash (cost=23642.76..23642.76 rows=683176 width=8)
13	-> Seq Scan on flight f (cost=0.00..23642.76 rows=683176 width=8)
14	-> Hash (cost=795826.94..795826.94 rows=16312309 width=22)
15	-> Hash Join (cost=295294.01..795826.94 rows=16312309 width=22)
16	Hash Cond: (p.booking_id = b.booking_id)
17	-> Seq Scan on passenger p (cost=0.00..302486.09 rows=16312309 width=8)
18	-> Hash (cost=196373.67..196373.67 rows=5690667 width=14)
19	-> Seq Scan on booking b (cost=0.00..196373.67 rows=5690667 width=14)
20	-> Sort (cost=14392.62..14411.94 rows=7728 width=4)
21	Sort Key: flight.flight_id
22	-> Bitmap Heap Scan on flight (cost=167.64..13893.55 rows=7728 width=4)
23	Recheck Cond: ((scheduled_departure >= '2020-07-03 00:00:00-05'::timestamp with time zone) AN...
24	-> Bitmap Index Scan on flight_scheduled_departure (cost=0.00..165.71 rows=7728 width=0)
25	Index Cond: ((scheduled_departure >= '2020-07-03 00:00:00-05'::timestamp with time zone) A...

Рис. 6.15 ❖ План выполнения

Вместо этого фильтрацию можно выполнить во внутреннем предложении WHERE. После внесения этого изменения необходимость во вложенной команде SELECT отпадает – см. листинг 6.19.

Листинг 6.19 ❖ Условие перемещено на уровень группировки

```
SELECT bl.flight_id,
       departure_airport,
       (avg(price))::numeric (7,2) AS avg_price,
       count(DISTINCT passenger_id) AS num_passengers
  FROM booking b
       JOIN booking_leg bl USING (booking_id)
       JOIN flight f USING (flight_id)
       JOIN passenger p USING (booking_id)
 WHERE scheduled_departure BETWEEN '07-03-2020' AND '07-05-2020'
 GROUP BY 1,2
```

Время выполнения составляет около одной минуты, а план выполнения представлен на рис. 6.16. Можно обобщить технику, показанную в предыдущем примере.

	QUERY PLAN text
1	GroupAggregate (cost=713523.49..730858.67 rows=630370 width=30)
2	Group Key: bl.flight_id, f.departure_airport
3	-> Sort (cost=713523.49..715099.42 rows=630370 width=18)
4	Sort Key: bl.flight_id, f.departure_airport
5	-> Nested Loop (cost=9231.18..639871.46 rows=630370 width=18)
6	Join Filter: (b.booking_id = p.booking_id)
7	-> Nested Loop (cost=9230.75..475067.40 rows=218753 width=26)
8	-> Hash Join (cost=9230.32..366707.88 rows=218753 width=12)
9	Hash Cond: (bl.flight_id = f.flight_id)
10	-> Seq Scan on booking_leg bl (cost=0.00..310506.66 rows=17893566 width=8)
11	-> Hash (cost=9125.92..9125.92 rows=8352 width=8)
12	-> Bitmap Heap Scan on flight f (cost=178.03..9125.92 rows=8352 width=8)
13	Recheck Cond: ((scheduled_departure >= '2020-07-03 00:00:00-05'::timestamp with time zone) AND (scheduled_departure <= '2020
14	-> Bitmap Index Scan on flight_scheduled_departure (cost=0.00..175.94 rows=8352 width=0)
15	Index Cond: ((scheduled_departure >= '2020-07-03 00:00:00-05'::timestamp with time zone) AND (scheduled_departure <= '2020
16	-> Index Scan using booking_pkey on booking b (cost=0.43..0.50 rows=1 width=14)
17	Index Cond: (booking_id = bl.booking_id)
18	-> Index Scan using passenger_booking_id on passenger p (cost=0.43..0.64 rows=9 width=8)
19	Index Cond: (booking_id = bl.booking_id)

Рис. 6.16 ❖ План выполнения с фильтрацией на уровне группировки

Перед группировкой отфильтруйте строки, не нужные для агрегации.

Даже оптимальное выполнение этого запроса не происходит мгновенно, но это лучшее, чего мы можем добиться. Сейчас самое время напомнить, что

цели оптимизации должны быть реалистичными. Длинный запрос к большому объему данных нельзя выполнить за доли секунды, даже если он выполняется оптимально. Главное – использовать так мало строк, как только возможно, но не меньше того.

Группировка: сначала группируем, затем выбираем

В некоторых случаях порядок действий должен быть противоположным: GROUP BY следует выполнить как можно раньше, а затем выполнить другие операции. Как вы уже догадались, такой порядок желателен, когда группировка уменьшает размер промежуточного набора данных.

Запрос из листинга 6.20 вычисляет количество пассажиров, вылетающих из каждого города, по месяцам. В этом случае невозможно уменьшить количество необходимых строк, так как в расчетах используются все рейсы.

Листинг 6.20 ❖ Расчет количества пассажиров по городу и месяцу

```
SELECT city,
       date_trunc('month', scheduled_departure) AS month,
       count(*) passengers
  FROM airport a
       JOIN flight f ON airport_code = departure_airport
       JOIN booking_leg l ON f.flight_id =l.flight_id
       JOIN boarding_pass b ON b.booking_leg_id = l.booking_leg_id
 GROUP BY 1,2
 ORDER BY 3 DESC
```

Время выполнения этого запроса составляет более 7 минут, а план выполнения показан на рис. 6.17.

Выполнение этого запроса можно значительно улучшить, переписав его, как показано в листинге 6.21.

Листинг 6.21 ❖ Переписанный запрос, в котором сначала выполняется группировка

```
SELECT city,
       date_trunc('month', scheduled_departure),
       sum(passengers) passengers
  FROM airport a
       JOIN flight f ON airport_code = departure_airport
       JOIN (
         SELECT flight_id, count(*) passengers
           FROM booking_leg l
                JOIN boarding_pass b USING (booking_leg_id)
          GROUP BY flight_id
       ) cnt USING (flight_id)
 GROUP BY 1,2
 ORDER BY 3 DESC
```

	QUERY PLAN text
1	Sort (cost=11691560.21..11754793.94 rows=25293490 width=24)
2	Sort Key: (count(*)) DESC
3	-> GroupAggregate (cost=6456183.43..7025286.95 rows=25293490 width=24)
4	Group Key: a.city, (date_trunc('month'::text, f.scheduled_departure))
5	-> Sort (cost=6456183.43..6519417.15 rows=25293490 width=16)
6	Sort Key: a.city, (date_trunc('month'::text, f.scheduled_departure))
7	-> Hash Join (cost=804544.88..2049270.66 rows=25293490 width=16)
8	Hash Cond: (f.departure_airport = a.airport_code)
9	-> Hash Join (cost=804519.89..1919194.04 rows=25293490 width=12)
10	Hash Cond: (l.flight_id = f.flight_id)
11	-> Hash Join (cost=769001.43..1616337.75 rows=25293490 width=4)
12	Hash Cond: (b.booking_leg_id = l.booking_leg_id)
13	-> Seq Scan on boarding_pass b (cost=0.00..513692.90 rows=25293490 width=8)
14	-> Hash (cost=476508.08..476508.08 rows=17828108 width=8)
15	-> Seq Scan on booking_leg l (cost=0.00..476508.08 rows=17828108 width=8)
16	-> Hash (cost=23642.76..23642.76 rows=683176 width=16)
17	-> Seq Scan on flight f (cost=0.00..23642.76 rows=683176 width=16)
18	-> Hash (cost=16.66..16.66 rows=666 width=12)
19	-> Seq Scan on airport a (cost=0.00..16.66 rows=666 width=12)

Рис. 6.17 ❖ План выполнения с группировкой в конце

Что здесь происходит? Сначала количество вылетающих пассажиров суммируется для каждого рейса во вложенном представлении `cnt`. После этого результат объединяется с таблицей `flight` для получения кода аэропорта, а затем соединяется с таблицей `airport`, чтобы найти город, в котором расположен каждый аэропорт. После этого суммы пассажиров по рейсам суммируются по городам. Таким образом время выполнения составляет 2,5 минуты. План выполнения показан на рис. 6.18.

Использование операций над множествами

Операции над множествами редко используются в SQL-запросах. Однако для больших запросов эти операции могут побудить оптимизатор выбрать более эффективные алгоритмы.

Иногда операции над множествами позволяют выбрать эффективный альтернативный план выполнения и повысить удобочитаемость.

Часто можно:

- использовать `EXCEPT` вместо `NOT EXISTS` и `NOT IN`;
- использовать `INTERSECT` вместо `EXISTS` и `IN`;
- использовать `UNION` вместо сложных критериев выбора с `OR`.

Иногда может наблюдаться значительный прирост производительности, а иной раз время выполнения изменяется лишь незначительно, но код ста-

новится чище и проще в сопровождении. В листинге 6.22 показан переписанный запрос из листинга 6.8, возвращающий рейсы без бронирований.

	QUERY PLAN text
3	-> GroupAggregate (cost=5503000.88..5508006.90 rows=200241 width=48)
4	Group Key: a.city, (date_trunc('month'::text, f.scheduled_departure))
5	-> Sort (cost=5503000.88..5503501.48 rows=200241 width=24)
6	Sort Key: a.city, (date_trunc('month'::text, f.scheduled_departure))
7	-> Hash Join (cost=5280706.05..5481259.28 rows=200241 width=24)
8	Hash Cond: (f.departure_airport = a.airport_code)
9	-> Hash Join (cost=5280681.07..5480204.71 rows=200241 width=20)
10	Hash Cond: (cnt.flight_id = f.flight_id)
11	-> Subquery Scan on cnt (cost=5253406.61..5447112.62 rows=200241 width=12)
12	-> GroupAggregate (cost=5253406.61..5445110.21 rows=200241 width=12)
13	Group Key: l.flight_id
14	-> Sort (cost=5253406.61..5316640.34 rows=25293492 width=4)
15	Sort Key: l.flight_id
16	-> Hash Join (cost=604073.24..1451664.58 rows=25293492 width=4)
17	Hash Cond: (b.booking_leg_id = l.booking_leg_id)
18	-> Seq Scan on boarding_pass b (cost=0.00..513692.92 rows=25293492 width=8)
19	-> Hash (cost=310506.66..310506.66 rows=17893566 width=8)
20	-> Seq Scan on booking_leg l (cost=0.00..310506.66 rows=17893566 width=8)
21	-> Hash (cost=15398.76..15398.76 rows=683176 width=16)
22	-> Seq Scan on flight f (cost=0.00..15398.76 rows=683176 width=16)
23	-> Hash (cost=16.66..16.66 rows=666 width=12)
24	-> Seq Scan on airport a (cost=0.00..16.66 rows=666 width=12)

Рис. 6.18 ❖ План выполнения с группировкой в начале

Листинг 6.22 ❖ Использование EXCEPT вместо NOT IN

```
SELECT flight_id FROM flight f
EXCEPT
SELECT flight_id FROM booking_leg
```

Время выполнения составляет одну минуту и три секунды, что почти в два раза быстрее, чем с антисоединением. План выполнения с операцией `EXCEPT` показан на рис. 6.19.

В листинге 6.23 показан переписанный с использованием операций над множествами запрос из листинга 6.4, выбирающий все рейсы с хотя бы одним бронированием.

Листинг 6.23 ❖ Использование INTERSECT вместо IN

```
SELECT flight_id FROM flight f
INTERSECT
SELECT flight_id FROM booking_leg
```

	QUERY PLAN Read-only column
1	SetOp Except (cost=3518478.09..3611034.51 rows=683176 width=8)
2	-> Sort (cost=3518478.09..3564756.30 rows=18511284 width=8)
3	Sort Key: "*SELECT* 1".flight_id
4	-> Append (cost=0.00..777820.10 rows=18511284 width=8)
5	-> Subquery Scan on "*SELECT* 1" (cost=0.00..30474.52 rows=683176 width=8)
6	-> Seq Scan on flight f (cost=0.00..23642.76 rows=683176 width=4)
7	-> Subquery Scan on "*SELECT* 2" (cost=0.00..654789.16 rows=17828108 width=8)
8	-> Seq Scan on booking_leg (cost=0.00..476508.08 rows=17828108 width=4)

Рис. 6.19 ❖ План выполнения с EXCEPT

Время выполнения этого запроса составляет 49 секунд. Это меньше, чем версия запроса с ключевым словом `IN`, и примерно равно времени выполнения запроса со сканированием только индекса (*см.* листинг 6.10). План выполнения показан на рис. 6.20.

	QUERY PLAN text
1	HashSetOp Intersect (cost=0.00..824098.31 rows=139576 width=8)
2	-> Append (cost=0.00..777820.10 rows=18511284 width=8)
3	-> Subquery Scan on "*SELECT* 2" (cost=0.00..654789.16 rows=17828108 width=8)
4	-> Seq Scan on booking_leg (cost=0.00..476508.08 rows=17828108 width=4)
5	-> Subquery Scan on "*SELECT* 1" (cost=0.00..30474.52 rows=683176 width=8)
6	-> Seq Scan on flight f (cost=0.00..23642.76 rows=683176 width=4)

Рис. 6.20 ❖ План выполнения с INTERSECT

Нам редко приходится переписывать сложные критерии выбора с помощью `OR` в `UNION ALL`, потому что в большинстве случаев оптимизатор PostgreSQL справляется с анализом таких критериев и использованием всех подходящих индексов. Однако иногда такое переписывание делает код более удобным для сопровождения, особенно когда запрос содержит большое количество различных критериев выбора, связанных `OR`. Листинг 6.24 представляет запрос, который вычисляет количество пассажиров на задержанных рейсах из аэропорта Франкфурта-на-Майне (FRA) с использованием двух различных наборов условий. Первая группа – это пассажиры рейсов, задержанных более чем на час, при изменении в посадочном талоне более чем через 30 минут после вылета по расписанию. Второй – это пассажиры рейсов, задержанных более чем на полчаса, но менее чем на час.

Перезапись этого запроса с использованием `UNION ALL` показана в листинге 6.25. Разница во времени выполнения незначительна (около трех секунд), но код более удобен для сопровождения.

Листинг 6.24 ❖ Запрос со сложными условиями с OR

```
SELECT CASE
         WHEN actual_departure > scheduled_departure + interval '1 hour'
           THEN 'Late group 1'
         ELSE 'Late group 2'
       END AS grouping,
       flight_id,
       count(*) AS num_passengers
  FROM boarding_pass bp
       JOIN booking_leg bl USING (booking_leg_id)
       JOIN booking b USING (booking_id)
       JOIN flight f USING (flight_id)
 WHERE departure_airport = 'FRA'
   AND actual_departure > '2020-07-01'
   AND (
         (     actual_departure > scheduled_departure + interval '30 minute'
           AND actual_departure <= scheduled_departure + interval '1 hour'
         )
         OR
         (     actual_departure>scheduled_departure + interval '1 hour'
           AND bp.update_ts > scheduled_departure + interval '30 minute'
         )
       )
 GROUP BY 1,2
```

Листинг 6.25 ❖ Сложное условие с OR переписано, используя UNION ALL

```
SELECT 'Late group 1' AS grouping,
       flight_id,
       count(*) AS num_passengers
  FROM boarding_pass bp
       JOIN booking_leg bl USING (booking_leg_id)
       JOIN booking b USING (booking_id)
       JOIN flight f USING (flight_id)
 WHERE departure_airport = 'FRA'
   AND actual_departure > scheduled_departure + interval '1 hour'
   AND bp.update_ts > scheduled_departure + interval '30 minutes'
   AND actual_departure > '2020-07-01'
 GROUP BY 1,2
UNION ALL
SELECT 'Late group 2' AS grouping,
       flight_id,
       count(*) AS num_passengers
  FROM boarding_pass bp
       JOIN booking_leg bl USING(booking_leg_id)
       JOIN booking b USING (booking_id)
       JOIN flight f USING (flight_id)
 WHERE departure_airport = 'FRA'
   AND actual_departure > scheduled_departure + interval '30 minute'
   AND actual_departure <= scheduled_departure + interval '1 hour'
   AND actual_departure > '2020-07-01'
 GROUP BY 1,2
```

Стоит отметить, что с большими запросами всегда нужно учитывать, какой оперативной памятью вы располагаете. Скорость выполнения как соединений хешированием, так и операций со множествами значительно уменьшается, если участвующие наборы данных не помещаются в оперативную память.

Избегаем многократного сканирования

Еще одна причина медленного выполнения запросов – это многократные сканирования таблиц. Эта распространенная проблема является прямым результатом неудачного проектирования схемы данных. Схему теоретически можно исправить. Но поскольку часто мы оказываемся в ситуациях, когда не можем этого сделать, мы предложим способы написания эффективных запросов даже при несовершенной схеме.

Ситуация, которую мы моделируем в нашей схеме `postgres_air`, не редкость в реальном мире. Система уже запущена и работает, и внезапно нам потребовалось сохранять некую дополнительную информацию для объектов, которые уже присутствуют в базе данных.

В течение последних 30 лет самым простым решением в таких случаях было использование таблицы *сущность–атрибут–значение* (entity-attribute-value, EAV), в которой могут храниться произвольные атрибуты – те, которые нужны сейчас, и те, которые понадобятся когда-нибудь. В схеме `postgres_air` этот шаблон реализован в таблице `custom_field`. Здесь хранятся номер паспорта каждого пассажира, срок действия паспорта и название страны, выдавшей его. Атрибуты соответственно именуются `passport_num`, `passport_exp_date` и `passport_country`.

Эта таблица не входит в дистрибутив postgres_air. Чтобы запустить пример локально, выполните следующий скрипт из репозитория postgres_air на GitHub:

https://github.com/hettie-d/postgres_air/blob/main/tables/custom_field.sql.

Теперь представьте, что требуется отчет, показывающий имена пассажиров и их паспортные данные. Листинг 6.26 представляет собой типичное предлагаемое решение: таблица `custom_field` сканируется трижды! Во избежание сброса промежуточных данных во временные файлы количество пассажиров ограничено первыми пятью миллионами, что позволяет показать истинное соотношение времен выполнения. План выполнения на рис. 6.21 подтверждает три сканирования таблицы, а время выполнения этого запроса составляет 5 минут.

Трехкратное сканирование этой таблицы напоминает такой способ разложить яблоки, апельсины и лимоны из черного ящика по ведрам, при котором сначала в одно ведро откладываются все яблоки (а апельсины с лимонами остаются в ящике), затем в другое ведро откладываются апельсины, и наконец в третье ведро из ящика перекладываются лимоны. Более эффективный

способ выполнить эту работу – поставить все три ведра перед собой и, вынимая фрукт из черного ящика, сразу класть его в правильное ведро.

Листинг 6.26 ❖ Многократные сканирования большой таблицы

```
SELECT first_name,
       last_name,
       pn.custom_field_value AS passport_num,
       pe.custom_field_value AS passport_exp_date,
       pc.custom_field_value AS passport_country
  FROM passenger p
       JOIN custom_field pn ON pn.passenger_id = p.passenger_id
                           AND pn.custom_field_name = 'passport_num'
       JOIN custom_field pe ON pe.passenger_id = p.passenger_id
                           AND pe.custom_field_name = 'passport_exp_date'
       JOIN custom_field pc ON pc.passenger_id = p.passenger_id
                           AND pc.custom_field_name = 'passport_country'
 WHERE p.passenger_id < 5000000
```

	QUERY PLAN text
1	Hash Join (cost=3344203.17..4769951.67 rows=4306126 width=51)
2	Hash Cond: (pe.passenger_id = pn.passenger_id)
3	-> Seq Scan on custom_field pe (cost=0.00..1022564.75 rows=16454200 width=17)
4	Filter: (custom_field_name = 'passport_exp_date'::text)
5	-> Hash (cost=3244130.53..3244130.53 rows=4494451 width=50)
6	-> Hash Join (cost=1806471.88..3244130.53 rows=4494451 width=50)
7	Hash Cond: (pc.passenger_id = pn.passenger_id)
8	-> Seq Scan on custom_field pc (cost=0.00..1022564.75 rows=16273188 width=17)
9	Filter: (custom_field_name = 'passport_country'::text)
10	-> Hash (cost=1710124.97..1710124.97 rows=4743193 width=33)
11	-> Hash Join (cost=430701.56..1710124.97 rows=4743193 width=33)
12	Hash Cond: (pn.passenger_id = p.passenger_id)
13	-> Seq Scan on custom_field pn (cost=0.00..1022564.75 rows=16194911 width=17)
14	Filter: (custom_field_name = 'passport_num'::text)
15	-> Hash (cost=343266.86..343266.86 rows=5029896 width=16)
16	-> Seq Scan on passenger p (cost=0.00..343266.86 rows=5029896 width=16)
17	Filter: (passenger_id < 5000000)

Рис. 6.21 ❖ План выполнения с многократными сканированиями

При извлечении нескольких атрибутов из таблицы *сущность–атрибут–значение* выполняйте соединение с таблицей только один раз и используйте операторы `CASE` в списке `SELECT` для получения нужных значений в каждом столбце.

Чтобы применить этот прием к таблице `custom_field`, запрос можно переписать, как показано в листинге 6.27.

Листинг 6.27 ❖ Одно сканирование таблицы для получения нескольких атрибутов

```
SELECT last_name,
       first_name,
       coalesce(max(CASE WHEN custom_field_name = 'passport_num'
                         THEN custom_field_value ELSE NULL
                    END),'') AS passport_num,
       coalesce(max(CASE WHEN custom_field_name = 'passport_exp_date'
                         THEN custom_field_value ELSE NULL
                    END),'') AS passport_exp_date,
       coalesce(max(CASE WHEN custom_field_name = 'passport_country'
                         THEN custom_field_value ELSE NULL
                    END),'') AS passport_country
  FROM passenger p
       JOIN custom_field cf USING (passenger_id)
 WHERE cf.passenger_id < 5000000
   AND p.passenger_id < 5000000
 GROUP BY 1,2
```

План выполнения для запроса из листинга 6.27 показан на рис. 6.22.

	QUERY PLAN text
1	GroupAggregate (cost=2504788.89..2619372.25 rows=1590363 width=108)
2	Group Key: p.last_name, p.first_name
3	-> Sort (cost=2504788.89..2515753.30 rows=4385766 width=41)
4	Sort Key: p.last_name, p.first_name
5	-> Hash Join (cost=430701.56..1751113.47 rows=4385766 width=41)
6	Hash Cond: (cf.passenger_id = p.passenger_id)
7	-> Seq Scan on custom_field cf (cost=0.00..1022564.75 rows=14974531 width=33)
8	Filter: (passenger_id < 5000000)
9	-> Hash (cost=343266.86..343266.86 rows=5029896 width=16)
10	-> Seq Scan on passenger p (cost=0.00..343266.86 rows=5029896 width=16)
11	Filter: (passenger_id < 5000000)

Рис. 6.22 ❖ Одно сканирование таблицы

Одно сканирование таблицы выглядит лучше, вот только выполняться запрос будет значительно дольше. Если присмотреться, то понятно, почему: пассажиров с одинаковыми именами и фамилиями может быть много, поэтому выполнение не только занимает больше времени, но и дает неправильный результат. Давайте еще раз изменим запрос – см. листинг 6.28.

Листинг 6.28 ❖ Исправление запроса из листинга 6.27

```
SELECT last_name,
       first_name,
       p.passenger_id,
       coalesce(max(CASE WHEN custom_field_name = 'passport_num'
                         THEN custom_field_value ELSE NULL
                    END),'') AS passport_num,
       coalesce(max(CASE WHEN custom_field_name = 'passport_exp_date'
                         THEN custom_field_value ELSE NULL
                    END),'') AS passport_exp_date,
       coalesce(max(CASE WHEN custom_field_name = 'passport_country'
                         THEN custom_field_value ELSE NULL
                    END),'') AS passport_country
  FROM passenger p
       JOIN custom_field cf USING (passenger_id)
 WHERE cf.passenger_id < 5000000
   AND p.passenger_id < 5000000
 GROUP BY 3,1,2
```

План выполнения на рис. 6.23 выглядит намного лучше – теперь группировка выполняется по столбцу `passenger_id`.

Data Output | Explain | Messages | Notifications

	QUERY PLAN text
1	GroupAggregate (cost=2504788.89..2636361.87 rows=4385766 width=112)
2	Group Key: p.passenger_id
3	-> Sort (cost=2504788.89..2515753.30 rows=4385766 width=45)
4	Sort Key: p.passenger_id
5	-> Hash Join (cost=430701.56..1751113.47 rows=4385766 width=45)
6	Hash Cond: (cf.passenger_id = p.passenger_id)
7	-> Seq Scan on custom_field cf (cost=0.00..1022564.75 rows=14974531 width=33)
8	Filter: (passenger_id < 5000000)
9	-> Hash (cost=343266.86..343266.86 rows=5029896 width=16)
10	-> Seq Scan on passenger p (cost=0.00..343266.86 rows=5029896 width=16)
11	Filter: (passenger_id < 5000000)

Рис. 6.23 ❖ План выполнения для запроса из листинга 6.28

Здесь нужно выполнить еще одну оптимизацию; атрибуты из таблицы EAV часто соединяются с другими таблицами, и можно уменьшить размер промежуточного набора результатов, «свернув» и отфильтровав эту таблицу до необходимых значений перед выполнением других соединений.

Это более частный случай обобщенной техники группировки перед соединением.

Выносите значения из таблицы EAV в подзапрос перед соединением с другими таблицами.

Проделав это для примера с паспортом, получаем запрос, показанный в листинге 6.29.

Листинг 6.29 ❖ Перенос группировки в подзапрос

```
SELECT last_name,
       first_name,
       passport_num,
       passport_exp_date,
       passport_country
  FROM passenger p
       JOIN (
             SELECT cf.passenger_id,
                    coalesce(max(CASE WHEN custom_field_name = 'passport_num'
                                      THEN custom_field_value ELSE NULL
                                 END),'') AS passport_num,
                    coalesce(max(CASE WHEN custom_field_name = 'passport_exp_date'
                                      THEN custom_field_value ELSE NULL
                                 END),'') AS passport_exp_date,
                    coalesce(max(CASE WHEN custom_field_name = 'passport_country'
                                      THEN custom_field_value ELSE NULL
                                 END),'') AS passport_country
               FROM custom_field cf
              WHERE cf.passenger_id < 5000000
              GROUP BY 1
            ) info USING (passenger_id)
 WHERE p.passenger_id < 5000000
```

План выполнения показан на рис. 6.24.

	QUERY PLAN text
1	Merge Join (cost=4035614.85..5308045.32 rows=3425544 width=108)
2	Merge Cond: (p.passenger_id = cf.passenger_id)
3	-> Index Scan using passenger_pkey on passenger p (cost=0.43..676151.28 rows=5029896 width=16)
4	Index Cond: (passenger_id < 5000000)
5	-> GroupAggregate (cost=4035614.41..4446197.84 rows=11109281 width=100)
6	Group Key: cf.passenger_id
7	-> Sort (cost=4035614.41..4073050.74 rows=14974531 width=33)
8	Sort Key: cf.passenger_id
9	-> Seq Scan on custom_field cf (cost=0.00..1022564.75 rows=14974531 width=33)
10	Filter: (passenger_id < 5000000)

Рис. 6.24 ❖ План выполнения с группировкой, перенесенной в подзапрос

Выводы

В этой главе формально определены длинные запросы и рассмотрены методы их оптимизации.

Первое важное положение этой главы состоит в том, что индексы не обязательно ускоряют выполнение запросов, а фактически могут замедлить выполнение длинных запросов. Распространенное заблуждение состоит в том, что если невозможно построить индекс, то ничего нельзя сделать для оптимизации полного сканирования таблицы. Надеемся, эта глава убедительно продемонстрировала, что для оптимизации полного сканирования существует множество возможностей.

Как и короткие запросы, длинные запросы оптимизируются за счет уменьшения размера промежуточных результатов и выполнения необходимой работы с как можно меньшим количеством строк. В случае с короткими запросами это достигается путем применения индексов по наиболее ограничительным критериям. В случае с длинными запросами это достигается за счет выбора правильного порядка соединений, использования полу- и антисоединений, применения фильтрации перед группировкой и группировки перед соединением, а также использования операций над множествами.

Глава 7

Длинные запросы: дополнительные приемы

В главе 6 обсуждались способы повышения производительности длинных запросов. До сих пор все описанные методы относились к переписыванию запросов без создания каких-либо дополнительных объектов базы данных. В этой главе рассматриваются дополнительные приемы улучшения производительности длинных запросов, включая различные способы материализации промежуточных результатов. Временные таблицы, общие табличные выражения, представления и материализованные представления – каждый из этих инструментов может быть полезен для повышения производительности, но при неправильном использовании может привести и к снижению. Наконец, в этой главе рассказывается о секционировании и параллельном выполнении.

Структурирование запросов

Те из вас, кто знаком с объектно-ориентированным программированием (ООП), знакомы и с концепциями декомпозиции и инкапсуляции. Лучшие практики ООП предписывают разбивать код на множество более мелких классов и объектов, отвечающих за четко определенную часть поведения системы, а также инкапсулировать логику, ограничивая прямой доступ к компонентам и, таким образом, скрывая их реализацию. Эти два принципа облегчают чтение кода приложения и его сопровождение, а также упрощают внесение изменений.

Исходя из этой парадигмы, когда кто-то сталкивается с запросом из пяти сотен строк, он испытывает понятный соблазн применить те же принципы, чтобы разбить код на более мелкие части и инкапсулировать часть логики.

Однако декларативный характер SQL диктует совершенно иной стиль декомпозиции запросов, нежели тот, который использовался бы для кода приложения. Код SQL, как и код на любом другом языке, должен быть простым для понимания и изменения, но не за счет производительности.

В SQL можно подходить к декомпозиции и инкапсуляции разными способами, каждый из которых имеет свои преимущества и недостатки. Некоторые из них используются (с разной степенью эффективностьи) для улучшения производительности и хранения промежуточных результатов. Другие используются, чтобы код можно было использовать повторно. Третьи влияют на способ хранения данных. В этой главе рассматривается только несколько подходов, а другие, например функции, будут подробно рассмотрены в последующих главах.

В любом случае любая декомпозиция или инкапсуляция должны соответствовать логической сущности, например отчету или ежедневному обновлению.

Временные таблицы и общие табличные выражения

В главе 6 мы упоминали, что иногда попытка разработчиков SQL ускорить выполнение запроса, наоборот, может привести к замедлению. Такое часто случается, когда они решают использовать *временные таблицы*.

Временные таблицы

Чтобы создать временную таблицу, нужно выполнить обычную команду `create table`, добавив ключевое слово `temporary` или просто `temp`:

```
CREATE TEMP TABLE interim_results
```

Временные таблицы видны только текущему сеансу и удаляются при его завершении, если не были явно удалены до этого. В остальном они ничем не уступают обычным таблицам: их можно использовать в запросах без всяких ограничений и даже можно индексировать. Временные таблицы часто используются для хранения промежуточных результатов запросов, поэтому команда `CREATE` нередко выглядит так:

```
CREATE TEMP TABLE interim_results AS
    SELECT ...
```

Это смотрится очень удобно, так что же плохого в таком подходе?

Все это отлично работает, если вы сохраняете результаты своего запроса во временной таблице для некоего анализа, а затем удаляете ее. Но если использовать временные таблицы для хранения результатов каждого шага, код начинает выглядеть так:

```
CREATE TEMP TABLE T1 AS
    SELECT ...;
CREATE TEMP TABLE T2 AS
    SELECT ...
      FROM T1 INNER JOIN ...
...
```

Цепочка временных таблиц может стать довольно длинной. Вызывает ли это какие-то проблемы? Да, и много, в том числе следующие:

- *индексы* – после того как выбранные данные будут сохранены во временной таблице, мы не можем использовать индексы, созданные в исходной таблице (или таблицах). Нам либо придется обойтись без индексов, либо создать новые во временных таблицах, что требует ресурсов;
- *статистика* – поскольку мы создали новую таблицу, оптимизатор не может использовать статистические данные о распределении значений из исходной таблицы (или таблиц), поэтому нам придется либо обойтись без статистики, либо выполнить команду `ANALYZE` для временной таблицы;
- *место на диске* – когда промежуточные результаты не помещаются в доступную оперативную память, временные таблицы сохраняются на диске. Как бы маловероятно это ни звучало, мы наблюдали ситуации, когда большие запросы, использующие соединения, сортировки и группировки, конкурировали за пространство с временными таблицами, в результате чего запросы отменялись;
- *чрезмерный ввод-вывод* – временные таблицы – это таблицы, и они могут быть записаны на диск, а для записи на диск и чтения с диска требуется дополнительное время.

Но наиболее важным отрицательным следствием чрезмерного использования временных таблиц является то, что такая практика не дает оптимизатору переписывать запросы.

Сохраняя результаты каждого соединения во временную таблицу, вы не позволяете оптимизатору выбирать оптимальный порядок соединений; вы фиксируете тот порядок, в котором создаете временные таблицы.

Когда мы рассматривали план выполнения запроса из листинга 6.15, то заметили, что PostgreSQL может переместить условие фильтрации на уровень группировки. Что произойдет, если для промежуточных результатов будет создана временная таблица?

Листинг 7.1 ❖ Неэффективное использование временных таблиц

```
CREATE TEMP TABLE flights_totals AS
SELECT bl.flight_id,
       departure_airport,
       (avg(price))::numeric(7,2) AS avg_price,
       count(DISTINCT passenger_id) AS num_passengers
  FROM booking b
       JOIN booking_leg bl USING (booking_id)
       JOIN flight f USING (flight_id)
       JOIN passenger p USING (booking_id)
 GROUP BY 1,2;

SELECT flight_id,
       avg_price,
       num_passengers
  FROM flights_totals
 WHERE departure_airport = 'ORD'
```

Создание временной таблицы заняло 15 минут и дало более 500 000 строк, из которых нам нужно всего 10 000. В то же время для выполнения запроса из листинга 6.15 потребовалось чуть больше минуты.

Общие табличные выражения (CTE)

Если временные таблицы могут так навредить, нельзя ли использовать вместо них *общие табличные выражения* (common table expression, CTE)? Разберемся для начала, что это такое.

Общие табличные выражения можно рассматривать как временные таблицы, существующие только для одного запроса. В предложении `WITH` могут использоваться команды `SELECT`, `INSERT`, `UPDATE` или `DELETE`, а само предложение `WITH` присоединяется к основному оператору, который также может быть командой `SELECT`, `INSERT`, `UPDATE` или `DELETE`.

Давайте попробуем применить CTE. В листинге 7.2 запрос из листинга 7.1 изменен, чтобы использовать общее табличное выражение вместо временной таблицы.

Листинг 7.2 ❖ Пример запроса с общим табличным выражением

```
WITH flights_totals AS (
    SELECT bl.flight_id,
           departure_airport,
           (avg(price))::numeric(7,2) AS avg_price,
           count(DISTINCT passenger_id) AS num_passengers
      FROM booking b
           JOIN booking_leg bl USING (booking_id)
           JOIN flight f USING (flight_id)
           JOIN passenger p USING (booking_id)
     GROUP BY 1,2
)
SELECT flight_id,
       avg_price,
       num_passengers
  FROM flights_totals
 WHERE departure_airport = 'ORD'
```

То, что вы увидите в плане выполнения, зависит от того, какую версию PostgreSQL вы используете. Для всех версий ниже 12 общее табличное выражение обрабатывалось точно так же, как временная таблица. Результаты материализовались в основной памяти с возможным сбросом на диск. Это означает, что использование CTE не имело никаких преимуществ перед временной таблицей.

Вообще-то CTE предполагалось использовать для другого. Идея была в том, что если какой-то вложенный подзапрос используется в запросе несколько раз, то его можно определить как общее табличное выражение и ссылаться

на него столько раз, сколько потребуется. В этом случае PostgreSQL вычислит результаты только один раз и повторно использует их при повторных обращениях.

Из-за такого предполагаемого применения оптимизатор планировал выполнение CTE отдельно от остальной части запроса и не перемещал никакие условия соединения внутрь общего табличного выражения, образуя *оптимизации*. Это особенно важно, если `WITH` используется в командах `INSERT`, `DELETE` или `UPDATE`, которые могут иметь побочные эффекты, или в рекурсивных вызовах CTE. Кроме того, наличие барьера оптимизации означает, что таблицы, участвующие в общем табличном выражении, не входят в ограничение, устанавливаемое параметром `join_collapse_limit`. Таким образом, мы можем использовать оптимизатор PostgreSQL для запросов, соединяющих большое количество таблиц.

Для запроса из листинга 7.2 в версиях PostgreSQL до 12 общее табличное выражение `flight_totals` будет рассчитано для всех рейсов, и только после этого будет выбрано подмножество рейсов.

PostgreSQL 12 радикально изменил оптимизацию общих табличных выражений. Если нерекурсивное CTE используется в запросе только один раз, то барьер оптимизации снимается и CTE встраивается во внешний запрос. Если CTE вызывается несколько раз, то сохраняется старое поведение.

Описанное поведение используется по умолчанию, но его можно изменить, используя ключевые слова `MATERIALIZED` и `NOT MATERIALIZED` (см. листинг 7.3). Первое ключевое слово вызывает старое поведение, а второе – встраивание, независимо от всех других соображений.

Листинг 7.3 ❖ Использование ключевого слова MATERIALIZED

```
WITH flights_totals AS MATERIALIZED (
    SELECT bl.flight_id,
           departure_airport,
           (avg(price))::numeric(7,2) AS avg_price,
           count(DISTINCT passenger_id) AS num_passengers
      FROM booking b
           JOIN booking_leg bl USING (booking_id)
           JOIN flight f USING (flight_id)
           JOIN passenger p USING (booking_id)
     GROUP BY 1,2
)
SELECT flight_id,
       avg_price,
       num_passengers
  FROM flights_totals
 WHERE departure_airport = 'ORD'
```

На рис. 7.1 представлен план выполнения для запроса из листинга 7.2, как он работает в PostgreSQL 12. Если добавить ключевое слово `MATERIALIZE`, как в листинге 7.3, запрос выполняется по-старому, как показано на рис. 7.2.

	QUERY PLAN text
1	Subquery Scan on flights_totals (cost=899526.09..935621.26 rows=962538 width=26)
2	-> GroupAggregate (cost=899526.09..925995.88 rows=962538 width=30)
3	Group Key: bl.flight_id, f.departure_airport
4	-> Sort (cost=899526.09..901932.43 rows=962538 width=18)
5	Sort Key: bl.flight_id
6	-> Nested Loop (cost=9545.27..784126.73 rows=962538 width=18)
7	Join Filter: (b.booking_id = p.booking_id)
8	-> Nested Loop (cost=9544.84..532480.61 rows=334023 width=26)
9	-> Hash Join (cost=9544.41..367021.97 rows=334023 width=12)
10	Hash Cond: (bl.flight_id = f.flight_id)
11	-> Seq Scan on booking_leg bl (cost=0.00..310506.66 rows=17893566 width=8)
12	-> Hash (cost=9384.99..9384.99 rows=12753 width=8)
13	-> Bitmap Heap Scan on flight f (cost=243.26..9384.99 rows=12753 width=8)
14	Recheck Cond: (departure_airport = 'ORD'::bpchar)
15	-> Bitmap Index Scan on flight_departure_airport (cost=0.00..240.07 rows=12753 width=0)
16	Index Cond: (departure_airport = 'ORD'::bpchar)
17	-> Index Scan using booking_pkey on booking b (cost=0.43..0.50 rows=1 width=14)
18	Index Cond: (booking_id = bl.booking_id)
19	-> Index Scan using passenger_booking_id on passenger p (cost=0.43..0.64 rows=9 width=8)
20	Index Cond: (booking_id = bl.booking_id)

Рис. 7.1 ❖ План выполнения для CTE со встраиванием

До этих недавних изменений мы бы отговаривали разработчиков SQL от использования нескольких общих табличных выражений, при котором инструкция SQL выглядит так:

```
WITH x AS (
    SELECT ...
), y AS (
    SELECT ... FROM t1 JOIN x
), z AS (
    SELECT...
)
SELECT ...
  FROM (
        SELECT
               (
                SELECT ...
                  FROM c JOIN y ...
               ) b
       ) a JOIN z ...
```

Однако после изменений, появившихся в PostgreSQL 12, такие запросы стали намного более управляемыми. Мы по-прежнему призываем разработчиков SQL не навязывать неоптимальный план выполнения, но использование цепочки

общих табличных выражений намного лучше, чем использование последовательности временных таблиц; в последнем случае оптимизатор беспомощен.

