HIGH TECH

СТЕФАН ФАРО

РЕФАКТОРИНГ SQL ПРИЛОЖЕНИЙ

СИМВОЛ®

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 978-5-93286-145-5, название «Рефакторинг SQL-приложений» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.

Refactoring SQL Applications

Stephane Faroult with Pascal L'Hermite

O'REILLY®

Рефакторинг SQL-приложений

Стефан Фаро и Паскаль Лерми

Санкт-Петербург — Москва
2009

Стефан Фаро и Паскаль Лерми

Рефакторинг SQL-приложений

Перевод Ф. Гороховского

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Выпускающий редактор	*А. Пасечник*
Редакторы	*Д. Мотрич, И. Зайковская*
Научный редактор	*Б. Попов*
Корректор	*С. Беляева*
Верстка	*Д. Петров*

Фаро С., Паскаль Л.

Рефакторинг SQL-приложений. – Пер. с англ. – СПб: Символ-Плюс, 2009. – 336 с., ил.

ISBN: 978-5-93286-145-5

Когда поднимается вопрос рефакторинга кода, специалист может быть уверен, что либо возникла серьезная проблема, либо предполагается, что она проявится в ближайшее время. Как правило, при этом известно, что следует улучшить в плане функциональности, но прежде необходимо понять природу проблемы.

В книге делается попытка дать реалистичный и честный обзор методов усовершенствования приложений SQL и определить рациональную концепцию для тактических маневров. Часто рефакторинг напоминает безумный поиск быстрых побед и эффектных усовершенствований, которые можно вписать в бюджет и сохранить голову на плечах. Но разумное и систематическое применение правильных принципов может привести к впечатляющим результатам. Эта книга поможет выработать правильную тактику и оценить перспективы различных решений.

Книга предназначена для профессионалов в области информационных технологий, разработчиков, менеджеров проектов, служб поддержки, администраторов баз данных и специалистов по настройке, которым приходится принимать участие в операциях по спасению приложений со значительным объемом кода управления базами данных.

ISBN: 978-5-93286-145-5
ISBN: 978-0-596-51497-6 (англ)

© Издательство Символ-Плюс, 2009

Authorized translation of the English edition © 2008 O'Reilly Media, Inc. This translation is published and sold by permission of O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

Все права на данное издание защищены Законодательством РФ, включая право на полное или частичное воспроизведение в любой форме. Все товарные знаки или зарегистрированные товарные знаки, упоминаемые в настоящем издании, являются собственностью соответствующих фирм.

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.

Подписано в печать 31.03.2009. Формат 70×100 1/16. Печать офсетная.
Объем 21 печ. л. Тираж 1500 экз. Заказ №

Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Об авторах

Стефан Фаро (Stéphane Faroult) занимается реляционными базами данных и языком SQL с 1983 года. Он присоединился к Oracle France в самом начале существования этой организации (после недолгого периода работы в IBM и преподавания в Университете Оттавы) и вскоре сосредоточился на вопросах производительности и настройки. Покинув Oracle в 1988 году, он некоторое время занимался исследованием операций, но через год снова вернулся к реляционным базам данных и с тех пор занимается консультированием в этой области. В 1998 году основал компанию RoughSea Ltd.

Паскаль Лерми (Pascal L'Hermite) последние 12 лет работает специалистом по средам оперативной обработки транзакций и разработки реляционных баз данных Oracle, а последние пять лет – и разработки баз данных Microsoft SQL Server.

Предисловие

Желая написать нечто полезное для людей понимающих,
я стремлюсь скорее следовать реальной природе вещей,
чем нашим представлениям о них.

Никколо Макиавелли
Государь, XV

Эту книгу я хочу предварить небольшой историей. Я с большим трудом закончил книгу «The Art of SQL». Она еще не поступила в продажу, когда мой редактор, Джонатан Генник, подал идею о написании книги по рефакторингу SQL. Что такое SQL, я знал хорошо. Однако слова *рефакторинг* я прежде не слышал. Я поискал это слово в Google. В знаменитой комедии Мольера богатый, но малообразованный человек был поражен, когда выяснил, что он всю жизнь говорил прозой. Подобно господину Журдену, я обнаружил, что годами занимался рефакторингом кода SQL, сам того не зная – анализ производительности для моих заказчиков естественным образом приводил к усовершенствованию кода посредством внесения небольших последовательных изменений, не менявших поведение программы. Одно дело пытаться спроектировать базу данных в меру своих возможностей и спланировать архитектуру и программы, которые обеспечат эффективный доступ к базе данных. Совсем другое дело выжать максимальную производительность из систем, которые вовсе не обязательно были удачно спроектированы с самого начала или бесконтрольно росли из года в год, но работа которых является жизненно важной. И меня привлекла идея рассказа об SQL с собственной профессиональной точки зрения.

Последнее, что хочется делать после окончания работы над книгой, – это приступать к написанию новой книги. Но идея меня слишком заинтересовала. Я обсудил ее с несколькими друзьями, среди которых был один из самых авторитетных специалистов по SQL, кого я знал. Этот друг впал в негодование от моих слов, но на сей раз я с ним не согласился. Действительно, идею, которую первым популяризировал Мартин Фаулер[1] – усовершенствование кода с помощью маленьких локализованных изменений, – можно воспринимать как причуду – чушь, которой заполняют отчеты корпоративные консультанты, только что получившие диплом. Но что касается меня, действительная важность

[1] М. Фаулер. «Рефакторинг: улучшение существующего кода». – Пер. с англ. – СПб: Символ-Плюс, 2002.

рефакторинга заключается в том, что первоначальный код уже нельзя считать хорошим, и в признании того, что множество посредственных систем может после небольших усилий выполнять свои задачи значительно лучше. Рефакторинг является также подтверждением того, что причиной падения производительности до не удовлетворяющего нас уровня являемся мы сами, а не гримасы судьбы. И для корпоративного мира это некоторое откровение.

Я видел множество сайтов, администраторы которых придавали производительности слишком большое значение, для них такие проблемы были ударом судьбы и они возлагали последнюю надежду на настройку. Если усилия администраторов баз данных и системных администраторов оказывались безуспешными, единственным остающимся для них выходом была закупка более мощной техники. Я читал слишком много аудиторских отчетов самозваных экспертов по базам данных, которые после переформатирования вывода системных утилит делали вывод, что нужно увеличить несколько параметров и добавить еще памяти. Ради справедливости замечу, что в некоторых таких отчетах упоминалась «необходимость настроить» пару ужасных запросов, но в лучшем случае в приложение к отчету был вставлен план действий.

Я годами не касался параметров базы данных (технические службы моих заказчиков обычно достаточно компетентны). Но мне удалось усовершенствовать множество программ, и я старался как можно больше сотрудничать с разработчиками, а не запираться в кабинете. В большинстве случаев я встречал людей, которые горели желанием учиться и узнавать, которых достаточно было немного подбодрить и подтолкнуть в нужном направлении, которые получали удовольствие от усовершенствования своих навыков работы с SQL и которые вскоре начинали ставить для себя задачи повышения производительности.

Когда через некоторое время из моей головы выветрились воспоминания о трудностях, связанных с написанием предыдущей книги, я приступил к новой с намерением подробно изложить идеи, которые я обычно пытаюсь передать разработчикам. Вопросы доступа к базам данных, вероятно, являются одной из областей с наибольшим простором для усовершенствования кода. Моей задачей при написании этой книги было не предложение готовых рецептов, а концепция, в соответствии с которой далеко не идеальные приложения SQL можно было бы усовершенствовать, не переписывая их с нуля (несмотря на периодически возникающее сильное искушение).

Почему нужно прибегать к рефакторингу?

