Содержание

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.

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.

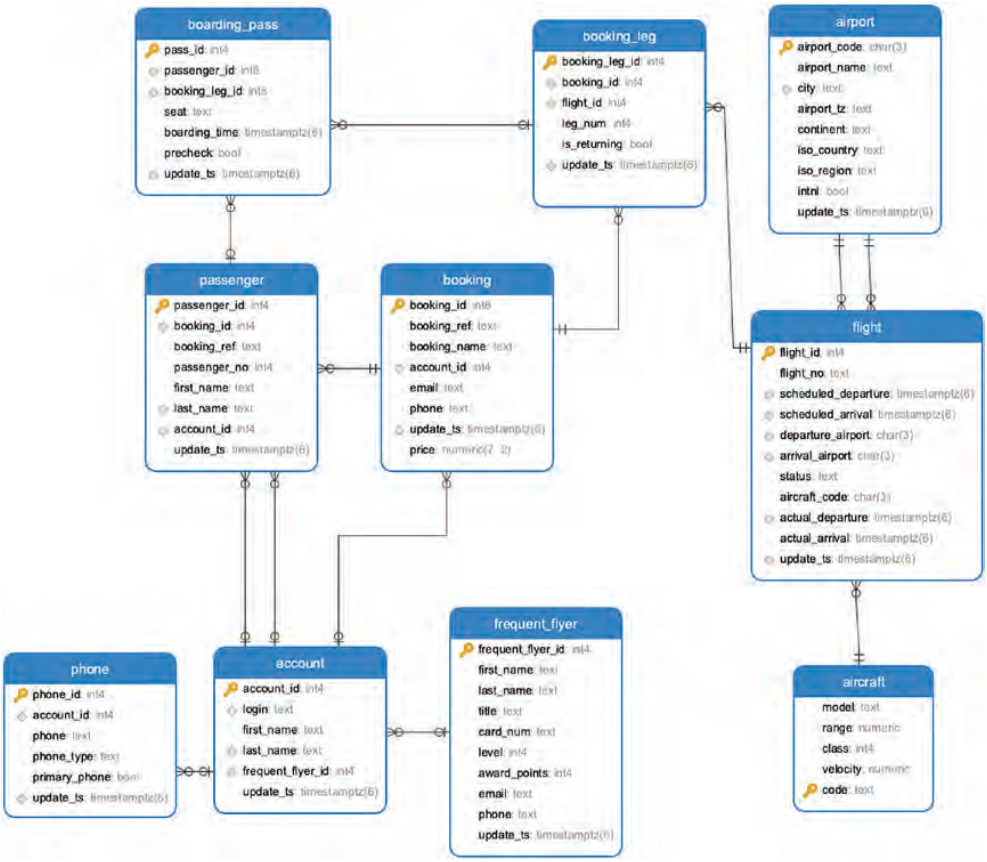

Рис. В.2 ❖ ER-диаграмма схемы бронирования

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

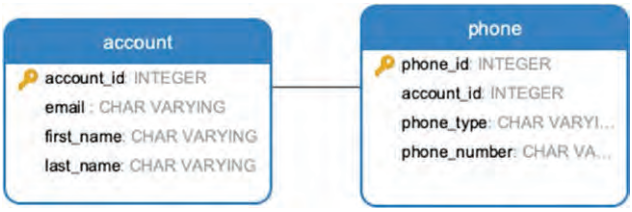

Рис. 1.2 ❖ Вариант с двумя таблицами

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.