	QUERY PLAN text
1	CTE Scan on flights_totals (cost=14081839.16..15242007.67 rows=257815 width=26)
2	Filter: (departure_airport = 'ORD'::bpchar)
3	CTE flights_totals
4	-> GroupAggregate (cost=12663855.42..14081839.16 rows=51563045 width=30)
5	Group Key: bl.flight_id, f.departure_airport
6	-> Sort (cost=12663855.42..12792763.04 rows=51563045 width=18)
7	Sort Key: bl.flight_id, f.departure_airport
8	-> Hash Join (cost=1151589.89..2886328.12 rows=51563045 width=18)
9	Hash Cond: (b.booking_id = bl.booking_id)
10	-> Hash Join (cost=314001.30..846710.40 rows=16313907 width=22)
11	Hash Cond: (p.booking_id = b.booking_id)
12	-> Seq Scan on passenger p (cost=0.00..334787.07 rows=16313907 width=8)
13	-> Hash (cost=215591.02..215591.02 rows=5661302 width=14)
14	-> Seq Scan on booking b (cost=0.00..215591.02 rows=5661302 width=14)
15	-> Hash (cost=526548.02..526548.02 rows=17893566 width=12)
16	-> Hash Join (cost=26607.46..526548.02 rows=17893566 width=12)
17	Hash Cond: (bl.flight_id = f.flight_id)
18	-> Seq Scan on booking_leg bl (cost=0.00..310506.66 rows=17893566 width=8)
19	-> Hash (cost=15398.76..15398.76 rows=683176 width=8)
20	-> Seq Scan on flight f (cost=0.00..15398.76 rows=683176 width=8)

Рис. 7.2 ❖ Принудительная материализация CTE

В заключение этого раздела хотим упомянуть, что существуют ситуации, когда полезно сохранять промежуточные результаты. Однако почти всегда есть способы лучше, чем использование временных таблиц. Мы обсудим другие варианты позже в этой главе.

Представления: использовать или не использовать

Представления – самый противоречивый объект базы данных. Они кажутся простыми для понимания, а преимущества создания представления выглядит очевидными. Почему же они могут приводить к проблемам?

Хотя мы уверены, что большинство читателей создавали в своей практике хотя бы пару представлений, давайте дадим формальное определение. Вот самое простое:

Представление – это объект базы данных, в котором хранится запрос, определяющий виртуальную таблицу.

Представление – это виртуальная таблица в том смысле, что синтаксически представления могут использоваться в запросах так же, как и таблицы. Однако они значительно отличаются от таблиц тем, что не хранят данные; в базе сохраняется только запрос, определяющий представление.

Посмотрим еще раз на запрос из листинга 6.14. Этот запрос вычисляет показатели для всех рейсов в схеме `postgres_air`, но мы хотим использовать ту же логику расчета для конкретных рейсов, или отдельных аэропортов вылета, или для того и другого. Листинг 7.4 создает представление, которое инкапсулирует эту логику.

Листинг 7.4 ❖ Создаем представление

```
CREATE VIEW flight_stats AS
SELECT bl.flight_id,
       departure_airport,
       (avg(price))::numeric(7,2) AS avg_price,
       count(DISTINCT passenger_id) AS num_passengers
  FROM booking b
       JOIN booking_leg bl USING (booking_id)
       JOIN flight f USING (flight_id)
       JOIN passenger p USING (booking_id)
 GROUP BY 1,2
```

Теперь можно легко получить статистику для любого конкретного рейса:

```
SELECT *
  FROM flight_stats
 WHERE flight_id = 222183
```

Этот план запроса идентичен плану на рис. 6.13. Причина состоит в том, что на первом этапе обработки запроса синтаксический анализатор преобразует представления во вложенные подзапросы.

В данном случае это работает в наших интересах, поскольку условие фильтрации помещается на уровень группировки. Но если использовать более сложное условие поиска, чем сравнение с константой, результаты могут быть неутешительными. В листинге 7.5 статистика рейсов из представления `flight_stats` ограничена датой вылета.

Листинг 7.5 ❖ Запрос с использованием представления

```
SELECT *
  FROM flight_stats fs
       JOIN (
              SELECT flight_id
                FROM flight f
               WHERE actual_departure between '2020-08-01' and '2020-08-16'
            ) fl
       ON fl.flight_id = fs.flight_id
```

План выполнения этого запроса показан на рис. 7.3.

	QUERY PLAN text
1	Hash Join (cost=12683989.30..14734550.53 rows=4063504 width=34)
2	Hash Cond: (bl.flight_id = f.flight_id)
3	-> GroupAggregate (cost=12664527.22..14069895.72 rows=51104309 width=30)
4	Group Key: bl.flight_id, f_1.departure_airport
5	-> Sort (cost=12664527.22..12792287.99 rows=51104309 width=18)
6	Sort Key: bl.flight_id, f_1.departure_airport
7	-> Hash Join (cost=1130162.26..2977281.94 rows=51104309 width=18)
8	Hash Cond: (bl.booking_id = b.booking_id)
9	-> Hash Join (cost=34851.46..700111.61 rows=17828108 width=12)
10	Hash Cond: (bl.flight_id = f_1.flight_id)
11	-> Seq Scan on booking_leg bl (cost=0.00..476508.08 rows=17828108 width=8)
12	-> Hash (cost=23642.76..23642.76 rows=683176 width=8)
13	-> Seq Scan on flight f_1 (cost=0.00..23642.76 rows=683176 width=8)
14	-> Hash (cost=795826.94..795826.94 rows=16312309 width=22)
15	-> Hash Join (cost=295294.01..795826.94 rows=16312309 width=22)
16	Hash Cond: (p.booking_id = b.booking_id)
17	-> Seq Scan on passenger p (cost=0.00..302486.09 rows=16312309 width=8)
18	-> Hash (cost=196373.67..196373.67 rows=5690667 width=14)
19	-> Seq Scan on booking b (cost=0.00..196373.67 rows=5690667 width=14)
20	-> Hash (cost=18783.05..18783.05 rows=54322 width=4)
21	-> Bitmap Heap Scan on flight f (cost=1157.22..18783.05 rows=54322 width=4)
22	Recheck Cond: ((actual_departure >= '2020-08-01 00:00:00-05'::timestamp with time zone) AND (actual_departure <= '2020-08-16 00:00:00-05'::timestamp with ...
23	-> Bitmap Index Scan on flight_actual_departure_not_null (cost=0.00..1143.64 rows=54322 width=0)
24	Index Cond: ((actual_departure >= '2020-08-01 00:00:00-05'::timestamp with time zone) AND (actual_departure <= '2020-08-16 00:00:00-05'::timestamp with ...

Рис. 7.3 ❖ План выполнения, в котором условие невозможно переместить

Мы видим, что сначала рассчитывается статистика для всех рейсов, и только после этого результаты соединяются с необходимыми рейсами. Время выполнения этого запроса – 10 минут.

Не используя представление, мы следуем шаблону, описанному в главе 6, фильтруя рейсы перед группировкой, как показано в листинге 7.6.

План выполнения этого запроса показан на рис. 7.4. Здесь видно, что в первую очередь применяются ограничения для таблицы `flight`. Время выполнения этого запроса – три минуты.

Когда учебники по базам данных, в том числе посвященные основам PostgreSQL, утверждают, что представления можно использовать «как таблицы», это вводит в заблуждение. На практике представления, которые изначально были созданы исключительно для инкапсуляции отдельного запроса, часто

используются разработчиками в других запросах, соединяются с другими таблицами и представлениями, в том числе многократно соединяются с таблицами, уже включенными в само представление, – без понимания того, что при этом происходит.

Листинг 7.6 ❖ Переписываем запрос без представления

```
SELECT bl.flight_id,
       departure_airport,
       (avg(price))::numeric(7,2) AS avg_price,
       count(DISTINCT passenger_id) AS num_passengers
  FROM booking b
       JOIN booking_leg bl USING (booking_id)
       JOIN flight f USING (flight_id)
       JOIN passenger p USING (booking_id)
 WHERE actual_departure between '2020-08-01' AND '2020-08-16'
 GROUP BY 1,2
```

	QUERY PLAN text
1	GroupAggregate (cost=2050470.84..2162217.23 rows=4063505 width=30)
2	Group Key: bl.flight_id, f.departure_airport
3	-> Sort (cost=2050470.84..2060629.60 rows=4063505 width=18)
4	Sort Key: bl.flight_id, f.departure_airport
5	-> Hash Join (cost=896320.64..1437743.94 rows=4063505 width=18)
6	Hash Cond: (p.booking_id = b.booking_id)
7	-> Seq Scan on passenger p (cost=0.00..302486.09 rows=16312309 width=8)
8	-> Hash (cost=868909.85..868909.85 rows=1417583 width=26)
9	-> Hash Join (cost=567411.02..868909.85 rows=1417583 width=26)
10	Hash Cond: (b.booking_id = bl.booking_id)
11	-> Seq Scan on booking b (cost=0.00..196373.67 rows=5690667 width=14)
12	-> Hash (cost=542769.23..542769.23 rows=1417583 width=12)
13	-> Hash Join (cost=19462.08..542769.23 rows=1417583 width=12)
14	Hash Cond: (bl.flight_id = f.flight_id)
15	-> Seq Scan on booking_leg bl (cost=0.00..476508.08 rows=17828108 width=8)
16	-> Hash (cost=18783.05..18783.05 rows=54322 width=8)
17	-> Bitmap Heap Scan on flight f (cost=1157.22..18783.05 rows=54322 width=8)
18	Recheck Cond: ((actual_departure >= '2020-08-01 00:00:00-05'::timestamp with time zone) AND (actual_depa...
19	-> Bitmap Index Scan on flight_actual_departure_not_null (cost=0.00..1143.64 rows=54322 width=0)
20	Index Cond: ((actual_departure >= '2020-08-01 00:00:00-05'::timestamp with time zone) AND (actual_depa...

Рис. 7.4 ❖ План выполнения для запроса из листинга 7.6

С одной стороны, представление обычно создают именно для инкапсуляции, чтобы другие могли использовать его, не разбираясь в логике запроса. С другой стороны, эта непрозрачность приводит к низкой производительности. Это особенно ярко проявляется, когда некоторые столбцы в представлении являются результатами преобразования.

Рассмотрим представление `flight_departure` из листинга 7.7.

Листинг 7.7 ❖ Представление с преобразованием столбца

```
CREATE VIEW flight_departure as
SELECT bl.flight_id,
       departure_airport,
       coalesce(actual_departure, scheduled_departure)::date AS departure_date,
       count(DISTINCT passenger_id) AS num_passengers
  FROM booking b
       JOIN booking_leg bl USING (booking_id)
       JOIN flight f USING (flight_id)
       JOIN passenger p USING (booking_id)
 GROUP BY 1,2,3
```

При выполнении запроса

```
SELECT flight_id,
       num_passengers
  FROM flight_departure
 WHERE flight = 22183
```

фильтр для рейса будет перемещен внутрь представления, и запрос будет выполнен менее чем за секунду. Пользователь, который не знает, что `flight_departure` является представлением, может подумать, что все столбцы обеспечивают сопоставимую производительность, и может удивиться, выполнив следующий запрос:

```
SELECT flight_id,
       num_passengers
  FROM flight_departure
 WHERE departure_date = '2020-08-01'
```

На его выполнение уходит почти две минуты. Разница связана с тем, что столбец `leave_date` преобразует данные, а, как обсуждалось в главе 5, индексы в этом случае использовать невозможно. План выполнения этого запроса показан на рис. 7.5.

Ситуация еще более сильного снижения производительности показана в листинге 7.8. К сожалению, это реальный случай. Когда нет понимания, какой запрос скрывает представление, оно может использоваться для выбора данных, которые намного легче получить из базовых таблиц.

Листинг 7.8 ❖ Выбор только одного столбца из представления

```
SELECT flight_id
  FROM flight_departure
 WHERE departure_airport = 'ORD'
```

Этому запросу не требуется количество пассажиров на рейсе; он просто выбирает рейсы, вылетающие из аэропорта О'Хара, на которые были проданы билеты. И все же план выполнения для запроса из листинга 7.8 довольно сложен – см. рис. 7.6.

	QUERY PLAN text
1	Subquery Scan on flight_departure (cost=1178105.06..1187048.64 rows=255531 width=12)
2	-> GroupAggregate (cost=1178105.06..1184493.33 rows=255531 width=20)
3	Group Key: bl.flight_id, f.departure_airport, ((COALESCE(f.actual_departure, f.scheduled_departure))::date)
4	-> Sort (cost=1178105.06..1178743.88 rows=255531 width=16)
5	Sort Key: bl.flight_id, f.departure_airport
6	-> Hash Join (cost=655797.98..1150786.36 rows=255531 width=16)
7	Hash Cond: (p.booking_id = b.booking_id)
8	-> Seq Scan on passenger p (cost=0.00..302486.09 rows=16312309 width=8)
9	-> Hash (cost=653986.68..653986.68 rows=89144 width=36)
10	-> Nested Loop (cost=27101.77..653986.68 rows=89144 width=36)
11	-> Hash Join (cost=27101.34..550408.49 rows=89144 width=28)
12	Hash Cond: (bl.flight_id = f.flight_id)
13	-> Seq Scan on booking_leg bl (cost=0.00..476508.08 rows=17828108 width=8)
14	-> Hash (cost=27058.64..27058.64 rows=3416 width=24)
15	-> Seq Scan on flight f (cost=0.00..27058.64 rows=3416 width=24)
16	Filter: ((COALESCE(actual_departure, scheduled_departure))::date = '2020-08-01'::date)
17	-> Index Only Scan using booking_pkey on booking b (cost=0.43..1.16 rows=1 width=8)
18	Index Cond: (booking_id = bl.booking_id)

Рис. 7.5 ❖ План выполнения, когда нельзя использовать индексы

Этот запрос выполняется в течение 1 минуты 42 секунд. Однако запрос, который выбирает ту же информацию без использования представления, использует доступные индексы и выполняется всего три секунды.

```
SELECT flight_id
  FROM flight
 WHERE departure_airport = 'ORD'
   AND flight_id IN (SELECT flight_id FROM booking_leg)
```

Зачем использовать представления?

Теперь, когда мы увидели столько примеров негативного влияния представлений, можно ли что-то сказать в их защиту? Существуют ли ситуации, в которых представления могут улучшить производительность запросов?

Внутри PostgreSQL любое создание представления включает в себя создание *правил*, в большинстве случаев неявно. Правила `select` могут ограничивать доступ к базовым таблицам. Правила, триггеры и автоматическое обновление делают представления в PostgreSQL чрезвычайно сложными и обеспечивают функциональность, очень похожую на таблицы.

	QUERY PLAN text
1	Subquery Scan on flight_departure (cost=886771.95..918054.44 rows=962538 width=4)
2	-> GroupAggregate (cost=886771.95..908429.06 rows=962538 width=20)
3	Group Key: bl.flight_id, f.departure_airport, ((COALESCE(f.actual_departure, f.scheduled_departure))::date)
4	-> Sort (cost=886771.95..889178.30 rows=962538 width=12)
5	Sort Key: bl.flight_id, ((COALESCE(f.actual_departure, f.scheduled_departure))::date)
6	-> Nested Loop (cost=9545.27..774662.60 rows=962538 width=12)
7	Join Filter: (b.booking_id = p.booking_id)
8	-> Nested Loop (cost=9544.84..520610.13 rows=334023 width=36)
9	-> Hash Join (cost=9544.41..367021.97 rows=334023 width=28)
10	Hash Cond: (bl.flight_id = f.flight_id)
11	-> Seq Scan on booking_leg bl (cost=0.00..310506.66 rows=17893566 width=8)
12	-> Hash (cost=9384.99..9384.99 rows=12753 width=24)
13	-> Bitmap Heap Scan on flight f (cost=243.26..9384.99 rows=12753 width=24)
14	Recheck Cond: (departure_airport = 'ORD'::bpchar)
15	-> Bitmap Index Scan on flight_departure_airport (cost=0.00..240.07 rows=12753 width=0)
16	Index Cond: (departure_airport = 'ORD'::bpchar)
17	-> Index Only Scan using booking_pkey on booking b (cost=0.43..0.46 rows=1 width=8)
18	Index Cond: (booking_id = bl.booking_id)
19	-> Index Only Scan using passenger_booking_id on passenger p (cost=0.43..0.64 rows=9 width=4)
20	Index Cond: (booking_id = bl.booking_id)

Рис. 7.6 ❖ План выполнения для запроса из листинга 7.8

Однако они не дают никакого преимущества в плане производительности. Лучшее и, возможно, единственно оправданное использование представлений – это построение уровня безопасности или определение элементов отчетов, гарантирующих, что все соединения и бизнес-логика определены правильно.

Материализованные представления

Большинство современных систем баз данных позволяют пользователям создавать материализованные представления, но их реализации и точное поведение различаются.

Начнем с определения.

Материализованное представление – это объект базы данных, который объединяет в себе и определение запроса, и таблицу для хранения результатов этого запроса, когда он выполнен.

Материализованное представление отличается от обычного представления, потому что сохраняются результаты запроса, а не только определение представления. Это означает, что материализованное представление отражает не актуальные данные, а данные на момент последнего обновления. Материализованное представление отличается от таблицы, потому что данные в нем нельзя изменять напрямую, их можно только обновить с помощью предопределенного запроса.

Создание и использование материализованных представлений

Рассмотрим пример создания материализованного представления в листинге 7.9.

Листинг 7.9 ❖ Создаем материализованное представление

```
CREATE MATERIALIZED VIEW flight_departure_mv AS
SELECT bl.flight_id,
       departure_airport,
       coalesce(actual_departure, scheduled_departure)::date departure_date,
       count(DISTINCT passenger_id) AS num_passengers
  FROM booking b
       JOIN booking_leg bl USING (booking_id)
       JOIN flight f USING (flight_id)
       JOIN passenger p USING (booking_id)
 GROUP BY 1,2,3
```

Что происходит при выполнении этой команды? В данном конкретном случае на выполнение потребуется очень много времени. Но когда команда завершится, в базе данных появится новый объект, хранящий результаты выполнения запроса. В дополнение вместе с данными будет храниться и сам запрос. Когда запросы ссылаются на материализованные представления, те (в отличие от обычных представлений) ведут себя точно так же, как таблицы. Оптимизатор не будет заменять материализованные представления определяющими запросами, они будут доступны в виде таблиц. Для материализованных представлений могут быть созданы индексы, однако у них не может быть первичного и внешнего ключей:

```
CREATE UNIQUE INDEX flight_departure_flight_id
ON flight_departure_mv (flight_id);
--
CREATE INDEX flight_departure_dep_date
ON flight_departure_mv (departure_date);
--
CREATE INDEX flight_departure_dep_airport
ON flight_departure_mv (departure_airport);
```

Выполнение этого запроса займет всего 400 мс, а в плане выполнения будет показано сканирование индекса.

```
SELECT flight_id,
       num_passengers
  FROM flight_departure_mv
 WHERE departure_date_= '2020-08-01'
```

Обновление материализованных представлений

Команда REFRESH заполняет материализованное представление результатами базового запроса во время выполнения обновления. Синтаксис этой команды выглядит так:

```
REFRESH MATERIALIZED VIEW flight_departure_mv
```

Материализованные представления в PostgreSQL менее зрелые, чем в некоторых других СУБД, таких как Oracle. Материализованные представления не могут обновляться инкрементально, а расписание обновления нельзя указать в определении материализованного представления. Каждый раз при выполнении команды REFRESH таблица материализованного представления опустошается, и в нее вставляются результаты сохраненного запроса. Если во время обновления возникает ошибка, процесс обновления откатывается, и материализованное представление остается неизменным.

Во время обновления материализованное представление блокируется, и его содержимое недоступно для других процессов. Чтобы сделать предыдущую версию материализованного представления доступной во время обновления, к команде добавляется ключевое слово CONCURRENTLY:

```
REFRESH MATRIALIZED VIEW CONCURRENTLY flight_departure_mv
```

Материализованное представление может обновляться в таком режиме, только если имеет уникальный индекс. Обновление при этом займет больше времени, чем обычно, но доступ к материализованному представлению не будет заблокирован.

Создавать материализованное представление или нет?

Трудно сформулировать точные и универсальные условия, при которых создание материализованного представления будет выгодно, но некоторые рекомендации для принятия решения все же можно дать. Обновление материализованного представления требует времени, а выбор из материализованного представления происходит намного быстрее, чем из обычного представления. Поэтому учтите следующее:

- как часто меняются данные в базовых таблицах?
- насколько важно иметь самые свежие данные?
- как часто нужно выбирать эти данные (а точнее, сколько чтений ожидается на одно обновление)?
- сколько разных запросов будут использовать эти данные?

Какими должны быть пороговые значения для характеристик «часто» и «много»? Это субъективные оценки, но посмотрим на некоторые примеры. Листинг 7.10 определяет материализованное представление, очень похожее на представление из листинга 7.9, за исключением того, что оно выбирает рейсы, которые вылетели вчера.

Листинг 7.10 ❖ Материализованное представление для вчерашних рейсов

```
CREATE MATERIALIZED VIEW flight_departure_prev_day AS
SELECT bl.flight_id,
       departure_airport,
       coalesce(actual_departure,
                scheduled_departure)::date departure_date,
       count(DISTINCT passenger_id) AS num_passengers
  FROM booking bJOIN booking_leg bl USING (booking_id)
       JOIN flight f USING (flight_id)
       JOIN passenger p USING (booking_id)
 WHERE (actual_departure BETWEEN CURRENT_DATE - 1 AND CURRENT_DATE)
    OR (actual_departure IS NULL AND
        scheduled_departure BETWEEN CURRENT_DATE - 1 AND CURRENT_DATE)
 GROUP BY 1,2,3
```

Информация о рейсах, которые вылетели вчера, не изменится, поэтому можно с уверенностью предположить, что представление не нужно будет обновлять до следующего дня. С другой стороны, это материализованное представление можно использовать в различных запросах, которые будут выполняться быстрее, если результаты запроса будут материализованы.

Рассмотрим еще одного потенциального кандидата на материализацию – запрос из листинга 6.29.

Предположим, что мы создали материализованное представление с вложенным запросом, как в листинге 7.11.

Листинг 7.11 ❖ Создаем материализованное представление из подзапроса

```
CREATE MATERIALIZED VIEW passenger_passport AS
SELECT cf.passenger_id,
       coalesce(max(CASE WHEN custom_field_name = 'passport_num'
                         THEN custom_field_value ELSE NULL
                    END),'') AS passport_num,
       coalesce(max(CASE WHEN custom_field_name = 'passport_exp_date'
                         THEN custom_field_value ELSE NULL
                    END),'') AS passport_exp_date,
       coalesce(max(CASE WHEN custom_field_name = 'passport_country'
                         THEN custom_field_value ELSE NULL
                    END),'') AS passport_country
  FROM custom_field cf
 GROUP BY 1
```

Кажется, что это материализованное представление будет очень полезно. Во-первых, уже было показано, что для выполнения такого запроса требуется много времени, поэтому можно сэкономить за счет предварительного рас-

чета результатов. Во-вторых, паспортные данные не меняются (эта информация связана с бронированием, и одному и тому же человеку будет назначаться иной идентификатор `passenger_id` при другом бронировании). Этот запрос был бы отличным кандидатом на роль материализованного представления, если бы не несколько потенциальных проблем.

Во-первых, пассажиры не обязаны указывать свои паспортные данные при бронировании. Хотя после ввода информация уже не меняется, но паспортные данные могут быть введены в любой момент до закрытия выхода на посадку. Следовательно, это материализованное представление необходимо постоянно обновлять, и каждое обновление занимает около 10 минут.

Во-вторых, это материализованное представление будет расти. В отличие от предыдущего примера, в котором ежедневное обновление охватывает данные исключительно за предыдущий день, количество паспортных данных будет увеличиваться, и обновление материализованного представления будет занимать все больше и больше времени.

Такие ситуации часто упускаются из виду на ранней стадии проекта, когда во всех таблицах мало данных и материализованные представления обновляются быстро. Поскольку PostgreSQL не допускает инкрементное обновление материализованных представлений, возможным решением будет создание другой таблицы с той же структурой, что и материализованное представление из листинга 7.11, и периодическое обновление строк, когда появляются новые паспортные данные.

Однако, если будет принято такое решение, не понятно, зачем вообще нужна таблица `custom_field`, если данные необходимы в формате, заданном материализованным представлением `passenger_passport`. Это будет темой следующей главы, в которой обсуждается влияние проектирования схемы данных на производительность.

Нужно ли оптимизировать материализованные представления?

Хотя запрос материализованного представления выполняется реже, чем используется само представление, нам все же нужно обращать внимание на время его выполнения. Даже если материализованное представление представляет собой короткий запрос (например, когда оно содержит данные предыдущего дня, как в листинге 7.9), он может полностью сканировать большие таблицы, если отсутствуют правильные индексы или план выполнения неоптимален.

Как упоминалось ранее, мы не принимаем оправданий, что запрос не нуждается в оптимизации, потому что выполняется нечасто, будь то один раз в месяц, в неделю или в день. Никому не нравятся отчеты, даже нечасто нужные, если они создаются шесть часов. Кроме того, запуск этих периодических отчетов часто планируется на одно и то же время – обычно на 9 часов утра понедельника, – что добавляет никому не нужный стресс к началу недели. Техники, обсуждаемые в главах 5 и 6, могут и должны применяться к материализованным представлениям.

Зависимости

Когда создаются представления и материализованные представления, побочным эффектом является создание *зависимостей*. И обычные, и материализованные представления имеют связанные с ними запросы, и при изменении любого объекта базы данных, участвующего в этих запросах, необходимо пересоздать зависимые от них представления.

Фактически PostgreSQL даже не позволяет изменять или удалять таблицы либо материализованные представления, если от них зависят другие представления или материализованные представления. Для внесения изменений необходимо указывать ключевое слово `CASCADE` в командах `ALTER` или `DROP`.

Обратите внимание: даже если удаляемый или изменяемый столбец не участвует в каком-либо зависимом объекте, эти зависимые объекты все равно должны быть удалены и созданы заново.

Если обычные или материализованные представления создаются поверх других представлений, изменение одного столбца в одной базовой таблице может привести к пересозданию нескольких десятков зависимых объектов базы данных. Обычное представление создается быстро, но перестройка нескольких зависимых материализованных представлений может занять значительное время, в течение которого материализованные представления будут недоступны, даже если они допускают одновременное обновление.

В последующих главах обсуждаются функции и хранимые процедуры, которые могут устранить такие зависимости.

Секционирование

До сих пор в этой главе обсуждались различные способы разделения запросов на более мелкие части.

Секция представляет собой другой вид разделения – не кода, а данных. Секционированная таблица состоит из нескольких секций, каждая из которых определяется как отдельная таблица. Каждая табличная строка хранится в одной из секций в соответствии с правилами, указанными при создании секционированной таблицы.

Поддержка секций появилась в PostgreSQL относительно недавно; начиная с версии 10 в каждый выпуск вносятся улучшения, упрощающие использование секционированных таблиц.

Наиболее распространенный случай – секционирование по диапазонам. Каждая секция содержит строки, в которых значение атрибута находится в диапазоне, назначенном секции. Диапазоны, назначенные разным секциям, не могут пересекаться, а строку, которая не попадает ни в одну секцию, нельзя вставить.

В качестве примера создадим версию таблицы boarding_pass с секциями. Последовательность команд показана в листинге 7.12.

Листинг 7.12 ❖ Создание секционированной таблицы

```
-- создание таблицы
--
CREATE TABLE boarding_pass_part (
      boarding_pass_id SERIAL,
      passenger_id BIGINT NOT NULL,
      booking_leg_id BIGINT NOT NULL,
      seat TEXT,
      boarding_time TIMESTAMPTZ,
      precheck BOOLEAN NOT NULL,
      update_ts TIMESTAMPTZ
)
PARTITION BY RANGE (boarding_time);

-- создание секций
--
CREATE TABLE boarding_pass_may
PARTITION OF boarding_pass_part
FOR VALUES FROM ('2020-05-01'::timestamptz) TO ('2020-06-01'::timestamptz) ;
--
CREATE TABLE boarding_pass_june
PARTITION OF boarding_pass_part
FOR VALUES FROM ('2020-06-01'::timestamptz) TO ('2020-07-01'::timestamptz);
--
CREATE TABLE boarding_pass_july
PARTITION OF boarding_pass_part
FOR VALUES FROM ('2020-07-01'::timestamptz) TO ('2020-08-01'::timestamptz);
--
CREATE TABLE boarding_pass_aug
PARTITION OF boarding_pass_part
FOR VALUES FROM ('2020-08-01'::timestamptz) TO ('2020-09-01'::timestamptz);
--
INSERT INTO boarding_pass_part SELECT * from boarding_pass;
```

Зачем создавать секционированную таблицу?

Секции можно добавлять к секционированной таблице и можно удалять. Команда DROP выполняется значительно быстрее, чем команда DELETE, удаляющая множество строк, и не требует последующей очистки. Типичный вариант использования – к таблице, разделенной по диапазонам дат (например, по секции на каждый месяц), в конце каждого месяца добавляется новая секция, а самая старая удаляется.

Секционирование можно использовать для распределения больших объемов данных по нескольким серверам баз данных: секция может быть сторонней таблицей.

С точки зрения производительности, секционирование может сократить время, необходимое для полного сканирования таблицы: если запрос содержит условия для ключа секционирования, сканирование ограничивается

только подходящими секциями. Это делает секционирование особенно полезным для длинных запросов, часто использующих полное сканирование.

Как выбрать ключ секционирования для таблицы? Основываясь на предыдущем наблюдении, ключ нужно выбирать таким образом, чтобы он использовался для поиска либо в достаточно большом количестве запросов, либо в наиболее критичных из них.

Посмотрим на пример из главы 6, листинг 6.21.

```
SELECT city,
       date_trunc('month', scheduled_departure),
       sum(passengers) passengers
  FROM airport a
       JOIN flight f ON airport_code = departure_airport
       JOIN (
             SELECT flight_id, count(*) passengers
               FROM booking_leg l
                    JOIN boarding_pass b USING (booking_leg_id)
              WHERE boarding_time > '07-15-20'
                AND boarding_time < '07-31-20'
              GROUP BY flight_id
            ) cnt USING (flight_id)
 GROUP BY 1,2
 ORDER BY 3 DESC
```

Хотя этот запрос ограничен датой посадки с 15 по 31 июля, он все равно является длинным и полностью сканирует таблицу `boarding_pass`. План выполнения идентичен плану на рис. 6.18.

Однако при выполнении аналогичного запроса с использованием секционированной таблицы `boarding_pass_part` (см. листинг 7.13) этот запрос будет использовать секции.

Листинг 7.13 ❖ Запрос к секционированной таблице

```
SELECT city,
       date_trunc('month', scheduled_departure),
       sum(passengers) passengers
  FROM airport a
       JOIN flight f ON airport_code = departure_airport
       JOIN (
             SELECT flight_id, count(*) passengers
               FROM booking_leg l
                    JOIN boarding_pass_part b USING (booking_leg_id)
              WHERE boarding_time > '07-15-20'
                AND boarding_time < '07-31-20'
              GROUP BY flight_id
            ) cnt USING (flight_id)
 GROUP BY 1,2
 ORDER BY 3 DESC
```

План выполнения на рис. 7.7 показывает, что вместо сканирования всей таблицы оптимизатор решает сканировать только одну секцию, поскольку

запрос содержит фильтр по времени посадки. Время выполнения запроса для несекционированной таблицы примерно одинаково независимо от фильтрации по времени посадки, но для секционированной таблицы время выполнения уменьшается в два с лишним раза, потому что все строки находятся в одной секции.

	QUERY PLAN text
1	Sort (cost=2266277.54..2266626.48 rows=139576 width=48)
2	Sort Key: (sum(cnt.passengers)) DESC
3	-> GroupAggregate (cost=2246566.38..2250055.78 rows=139576 width=48)
4	Group Key: a.city, (date_trunc('month'::text, f.scheduled_departure))
5	-> Sort (cost=2246566.38..2246915.32 rows=139576 width=24)
6	Sort Key: a.city, (date_trunc('month'::text, f.scheduled_departure))
7	-> Hash Join (cost=2198268.36..2231776.13 rows=139576 width=24)
8	Hash Cond: (f.departure_airport = a.airport_code)
9	-> Hash Join (cost=2198243.38..2231033.48 rows=139576 width=20)
10	Hash Cond: (f.flight_id = cnt.flight_id)
11	-> Seq Scan on flight f (cost=0.00..23642.76 rows=683176 width=16)
12	-> Hash (cost=2195816.68..2195816.68 rows=139576 width=12)
13	-> Subquery Scan on cnt (cost=2143736.04..2195816.68 rows=139576 width=12)
14	-> GroupAggregate (cost=2143736.04..2194420.92 rows=139576 width=12)
15	Group Key: l.flight_id
16	-> Sort (cost=2143736.04..2160165.75 rows=6571882 width=4)
17	Sort Key: l.flight_id
18	-> Hash Join (cost=769001.43..1219836.23 rows=6571882 width=4)
19	Hash Cond: (b.booking_leg_id = l.booking_leg_id)
20	-> Seq Scan on boarding_pass_july b (cost=0.00..312597.60 rows=6571882 widt...
21	Filter: ((boarding_time >= '2020-07-15 00:00:00-05'::timestamp with time zone) ...
22	-> Hash (cost=476508.08..476508.08 rows=17828108 width=8)
23	-> Seq Scan on booking_leg l (cost=0.00..476508.08 rows=17828108 width=8)
24	-> Hash (cost=16.66..16.66 rows=666 width=12)
25	-> Seq Scan on airport a (cost=0.00..16.66 rows=666 width=12)

Рис. 7.7 ❖ План выполнения с секционированной таблицей

У секций могут быть собственные индексы, которые, очевидно, меньше индекса всей секционированной таблицы. Этот вариант может быть полезен для коротких запросов, однако производительность возрастет существенно только в том случае, если почти все запросы извлекают данные из одной и той же секции. Затраты на поиск в B-дереве пропорциональны его глубине. Индекс по секции, скорее всего, устранит только один уровень B-дерева, а выбор необходимой секции также требует некоторого количества ресурсов.

Эти ресурсы, вероятно, сопоставимы со стоимостью чтения дополнительного уровня индекса. Конечно, запрос может обращаться к конкретной секции, а не ко всей секционированной таблице, скрывая затраты на выбор секции для приложения, отправляющего запрос.

Поэтому не следует переоценивать преимущества секционирования для коротких запросов.

Параллелизм

Во введении к этой книге говорилось, что параллельное выполнение в ней не рассматривается по двум причинам. Во-первых, параллельное выполнение появилось относительно недавно, в PostgreSQL 10. Во-вторых, никто из авторов не имеет промышленного опыта использования параллелизма в PostgreSQL и не может добавить что-либо к существующей документации. Каждая новая версия PostgreSQL вносит дополнительные улучшения в параллельное выполнение.

Однако параллелизм часто представляется как панацея от всех проблем, связанных с производительностью, и мы должны предупредить: не возлагайте слишком больших надежд на параллелизм – и это касается любой СУБД, а не только PostgreSQL.

Параллельное выполнение можно рассматривать как еще один способ разбить запрос: объем работы, необходимый для выполнения, распределяется между процессорами или ядрами.

В любом параллельном алгоритме есть определенная часть, которая должна выполняться на одном устройстве. Также появляются дополнительные накладные расходы на синхронизацию между параллельными процессами. Поэтому параллельное выполнение особенно привлекательно для обработки больших объемов данных. В частности, параллелизм полезен для сканирований больших таблиц и соединений хешированием. И сканирования, и соединения хешированием типичны для длинных запросов, для которых ускорение обычно наиболее значимо.

Напротив, ускорение коротких запросов обычно незначительно. Однако параллельное выполнение разных запросов может улучшить пропускную способность, но это не связано с параллельным выполнением одного запроса.

Иногда оптимизатор может заменить доступ на основе индекса (который использовался бы при последовательном выполнении) параллельным сканированием таблицы. Это может быть вызвано неточной оценкой стоимости. В таких случаях параллельное выполнение может оказаться медленнее, чем последовательное.

Все планы выполнения в этой книге были созданы при отключенном параллелизме.

Кроме того, преимущества масштабируемости, обеспечиваемые параллельным выполнением, не могут исправить плохую схему данных или ком-

пенсировать неэффективный код по простой математической причине: преимущества масштабируемости от параллелизма в лучшем случае линейны, в то время как стоимость вложенных циклов квадратична.

Выводы

В этой главе были рассмотрены различные способы разбиения запросов на более мелкие функциональные части, а также преимущества и недостатки этих способов. Были рассмотрены потенциальные ловушки одного из часто используемых инструментов оптимизации – временных таблиц – и было показано, как обобщенные табличные выражения можно использовать в качестве альтернативы, которая не мешает оптимизатору запросов. Мы также обсудили обычные и материализованные представления и их влияние на производительность. Наконец, были кратко рассмотрены секционирование и параллельное выполнение.

Глава 8

Оптимизация модификации данных

До этого момента основное внимание уделялось оптимизации запросов, то есть охватывался только поиск данных. Мы не затронули ничего, связанного с обновлением, удалением или добавлением записей в базу данных. Это и есть тема данной главы, в которой обсуждается, как модификация данных влияет на производительность и что в этом процессе можно улучшить.

Что такое DML?

У любой системы баз данных есть два языка: DDL (*Data Definition Language – язык описания данных*), используемый для создания таблиц и других объектов базы данных, и DML (*Data Manipulation Language – язык управления данными*), который применяется, чтобы запрашивать и изменять данные в базе. В PostgreSQL DDL и DML являются частью SQL. Некоторые команды связаны с DDL (`ALTER TABLE`, `CREATE MATERIALIZED VIEW`, `DROP INDEX` и т. д.), а другие относятся к DML (`INSERT`, `UPDATE`, `DELETE`). Эти команды часто называют теми же словами DDL и DML, поэтому фраза «выполнение DDL» означает выполнение команд определения данных, а «выполнение DML» – команд `INSERT`, `UPDATE` или `DELETE`.

Два способа оптимизации модификации данных

Выполнение любой команды DML состоит из двух частей: выбора записей, которые нужно изменить, и самого изменения данных. В случае с `INSERT` первая часть может быть опущена при вставке констант. Однако если используется конструкция `INSERT-SELECT`, сначала должны быть найдены записи, необходимые для вставки.

По этой причине оптимизация команды DML состоит из двух частей: оптимизации выборки и оптимизации модификации данных.

Если проблема заключается в выборке, то следует оптимизировать именно часть SELECT, что подробно описано в предыдущих главах. Данная глава посвящена второй части – оптимизации записи данных.

В подавляющем большинстве случаев даже системы OLTP выполняют значительно меньше команд DML, чем команд SELECT. Это основная причина того, что люди редко говорят об оптимизации DML. Однако длительно работающие команды DML могут вызвать проблемы не только потому, что обновленные данные не будут своевременно доступны в системе, но также из-за возможности появления *блокирующих замко́в* (locks), которые замедляют выполнение других команд.

Как работает DML?

Чтобы обсуждать оптимизации, применимые к командам SQL для модификации данных, потребуется еще немного теории.

Низкоуровневый ввод-вывод

В конечном итоге любая операция SQL, какой бы сложной она ни была, сводится к паре низкоуровневых операций: чтению и записи отдельных блоков базы данных. Причина проста: данные, содержащиеся в базе, могут быть обработаны только при извлечении блоков в оперативную память, и все изменения сначала выполняются в памяти, а уже затем записываются на диск.

Фундаментальное различие между операциями чтения и записи состоит в том, что чтение с диска должно быть завершено до того, как данные можно будет обработать; таким образом, команду SELECT нельзя завершить до того, как все необходимые блоки будут загружены в память. Напротив, изменения данных внутри блока завершаются до начала записи; таким образом, операцию SQL можно завершить без задержек. Нет необходимости ждать, пока измененные данные действительно будут записаны на диск. Это может показаться нелогичным: обычно представляется, что обновление требует больше ресурсов, чем чтение.

Конечно, для записи действительно требуется гораздо больше ресурсов, чем для чтения: база данных должна изменять индексы и регистрировать обновления в журнале предзаписи (WAL, write-ahead log). Тем не менее это происходит в оперативной памяти, если речь идет об отдельных командах DML. Журнальные записи принудительно сбрасываются на диск только при фиксации.

Выходит, любая команда INSERT, UPDATE или DELETE выполняется намного быстрее, чем SELECT. Это здорово, но зачем тогда нужна оптимизация?