У большинства приложений рано или поздно снижается производительность. В лучших случаях успех некоторых старых удачных приложений приводил к тому, что им приходилось обрабатывать такие объемы данных, для которых они не были исходно предназначены, поэтому старым программам нужно было продлить жизнь до внедрения новой

программы. В худших случаях тесты производительности до введения приложения в эксплуатацию показывали полное несоответствие требованиям к системе. В промежуточных случаях при росте объемов данных, добавлении новой функциональности, обновлении программного обеспечения или изменениях конфигурации обнаруживались дефекты, которые до определенного момента были незаметны, а возврат к прежнему состоянию не всегда мог исправить ситуацию. Все эти проблемы объединяют чрезвычайно сжатые сроки для увеличения производительности и постоянное давление на разработчиков.

Первые действия по разрешению проблемы обычно выполняют системные инженеры и администраторы баз данных, которых просят «поколдовать» над параметрами. Если не обнаруживается какая-то особо серьезная ошибка (такое случается), настройки операционной системы и базы данных часто дают только незначительный прирост производительности.

После этого традиционным следующим шагом является увеличение мощности оборудования. Это очень дорогой вариант, поскольку к цене оборудования, вероятно, добавится увеличение стоимости лицензий на использование программного обеспечения. Это приведет к прерыванию бизнес-процесса, потребуется планирование. Что самое печальное, нет никакой реальной гарантии рентабельности этих вложений. Случается, что массированное обновление оборудования не оправдывало надежд. Ходят ужасные истории о том, как после подобных обновлений производительность падала еще больше. Бывает, что добавление процессоров только увеличивает конкуренцию между процессами.

Концепция рефакторинга подразумевает необходимый промежуточный этап между настройкой и приобретением нового оборудования. Вышеупомянутая конструктивная книга Мартина Фаулера фокусируется на объектных технологиях. Но контекст приложений баз данных значительно отличается от контекста прикладных программ, написанных на объектно-ориентированных или процедурных языках, эти различия вносят значительные отличия в рефакторинг. Например:

Изменения, кажущиеся малыми, не всегда оказываются таковыми

Благодаря декларативной природе языка **SQL, незначительная модификация** кода часто может принести значительные изменения в то, как **SQL обрабатывает данные, что приводит к серьезным изменениям** производительности – как в лучшую, так и в худшую сторону.

Проверка правильности изменений может быть затруднена

Сравнительно несложно убедиться, что значение, возвращаемое функцией, одинаково во всех случаях до и после изменения кода. Значительно сложнее проверить, остается ли прежним содержимое большой таблицы после изменения главного оператора обновления.

Контекст часто оказывается критичным

Приложения баз данных могут годами работать удовлетворительно без заметных проблем. Часто случается, что объем данных или нагрузка переходят некий порог либо обновление программного обеспечения изменяет работу оптимизатора, после чего производительность внезапно становится неудовлетворительной. Работы по улучшению производительности баз данных обычно происходят в условиях кризиса.

В результате рефакторинг приложений баз данных происходит на сложном фоне, но в то же время такая попытка может быть (и часто бывает) очень успешной.

Рефакторинг доступа к базе данных

Специалисты по базам данных давно знают, что наиболее эффективный способ повышения производительности после проверки индексов – пересмотреть шаблоны доступа к базе данных. Несмотря на явную декларативную природу SQL, этот язык печально известен склонностью к колоссальным различиям времени выполнения по-разному написанных функционально идентичных операторов.

Однако рефакторинг доступа к базе данных представляет собой нечто большее, чем единичное изменение проблемных запросов, хотя большинство людей на этом останавливается. Например, медленное, но безостановочное развитие языка SQL в течение многих лет иногда позволяет разработчикам писать эффективные операторы, заменяющие тот код, который раньше можно было реализовать только с помощью сложных процедур с множеством операторов, на одно-единственное выражение. Новые встроенные механизмы баз данных дают вам возможность думать по-иному, значительно эффективнее, чем в прошлом. Пересмотр старых программ в свете новых возможностей часто приводит к значительному увеличению производительности.

Это был бы действительно дивный новый мир, если бы вслед за рефакторингом было желание обновить старые приложения, использовав преимущества новых возможностей. Правильный подход к приложениям баз данных может творить чудеса с тем, что я тактично называю «не совсем оптимальным кодом».

Изменение части логики приложения может показаться противоречащим установленной цели минимальных изменений. На самом деле ваше понимание того, какой способ является деликатным и пошаговым, зависит от пройденного вами пути; когда вы в первый раз едете в неизвестное место, дорога всегда кажется значительно длиннее, чем когда вы возвращаетесь в знакомое место.

Что мы можем ожидать от рефакторинга?

Важно понимать, что есть два фактора, которые в основном определяют возможные результаты рефакторинга (в реальном мире факторы конфликтуют между собой):

- Во-первых, выгоды от рефакторинга напрямую связаны с исходным приложением: если качество кода низкое, есть сомнения, что приложение удастся эффективно улучшить. Если код был оптимальным, может не быть (несмотря на применение новых методов) возможности для рефакторинга, и на этом все закончится. Все происходит так же, как и с компаниями: только плохо управляемые фирмы можно эффективно реорганизовать.
- Во-вторых, если база данных спроектирована действительно плохо, рефакторинг вряд ли даст большой эффект. Небольшие улучшения редко приводят к удовлетворительным результатам. Рефакторинг является эволюционным процессом. Например, если в базах данных нет и следов исходного качественного проектирования, даже осмысленная эволюция не поможет приложению выжить.

Маловероятно, что великий римский поэт Гораций имел в виду рефакторинг, когда писал о золотой посредственности, но именно на посредственные приложения мы можем возлагать наибольшие надежды. С ними есть достаточный запас, поскольку слишком часто «первым способом, про который все согласятся, что он будет работать функционально, становится дизайн», как писал рецензент этой книги Рой Оуэнс.

Как устроена эта книга

В этой книге делается попытка дать реалистичный и честный обзор усовершенствования приложений со значительной долей SQL и определить рациональную концепцию для тактических маневров. Часто рефакторинг напоминает безумный поиск быстрых побед и эффектных усовершенствований, которые можно вписать в бюджет и сохранить голову на плечах. Во время общей паники особенно важно сохранять ясность мышления и подходить к делу методично. Давайте условимся, что чудеса – удел очень талантливых личностей, и они обычно занимаются более серьезными вещами, чем наши приложения (что бы вы о них ни думали). Но разумное и систематическое применение правильных принципов тем не менее может привести к впечатляющим результатам. Эта книга должна помочь вам выработать различные тактики, а также оценить возможности различных решений и риски.

Очень часто рефакторинг приложений SQL происходит в порядке, противоположном порядку разработки: вы начинаете с легких вещей и медленно идете назад, углубляясь все дальше и дальше, пока не доберетесь до места, где кроется проблема, или не исчерпаете тот лимит, который вы установите для себя. Я пытался следовать тому же порядку в этой книге, организованной следующим образом:

Глава 1. Оценка

Эту главу, посвященную оценке ситуации, можно расценивать как пролог. Рефакторинг обычно связывают со временем, когда ресурсы являются дефицитом и подходить к их выделению требуется со всей тщательностью. Здесь нет допуска для ошибок или для неправильного выбора объекта усовершенствования. В этой главе мы попытаемся оценить, во-первых, есть ли хоть какие-нибудь надежды на успешность рефакторинга, а во-вторых, понять, какие надежды можно считать разумными.

Следующие две главы посвящены мечте любого менеджера: быстрым победам. В этих главах рассматриваются изменения, которые будут сделаны в первую очередь на стороне базы данных, а не в прикладной программе. Иногда вы даже сможете применить некоторые из этих изменений к приложениям, доступа к кодам которых у вас нет.

Глава 2. Проверка работоспособности

Эта глава о моментах, которые нужно проверять поочередно, в частности о проверке индексов.

Глава 3. Пользовательские функции и представления

Здесь объясняется, как написанные разработчиками функции и активное использование представлений иногда может затруднить функционирование приложений и как вы можете попытаться минимизировать их влияние на производительность.

В следующих трех главах речь ведется о правильных изменениях, которые вы можете внести в приложение.