Есть две основные причины. Во-первых, запись на диск все равно необходима и, следовательно, потребляет некоторое количество аппаратных ресур-

сов, в основном пропускную способность ввода-вывода. Затраты на операции записи амортизируются и не обязательно отражаются на какой-либо отдельной операции, но все же замедляют обработку и могут даже повлиять на производительность запросов. Дополнительную рабочую нагрузку создают фоновые и обслуживающие процедуры (например, запись измененных блоков на диск). Обычно при обслуживании выполняется реструктуризация данных, например при операции `VACUUM` в PostgreSQL. Некоторые задачи реструктуризации блокируют доступ к изменяемому объекту на все время реструктуризации.

Во-вторых, модификации могут мешать другим модификациям и даже поиску. Пока данные не изменяются, порядок обработки не имеет значения. Данные могут быть доступны одновременно из разных команд `SELECT`. Но модификации не могут выполняться одновременно, и в этом случае все решает порядок выполнения операций. Чтобы обеспечить корректность данных, некоторые операции приходится откладывать или даже отклонять. За корректность отвечает подсистема управления одновременным доступом (подсистема обработки транзакций). Обработка транзакций не является темой данной книги; однако при обсуждении модификаций нельзя избежать соображений, связанных с транзакционным поведением СУБД.

Влияние одновременного доступа

Для обеспечения правильного порядка операций диспетчеры транзакций обычно используют замки (блокировки).

Если транзакции необходима блокировка, а другая транзакция уже удерживает конфликтующую блокировку, выполнение откладывается до тех пор, пока конфликтующая блокировка не будет снята. Такое ожидание блокировки – основная причина задержек в операциях модификации.

Еще одна задача управления одновременным доступом – гарантировать, что обновления не будут потеряны. Любые обновления, выполненные зафиксированной транзакцией, должны надежно сохраняться на жестком диске перед фиксацией. Для этого используется журнал предзаписи. Все изменения данных регистрируются в журнальных записях на жестком диске, прежде чем транзакцию можно будет зафиксировать. Журнальные записи ведутся последовательно, а на медленных вращающихся дисках последовательные операции чтения и записи выполняются на два порядка быстрее, чем произвольные. На твердотельных накопителях эта разница незначительна. Фиксация не должна ждать, пока в базу данных будут записаны все изменения из кеша, но обязана дождаться сброса журнала предзаписи. В результате слишком частые фиксации могут значительно замедлить обработку. Крайний случай – когда каждая команда DML выполняется в отдельной транзакции. Так действительно бывает, если приложение не использует команды управления транзакциями, и поэтому база данных превращает каждую команду в отдельную транзакцию. С другой стороны, слишком длинные транзакции могут вызвать замедление из-за блокировок.

Приведенные выше соображения применимы к любой высокопроизводительной базе данных. Рассмотрим специфику PostgreSQL.

Одна из отличительных особенностей PostgreSQL заключается в том, что она никогда не выполняет обновления на месте. Вместо этого новая версия элемента (например, строки таблицы) вставляется и сохраняется в свободное место в том же или в новом выделенном блоке, в то время как предыдущая версия не перезаписывается тотчас же.

На рис. 8.1 показана структура блока с рис. 3.1 после удаления (или обновления) второй строки. Пространство, ранее занятое этой строкой, нельзя использовать для другой строки; по сути, удаленные данные все еще доступны.

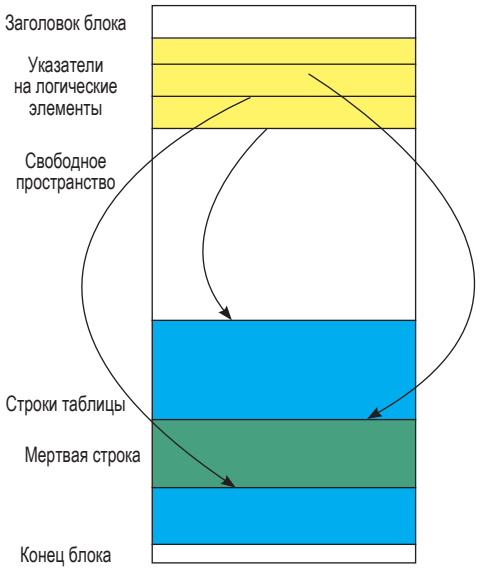


Рис. 8.1 ❖ Структура блока после удаления строки

Эта особенность может как положительно, так и отрицательно влиять на производительность.

Устаревшие версии не хранятся вечно. Операция очистки удаляет их и объединяет свободное место в блоке, когда старые версии больше не нужны для текущих транзакций.

PostgreSQL использует протокол изоляции снимков (SI, snapshot isolation) для управления одновременным доступом, чтобы предотвращать нежелательное влияние транзакций друг на друга. Обратите внимание, что в учебниках по базам данных обычно объясняется, как работают блокировки в протоколе двухфазного блокирования, а это значительно отличается от способа использования блокировок в PostgreSQL. Сведения, полученные из учебников или опыта работы с другими системами, могут ввести вас в заблуждение.

При изоляции снимков транзакция всегда считывает последнюю зафиксированную версию строки. Если другая транзакция обновила эту строку, но не выполнила фиксацию до начала операции чтения, будет прочитана устаревшая версия. Это преимущество, поскольку старая версия остается доступной

и для ее чтения не требуется блокировка. Многоверсионное управление доступом улучшает пропускную способность, поскольку нет необходимости откладывать операции чтения.

Согласно изоляции снимков, одновременная запись одних и тех же данных не допускается: если две конкурирующие (выполняющиеся одновременно) транзакции пытаются изменить одни и те же данные, одна из двух транзакций должна быть оборвана. Есть две стратегии обеспечения этого правила: *побеждает первое обновление* (first update wins) и *побеждает первая фиксация* (first commit wins). Первую стратегию реализовать проще: про выполненное обновление известно сразу, и вторая транзакция может быть прервана без ожидания. Однако PostgreSQL использует вторую стратегию.

Чтобы обеспечить соблюдение этого правила, PostgreSQL использует блокировку операции записи для любой операции модификации. Прежде чем транзакция сможет внести какие-либо изменения в данные, она должна установить блокировку для обновлений. Если блокировка не может быть получена из-за того, что другая транзакция изменяет те же данные, то операция откладывается до завершения транзакции, удерживающей конфликтующую блокировку. Если удерживающая транзакция обрывается, то блокировка снимается и предоставляется ожидающей транзакции, которая теперь может завершить операцию изменения данных. В противном случае, если удерживающая блокировку транзакция завершается успешно, последующее поведение зависит от уровня изоляции транзакции. На уровне изоляции `READ COMMITTED`, который используется в PostgreSQL по умолчанию, ожидающая транзакция прочитает измененные данные, установит блокировку операции записи и завершит модификацию. Такое поведение возможно, потому что на этом уровне изоляции операция чтения может прочитать версию, зафиксированную на момент начала команды `SELECT`, а не на момент начала транзакции. На уровне изоляции `REPEATABLE READ` ожидающая транзакция будет прервана. Такая реализация приводит к ожиданиям, но позволяет избежать ненужных обрывов.

Мы не обсуждаем уровень `SERIALIZABLE`, потому что он используется крайне редко.

Теперь рассмотрим несколько важных частных случаев.

Модификация данных и индексы

В главе 5, когда мы говорили о создании новых индексов, мы упоминали, что добавление индексов к таблице может потенциально замедлить операции DML. Насколько медленнее они становятся, зависит от характеристик хранилища и системы (дисков, процессоров и памяти), но экспертное мнение таково, что добавление дополнительного индекса в среднем приводит к увеличению времени выполнения команд `INSERT` и `UPDATE` всего на 1 %.

Вы можете провести несколько экспериментов, используя схему `postgres_air`. Например, начните с таблицы, в которой есть много индексов, такой как `flight`.

Сначала создайте копию этой таблицы без индексов:

```
CREATE TABLE flight_no_index AS
     SELECT * FROM flight LIMIT 0;
```

Затем вставьте строки из таблицы `flight` в таблицу `flight_no_index`:

```
INSERT INTO flight_no_index
     SELECT * FROM flight LIMIT 100
```

Теперь опустошите новую таблицу и создайте для нее те же индексы, что создавались в главе 5 для таблицы `flight`. Повторите вставку. Вы не заметите разницы во времени выполнения для небольшого количества (около пары сотен) строк, но вставка 100 000 строк будет выполняться чуть медленнее. Однако для типичных операций, выполняемых в окружении OLTP, существенной разницы не будет.

Естественно, создание индексов требует времени, и стоит отметить, что операция `CREATE INDEX` в PostgreSQL накладывает исключительную блокировку на таблицу, что может повлиять на другие операции. Команда `CREATE INDEX CONCURRENTLY` занимает больше времени, но оставляет таблицу доступной для других процессов.

Как мы упоминали ранее, PostgreSQL вставляет новые версии обновленных строк. Это оказывает определенное отрицательное влияние на производительность: как правило, поскольку новые версии имеют другое расположение, необходимо обновить все индексы в таблице. Чтобы уменьшить этот негативный эффект, PostgreSQL использует технику, которую иногда называют HOT (heap-only tuples). Если при изменении строки в блоке достаточно свободного места, чтобы вставить новую версию в него же, и если обновление не требует внесения каких-либо изменений в индексированные столбцы, то нет необходимости изменять индексы.

Массовые обновления и частые обновления

Как упоминалось ранее, PostgreSQL никогда не уничтожает данные сразу. Команда `DELETE` помечает удаленные строки как удаленные, а команда `UPDATE` вставляет новую версию строки и помечает предыдущую версию как устаревшую. Когда эти строки больше не нужны для активных транзакций, они становятся «мертвыми». Наличие «мертвых» строк уменьшает количество активных строк в блоке и, таким образом, замедляет последующие сканирования таблиц.

Пространство, занятое «мертвыми» строками (то есть удаленными версиями строк), остается неиспользованным до тех пор, пока не будет возвращено операцией очистки `VACUUM`. В большинстве случаев, даже при относительно высокой частоте обновлений, обычная очистка, инициированная демоном автоматической очистки, быстро устраняет мертвые версии строк, так что они не вызывают каких-либо значительных задержек.

Однако если выполняется массовое обновление или удаление (то есть любая операция, влияющая на большую часть таблицы), запрос к этой таблице

может значительно замедлиться, потому что карта видимости заставит перепроверять видимость, обращаясь к блокам таблицы. Также, как упоминалось ранее, количество активных версий на странице уменьшится. Это приведет к тому, что каждый запрос будет считывать в память большее количество блоков, а это в конечном итоге может привести к тому, что база данных начнет выполнять собственные внутренние операции подкачки.

В таком случае требуется агрессивная настройка автоматической очистки или ручное выполнение операций `VACUUM ANALYZE`.

Выполнение очистки может вызвать существенное увеличение ввода-вывода, что приводит к ухудшению производительности других активных сеансов. Очистку можно настроить так, чтобы распределить ее воздействие во времени и таким образом уменьшить количество резких всплесков ввода-вывода. Однако в результате очистка будет занимать больше времени.

Теперь рассмотрим другой случай: таблица часто обновляется, хотя каждое из этих обновлений влияет на одну или очень небольшое количество строк.

Как обсуждалось ранее, любое обновление строки создает новую копию. Однако если новая версия хранится в том же блоке и значение индексированного столбца не изменяется, тогда нет необходимости обновлять индекс, и PostgreSQL не будет его трогать.

Чтобы описанная функция работала, блоки должны содержать достаточное количество свободного места.

Процент свободного пространства в табличных блоках можно задать с помощью параметра хранения `fillfactor` в предложении `WITH` команды `CREATE TABLE`. По умолчанию значение этого параметра равно 100: PostgreSQL старается уместить в блоках как можно больше строк и минимизировать свободное место. Таким образом, обычно свободное место может появиться только после обновлений или удалений с последующей очисткой.

Чтобы уменьшить накладные расходы на обновление индексов, можно указать меньшие значения `fillfactor`. PostgreSQL допускает значения вплоть до 10, оставляя 90 % блочного пространства для обновленных версий строк. Конечно, маленькие значения параметра `fillfactor` приводят к увеличению количества блоков, необходимых для хранения данных, и, следовательно, к увеличению количества чтений, необходимых для сканирования таблицы. Это значительно замедляет длинные запросы, но может быть менее заметным для коротких запросов, особенно если из блока выбирается только одна строка.

Ссылочная целостность и триггеры

Наличие нескольких внешних ключей в таблице потенциально может замедлить DML. Это не означает, что не надо проверять ссылочную целостность. Напротив, способность поддерживать ссылочную целостность – одна из самых мощных функций реляционных систем. Причина, по которой операции манипулирования данными могут замедляться, заключается в том, что для каждой операции `INSERT` или `UPDATE` в таблице с ограничениями целостности движок базы данных должен проверить, присутствуют ли новые значения ограниченных столбцов в соответствующих родительских таблицах, таким

образом выполняя дополнительные неявные запросы. Эти проверки могут выполняться мгновенно, например если родительская таблица представляет собой небольшой справочник, содержащий всего несколько строк. Однако если размер родительской таблицы сопоставим с размером дочерней таблицы, накладные расходы могут быть более заметными. Как и в большинстве других случаев, фактическое время задержки зависит от параметров системы и характеристик аппаратного обеспечения.

Время выполнения с ограничениями и без них можно сравнить, создав копию таблицы `flight`:

```
CREATE TABLE flight_no_constr AS
     SELECT * FROM flight LIMIT 0;
```

Теперь добавьте к новой таблице те же ограничения, что и у таблицы `flight`, и снова попробуйте выполнить операции вставки. Вы можете заметить, что добавление проверки целостности для атрибута `aircraft_code` не влияет на время вставки, но добавление ограничений для `departure_airport` и `arrival_airport` заметно его увеличит.

Обратите внимание, что влияние оказывается и на операции с родительской таблицей: когда запись в родительской таблице обновляется или удаляется, движок базы данных должен проверить, нет ли записей в каждой из дочерних таблиц, которые ссылаются на обновленное или удаленное значение.

Триггеры также могут замедлять операции модификации данных по той же причине, что и ограничения ссылочной целостности: каждый вызов триггера может привести к выполнению нескольких дополнительных команд SQL. Степень, в которой каждый триггер замедляет выполнение, зависит от его сложности.

Стоит отметить, что ограничения ссылочной целостности в PostgreSQL реализуются с помощью системных триггеров, поэтому все наблюдения, касающиеся ограничений целостности, применимы и к триггерам. Тот факт, что наличие триггеров может повлиять на производительность, не означает, что триггеры не должны использоваться. Напротив, если есть какие-то действия или проверки, которые должны быть выполнены для любой операции DML в таблице, полезно реализовать их с помощью триггеров базы данных, вместо того чтобы программировать эти проверки в приложении. Последний подход будет менее эффективным и не будет охватывать случаи, когда данные в таблице изменяются непосредственно в базе данных, а не через приложение.

Выводы

В этой главе мы кратко обсудили влияние операций управления данными на производительность системы. Обычно команды DML выполняются по крайней мере на порядок реже, чем запросы. Однако если модификации данных выполняются неэффективно, это может привести к блокирующим замкам и, таким образом, повлиять на производительность всех частей приложения.

Глава 9

Проектирование имеет значение

Во введении мы отметили, что оптимизация начинается во время сбора требований и проектирования. Если быть точным, все начинается с проектирования системы, включая проектирование схемы базы данных, но невозможно правильно спроектировать базу данных, если не потратить время на сбор информации об объектах, которые должны в ней храниться. В этой главе мы обсудим различные возможные решения и покажем, как они могут повлиять на производительность.

Проектирование имеет значение

В главе 1 описаны два различных решения для хранения информации о телефонных номерах, показанные на рис. 1.1 и 1.2. Вернемся к этому примеру.

В листинге 9.1 показаны определения таблиц, используемые в схеме `postgres_air`. Таблица `account` содержит информацию об учетных записях пользователей, а таблица `phone` содержит информацию обо всех телефонах, связанных с учетными записями. Эта связь поддерживается ограничением внешнего ключа.

В листинге 9.2 показан альтернативный вариант, в котором все телефоны хранятся вместе с информацией об учетной записи.

Есть несколько причин, по которым для схемы `postgres_air` был выбран вариант с двумя таблицами; как уже говорилось в главе 1, у многих нет домашних стационарных телефонов или отдельного рабочего телефона, а у кого-то есть несколько сотовых телефонов или виртуальный номер, такой как Google Voice. Все эти сценарии могут поддерживаться решением с двумя таблицами, но они не вписываются в вариант с одной таблицей, если только мы не начнем для каждого случая добавлять новый столбец. Указание основного телефона в решении с одной таблицей потребует повторения одного из номеров в столбце `primary_phone`, создавая возможность для несогласованности. С точки зрения производительности решение с двумя таблицами также более выгодно.

Листинг 9.1 ❖ Вариант с двумя таблицами

```
/* таблица учетных записей */
CREATE TABLE account
(   account_id integer,
    login text,
    first_name text,
    last_name text,
    frequent_flyer_id integer,
    update_ts timestamp with time zone,
    CONSTRAINT account_pkey PRIMARY KEY (account_id),
    CONSTRAINT frequent_flyer_id_fk
        FOREIGN KEY (frequent_flyer_id) REFERENCES frequent_flyer (frequent_flyer_id)
);

/* таблица телефонов */
CREATE TABLE phone
(   phone_id integer,
    account_id integer,
    phone text,
    phone_type text,
    primary_phone boolean,
    update_ts timestamp with time zone,
    CONSTRAINT phone_pkey PRIMARY KEY (phone_id),
    CONSTRAINT phone_account_id_fk
        FOREIGN KEY (account_id) REFERENCES account (account_id)
);
```

Листинг 9.2 ❖ Вариант с одной таблицей

```
/* таблица учетных записей */
CREATE TABLE account
(   account_id integer,
    login text,
    first_name text,
    last_name text,
    frequent_flyer_id integer,
    home_phone text,
    work_phone text,
    cell_phone text,
    primary_phone text,
    update_ts timestamp with time zone,
    CONSTRAINT account_pkey PRIMARY KEY (account_id),
    CONSTRAINT frequent_flyer_id_fk
        FOREIGN KEY (frequent_flyer_id) REFERENCES frequent_flyer (frequent_flyer_id)
);
```

В варианте с двумя таблицами поиск учетной записи по номеру телефона – простой запрос:

```
SELECT DISTINCT account_id
  FROM phone
 WHERE phone = '8471234567'
```

Этот запрос будет выполнен сканированием только индекса.

В варианте с одной таблицей аналогичный запрос будет выглядеть так:

```
SELECT account_id
  FROM account
 WHERE home_phone = '8471234567'
    OR work_phone = '8471234567'
    OR cell_phone = '8471234567'
```

Чтобы избежать полного сканирования, необходимо построить три разных индекса.

Означает ли это, что вариант с одной таблицей хуже, чем вариант с двумя таблицами? Все зависит от того, как обращаются к данным. Если схема поддерживает систему для туристических агентств, то наиболее вероятное использование состоит в получении учетной записи клиента по телефонному номеру. Когда агент спрашивает у клиента номер телефона, он не интересуется типом этого номера.

С другой стороны, отчет по учетным записям клиентов, которые были обновлены за последние сутки, должен включать домашний, рабочий и сотовый телефоны в отдельных столбцах, независимо от того, пусты ли они, и включать учетные записи, в которые были внесены какие-либо изменения за последние 24 часа, включая обновления номеров телефонов. В этом случае решение с одной таблицей, показанное в листинге 9.3, намного проще и эффективнее.

Листинг 9.3 ❖ Использование одной таблицы

```
SELECT *
  FROM account
 WHERE update_ts BETWEEN now()- interval '1 day' AND now();
```

Получить тот же результат в варианте с двумя таблицами сложнее, см. листинг 9.4.

Эти два примера иллюстрируют еще один момент – запрос, для которого решение с двумя таблицами является предпочтительным, с бóльшей вероятностью появится в системе OLTP, а запрос, который лучше обслуживается решением с одной таблицей, более характерен для системы OLAP. Для преобразования данных из OLTP-систем в формат, который лучше подходит для нужд бизнес-аналитики, можно использовать ETL-инструменты.

Похожая ситуация была показана в главе 6, где неоптимальная схема базы данных приводила к неоптимальному запросу (см. листинг 6.26). Даже оптимизированная версия запроса оставалась относительно медленной. Эти примеры показывают влияние проектирования схемы базы данных на производительность. Иногда негативные последствия неподходящей схемы нельзя исправить улучшением запроса или построением дополнительных индексов.

В следующих разделах этой главы рассматриваются проектные решения, которые чаще всего отрицательно влияют на производительность.

Листинг 9.4 ❖ Тот же запрос в варианте с двумя таблицами

```
SELECT a.account_id,
       login,
       first_name,
       last_name,
       frequent_flyer_id,
       home_phone
       work_phone,
       cell_phone,
       primary_phone
  FROM account a
       JOIN (
         SELECT account_id,
                max(phone) FILTER (WHERE phone_type = 'home') AS home_phone,
                max(phone) FILTER (WHERE phone_type = 'work') AS work_phone,
                max(phone) FILTER (WHERE phone_type = 'mobile') AS cell_phone,
                max(phone) FILTER (WHERE primary_phone IS true) AS primary_phone
           FROM phone
          WHERE account_id IN (
                  SELECT account_id
                    FROM phone
                   WHERE update_ts BETWEEN now()- interval '1 day' AND now()
                  UNION
                  SELECT account_id
                    FROM account
                   WHERE update_ts BETWEEN now()- interval '1 day' AND now()
                )
          GROUP BY 1
       ) p USING (account_id)
```

Зачем использовать реляционную модель?

Хотя все предыдущие примеры являются реляционными, поскольку PostgreSQL построена на реляционной модели, мы знаем, что многие считают реляционные базы данных устаревшими или вышедшими из моды. Выступления с заголовками «Что последует за реляционными базами данных?» появляются с регулярной частотой.

В данном разделе мы не пытаемся защищать реляционные базы данных. Они не нуждаются в защите, и до сих пор ни один потенциальный преемник не достиг даже сопоставимого распространения. Наша цель состоит в том, чтобы объяснить ограничения других моделей.

Типы баз данных

Итак, какие есть альтернативы реляционным моделям? В настоящее время в ходу большое количество систем баз данных и хранилищ данных, использующих самые разные модели данных и методы хранения. К ним относятся

традиционные реляционные системы с хранением данных по строкам или по колонкам, масштабируемые распределенные системы, системы потоковой обработки и многое другое.

Мы видели, как различные нереляционные системы прошли гартнеровский цикл хайпа от пика чрезмерных ожиданий до избавления от иллюзий. Однако стоит отметить, что ядром реляционной модели является язык запросов, основанный на алгебре логики, а не на каком-либо конкретном способе хранения данных. Вероятно, в этом причина того, что многие системы, созданные в качестве альтернативы традиционным РСУБД, в конечном итоге использовали варианты SQL в качестве языка запросов высокого уровня и, следовательно, связанную с ним алгебру логики.

Несмотря на то что реляционные базы данных еще не скоро будут свергнуты с престола, существуют технологии, разработанные и проверенные в новых системах, которые оказались полезными и получили широкое распространение. Мы рассмотрим три популярных подхода: сущность–атрибут–значение, ключ–значение и иерархические системы, которые часто называют хранилищами документов.

Модель «сущность–атрибут–значение»

В модели *сущность–атрибут–значение* (entity-attribute-value, EAV) значения являются скалярными (обычно текстовыми, чтобы представлять разные типы данных). В главе 6 говорилось, что в этой модели есть таблица с тремя столбцами: первый для идентификатора сущности, второй для идентификатора атрибута этой сущности, а третий – значение этого атрибута сущности. Это сделано во имя «гибкости», которая на самом деле означает неточные или неизвестные требования. Неудивительно, что такая гибкость достигается за счет производительности. В главе 6 была представлена таблица `custom_field`. Мы отметили, что эта схема не является оптимальной, и показали возможное отрицательное влияние на производительность. Даже после применения методов оптимизации для устранения многократного сканирования таблицы выполнение оставалось относительно медленным.

Помимо влияния на производительность, такая схема ограничивает управление качеством данных. В случае, представленном в главе 6, три поля содержат данные трех разных типов: `passport_num` – число, `passport_exp_date` – дата, а `passport_country` – текстовое поле, которое должно содержать название существующей страны. Однако в таблице `custom_field` все эти значения хранятся в текстовом поле `custom_field_value`, что делает невозможным строгую типизацию и не позволяет создавать ограничения ссылочной целостности.

Модель «ключ–значение»

Тип, используемый в модели «ключ–значение», хранит сложные объекты в одном поле, структура которого не видна базе данных. В этом случае отдельные атрибуты объекта гораздо сложнее извлечь, и движок базы данных

может, по сути, только возвращать один объект по первичному ключу. Все поля, кроме первичного ключа, даже могут быть упакованы в один объект JSON.

После того как PostgreSQL представил поддержку JSON в версии 9.2, этот подход стал очень популярным среди разработчиков баз данных и приложений. В версии 9.4 был представлен JSONB, и в каждой последующей версии следовали дальнейшие улучшения. Благодаря этой поддержке столбцы таблицы, определенные как JSON, являются обычным делом. Например, таблицу пассажиров из схемы `postgres_air` можно определить, как показано в листинге 9.5.

Листинг 9.5 ❖ Таблица с JSON

```
CREATE TABLE passenger_json (
    passenger_id int,
    passenger_info json
);
```

Пример значения JSON для `passenger_info` показан в листинге 9.6.

Листинг 9.6 ❖ Пример значения JSON

```
{
    "booking_ref":  "8HNB12",
    "passenger_no": "1",
    "first_name":   "MARIAM",
    "last_name":    "WARREN",
    "update_ts":    "2020-04-17T19:45:55.022782-05:00",
}
```

Да, предлагаемая схема выглядит универсальной и не требует каких-либо изменений DDL независимо от того, сколько новых элементов данных будет добавлено в будущем. Однако этой модели присущи те же проблемы, что и модели EAV. При таком подходе невозможно проверять типы скалярных значений и невозможно определить ограничения ссылочной целостности.

Инструменты и подходы для работы с полями JSON обсуждаются далее в данной главе.

Иерархическая модель

Иерархические структуры легко понять и использовать. Фактически впервые они были реализованы в базах данных в 1960-х годах благодаря простоте применения и нетребовательности к памяти. Конечно, в то время не было ни XML, ни JSON. Эти структуры отлично работают, пока все укладывается в одну иерархию. Но как только данные попадают в несколько иерархий, подход становится и сложным, и неэффективным.

Проиллюстрируем это на примерах из схемы `postgres_air`, показанных на рис. 9.1. Для аэропорта список вылетающих рейсов представляет собой одну иерархию, а список прибывающих рейсов – другую. Посадочные тало-

ны могут входить в ту же иерархию, что и вылетающие рейсы. В то же время они могут быть частью совершенно другой иерархии, которая начинается с бронирований. Обратите внимание, что пассажиры и перелеты не могут входить в одну и ту же иерархию без дублирования.

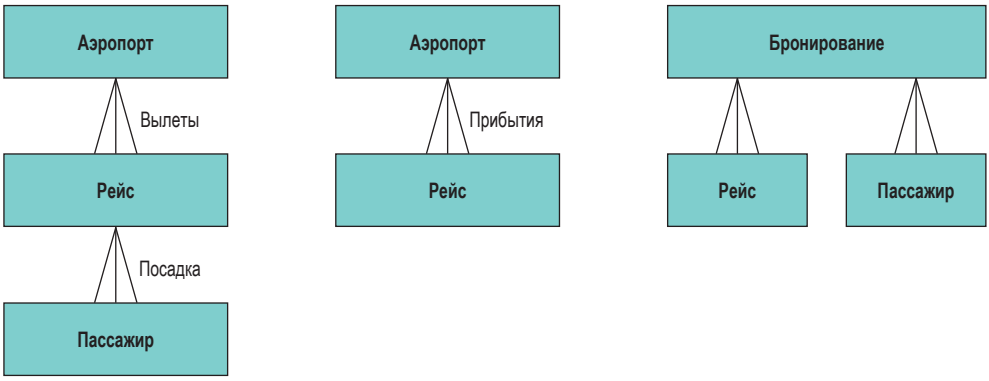


Рис. 9.1 ❖ Примеры иерархий в схеме postgres_air

Первые иерархические базы данных (IMS/360) предоставляли несколько иерархических представлений данных клиентскому приложению, но внутри поддерживали более сложные структуры данных.

Лучшее из разных миров

PostgreSQL – не просто реляционная, а объектно-реляционная система. Это означает, что типы данных столбцов не обязательно являются скалярными. Столбцы могут хранить структурированные типы, включая массивы, составные типы или объекты, представленные в виде документов JSON или XML.

Ответственное использование этих функций обеспечивает все потенциальные преимущества нескольких альтернативных подходов в сочетании с более традиционными реляционными возможностями.

В данном случае слово «ответственное» является ключевым. Например, PostgreSQL допускает подход с множественной иерархией, упомянутый в предыдущем разделе. Мы можем создать иерархические представления для клиентского приложения поверх внутренней реляционной структуры в базе данных. Такой подход сочетает в себе лучшее из обоих миров: для эффективного извлечения данных используется вся мощь реляционных запросов, а приложение получает сложные объекты в удобном формате обмена данными. Более подробная информация об этом подходе приведена в главе 13.

Хотя в этой книге мы не рассматриваем распределенные системы, стоит упомянуть, что у PostgreSQL имеется огромный набор расширений (дополнительных библиотек, не входящих в базовый дистрибутив), которые поддерживают распределенные запросы, включая запросы к СУБД, отличным от PostgreSQL. Эти расширения называются *обертками сторонних данных*

(FDW, foreign data wrappers). Они предоставляют почти прозрачные способы доступа к данным, которые могут находиться в более чем 60 типах СУБД, как реляционных, так и нереляционных.

Гибкость против эффективности и корректности

Частым аргументом в пользу гибкой схемы служит посыл «определение схемы или структуры данных может поменяться», ведь добавление столбца – это команда DDL, а добавление строки (в модели «ключ–значение») – это просто добавление строки.

Действительно, реальные системы эволюционируют, и, чтобы адекватно отражать эти изменения, существующие структуры данных должны меняться. Это может повлечь за собой добавление или удаление некоторых атрибутов либо изменение типов данных или связей. Тем не менее неизбежность изменений не обязывает использовать альтернативные модели, такие как хранилища документов или системы «ключ–значение». Стоимость внесения изменений в схему базы данных всегда необходимо сопоставлять с возможными проблемами производительности и целостности данных этих гибких решений.

В предыдущем разделе говорилось о сложности создания ограничений целостности при нереляционном подходе. По некоторым причинам широко распространено мнение, что базы данных NoSQL «быстрее», чем реляционные базы данных. Это утверждение может быть верным в очень ограниченном количестве сценариев, но в большинстве случаев ситуация обратная. Горизонтальное распределение может дать прирост производительности, но он уравновешивается стоимостью дополнительных шагов, необходимых для проверки целостности данных. Дополнительные потенциальные проблемы с производительностью возникают из-за трудностей создания индексов в моделях EAV и «ключ–значение».

Например, в случае с таблицей `custom_field` столбец `passport_exp_date` должен быть датой, и ее часто сравнивают с другими датами, например с датой рейса, чтобы убедиться, что срок действия паспорта не истекает до даты вылета. Однако эта дата хранится в текстовом поле, а это означает, что ее нужно приводить к типу `date`, чтобы корректно сравнивать значения. Более того, это приведение может выполняться только для строк, содержащих значения типа `date`.

PostgreSQL имеет частичные индексы, поэтому можно создать индекс только для тех строк, которые содержат дату истечения срока действия паспорта. Однако их нельзя проиндексировать как дату, которую можно использовать в качестве критерия поиска, поскольку индексы нельзя строить по изменчивым функциям:

```
CREATE INDEX custom_field_exp_date_to_date
ON custom_field (to_date(custom_field_value, 'MM-DD-YYYY'))
WHERE custom_field_name = 'passport_exp_date'
```

Это связано с тем, что все функции преобразования даты и времени являются изменчивыми, поскольку зависят от настроек текущего сеанса. Чтобы функцию преобразования можно было использовать в индексе, придется написать собственную. В главе 10 рассказывается о создании пользовательских функций. Эта функция должна будет обрабатывать исключения, и поэтому значение, ошибочно добавленное в неправильном формате, не будет проиндексировано. Кроме того, сам поиск будет значительно медленнее, чем с полем `date`.

Что насчет упаковки всех атрибутов в столбец JSON? Возникают аналогичные проблемы с индексацией. По столбцу JSON можно создать индекс; например, для таблицы `passenger_json` из листинга 9.5 можно создать индекс по `booking_ref`, как показано в листинге 9.7.

Листинг 9.7 ❖ Индексирование столбца JSON

```
CREATE INDEX passenger_j_booking_ref
ON passenger_json ((passenger_info->>'booking_ref'));
```

Он будет работать медленнее, чем индекс по исходной таблице `passenger`, но лучше, чем последовательное сканирование. Однако для любого значения, которое должно быть числовым или содержать дату, потребуется то же преобразование, что и в предыдущем примере.

Это не означает, что ни одно из этих нереляционных решений не имеет права на жизнь.

Например, одна таблица, описывающая некоторые нормативные акты Евросоюза, содержала около 500 столбцов. При смене правил, примерно раз в пять лет, в нее добавлялась одна строка. Замена этой таблицы вариацией модели «ключ–значение» (с парой дополнительных столбцов, характеризующих значение) пришлась по душе и разработчикам баз данных, и разработчикам приложений. Благодаря размеру данных проблем с эффективностью не возникло.

Мы рекомендуем применять столбцы JSON только в тех случаях, когда данные используются как единый объект, например при хранении внешних документов, кредитных отчетов и т. п. Даже в этих случаях те атрибуты, которые будут применяться для поиска, рекомендуется по возможности разложить по отдельным столбцам и хранить в дополнение ко всему объекту.

Нужна ли нормализация?

Во всей реляционной теории чаще всего неправильно используется термин «нормализация». Обычно все администраторы и разработчики баз данных, системные архитекторы и все остальные заявляют, что система должна быть «нормализована», но лишь немногие могут объяснить, что они хотят этим достичь, не говоря уже об определении нормализации.

Это не просто снобизм; нет никакой необходимости запоминать определение каждой нормальной формы всем, кто работает в области управления данными. Алгоритмы реляционной нормализации на практике используются нечасто. В этом смысле нормализация – это «мертвая» теория,

как латынь – мертвый язык. Тем не менее ученые все еще находят пользу в изучении латыни, а некоторые знания нормализации необходимы для качественного проектирования базы данных.

Неформально схема базы данных нормализована, если все значения столбцов зависят только от первичного ключа таблицы, а данные разбиты на несколько таблиц, чтобы избежать повторений.

Один из способов создания нормализованной схемы – начать с построения модели «сущность–связь»: если сущности определены правильно, схема базы данных, созданная из этой модели, будет нормализована. Можно сказать, что модель «сущность–связь» неявно включает в себя обнаружение зависимостей. Если она не нормализована, обычно это означает, что какие-то сущности отсутствуют.

Действительно ли важно нормализовать схему базы данных? Улучшает ли это производительность? Как обычно, все зависит от обстоятельств.

Повышение производительности *не* является основной целью нормализации. Нормализация создает чистую логическую структуру и помогает обеспечить целостность данных, особенно когда она поддерживается ограничениями ссылочной целостности. Нормализация необходима по тем же причинам, что и реляционная модель: не столько для хранения данных, сколько для обеспечения целостности и возможности использовать язык реляционных запросов. Отображение между логической структурой и структурой хранения не обязательно взаимно однозначно. В идеале для приложения должна быть предусмотрена чистая логическая структура, основанная на структуре хранения, оптимизированной для производительности.

С другой стороны, в мире есть много всего денормализованного, для чего нормализация не дает никаких преимуществ. Самый известный пример – почтовый адрес. Почтовый адрес США состоит из улицы с домом, города, почтового индекса и штата.

Адрес не является нормализованным. Это знают все, кто когда-либо отправлял посылки через киоски-автоматы USPS. Автоматическая проверка не позволит вам ввести почтовый индекс, не совпадающий с уже введенным адресом. Однако мы сомневаемся, что кто-то решит нормализовать адреса, хранящиеся в таблице базы данных.

Часто в поддержку денормализованной структуры данных приводится аргумент, что «соединения требуют времени» и поэтому денормализация необходима, чтобы запросы выполнялись быстрее. Для коротких запросов, когда они построены правильно, дополнительное время на соединения незначительно, как мы обсуждали в главе 5, и не следует жертвовать ради него точностью данных.

Во многих же случаях нормализация позволяет улучшить производительность, например когда нужно выбрать отдельные значения некоего атрибута с высокой избирательностью или, в общем случае, любое подмножество столбцов с повторяющимися значениями в ненормализованной таблице. В схеме `postgres_air` статус рейсов в таблице `flight` указывается явно. Это означает, что для получения списка возможных статусов рейсов необходимо выполнить следующий запрос:

```
SELECT DISTINCT status FROM flight
```

Правильное и неправильное использование суррогатных ключей

Суррогатные ключи – это уникальные значения, генерируемые системой для идентификации объектов, хранящихся в базе данных. В PostgreSQL суррогатные значения могут быть выбраны из последовательности. Когда строка вставляется, столбец с псевдотипом `serial` автоматически получает следующее значение из последовательности, связанной с таблицей.

Суррогатные ключи широко используются. Внутренние стандарты некоторых компаний требуют использования суррогатных ключей для любой таблицы. Однако у этих ключей есть не только преимущества, но и недостатки.

Преимущество суррогатов состоит в том, что значения, присвоенные различным объектам, гарантированно уникальны. Однако значение суррогатного ключа не связано с другими атрибутами и бесполезно при сопоставлении сохраненного объекта с объектом реальным.

Уникальность суррогатных ключей может скрывать определенные ошибки. Реальный объект может быть представлен в базе данных несколько раз с разными суррогатными ключами. Например, если одна покупка регистрируется в системе дважды, с карты клиента будет дважды списана оплата за один продукт, и проблему трудно будет решить без ручного вмешательства. При этом, хотя мы рекомендуем использовать какой-нибудь реальный уникальный атрибут для первичного ключа, это не всегда возможно. В базе данных супермаркета невозможно различить две бутылки колы, просканированные одним и тем же покупателем на кассе самообслуживания. Случаи покупки двух бутылок колы в одной транзакции и дублирования транзакции покупки одной колы должны быть различимы в исходной системе. Точно так же в больничных системах может быть несколько учетных номеров, связанных с одним пациентом; в этом случае очень важно иметь суррогатный ключ, чтобы все клинические данные пациента хранились вместе.

Иногда наличие суррогатного ключа в таблице ошибочно связывают с нормализацией. Внутренние стандарты некоторых компаний требуют наличия суррогатного ключа для каждой таблицы. Обычно это объясняется как способ нормализовать схему базы данных. И действительно, если каждой строке назначен уникальный идентификатор, все будет нормализовано. Но поскольку уникальные идентификаторы не имеют никакого отношения к объектам реального мира, одному настоящему объекту может оказаться сопоставлено несколько объектов в базе данных. Например, нам встречалась система, которая каждый раз при вводе покупателем неизвестного города присваивала этому городу уникальный идентификатор. Таким образом в системе набралось шесть разных версий Чикаго. Излишне говорить, что это не имеет ничего общего с нормализацией и может поставить под угрозу и точность данных, и производительность.

Использование суррогатов может привести к дополнительным соединениям. Таблица `flight` ссылается на таблицу `airport`, используя трехсимвольные коды, широко применяемые на практике. В этой схеме коды аэропортов можно извлечь из таблицы `flight`:

```
SELECT departure_airport, arrival_airport, scheduled_departure FROM flight
...
```

Однако если бы для таблицы `airport` использовался суррогатный ключ, то получение кодов аэропортов потребовало бы двух просмотров таблицы `airport`:

```
SELECT d.airport_code,
       a.airport_code,
       f.scheduled_departure
  FROM flight f
       JOIN airport d ON d.airport_id = f.departure_airport_id
       JOIN airport a ON a.airport_id = f.arrival_airport_id
```

Подробнее рассмотрим использование суррогатных ключей в схеме `postgres_air`.

Определение таблицы `airport` в схеме `postgres_air` показано в листинге 9.8.

Первичный ключ этой таблицы – `airport_code`. Этот столбец содержит трехзначные коды, которые используются для идентификации аэропортов во всех системах бронирования рейсов по всему миру, и эти коды никогда не меняются. Следовательно, они надежны как уникальные идентификаторы, и суррогатные ключи не нужны.

Листинг 9.8 ❖ Таблица аэропортов

```
CREATE TABLE airport
(   airport_code char(3) NOT NULL,
    airport_name text NOT NULL,
    city text NOT NULL,
    airport_tz text NOT NULL,
    continent text,
    iso_country text,
    iso_region text,
    intnl boolean NOT NULL,
    update_ts timestamptz,
    CONSTRAINT airport_pkey PRIMARY KEY (airport_code)
);
```

Точно так же самолеты идентифицируются трехсимвольными кодами, и мы используем их в качестве первичного ключа для таблицы `aircraft`, см. листинг 9.9.

Листинг 9.9 ❖ Таблица самолетов

```
CREATE TABLE aircraft
(   model text,
    range numeric NOT NULL,
    class integer NOT NULL,
    velocity numeric NOT NULL,
    code text NOT NULL,
    CONSTRAINT aircraft_pkey PRIMARY KEY (code)
)
```

Для таблицы бронирования (показанной в листинге 9.10) используется суррогатный первичный ключ `booking_id`, несмотря на то что имеется шестизначный номер `booking_ref`, который однозначно идентифицирует бронирование и никогда не меняется. Этот номер тоже является суррогатным ключом, хотя он и не получен из последовательности базы данных. Мы могли бы использовать его в качестве первичного ключа. Таким образом, столбец `booking_id` является избыточным, хотя он может пригодиться, если бронирования будут поступать из разных приложений. Определения таблиц такого вида встречаются во многих промышленных системах.

Листинг 9.10 ❖ Таблица бронирования

```
CREATE TABLE booking
(   booking_id bigint NOT NULL,
    booking_ref text NOT NULL,
    booking_name text,
    account_id integer,
    email text NOT NULL,
    phone text NOT NULL,
    update_ts timestamptz,
    price numeric(7,2),
    CONSTRAINT booking_pkey PRIMARY KEY (booking_id),
    CONSTRAINT booking_booking_ref_key UNIQUE (booking_ref),
    CONSTRAINT booking_account_id_fk FOREIGN KEY (account_id)
        REFERENCES account (account_id)
);
```

Таблица перелетов `booking_leg` (листинг 9.11) связывает бронирования с рейсами. Следовательно, естественный ключ для этой таблицы будет состоять из `flight_id` и `booking_id`, то есть из двух внешних ключей, ссылающихся на таблицы `flight` и `booking`. Эта пара столбцов была бы отличным первичным ключом. Решение создать дополнительный суррогатный ключ `booking_leg_id` было вызвано желанием избежать ссылок на составной ключ из зависимой таблицы (на таблицу `booking_leg` ссылается таблица `boarding_pass`, которая является самой большой таблицей в базе данных).

Листинг 9.11 ❖ Таблица перелетов

```
CREATE TABLE booking_leg
(   booking_leg_id serial,
    booking_id integer NOT NULL,
    booking_ref text NOT NULL,
    flight_id integer NOT NULL,
    leg_num integer,
    is_returning boolean,
    update_ts timestamp with time zone,
    CONSTRAINT booking_leg_pkey PRIMARY KEY (booking_leg_id),
    CONSTRAINT booking_id_fk FOREIGN KEY (booking_id)
        REFERENCES booking (booking_id),
    CONSTRAINT flight_id_fk FOREIGN KEY (flight_id)
        REFERENCES flight (flight_id)
)
```

Для таблицы passenger необходим суррогатный ключ (см. листинг 9.12), потому что один и тот же человек может быть пассажиром в нескольких бронированиях, и пассажир не обязательно регистрируется как клиент в системе бронирования (рейс может быть забронирован кем-то другим от имени этого пассажира).

Листинг 9.12 ❖ Таблица пассажиров

```
CREATE TABLE passenger
(   passenger_id serial,
    booking_id integer NOT NULL,
    booking_ref text,
    passenger_no integer,
    first_name text NOT NULL,
    last_name text NOT NULL,
    account_id integer,
    update_ts timestamptz,
    CONSTRAINT passenger_pkey PRIMARY KEY (passenger_id),
    CONSTRAINT pass_account_id_fk FOREIGN KEY (account_id)
        REFERENCES account (account_id),
    CONSTRAINT pass_booking_id_fk FOREIGN KEY (booking_id)
        REFERENCES booking (booking_id,
    CONSTRAINT pass_frequent_flyer_id_fk FOREIGN KEY (account_id)
        REFERENCES account (account_id)
);
```

Нет очевидного способа идентифицировать учетные записи; поэтому для таблицы account, показанной в листинге 9.13, необходимо использовать суррогатный ключ.

Листинг 9.13 ❖ Таблица учетных записей

```
CREATE TABLE account
(   account_id serial,
    login text NOT NULL,
    first_name text NOT NULL,
    last_name text NOT NULL,
    frequent_flyer_id integer,
    update_ts timestamp with time zone,
    CONSTRAINT account_pkey PRIMARY KEY (account_id),
    CONSTRAINT frequent_flyer_id_fk FOREIGN KEY (frequent_flyer_id)
        REFERENCES frequent_flyer (frequent_flyer_id)
);
```

Клиентов программы лояльности можно идентифицировать по номеру карты. Однако отдельный суррогатный ключ partial_flyer_id дает возможность выпустить замену потерянной или украденной карте без потери всех преимуществ постоянного клиента.

Напротив, в таблице flight необходим суррогатный ключ flight_id. Естественная идентификация рейса состоит из flight_num и schedule_departure.

Номер рейса в разные дни один и тот же, но время вылета может отличаться в разные дни и может немного меняться (например, переноситься на 5–10 минут позже), когда рейс уже частично забронирован. Столбец `flight_id` представляет конкретный рейс с определенным номером, как показано в листинге 9.14.

Листинг 9.14 ❖ Таблица рейсов

```
CREATE TABLE flight
(   flight_id serial,
    flight_no text NOT NULL,
    scheduled_departure timestamptz NOT NULL,
    scheduled_arrival timestamptz NOT NULL,
    departure_airport character(3) NOT NULL,
    arrival_airport character(3) NOT NULL,
    status text NOT NULL,
    aircraft_code character(3) NOT NULL,
    actual_departure timestamptz,
    actual_arrival timestamptz,
    update_ts timestamptz,
    CONSTRAINT flight_pkey PRIMARY KEY (flight_id),
    CONSTRAINT aircraft_code_fk FOREIGN KEY (aircraft_code)
        REFERENCES aircraft (code),
    CONSTRAINT arrival_airport_fk FOREIGN KEY (departure_airport)
        REFERENCES airport (airport_code),
    CONSTRAINT departure_airport_fk FOREIGN KEY (departure_airport)
        REFERENCES airport (airport_code)
);
```

У таблицы `boarding_pass` (листинг 9.15) есть суррогатный ключ, но на него не ссылаются другие таблицы, и поэтому он бесполезен. Естественный ключ этой таблицы состоит из двух столбцов: `flight_id` и `passenger_id`.

Листинг 9.15 ❖ Таблица посадочных талонов

```
CREATE TABLE boarding_pass
(   pass_id integer NOT NULL,
    passenger_id bigint,
    booking_leg_id bigint,
    seat text,
    boarding_time timestamptz,
    precheck boolean,
    update_ts timestamptz,
    CONSTRAINT boarding_pass_pkey PRIMARY KEY (pass_id),
    CONSTRAINT booking_leg_id_fk FOREIGN KEY (booking_leg_id)
        REFERENCES booking_leg (booking_leg_id),
    CONSTRAINT passenger_id_fk FOREIGN KEY (passenger_id)
        REFERENCES passenger (passenger_id)
);
```

Выводы

В этой главе обсуждалось влияние проектных решений на производительность. Мы охватили вариации в рамках реляционной модели, относящиеся к нормализации и суррогатным ключам, а также популярные нереляционные модели. Были изучены ограничения этих моделей и приведены примеры альтернативных подходов.

Глава 10

Разработка приложений и производительность

В середине этой книги, уже рассмотрев различные приемы оптимизации, пора сделать шаг назад и обратиться к дополнительным аспектам производительности, о которых говорилось в главе 1. В ней было сказано, что подход этой книги шире, чем просто оптимизация отдельных запросов.

Запросы к базе данных являются частью приложений, и в этой главе рассматривается оптимизация процессов, а не отдельных запросов. Хотя такая оптимизация обычно не считается «оптимизацией базы данных» в ее традиционном понимании, недостатки процесса могут легко свести на нет любое повышение производительности отдельных запросов. И поскольку разработчики приложений и баз данных склонны игнорировать эту область потенциальных улучшений, о ней мы и поговорим.

Время отклика имеет значение

В главе 1 под названием «Зачем нужна оптимизация?» перечислены причины низкой производительности, а также объясняется, для чего необходима оптимизация запросов. Однако ничего не было сказано о том, *почему* приложение должно быть эффективным.

Надеемся, что, прочитав добрую половину этой книги, вы еще не забыли, зачем вы вообще начали ее читать. Возможно, вы столкнулись с ситуацией, когда назрела необходимость улучшить общую производительность системы или производительность отдельной ее части. Однако, как это ни удивительно, все же нередко можно услышать мнение, что в большом времени отклика нет ничего страшного.

Мы категорически отвергаем это: время отклика важно. Спросите хотя бы у отдела маркетинга. С учетом сегодняшних ожиданий потребителей поговорка «время – деньги» как нельзя лучше подходит к данной ситуации.

Многочисленные маркетинговые исследования[1] показали, что быстрое время отклика веб-сайта или мобильного приложения имеет решающее значение для привлечения и поддержания входящего трафика. В большинстве случаев приемлемое время отклика составляет менее полутора секунд. Если время отклика увеличивается до трех секунд, половина посетителей покидают сайт, и более трех четвертей из них уже никогда не возвращаются.

Конкретные примеры включают цифры, представленные Google, которые демонстрируют, что замедление поиска на 0,4 секунды приводит к потере восьми миллионов запросов в день. Еще один пример – Amazon обнаружила, что замедление времени загрузки страницы на одну секунду приводит к потере продаж на сумму 1,6 млрд долларов в год. Какую проблему необходимо решать в таких случаях, чтобы улучшить ситуацию?

Всемирное ожидание

Если вы когда-либо беседовали с разработчиком, который трудится над приложением базы данных, или если вы сами являетесь одним из таких разработчиков, то следующая точка зрения может показаться вам знакомой: «Приложение работает отлично, пока дело не доходит до базы данных!»

Утверждение, которое мы интерпретируем как «у приложения часто возникают проблемы с производительностью при взаимодействии с базой данных», часто понимается как «базы данных работают медленно», что довольно неприятно слышать. В конце концов, СУБД – это специализированное программное обеспечение, предназначенное для обеспечения *более быстрого* доступа к данным, а не для замедления работы.

На самом деле если спросить администратора баз данных того же проекта, то он скажет, что база работает идеально. Но если это так, то почему же пользователи по всему миру вынуждены постоянно ждать (рис. 10.1)?

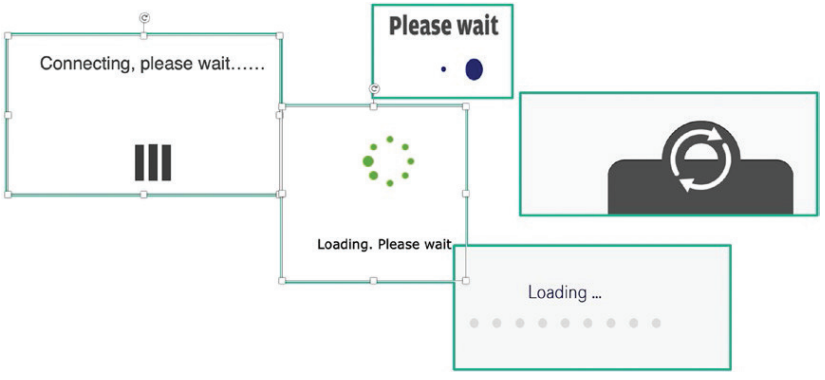


Рис. 10.1 ❖ Всемирное ожидание

[1] www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales/; https://builtvisible.com/improving-site-speed-talk-about-the-business-benefit/.

Хотя каждый запрос к базе данных, выполняемый приложением, обычно возвращает результаты менее чем за десятую долю секунды, время ответа страницы приложения может составлять десятки секунд. Таким образом, проблема не в скорости выполнения каждого отдельного запроса, а в том, как приложение взаимодействует с базой данных.

Показатели производительности

Когда в главе 1 обсуждались цели оптимизации, было упомянуто, что многие показатели производительности, такие как удовлетворенность клиентов, являются «внешними» по отношению к базе данных и не могут использоваться оптимизатором. Фактически эти показатели являются внешними не только по отношению к базе данных, но и к приложению в целом.

Время, необходимое для выполнения отдельной бизнес-функции, действительно трудно измерить и, как следствие, трудно улучшить. Разработчик приложения может заставить пользователя нажать десять кнопок вместо одной, и иногда это сокращает время отклика для каждой из десяти кнопок. Это может улучшить результаты каких-нибудь тестов, но вряд ли облегчит жизнь пользователю и сделает его счастливее.

Однако предыдущий раздел ясно показывает, что именно эти показатели интересуют конечного пользователя. Его не волнует какой-либо отдельный запрос; ему важно взаимодействие в целом, то есть он хочет, чтобы приложение отвечало быстро и ему не приходилось смотреть на «песочные часы».

Потеря соответствия

Так в чем же причины плохой общей производительности?

Если говорить в общих чертах, то причина заключается в несовместимости моделей баз данных и моделей языков программирования, которое можно выразить с помощью метафоры *несоответствие импеданса* или *потеря соответствия*. В электротехнике импеданс – это обобщение понятия сопротивления для случая переменного тока. Фазовый угол импеданса – это фазовый сдвиг между напряжением на элементе электрической цепи и током через этот элемент; если фазовый угол близок к 90°, то мощность близка к нулю даже при высоком напряжении и сильном токе.

Примерно так же сила выразительности и эффективности языков запросов к базам данных не соответствует сильным сторонам императивных языков программирования. По отдельности их сила велика, но вместе они обеспечивают меньшую мощность, чем ожидалось.

И императивные языки программирования, и декларативные языки запросов отлично справляются с задачами, для которых они были разработаны. Сложности начинаются, когда мы пытаемся заставить их работать вместе. Таким образом, причиной плохой работы является несовместимость моделей баз данных и моделей языков программирования.

Приложения и базы данных используют:

- данные разной гранулярности – отдельные объекты и множества объектов;
- разный тип доступа – навигационный и поиск по значениям атрибутов;
- разные средства идентификации – адрес и набор значений атрибутов.

В оставшихся разделах данной главы мы более подробно обсудим последствия этой несовместимости.

Дорога, вымощенная благими намерениями

Предыдущие разделы, возможно, звучат как обвинение разработчиков приложений во всех проблемах, связанных с производительностью, и в их нежелании «думать как база данных». Но обвинять кого-либо – неэффективный способ решения любых проблем, включая и низкую производительность приложений. Более продуктивно было бы попытаться понять, как благие намерения могут привести к таким тяжелым результатам.

Начнем с изучения шаблонов, которым рекомендуют следовать разработчикам приложений.

Шаблоны разработки приложений

Самый распространенный шаблон проектирования в современной программной инженерии – это многоуровневая архитектура. Обычно выделяют четыре уровня:

- интерфейс конечного пользователя;
- бизнес-логика;
- персистентность;
- база данных.

Каждый уровень может взаимодействовать только со смежными уровнями, а инкапсуляция и независимость поощряется и внутри каждого уровня, и, конечно, между уровнями. Таким образом, бизнес-объект `customer` (клиент) ничего не знает про таблицу базы данных `Customer` и фактически может быть подключен к любой произвольной базе данных, если на уровне персистентности определены соответствия между данными в базе и объектами на уровне бизнес-логики.

Для этого есть несколько важных причин, главные из которых – ускорение разработки, удобство сопровождения и простота модификации приложения, а также возможность многократного использования компонентов. На первый взгляд кажется очевидным, что изменение интерфейса конечного пользователя не должно де-факто вызывать изменение схемы базы данных. Строгое разделение также способствует быстрой совместной работе: разработчики могут заниматься разными частями приложения и быть уверенными, что изменения, которые они вносят во внутреннюю структуру или реализацию объектов, никак не повлияют на другие части приложения.

И конечно, кажется полезным, что на основе одной и той же бизнес-логики могут создаваться несколько приложений – внутренняя логика приложения не должна дублироваться для каждого нового окружения.

Пока все хорошо, так в чем проблема? К сожалению, есть много подводных камней, и методология не совсем обеспечивает обещанную выгоду – по крайней мере в том виде, в каком она применяется в реальных условиях.

Рассмотрим идею централизации бизнес-логики. Во-первых, преимущества объединения всей логики в одном месте, на бизнес-уровне, несколько уменьшаются, когда это «одно место» составляют несколько сотен тысяч строк кода. На практике такой большой бизнес-уровень приводит к дублированию кода или, что еще хуже, к попыткам дублирования кода. Когда уровень бизнес-логики увеличивается в размерах, трудно найти функцию, которая делает именно то, что нужно. В результате авторы часто наблюдали одну и ту же бизнес-логику, реализованную разными способами, разными методами и с разными результатами.

Во-вторых, бизнес-логика может быть доступна для дополнительных пользовательских интерфейсов, но она недоступна для других бизнес-приложений, которые напрямую взаимодействуют с базой данных, – в первую очередь для создания отчетов. Таким образом, авторы отчетов в конечном итоге дублируют логику приложения или в хранилище данных, или, что еще хуже, в отдельных отчетах, без гарантий эквивалентности исходной логике приложения.

Кроме того, при таком подходе взаимодействие с уровнем персистентности ограничивается отдельными объектами или даже отдельными скалярными значениями, что сводит на нет возможности движка базы данных. Интерфейс конечного пользователя может знать все необходимые ему элементы данных, но, поскольку он не взаимодействует напрямую с уровнем персистентности, запросы данных опосредуются уровнем бизнес-логики.

Типичная реализация уровня персистентности содержит классы доступа к данным, которые однозначно соответствуют классам бизнес-объектов. Базовые DML-функции (`INSERT`, `UPDATE`, `DELETE`) написать легко, но что происходит, когда операции должны выполняться над множеством объектов этого класса? Есть два пути: разработчик может создать еще один набор методов, которые будут повторять те же функции для объектов из множества. Однако это нарушит принцип повторного использования кода. Есть альтернативный и более распространенный вариант: разработчик просто перебирает элементы коллекции, вызывая функции, определенные для обработки отдельных объектов.

Представьте себе интерфейс приложения, в котором перечислены все пассажиры, вылетевшие из аэропорта О'Хара. Разработчик базы данных сообразит, что для составления списка всех пассажиров, вылетевших из аэропорта, необходимо соединить таблицу `flight` с таблицей `boarding_pass`. Вся информация будет получена за одно обращение. Для разработчика приложения задача может оказаться сложнее. У него может быть метод `GetFlightByDepartureAirport`, который принимает код аэропорта в качестве параметра и возвращает набор рейсов. Затем он может перебрать рейсы, возвращая все посадочные талоны на рейс. Фактически он реализует алгоритм соединения

вложенными циклами внутри приложения.

Чтобы избежать этого, можно использовать несколько разных решений. Можно добавить атрибут аэропорта вылета к объекту посадочного талона. Но тогда возможны проблемы с целостностью данных: что, если время вылета рейса обновится в записи рейса, но не во всех посадочных талонах? Можно определить метод получения посадочных талонов для данного аэропорта вылета, но это нарушит правило, согласно которому объекты не знают друг о друге. При чистом многоуровневом подходе объект посадочного талона ничего не знает об объекте рейса, а объект рейса ничего не знает про объект посадочного талона. Метод, извлекающий данные для обоих объектов, не может принадлежать ни к одному из них.

Проблема списка покупок

Стефан Фаро[1] описывает ситуацию, называемую «проблемой списка покупок».

Предположим, у вас есть список покупок для продуктового магазина. В реальной жизни вы садитесь в машину, едете в магазин, покупаете все продукты из своего списка, кладете их в багажник, едете домой, заносите их в дом и кладете в холодильник. А теперь представьте, что вместо этого вы едете в магазин, заходите и берете только первый продукт из своего списка, едете домой, кладете его в холодильник и снова отправляетесь в магазин! И продолжаете повторять ту же самую последовательность действий для каждого пункта из вашего списка.

Звучит нелепо? Да, но именно этим и занимаются многие приложения, взаимодействуя с базами данных.

А теперь представьте, что для повышения скорости совершения покупок эксперты предложили бы увеличить ширину проходов в магазине, построить более качественные дороги или оснастить автомобиль более мощным двигателем.

Некоторые из этих предложений действительно могут улучшить ситуацию. Но даже если бы можно было сократить время покупок на 30 %, это улучшение нельзя сравнивать с успехом, которого можно достичь с помощью одного простого улучшения процесса: купить все продукты за одну поездку.

Как перенести проблему со списком покупок на поведение приложения? Большинство проблем с производительностью вызваны **слишком большим количеством слишком мелких запросов**. И так же, как более качественные дороги не помогут, если за каждым продуктом ездить отдельно, так и следующие популярные предложения не улучшат производительность приложения:

- *более мощные компьютеры* практически не помогают, так как и приложение, и база данных находятся в состоянии ожидания 99 % времени;
- *более высокая пропускная способность сети* тоже не помогает. Сети с высокой пропускной способностью эффективны для передачи больших объемов данных, но не могут значительно сократить время, необходи-

[1] Stephane Faroult and Peter Robson, *The Art of SQL*.

мое для отправки сообщения серверу и получения результата. Время зависит от количества пересылок и количества сообщений, но почти не зависит от размера сообщений. Кроме того, размер заголовка пакета фиксирован; следовательно, для очень коротких сообщений полезная нагрузка использует только небольшую долю пропускной способности;

- *распределенные серверы* могут улучшить пропускную способность, но не время отклика, поскольку приложение отправляет запросы последовательно.

Антипаттерн «слишком много слишком мелких запросов» наблюдается уже несколько десятилетий. Примерно 20 лет назад одному из авторов пришлось проанализировать приложение, которому требовалось 5–7 минут для создания HTML-формы, содержащей около 100 полей. Код приложения был идеально структурирован, небольшие методы сопровождались комментариями и были отлично отформатированы. Однако трассировка базы данных показала, что для создания этой формы приложение выдавало около 16 000 запросов – больше, чем символов в самой форме. Дальнейший анализ показал, что несколько тысяч запросов исходили от метода `GetObjectIdByName`. За каждым из этих вызовов следовал запрос от метода `GetNameByObjectId`, который вызывался из другой части приложения, вероятно написанной другим разработчиком. Значения `name` были уникальными; следовательно, второй вызов всегда возвращал параметр первого. Один запрос, извлекающий все данные, необходимые для построения формы, вернул результат менее чем за 200 миллисекунд.

Несмотря на эти известные недостатки, многие компании упорно продолжают применять одни и те же средства снова и снова, каждый раз с одним и тем же результатом. Даже если изначально им удается добиться некоторого улучшения, это длится недолго. В течение ряда лет мы наблюдали за усилиями по оптимизации в одной компании.

Поскольку оптимизатор PostgreSQL пользуется преимуществами доступной оперативной памяти, эта компания увеличивала аппаратные ресурсы, чтобы вся (или почти вся) база данных помещалась в память. Мы наблюдали за миграцией с машин с 512 ГБ ОЗУ на машины с 1 ГБ, 2 ГБ, а затем 4 ГБ оперативной памяти, как только соответствующая конфигурация появлялась в наличии. Каждый раз после короткого периода относительного удовлетворения возникала та же проблема: база данных увеличивалась и выходила за пределы оперативной памяти.

Еще одно средство, которое часто применяется, – использование хранилища «ключ–значение» вместо полноценной базы данных. Аргументируется это тем, что «в приложении не используется ничего, кроме доступа по первичному ключу, поэтому механизм запросов нам не нужен». Действительно, такой подход может улучшить время отклика для любого отдельного доступа к данным, но не сократит время, необходимое для выполнения бизнес-функции. В одном из крайних случаев, который мы наблюдали, поиск записи по первичному ключу в среднем занимал около 10 миллисекунд. При этом количество обращений к базе данных, выполняемых за одно действие контроллера приложения, составило почти тысячу, с предсказуемым влиянием на общую производительность.

Интерфейсы

Еще одна причина неоптимального взаимодействия между приложением и базой данных кроется на уровне интерфейсов. Обычно приложения используют обобщенный интерфейс, такой как ODBC или JDBC. Эти интерфейсы слишком упрощенно представляют базы данных в виде набора плоских таблиц. И приложение, и база данных могут работать со сложными структурированными объектами; однако интерфейс не позволяет им обмениваться подобными высокоуровневыми структурами. Таким образом, даже если база данных поддерживает высокоуровневую модель, приложение от этого не выигрывает.

Чтобы получить сложный объект базы данных, приложение вынуждено отдельно запрашивать каждую часть объекта или, как вариант, разбирать плоское текстовое представление, возвращаемое через интерфейс, и создавать на его основе сам сложный объект.

Несовершенство доминирующих практик разработки хорошо известно профессионалам. Почему же они так распространены?

Причины лежат вне технической плоскости. Разработчики приложений почти всегда работают в условиях нехватки времени. Крайний срок выпуска нового продукта или новой функции часто определяется фразой «как можно скорее». Выпустить продукт как можно раньше существенно выгоднее, чем выпустить продукт лучшего качества, но позже.

Добро пожаловать в мир ORM

Желание изолировать язык базы данных (например, SQL) от разработчиков приложений и таким образом упростить им задачу (а также уменьшить потребность в навыках работы с базами данных) приводит к появлению программного обеспечения, преобразующего функции базы данных в объектные методы.

Инструмент объектно-реляционного отображения (ORM) – это программа, которая отображает объект базы данных в объект приложения в памяти.

Некоторые ORM-разработчики заявляют, что проблема потери соответствия решена. Объекты однозначно отображаются в таблицы баз данных, а структура базы данных, как и сгенерированный для взаимодействия с ней SQL, не имеет значения для разработчика приложения. К сожалению, цена такого решения – недопустимое снижение производительности.

Как работает объектно-реляционное отображение? Процесс показан на рис. 10.2.

1. Приложение разбирает объект на неделимые (скалярные) части.
2. Эти части отправляются в базу данных или из нее по отдельности.

3. База данных использует сложную структуру данных, но все запросы выполняются отдельно.

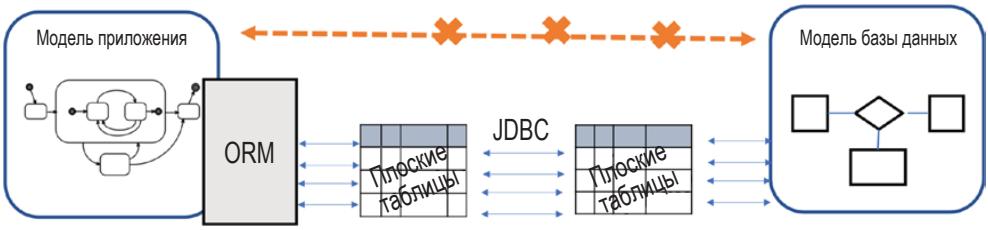


Рис. 10.2 ❖ Как работает объектно-реляционное отображение

Теоретически инструмент объектно-реляционного отображения не мешает приложению выполнять произвольные запросы к базе данных; обычно он предоставляет для этого определенные средства. Однако на практике почти всегда используются сгенерированные запросы из-за нехватки времени и простоты, с которой они создаются в приложении.

Поскольку фактический код базы данных скрыт от разработчика, операции базы данных со множествами объектов в конечном итоге реализуются так: метод ORM возвращает список идентификаторов объектов из базы данных, а затем каждый объект извлекается из базы данных отдельным запросом (также сгенерированным ORM). Таким образом, для обработки N объектов ORM выдает $N + 1$ запрос к базе данных, воплощая, по сути, описанную в предыдущем разделе «проблему списка покупок».

Такое отображение решает проблему абстрагирования от деталей хранения данных, но не обеспечивает эффективных средств манипулирования наборами данных.

Кроме того, ORM может скрывать важные детали реализации. Возьмем один пример из промышленной системы: параметр `IsActive` для объекта `Customer`, указывающий активность клиента в последнее время. Разработчик может подумать, что это просто атрибут, хранящийся в базе данных в таблице клиентов, но на самом деле его значение зависит от сложного набора критериев, основанных на поведении клиентов, и при каждом обращении к атрибуту выполнялся запрос. Хуже того, этот атрибут использовался и часто проверялся в коде, чтобы в зависимости от статуса клиента изменять компоненты пользовательского интерфейса. Чтобы отобразить одну страницу, этот сложный запрос выполнялся несколько раз.

В поисках более подходящего решения

Подведем итог сказанному: чтобы работать эффективно, классы и методы уровня приложения для таблиц и множеств должны быть интегрированы с базой данных (методы должны выполняться движком базы данных). Одна-

ко большинство архитектур не допускают такого рода интеграцию, что приводит к повторной реализации функций базы данных на уровне приложений.

Этот частный случай потери соответствия называется *объектно-реляционной потерей соответствия*.

Следовательно, традиционные способы организации взаимодействия между приложениями и базами данных являются наиболее значительным источником замедления работы приложений. Здесь нет злого умысла: разработчики приложений и баз данных делают все, что в их силах, с помощью имеющихся у них инструментов.

Чтобы решить эту проблему, нужно найти способ *передачи коллекций сложных объектов*. Обратите внимание, что на самом деле мы ищем решение двух тесно связанных проблем. Первая проблема – это невозможность передать «все данные сразу», то есть думать и действовать в терминах множеств. Вторая проблема – это невозможность передавать сложные объекты без их предварительной разборки на простые части.

Чтобы проиллюстрировать желаемый результат, рассмотрим пример того, как веб-приложение может взаимодействовать с базой данных `postgres_air`. Когда пользователь входит в систему онлайн-бронирования, первое, что он видит, – его уже существующие бронирования. Когда он выбирает конкретное бронирование, то видит экран, похожий на скриншот, показанный на рис. 10.3.

Ваш полет до Чикаго (ORD) 02.07.2020

Рейс 3013 Мюнхен (MUC) – Нью-Йорк (JFK) 16:10
Рейс 3014 Нью-Йорк (JFK) – Чикаго (ORD) 13:10 – следующий день

Обратный полет – 09.07.2020

Рейс 3317 Чикаго (ORD) – Нью-Йорк (JFK) 19:00
Рейс 3014 Нью-Йорк (JFK) – Мюнхен (MUC) 05:05 – следующий день

Пассажиры
Пассажир 1: Кэтлин Смит
Пассажир 2: Дуг Уайт
Пассажир 3: Марта Уайт

Рис. 10.3 ❖ Экран вашего бронирования

Информация, которая отображается на вашем экране, выбирается из нескольких таблиц: `booking`, `booking_leg`, `flight`, `passenger` и `airport`. После регистрации вы также увидите посадочные талоны.

Веб-приложение, разработанное с использованием традиционного подхода, будет обращаться к базе данных 17 раз для отображения этих результатов: сначала чтобы выбрать список `booking_id` для текущего пользователя, затем выбрать детали бронирования из таблицы `booking`, потом выбрать детали для каждого сегмента бронирования (всего четыре), далее сведения о каждом рейсе (еще четыре), сведения об аэропорте (еще четыре) и сведения о пас-

сажирах (еще три). При этом разработчик приложения точно знает, какой объект ему следует создать, чтобы отобразить результаты бронирования. И разработчик базы данных тоже знает, как выбрать всю информацию, необходимую для создания такого объекта. Если бы мы нарисовали черновик структуры этого объекта, у нас получилось бы что-то, похожее на рис. 10.4.

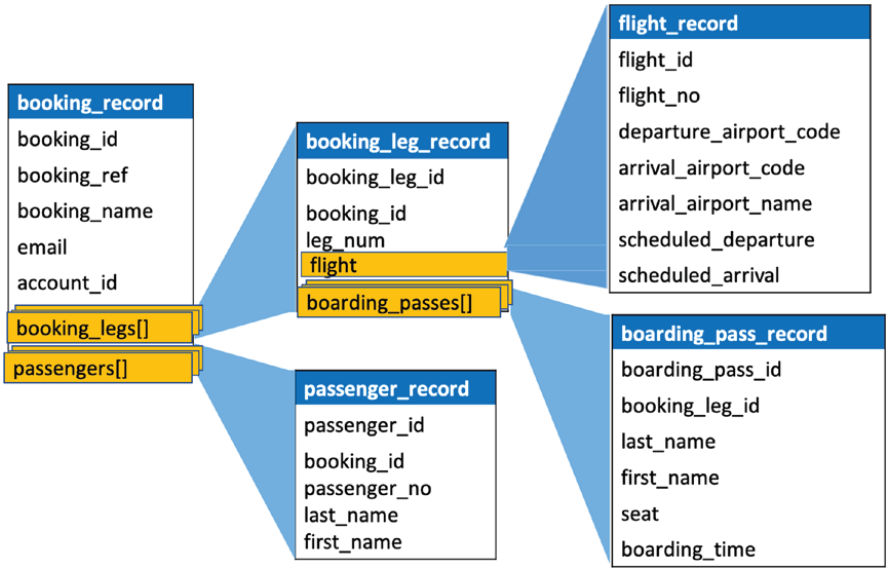


Рис. 10.4 ❖ Отображение сложных объектов

Если бы мы могли упаковать данные на стороне базы данных в такой или похожий объект и отправить их в приложение с помощью одной команды, количество обращений к базе данных значительно уменьшилось бы. К счастью, PostgreSQL обладает необходимыми возможностями для создания таких объектов:

- PostgreSQL – объектно-реляционная база данных;
- PostgreSQL позволяет создавать собственные типы;
- функции PostgreSQL могут возвращать множества, включая множества записей.

В следующих главах будут обсуждаться функции, возвращающие множества записей и поддерживающие типы JSON, JSONB и пользовательские типы данных. Также будут показаны примеры создания таких функций и использования их в приложениях.

Выводы

В этой главе обсуждались дополнительные аспекты производительности, которые обычно считаются не связанными с оптимизацией баз данных. Хотя

с технической точки зрения речь шла не об оптимизации запросов, был представлен подход к оптимизации общей производительности приложения. Как мы часто заявляли, запросы SQL не выполняются в вакууме; они являются частью приложения, и «промежуточная» область взаимодействия между приложением и базой данных часто не принимается во внимание разработчиками баз данных и приложений.

По этой причине мы позволили себе заявить права на эту неизведанную территорию и предложить пути к улучшению. Примечательно, что в этой главе не приводятся какие-либо практические решения или примеры того, «как сделать правильно». В последующих главах мы обсудим несколько методов, которые предоставляют разработчикам мощный механизм для преодоления ограничений традиционных инструментов объектно-реляционного отображения.

Глава 11

Функции

В этой главе основное внимание уделяется наиболее недооцененным и неправильно используемым объектам PostgreSQL – функциям. Поскольку все современные языки программирования позволяют определять функции, часто предполагают, что функции баз данных устроены так же и что умение писать функции на языке прикладного программирования можно применить и к PostgreSQL. Но это совсем не так.

В данной главе обсуждается, чем функции PostgreSQL отличаются от функций других языков программирования; когда их надо создавать, а когда нет; как использование функций может повысить производительность и как оно может привести к ее значительному снижению.

Прежде чем продолжить, разберемся с широко распространенным мнением, что использование функций снижает переносимость. Это правда, но учтите следующее:

- и инструкции SQL, и инструменты объектно-реляционного отображения не переносимы на 100 %; потрудиться придется в любом случае, хотя, скорее всего, не очень сильно;
- замена базы данных в существующей промышленной системе – это серьезный проект, который никогда не выполняется «на лету». Неизбежны и некоторые изменения в самом приложении. Так что преобразование функций не сильно усложняет такой проект.

Создание функций

В PostgreSQL есть и встроенные (внутренние) функции, и пользовательские функции. В этом отношении нет отличий от других языков программирования.

Встроенные функции

Встроенные функции написаны на языке C и интегрированы с сервером PostgreSQL. Для каждого типа данных, поддерживаемого PostgreSQL, существует ряд функций, которые выполняют различные операции с переменными или

значениями столбцов этого типа. Подобно императивным языкам, имеются функции для математических операций, функции для работы со строками, функции для работы с датой и временем и т. п. Более того, список доступных функций и поддерживаемых типов расширяется с каждым новым выпуском.

Примеры встроенных функций показаны в листинге 11.1.

Листинг 11.1 ❖ Примеры встроенных функций

```
sin(x);
substr(first_name,1,1);
now();
```

Пользовательские функции

Пользовательские функции – это функции, которые создает сам пользователь. PostgreSQL поддерживает три типа пользовательских функций:

- функции на языке запросов, то есть функции, написанные на SQL;
- C-функции, написанные на языке C (или C-подобных языках, например C++);
- функции, написанные на одном из поддерживаемых процедурных языков.

В листинге 11.2 представлена команда `CREATE FUNCTION`.

Листинг 11.2 ❖ Команда CREATE FUNCTION

```
CREATE FUNCTION function_name (par_name1 par_type1, ...)
RETURNS return_type
AS
    <function body>
LANGUAGE plpgsql;
```

С точки зрения PostgreSQL, движок базы данных воспринимает только сигнатуру функции – имя функции, список ее параметров (которых может и не быть) и тип возвращаемого значения (который может быть пустым), – а также некоторые спецификации, например язык, на котором написана функция. Тело функции упаковано в строковый литерал, который передается специальному обработчику, знающему, как работать с указанным языком. Обработчик может либо сам выполнять всю работу по разбору, синтаксическому анализу, выполнению и т. д., либо может служить связующим звеном между PostgreSQL и существующей реализацией языка программирования. Стандартный дистрибутив PostgreSQL поддерживает четыре процедурных языка: PL/pgSQL, PL/Tcl, PL/Perl и PL/Python. В этой книге мы будем обсуждать только функции, написанные на PL/pgSQL.

Знакомство с процедурным языком

Поскольку мы рассматриваем функции, стоит формально представить язык (или языки), на котором могут быть написаны эти функции.

До сих пор в этой книге использовался только SQL, и единственная команда, кроме CREATE, которую вы могли видеть во фрагментах кода, была SELECT. Теперь пора познакомиться с процедурными языками. В этой книге мы обсуждаем лишь встроенный в PostgreSQL процедурный язык PL/pgSQL.

Функция, написанная на PL/pgSQL, может включать в себя любые инструкции SQL (возможно, с некоторыми изменениями), управляющие структуры (IF THEN ELSE, CASE, LOOP) и вызовы других функций.

В листинге 11.3 представлен пример функции, написанной на PL/pgSQL, которая преобразует текстовую строку в число, а если это возможно, то возвращает неопределенное значение.

Листинг 11.3 ❖ Функция преобразования текстовой строки в число

```
CREATE OR REPLACE FUNCTION text_to_numeric(input_text text)
    RETURNS numeric AS
$BODY$
BEGIN
    RETURN replace(input_text, ',', '')::numeric;
EXCEPTION
    WHEN OTHERS THEN
        RETURN NULL::numeric;
END;
$BODY$
LANGUAGE plpgsql;
```

Используя информацию из листинга 11.2, мы можем назвать части, общие для всех пользовательских функций. Имя функции – text_to_numeric, и у нее есть только один параметр input_text типа text.

Предложение RETURNS определяет тип значения, возвращаемого функцией (numeric), а предложение LANGUAGE определяет язык, на котором написана функция (plpgsql).

Теперь подробнее рассмотрим тело функции.

Долларовые кавычки

В предыдущем разделе мы сказали, что тело функции представлено как строковый литерал; однако вместо кавычек он начинается и заканчивается символами $BODY$. Такая нотация в PostgreSQL называется *долларовыми кавычками*, и она особенно полезна при создании функций. Действительно, в большом тексте функции, скорее всего, встретятся одинарные кавычки или обратная косая черта, и вам придется экранировать их в каждом случае. С долларовыми кавычками вы определяете строковый литерал, используя два знака доллара, возможно, с некоторым тегом между ними.

Возможность указывать теги делает этот способ определения строковых констант особенно удобным, поскольку позволяет вкладывать строки с разными тегами. Например, начало тела функции может выглядеть, как в листинге 11.4.

Листинг 11.4 ❖ Использование вложенных долларовых кавычек

```
$function$
DECLARE
    v_error_message text := 'Ошибка:';
    v_record_id integer;
BEGIN
    ...
    v_error_message := v_error_message ||
        $em$Невозможно обновить запись № $em$ || quote_literal(v_record_id);
    ...
END;
$function$
```

Здесь для тела функции мы используем долларовые кавычки с тегом `function`. Обратите внимание, что нет правил, предписывающих для тела функции какой-то определенный тег. Можно использовать и пустой; единственное требование – строковый литерал должен заканчиваться тем же тегом, с которого он начался. В приведенных здесь примерах будут использоваться разные теги как напоминание о том, что они не предопределены.