Глава 4. Концепция тестирования

В этой главе описана правильная концепция тестирования. При модификации кода важно обеспечить получение тех же результатов, что и до внесения изменений, поскольку любая модификация, даже незначительная, может привести к появлению ошибок; изменений абсолютно без всякого риска не бывает. Здесь мы обсудим тактики сравнения результатов исходной и модифицированной версий программ.

Глава 5. Рефакторинг операторов

Здесь подробно обсуждается правильный подход к написанию различных операторов SQL. Оптимизаторы переписывают недостаточно оптимальные операторы. Во всяком случае, именно для этого оптимизаторы и существуют. Но даже самый совершенный оптимизатор может только попытаться выжать максимум из существующей ситуации. Рассмотрим, как анализировать и переписывать операторы SQL, чтобы превратить оптимизатор в друга, а не во врага.

Глава 6. Рефакторинг задачи

Здесь содержится продолжение обсуждения, начатого в пятой главе, и объяснение того, как изменение эксплуатационного режима,

в частности избавление от построчной обработки, может поднять наше приложение на более высокий уровень. Чаще всего переписывание отдельных операторов дает лишь малую долю возможных улучшений. Дерзкие изменения, например объединение нескольких операторов или замена итеративных, процедурных операторов на быстрые операторы SQL, часто приводят к впечатляющим результатам. Для этого требуются хорошие навыки работы с языком SQL и соответствующее мышление, сильно отличающееся от мышления, подходящего для работы с традиционными процедурными и объектно-ориентированными языками. Рассмотрим несколько примеров.

Если на этом этапе вы по-прежнему не удовлетворены производительностью, вашей последней надеждой станет следующая глава.

Глава 7. Рефакторинг потоков и баз данных

В этой главе мы возвращаемся к базе данных и обсуждаем более фундаментальные изменения. Сначала я расскажу, как можно увеличить производительность, изменив потоки и введя параллелизм, и поговорю о таких вещах, как целостность данных, конкуренция и блокировка, которые вы должны принимать во внимание при введении параллельных процессов. Затем я расскажу об изменениях, которые вы иногда можете внести, физически и логически, в структуру баз данных – как последний шанс попытаться получить дополнительный рост производительности.

И в заключение.

Глава 8. Как это работает: рефакторинг на практике

Эта глава представляет собой резюме всей книги в виде расширенной технологической карты. Здесь я опишу, со ссылками на предыдущие главы, о чем приходится думать и что нужно делать для разрешения проблем с производительностью приложений баз данных. Для меня это был трудный опыт, поскольку иногда эксперимент предлагает кратчайший путь, который на самом деле не является осознанным результатом точного логического анализа. Но я надеюсь, что эта глава послужит вам полезным источником информации.

Приложение A «Сценарии и примеры программ», и приложение B «Инструменты»

Описывают сценарии, примеры программ и инструменты, которые можно загрузить со страницы сайта O'Reilly, посвященной этой книге: *http://www.oreilly.com/catalog/9780596514976*.

Аудитория

Эта книга написана для профессионалов в области информационных технологий, разработчиков, менеджеров проектов, служб поддержки, администраторов баз данных и специалистов по настройке, которым приходится принимать участие в операциях по спасению приложений со значительным объемом кода управления базами данных.

Допущения, сделанные в этой книге

Предполагается, что читатель этой книги имеет достаточно серьезные практические знания языка SQL и, конечно, прилично знает хотя бы один язык программирования.

Используемые в книге обозначения

В этой книге используются следующие типографские обозначения:

Курсив

Указывает на выделение, новые термины, адреса URL, имена и расширения файлов.

`Моноширинный шрифт`

Указывает на компьютерный код в широком смысле, в том числе команды, параметры, переменные, атрибуты, ключи, запросы, функции, методы, типы, классы, модули, свойства, параметры, значения, объекты, события, обработчики событий, теги XML и XHTML, макросы и ключевые слова. Он также указывает на такие идентификаторы, как имена таблиц столбцов, и используется для примеров кода и вывода команд.

`Жирный моноширинный шрифт`

Выделения в примерах кода.

`Курсивный моноширинный шрифт`

Указывает текст, который нужно заменить на конкретные значения.

Использование примеров кода

Задачей этой книги является улучшение вашей работы. Вообще вы можете использовать код, приведенный в этой книге, в ваших программах и документации. Вам не нужно связываться с нами для получения разрешения, если вы копируете незначительную часть кода. Например, написание программы, в которой используется несколько кусков кода из этой книги, разрешения не требует. Продажа или распространение компакт-диска с примерами из книг издательства O'Reilly требует разрешения. Ответы на вопросы путем цитирования этой книги и примеров кода разрешения не требует. Объединение значительных объемов кода из этой книги в документацию по вашему продукту требует разрешения.

Мы будем благодарны за ссылку, хотя и не требуем ее. В ссылку обычно включается заголовок, автор, издатель и код ISBN. Например: «Refactoring SQL Applications by Stéphane Faroult with Pascal L'Hermite. Copyright 2008 Stéphane Faroult and Pascal L'Hermite, 978-0-596-51497-6».

Если вы предполагаете, что использование вами кода примеров может потребовать вышеуказанных разрешений, свяжитесь с нами по адресу *permissions@oreilly.com*.

Комментарии и вопросы

Комментарии и вопросы, связанные с этой книгой, направляйте издателю по адресу:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (в США или Канаде)

707-829-0515 (международный или локальный)

707-829-0104 (факс)

У нас есть веб-страница, посвященная этой книге, где мы публикуем опечатки, примеры и любую дополнительную информацию. Адрес этой страницы:

http://www.oreilly.com/catalog/9780596514976

Для комментариев к этой книге и технических вопросов направляйте электронные письма по адресу:

bookquestions@oreilly.com

Дополнительная информация о наших книгах, конференциях, ресурсных центрах и Сети O'Reilly на нашем веб-сайте по адресу:

http://www.oreilly.com

Safari® Books Online

Когда вы видите знак Safari® Books Online на обложке вашей любимой технической книги, это означает, что книгу можно приобрести в O'Reilly Network Safari Bookshelf.

Safari предлагает решение лучшее, чем электронные книги. Это виртуальная библиотека, которая позволяет вам с легкостью осуществлять поиск лучших технических книг, копировать примеры кода, загружать главы и быстро находить ответы, когда вам нужна наиболее точная и свежая информация. Посетите беплатный сайт *http://safari.oreilly.com*.

Благодарности

Любая книга представляет собой результат работы значительно большего количества людей, чем указано на обложке. В первую очередь я хочу поблагодарить Паскаля Лерми (Pascal L'Hermite), чьи знания Oracle и SQL Server оказались чрезвычайно ценными в процессе написания этой книги. В технических книгах написание текста является только видимой стороной дела. Настройка среды тестирования, разработка примеров программ, перенос их на различные продукты, а также апробация идей, которые в результате оказываются тупиковыми, – все

это задачи, отнимающие уйму времени. Многое из проделанной работы проявляется в законченной книге только в виде случайных ссылок. Без помощи Паскаля эту книгу пришлось бы писать значительно дольше.

Для каждого проекта необходим координатор, и Мэри Треселер, мой редактор, выступила в этой роли со стороны издательства O'Reilly. Мэри выбрала прекрасную команду рецензентов, некоторые из которых сами являются авторами. Первым среди них был Брэнд Хант, редактор-консультант этой книги. Я хочу сердечно поблагодарить Брэнда, который помог мне придать этой книге законченный вид, а также Дуэйна Кинга, в частности за его внимание как к тексту, так и к примерам кода. Дэвид Нур, Рой Оуэнс и Майкл Блаха (Blaha) тоже принесли большую пользу. Я также хочу поблагодарить двух старых друзей-экспертов, Филиппа Бертолино и Сирила Танкаппана, которые также сделали тщательные критические обзоры моих первых черновиков.

Кроме исправления некоторых ошибок, все эти рецензенты внесли замечания или уточнения, которые нашли свое место в конечном продукте и сделали его лучше.

1

Оценка

Из пепла несчастий растут розы успеха!