Тело функции начинается с предложения `DECLARE`, которое является необязательным, если переменные не нужны. Ключевое слово `BEGIN` (без точки с запятой) обозначает начало раздела инструкций, а ключевое слово `END` должно быть последней инструкцией тела функции.

Этот раздел не является исчерпывающим руководством по созданию функций. Более подробную информацию можно найти в документации PostgreSQL.

Другие особенности обсуждаются по мере их появления в следующих примерах.

Параметры и возвращаемое значение

Чаще всего у функции есть один или несколько параметров, но может не быть и ни одного. Например, встроенная функция `now`, которая возвращает текущее время, не имеет параметров. Любому параметру функции мы можем присвоить значение по умолчанию, которое будет использоваться, если конкретное значение не передается явно.

Кроме того, есть несколько способов определить параметры функции. В примере из листинга 11.3 параметры имеют имена, но на них также можно ссылаться по номеру позиции (`$1`, `$2` и т. д.). Вместо указания типа возвращаемого значения функции некоторые параметры могут быть определены как `OUT` или `INOUT`. В этой главе мы не пытаемся охватить все возможные варианты, потому что производительность функций от них не зависит.

Упомянем еще, что функция может и не возвращать значение; в этом случае она определяется как `RETURNS void`. Такие функции существуют потому,

что ранее PostgreSQL не поддерживал хранимые процедуры, и упаковать несколько команд вместе можно было только внутри функции.

Перегрузка функций

Как и в других языках программирования, функции в PostgreSQL могут быть полиморфными, то есть несколько функций могут использовать одно и то же имя с разной сигнатурой. Эта возможность называется *перегрузкой функций*. Как упоминалось ранее, функция определяется именем и набором входных параметров; для разных наборов входных параметров тип возвращаемого значения может быть различным, но по очевидным причинам две функции не могут разделять одно и то же имя и один и тот же набор входных параметров.

Посмотрим на примеры из листинга 11.5. В первом случае создается функция, которая рассчитывает количество пассажиров на конкретном рейсе. Во втором случае создается одноименная функция, которая вычисляет количество пассажиров, вылетающих в определенный день из определенного аэропорта.

Однако если вы попытаетесь выполнить третий фрагмент и создать функцию, которая вычисляет количество пассажиров на конкретном рейсы на конкретную дату, то получите сообщение об ошибке:

```
ERROR: cannot change name of input parameter "p_airport_code".
```

Вы можете создать функцию с тем же именем и другим набором параметров и типом возвращаемого значения; таким образом, можно создать другую функцию с тем же именем, как показано в четвертом случае. Однако если вы попытаетесь создать функцию с тем же именем и с с теми же параметрами, но с другим типом возвращаемого значения (пятый случай), вы снова получите сообщение об ошибке:

```
ERROR: cannot change return type of existing function
```

Листинг 11.5 ❖ Перегрузка функций

```
-- № 1
CREATE OR REPLACE FUNCTION num_passengers (p_flight_id int)
RETURNS integer;

-- № 2
CREATE OR REPLACE FUNCTION num_passengers (p_airport_code text, p_departure date)
RETURNS integer;

-- № 3
CREATE OR REPLACE FUNCTION num_passengers (p_flight_no text, p_departure date)
RETURNS integer;

-- № 4
CREATE OR REPLACE FUNCTION num_passengers (p_flight_no text)
RETURNS numeric;

-- № 5
CREATE OR REPLACE FUNCTION num_passengers (p_flight_id int)
RETURNS numeric;
```

Обратите внимание, что исходный код этих функций значительно отличается. В листинге 11.6 показан исходный код функции num_passengers(integer), а в листинге 11.7 – код функции num_passengers(text,date).

Листинг 11.6 ❖ Исходный код функции num_passengers(integer)

```
CREATE OR REPLACE FUNCTION num_passengers (p_flight_id int)
RETURNS integer
AS $$
BEGIN
    RETURN (
        SELECT count(*)
          FROM booking_leg bl
               JOIN booking b USING (booking_id)
               JOIN passenger p USING (booking_id)
         WHERE flight_id = p_flight_id
    );
END;
$$ LANGUAGE plpgsql;
```

Листинг 11.7 ❖ Исходный код функции num_passengers(text,date)

```
CREATE OR REPLACE FUNCTION num_passengers (p_airport_code text, p_departure date)
RETURNS integer
AS $$
BEGIN
    RETURN (
        SELECT count(*)
          FROM booking_leg bl
               JOIN booking b USING (booking_id)
               JOIN passenger p USING (booking_id)
               JOIN flight f USING (flight_id)
         WHERE departure_airport = p_airport_code
           AND scheduled_departure BETWEEN p_departure AND p_departure + 1
    );
END;
$$ LANGUAGE plpgsql;
```

Выполнение функций

Чтобы выполнить функцию, мы используем команду SELECT. В листинге 11.8 показаны два способа выполнения функции num_passengers с параметром p_flight_id, равным 13.

Листинг 11.8 ❖ Выполнение функции

```
SELECT num_passengers(13);

SELECT * FROM num_passengers(13);
```

Для функций, возвращающих скалярные значения, любой синтаксис даст идентичные результаты. Сложные типы рассматриваются позже в этой главе.

Также стоит отметить, что определяемые пользователем скалярные функции могут использоваться в запросах точно так же, как и встроенные функции. Вспомните функцию `text_to_numeric` из листинга 11.3. Вы можете спросить, зачем создавать пользовательскую функцию преобразования, когда в PostgreSQL уже есть три разных способа преобразовать строку в целое число. Для справки, вот эти три способа:

- `CAST (text_value AS numeric);`
- `text_value::numeric` – альтернативный синтаксис для `CAST`;
- `to_number(text_value, '999999999999')` – с использованием встроенной функции.

Зачем нужна еще одна функция преобразования? Для любого из методов, перечисленных в предыдущем списке, попытка преобразования приводит к ошибке, если входная текстовая строка содержит символы, отличные от тех, что могут встречаться в числах.

Чтобы функция преобразования не завершалась ошибкой, мы включаем в тело функции *раздел обработки исключений*. Раздел начинается с ключевого слова `EXCEPTION`; ключевое слово `WHEN` может указывать конкретные типы исключений. В этой главе мы будем использовать его только в форме `WHEN OTHERS`, что соответствует всем типам исключений, не включенным в предыдущие условия `WHEN`. Если, как в листинге 11.3, конструкция `WHEN OTHERS` используется сама по себе, это означает, что все исключения должны обрабатываться одинаково.

В листинге 11.3 при ошибке преобразования (фактически при любой ошибке) функция не завершится аварийно, а вернет неопределенное значение. Почему так важно, чтобы не происходил сбой, когда функции передается «плохой» параметр? Потому что она используется в списке `SELECT`.

В главе 7 мы создали материализованное представление `passenger_passport` (см. листинг 7.11). Различные столбцы этого материализованного представления должны содержать значения разных типов данных, но поскольку в исходных данных все эти поля являются текстовыми, мы мало что можем сделать. Если вы хотите выбрать номер паспорта как числовое значение, ваш код может выглядеть так:

```
SELECT passenger_id,
       passport_num::numeric AS passport_number
  FROM passenger_passport
```

Если хотя бы в одной строке столбец `passport_num` содержит нечисловое значение (например, пробел или пустую строку), то весь запрос завершится ошибкой. Вместо этого можно использовать созданную нами функцию `text_to_integer`:

```
SELECT passenger_id,
       text_to_numeric(passport_num) AS passport_number
FROM passenger_passport
```

Создадим еще одну пользовательскую функцию, `text_to_date`, которая преобразует строку, содержащую дату, в тип `date`, см. листинг 11.9.

Листинг 11.9 ❖ Функция преобразования текстовой строки в дату

```
CREATE OR REPLACE FUNCTION text_to_date (input_text text)
RETURNS date
AS $BODY$
BEGIN
    RETURN input_text::date;
EXCEPTION
    WHEN OTHERS THEN
        RETURN NULL::date;
END;
$BODY$
LANGUAGE plpgsql;
```

Теперь мы можем использовать обе функции в листинге 11.10.

Листинг 11.10 ❖ Использование функций в списке SELECT

```
SELECT passenger_id,
       text_to_integer(passport_num) AS passport_num,
       text_to_date(passport_exp_date) AS passport_exp_date
  FROM passenger_passport
```

Хотя этот пример кажется идеальным вариантом использования функций в PostgreSQL, на самом деле с точки зрения производительности это далеко не лучшее решение, и скоро мы узнаем, почему.

Как происходит выполнение функций

В этом разделе объясняются некоторые особенности выполнения функций, специфичные для PostgreSQL. Если у вас есть опыт работы с такими СУБД, как Oracle или MS SQL Server, некоторые ваши предположения о том, как выполняются функции, могут оказаться неверны для PostgreSQL.

Первый сюрприз может поджидать вас, когда инструкция `CREATE FUNCTION` выдает сообщение о завершении вроде такого:

```
CREATE FUNCTION
Query returned successfully in 127 msec.
```

Читая сообщение, вы можете предположить, что ваша функция не содержит ошибок. Чтобы показать, что может пойти не так, выполним код из листинга 11.11. Если вы скопируете и выполните эту инструкцию, то получите сообщение об успешном создании.

Однако попытайтесь выполнить эту функцию:

```
SELECT num_passengers('ORD', '2020-07-05')
```

Вы получите сообщение об ошибке:

```
ERROR: column "airport_code" does not exist
```

Листинг 11.11 ❖ Создание функции, которое завершается без ошибок

```
CREATE OR REPLACE FUNCTION num_passengers (p_airport_code text, p_departure date)
RETURNS integer
AS $$
BEGIN
    RETURN (
        SELECT count(*)
          FROM booking_leg bl
               JOIN booking b USING (booking_id)
               JOIN passenger p USING (booking_id)
               JOIN flight f USING (flight_id)
         WHERE airport_code = p_airport_code
           AND scheduled_departure BETWEEN p_date AND p_date + 1
    );
END;
$$ LANGUAGE plpgsql;
```

Что пошло не так? Функция использует `airport_code` вместо `departure_airport`. Такую ошибку легко совершить, но может оказаться неожиданным, что PostgreSQL не сообщил о ней при создании функции.

Теперь, если вы исправите эту ошибку и снова создадите функцию (см. листинг 11.12), при ее выполнении вы получите еще одну ошибку:

```
ERROR: column "p_date" does not exist
```

Листинг 11.12 ❖ Создаем функцию: одна ошибка исправлена, осталась еще одна

```
CREATE OR REPLACE FUNCTION num_passengers (p_airport_code text, p_departure date)
RETURNS integer
AS $$
BEGIN
    RETURN (
        SELECT count(*)
          FROM booking_leg bl
               JOIN booking b USING (booking_id)
               JOIN passenger p USING (booking_id)
               JOIN flight f USING (flight_id)
         WHERE departure_airport = p_airport_code
           AND scheduled_departure BETWEEN p_date AND p_date + 1
    );
END;
$$ LANGUAGE plpgsql;
```

И PostgreSQL прав, поскольку параметр называется `p_departure_date`, а не `p_date`. Тем не менее почему сообщение об этой ошибке не появилось раньше?

Во время создания функции PostgreSQL выполняет только начальный разбор текста, в процессе которого обнаруживаются лишь тривиальные синтаксические ошибки. Все более сложное не проявится до момента выполнения. Это плохие новости, если вы только что пришли с Oracle и предполагаете, что

при создании функции движок базы данных компилирует ее и сохраняется в скомпилированном виде. Функции не только хранятся в виде исходного кода, но и, в отличие от других СУБД, они интерпретируются, а не компилируются.

Интерпретатор PL/pgSQL разбирает исходный текст функции и создает (внутреннее) дерево инструкций при первом вызове функции в каждом сеансе. Даже в этом случае отдельные выражения SQL и команды, используемые в функции, не транслируются тотчас же. Только когда выполнение доходит до определенной команды, она анализируется и для нее создается *подготовленный оператор*. Он будет использован повторно, если та же функция будет выполнена снова в том же сеансе. Одно из следствий этого состоит в том, что если ваша функция содержит условия (например, инструкции `IF THEN ELSE` или `CASE`), вы можете не обнаружить даже синтаксическую ошибку в своем коде, если не наткнулись на нее во время выполнения. Мы видели, как такие неприятные открытия совершались уже после того, как функция перешла в промышленное окружение. Подводя итог, можно сказать, что при создании функции PL/pgSQL:

1) план выполнения не сохраняется;
2) никакие проверки на наличие таблиц, столбцов или других функций не выполняются;
3) вы не знаете, работает ваша функция или нет, пока не выполните ее (и, как правило, не один раз, если есть несколько путей выполнения).

Еще одно важное свойство функций PostgreSQL, которое следует из предыдущего объяснения, состоит в том, что функции являются «атомарными» в нескольких разных смыслах. Во-первых (к разочарованию пользователей Oracle), вы не можете начинать транзакции внутри функций PostgreSQL, поэтому инструкции DML – это всегда «все или ничего». Во-вторых, когда оптимизируется план выполнения запроса, планировщик PostgreSQL ничего не знает о пользовательских функциях. Например, выполните

```
EXPLAIN SELECT num_passengers(13)
```

План выполнения будет выглядеть примерно так:

```
Result (cost=0.00..0.26 rows=1 width=4)
```

Если вам нужно выяснить, какие планы используются для выполнения запросов внутри функции, вам нужно будет подставить какие-нибудь фактические значения вместо параметров и для каждой инструкции выполнить команду `EXPLAIN`.

Одно из ключевых слов в команде `CREATE FUNCTION` (помните, мы перечислили не все из них!) – `COST`. Оно позволяет разработчику явно установить стоимость выполнения функции, которая будет использоваться оптимизатором. Значение по умолчанию – 100, и мы не рекомендуем изменять его, если у вас нет на то действительно веских причин.

Функции и производительность

Закончим на этом наше краткое введение. Пришло время обратиться к центральной теме этой книги: как функции влияют на производительность? В главе 7 рассматривалась декомпозиция кода и в общих чертах обрисовывались различные последствия декомпозиции в императивных языках и в SQL. Было рассмотрено несколько возможных методов, а функции были упомянуты как заслуживающие более подробного обсуждения, которое и следует ниже.

Зачем создавать функции в PostgreSQL? В императивных языках использование функций является очевидным выбором: функции повышают читаемость кода, облегчают повторное использование и не оказывают отрицательного влияния на производительность. Напротив, функции в PostgreSQL могут как улучшить читаемость кода, так и ухудшить ее и могут значительно снизить производительность. Обратите внимание на слово «могут»; остальная часть главы посвящена способам разумного использования функций, повышающим производительность, а не снижающим ее.

Как использование функций может ухудшить производительность

В предыдущем разделе мы создали функцию `num_passengers(int)`, которая вычисляет количество пассажиров на заданном рейсе. Эта функция отлично подходит для отдельного рейса, возвращая результат за 150 мс.

Давайте посмотрим, что произойдет, если эту функцию включить в список `SELECT`. В листинге 11.13 выбираются все рейсы, вылетевшие из аэропорта О'Хара в период с 5 по 13 июля, и для каждого из этих рейсов рассчитывается количество пассажиров.

Листинг 11.13 ❖ Использование функции в списке SELECT снижает производительность

```
SELECT flight_id,
       num_passengers(flight_id) AS num_pass
  FROM flight f
 WHERE departure_airport = 'ORD'
   AND scheduled_departure BETWEEN '2020-07-05' AND '2020-07-13'
```

Время выполнения этой инструкции составляет 3,5 секунды. Если же вместо функции используется инструкция SQL, выполняющая точно такие же вычисления (листинг 11.14), время выполнения составит около 900 мс.

Откуда такая большая разница? В главе 7 мы объяснили, что представления и общие табличные выражения могут работать как оптимизации. Этот эффект еще более выражен в случае функций. Поскольку функция является настоящим черным ящиком для объемлющей инструкции SQL, PostgreSQL остается только выполнить каждую функцию столько раз, сколько выбрано строк.

Листинг 11.14 ❖ Те же результаты без использования функции

```
SELECT f.flight_id,
       count(*) AS num_pass
  FROM booking_leg bl
       JOIN booking b USING (booking_id)
       JOIN passenger p USING (booking_id)
       JOIN flight f USING (flight_id)
 WHERE departure_airport = 'ORD'
   AND scheduled_departure BETWEEN '2020-07-05' AND '2020-07-13'
 GROUP BY 1
```

Говоря точнее, немного времени экономится за счет того, что для последующих вызовов функций из того же сеанса PostgreSQL использует подготовленный оператор, но этот факт может как ускорить, так и замедлить выполнение, потому что план выполнения может не учитывать различия в статистике между вызовами функций.

Разница во времени выполнения между 0,9 секунды и 3,5 секунды может показаться не такой уж большой, и можно счесть, что допустимо некоторое замедление ради простоты сопровождения кода, но обратите внимание, что между 0,9 и 3,5 секунды лежит то пороговое значение времени, которое пользователь готов ждать. К тому же в этом примере запрос внутри функции довольно легкий и выполняется за миллисекунды.

Хорошо, мы поняли, что выполнение запросов, встроенных в список SELECT другой команды, – не лучшая идея. Но как насчет функций, выполняющих простые преобразования данных, вроде тех, которые мы создали для приведения типов? В этом случае разница может быть не такой значительной, пока запрос возвращает не слишком много строк, но она все равно будет видна.

Сравним время выполнения команды из листинга 11.10 со временем выполнения команды из листинга 11.15.

Листинг 11.15 ❖ Выбор паспортных данных без приведения типов

```
SELECT passenger_id,
       passport_num,
       passport_exp_date
  FROM passenger_passport
```

Оба они выбирают данные из одной таблицы и не применяют никаких фильтров, поэтому единственные временные затраты будут связаны с выполнением функций из списка SELECT. Материализованное представление `passenger_passport` содержит более 16 млн строк. Время выполнения команды из листинга 11.15 составляет 41 секунду. Если применить приведение типов без вызова функции (листинг 11.16), то время выполнения составит две минуты.

При выполнении команды из листинга 11.10 время выполнения составит более девяти минут!

Листинг 11.16 ❖ Выбор паспортных данных с приведением типов

```
SELECT passenger_id,
       passport_num::numeric,
       passport_exp_date::date
  FROM passenger_passport
```

В данном конкретном случае мало что можно сделать для повышения производительности, кроме перехода на более подходящую схему данных, но позже в этой книге мы рассмотрим другие примеры, в которых некоторые улучшения производительности возможны.

Могут ли функции улучшить производительность?

После стольких примеров отрицательного влияния функций на производительность можно задаться вопросом, существуют ли вообще условия, при которых функции могут улучшить производительность. Как и во многих других случаях, это зависит от обстоятельств.

Если мы говорим об улучшении производительности отдельной инструкции SQL, то включение ее в функцию не может ускорить выполнение. Однако функции могут быть чрезвычайно полезны, когда оптимизируется процесс.

Функции и пользовательские типы

До сих пор во всех примерах созданные нами функции возвращали скалярные значения. Теперь давайте посмотрим, какие дополнительные преимущества предоставляют функции, возвращающие пользовательские типы данных.

Пользовательские типы данных

В дополнение к собственному богатому набору типов данных PostgreSQL позволяет создавать практически неограниченное количество пользовательских типов.

Пользовательские типы могут быть простыми или составными. Простые пользовательские типы включают в себя домены, перечисления и диапазоны.

Ниже приведены примеры создания простых типов:

```
CREATE DOMAIN timeperiod AS tstzrange;
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
CREATE TYPE mood_range AS RANGE...
CREATE TYPE <base type>
```

Так же, как мы можем определять *массивы* базовых типов, можно определять и массивы пользовательских типов:

```
DECLARE
    v_moods_set mood[];
```

Составные типы дают нам еще больше возможностей.

Составной тип представляет собой строку (как в таблице) или запись. Определение типа состоит из последовательности имен полей и соответствующих им типов данных. Например, листинг 11.17 определяет тип `boarding_pass_record`.

Листинг 11.17 ❖ Тип boarding_pass_record

```
CREATE TYPE boarding_pass_record AS (
    boarding_pass_id int,
    booking_leg_id int,
    flight_no text,
    departure_airport text,
    arrival_airport text,
    last_name text,
    first_name text,
    seat text,
    boarding_time timestamptz
)
```

Теперь, когда определен тип `boarding_pass_record`, мы можем объявлять переменные этого типа так же, как можно объявлять переменные базовых типов:

```
DECLARE
    v_new_boarding_pass_record boarding_pass_record;
```

Более того, мы можем создавать функции, возвращающие *множества составных типов*.

Функции, возвращающие составные типы

Почему так важен тот факт, что функции могут возвращать множества составных типов? Зачем это нужно? Вспомните, что в главе 9 мы пришли к необходимости извлекать из базы данных весь объект, а не просто один компонент за другим. Теперь все, что обсуждалось ранее, можно собрать воедино.

Приведем пример. В листинге 11.18 мы представляем функцию, которая возвращает все посадочные талоны для указанного рейса.

Чтобы вызвать эту функцию, выполните:

```
SELECT * FROM boarding_passes_flight(13);
```

Результат представлен на рис. 11.1.

Листинг 11.18 ❖ Функция, возвращающая все посадочные талоны для рейса

```
CREATE OR REPLACE FUNCTION boarding_passes_flight (p_flight_id int)
RETURNS SETOF boarding_pass_record
AS $body$
BEGIN
    RETURN QUERY
        SELECT pass_id,
               bp.booking_leg_id,
               flight_no,
               departure_airport::text,
               arrival_airport::text,
               last_name,
               first_name,
               seat,
               boarding_time
          FROM flight f
               JOIN booking_leg bl USING (flight_id)
               JOIN boarding_pass bp USING (booking_leg_id)
               JOIN passenger USING (passenger_id)
         WHERE bl.flight_id = p_flight_id;
END;
$body$
LANGUAGE plpgsql;
```

	boarding_pass_id integer	booking_leg_id bigint	flight_no text	departure_airport text	arrival_airport text	last_name text	first_name text	seat text	boarding_time timestamp with time zo
1	215158	15576	1699	KTM	CTU	LOVETT	MICHAEL	0B	2020-06-01 23:40:00-05
2	215159	15578	1699	KTM	CTU	JOSEPH	AMELIA	0C	2020-06-01 23:40:00-05
3	215160	15580	1699	KTM	CTU	PETERS	ARIAH	0D	2020-06-01 23:40:00-05
4	215161	15582	1699	KTM	CTU	MC DONALD	EVERLY	0E	2020-06-01 23:40:00-05
5	215162	15584	1699	KTM	CTU	NO	ALDEN	0F	2020-06-01 23:40:00-05

Рис. 11.1 ❖ Результат выполнения функции `boarding_passes_flight`

Теперь создадим еще одну функцию, которая будет выбирать только один посадочный талон по идентификатору `pass_id`. Обратите внимание: поскольку обе функции принимают один целочисленный параметр, перегрузка в этом случае невозможна. Новая функция показана в листинге 11.19.

Выполним эту функцию:

```
SELECT * FROM boarding_passes_pass(215158);
```

Результатом выполнения будет набор, состоящий только из одной строки, но его структура будет такой же (см. рис. 11.2).

	boarding_pass_id integer	booking_leg_id bigint	flight_no text	departure_airport text	arrival_airport text	last_name text	first_name text	seat text	boarding_time timestamp with time zor
1	215158	15576	1699	KTM	CTU	LOVETT	MICHAEL	0B	2020-06-01 23:40:00-05

Рис. 11.2 ❖ Результат выполнения функции `boarding_passes_pass`

Листинг 11.19 ❖ Функция, возвращающая один посадочный талон

```
CREATE OR REPLACE FUNCTION boarding_passes_pass (p_pass_id int)
RETURNS SETOF boarding_pass_record
AS $body$
BEGIN
    RETURN QUERY
        SELECT pass_id,
               bp.booking_leg_id,
               flight_no,
               departure_airport::text,
               arrival_airport::text,
               last_name,
               first_name,
               seat,
               boarding_time
          FROM flight f
               JOIN booking_leg bl USING (flight_id)
               JOIN boarding_pass bp USING (booking_leg_id)
               JOIN passenger USING (passenger_id)
         WHERE pass_id = p_pass_id;
END;
$body$
LANGUAGE plpgsql;
```

Почему использование этих функций могло бы улучшить производительность? Как мы обсуждали в главе 10, приложения редко выполняют инструкции SQL напрямую; вместо этого они часто используют инструкции SQL, генерируемые инструментами объектно-реляционного отображения. Скорее всего, в этом случае обращение к таблицам `boarding passes`, `passengers` и `flights` будет выполняться иначе. Для выбора тех же данных, которые возвращает функция `boarding_passes_flight`, нам, вероятно, понадобится один метод для выбора аэропортов отправления и назначения и запланированного времени вылета по номеру рейса, еще один метод потребуется для выбора всех сегментов бронирования для этого рейса, еще один метод – для посадочных талонов и еще один – для информации о пассажирах. Если удастся убедить разработчиков приложений, то, объединив все это в единую функцию, вы значительно повысите производительность.

Выбор всех посадочных талонов для рейса с 600 пассажирами с помощью функции занимает 220 мс: `SELECT * FROM boarding_passes_flight(13650)`. С другой стороны, любая отдельная инструкция `SELECT` из любой таблицы занимает около 150 мс. Поскольку каждое обращение к таблице возвращает данные в приложение, совершая путь к серверу баз данных и обратно, время выполнения суммируется, и несколько вызовов очень быстро превысят время выполнения функции.

Ранее мы уже выяснили, что для скалярных функций нет разницы между синтаксисом `SELECT * FROM имя_функции` и `SELECT имя функции`. Но когда функция возвращает составной тип, разница появляется.

На рис. 11.1 показаны результаты выполнения команды

```
SELECT * FROM boarding_passes_flight(13)
```

На рис. 11.3 показаны результаты команды

```
SELECT boarding_passes_flight(13)
```

	boarding_passes_flight postgres_air.boarding_pass_record
1	(215158,15576,1699,KTM,CTU,LOVETT,MICHAEL,0B,"2020-06-01 23:40:00-05")
2	(215159,15578,1699,KTM,CTU,JOSEPH,AMELIA,0C,"2020-06-01 23:40:00-05")
3	(215160,15580,1699,KTM,CTU,PETERS,ARIAH,0D,"2020-06-01 23:40:00-05")
4	(215161,15582,1699,KTM,CTU,"MC DONALD",EVERLY,0E,"2020-06-01 23:40:00-05")
5	(215162,15584,1699,KTM,CTU,NO,ALDEN,0F,"2020-06-01 23:40:00-05")

Рис. 11.3 ❖ Результаты функции в виде набора записей

Использование составных типов с вложенной структурой

Можно ли использовать составные типы как элементы других составных типов? Да, PostgreSQL это позволяет.

На рис. 10.4 мы представили структуру сложного объекта `booking_record`. Одна из его составляющих – сложный объект `booking_leg_record`. Чтобы создать представление этого объекта в виде составного типа, начнем с типов `flight_record` и `boarding_pass_record`, а затем построим на их основе тип `booking_leg_record`, как показано в листинге 11.20.

Тип `booking_leg_record` содержит в качестве одного из своих элементов составной тип `flight_record`, а в качестве другого компонента – массив элементов `boarding_pass_record`.

Похоже, мы решили задачу, изложенную в главе 10: мы можем создавать составные типы с вложенной структурой и создавать функции, возвращающие такие объекты. Однако осталось решить еще много других проблем.

Чтобы проиллюстрировать эти проблемы, создадим функцию, которая будет возвращать весь объект `booking_leg_record` по идентификатору `booking_leg_id`. Код этой функции представлен в листинге 11.21.

Не пугайтесь – функция длинная, но не слишком сложная. Рассмотрим ее подробнее.

Основной запрос извлекает данные из таблицы `booking_leg`, используя значение параметра функции в качестве критерия поиска. Первые три элемента записи – `booking_leg_id`, `leg_num`, `booking_id` – берутся непосредственно из таблицы `booking_leg`. Следующий элемент записи – `flight_record`, где `flight_id` – идентификатор рейса из выбранного сегмента бронирования. Это условие задается в предложении `WHERE` подзапроса:

```
WHERE flight_id = bl.flight_id
```

Листинг 11.20 ❖ Дополнительные определения типов записей

```
CREATE TYPE flight_record AS (
    flight_id int,
    flight_no text,
    departure_airport_code text,
    departure_airport_name text,
    arrival_airport_code text,
    arrival_airport_name text,
    scheduled_departure timestamptz,
    scheduled_arrival timestamptz
);

CREATE TYPE boarding_pass_record AS (
    boarding_pass_id integer,
    booking_leg_id bigint,
    flight_no text,
    departure_airport text,
    arrival_airport text,
    last_name text,
    first_name text,
    seat text,
    boarding_time timestamptz
);

CREATE TYPE booking_leg_record AS (
    booking_leg_id int,
    leg_num int,
    booking_id int,
    flight flight_record,
    boarding_passes boarding_pass_record[]
);
```

Мы выбираем информацию о рейсе, на который ссылается выбранный сегмент бронирования.

Встроенная функция `row` создает строку из набора элементов, и эта строка приводится к типу `flight_record`, ожидаемому в `booking_leg_record`.

Последний элемент `booking_leg_record` – это массив посадочных талонов. Его размер определяется количеством пассажиров в бронировании. Рассмотрим подробнее этот подзапрос:

```
( SELECT array_agg (row(
            pass_id,
            bp.booking_leg_id,
            flight_no,
            departure_airport ,
            arrival_airport,
            last_name,
            first_name,
            seat,
            boarding_time
         )::boarding_pass_record)
```

```
    FROM flight f1
         JOIN boarding_pass bp ON f1.flight_id = bl.flight_id
                              AND bp.booking_leg_id = bl.booking_leg_id
         JOIN passenger p ON p.passenger_id = bp.passenger_id
)
```

Листинг 11.21 ❖ Функция, возвращающая сложный объект с вложенной структурой

```
CREATE OR REPLACE FUNCTION booking_leg_select (p_booking_leg_id int)
RETURNS SETOF booking_leg_record
AS $body$
BEGIN
   RETURN QUERY
   SELECT bl.booking_leg_id,
          leg_num,
          bl.booking_id,
          ( SELECT row(
                      flight_id,
                      flight_no,
                      departure_airport,
                      da.airport_name,
                      arrival_airport,
                      aa.airport_name,
                      scheduled_departure,
                      scheduled_arrival
                    )::flight_record
              FROM flight f
                   JOIN airport da ON da.airport_code = departure_airport
                   JOIN airport aa ON aa.airport_code = arrival_airport
             WHERE flight_id = bl.flight_id
          ),
          ( SELECT array_agg (row(
                      pass_id,
                      bp.booking_leg_id,
                      flight_no,
                      departure_airport ,
                      arrival_airport,
                      last_name,
                      first_name,
                      seat,
                      boarding_time
                    )::boarding_pass_record)
              FROM flight f1
                   JOIN boarding_pass bp ON f1.flight_id = bl.flight_id
                                        AND bp.booking_leg_id = bl.booking_leg_id
                   JOIN passenger p ON p.passenger_id = bp.passenger_id
          )
     FROM booking_leg bl
    WHERE bl.booking_leg_id = p_booking_leg_id;
END;
$body$
LANGUAGE plpgsql;
```

Бросается в глаза, что по сути тот же самый запрос мы уже использовали в функции `boarding_pass_flight`. Отличия заключаются в следующем:

- не требуется соединение с таблицей `booking_leg`, так как она уже была выбрана во внешнем запросе. Информация из таблицы `flight` нам нужна, но мы можем использовать `flight_id` из выбранного сегмента бронирования. Таким образом, получается декартово произведение с одной строкой из таблицы `flight`;
- точно так же для посадочного талона нет соединения с таблицей `booking_leg`; мы просто используем имеющийся `booking_leg_id`.

Наконец, мы используем встроенную функцию `array_agg` для создания единого *массива записей*, который ожидается в качестве последнего элемента `booking_leg_record`.

Примечание Здесь показан лишь один из многих способов создания объекта с вложенной структурой. В следующих главах мы представим другие способы, которые могут оказаться более полезными в иных ситуациях.

А теперь плохие новости. Мы приложили столько усилий, чтобы создать эту функцию, но результат ее выполнения, показанный на рис. 11.4, несколько разочаровывает:

```
SELECT * FROM booking_leg_select (17564910)
```

	booking_leg_id integer	leg_num integer	booking_id integer	flight postgres_air.flight_record	boarding_passes postgres_air.boarding_pass_record[]
1	17564910	2	232346	(13650,1245,JFK,"John F Kennedy ...	{"(1247796,17564910,1245,JFK,CDG,LEWIS,ELIA,1E,\"2020-06-04 23:50:00-05\")","(124779...

Рис. 11.4 ❖ Возвращенный сложный объект с вложенной структурой

Результат выглядит так, как мы хотели, но есть одна важная деталь. Для скалярных элементов PostgreSQL сохраняет имена (как если бы мы выбирали их из таблицы), но структура элементов, которые сами являются сложными объектами, не раскрывается. Обратите внимание, что внутри PostgreSQL по-прежнему сохраняет представление о структуре вложенного типа, но не передает ее на верхний уровень.

Почему это плохо? В главе 10 были рассмотрены подводные камни объектно-реляционного отображения, и на рис. 10.4 была дана схема возможного решения. На тот момент мы не обсуждали детали реализации, но с функциями, которые могут возвращать сложные типы, цель кажется достижимой. Однако возвращаемое функцией значение бесполезно для приложения, если оно не может определить ни имя элемента, ни его тип.

Решение приводится в главе 13, но пока давайте сосредоточимся на функциях, возвращающих записи без вложенной структуры.

Функции и зависимости типов

В главе 7 упоминаются зависимости в контексте обычных и материализованных представлений. Определение материализованных представлений нельзя изменить без предварительного удаления объекта, а обычные представления придется удалить, если изменяется набор столбцов. Это означает, что в таких случаях все зависимые объекты должны быть удалены и созданы заново. Если эти зависимые объекты, в свою очередь, используются в других представлениях или материализованных представлениях, их зависимые объекты также должны быть удалены.

Это может привести к очень нежелательным последствиям. Мы наблюдали ситуации в промышленных системах, когда одно изменение приводило к удалению более 60 зависимых объектов, которые приходилось создавать заново в определенном порядке.

К счастью, у нас нет такой проблемы с функциями. Поскольку инструкции SQL в теле функции не разбираются при создании функции, то нет и зависимостей от таблиц, представлений, материализованных представлений или других функций и хранимых процедур, которые используются в теле функции. По этой причине функции нужно пересоздавать только тогда, когда это действительно необходимо, а не просто из-за каскадного удаления.

Однако функции создают новый тип зависимостей: функции зависят от типов возвращаемых значений, включая пользовательские типы. Как и материализованные представления, пользовательские типы данных нельзя изменить без предварительного удаления. Чтобы удалить тип, придется удалить и все другие пользовательские типы, которые включают его как элемент, и все функции, которые зависят от того типа. Это может показаться еще более серьезной проблемой, но на самом деле это полезное свойство. Если пользовательский тип изменен, некоторые его элементы, скорее всего, были добавлены, удалены или изменены. А это означает, что запросы, возвращающие данный тип записи, должны быть пересмотрены, поэтому функции следует удалить.

Кроме того, в отличие от создания материализованного представления, которое может занять некоторое время, создание функции происходит практически мгновенно.

Управление данными с помощью функций

До сих пор в этой главе рассматривались только функции, выбирающие данные. Но функции PL/pgSQL позволяют выполнять любую команду SQL, включая команды DML.

В листинге 11.22 показана функция, которая выдает пассажиру новый посадочный талон.

Листинг 11.22 ❖ Создаем новый посадочный талон

```
CREATE OR REPLACE FUNCTION issue_boarding_pass (
    p_booking_leg_id int,
    p_passenger_id int,
    p_seat text,
    p_boarding_time timestamptz
)
RETURNS SETOF boarding_pass_record
AS $body$
DECLARE
    v_pass_id int;
BEGIN
    INSERT INTO boarding_pass (
        passenger_id,
        booking_leg_id,
        seat,
        boarding_time,
        update_ts
    ) VALUES (
        p_passenger_id,
        p_booking_leg_id,
        p_seat,
        p_boarding_time,
        now()
    )
    RETURNING pass_id INTO v_pass_id;
    RETURN QUERY
        SELECT * FROM boarding_passes_pass(v_pass_id);
END;
$body$
LANGUAGE plpgsql;
```

Обратите внимание, что в теле вызывается функция `boarding_passes_pass`. Ее мы создали ранее, но даже если бы такой функции не существовало, команда `CREATE FUNCTION` не сигнализировала бы об ошибке до момента выполнения. У такого поведения есть свои плюсы и минусы. Оно дает больше гибкости во время разработки, но может и создавать проблемы, поскольку легко не заметить, что использованная функция была удалена или не работает должным образом.

Созданная функция вызывается аналогично любой другой функции:

```
SELECT * FROM issue_boarding_pass(175820, 462972, '22C',
                                  '2020-06-16 21:45'::timestamptz)
```

Обратите внимание, что это не имеет особого смысла, поскольку рейс уже давно завершился, так что этот вызов приведен здесь только в качестве примера. На рис. 11.5 представлен результат этого выполнения – данные имеют тот же формат, что и для других функций, возвращающих посадочные талоны.

При создании этой функции мы сделали предположения, которые не выполняются в реальной жизни. Например, функция не проверяет, был ли уже выдан посадочный талон для этого пассажира на данный рейс, не проверяет

занятость места и не перехватывает возможные ошибки при вставке. В промышленном окружении эта функция получилась бы намного сложнее.

	boarding_pass_id integer	booking_leg_id bigint	flight_no text	departure_airport text	arrival_airport text	last_name text	first_name text	seat text	boarding_time timestamp with time zo
1	25293491	175820	68	AMS	FRA	WOOD	KAITLYN	22C	2020-06-16 21:45:00-05

Рис. 11.5 ❖ Функция DML, возвращающая пользовательский тип

Функции и безопасность

В этой книге мы не рассматриваем управление доступом к данным в PostgreSQL в основном потому, что эта тема не связана с производительностью. Однако мы немного расскажем о настройке безопасности для функций PostgreSQL и интересной связи между этими настройками и производительностью.

Одним из параметров команды `CREATE FUNCTION`, который ранее не рассматривался, является `SECURITY`. Для него есть только два допустимых значения: `INVOKER` и `DEFINER`. Первое – значение по умолчанию; оно указывает, что функция будет выполняться с использованием набора привилегий того пользователя, который вызывает функцию. Это означает, что для выполнения функции пользователь должен иметь подходящий доступ ко всем объектам базы данных, которые используются в теле функции. Если мы явно укажем `SECURITY DEFINER`, функция будет выполняться с полномочиями пользователя, создавшего функцию. Обратите внимание, что в отличие от полномочий на другие объекты базы данных привилегия выполнения любой функции по умолчанию предоставляется роли `PUBLIC`.

И авторы, и многие из вас бывали в ситуации, когда опытному бизнес-пользователю нужен доступ к некоторым критически важным данным, но вы не хотите давать ему полный доступ на чтение, потому что не совсем уверены в его навыках работы с SQL и опасаетесь, что его запросы могут навредить всей системе.

В этом случае можно найти компромисс – создать функцию с параметром `SECURITY DEFINER`, которая извлекает все необходимые данные с помощью эффективного запроса, и затем предоставить этому пользователю полномочия на выполнение, предварительно отозвав такие полномочия у всех остальных. Последовательность действий представлена в листинге 11.23.

Листинг 11.23 ❖ Использование функции с правами создавшего

```
CREATE FUNCTION critical_function (par1 ...)
RETURNING SETOF ...
AS $FUNC$
   ...
$FUNC$
LANGUAGE plpgsql
SECURITY DEFINER;
--
REVOKE EXECUTE ON critical_function (par1 ...) FROM public;
GRANT EXECUTE ON critical_function (par1 ...) TO powerbusinessuser;
```

Как насчет бизнес-логики?

Если вы сможете убедить разработчиков приложений использовать функции для обмена данными с базой, производительность значительно вырастет. Сам факт устранения множественных обращений к серверу может легко улучшить производительность приложения в десятки или даже сотни раз, если измерять время отклика приложения, а не время отклика базы данных.

Одним из самых серьезных препятствий на пути к успеху является концепция *бизнес-логики*. Одно из определений бизнес-логики (данное на сайте Investopedia.com) звучит так:

> *Бизнес-логика – это настраиваемые правила или алгоритмы, которые управляют обменом информацией между базой данных и пользовательским интерфейсом. Бизнес-логика – это та часть компьютерной программы, которая содержит информацию (в виде бизнес-правил), определяющую или ограничивающую работу бизнеса.*

Бизнес-логика часто рассматривается как отдельный уровень приложения, и когда мы помещаем «слишком много логики» в функции базы данных, это расстраивает разработчиков приложений.