Ричард М. Шерман (р. 1928) и Роберт Б. Шерман (р. 1925),
из «Chitty Chitty Bang Bang»
по мотивам Яна Флеминга (1908–1964)

Когда возникает вопрос рефакторинга кода, вы можете быть уверены, что либо возникла проблема, либо предполагается, что она проявится в ближайшее время. Вы знаете, что вам нужно улучшить в плане функциональности, но вам нужно понять природу проблемы.

Работа с любым компьютерным приложением всегда сводится к загрузке процессора, использованию памяти и операциям ввода-вывода на диск, сетевой ресурс или другое устройство. Когда вопрос касается производительности, первое, что нужно диагностировать, – не достиг ли один из этих трех ресурсов проблемного уровня, поскольку это поможет вам выяснить, что нужно улучшить и как это сделать.

Приложения баз данных отличаются тем, что вы можете попытаться усовершенствовать использование ресурсов на разных уровнях. Если вы действительно хотите увеличить производительность приложения SQL, вы можете остановиться на том, что кажется очевидным узким местом, и попытаться разрешить проблему в этом месте (например, «давайте добавим памяти для СУБД» или «давайте использовать более быстрые диски»).

Такой подход считался разумным большую часть восьмидесятых годов, когда SQL стал стандартным языком доступа к корпоративным данным. И сегодня многие полагают, что лучший, если не единственный, способ увеличить производительность баз данных – это либо поменять значения нескольких, желательно малоизвестных параметров базы данных, либо обновить оборудование. На более высоком уровне вы можете отслеживать полный перебор больших таблиц и добавлять

индексы, чтобы устранить проблемы. На еще более высоком уровне вы можете попытаться настроить операторы SQL и переписать их, чтобы оптимизировать план их выполнения. Можно пересмотреть и весь процесс.

Эта книга фокусируется на трех последних вариантах и исследует различные способы увеличения производительности, которые иногда оказываются эффективными и не зависят от настройки параметров базы данных или обновления оборудования.

Прежде чем пытаться определить, как вы можете с уверенностью оценить, будет ли какая-нибудь польза от рефакторинга конкретного фрагмента кода, давайте возьмем простой, но не слишком тривиальный пример, который проиллюстрирует разницу между рефакторингом и настройкой. Следующий пример является искусственным, но он навеян случаями из реальной практики.

> **Примечание**
>
> Тесты в этой книге были выполнены на различных машинах, обычно после установки «с нуля», и хотя для генерирования данных на всех трех базах (MySQL, Oracle и SQL Server) использовалась одна и та же программа, что было удобнее, чем переносить данные, использование случайных чисел привело к появлению идентичных глобальных объемов, но различных наборов данных с сильно отличающимся количеством обрабатываемых строк. По этой причине сравнение времени исполнения на различных продуктах бессмысленно. А вот относительное различие между программами для одного продукта смысл имеет, также как и общие схемы.

Простой пример

Предположим, у вас есть несколько «областей» (это название условное), к которым подключены «счета», и с этими счетами связаны суммы в различных валютах. Каждая сумма соответствует транзакции. Вы хотите проверить для одной области, не превышают ли какие-нибудь суммы заданный предел для транзакций, произошедших за последние 30 дней до указанной даты. Этот предел зависит от валюты и он определен не для всех валют. Если предел определен и сумма превышает предел для данной валюты, вы должны зарегистрировать идентификатор транзакции и сумму, сконвертированную в локальную валюту по данным на конкретную дату валютирования.

Я сгенерировал для этого примера таблицу транзакций из двух миллионов строк и использовал некий абстрактный код Java™/JDBC, чтобы показать, как различные способы написания кода могут повлиять на производительность. Код на языке Java является упрощенным, так что каждый, кто знает какой-нибудь язык программирования, сможет понять суть.

Предположим, базовая часть приложения выглядит следующим образом (в арифметике дат в нижеприведенном коде используется синтаксис MySQL). Эту программу я назвал *FirstExample.java*:

```
try {
    long txid;
    long accountid;
    float amount;
    String curr;
    float conv_amount;

    PreparedStatement st1 = con.prepareStatement("select accountid"
                          + " from area_accounts"
                          + " where areaid = ?");
    ResultSet rs1;
    PreparedStatement st2 = con.prepareStatement("select txid,amount,curr"
                          + " from transactions"
                          + " where accountid=?"
                          + " and txdate >= date_sub(?, interval 30 day)"
                          + " order by txdate");
    ResultSet rs2 = null;
    PreparedStatement st3 = con.prepareStatement("insert into check_log(txid,"
                          + " conv_amount)"
                          + " values(?,?)");

    st1.setInt(1, areaid);
    rs1 = st1.executeQuery();
    while (rs1.next()) {
        accountid = rs1.getLong(1);
        st2.setLong(1, accountid);
        st2.setDate(2, somedate);
        rs2 = st2.executeQuery();
        while (rs2.next()) {
            txid = rs2.getLong(1);
            amount = rs2.getFloat(2);
            curr = rs2.getString(3);
            if (AboveThreshold(amount, curr)) {
                // Конвертация
               conv_amount = Convert(amount, curr, valuationdate);
               st3.setLong(1, txid);
               st3.setFloat(2, conv_amount);
               dummy = st3.executeUpdate();
            }
        }
    }
    rs1.close();
    st1.close();
```

```
44     if (rs2 != null) {
45         rs2.close();
46     }
47     st2.close();
48     st3.close();
49 } catch(SQLException ex){
50     System.err.println("==> SQLException: ");
51     while (ex != null) {
52         System.out.println("Message: " + ex.getMessage ());
53         System.out.println("SQLState: " + ex.getSQLState ());
54         System.out.println("ErrorCode: " + ex.getErrorCode ());
55         ex = ex.getNextException();
56         System.out.println("");
57     }
58 }
```

Этот фрагмент напоминает тот код, который используется в реальных приложениях. Небольшое пояснение по JDBC:

- У нас есть три оператора SQL (строки 8, 12 и 18), которые являются подготовленными операторами. Использование подготовленных операторов – это правильный способ работы с JDBC, когда мы многократно исполняем идентичные операторы, отличающиеся лишь некоторыми значениями при каждом вызове (о подготовленных операторах я буду говорить подробнее во второй главе). Эти значения представлены знаками вопроса, вместо которых при каждом вызове будут подставлены конкретные величины с помощью функций `setInt()` в строке 22 или `setLong()` и `setDate()` в строках 26 и 27.
- В строке 22 я установил значение (areaid), которое я определил и инициализировал в той части кода, которая здесь не показана.
- После того как шаблоны подстановки привязаны к реальным значениям, я могу вызвать функцию `executeQuery()`, как в строке 23, если оператором SQL является `select`, или функцию `executeUpdate()`, как в строке 38, если используется любой другой оператор. Для операторов `select` я получаю результирующий набор, из которого могу в цикле извлечь все значения, например как в строках 30, 31 и 32.

В коде есть два вызова служебных функций: `AboveThreshold()` в строке 33 проверяет, не превышает ли сумма предел для данной валюты, а `Convert()` в строке 35 преобразует сумму, превышающую предел, в валюту отчета. Вот код этих двух функций:

```
private static boolean AboveThreshold(float amount,
                                       String iso) throws Exception {
PreparedStatement thresholdstmt = con.prepareStatement("select threshold"
                                      + " from thresholds"
                                      + " where iso=?");
    ResultSet         rs;
    boolean           returnval = false;

    thresholdstmt.setString(1, iso);
```

```
        rs = thresholdstmt.executeQuery();
        if (rs.next()) {
                if (amount >= rs.getFloat(1)){
                    returnval = true;
                } else {
                    returnval = false;
                }
        } else { // не найдено - нет проблемы
            returnval = false;
        }
        if (rs != null) {
            rs.close();
        }
        thresholdstmt.close();
        return returnval;
    }

    private static float Convert(float amount,
                                 String iso,
                                 Date valuationdate) throws Exception {
        PreparedStatement conversionstmt = con.prepareStatement("select ? *
    rate"
                                            + " from currency_rates"
                                            + " where iso = ?"
                                            + " and rate_date = ?");
        ResultSetrs;
        floatval = (float)0.0;

        conversionstmt.setFloat(1, amount);
        conversionstmt.setString(2, iso);
        conversionstmt.setDate(3, valuationdate);
        rs = conversionstmt.executeQuery();
        if (rs.next()) {
            val = rs.getFloat(1);
        }
        if (rs != null) {
            rs.close();
        }
        conversionstmt.close();
        return val;
    }
```

У всех таблиц определены первичные ключи. Когда я запустил эту программу с тестовыми данными, проверяя приблизительно одну седьмую набора из двух миллионов строк и регистрируя в конечном итоге очень мало строк, время работы программы составило приблизительно 11 минут при работе с MySQL[1] на моем тестовом компьютере.

[1] MySQL 5.1.

После небольшой модификации кода SQL, учитывающей различные способы представления месяца, предшествующего данной дате в разных диалектах языка, я запустил эту же программу с тем же объемом данных на SQL Server и Oracle[1].

Работа программы заняла около пяти с половиной минут на SQL Server и чуть меньше трех минут на Oracle. Для сравнения в табл. 1.1 приведено время работы программ с каждой системой управления базами данных (СУБД). Как видите, во всех трех случаях программа выполняла свою задачу слишком долго. Что мы можем сделать, прежде чем решиться на покупку более быстродействующего оборудования?

Таблица 1.1. Исходная ситуация для программы SimpleExample.java

СУБД	Исходный результат
MySQL	11 минут
Oracle	3 минуты
SQL Server	5,5 минут

Настройка SQL, традиционный способ

Обычным подходом на этом этапе является передача программы местному специалисту по настройке (обычно администратору баз данных). Администратор MySQL, вероятно, снова запустит программу в среде тестирования, после того как тестовая база данных будет запущена со следующими двумя параметрами:

```
--log-slow-queries
--log-queries-not-using-indexes
```

Полученный файл журнала покажет много повторяющихся вызовов главного виновника, каждый из которых выполнялся от трех до четырех секунд, а именно, следующего запроса:

```
select txid,amount,curr
from transactions
where accountid=?
  and txdate >= date_sub(?, interval 30 day)
order by txdate
```

Изучая базу данных `information_schema` (или используя такой инструмент, как *phpMyAdmin*), мы быстро обнаружим, что таблица транзакций имеет единственный индекс – индекс по первичному ключу по столбцу `txid`, который в данном случае неприменим, поскольку по этому столбцу у нас нет критерия выборки. В результате сервер базы данных ничего не может сделать, кроме как выполнять в цикле перебор записей

[1] SQL Server 2005 и Oracle 11.

большой таблицы от начала до конца. Решение очевидно: создать дополнительный индекс по столбцу `accountid` и запустить процесс снова. Каков же результат? Теперь программа работает немного меньше четырех минут, значит, производительность возросла в 3,1 раза.

Для нашего администратора MySQL, это, скорее всего, конец истории. Однако для его коллег, работающих с Oracle и SQL Server, все не так просто. Не менее опытный, чем администратор MySQL, администратор Oracle активизировал бы магическое оружие настройки Oracle, известное среди посвященных как *event 10046 level 8* (или использовал бы с тем же эффектом «advisor»), и получил бы файл трассировки, ясно показывающий, как тратилось время. Из такого файла трассировки вы можете определить, сколько раз выполнялись операторы, сколько процессорного времени они использовали, фактическую продолжительность работы и другую важную информацию, например количество логических считываний (которые показаны в файле трассировки как запрос и текущая запись), то есть количество блоков данных, к которым был осуществлен доступ для обработки запроса, и периоды ожидания, которые объясняют, по крайней мере частично, разницу между затраченным процессорным временем и фактическим временем работы программы:

```
********************************************************************************

SQL ID : 1nup7kcbvt072
select txid,amount,curr
from
  transactions where accountid=:1 and txdate >= to_date(:2, 'DD-MON-YYYY') -
      30 order by txdate


call       count  cpu    elapsed  disk    query current     rows
------- ------ -------- ---------- ---------- ---------- --------
Parse          1    0.00    0.00     0          0          0        0
Execute      252    0.00    0.01     0          0          0        0
Fetch      11903   32.21   32.16     0  2163420          0   117676
------- ------ -------- ---------- ---------- ---------- --------
total      12156   32.22   32.18     0  2163420          0   117676
Misses in library cache during parse: 1
Misses in library cache during execute: 1
Optimizer mode: ALL_ROWS
Parsing user id: 88

Rows Row Source Operation
------- ---------------------------------------------------
    495 SORT ORDER BY (cr=8585 [...] card=466)
    495 TABLE ACCESS FULL TRANSACTIONS (cr=8585 [...] card=466)

Elapsed times include waiting on following events:
  Event waited on                          Times  Max. Wait  Total Waited
   ---------------------------------- Waited ---------- ------------
  SQL*Net message to client                 11903       0.00          0.02
```

```
  SQL*Net message from client          11903       0.00          2.30
********************************************************


SQL ID : gx2cn564cdsds
select threshold
from
  thresholds where iso=:1


call         count    cpu  elapsed   disk   query   current    rows
------- ------ -------- ---------- ---------- ---------- ----------
Parse       117674   2.68      2.63       0       0          0        0
Execute     117674   5.13      5.10       0       0          0        0
Fetch       117674   4.00      3.87       0  232504          0   114830
------- ------ -------- ---------- ---------- ---------- ----------
total       353022  11.82     11.61       0  232504          0   114830

Misses in library cache during parse: 1
Misses in library cache during execute: 1
Optimizer mode: ALL_ROWS
Parsing user id: 88

Rows Row Source Operation
------- ---------------------------------------------------
   1 TABLE ACCESS BY INDEX ROWID THRESHOLDS (cr=2 [...] card=1)
   1 INDEX UNIQUE SCAN SYS_C009785 (cr=1 [...] card=1)(object id 71355)


Elapsed times include waiting on following events:
  Event waited on                      Times   Max. Wait  Total Waited
  ---------------------------------- Waited ---------- --------------
  SQL*Net message to client          117675      0.00           0.30
  SQL*Net message from client        117675      0.14          25.04
********************************************************************************
```

Видя `TABLE ACCESS FULL TRANSACTION` **в плане выполнения самого медленного запроса (особенно, когда он исполнен 252 раза), администратор Oracle отреагирует так же, как и администратор MySQL. В Oracle тот же самый индекс по столбцу** `accountid` **увеличил производительность в 1,2 раза, уменьшив время выполнения приблизительно до минуты и 20 секунд.**

Администратор SQL Server может использовать SQL Profiler или запустить следующий скрипт:

```
select a.*
from (select execution_count,
    total_elapsed_time,
    total_logical_reads,
    substring(st.text, (qs.statement_start_offset/2) + 1,
        ((case statement_end_offset
```

```
            when -1 then datalength(st.text)
            else qs.statement_end_offset
        end
        - qs.statement_start_offset)/2) + 1) as statement_text
    from sys.dm_exec_query_stats as qs
        cross apply sys.dm_exec_sql_text(qs.sql_handle) as st) a
where a. statement_text not like '%select a.*%'
order by a.creation_time
```

и в результате получить:

```
execution_count total_elapsed_time total_logical_reads statement_text
            228           98590420       3062040 select txid, ...
         212270           22156494        849080 select threshold ...
              1            2135214         13430 ...
...
```

Администратор SQL Server заметит, что самыми долгими запросами являются, несомненно, операторы `select` в транзакциях, и сделает то же заключение, что и остальные: в таблице транзакций недостает индекса. К сожалению, результат исправления разочаровывает. Создание индекса по столбцу `accountid` увеличивает производительность в скромные 1,3 раза, всего до четырех минут с небольшим, чего явно недостаточно, чтобы этим восторгаться. В табл. 1.2 показано увеличение скорости, полученное в результате применения нового индекса, для каждой СУБД.