Мы потратили много времени, пытаясь найти общий язык как с бизнесом, так и с разработчиками приложений. Результат этих обсуждений можно резюмировать следующим образом:

- нам нужна часть бизнес-логики для выполнения соединений и выборок;
- преобразования полученных данных не обязательно должны выполняться на стороне базы данных.

На практике это означает, что решения о том, что переносить в базу данных, а что должно оставаться в приложении, надо принимать на основании того, улучшит ли такой перенос производительность (облегчит ли соединения или позволит использовать индексы). Если да, то логика перемещается в функцию и считается «логикой базы данных»; в противном случае данные возвращаются в приложение для дальнейшей обработки бизнес-логикой.

Например, для приложения бронирования авиабилетов можно создать функцию получения доступных поездок, то есть потенциальных бронирований. Параметры этой функции включают пункты отправления и назначения и даты начала и завершения поездки. Чтобы иметь возможность эффективно извлекать все возможные поездки, функция должна знать, как соединить таблицы `airport` и `flight` и как рассчитать продолжительность полета. Вся эта информация принадлежит логике базы данных.

Однако мы не хотим, чтобы функция принимала окончательное решение о том, какую поездку выбрать. Окончательные критерии отбора могут меняться и обрабатываются приложением; они принадлежат бизнес-логике.

Последовательное применение этого критерия можно быстро включить в обычный цикл разработки, и это поощряет разрабатывать приложения правильно с самого начала.

Функции в системах OLAP

Надеемся, что к этому времени мы убедили вас в том, что использование функций PostgreSQL в системах OLTP выгодно. А как насчет OLAP?

Если вы с этим не сталкивались, то можете не знать, что многие инструменты создания отчетов, в том числе Cognos, Business Objects и Looker, могут представлять результаты выполнения функций. По факту выполнение функции, возвращающей набор записей, аналогично выполнению `SELECT * FROM <таблица>`.

Однако тот факт, что программное обеспечение может что-то делать, не означает, что именно так и надо делать. Так в чем же состоит преимущество использования функций в окружении OLAP?

Параметризация

Представление или материализованное представление нельзя параметризовать. Это может не представлять сложностей, если мы хотим выполнить отчет на самую последнюю дату, за вчерашний день, последнюю неделю и т. д., потому что мы можем использовать такие встроенные функции, как `CURRENT_DATE` или `CURRENT_TIMESTEMP`. Но такой отчет не получится повторно запустить за любой из прошедших временных интервалов без внесения изменений в представление. Например, представление может включать условие

```
WHERE scheduled_departure BETWEEN CURRENT_DATE-7 AND CURRENT_DATE
```

Тогда вам придется пересоздать представление, чтобы получить данные для других дат. Но если этот запрос упакован в функцию `recent_flights(p_period_start date)`, вы можете просто выполнить ее с разными значениями параметра:

```
SELECT * FROM recent_flights(CURRENT_DAY)
SELECT * FROM recent_flights('2020-08-01')
```

Отсутствие явной зависимости от таблиц и представлений

Если отчет выполняется как вызов функции, его можно оптимизировать без необходимости удалять и пересоздавать. Более того, можно изменить базовые таблицы или вообще использовать другие таблицы, а конечный пользователь ничего не заметит.

Возможность выполнять динамический SQL

Это еще одна исключительно мощная возможность PostgreSQL, которую часто недооценивают и которая обсуждается более подробно в главе 12.

Хранимые процедуры

К большому разочарованию первых последователей, пришедших из коммерческих систем, в PostgreSQL какое-то время не было хранимых процедур, Что касается авторов, то нас особенно разочаровала атомарность функций, не позволяющая управлять транзакциями, в том числе фиксировать промежуточные результаты.

Функции, не возвращающие результат

Какое-то время у разработчиков PostgreSQL не было другого выбора, кроме как использовать функции вместо хранимых процедур. Для этого можно использовать функции, возвращающие значение типа `void`, например

```
CREATE OR REPLACE function cancel_flight (p_filght_id int)
RETURNS void
AS <...>
```

В PL/pgSQL есть также альтернативный способ вызова функций:

```
PERFORM issue_boarding_pass (175820,462972, '22C', '2020-06-16 21:45'::timestamptz)
```

При этом функция выполнится и создаст посадочный талон, но не вернет результат.

Функции и хранимые процедуры

Разница между функциями и хранимыми процедурами заключается в том, что процедуры не возвращают никаких значений; таким образом, мы не указываем возвращаемый тип. В листинге 11.24 представлена команда `CREATE PROCEDURE`, которая очень похожа на команду `CREATE FUNCTION`.

Листинг 11.24 ❖ Команда `CREATE PROCEDURE`

```
CREATE PROCEDURE procedure_name (par_name1 par_type1, ...)
AS
    <procedure body>
LANGUAGE plpgsql;
```

Синтаксис здесь такой же, как и у функции, за исключением того, что нет необходимости в типе возвращаемого значения. Все предыдущие разделы о внутреннем устройстве функций также применимы и к хранимым процедурам. Для выполнения хранимой процедуры используется команда `CALL`:

```
CALL cancel_flight(13);
```

Управление транзакциями

Наиболее важное различие между выполнением функций и хранимых процедур заключается в том, что в теле процедуры можно фиксировать или откатывать транзакции.

В начале выполнения процедуры начинается новая транзакция, и любая команда `COMMIT` или `ROLLBACK` в теле процедуры завершает текущую транзакцию и начинает новую. Один из вариантов использования этой возможности – массовая загрузка данных. Мы считаем полезным фиксировать изменения частями разумного размера, например каждые 50 000 записей. Структура хранимой процедуры может выглядеть, как в листинге 11.25.

Листинг 11.25 ❖ Пример хранимой процедуры с транзакциями

```
CREATE PROCEDURE load_with_transform()
AS $load$
DECLARE
    v_cnt int := 0;
    v_record record;
BEGIN
    FOR v_record IN (SELECT * FROM data_source) LOOP
        PERFORM transform (v_rec.id);
        CALL insert_data (v_rec.*);
        v_cnt := v_cnt + 1;
        IF v_cnt >= 50000 THEN
            COMMIT;
            v_cnt := 0;
        END IF;
    END LOOP;
    COMMIT;
END;
$load$
LANGUAGE plpgsql;
```

В этом примере данные обрабатываются перед загрузкой и фиксируются, когда мы обработаем очередные 50 000 записей. Дополнительная фиксация после выхода из цикла необходима для оставшихся записей, обработанных после последней фиксации в цикле.

Обратите внимание, что если внутри процедуры не было выдано никаких команд управления транзакциями, то все операции будут обрабатываться как часть внешней транзакции, то есть транзакции, инициировавшей выполнение.

Обработка исключений

Как и в случае с функциями, вы можете указать инструкции по обработке определенных исключительных ситуаций. В листинге 11.3 мы представили пример обработки исключения в функции. Аналогичная обработка исключений может выполняться в процедурах.

Кроме того, можно создавать внутренние блоки внутри тела функции или процедуры и в каждом из них обрабатывать исключения по-своему. Структура тела процедуры для этого случая показана в листинге 11.26.

Листинг 11.26 ❖ Вложенные блоки в теле процедуры

```
CREATE PROCEDURE multiple_blocks
AS $mult$
BEGIN
    -- случай № 1
    BEGIN
        <...>
    EXCEPTION
        WHEN OTHERS THEN
            RAISE NOTICE 'CASE#1";
    END; --  случай № 1
    BEGIN
        -- случай № 2
        BEGIN
            <...>
        EXCEPTION
            WHEN OTHERS THEN
                RAISE NOTICE 'CASE#2";
        END; -- случай № 2
        -- случай № 3
        BEGIN
            <...>
        EXCEPTION
            WHEN OTHERS THEN
                RAISE NOTICE 'CASE#3";
        END; -- случай № 3
    END;
END;
$mult$
LANGUAGE plpbsql;
```

Обратите внимание, что `BEGIN` в теле процедуры отличается от команды `BEGIN`, которая начинает транзакцию.

Выводы

Функции и хранимые процедуры в PostgreSQL – исключительно мощные инструменты, которые зачастую игнорируются разработчиками баз данных. Они могут как существенно улучшить, так и сильно ухудшить производительность и успешно использоваться как в окружении OLTP, так и OLAP.

Эта глава представляет собой краткий обзор различных способов применения функций. Дополнительные сведения о том, как определять и использовать функции и хранимые процедуры, можно найти в документации PostgreSQL.

Глава 12

Динамический SQL

Что такое динамический SQL

Динамический SQL – это любая инструкция SQL, которая сначала создается как текстовая строка, а затем выполняется с помощью команды EXECUTE. Пример динамического SQL показан в листинге 12.1. Открывающиеся при этом возможности недостаточно используются в большинстве СУБД, и особенно в PostgreSQL. Рекомендации, представленные в этой главе, идут вразрез с тем, что говорится во многих учебниках по базам данных, но, как и в предыдущих случаях, все советы основаны исключительно на нашем практическом опыте.

Листинг 12.1 ❖ Динамический SQL

```
DECLARE
    v_sql text;
    cnt int;
BEGIN
    v_sql := $$SELECT count(*) FROM booking WHERE booking_ref = '0Y7W22'$$;
    EXECUTE v_sql INTO cnt;
END;
```

Почему в Postgres это работает лучше

Вы можете задаться вопросом, что же такого особенного в PostgreSQL по сравнению с другими СУБД, что приведенные здесь рекомендации так отличаются от общепринятого мнения. Обратите внимание на следующие моменты.

Во-первых, в PostgreSQL планы выполнения не кешируются даже для *подготовленных операторов* (то есть запросов, которые предварительно разбираются, анализируются и перезаписываются с помощью команды PREPARE). Это означает, что оптимизация всегда происходит непосредственно перед выполнением.

Во-вторых, этап оптимизации в PostgreSQL происходит позже, чем в других системах. Например, в Oracle план выполнения параметризованного запроса не учитывает значения параметров, даже если они указаны. Более того, план с переменными привязки кешируется на случай, если тот же запрос будет повторно выполнен с другими значениями параметров. Оптимизатор учитывает статистику таблиц и индексов, но не принимает во внимание конкретные значения параметров. PostgreSQL делает наоборот. План выполнения создается для определенных значений.

Как упоминалось ранее, динамическими запросами пренебрегают и в других СУБД – и незаслуженно, ведь для длительных запросов (десятки секунд и более) накладные расходы обычно незначительны.

Что с внедрением SQL-кода?

Часто за предложением использовать динамический SQL для повышения производительности следуют встревоженные взгляды разработчиков: а как же внедрение SQL-кода? Ведь все слышали истории об украденных паролях и удаленных данных, когда в регистрационной форме вместо даты рождения кто-то умудрился ввести команду. Действительно, у хакеров есть много способов добраться до данных, к которым они не должны получать доступ. Однако в случае динамического SQL есть несколько простых правил, которые помогают минимизировать возможные риски.

Когда функции получают значения параметров из базы данных напрямую (используя идентификаторы), такие значения не могут содержать никаких вредоносных конструкций. Значения, полученные в результате ввода данных пользователем, должны быть защищены функциями PostgreSQL (`quote_literal`, `quote_indent` и т. п. или `format`). Использование этих функций будет продемонстрировано позже в данной главе.

Как использовать динамический SQL в OLTP-системах

Часто может быть полезно создать динамический SQL внутри функции и затем выполнить его, а не передавать значения параметров в качестве переменных привязки. В предыдущих главах мы изложили причины повышения производительности в таких ситуациях, поэтому давайте перейдем к примерам.

Вспомните запрос из листинга 6.6, у которого два критерия выбора: по стране аэропорта вылета и по времени последнего обновления бронирования. В главе 6 мы продемонстрировали, как PostgreSQL изменяет план выполнения в зависимости от конкретных значений этих параметров.

В данной главе мы увидим, что происходит с этим запросом, если он выполняется внутри функции.

Начнем с создания типа возвращаемого значения в листинге 12.2.

Листинг 12.2 ❖ Создаем тип возвращаемого значения

```
DROP TYPE IF EXISTS booking_leg_part;
CREATE TYPE booking_leg_part AS (
    departure_airport char(3),
    booking_id int,
    is_returning boolean
);
```

Теперь создадим функцию с двумя параметрами: кодом страны по стандарту ISO и временем последнего обновления. Эта функция показана в листинге 12.3.

Листинг 12.3 ❖ Запрос из листинга 6.6, упакованный в функцию

```
CREATE OR REPLACE FUNCTION select_booking_leg_country (
    p_country text,
    p_updated timestamptz
)
RETURNS SETOF booking_leg_part
AS $body$
BEGIN
    RETURN QUERY
        SELECT departure_airport,
               booking_id,
               is_returning
          FROM booking_leg bl
               JOIN flight f USING (flight_id)
         WHERE departure_airport IN (
                   SELECT airport_code
                     FROM airport
                    WHERE iso_country = p_country
               )
           AND bl.booking_id IN (
                   SELECT booking_id
                     FROM booking
                    WHERE update_ts > p_updated
               );
END;
%body$
LANGUAGE plpgsql;
```

В главе 6 показано, как PostgreSQL выбирает разные планы выполнения в зависимости от значений параметров для поиска страны и времени и как это влияет на скорость выполнения.

Поскольку функции в PostgreSQL (как и в других системах) являются атомарными, мы не можем воспользоваться командой `EXPLAIN`, чтобы увидеть план выполнения запроса внутри функции (точнее, команду `EXPLAIN` может выполнить, но она покажет только сам факт выполнения функции), но поскольку ожидаемое время ответа на запрос известно, можно понять, что происходит внутри.

Напомним, что ранее команда из листинга 12.4 выполнялась около 40 секунд (с использованием двух соединений хешированием).

Листинг 12.4 ❖ Запрос с двумя соединениями хешированием

```
SELECT departure_airport,
       booking_id,
       is_returning
  FROM booking_leg bl
       JOIN flight f USING (flight_id)
 WHERE departure_airport IN (
           SELECT airport_code
             FROM airport
            WHERE iso_country = 'US'
       )
   AND bl.booking_id IN (
           SELECT booking_id
             FROM booking
            WHERE update_ts > '2020-07-01'
       );
```

Напомним также, что при перемещении границы `update_ts` к 17 августа («текущей дате» набора данных) время выполнения существенно не меняется. Время выполнения запроса с условием `update_ts > '2020-08-01'` составляет около 35 секунд – некоторое ускорение связано с меньшим промежуточным набором данных. План выполнения для этого случая показан на рис. 12.1.

	QUERY PLAN text
1	Hash Join (cost=248176.42..979104.02 rows=395028 width=9)
2	Hash Cond: (bl.flight_id = f.flight_id)
3	-> Hash Join (cost=220335.88..921529.15 rows=1865879 width=9)
4	Hash Cond: (bl.booking_id = booking.booking_id)
5	-> Seq Scan on booking_leg bl (cost=0.00..477162.64 rows=17893564 width=9)
6	-> Hash (cost=210600.34..210600.34 rows=593403 width=8)
7	-> Seq Scan on booking (cost=0.00..210600.34 rows=593403 width=8)
8	Filter: (update_ts > '2020-08-01 00:00:00-05'::timestamp with time zone)
9	-> Hash (cost=25467.60..25467.60 rows=144636 width=8)
10	-> Hash Join (cost=20.09..25467.60 rows=144636 width=8)
11	Hash Cond: (f.departure_airport = airport.airport_code)
12	-> Seq Scan on flight f (cost=0.00..23642.76 rows=683176 width=8)
13	-> Hash (cost=18.33..18.33 rows=141 width=4)
14	-> Seq Scan on airport (cost=0.00..18.33 rows=141 width=4)
15	Filter: (iso_country = 'US'::text)

Рис. 12.1 ❖ План выполнения с двумя соединениями хешированием

В какой-то момент, когда значение `update_ts` будет достаточно близко к 17 августа, PostgreSQL выберет индексный доступ. Для запроса из листинга 12.5 время выполнения составляет 12 секунд.

Листинг 12.5 ❖ Одно соединение хешированием заменяется вложенным циклом

```
SELECT departure_airport,
       booking_id,
       is_returning
  FROM booking_leg bl
       JOIN flight f USING (flight_id)
 WHERE departure_airport IN (
           SELECT airport_code
             FROM airport
            WHERE iso_country = 'US'
       )
   AND bl.booking_id IN (
           SELECT booking_id
             FROM booking
            WHERE update_ts > '2020-08-15'
       );
```

План выполнения для этого случая представлен на рис. 12.2.

	QUERY PLAN text
1	Hash Join (cost=106451.03..632756.29 rows=21758 width=9)
2	Hash Cond: (bl.flight_id = f.flight_id)
3	-> Hash Join (cost=78610.48..602743.76 rows=102774 width=9)
4	Hash Cond: (bl.booking_id = booking.booking_id)
5	-> Seq Scan on booking_leg bl (cost=0.00..477162.64 rows=17893564 width=9)
6	-> Hash (cost=78201.92..78201.92 rows=32685 width=8)
7	-> Bitmap Heap Scan on booking (cost=965.74..78201.92 rows=32685 width=8)
8	Recheck Cond: (update_ts > '2020-08-15 00:00:00-05'::timestamp with time zone)
9	-> Bitmap Index Scan on booking_update_ts (cost=0.00..957.57 rows=32685 width=0)
10	Index Cond: (update_ts > '2020-08-15 00:00:00-05'::timestamp with time zone)
11	-> Hash (cost=25467.60..25467.60 rows=144636 width=8)
12	-> Hash Join (cost=20.09..25467.60 rows=144636 width=8)
13	Hash Cond: (f.departure_airport = airport.airport_code)
14	-> Seq Scan on flight f (cost=0.00..23642.76 rows=683176 width=8)
15	-> Hash (cost=18.33..18.33 rows=141 width=4)
16	-> Seq Scan on airport (cost=0.00..18.33 rows=141 width=4)
17	Filter: (iso_country = 'US'::text)

Рис. 12.2 ❖ План выполнения
с одним соединением хешированием и одним вложенным циклом

Взяв эти цифры в качестве опорных значений, давайте посмотрим, как запрос работает в функции.

Попробуем воспроизвести то же поведение, которое мы наблюдали в главе 6 для длинного запроса с разными условиями поиска, и выполним команды, показанные в листинге 12.6.

Листинг 12.6 ❖ Примеры вызовов функций

```
-- № 1
SELECT * FROM select_booking_leg_country('US', '2020-07-01');
-- № 2
SELECT * FROM select_booking_leg_country('US', '2020-08-01');
-- № 3
SELECT * FROM select_booking_leg_country('US', '2020-08-15');
-- № 4
SELECT * FROM select_booking_leg_country('CZ', '2020-08-01');
```

Наблюдаемое время выполнения будет отличаться в зависимости от того, какие параметры передавались функции при первых вызовах. В результате время выполнения команды № 3, которое должно занять около 10 секунд, может варьироваться от 10 секунд до одной минуты, в зависимости от последовательности вызовов и времени, в течение которого вы делаете паузу между ними. Можно открыть несколько соединений с вашим локальным PostgreSQL и попробовать разный порядок вызовов.

Почему функция ведет себя так непоследовательно? Вспомните главу 11, где говорится, что PostgreSQL *может* сохранять план выполнения подготовленного оператора. Когда функция вызывается в первый раз в сеансе, каждая выполняемая инструкция SQL разбирается, ее план выполнения оптимизируется и *может* быть закеширован для последующих выполнений.

Мы намеренно не описываем конкретное поведение для каждой последовательности вызовов, потому что здесь нет гарантий. И хотя отсутствие гарантий допустимо для обучающей базы данных, оно совершенно неприемлемо в промышленном окружении, особенно когда в системе OLTP ограничено максимальное время ожидания и транзакции, превышающие установленный предел, прерываются.

Чтобы гарантировать, что каждый раз при вызове функции *план выполнения будет оптимизироваться для определенных значений*, мы создаем функции, выполняющие динамический SQL.

В листинге 12.7 показана функция `select_booking_leg_country_dynamic`, которая выполняет точно такой же запрос, что и функция `select_booking_leg_country`. Единственное отличие состоит в том, что первая функция создает команду `SELECT` внутри функции, а затем выполняет ее.

Эта функция принимает тот же набор параметров, что и `select_booking_leg_country`, и возвращает тот же результат. Но обратите внимание, что время ее выполнения для каждого набора параметров неизменно, а это именно то, что нужно в промышленных системах.

Почему поведение изменилось? Поскольку SQL создается непосредственно перед выполнением, оптимизатор не использует кешированный план. Вместо этого он заново строит план для каждого выполнения. Может пока-

заться, что на это требуется дополнительное время, но на самом деле происходит обратное. Время планирования составляет менее 100 мс; это окупается за счет более подходящего плана выполнения, и в итоге экономится значительное время.

Листинг 12.7 ❖ Функция, выполняющая динамический SQL

```
CREATE OR REPLACE FUNCTION select_booking_leg_country_dynamic (
    p_country text,
    p_updated timestamptz
)
RETURNS SETOF booking_leg_part
AS $body$
BEGIN
    RETURN QUERY EXECUTE $$
        SELECT departure_airport,
               booking_id,
               is_returning
          FROM booking_leg bl
               JOIN flight f USING (flight_id)
         WHERE departure_airport IN (
                   SELECT airport_code
                     FROM airport
                    WHERE iso_country = $$|| quote_literal(p_country) ||$$
               )
           AND bl.booking_id IN (
                   SELECT booking_id
                     FROM booking
                    WHERE update_ts > $$|| quote_literal(p_updated) ||$$
               );$$
END;
%body$
LANGUAGE plpgsql;
```

Также обратите внимание на использование функции `quote_literal` для защиты от внедрения SQL-кода.

Это первая, но не единственная причина, почему динамический SQL в функциях оказывается полезен. Позже мы рассмотрим другие случаи в поддержку данного утверждения.

Как использовать динамический SQL в системах OLAP

Не поймите название этого раздела неправильно. Способ, который мы продемонстрируем, можно использовать в любой системе, однако наиболее впечатляющих результатов можно достичь при большом результирующем множестве. Чем больше строк в результате, тем сильнее проявляется выгода.

Представим, что для статистического анализа нам нужно отсортировать пассажиров по возрасту.

Функция определения возрастных категорий представлена в листинге 12.8.

Листинг 12.8 ❖ Функция, назначающая возрастную категорию

```
CREATE OR REPLACE FUNCTION age_category (p_age int)
RETURNS TEXT
LANGUAGE plpgsql
AS $body$
BEGIN
    RETURN (CASE
        WHEN p_age <= 2 THEN 'Infant'
        WHEN p_age <= 12 THEN 'Child'
        WHEN p_age < 65 THEN 'Adult'
        ELSE 'Senior'
    END);
END;
$body$;
```

Если эта функция используется для статистических отчетов, нам может потребоваться вычислить возрастную категорию для всех пассажиров. В главе 11 мы упоминали, что выполнение функций из списка SELECT может замедлить работу, но функции были более сложными. Здесь функция `age_category` выполняет очень простую замену. Тем не менее вызов функции требует времени. Таким образом, для выполнения

```
SELECT passenger_id, age_category(age)
  FROM passenger
 LIMIT 5000000
```

требуется 25 секунд, в то время как

```
SELECT passenger_id,
       CASE
           WHEN age <= 2 THEN 'Infant'
           WHEN age <= 12 THEN 'Child'
           WHEN age < 65 THEN 'Adult'
           ELSE 'Senior'
       END
  FROM passenger
 LIMIT 5000000
```

занимает всего 6 секунд.

В данном конкретном случае использование функции не является обязательным, потому что она нужна нам всего один раз, и даже в одной из наших самых больших таблиц, `passenger`, только 16 млн строк.

В реальных аналитических запросах количество строк, которые нужно обработать, может исчисляться сотнями миллионов, и может понадобиться использовать несколько подобных функций. В одном из реальных случаев время выполнения с функциями составляло четыре часа, а замена всего лишь одной функции на оператор `CASE` сократила время до полутора часов.

Означает ли это, что мы хотим любой ценой избегать использования функций в списке SELECT? Возможно, есть причина, по которой наша аналитиче-

ская команда хочет упаковать назначение возрастной категории в функцию. Скорее всего, они собираются использовать эту функцию в разных запросах и с разными таблицами и не хотят пересматривать и исправлять все свои запросы, если алгоритм назначения категории изменится.

Более производительное решение, сохраняющее простоту сопровождения, состоит в создании другой функции, которая содержит *фрагмент кода в виде текста*, – см. листинг 12.9.

Листинг 12.9 ❖ Функция, создающая часть динамического SQL

```
CREATE OR REPLACE FUNCTION age_category_dyn (p_age text)
RETURNS text
LANGUAGE plpgsql
AS $body$
BEGIN
    RETURN ($$CASE
        WHEN $$ || p_age || $$ <= 2 THEN 'Infant'
        WHEN $$ || p_age || $$ <= 12 THEN 'Child'
        WHEN $$ || p_age || $$ < 65 THEN 'Adult'
        ELSE 'Senior'
    END$$);
END;
$body$;
```

Обратите внимание на разницу. Такой запрос возвращает значение `'Adult'`:

```
SELECT age_category(25)
```

А следующий запрос

```
SELECT age_category_dyn('age')
```

вернет текстовую строку, содержащую фрагмент кода

```
CASE
    WHEN age <= 2 THEN 'Infant'
    WHEN age <= 12 THEN 'Child'
    WHEN age < 65 THEN 'Adult'
    ELSE 'Senior'
END
```

Чтобы использовать эту функцию, вам нужно будет упаковать запрос в функцию, но мы уже знаем, как это сделать, – см. листинг 12.10.

Теперь мы можем выполнить следующую команду:

```
SELECT * FROM passenger_age_category_select (5000000)
```

На выполнение уйдет около 11 секунд, что больше, чем у команды без каких-либо вызовов функций, но все же меньше, чем у исходной версии функции `age_category`. И опять же, в реальных аналитических запросах эффект будет более заметным.

Листинг 12.10 ❖ Использование новой функции age_category_dyn для построения динамического запроса

```
CREATE TYPE passenger_age_cat_record AS (
    passenger_id int,
    age_category text
);

CREATE OR REPLACE FUNCTION passenger_age_category_select (p_limit int)
RETURNS ЫУЕЩА passenger_age_cat_record
ASc$body$
BEGIN
    RETURN QUERY EXECUTE
        $$SELECT passenger_id,
                 $$ || age_category_dyn('age') || $$ AS age_category
            FROM passenger
           LIMIT $$ || p_limit::text;
END;
$body$
LANGUAGE plpgsql;
```

Кто-то может возразить, что создание функций, генерирующих код, не стоит повышения производительности. Повторим: не существует универсального принципа, определяющего, выгодно ли создание функций с точки зрения производительности, декомпозиции кода или переносимости. В главе 11 упоминалось, что декомпозиция для функций PL/pgSQL работает не так, как для объектно-ориентированных языков программирования, и обещали показать примеры. Один из примеров как раз приведен в этом разделе. Здесь функция age_category_dyn помогает декомпозиции кода, поскольку алгоритм присвоения возрастной категории можно менять только в одном месте. В то же время это меньше влияет на производительность, чем более традиционная функция с параметрами. В большинстве случаев создание функции, выполняющей динамический SQL, занимает больше времени, поскольку отладка усложняется. Однако когда функция уже создана, изменения можно вносить быстро. Решить, что более важно – начальное время разработки или среднее время выполнения, – могут только разработчики приложений и баз данных.

Использование динамического SQL для гибкости

Техника, описанная в этом разделе, чаще всего используется в системах OLTP, хотя, опять же, строгого ограничения на тип окружения здесь нет.

Часто системы позволяют пользователю выбирать произвольный список критериев поиска, возможно, с помощью выпадающих списков или других графических способов построения запросов.

Пользователь ничего не знает (и не должен знать) о том, как данные хранятся в базе. Однако поля поиска могут принадлежать разным таблицам,

критерии поиска могут иметь разную селективность, и, в общем, запросы могут сильно отличаться в зависимости от выбранных критериев.

Посмотрим на один пример. Предположим, нам необходима функция для поиска бронирования по любой комбинации следующих значений:

- адрес электронной почты (или начальная часть адреса);
- аэропорт вылета;
- аэропорт прибытия;
- дата отправления;
- идентификатор рейса.

Можно ли обойтись без Elasticsearch, чтобы эффективно реализовать эту функцию?

Когда нужно реализовать такую функциональность, разработчик обычно придумывает что-нибудь, похожее на функцию из листинга 12.11.

Листинг 12.11 ❖ Функция для поиска по различным комбинациям параметров

```
CREATE TYPE booking_record_basic AS (
    booking_id bigint,
    booking_ref text,
    booking_name text ,
    account_id integer,
    email text
);

CREATE OR REPLACE FUNCTION select_booking (
    p_email text,
    p_dep_airport text,
    p_arr_airport text,
    p_dep_date date,
    p_flight_id int
)
RETURNS SETOF booking_record_basic
AS $func$
BEGIN
    RETURN QUERY
        SELECT DISTINCT
               b.booking_id,
               b.booking_ref,
               booking_name,
               account_id,
               email
          FROM booking b
               JOIN booking_leg bl USING (booking_id)
               JOIN flight f USING (flight_id)
         WHERE (p_email IS NULL OR lower(email) LIKE p_email||'%')
           AND (p_dep_airport IS NULL OR departure_airport = p_dep_airport)
           AND (p_arr_airport IS NULL OR arrival_airport = p_arr_airport)
           AND (p_dep_date IS NULL OR scheduled_departure BETWEEN p_dep_date
                                                              AND p_dep_date + 1)
           AND (p_flight_id IS NULL OR bl.flight_id = p_flight_id);
END;
$func$
LANGUAGE plpgsql;
```

Эта функция всегда будет возвращать правильный результат, но с точки зрения производительности ее поведение как минимум трудно предсказать. Обратите внимание, что при поиске по адресу электронной почты соединения с таблицами `booking_leg` и `flight` не нужны, но они все равно будут выполняться.

Давайте сравним время выполнения на нескольких примерах.

Пример 1. Поиск по электронной почте простым запросом занимает 4,5 секунды:

```
SELECT DISTINCT b.booking_id, b.booking_ref, b.booking_name, b.email
  FROM booking b
 WHERE lower(email) LIKE 'lawton52%'
```

Выполнение функции занимает 13 секунд:

```
SELECT * FROM select_booking ('lawton52', NULL, NULL, NULL, NULL)
```

Пример 2. Поиск по электронной почте и идентификатору рейса.

Выполнение запроса занимает 150 мс:

```
SELECT DISTINCT b.booking_id, b.booking_ref, b.booking_name, b.email
  FROM booking b
       JOIN booking_leg bl USING (booking_id)
 WHERE lower(email) LIKE 'lawton52%'
   AND flight_id = 27191
```

При этом выполнение функции занимает 102 мс:

```
SELECT * FROM select_booking ('lawton52', NULL, NULL, NULL, 27191)
```

Пример 3. Поиск по электронной почте, аэропорту вылета и аэропорту прибытия.

Выполнение запроса занимает 200 мс:

```
SELECT DISTINCT b.booking_id, b.booking_ref, b.booking_name, b.email
  FROM booking b
       JOIN booking_leg bl USING (booking_id)
       JOIN flight f USING (flight_id)
 WHERE lower(email) LIKE 'lawton52%'
   AND departure_airport = 'ORD'
   AND arrival_airport = 'JFK'
```

Выполнение функции с теми же параметрами занимает 910 мс:

```
SELECT * FROM select_booking ('lawton52', 'ORD', 'JFK', NULL, NULL)
```

Пример 4. Поиск по электронной почте, аэропорту вылета, аэропорту прибытия и дате вылета.

Запрос выполняется за 95 мс:

```
SELECT DISTINCT b.booking_id, b.booking_ref, b.booking_name, b.email
  FROM booking b
       JOIN booking_leg bl USING (booking_id)
       JOIN flight f USING (flight_id)
 WHERE lower(email) LIKE 'lawton52%'
   AND departure_airport = 'ORD'
   AND arrival_airport = 'JFK'
   AND scheduled_departure BETWEEN '2020-07-30' AND '2020-07-31'
```

Выполнение функции занимает одну секунду:

```
SELECT * FROM select_booking ('lawton52', 'ORD', 'JFK', '2020-07-30', NULL)
```

Пример 5. Поиск по электронной почте и дате вылета.
Выполнение запроса занимает 10 секунд:

```
SELECT DISTINCT b.booking_id, b.booking_ref, b.booking_name, b.email
  FROM booking b
       JOIN booking_leg bl USING (booking_id)
       JOIN flight f USING (flight_id)
 WHERE lower(email) LIKE 'lawton52%'
   AND scheduled_departure BETWEEN '2020-07-30' AND '2020-07-31'
```

Выполнение функции занимает 13 секунд.

```
SELECT * FROM select_booking ('lawton52', NULL, NULL, '2020-07-30', NULL)
```

Пример 6. Поиск по идентификатору рейса.
Выполнение запроса занимает 130 мс:

```
SELECT DISTINCT b.booking_id, b.booking_ref, b.booking_name, b.email
  FROM booking b
       JOIN booking_leg bl USING (booking_id)
 WHERE flight_id = 27191
```

Выполнение функции занимает 133 мс:

```
SELECT * FROM select_booking (NULL, NULL, NULL, NULL, 27191)
```

В действительности, как мы обсуждали ранее, время выполнения для различных вызовов функций может оказаться даже больше, если первые вызовы функции в текущем сеансе приводят к плану, неоптимальному для последующих выполнений. Экспериментируя с этой функцией, нам удалось найти последовательность вызовов, из-за которой последний пример выполнялся в течение трех минут.

Как решить эту проблему? Так же, как в предыдущем примере, можно написать функцию, которая динамически создает запрос в зависимости от передаваемых параметров. Запрос будет планироваться заново перед каждым выполнением.

Исходный код новой функции представлен в листинге 12.12.

Листинг 12.12 ❖ Функция, создающая динамический SQL для поиска по разным критериям

```
CREATE OR REPLACE FUNCTION select_booking_dyn (
    p_email text,
    p_dep_airport text,
    p_arr_airport text,
    p_dep_date date,
    p_flight_id int
)
RETURNS SETOF booking_record_basic
AS $func$
DECLARE
    v_sql text := 'SELECT DISTINCT
                          b.booking_id,
                          b.booking_ref,
                          booking_name,
                          account_id,
                          email
                     FROM booking b ';
    v_where_booking text;
    v_where_booking_leg text;
    v_where_flight text;
BEGIN
    IF p_email IS NOT NULL THEN
        v_where_booking := $$ lower(email) LIKE $$ || quote_literal(p_email||'%');
    END IF;
    IF p_flight_id IS NOT NULL THEN
        v_where_booking_leg := $$ flight_id = $$ || p_flight_id::text;
    END IF;
    IF p_dep_airport IS NOT NULL THEN
        v_where_flight := concat_ws(
            $$ AND $$,
            v_where_flight,
            $$departure_airport = $$ || quote_literal(p_dep_airport));
    END IF;
    IF p_arr_airport IS NOT NULL THEN
        v_where_flight := concat_ws(
            $$ AND $$,
            v_where_flight,
            $$arrival_airport = $$ || quote_literal(p_arr_airport));
    END IF;
    IF p_dep_date IS NOT NULL THEN
        v_where_flight := concat_ws(
            $$ AND $$,
            v_where_flight,
            $$scheduled_departure BETWEEN $$ || quote_literal(p_dep_date) ||
                $$::date AND $$ || quote_literal(p_dep_date) || $$::date + 1$$);
    END IF;
    IF v_where_flight IS NOT NULL OR v_where_booking_leg IS NOT NULL THEN
        v_sql := v_sql || $$ JOIN booking_leg bl USING (booking_id) $$;
    END IF;
```

```
    IF v_where_flight IS NOT NULL THEN
        v_sql := v_sql || $$ JOIN flight f USING (flight_id) $$;
    END IF;
    v_sql := v_sql || $$ WHERE $$ || concat_ws(
        $$ AND $$,
        v_where_booking, v_where_booking_leg, v_where_flight);
    --RAISE NOTICE 'sql:%', v_sql;
    RETURN QUERY EXECUTE (v_sql);
END;
$func$
LANGUAGE plpgsql;
```

Здесь очень много кода! Давайте пройдемся по нему и разберемся, что именно тут происходит.

Параметры новой функции точно такие же, как и у старой, и тип результата тоже совпадает, но тело функции полностью другое. На высоком уровне эта функция создает в текстовой переменной `v_sql` инструкцию, которая затем выполняется.

Динамическое построение запроса позволяет нам включать в него только те соединения, которые необходимы. Таблица `booking` нужна всегда, поэтому начальное значение переменной `v_sql` равно

```
'SELECT DISTINCT
        b.booking_id,
        b.booking_ref,
        booking_name,
        account_id,
        email
   FROM booking b '
```

Затем, в зависимости от того, какие параметры передаются непустыми (`NOT NULL`), функция определяет, какие еще таблицы необходимы. Это может быть только таблица `booking_leg`, если не заданы параметры, связанные с рейсом, или это могут быть обе таблицы `booking_leg` и `flight`.

После добавления всех необходимых таблиц строятся критерии поиска путем объединения всех условий с разделителем `AND`. На этом построение оператора `v_sql` завершается, и он выполняется. Чтобы увидеть окончательный запрос для различных вызовов функции, раскомментируйте инструкцию `RAISE NOTICE`.

Не слишком ли много работы для повышения производительности? Попробуйте создать эту функцию и выполнить ее с параметрами из предыдущих примеров. Быстро станет ясно, что время выполнения функции `select_booking_dyn` не превышает время выполнения соответствующих инструкций SQL для каждого набора параметров. Причем время выполнения предсказуемо и не зависит от истории вызовов в текущем сеансе.

Да, динамические функции нелегко отлаживать, и может потребоваться включить отладочную печать, но если производительность промышленной системы действительно важна, то результаты того стоят.

Использование динамического SQL в помощь оптимизатору

Поскольку вся эта глава целиком посвящена способам повышения производительности запросов с помощью динамического SQL, заголовок может вызвать некоторое недоумение. Но в этом разделе мы разбираемся с проблемами производительности другого рода. Здесь динамический SQL будет использоваться не с целью построения разных запросов для разных условий, а для того, чтобы подтолкнуть оптимизатор к выбору более подходящего плана выполнения.

Если внимательно присмотреться ко всем примерам из предыдущего раздела, то можно заметить, что одна комбинация критериев поиска работает очень плохо, даже несмотря на небольшое результирующее множество: это случай, когда поиск ведется по адресу электронной почты и аэропорту вылета. Даже в тех случаях, когда электронная почта достаточно ограничивает выборку, оптимизатор не может использовать индекс для `booking_id` во втором соединении. Если мы выполним запрос из листинга 12.13, то план выполнения покажет соединения хешированием – см. рис. 12.3.

	QUERY PLAN text
1	Unique (cost=611071.11..611086.92 rows=1265 width=41)
2	-> Sort (cost=611071.11..611074.27 rows=1265 width=41)
3	Sort Key: b.booking_id, b.booking_ref, b.booking_name, b.email
4	-> Hash Join (cost=86637.80..611005.93 rows=1265 width=41)
5	Hash Cond: (bl.flight_id = f.flight_id)
6	-> Hash Join (cost=71195.67..595328.95 rows=89468 width=45)
7	Hash Cond: (bl.booking_id = b.booking_id)
8	-> Seq Scan on booking_leg bl (cost=0.00..477162.64 rows=17893564 width=8)
9	-> Hash (cost=70842.97..70842.97 rows=28216 width=41)
10	-> Bitmap Heap Scan on booking b (cost=873.77..70842.97 rows=28216 width=41)
11	Filter: (lower(email) ~~ 'lawton510%'::text)
12	-> Bitmap Index Scan on booking_email_lower_pattern (cost=0.00..866.72 rows=28216 width=0)
13	Index Cond: ((lower(email) ~>=~ 'lawton510'::text) AND (lower(email) ~<~ 'lawton511'::text))
14	-> Hash (cost=15321.42..15321.42 rows=9656 width=4)
15	-> Bitmap Heap Scan on flight f (cost=227.26..15321.42 rows=9656 width=4)
16	Recheck Cond: (departure_airport = 'JFK'::bpchar)
17	-> Bitmap Index Scan on flight_depart_arr_sched_dep (cost=0.00..224.85 rows=9656 width=0)
18	Index Cond: (departure_airport = 'JFK'::bpchar)

Рис. 12.3 ❖ План выполнения для запроса из листинга 12.13 с соединениями хешированием

Время выполнения этого запроса составляет около семи секунд, а результат содержит всего 224 строки, так что это небольшой запрос, и время выполнения должно быть меньше.