Таблица 1.2. Коэффициент увеличения скорости после добавления индекса к транзакциям

СУБД	Увеличение скорости
MySQL	3,1
Oracle	1,2
SQL Server	1,3

Настройка путем индексирования очень популярна среди разработчиков, поскольку в код не нужно вносить никаких изменений. Этот прием точно так же популярен и среди администраторов баз данных, которые не часто видят код и знают, что правильное индексирование со значительно большей вероятностью приведет к заметным результатам, чем изменение малоизученных параметров. Но я хочу повести вас дальше и показать, чего вы можете добиться ценой небольших усилий.

Припудривание кода

Я модифицировал код программы *FirstExample.java* и создал на ее основе программу *SecondExample.java*. В исходный код я внес два усовершенствования. Возможно, вам неясно, какова цель оператора `order by` в главном запросе:

```
select txid,amount,curr
from transactions
where accountid=?
  and txdate >= date_sub(?, interval 30 day)
order by txdate
```

Мы всего лишь берем данные из одной таблицы для заполнения другой. Если нам нужен отсортированный результат, мы добавим оператор `order by` в запрос, который получает данные из итоговой таблицы для предоставления конечному пользователю. На нынешнем, промежуточном, этапе оператор `order by` не нужен; это очень распространенная ошибка, заметить ее можно только наметанным взглядом.

Второе усовершенствование связано с частой вставкой данных со средней скоростью (я получил в общей сложности несколько сотен строк в таблице журнала). По умолчанию для соединения JDBC применяется режим автоматической фиксации изменений. В данном случае это означает, что после каждой вставки будет срабатывать принудительный оператор `commit` и каждое изменение будет синхронно записываться на диск. Такая запись в постоянное хранилище гарантирует, что изменения не будут потеряны даже в случае сбоя системы всего через миллисекунду после их внесения, а без фиксации изменения хранятся только в памяти и поэтому могут быть потеряны. В данном случае это излишняя предосторожность. Если произойдет сбой системы, можно просто запустить программу снова, особенно если удастся сделать ее быстрой – маловероятно, что сбои будут происходить очень часто. Поэтому я вставил в начале программы оператор, отключающий автоматическую фиксацию, а в конце – оператор, принудительно фиксирующий изменения:

```
// Отключение автоматической фиксации
con.setAutoCommit(false);
```

и

```
con.commit();
```

Эти два очень маленьких изменения дали некоторое ускорение работы программы: их совместный эффект на версии для MySQL увеличил скорость примерно на 10%. Однако с Oracle и SQL Server мы не получили никаких заметных результатов (табл. 1.3).

Таблица 1.3. Коэффициент увеличения скорости после добавления индекса к транзакциям, очистки кода и отключения автоматической фиксации изменений

СУБД	Увеличение скорости
MySQL	3,2
Oracle	1,2
SQL Server	1,3

Настройка SQL заново

Когда один индекс не дает тех результатов, на которые мы рассчитываем, иногда улучшить производительность может другой индекс. Прежде всего, почему индекс создан только по одному столбцу `accountid`? По существу, индекс представляет собой отсортированный по древовидной схеме список значений ключа, связанных с физическими адресами строк, соответствующих значениям этих ключей, – точно так же, как предметный указатель этой книги представляет собой отсортированный список ключевых слов, связанных с номерами страниц. Если мы осуществляем поиск по значениям из двух столбцов, а проиндексирован только один из них, нам придется зайти на все строки, соответствующие ключу, который мы ищем, и отбросить подмножество из этих строк, которое не соответствует ключу для другого столбца. Если мы индексируем оба столбца, то сразу получаем то, что нам нужно.

Мы можем создать индекс по (`accountid`, `txdate`), поскольку дата транзакции является вторым критерием запроса. Создав составной индекс по обоим столбцам, можно быть уверенным, что SQL-машина сумеет осуществить эффективный поиск (известный как сканирование диапазона) по индексу. Если с моими тестовыми данными индекс по одному столбцу увеличил производительность на MySQL в 3,1 раза, то с индексом по двум столбцам увеличение скорости составило 3,4 раза, так что теперь программа работает около трех с половиной минут. Однако Oracle и SQL Server даже с индексом по двум столбцам не дали ускорения по сравнению с индексом по одному столбцу (табл. 1.4).

Таблица 1.4. Коэффициент увеличения скорости после изменения индекса

СУБД	Увеличение скорости
MySQL	3,4
Oracle	1,2
SQL Server	1,3

То, что мы делали до сих пор, – сочетание некоторых небольших усовершенствований операторов SQL, разумное использование таких возможностей, как управление транзакциями и правильная стратегия индексирования – называется «традиционным подходом» к настройке. Теперь используем подход более радикальный и рассмотрим одну за другой две точки зрения. Давайте сначала обсудим, как организована программа.

Рефакторинг, первая точка зрения

Как и у многих процессов, которые встречаются в реальной жизни, примечательной особенностью моего примера являются вложенные циклы. Глубоко внутри этих циклов мы обнаружим служебную функцию

AboveThreshold(), которая выполняется для каждой возвращаемой строки. Я уже упоминал, что таблица транзакций содержит два миллиона строк и что одна седьмая всех строк относится к рассматриваемой «области». В результате вызов функции AboveThreshold() происходит многократно. Когда вызов функции происходит часто, даже небольшое ускорение ее работы дает заметный результат. Например, предположим, что нам удалось уменьшить продолжительность вызова с пяти миллисекунд до четырех, тогда при 200 000 вызовах мы экономим в общей сложности 200 секунд, то есть свыше трех минут. Если ожидается 20-кратное увеличение объемов в ближайшие месяцы, сэкономленное время увеличится до одного часа.

Хорошим способом сократить время работы программы является уменьшение количества обращений к базе данных. Хотя многие разработчики рассматривают базу данных как мгновенно доступный ресурс, запрос к ней требует некоторого времени. На самом деле запрос к базе данных – это дорогая операция. Вы должны связаться с сервером, что вызывает некоторые сетевые задержки, особенно если ваша программа запущена не на сервере. Кроме того, то, что вы посылаете на сервер, является не непосредственно исполняемым машинным кодом, а оператором SQL. Сервер должен проанализировать его и перевести в реальный машинный код. Возможно, сервер уже исполнял подобный оператор, тогда вычисления «подписи» оператора может быть достаточно, чтобы сервер использовал оператор из кэша. Если оператор встречается впервые, серверу потребуется определить правильный план исполнения и запустить рекурсивные запросы к словарю данных. Если же оператор уже выполнялся, но после заполнения кэша был затерт другими операторами, то случай аналогичен тому, как если бы он выполнялся впервые. Затем команде SQL нужно исполнить и вернуть по сети данные, которые либо хранились в кэше сервера базы данных, либо были считаны с диска. Другими словами, обращение к базе данных преобразуется в последовательность операций, каждая из которых необязательно является длительной, но ведет к расходованию ресурсов – пропускной способности сети, памяти, процессора и операций ввода-вывода. Конкуренция между сеансами может добавить время на ожидание доступа к неразделяемым ресурсам, к которым происходят параллельные обращения.

Давайте вернемся к функции AboveThreshold(). В ней мы проверяем пределы, связанные с валютой. Валюты имеют особенность – хоть в мире есть около 170 валют, даже большие финансовые институты имеют дело только с небольшим количеством – локальной валютой, валютами главных торговых партнеров страны и несколькими неизбежными валютами, имеющими большой вес в мировой торговле: долларом США, евро и, возможно, японской йеной, английским фунтом стерлингов и некоторыми другими.

Когда я подготавливал данные, я основывался на распределении валют из выборки, взятой из приложения в большом банке в еврозоне, и вот (реальное) распределение, которое я использовал при генерировании данных для моей тестовой таблицы:

Код валюты	Название валюты	Доля
EUR	Евро	41,3
USD	Доллар США	24,3
JPY	Японская йена	13,6
GBP	Английский фунт стерлингов	11,1
CHF	Швейцарский франк	2,6
HKD	Гонконгский доллар	2,1
SEK	Шведская крона	1,1
AUD	Австралийский доллар	0,7
SGD	Сингапурский доллар	0,5

Общая доля главных валют равна 97,3%. Я добавил оставшиеся 2,7% случайно выбранными валютами из 170 записанных валют (включая главные валюты для этого конкретного банка).