Листинг 12.13 ❖ Выбор бронирования по электронной почте и аэропорту вылета

```
SELECT DISTINCT
       b.booking_id,
       b.booking_ref,
       b.booking_name,
       b.email
  FROM booking b
       JOIN booking_leg bl USING (booking_id)
       JOIN flight f USING (flight_id)
 WHERE lower(email) LIKE 'lawton510%'
   AND departure_airport = 'JFK'
```

Причина такого неоптимального плана уже упоминалась ранее – оптимизатор PostgreSQL неправильно оценивает размер промежуточных наборов результатов. Фактическое количество строк, отфильтрованных по шаблону почты, составляет 3941, а оценка в плане – 28 216.

Методика оптимизации этого запроса в буквальном смысле помогает оптимизатору выполнять свою работу и устраняет необходимость оценивать размер результирующего множества. Как этого достичь? Сначала найдем идентификаторы бронирований, соответствующие указанному адресу электронной почты, а затем передадим этот список идентификаторов основному запросу. Обратите внимание: функция, которую мы используем, чтобы проиллюстрировать этот случай, очень специфична и используется только как пример (листинг 12.14). Функция более общего вида, которая могла бы использоваться в промышленной системе, получилась бы значительно больше.

Почему это работает? Мы знаем, что поиск по электронной почте будет довольно ограничительным, потому что передается почти весь адрес или, по крайней мере, его существенная часть. Итак, на первом этапе выбирается относительно небольшое количество бронирований с этим адресом электронной почты, и их идентификаторы сохраняются в текстовой переменной `v_booking_ids`. Затем создается запрос с явным списком идентификаторов.

Выполнение этой новой функции занимает от 0,5 до 0,6 секунды:

```
SELECT * FROM select_booking_email_departure('lawton510','JFK')
```

Изучив вывод команды `EXPLAIN` для сгенерированного SQL, вы увидите план выполнения в том виде, в каком он представлен на рис. 12.4.

Даже с несколькими тысячами идентификаторов доступ на основе индекса оказывается более эффективным.

Листинг 12.14 ❖ Динамический SQL для улучшения кода из листинга 12.13

```
CREATE OR REPLACE FUNCTION select_booking_email_departure(
    p_email text,
    p_dep_airport text
)
RETURNS SETOF booking_record_basic
AS $body$
DECLARE
    v_sql text;
    v_booking_ids text;
BEGIN
    EXECUTE $$SELECT array_to_string(array_agg(booking_id), ',')
                FROM booking
               WHERE lower(email) LIKE $$ || quote_literal(p_email || '%')
    INTO v_booking_ids;
    v_sql := $$SELECT DISTINCT
                      b.booking_id,
                      b.booking_ref,
                      b.booking_name,
                      b.email
                 FROM booking b
                      JOIN booking_leg bl USING(booking_id)
                      JOIN flight f USING (flight_id)
                WHERE b.booking_id IN ($$ || v_booking_ids || $$)
                  AND departure_airport = $$ || quote_literal(p_dep_airport);
    RETURN QUERY EXECUTE v_sql;
END;
$body$
LANGUAGE plpgsql;
```

	QUERY PLAN text
1	Unique (cost=449047.92..449050.58 rows=177 width=45)
2	-> Sort (cost=449047.92..449048.37 rows=177 width=45)
3	Sort Key: b.booking_id, b.booking_ref, b.booking_name, b.account_id, b.email
4	-> Nested Loop (cost=1.42..449041.31 rows=177 width=45)
5	-> Nested Loop (cost=0.99..443421.61 rows=12496 width=49)
6	-> Index Scan using booking_pkey on booking b (cost=0.43..28636.50 rows=3941 width=45)
7	Index Cond: (booking_id = ANY ('{639209,641275,644191,645821,645904,646657,650455,650529,652968,655582,65...
8	-> Index Scan using booking_leg_booking_id on booking_leg bl (cost=0.56..104.95 rows=30 width=8)
9	Index Cond: (booking_id = b.booking_id)
10	-> Index Scan using flight_pkey on flight f (cost=0.42..0.45 rows=1 width=4)
11	Index Cond: (flight_id = bl.flight_id)
12	Filter: (departure_airport = 'JFK'::bpchar)

Рис. 12.4 ❖ План выполнения для динамического SQL со списком идентификаторов бронирований

Обертки сторонних данных и динамический SQL

Как упоминалось во введении, подробное обсуждение распределенных запросов выходит за рамки этой книги. Однако глава про динамический SQL дает нам хорошую возможность сделать несколько замечаний касательно *оберток сторонних данных*.

> Обертка сторонних данных – это библиотека, которая может взаимодействовать с внешним источником данных (с данными, которые находятся за пределами вашего сервера PostgreSQL), скрывая детали подключения к источнику данных и получения данных из него.

Обертка сторонних данных – очень мощный инструмент, и таких оберток появляется все больше и больше для различных типов баз данных. PostgreSQL отлично справляется с оптимизацией запросов, которые включают *сторонние таблицы*, то есть отображения таблиц из внешних систем. Однако, поскольку доступ к внешней статистике может быть ограничен, особенно когда внешние системы основаны не на PostgreSQL, оптимизация может быть не такой точной. Мы нашли очень полезным использовать методы, описанные в предыдущем разделе.

Один способ оптимизации – выполнить локальную часть запроса, определяющую, какие записи необходимы с удаленного сервера, а затем обратиться к удаленной таблице. Другой вариант – отправить запрос с условиями, заданными константами (например, `WHERE update_ts > CURRENT_DATE - 3`), на удаленный сервер, получить внешние данные и затем локально выполнить оставшуюся часть запроса. Использование одного из этих двух методов помогает минимизировать непостоянство времени выполнения.

Выводы

Динамический SQL – исключительно мощный инструмент PostgreSQL, который недостаточно используется разработчиками баз данных. Использование динамического SQL может улучшить производительность в ситуациях, когда все другие методы оптимизации не работают.

Динамический SQL лучше всего работает внутри функций; инструкция SQL создается на основе входных параметров функции и затем выполняется. Этот подход можно использовать как в окружении OLTP, так и в OLAP.

Если вы решите применить динамический SQL для своего проекта, будьте готовы к серьезной и трудоемкой отладке. Поначалу это может удручать, но улучшение производительности того стоит.

Глава 13

Как избежать подводных камней объектно-реляционного отображения

В главе 10 обсуждалось типичное взаимодействие между приложением и базой данных и объяснялось влияние объектно-реляционной потери соответствия на производительность. Там же утверждалось, что любое потенциальное решение должно позволять работать с большими объектами (то есть с наборами данных) и должно поддерживать обмен сложными структурами. В этой главе представлен разработанный нами подход, который успешно применяется в промышленном окружении. Он носит название NORM (No-ORM).

Мы ни в коем случае не претендуем на первенство в нашем стремлении преодолеть объектно-реляционную потерю соответствия, и не мы первые предлагаем альтернативу ORM. NORM – лишь одно из множества возможных решений. Особенность, которая выделяет NORM среди других инструментов, – простота использования.

Репозиторий NORM на GitHub (https://github.com/hettie-d/NORM) содержит документацию по этому подходу и пример кода, созданный в соответствии с методологией NORM.

Почему разработчикам приложений нравится NORM

Новые методологии разработки могут требовать от разработчиков приложений значительного изменения процесса разработки, а это неизбежно приводит к снижению производительности труда. Часто потенциальный прирост

производительности системы не оправдывает увеличения времени разработки. В конце концов, время разработчика – самый дорогой ресурс в любом проекте.

В главе 11 преимуществам использования функций предшествовало предупреждение: «Если вы сможете убедить разработчиков приложений». И часто их не удается убедить из-за трудностей адаптации к новому стилю программирования. Это не относится к NORM.

В следующих разделах мы объясним привлекательность этого подхода как для разработчиков приложений, так и для разработчиков баз данных.

Сравнение ORM и NORM

В главе 10 обсуждалось узкое место в обмене данными, создаваемое ORM. Рисунок 13.1 повторяет рис. 10.2 из главы 10. На нем изображен поток данных между приложением и базой данных.

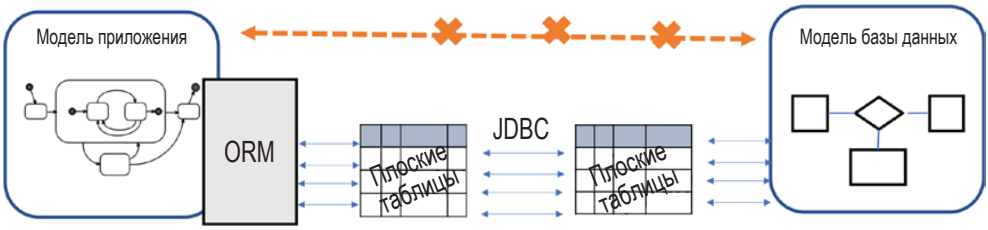


Рис. 13.1 ❖ Как работает ORM

Основная проблема заключается в том, что сложные объекты из модели приложения разбираются на атомарные части перед взаимодействием с базой данных, что приводит к слишком большому количеству мелких запросов и тем самым снижает производительность системы.

Подход, предлагаемый NORM, представлен на рис. 13.2.

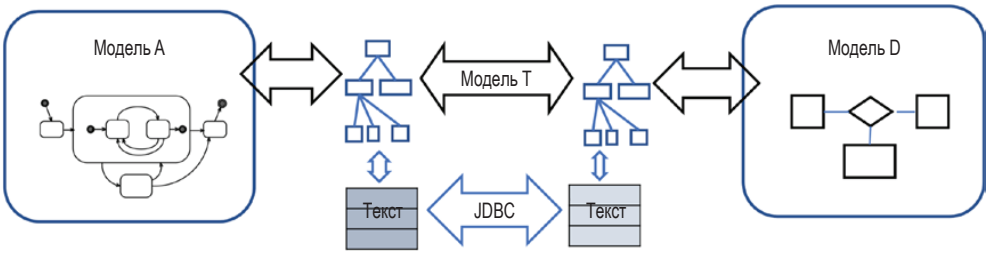


Рис. 13.2 ❖ Как работает NORM

На этом рисунке модель A – модель приложения, модель D – модель базы данных, а модель T – *транспортная модель*. Наличие транспортной моде-

ли – уникальная особенность подхода NORM, которая делает отображение *симметричным*. Мы не пытаемся построить модель базы данных на основе модели приложения, как не настаиваем и на том, чтобы объекты базы данных создавались первыми. Вместо этого мы призываем заключить *контракт* между уровнем приложения и базой данных, в том же смысле, в котором REST является контрактом для веб-служб. Контракт, или транспортная модель, представляет собой объект JSON. Контракт позволяет упростить долговременное хранение объектов путем их сериализации в объекты JSON, с которыми умеет работать база данных.

Таким образом, для сохранения объекта любой структуры и любой сложности требуется только одно обращение к базе данных.

Аналогично, при извлечении из базы данных приложение может десериализовать полученный объект в свою модель, и для этого требуется только один вызов. Приложение может также передавать параметры, определенные как часть контракта, чтобы сообщить базе данных, что ему нужны дополнительные части модели, как в запросах веб-служб OData.

Разработчикам приложений нравится упрощенная реализация уровня доступа к данным на стороне приложения. NORM использует контракт для определения входных и выходных данных каждого вызова базы данных. Это позволяет разработчикам приложений писать код, соответствующий контракту, и легко имитировать любые зависимости при тестировании, поскольку вызовы базы данных и полученный результат должны соответствовать контракту. Таким образом, после заключения контракта разработчики баз данных и приложений могут выполнять свои части работы одновременно и независимо друг от друга. Более того, разные группы разработчиков приложений могут использовать один и тот же контракт для разных проектов.

Что касается приложений, то все современные объектно-ориентированные языки имеют библиотеки для сериализации и десериализации объектов. При каждом новом взаимодействии с базой данных для реализации можно повторно использовать один и тот же шаблон.

Это позволяет разработчикам приложений тратить больше времени на проектирование структуры JSON, чтобы обеспечить ее соответствие текущим и будущим потребностям бизнеса. Повторное использование одного и того же шаблона взаимодействий также сокращает время реализации, уменьшает вероятность ошибок и позволяет минимальными изменениями кода повлиять на всю реализацию доступа к базе данных.

Как работает NORM

Чтобы продемонстрировать, как работает NORM, вернемся к примеру из главы 10.

На рис. 13.3 изображено подмножество диаграммы «сущность–связь» схемы `postgres_air`, использованной для построения примера.

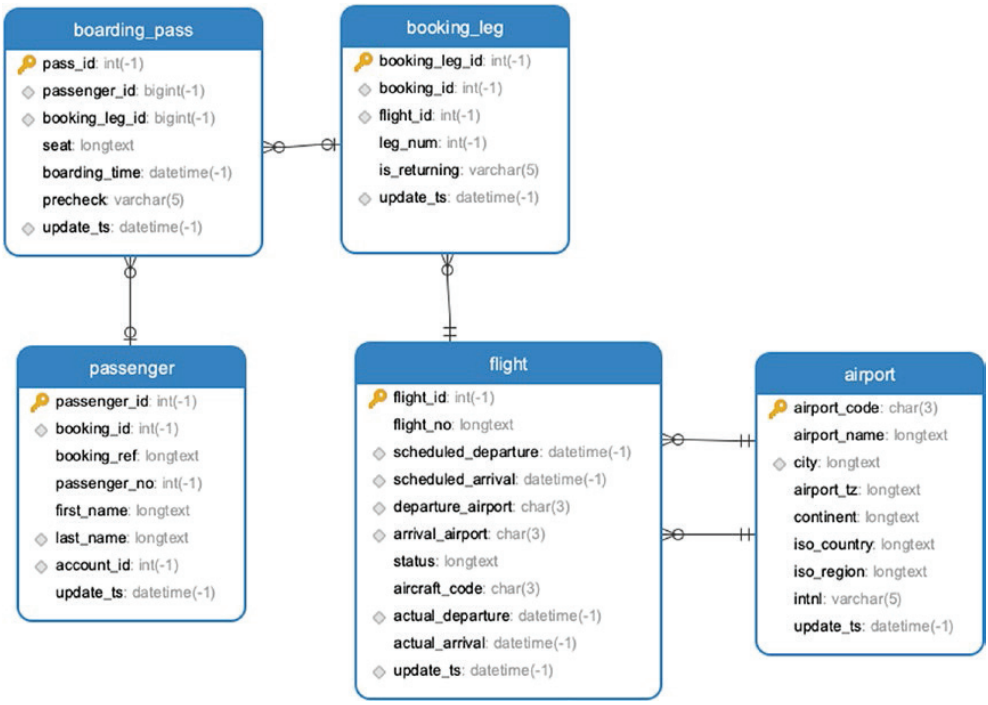


Рис. 13.3 ❖ Диаграмма «сущность–связь» для рассматриваемого примера

В главе 10, обсуждая взаимодействие между приложением и базой данных, мы набросали объект (который теперь можем назвать транспортным объектом) со всей информацией, связанной с бронированием. С точки зрения авиапассажиров, бронирование представляет собой маршрут их путешествия. В попытке сохранить читаемость фрагментов кода мы устранили один уровень вложенности и решили представить только сегмент бронирования, то есть один из рейсов маршрута. Таким образом, для целей данного примера наш транспортный объект является объектом сегмента бронирования. Диаграмма на рис. 13.3 показывает все таблицы и связи, необходимые для построения отображения объекта базы данных в транспортный объект. Соответствующий транспортный объект изображен на рис. 13.4.

Обратите внимание, что этот объект представляет контракт, то есть структуру объекта, которую приложение ожидает получить. Он значительно отличается от того, как данные хранятся в базе. Важно, что реализация базы данных никоим образом не влияет на то, как приложение взаимодействует с базой данных, пока ее ответ соответствует контракту.

Пример объекта JSON, следующего этому контракту, показан на рис. 13.5.

Взаимодействие между приложением и базой данных можно свести к следующему:

1) приложение сериализует данные в формат JSON, затем преобразует его в массив текстовых строк и вызывает соответствующую функцию базы данных;

2) функция базы данных разбирает JSON, который был передан ей в качестве параметра, и выполняет все, что должна делать функция: выбирает или преобразует данные;
3) результирующее множество преобразуется в JSON (или, точнее, в массив строк, представляющий массив объектов JSON) и передается в приложение, где он десериализуется и используется приложением.

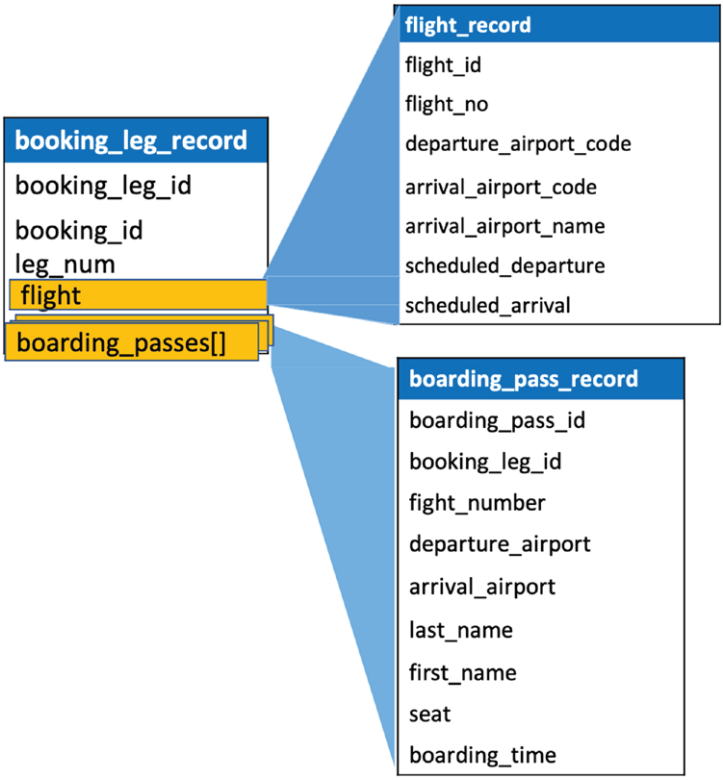


Рис. 13.4 ❖ Транспортный объект (контракт)

На стороне приложения классы Java, представленные в листингах 13.1, 13.2 и 13.3, отображаются в тот же транспортный объект.

Листинг 13.1 ❖ Класс FlightEntity

```
package com.xxx.adapter.repository.entity.tls;

import com.fasterxml.jackson.annotation.JsonProperty;

import java.time.ZonedDateTime;

public class FlightEntity {
    @JsonProperty("flight_id")
    private int flightId;
```

```
    @JsonProperty("flight_no")
    private String flightNumber;

    @JsonProperty("departure_airport_code")
    private String departureAirportCode;

    @JsonProperty("departure_airport_name")
    private String departureAirportName;

    @JsonProperty("arrival_airport_code")
    private String arrivalAirportCode;

    @JsonProperty("arrival_airport_name")
    private String arrivalAirportName;

    @JsonProperty("scheduled_departure")
    private ZonedDateTime scheduledDeparture;

    @JsonProperty("scheduled_arrival")
    private ZonedDateTime scheduledArrival;
}
```

```
{"booking_leg_id":17564910,
 "leg_num":2,
 "booking_id":232346,
 "flight":{"flight_id":13650,
           "flight_no":"1245",
           "departure_airport_code":"JFK",
           "departure_airport_name":"John F Kennedy International Airport",
           "arrival_airport_code":"CDG",
           "arrival_airport_name":"Charles de Gaulle International Airport",
           "scheduled_departure":"2020-06-05T00:20:00-05:00",
           "scheduled_arrival":"2020-06-05T02:45:0005:00"},
 "boarding_passes":
      [
        {"boarding_pass_id":1247796,
        "booking_leg_id":17564910,
        "flight_no":"1245",
        "departure_airport":"JFK",
        "arrival_airport":"CDG",
        "last_name":"LEWIS",
        "first_name":"ELIA",
        "seat":"1E",
        "boarding_time":"2020-06-04T23:50:00-05:00"
        },
        {"boarding_pass_id":1247797,
        "booking_leg_id":17564910,
        "flight_no":"1245",
        "departure_airport":"JFK",
        "arrival_airport":"CDG",
        "last_name":"LEVY",
        "first_name":"ALEXANDER",
        "seat":"1F",
        "boarding_time":"2020-06-04T23:50:00-05:00"}
      ]
   }
```

Рис. 13.5 ❖ Транспортный объект в виде JSON

Листинг 13.2 ❖ Класс BoardingPass

```
package com.xxx.adapter.repository.entity.tls;

import com.fasterxml.jackson.annotation.JsonProperty;

import java.time.ZonedDateTime;

public class BoardingPassEntity {
    @JsonProperty("boarding_pass_id")
    private int boardingPassId;

    @JsonProperty("booking_leg_id")
    private int bookingLegId;

    @JsonProperty("last_name")
    private String lastName;

    @JsonProperty("first_name")
    private String firstName;

    @JsonProperty("seat")
    private String seatNumber;

    @JsonProperty("boarding_time")
    private ZonedDateTime boardingTime;
}
```

Листинг 13.3 ❖ Класс BookingLegEntity

```
package com.braviant.adapter.repository.entity.tls;

import com.fasterxml.jackson.annotation.JsonProperty;

import java.util.List;

public class BookingLegEntity {
    @JsonProperty("booking_leg_id")
    private int bookingLegId;

    @JsonProperty("leg_num")
    private int legNumber;

    @JsonProperty("booking_id")
    private String booking_id;

    @JsonProperty("flight")
    private FlightEntity flight;

    @JsonProperty("boardingPass")
    private List<BoardingPassEntity> boardingPasses;
}
```

Стоит отметить, что мы можем создавать совершенно разные транспортные объекты для одного и того же набора таблиц. Например, перед отправ-

лением любого рейса должен быть создан так называемый манифест. В этом документе перечислены все пассажиры с указанием их мест. Транспортный объект для манифеста изображен на рис. 13.6.

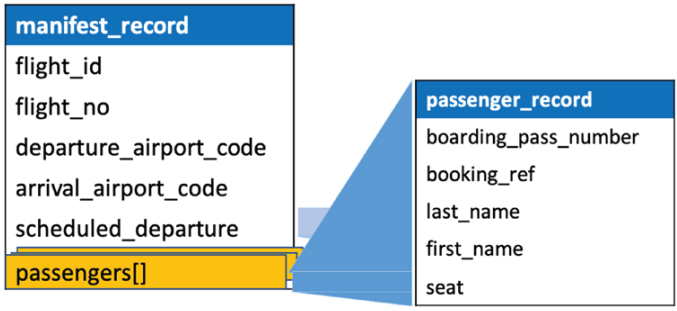


Рис. 13.6 ❖ Транспортный объект для манифеста рейса

Соответствующий JSON представлен на рис. 13.7.

```
{
    "flight": {
        "flight_id": 13650,
        "flight_no": "1245",
        "scheduled_arrival": "2020-06-05T02:45:00-05:00",
        "scheduled_departure": "2020-06-05T00:20:00-05:00",
        "arrival_airport_code": "CDG",
        "arrival_airport_name": "Charles de Gaulle International Airport",
        "departure_airport_code": "JFK",
        "departure_airport_name": "John F Kennedy International Airport"
    },
    "boarding_passes": [
        {
            "seat": "1E",
            "flight_no": "1245",
            "last_name": "LEWIS",
            "first_name": "ELIA",
            "boarding_time": "2020-06-04T23:50:00-05:00",
            "booking_leg_id": 17564910,
            "arrival_airport": "CDG",
            "boarding_pass_id": 1247796,
            "departure_airport": "JFK"
        },
        {
            "seat": "1F",
            "flight_no": "1245",
            "last_name": "LEVY",
            "first_name": "ALEXANDER",
            "boarding_time": "2020-06-04T23:50:00-05:00",
            "booking_leg_id": 17564910,
            "arrival_airport": "CDG",
            "boarding_pass_id": 1247797,
            "departure_airport": "JFK"
        },
        ...
    ]
}
```

Рис. 13.7 ❖ Манифест в виде JSON

Детали реализации

А теперь добавим конкретики и покажем, как достичь нашей цели.

В листинге 13.4 объединены определения типов из листингов 11.17 и 11.20.

Мы определяем типы `boarding_pass_record` и `flight_record`, а затем `booking_leg_record`, который использует эти типы в качестве компонентов.

Листинг 13.4 ❖ Определения типа booking_leg

```
CREATE TYPE boarding_pass_record AS (
    boarding_pass_id int,
    booking_leg_id int,
    flight_no text,
    departure_airport text,
    arrival_airport text,
    last_name text,
    first_name text,
    seat text,
    boarding_time timestamptz
);

CREATE TYPE flight_record AS (
    flight_id int,
    flight_no text,
    departure_airport_code text,
    departure_airport_name text,
    arrival_airport_code text,
    arrival_airport_name text,
    scheduled_departure timestamptz,
    scheduled_arrival timestamptz
);

CREATE TYPE booking_leg_record AS (
    booking_leg_id int,
    leg_num int,
    booking_id int,
    flight flight_record,
    boarding_passes boarding_pass_record[]
);
```

Эти определения типов и представляют собой транспортный объект `booking_leg` с рис. 13.4. Следующим шагом является создание этого объекта с использованием идентификатора сегмента бронирования. Такая функция уже была сделана в главе 11 в листинге 11.21. Однако, чтобы приложение могло ее использовать, необходимо внести несколько изменений. В частности, функция должна возвращать не множество записей, а объект JSON. Эта задача выполняется в два этапа.

Сначала используем слегка измененную функцию `booking_leg_select_json`, показанную в листинге 13.5.

Листинг 13.5 ❖ Функция, возвращающая транспортный объект booking_leg

```
CREATE OR REPLACE FUNCTION booking_leg_select_json (p_booking_leg_id int)
RETURNS booking_leg_record[]
AS $body$
DECLARE
    v_result booking_leg_record[];
    v_sql text;
BEGIN
    SELECT array_agg(single_item) FROM (
        SELECT row(
                    bl.booking_leg_id,
                    leg_num,
                    bl.booking_id,
                    ( SELECT row(
                                flight_id,
                                flight_no,
                                departure_airport,
                                da.airport_name,
                                arrival_airport,
                                aa.airport_name,
                                scheduled_departure,
                                scheduled_arrival
                           )::flight_record
                       FROM flight f
                            JOIN airport da
                                 ON da.airport_code = departure_airport
                            JOIN airport aa
                                 ON aa.airport_code = arrival_airport
                      WHERE flight_id = bl.flight_id
                    ),
                    ( SELECT array_agg (row(
                                pass_id,
                                bp.booking_leg_id,
                                flight_no,
                                departure_airport,
                                arrival_airport,
                                last_name,
                                first_name,
                                seat,
                                boarding_time
                           )::boarding_pass_record)
                       FROM flight f1
                            JOIN boarding_pass bp
                                 ON f1.flight_id = bl.flight_id
                                 AND bp.booking_leg_id = bl.booking_leg_id
                            JOIN passenger p
                                 ON p.passenger_id = bp.passenger_id
                    )
                )::booking_leg_record as single_item
          FROM booking_leg bl
         WHERE bl.booking_leg_id = p_booking_leg_id
    ) s
```

```
    INTO v_result;
    RETURN (v_result);
END;
$body$
LANGUAGE plpgsql;
```

Разница между двумя функциями минимальна: первая возвращает множество записей, агрегируя только набор посадочных талонов. Вторая агрегирует все результирующее множество в массив записей.

Но этого изменения недостаточно, чтобы решить проблему, описанную в главе 11: наличие множества специальных символов затрудняет использование возвращаемого объекта для приложения. Фактически выполнение

```
SELECT * FROM booking_leg_select_json(17564910)
```

приводит к строке трудно интерпретируемых символов (рис. 13.8).

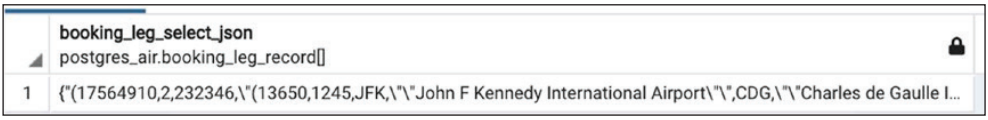


Рис. 13.8 ❖ Результат выполнения

Чтобы обойти эту проблему, мы написали центральную для предлагаемого решения функцию. Она представлена в листинге 13.6, а также является частью репозитория NORM на GitHub.

Листинг 13.6 ❖ Функция ARRAY_TRANSPORT

```
CREATE OR REPLACE FUNCTION array_transport (all_items anyarray)
RETURNS SETOF text
RETURNS NULL ON NULL INPUT
LANGUAGE plpgsql
AS $body$
DECLARE
    item record;
BEGIN
    FOREACH item IN ARRAY all_items
    LOOP
        RETURN NEXT(to_json(item)::text);
    END LOOP;
END;
$body$;
```

Эта функция принимает в качестве параметра любой массив; таким образом ее можно использовать для обработки результатов любой функции, которая возвращает массив любых пользовательских типов, независимо от сложности и уровня вложенности этих типов.

Она создает JSON для любой записи всего за один проход, используя стандартную функцию PostgreSQL `to_json`.

После построения массива JSON каждый элемент преобразуется в текстовую строку, чтобы его можно было передавать через JDBC. Вернувшись к рис. 13.2, вы увидите, что мы реализовали желаемый процесс обмена данными, по крайней мере в одном направлении. Теперь при выполнении

```
SELECT * FROM array_transport(booking_leg_select_json(17564910))
```

результат намного удобнее читать (см. рис. 13.9).

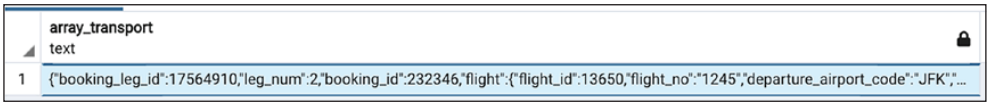


Рис. 13.9 ❖ Текстовое представление JSON

Проницательные читатели могут заметить, что этот результат уже был показан на рис. 13.5 как транспортный объект, и задаться вопросом, зачем разбивать этот процесс на два шага и почему бы сразу не вернуть набор текстовых строк.

Причина состоит в том, что нам нужно сохранить *строгие зависимости типов*. Процесс разработки управляется контрактом; следовательно, существует обязательство по возврату объектов указанного типа. У JSON нет типа; любая текстовая строка, содержащая грамматически правильные объекты JSON, является допустимой. Если тип возвращаемого значения меняется из-за смены требований, нам нужно удалить его, а этого не произойдет, если не будут каскадно удалены все зависимые объекты, включая все функции, возвращающие значения данного типа. По сути, это гарантирует, что база данных соблюдает контракт.

Наконец, стоит отметить, что использование вложенных запросов в списке SELECT, как показано в листинге 13.3, хорошо работает, когда результирующие множества невелики и содержат всего несколько записей. Если ожидается большее количество возвращаемых объектов или сами объекты более сложные, могут потребоваться иные методы. В репозитории NORM на GitHub есть несколько подходящих примеров; см. https://github.com/hettie-d/NORM/blob/master/sql/account_pkg.sql.

Сложный поиск

Функция из листинга 13.2 допускает только один критерий фильтрации – по идентификатору сегмента бронирования.

NORM, однако, позволяет использовать всю поисковую мощь движка реляционной базы данных и выполнять запросы любой сложности, при этом предоставляя результаты в формате, который удобен для приложения.

Следующий шаг уже должен быть очевиден. Мы объединяем динамический запрос, который мы написали в главе 12 (листинг 12.12), с тем, как мы

форматируем вывод функции (листинге 13.5). Результат представлен в листинге 13.7. Здесь мы показываем упрощенную версию функции.

Листинг 13.7 ❖ Поиск по сложным критериям

```
CREATE OR REPLACE FUNCTION search_booking_leg(p_json json)
RETURNS booking_leg_record[]
AS $func$
DECLARE
    v_search_condition text := null;
    v_rec record;
    v_result booking_leg_record[];
    v_sql text :=
        $$SELECT array_agg(single_item)
            FROM ( SELECT row(
                            bl.booking_leg_id,
                            leg_num,
                            bl.booking_id,
                            ( SELECT row(
                                        flight_id,
                                        flight_no,
                                        departure_airport,
                                        da.airport_name,
                                        arrival_airport,
                                        aa.airport_name,
                                        scheduled_departure,
                                        scheduled_arrival
                                       )::flight_record
                                FROM flight f
                                     JOIN airport da
                                          ON da.airport_code = departure_airport
                                     JOIN airport aa
                                          ON aa.airport_code = arrival_airport
                               WHERE flight_id = bl.flight_id
                            ),
                            ( SELECT array_agg (row(
                                        pass_id,
                                        bp.booking_leg_id,
                                        flight_no,
                                        departure_airport,
                                        arrival_airport,
                                        last_name,
                                        first_name,
                                        seat,
                                        boarding_time
                                       )::boarding_pass_record)
                                FROM flight f1
                                     JOIN boarding_pass bp
                                          ON f1.flight_id = bl.flight_id
                                          AND bp.booking_leg_id = bl.booking_leg_id
                                     JOIN passenger p
                                          ON p.passenger_id = bp.passenger_id)
```

```
                        )::booking_leg_record AS single_item
                  FROM booking_leg bl$$;
    v_where_booking_leg text;
    v_where_flight text;
BEGIN
    FOR v_rec IN (SELECT * FROM json_each_text(p_json))
    LOOP
        CASE
            WHEN v_rec.key IN ('departure_airport','arrival_airport') THEN
                IF v_where_flight IS NULL THEN
                    v_where_flight :=
                        v_rec.key || '=' || quote_literal(v_rec.value);
                ELSE
                    v_where_flight := v_where_flight || ' AND ' ||
                        v_rec.key || '=' || quote_literal(v_rec.value);
                END IF;
            WHEN v_rec.key = 'scheduled_departure' THEN
                IF v_where_flight IS NULL THEN
                    v_where_flight :=
                        v_rec.key || $$ BETWEEN $$ ||
                        quote_literal(v_rec.value) || $$::date AND $$ ||
                        quote_literal(v_rec.value) || $$::date + 1$$;
                ELSE
                    v_where_flight :=
                        v_where_flight || ' AND ' ||
                        v_rec.key || $$ BETWEEN $$ ||
                        quote_literal(v_rec.value) || $$::date AND $$ ||
                        quote_literal(v_rec.value) || $$::date + 1$$;
                END IF;
            WHEN v_rec.key = 'flight_id' THEN
                v_where_booking_leg := 'bl.flight_id = ' || v_rec.value;
            ELSE
                NULL;
        END CASE;
    END LOOP;
    IF v_where_flight IS NULL THEN
        v_search_condition :=
            $$ WHERE $$ || v_where_booking_leg;
    ELSE
        v_search_condition :=
            $$ JOIN flight f1 ON f1.flight_id = bl.flight_id
            WHERE $$ || concat_ws(' AND ', v_where_flight, v_where_booking_leg);
    END IF;
    v_sql := v_sql || v_search_condition || ') s';
    EXECUTE v_sql INTO v_result;
    RETURN (v_result);
END;
$func$
LANGUAGE plpgsql;
```

И снова здесь много кода, но разные его части уже появлялись в главе 12 и в начале этой главы. Динамические критерии поиска построены аналогич-

но примеру, представленному в листинге 12.12, а список SELECT такой же, как и в листинге 13.5.

Мы считаем необходимым показать вам рабочий пример. Вы можете выполнить этот код в своей локальной копии `postgres_air` и попробовать вызывать функцию с другими параметрами.

Как мы уже неоднократно упоминали, построение функции с динамическим SQL – непростая задача, особенно вначале, и отладка займет дополнительное время. Когда вкладывать это время в процесс разработки – решать вам. Мы призываем вас поэкспериментировать с этими функциями, чтобы убедиться в их эффективности и стабильном времени выполнения.

Кроме того, когда вы разработаете один набор функций для классов одного из объектов приложения, то увидите, что вам будет намного проще создать аналогичные наборы функций для всех классов объектов, используя один и тот же шаблон.

Наконец, мы хотели бы упомянуть, что есть и другие способы создания динамического SQL для произвольных критериев поиска. В репозитории NORM на GitHub вы можете найти другие примеры.

Обновления

NORM может работать с любыми операциями манипулирования данными, то есть `INSERT`, `UPDATE` и `DELETE`, которые все вместе называются запросами на обновление.

Запрос на обновление отправляется из приложения в виде сложного объекта и на уровне базы данных может привести к нескольким операциям обновления, применяемым к разным таблицам.

И снова разработка базы данных управляется контрактом. Функция базы данных получает объект JSON из приложения, разбирает его и интерпретирует действия, которые требуются на уровне базы данных.

Вставка

Поскольку посадочный талон никогда не выдается в момент создания бронирования, функция, которая вставляет сегмент бронирования в базу данных, очень проста и вставляет строку только в одну таблицу, см. листинг 13.8.

Обновление

Для сегмента бронирования допустим лишь ограниченный набор обновлений, хоть он и представлен сложным объектом. В тех случаях, когда перебронирование разрешено, можно изменить номер рейса в сегменте бронирования, но нельзя менять сам рейс – для этого предусмотрены другие функции. Кроме

того, посадочные талоны всегда выдаются отдельно. Таким образом, в этом случае обновление ограничивается изменением номера рейса, выдачей посадочного талона или его удалением (обновить посадочный талон нельзя).

Чтобы объем кода оставался в пределах разумного, в листинге 13.9 показана функция обновления с ограниченной функциональностью.

Листинг 13.8 ❖ Функция `booking_leg_insert`

```
CREATE OR REPLACE FUNCTION booking_leg_insert (p_object json)
RETURNS SETOF text
AS $body$
DECLARE
    v_result booking_leg_record[];
    v_sql text;
    v_rec record;
    v_booking_id int;
    v_flight_id int;
    v_leg_num int;
    v_is_returning boolean;
    v_booking_leg_id int;
BEGIN
    FOR v_rec IN (SELECT * FROM json_each_text(p_object))
    LOOP
        CASE
            WHEN v_rec.key = 'booking_id' THEN
                v_booking_id := v_rec.value;
            WHEN v_rec.key = 'flight_id' THEN
                v_flight_id := v_rec.value;
            WHEN v_rec.key = 'leg_num' THEN
                v_leg_num := v_rec.value;
            WHEN v_rec.key = 'is_returning' THEN
                v_is_returning := v_rec.value;
            ELSE
                NULL;
        END CASE;
    END LOOP;
    INSERT INTO booking_leg (
        booking_id, flight_id, leg_num, is_returning, update_ts
    ) VALUES (
        v_booking_id, v_flight_id, v_leg_num, v_is_returning, now()
    )
    RETURNING booking_leg_id INTO v_booking_leg_id;
    RETURN QUERY (
        SELECT * FROM array_transport(booking_leg_select_json(v_booking_leg_id))
    );
END;
$body$
LANGUAGE plpgsql;
```

Листинг 13.9 ❖ Функция обновления booking_leg

```
CREATE OR REPLACE FUNCTION booking_leg_update (
    p_booking_leg_id int,
    p_object json
)
RETURNS SETOF text
AS $body$
DECLARE
    v_result booking_leg_record[];
    v_sql text;
    v_rec record;
    v_flight_id int;
    v_flight_each record;
    v_booking_leg_update text;
BEGIN
    FOR v_rec IN (SELECT * FROM json_each_text(p_object))
    LOOP
        CASE
            WHEN v_rec.key = 'flight' THEN
                FOR v_flight_each IN (
                    SELECT * FROM json_each_text(v_rec.value::json)
                )
                LOOP
                    CASE
                        WHEN v_flight_each.key = 'flight_id' THEN
                            v_flight_id := v_flight_each.value;
                            v_booking_leg_update := concat_ws(
                                ', ', v_booking_leg_update,
                                'flight_id = ' || quote_literal(v_flight_each.value)
                            );
                        ELSE
                            NULL;
                    END CASE;
                END LOOP;
            WHEN v_rec.key IN ('leg_num', 'is_returning') THEN
                v_booking_leg_update := concat_ws(
                    ', ', v_booking_leg_update,
                    v_rec.key || '=' || quote_literal(v_rec.value)
                );
            ELSE
                NULL;
        END CASE;
    END LOOP;
    IF v_booking_leg_update IS NOT NULL THEN
        EXECUTE ($$UPDATE booking_leg SET $$ || v_booking_leg_update ||
                 $$WHERE booking_leg_id = $$ || p_booking_leg_id::text);
    END IF;
    RETURN QUERY (
        SELECT * FROM array_transport(booking_leg_select_json(p_booking_leg_id))
    );
END;
$body$
LANGUAGE plpgsql;
```

Первый параметр этой функции – идентификатор обновляемого сегмента бронирования. Второй параметр – объект JSON, который функция интерпретирует, чтобы определить, какие таблицы и поля следует обновить. Обратите внимание, что независимо от того, какие ключи передаются в этом параметре, функция игнорирует все ключи, кроме тех, для которых мы указали алгоритм обработки. Например, хотя функция может получить все значения для записи рейса, она обработает только идентификатор рейса `flight_id`, который используется для обновления таблицы `booking_leg`. Хотя рейс – это вложенный объект в сегменте бронирования, его нельзя обновить как часть бронирования; для обновления рейсов используется отдельная функция, а измененная информация затем отобразится во всех зависимых бронированиях.

Например, выполним функцию вставки:

```
SELECT * FROM booking_leg_insert (
    $${"leg_num":3,
       "booking_id":232346,
       "flight_id":13650,
       "is_returning":"false"
      }$$::json
)
```

Результатом будет новый сегмент бронирования:

Data Output | Explain | Messages | Notifications

	booking_leg_insert text
1	{"booking_leg_id":17893568,"leg_num":3,"booking_id":232346,"flight":{"flight_id":13650,"flight_no":"1245","departure_airport_code"...

Затем этот новый сегмент обновляется:

```
SELECT * FROM booking_leg_update (
    17893568,
    $${"flight":{"flight_id":13651,"flight_no":"1240"},
       "is_returning":"true"
      }$$
)
```

В результате будет показан обновленный сегмент бронирования:

Data Output | Explain | Messages | Notifications

	booking_leg_update text
1	{"booking_leg_id":17893568,"leg_num":3,"booking_id":232346,"flight":{"flight_id":13651,"flight_no":"1245","departure_airport_code":"JFK","departure_...

Обратите внимание, что хотя в записи рейса и передается номер рейса `flight_no`, это значение игнорируется. Эта команда не меняет запись в таблице `flight`; она меняет только идентификатор рейса в таблице `booking_leg`.

Мы также можем создать функцию для выдачи посадочного талона, аналогичную той, которая была создана в главе 11, но возвращающую новый тип `booking_leg_record`.

Удаление

Для удаления компонента из сложного объекта используется специальный ключ `command`, у которого есть только одно допустимое значение `delete`. Например, посадочные талоны нельзя обновить. При необходимости старый посадочный талон удаляется и выдается новый.

Вызов функции для удаления посадочного талона может выглядеть так:

```
SELECT * FROM booking_leg_update (
    17893568,
    $${"boarding_passes":[{"boarding_pass_id":1247796,"command":"delete" }]
      }$$
)
```

Дополнительную информацию об обновлениях сложных объектов можно почерпнуть из репозитория NORM на GitHub: https://github.com/hettie-d/NORM.

Почему бы не хранить JSON?

Здесь вы можете спросить: зачем нужны такие сложности, когда PostgreSQL поддерживает тип JSON? Почему бы не хранить JSON в базе данных «как есть»?

Причины уже обсуждались в главе 9. В частности, там рассматривались модель «ключ–значение» и иерархические модели, объяснялись их ограничения. Если бы сегмент бронирования, определенный в данной главе, хранился как JSON, информация о рейсе дублировалась бы, потому что она принадлежит иной иерархии. Другая причина состоит в том, что в формате JSON нет типов и, следовательно, он ненадежен с точки зрения обеспечения согласованного интерфейса для разработки.

Кроме того, хотя мы можем создавать индексы для поиска по определенным ключам JSON, по производительности они уступают B-деревьям для обычных столбцов. Индексирование JSON и связанные с этим вопросы производительности рассматриваются в главе 14.

Прирост производительности

Как использование NORM влияет на производительность? Как обсуждалось в главе 10, такую разницу в производительности сложно измерить. Нам нужно измерять общую производительность приложения, а не сравнивать скорость отдельных операций, а сами приложения могут быть написаны

с использованием совершенно разных стилей программирования. Мы не приводим никаких примеров кода приложения в этой главе, поскольку это выходит за рамки данной книги.

Однако, исходя из нашего промышленного опыта, использование подхода NORM вместо традиционного ORM может повысить производительность контроллеров приложений в 10–50 раз. Более того, производительность приложений стабилизируется, поскольку устраняется *проблема N + 1 запроса* (большинство ORM по умолчанию откладывают загрузку, поэтому для получения дочерних элементов сначала выдается один запрос в базу данных для родительской записи, а затем по одному запросу для *каждой* дочерней записи).

Совместная работа с разработчиками приложений

Как уже неоднократно говорилось, общая производительность системы не ограничивается производительностью базы данных, и оптимизация начинается со сбора требований. NORM – очень хорошая иллюстрация к этому утверждению.

В NORM разработка начинается с определения контракта, что позволяет разработчикам приложений и разработчикам баз данных параллельно работать над своими задачами. Кроме того, этот контракт означает, что будущие улучшения производительности на стороне базы данных могут быть сделаны без внесения каких-либо изменений в приложение.

Выводы

NORM – это подход к проектированию и разработке приложений, обеспечивающий бесшовный обмен данными между серверной частью и уровнем данных, устраняя необходимость в ORM. При последовательном применении он помогает создавать высокопроизводительные системы, упрощая разработку приложения.

NORM – одно из нескольких возможных решений; тем не менее оно доказало свою успешность и может быть использовано как шаблонное всеми, кто хочет избежать возможных подводных камней ORM.

Глава 14

Более сложная фильтрация и поиск

В предыдущих главах обсуждались различные способы выполнения фильтрации и поиска с помощью индексов в PostgreSQL. Зачем нужны другие типы индексов, и почему они еще не рассматривались? Предварительное обсуждение этой темы было сосредоточено на наиболее распространенных индексах, которые необходимы в любом приложении. Однако есть типы данных, которые не могут эффективно поддерживаться такими индексами, как B-деревья.

Полнотекстовый поиск

Все, что обсуждалось в предыдущих главах, применимо к структурированным данным, и все рассмотренные до сих пор запросы следуют булевой логике. Строка либо нужна для вычисления результата, либо нет; вычисленная строка либо принадлежит результату, либо нет. Третьего не дано. SQL – мощный язык для структурированных данных, который хорошо подходит для такого рода анализа.

В этом разделе мы рассматриваем неструктурированные данные. Самый простой пример неструктурированных данных – текст, написанный на естественном языке. Такие тексты обычно называют *документами*. В отличие от структурированных данных, поиск документов всегда неточен, потому что нас обычно интересует смысл, а он не выражен точно в содержании документа. Зато критерии должны быть точно выражены в запросе. Добро пожаловать в мир неопределенности!

Существует несколько различных моделей поиска документов; та, что реализована в PostgreSQL, называется булевой моделью. Обратите внимание, что современные поисковые системы в интернете используют более сложные модели.

В булевой модели поиска документ рассматривается как список *термов*. Каждый терм обычно соответствует слову из документа, преобразованному с помощью определенных лингвистических средств. Такое преобразование

необходимо для повышения качества поиска. Например, мы ожидаем, что слова «слово» и «слова» должны соответствовать одному и тому же терму. Преобразования нетривиальны: «лечь» и «лягу» – это формы одного и того же слова, но «лягушка» – совсем другое. И это не просто морфологические преобразования: значение зависит от контекста.

Например, слова «хост» и «компьютер» имеют одинаковое значение в документе, описывающем сетевые протоколы, но их смысл различается, когда речь идет об организации конференции.

В PostgreSQL лингвистические правила, определяющие преобразования, инкапсулируются в *конфигурацию*. Доступно несколько предопределенных конфигураций для разных языков, и могут быть определены дополнительные. Одна из предопределенных конфигураций не зависит от языка и ограничивается простыми преобразованиями.

Результат лингвистической обработки представлен как значение типа `tsvector`. Значения `tsvector` не относятся к какому-либо языку или даже к тексту и представляют собой списки термов. Значение типа `tsvector` можно построить из любого списка значений.

Почему текстовый поиск называется *полнотекстовым поиском*? Во времена средневековья (1970-е годы), когда емкость жестких дисков была небольшой, списки термов составлялись только из заголовков или аннотаций. Таким образом, «полный текст» означает, что все слова в документе рассматриваются как источник термов.

Похожим образом запрос для поиска документа представлен как значение типа `tsquery`. Эти текстовые значения могут содержать слова и логические связки «и», «или» и «не». Простой запрос состоит только из слов.

Документ соответствует такому запросу, если все термы в запросе присутствуют в `tsvector`, соответствующем документу.

Оператор сопоставления `@@` возвращает истину, если документ удовлетворяет запросу, и ложь в противном случае. Его можно использовать в предложении `WHERE` команды `SELECT` или в любом другом месте, где ожидается логическое выражение.

Булев поиск дает определенные результаты: документ либо соответствует запросу, либо нет. Но поиск при этом неточный? Да, так и есть. Часть информации теряется при преобразовании документа в `tsvector`, а часть – при преобразовании запроса в `tsquery`.

Полнотекстовый поиск работает и без индексов, но PostgreSQL предоставляет специальные типы индексов, которые могут его ускорить. Мы обсудим эти типы индексов далее в данной главе.

Многомерный и пространственный поиск

Условия фильтрации, рассмотренные в предыдущих главах, использовали скалярные атрибуты. Некоторые виды индексов, например составные индексы, могут содержать несколько атрибутов, но в этом случае важен порядок атрибутов: индекс бесполезен, если не указано значение первого атрибута.

Фиксированный порядок атрибутов нежелателен для некоторых приложений и типов данных. Такие приложения требуют, чтобы несколько атрибутов обрабатывались симметрично, не отдавая предпочтение одному из них. Типичными примерами являются объекты на плоскости или в трехмерном пространстве; в качестве атрибутов выступают координаты. Такие виды данных называются пространственными.

Что еще более важно, для пространственных данных часто используются другие критерии поиска. Наиболее распространены:

- *запросы по диапазону* – найти все объекты, расположенные не дальше определенного расстояния от указанной точки в пространстве;
- *запросы ближайших соседей* – найти *k* объектов, ближайших к указанной точке.

Эти запросы не поддерживаются одномерными индексами, и несколько одномерных индексов вместе тоже не помогут.

Конечно, поиск такого вида не ограничивается одним лишь пространством. Координаты могут иметь значения времени или представлять события в дискретной области.

PostgreSQL предоставляет типы индексов, подходящие для пространственных данных. Они кратко обсуждаются в следующем разделе.

Обобщенные типы индексов PostgreSQL

Оператор `CREATE INDEX` позволяет при создании индекса указать его тип. Ни в одном из предыдущих примеров это не было показано, потому что все ранее созданные индексы были B-деревьями, а именно B-дерево используется по умолчанию в качестве типа индекса. Другими возможными значениями типа на момент написания этой главы были `hash`, `gist`, `spgist`, `gin` и `brin`.

В данном разделе более подробно рассматриваются некоторые из этих типов.

Индексы GiST

Некоторые приложения используют объекты с несколькими атрибутами, например точки на поверхности или на плоскости. Ранее мы обсуждали составные индексы, включающие несколько атрибутов. Однако составные индексы не симметричны: атрибуты в них упорядочены. Напротив, координаты точки следует считать симметричными.

Для таких приложений подойдет тип индекса GiST, который индексирует точки. Условия поиска выражаются прямоугольником: возвращаются все точки, находящиеся внутри этого прямоугольника.

Если говорить точнее, GiST – это семейство индексных структур, каждая из которых поддерживает определенный тип данных. В дистрибутив PostgreSQL включена поддержка точек; некоторые другие типы данных могут быть установлены как расширения.

Конечно, атрибуты, включаемые в индекс GiST, не обязательно должны быть координатами. Такие атрибуты могут представлять время или другие упорядоченные типы данных.

Индексы для полнотекстового поиска

PostgreSQL предоставляет два типа индексов, которые поддерживают текстовый поиск. Начнем с обсуждения индексов GIN, что означает Generalized Inverted Index – обобщенный инвертированный индекс.

В целях индексации документ рассматривается как список термов (или фрагментов) и представляется значением типа данных `tsvector`, описанного ранее.

Для каждого терма, содержащегося хотя бы в одном документе, инвертированный индекс содержит список документов, содержащих этот терм. Таким образом, общая структура симметрична: у документа есть список термов, а у терма есть список документов. Эта симметрия объясняет, почему тип индекса называется инвертированным.

Инвертированные индексы могут эффективно поддерживать текстовый поиск. Например, чтобы найти документы, содержащие все термы, указанные в запросе, PostgreSQL сканирует все списки документов этих термов и оставляет только те документы, которые присутствуют в каждом из списков. Списки упорядочены, поэтому для получения набора результатов достаточно одного прохода по спискам.

Индекс GIN может быть создан как функциональный индекс с выражением, преобразующим индексируемый документ в `tsvector`, или же значения `tsvector` могут храниться как отдельный столбец. Преимущество первого подхода состоит в экономии места, а преимущество последнего в том, что индекс не зависит от конфигурации (поскольку конфигурация нужна только для вычисления значения `tsvector`). Если значения `tsvector` сохраняются, то можно индексировать документы, написанные на разных естественных языках и преобразованные в `tsvector` разными конфигурациями.

Структура GIN не является производной от естественного языка и никак не связана с ним; как отмечалось ранее, документы считаются списками. Таким образом, индекс может работать с данными, отличными от документов, – фактически с любым типом атрибута, содержащим несколько элементов, например с массивами. Индекс GIN будет находить строки с атрибутами, содержащими все элементы, указанные в запросе, точно так же как он находит все документы, содержащие указанные термы.

Документы (значения типа `tsvector`) также можно индексировать с помощью индекса GiST. Чтобы создать такой индекс, значения `tsvector` преобразуются в битовые карты фиксированной длины, построенные следующим образом. Каждому терму, который появляется в любом из индексируемых документов, соответствует позиция в битовой карте, которая определяется значением хеш-функции, вычисленной для этого терма. Каждый терм представляется битовой картой той же длины, в которой установлен в единицу единственный бит в вычисленной позиции; все остальные биты равны нулю. Битовая карта документа – это побитовое логическое «или» всех битовых

карт, которые представляют термы документв. Таким образом, в битовой карте документа установлены в единицу все биты, соответствующие термам, которые присутствуют в этом документе. Битовая карта запроса строится аналогичным образом.

Поиск по такому индексу основан на том, что документ удовлетворяет запросу, если его битовая карта содержит единицу в каждой позиции, в которой единица содержится в битовой карте запроса.

Хеш-функция может поместить различные термы в одну и ту же позицию. Поэтому индекс GiST может возвращать документы, не относящиеся к запросу, и обычно требуется перепроверка значений `tsvector`, но PostgreSQL выполняет ее автоматически.

Количество ложных совпадений увеличивается с ростом количества различных термов. Следовательно, индекс GiST эффективен для коллекций документов, в которых общее количество различных термов невелико. Для текстов, написанных на естественных языках, это, как правило, не так, поэтому индексы GIN обычно более эффективны. Но в некоторых случаях индексы GiST для текстового поиска могут оказаться полезными.

Индексирование очень больших таблиц

Любой индекс занимает некоторое место, а индексы для больших таблиц могут быть очень большими. Можно ли уменьшить размер индекса?

В учебниках по базам данных различают плотные и разреженные индексы. Все рассмотренные до настоящего момента индексы являются плотными: они содержат все значения индексированного столбца (или столбцов). Разреженный индекс содержит только часть всех значений, но уменьшает количество чтений, необходимых для поиска любого из значений индексированного атрибута. Такие индексы отличаются от частичных индексов, которые не ускоряют поиск значений, не включенных в индекс.

Некоторые системы баз данных всегда хранят строки таблицы в порядке (суррогатного) первичного ключа. В таких системах индекс по первичному ключу может содержать только одно значение на каждый блок таблицы и будет в таком случае разреженным. Более продвинутые системы баз данных позволяют упорядочивать таблицы по значениям других атрибутов, не обязательно уникальных. Такая организация таблиц также допускает разреженные индексы. Иногда разреженные индексы называют кластерными, поскольку строки с одинаковым значением индексируемого столбца размещены близко друг к другу.

PostgreSQL не предоставляет средств для явного управления порядком строк. Однако во многих случаях строки упорядочиваются естественным образом. Например, строки, регистрирующие определенные виды событий, будут, скорее всего, добавляться к таблице и будут естественным образом упорядочены по времени добавления, что позволяет использовать для них разреженную индексацию.

Обобщение разреженных индексов, реализованное в PostgreSQL, называется BRIN, что расшифровывается как Block Range Index, индекс зон блоков.

Таблица, для которой создается индекс BRIN, рассматривается как последовательность зон, где каждая зона состоит из фиксированного числа смежных блоков. Для каждой зоны запись индекса BRIN содержит сводку значений, находящихся в этой зоне. Например, сводка может содержать минимальное и максимальное значения столбца временной отметки в таблице журнала событий.

Чтобы найти любое значение индексированного атрибута, достаточно найти соответствующую зону блоков (используя индекс), а затем просканировать все блоки в этой зоне.

Способ вычисления сводки зависит от типа индексируемого столбца. Для интервалов сводка может быть интервалом, включающим все интервалы, содержащиеся в зоне блоков. Для пространственных данных сводка может быть ограничивающим прямоугольником, содержащим все прямоугольники в зоне блоков.

Если значения столбцов не упорядочены или строки в таблице не упорядочены, индекс BRIN вернет несколько зон блоков, каждую из которых придется просканировать.

Вычисление сводки обходится дорого. Поэтому PostgreSQL предоставляет несколько вариантов сопровождения индекса BRIN: индекс может обновляться с помощью триггеров, сводка может обновляться автоматически вместе с очисткой или может быть пересчитана вручную.

Индексирование JSON и JSONB

Иногда разработчики в поисках гибкости преобразуют строки таблицы в текстовый или слабоструктурированный формат (JSON или XML), а затем используют полнотекстовый поиск вместо более точных индексов. Такой подход определенно работает лучше, чем внешние инструменты индексирования, но он значительно медленнее, чем точные индексы.

Вернемся к вопросу, которым мы задались в конце главы 13: зачем возиться с созданием функций, которые преобразуют результаты поиска в JSON, если можно просто сохранить тип JSON непосредственно в базе данных?

Посмотрим, как такой подход будет работать на практике. Для этого создадим таблицу, в которой сохраним бронирования в виде объектов JSON. Первая проблема, с которой мы столкнемся, состоит в том, что нам могут понадобиться разные структуры JSON для разных точек входа приложения (мы уже создали несколько различных типов записей в главах 11–13). Но предположим, что можно объединить различные требования и хранить данные таким образом, чтобы это удовлетворяло большинству сценариев использования. Тогда мы можем взять код, представленный в листинге 14.1.

Обратите внимание, что мы создаем таблицу с типом столбца JSONB (JSON Binary), а не JSON. Единственная разница между этими типами заключается в том, что JSONB хранит двоичное представление данных JSON, а не строку. Для типа JSON можно создать только индексы на основе B-деревьев по определенным тегам, а для этого нужно указать полный путь в JSON, включая

индексы массивов, так что проиндексировать «любой» сегмент бронирования не получится.

Листинг 14.1 ❖ Создание таблицы с типом столбца JSONB

```
-- упрощенный тип для сегмента бронирования
CREATE TYPE booking_leg_record_2 AS (
    booking_leg_id integer,
    leg_num integer,
    booking_id integer,
    flight flight_record
);
-- упрощенный тип для бронирования
CREATE TYPE booking_record_2 AS (
    booking_id integer,
    booking_ref text,
    booking_name text,
    email text,
    account_id integer,
    booking_legs booking_leg_record_2[],
    passengers passenger_record[]
);
-- таблица
CREATE TABLE booking_jsonb AS
SELECT b.booking_id,
       to_jsonb( row (
                   b.booking_id,
                   b.booking_ref,
                   b.booking_name,
                   b.email,
                   b.account_id,
                   ls.legs,
                   ps.passengers
                 )::booking_record_2
       ) AS cplx_booking
  FROM booking b
       JOIN ( SELECT booking_id,
                     array_agg( row (
                         booking_leg_id,
                         leg_num,
                         booking_id,
                         row (
                              f.flight_id,
                              flight_no,
                              departure_airport,
                              dep.airport_name,
                              arrival_airport,
                              arv.airport_name,
                              scheduled_departure,
                              scheduled_arrival
                         )::flight_record
                     )::booking_leg_record_2) legs
```

```
                FROM booking_leg l
                     JOIN flight f ON f.flight_id = l.flight_id
                     JOIN airport dep ON dep.airport_code = f.departure_airport
                     JOIN airport arv ON arv.airport_code = f.arrival_airport
                GROUP BY booking_id
           ) ls ON b.booking_id = ls.booking_id
      JOIN ( SELECT booking_id,
                     array_agg( row(
                         passenger_id,
                         booking_id,
                         passenger_no,
                         last_name,
                         first_name
                     )::passenger_record) AS passengers
                FROM passenger
                GROUP BY booking_id
           ) ps ON ls.booking_id = ps.booking_id
;
```

Если мы хотим иметь высокопроизводительные индексы для столбцов JSON, необходимо использовать тип JSONB.

Создание таблицы займет некоторое время. Индекс GIN тоже строится не мгновенно:

```
CREATE INDEX idxgin ON booking_jsonb USING GIN (cplx_booking);
```

Но после того, как индекс создан, кажется, что в мире уже не осталось нерешенных проблем. Теперь мы можем получить все необходимые данные без каких-либо соединений и построения сложной структуры, используя простые запросы, подобные тому, что показан в листинге 14.2.

Листинг 14.2 ❖ Поиск с использованием индекса GIN по столбцу JSONB

```
SELECT *
  FROM booking_jsonb
 WHERE cplx_booking @@
       '$.**.departure_airport_code == "ORD" && $.**.arrival_airport_code == "JFK"'
```

План выполнения на рис. 14.1 показывает, что индекс GIN используется.

	QUERY PLAN text
1	Bitmap Heap Scan on booking_jsonb (cost=199.73..21438.53 rows=5643 width=1205)
2	Recheck Cond: (cplx_booking @@ '($.**."departure_airport_code" == "ORD" && $.**."arrival_airport_code" == "JFK")'::jsonpath)
3	-> Bitmap Index Scan on idxgin (cost=0.00..198.32 rows=5643 width=0)
4	Index Cond: (cplx_booking @@ '($.**."departure_airport_code" == "ORD" && $.**."arrival_airport_code" == "JFK")'::jsonpath)

Рис. 14.1 ❖ План выполнения с индексом GIN

Но есть несколько сложностей, которые делают этот подход менее привлекательным, чем кажется на первый взгляд. Во-первых, поиск работает медленнее, чем поиск с использованием B-дерева. Например, функция, ко-

торую мы создали в главе 13, выполняется в 2–2,5 раза быстрее, чем поиск с использованием индекса GIN.

```
SELECT *
  FROM search_booking_leg(
           $${"departure_airport":"ORD", "arrival_airport":"JFK"}$$::json
       )
```

Во-вторых, индексы GIN не поддерживают поиск по атрибутам даты и времени, поиск с использованием оператора `LIKE` или поиск по любым преобразованным значениям атрибутов, например `lower`. В предложении `WHERE` вы можете указать несколько сложных условий поиска с выражениями `json_path` и операторами и функциями JSONB, включая регулярные выражения, но проверяться они будут сканированием таблицы. Лучше будет объединить эти условия с теми, что поддерживаются индексами.

Вы можете поспособствовать поиску, создав дополнительные индексы триграмм. Фактически мы видели промышленную систему, которая была построена таким образом: на основе данных из нескольких таблиц и схем были созданы «поисковые документы» типа JSON, а затем были добавлены и проиндексированы столбцы `tsvector`.

Однако с этим подходом связана и третья сложность. Как уже говорилось, одна структура JSON будет поддерживать только одну иерархию. Если бы мы создали столбец `booking_jsonb`, как было описано ранее, то могли бы относительно легко обновить рейс в сегменте бронирования, но не смогли бы обновить фактическое время вылета или статус рейса.

Это означает, что таблицу `booking_jsonb` придется периодически перестраивать, чтобы она оставалась полезной. Действительно, упомянутая выше промышленная система имела сложную последовательность триггеров, которые перестраивали все потенциально затронутые данные JSON. При относительно низком ожидаемом количестве обновлений такое ограничение может быть некритичным, но к задержкам рейсов и изменению расписания это точно не относится.

Выводы

PostgreSQL располагает множеством различных типов индексов. В этой книге рассказывается о многих из них, но не обо всех; новые типы индексов появляются почти в каждой версии. Мы не удивимся, если к дате публикации книги появится еще какой-нибудь.

В главе 5 и в этой главе приводится ряд примеров того, как правильно выбрать индексы для поддержки различных типов поиска. Выбор индексов, которые лучше всего подходят для вашей системы, для конкретного поиска, – непростая задача. Не останавливайтесь на создании B-деревьев для отдельных столбцов. Нужны ли составные индексы? Функциональные индексы? Поможет ли индекс GiST решить вашу проблему? Насколько критично время ответа на этот конкретный запрос? Насколько сильно этот конкретный индекс влияет на обновления? Только вы можете ответить на данные вопросы.

Глава 15

Полный и окончательный алгоритм оптимизации

В предыдущих главах было рассмотрено множество методов оптимизации: не только различные способы оптимизации инструкций SQL, но и то, как схема базы данных влияет на производительность. Мы обсудили, почему важно взаимодействовать с разработчиками приложений, поговорили про использование функций и затронули много других аспектов производительности базы данных.

Тем не менее остается вопрос, который мы поставили во введении: с чего начать, когда возникает реальная проблема? Что делать, когда пользователи видят на экране песочные часы, а вы не знаете, в чем причина? Связанная с этим, но более сложная задача – понять, что делать с самого начала. Проблемы пока нет. Пока у вас есть задача, возможно черновик запроса, или, быть может, вам повезло и имеются подробные требования. Как убедиться, что вы все делаете правильно?

В этой главе мы представим пошаговое руководство, которое поможет вам сразу же писать запросы правильно и, при наличии возможности, выбирать подходящую вам схему базы данных.

Основные шаги

На рис. 15.1 представлена блок-схема, которая поможет определить лучшую стратегию для рассматриваемого запроса. В следующих разделах мы обсудим каждый шаг более подробно.

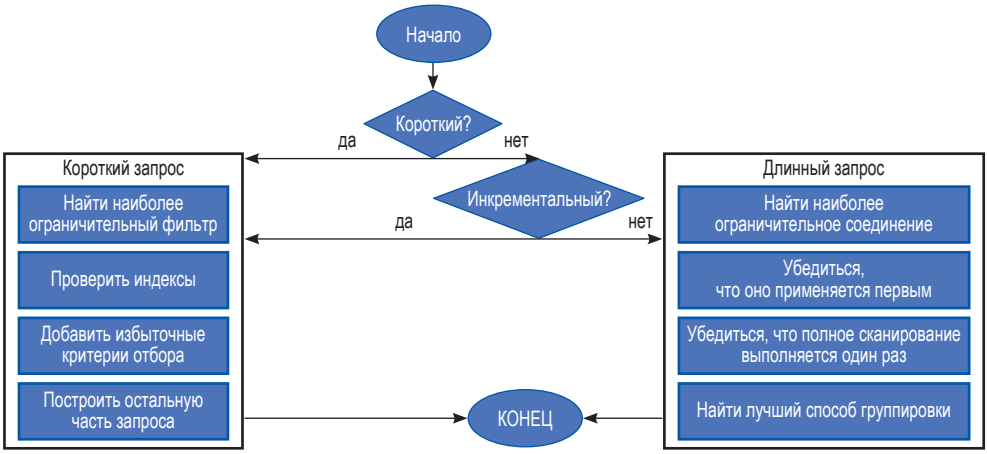


Рис. 15.1 ❖ Шаги полного и окончательного алгоритма оптимизации

Пошаговое руководство

Шаг 1. Короткий запрос или длинный?

Первый шаг – определить, является рассматриваемый запрос коротким или длинным. Как обсуждалось в главах 5 и 6, изучение самого запроса не обязательно поможет вам найти ответ. Вспомните, что оптимизация запросов начинается со сбора требований и важно работать вместе с владельцами бизнеса, с бизнес-аналитиками или со всеми вместе.

Проверьте, важны ли компании самые свежие данные или нужно исследовать исторические тенденции и т. д. Представители бизнеса могут сказать, что им нужны все отмененные рейсы, но стоит спросить, хотят ли они увидеть все отмененные рейсы с начала времен или все-таки за последние сутки.

Если вы определили, что рассматриваемый запрос является коротким, переходите к шагу 2; в противном случае переходите к шагу 3.

Шаг 2. Короткий запрос

Итак, вы имеете дело с коротким запросом. Какие шаги необходимо предпринять, чтобы не только написать запрос наилучшим образом, но и чтобы его производительность была стабильной даже при росте объема данных?

Шаг 2.1. Самые ограничительные критерии

Найдите самые ограничительные критерии для вашего запроса. Помните, что зачастую для этого недостаточно просто посмотреть на запрос. Проверьте таблицы, чтобы найти количество различных значений атрибутов. Помните о распределении значений, выясните, какие значения встречаются

реже всего. Когда наиболее ограничительные критерии будут определены, переходите к следующему шагу.

Шаг 2.2. Проверьте индексы

Проверьте, созданы ли индексы, поддерживающие поиск по наиболее ограничительным условиям. А именно:

- проверьте, проиндексированы ли все атрибуты поиска для наиболее ограничительного условия. Если индекс (или индексы) отсутствует, запросите создание или создайте сами;
- если задействовано несколько полей, проверьте, будет ли составной индекс работать лучше и оправдывает ли прирост производительности создание дополнительного индекса;
- проверьте, можно ли использовать сканирование только индекса с использованием составного или покрывающего индекса.

Шаг 2.3. Добавьте избыточный критерий отбора, если это применимо

Если наиболее ограничительное условие основано на комбинации атрибутов из разных таблиц и поэтому не может быть проиндексировано, подумайте о добавлении избыточного критерия отбора.

Шаг 2.4. Построение запроса

Начните писать запрос, применяя самые ограничительные критерии; для этого может потребоваться начать запрос с одной таблицы или с соединения, которое включает в себя наиболее ограничительный критерий.

Не пропускайте этот шаг. Часто, когда разработчики баз данных знают связи между объектами, они выписывают все соединения перед применением фильтрации. Хотя мы знаем, что такой подход нередко рекомендуется, мы считаем, что в случае сложных запросов со многими соединениями он может усложнить разработку. Мы предлагаем начать с запроса, который точно выполняется эффективно, и затем добавлять по одной таблице за раз.

Проверяйте производительность запроса и план выполнения каждый раз, когда добавляете новое соединение. Помните, что оптимизаторы склонны ошибаться при оценке размера промежуточных результатов по мере приближения к корню дерева выполнения. Если количество соединений в запросе приближается к десяти, стоит рассмотреть возможность использования общих табличных выражений (если вы используете версию 12 или выше) или подумать о создании динамического SQL.

Шаг 3. Длинный запрос

Вы имеете дело с длинным запросом. Сначала определите, можно ли использовать инкрементальное обновление. Здесь вам снова нужно поработать вместе с владельцем бизнеса и бизнес-аналитиками, чтобы лучше понять,

какова цель запроса. Часто требования формулируются без учета динамики данных. Когда результаты запроса хранятся в таблице и должны периодически обновляться, данные можно каждый раз получать заново (полное обновление, извлекающее все данные от начала времен до последнего момента), а можно получать постепенно, добавляя только то, что изменилось с момента последнего обращения. Последний способ и есть то, что мы подразумеваем под инкрементальным обновлением. В подавляющем большинстве случаев данные можно получать постепенно. Например, вместо создания материализованного представления `passenger_passport`, которое было показано в главе 7, создайте его как таблицу и добавляйте к ней строки (или обновляйте их) при вводе новых паспортных данных.

Если есть возможность использовать инкрементальное обновление, переходите к шагу 4; в противном случае переходите к шагу 5.

Шаг 4. Инкрементальные обновления

Рассматривайте запрос недавно добавленных или обновленных записей как короткий, причем наиболее ограничительным критерием в этом случае является время обновления. Переходите к шагу 2 и выполняйте шаги по оптимизации коротких запросов.

Шаг 5. Неинкрементальный длинный запрос

Если инкрементальное обновление невозможно, выполните следующие шаги по оптимизации длинных запросов:

- найдите наиболее ограничительное соединение, полусоединение или антисоединение, если это применимо (подробности см. в главе 6), и убедитесь, что оно выполняется первым;
- продолжайте соединять таблицы одну за другой и каждый раз проверяйте время выполнения и план выполнения;
- убедитесь, что большие таблицы не сканируются многократно. Планируйте запрос так, чтобы он читал большие таблицы только один раз, как описано в главе 6;
- обратите внимание на группировку. В большинстве случаев стоит отложить группировку до последнего шага, то есть нужно убедиться, что `GROUP BY` является последней командой в плане выполнения. Помните о случаях, описанных в главе 6, когда группировку следует выполнять раньше, чтобы минимизировать размер промежуточных наборов данных.

Но подождите, это еще не все!

На протяжении всей книги мы настаивали на том, что оптимизация базы данных не ограничивается оптимизацией отдельных запросов; отдельные запросы не берутся из ниоткуда. Алгоритм оптимизации, описанный в пре-

дыдущих разделах, является руководством только для процесса оптимизации отдельного запроса или, скорее, правильного написания запроса с самого начала. Однако мы рассмотрели и несколько других методов.

Вот что еще следует учитывать:

- *параметры* – скорее всего, запрос, который вы оптимизируете, параметризован, то есть если вы имеете дело с идентификатором рейса, то это не фиксированное значение `flight_id = 1234`, а произвольный номер. Как мы обсуждали в главе 5, от конкретных значений может зависеть, какое из условий окажется самым ограничительным (например, статус отмены рейса будет более ограничительным, чем большинство других критериев);
- *динамический SQL* – если от значений зависит, какое из условий является наиболее ограничительным, или сами условия могут меняться – правильным подходом будет использование динамического SQL;
- *функции* – как обсуждалось в главе 11, сами по себе функции не улучшают производительность и даже могут значительно увеличить время выполнения. Однако если необходим динамический SQL, без функций обойтись сложно;
- *изменения в структуре базы данных* – работая с запросами, вам может потребоваться выполнить команды DDL, от создания новых индексов до изменения схемы таблицы. Вам нужно будет поработать с администраторами баз данных и системными архитекторами, чтобы определить, какие изменения можно реализовать;
- *взаимодействие с приложением* – ваш запрос выполняется приложением; производительность запроса может быть довольно хорошей, а общая производительность приложения при этом – нет. Если вы и ваша команда решите использовать NORM или другой подобный подход, нужно будет поработать с разработчиками приложений, чтобы определить, что принадлежит бизнес-логике, а что – логике базы данных, как описано в главах 11 и 13.

Выводы

В этой главе содержится пошаговое руководство, которое поможет вам сориентироваться в том, как писать правильные запросы с самого начала. Мы рекомендуем попробовать выполнить эти шаги при работе над вашим следующим проектом.

Глава 16

Заключение

Как и все хорошее, эта книга подошла к концу.

Во введении мы рассказали, что написали эту книгу, потому что не могли не написать ее. Бесчисленное количество раз мы сталкивались с вопросом: «Есть ли книга, которую вы могли бы порекомендовать для начала работы с PostgreSQL?»

В большинстве случаев этот вопрос задавали не новички в разработке баз данных. PostgreSQL пока еще не преподается во всех образовательных учреждениях, и типичный разработчик, новичок в PostgreSQL, уже знает, как писать синтаксически правильные запросы, но незнаком с особенностями PostgreSQL. Сюда входят не только незначительные языковые различия, но, что более важно, различия в способах хранения данных и обработке запросов.

Да, конечно, есть документация, но в ней не всегда легко найти то, что вам нужно, если только вы знаете точно, что ищете. Есть и другие ресурсы, включая множество отличных руководств по различным темам, а также блоги ведущих экспертов по PostgreSQL. Однако в большинстве случаев они сосредоточены на чем-то конкретном, демонстрируют многочисленные замечательные возможности PostgreSQL, но не обязательно показывают, как именно эти возможности укладываются в общую картину.

Конечно, эта книга также не дает полной картины – PostgreSQL может многое предложить, и мы не пытались дать исчерпывающий обзор. Мы подходим к PostgreSQL с другой точки зрения, демонстрируя, как заставить работать эти замечательные функции.

Таким образом, мы надеемся, что книга станет полезной для разработчиков баз данных, которые начинают изучать PostgreSQL. Мы также надеемся, что те, кто уже какое-то время использует PostgreSQL, также найдут здесь полезную информацию, возможно какие-то методы, которые они раньше не использовали.

Наша цель – дать вам структуру, которая поможет ориентироваться в сложных задачах разработки базы данных, и источник сведений, с которым вы сможете свериться в ситуациях «а что, если». Хотя невозможно охватить в одной книге все, что может предложить PostgreSQL, мы надеемся, что ее использование в качестве руководства поможет вам найти более подробную информацию в документации PostgreSQL.

На протяжении всей книги мы пытались объяснить не только что надо делать, но и почему это работает. Ведь если вы знаете ответ на вопрос «почему», то сможете распознать и другие ситуации, в которых подобное решение может сработать. Понимание реляционной теории – ключ к пониманию этих «почему», поэтому книга начинается с внушительного объема теории. Труд тех, кто усердно изучал эти главы, будет вознагражден. Эти теоретические основы на один шаг приближают вас к тому, чтобы «мыслить как база данных», что позволяет сразу же писать запросы правильно, а не «сначала писать, а затем оптимизировать».

Кроме того, мы представили схему postgres_air с открытым исходным кодом, которая доступна на странице https://github.com/hettie-d/postgres_air. Мы надеемся, что этот реалистичный набор данных будет полезен в качестве инструмента для обучения, экспериментирования и демонстрации, а также как образовательный ресурс.

Во введении мы включили в нашу целевую аудиторию ИТ-специалистов, работающих с PostgreSQL и желающих разрабатывать производительные и масштабируемые приложения, всех, чья должность содержит слова «разработчик базы данных» или «администратор базы данных», и серверных разработчиков, взаимодействующих с базой данных. Мы надеемся, что одним из выводов, который вы сделаете после прочтения этой книги, станет то, что сотрудничество между всеми этими группами и владельцами бизнеса является ключом к разработке эффективных приложений.

Запросы к базе данных не выполняются в вакууме: база данных – это служба. Работа базы данных невидима, если все работает хорошо, и очень заметна, если что-то идет не так. Надеемся, что вашу работу будут замечать нечасто!

Получайте удовольствие от PostgreSQL!

Предметный указатель

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38;
тел.: **(499) 782-38-89**, электронная почта: **books@alians-kniga.ru**.
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: **http://www.galaktika-dmk.com/**.

Генриэтта Домбровская
Борис Новиков
Анна Бейликова

Оптимизация запросов в PostgreSQL

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Зам. главного редактора *Сенченкова Е. А.*
Перевод *Беликов Д. А.*
Редактор *Рогов Е. В.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 22,59. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**

Полное руководство по созданию эффективных запросов

Эта книга поможет вам писать запросы, которые выполняются быстро и вовремя доставляют результаты. Прочитав ее, вы научитесь смотреть на процесс написания запроса с точки зрения механизма базы данных и начнете думать, как оптимизатор базы данных.

Сначала рассказывается о том, что такое эффективная система, как измерить производительность и поставить связанные с ней цели. Представлены различные классы запросов и подходящие для каждого из них методы оптимизации, такие как использование индексов и определенных алгоритмов соединения. Объясняется, как читать и понимать планы выполнения запросов, какие существуют методы воздействия на эти планы с точки зрения оптимизации производительности. Далее рассматриваются сложные темы: использование функций и процедур, динамический SQL и сгенерированные запросы. Показано, как эти методы используются вместе для создания эффективных приложений.

Изучив книгу, вы сможете:

- определять цели оптимизации в системах OLTP и OLAP;
- читать и понимать планы выполнения PostgreSQL;
- различать короткие и длинные запросы;
- выбирать подходящую технику оптимизации для каждого типа запроса;
- определить индексы, которые улучшат производительность запросов;
- оптимизировать полное сканирование таблицы;
- избегать проблем, возникающих при работе с системами объектно-реляционного отображения;
- оптимизировать все приложение, а не только запросы к базе данных.

Издание предназначено разработчикам и администраторам баз данных, а также системным архитекторам, участвующим в проектировании прикладных систем, использующих базу данных PostgreSQL.

Интернет-магазин: www.dmkpress.com

Оптовая продажа: КТК «Галактика»
books@alians-kniga.ru

ДМК ПРЕСС издательство
www.дмк.рф

Apress
www.apress.com

ISBN 978-5-97060-963-7