В результате мы не только вызываем функцию `AboveThreshold()` сотни тысяч раз, но и сама эта функция вызывает те же строки из таблицы пределов. Вы можете подумать, что поскольку эти несколько строк будут, вероятно, храниться в кэше сервера базы данных, это не имеет большого значения. Но это на самом деле не так, и я продемонстрирую степень влияния этих неэффективных вызовов, переписав функцию более эффективным образом.

Я назвал новую версию программы *ThirdExample.java*. В ней я использовал для хранения данных некоторые специфичные коллекции языка Java – `HashMaps`. В этих коллекциях хранятся пары ключ-значение, а хеширование ключа позволяет получить индекс массива, сообщающий, куда пара должна идти. В других языках можно было бы использовать массивы. Но идея заключается в том, чтобы избегать запросов к базе данных, используя пространство памяти процесса как кэш. Когда какие-то данные запрашиваются первый раз, их получают из базы данных и сохраняют в коллекции, прежде чем вернуть значение вызывающей функции.

Следующий раз, когда я запрашиваю те же самые данные, я нахожу их в моем маленьком локальном кэше и возвращаю почти мгновенно. Два обстоятельства позволяют кэшировать данные:

- Поскольку эта программа не из реальной практики, и я знаю, что если буду повторно запрашивать значения предела для данной валюты, то буду каждый раз получать одно и то же значение: изменений между вызовами не будет.
- Я имею дело с небольшим количеством данных, поэтому не буду хранить гигабайты их в кэше. Требования к памяти – важный момент, который надо учитывать, если возможна ситуация с большим количеством конкурирующих сессий.

Поэтому я переписал две функции (наиболее критичной является `AboveThreshold()`**, но применить ту же самую логику к функции** `Convert()` **может оказаться полезным):**

```
// Используйте hashmaps для пределов и курсов обмена
private static HashMap thresholds = new HashMap();
private static HashMap rates = new HashMap();
private static Date previousdate = 0;

...

private static boolean AboveThreshold(float amount,
        String iso) throws Exception {
    float threshold;
    if (!thresholds.containsKey(iso)){
        PreparedStatement thresholdstmt =
                    con.prepareStatement("select threshold"
                    + " from thresholds"
                    + " where iso=?");

        ResultSet         rs;

        thresholdstmt.setString(1, iso);
        rs = thresholdstmt.executeQuery();
        if (rs.next()) {
            threshold = rs.getFloat(1);
            rs.close();
        } else {
            threshold = (float)-1;
        }
        thresholds.put(iso, new Float(threshold));
        thresholdstmt.close();
    } else {
        threshold = ((Float)thresholds.get(iso)).floatValue();
    }
    if (threshold == -1){
        return false;
    } else {
        return(amount >= threshold);
    }
}

private static float Convert(float amount,
        String iso,
        Date valuationdate) throws Exception {
    float rate;
    if ((valuationdate != previousdate)
       || (!rates.containsKey(iso))){
        PreparedStatement conversionstmt =
                con.prepareStatement("select rate"
                + " from currency_rates"
```

```
                + " where iso = ?"
                + " and rate_date = ?");
        ResultSet          rs;

        conversionstmt.setString(1, iso);
        conversionstmt.setDate(2, valuationdate);
        rs = conversionstmt.executeQuery();
        if (rs.next()) {
            rate = rs.getFloat(1);
            previousdate = valuationdate;
            rs.close();
        } else { // не нашли - должна быть проблема!
            rate = (float)1.0;
        }
        rates.put(iso, rate);
    conversionstmt.close();
    } else {
        rate = ((Float)rates.get(iso)).floatValue();
    }
    return(rate * amount);
}
```

После переписывания функций с учетом композитного индекса по двум столбцам (`accountid`, `txdate`) время выполнения программы резко уменьшилось: 30 секунд с MySQL, 10 секунд с Oracle и чуть меньше 9 секунд с SQL Server – улучшение с соответствующими коэффициентами 24, 16 и 38 по сравнению с исходной ситуацией (табл. 1.5).

Таблица 1.5. Коэффициент увеличения скорости с индексом по двум столбцам и переписанной функцией

СУБД	Увеличение скорости
MySQL	24
Oracle	16
SQL Server	38

О другом возможном усовершенствовании можно догадаться из журнала MySQL (так же, как из трассировочного файла Oracle и динамической таблицы `sys.dm_exec_query_stats` SQL Server). Речь идет о главном запросе:

```
select txid,amount,curr
from transactions
where accountid=?
  and txdate >= [выражение типа дата]
```

Он выполняется несколько сотен раз. Нет необходимости доказывать, что он работает значительно лучше, когда таблица правильно проиндексирована. Но значение для `accountid` – это не что иное, как результат

другого запроса. Нет необходимости делать запрос на сервер, получать значение accountid, передавать его в главный запрос и, наконец, выполнять главный запрос. Мы можем обойтись единственным запросом с запросом, «перетекающим» в значения accountid:

```
select txid,amount,curr
from transactions
where accountid in (select accountid
                    from area_accounts
                    where areaid = ?)
and txdate >= date_sub(?, interval 30 day)
```

Это только одно из усовершенствований, которые я сделал, чтобы получить версию *FourthExample.java*. В итоге вышел весьма разочаровывающий результат с Oracle (хотя и несколько лучший, чем в случае *ThirdExample.java*), но программа теперь выполняется с SQL Server за 7,5 секунд, а с MySQL за 20,5 секунд, соответственно в 44 и 34 раза быстрее, чем исходная версия (табл. 1.6). Однако с *FourthExample.java* появилось кое-что новое и интересное: со всеми продуктами скорость остается примерно одинаковой вне зависимости от того, существует или нет индекс по столбцу accountid в таблице transactions и построен ли индекс только по столбцу accountid или по обоим столбцам accountid и txdate.

Таблица 1.6. Коэффициент увеличения скорости с переписанным кодом SQL и переписанной функцией

СУБД	Увеличение скорости
MySQL	34
Oracle	16
SQL Server	44

Рефакторинг, вторая точка зрения

Предыдущее усовершенствование уже является изменением перспективы: вместо того, чтобы модифицировать код так, чтобы выполнять меньше операторов SQL, я начал заменять два оператора SQL на один. Я уже указывал на то, что циклы являются замечательной особенностью (и нередкой) моей тестовой программы. Большинство переменных программы используются для хранения данных, полученных с помощью предыдущего запроса, прежде чем они будут переданы в другой запрос: это тоже обычное явление во многих программах. Должно ли извлечение данных из одной таблицы для сравнения с данными из другой перед вставкой их в третью таблицу происходить в нашем коде? Теоретически, все операции должны происходить только на сервере, без многочисленных обменов между приложением и сервером базы данных. Мы можем написать хранимую процедуру для выполнения большей

части работы или даже всей работы на сервере либо просто написать единственный, заведомо не очень сложный оператор, выполняющий задачу. Более того, единственный оператор будет меньше зависеть от используемой СУБД, чем хранимая процедура:

```
try {
    PreparedStatement st = con.prepareStatement("insert into check_
log(txid,"
        + "conv_amount)"
        + "select x.txid,x.amount*y.rate"
        + " from(select a.txid,"
        + "             a.amount,"
        + "             a.curr"
        + "      from transactions a"
        + "      where a.accountid in"
        + "             (select accountid"
        + "              from area_accounts"
        + "              where areaid = ?)"
        + "       and a.txdate >= date_sub(?, interval 30 day)"
        + "       and exists (select 1"
        + "                   from thresholds c"
        + "                   where c.iso = a.curr"
        + "                     and a.amount >= c.threshold)) x,"
        + "     currency_rates y"
        + " where y.iso = x.curr"
        + " and y.rate_date=?");
...
    st.setInt(1, areaid);
    st.setDate(2, somedate);
    st.setDate(3, valuationdate);
    st.executeUpdate();
...
```

Интересно, что мой единственный запрос устраняет необходимость в двух служебных функциях, и это означает, что я вступил на совершенно иной путь рефакторинга, несовместимый с предыдущим случаем, когда я выполнял рефакторинг функций просмотра. Я проверяю пределы, объединяя их проверку с транзакциями, и конвертирую валюты, соединяя полученные транзакции, превышающие предел, с таблицей `currency_rates`. С одной стороны, мы получаем один более сложный (но по-прежнему понятный) запрос вместо нескольких более простых. С другой стороны, вызывающая программа *FifthExample.java* стала проще.

Прежде чем показать вам результат, я хочу представить вариант предыдущей программы, которая называется *SixthExample.java*, в которой я просто написал оператор SQL по-другому, используя больше соединений и меньше вложенных запросов:

```
PreparedStatement st = con.prepareStatement("insert into check_log(txid,"
        + "conv_amount)"
```

```
+ "select x.txid,x.amount*y.rate"
+ " from(select a.txid,"
+ "             a.amount,"
+ "             a.curr"
+ "      from transactions a"
+ "           inner join area_accounts b"
+ "                   on b.accountid = a.accountid"
+ "           inner join thresholds c"
+ "                   on c.iso = a.curr"
+ "      where b.areaid = ?"
+ "        and a.txdate >= date_sub(?, interval 30 day)"
+ "        and a.amount >= c.threshold) x"
+ "      inner join currency_rates y"
+ "              on y.iso = x.curr"
+ " where y.rate_date=?");
```

Сравнение и комментарии

Я запускал пять усовершенствованных версий, сначала без дополнительных индексов, затем с индексами по столбцу `accountid` и, наконец, с составным индексом (`accountid`, `txdate`) с MySQL, Oracle и SQL Server и измерял соотношение производительности по сравнению с исходной версией. Результаты работы программы *FirstExample.java* не так явно заметны на графиках (рис. 1.1, 1.2 и 1.3), но «пол» представляет исходную скорость работы программы *FirstExample*.

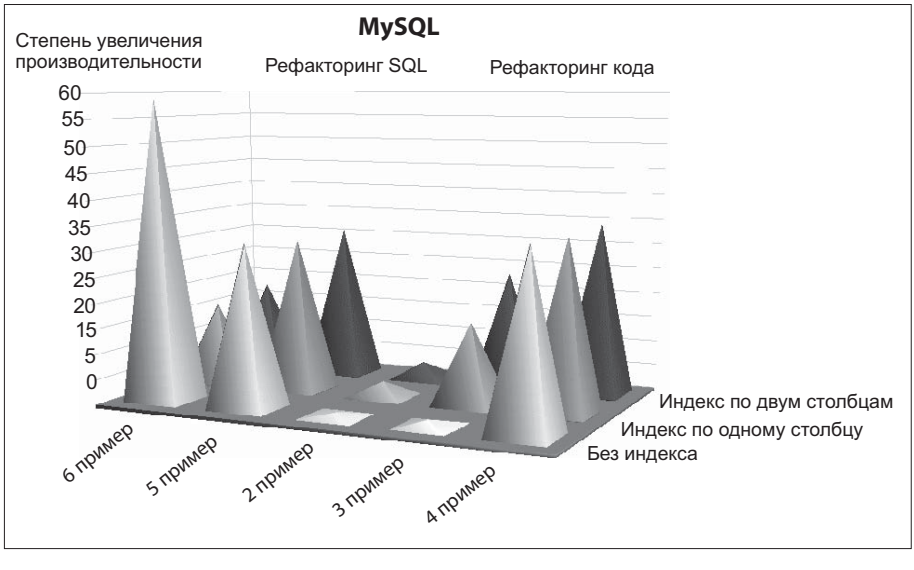

Рис. 1.1. *Результаты рефакторинга для MySQL*

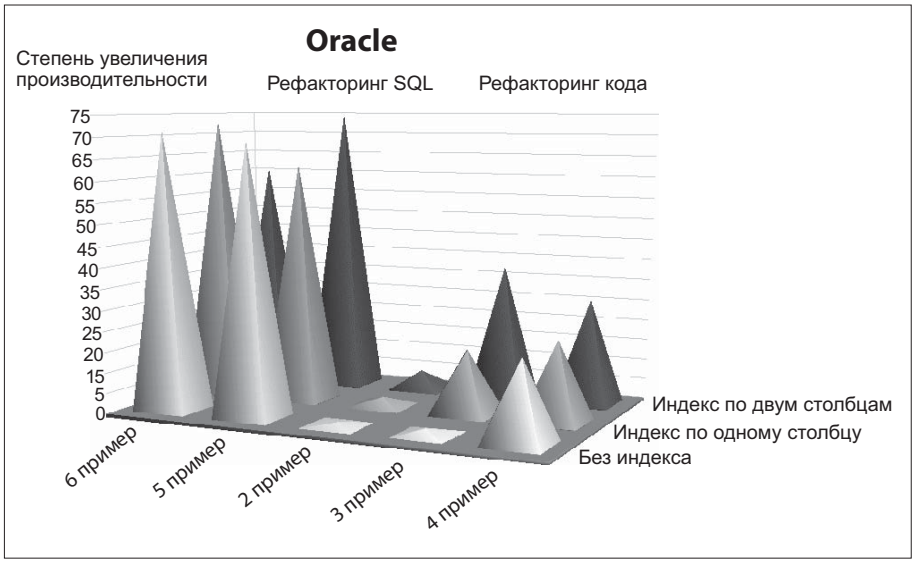

Рис. 1.2. *Результаты рефакторинга для Oracle*

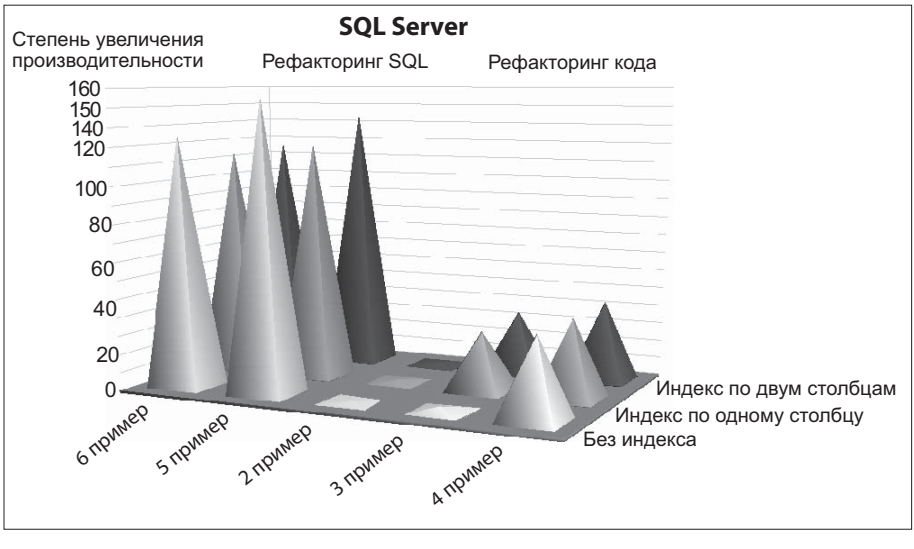

Рис. 1.3. *Результаты рефакторинга для SQL Server*

Что изображено на графиках?

На одной оси

Версия программы, в которой код был минимально усовершенствован, – посередине (*SecondExample.java*). С одной стороны – рефакторинг, ориентированный на код: *ThirdExample.java*, где минимизированы вызовы в функциях просмотра, и *FourthExample.java*, идентичный за исключением запроса с вложенным запросом, заменяющим два запроса. С другой стороны – результаты рефакторинга, ориентированного на SQL, в котором функции просмотра исчезли, с двумя вариантами главного оператора SQL.

На других осях

Различные варианты индексирования (без индекса, индекс по одному столбцу и индекс по двум столбцам).

Два обстоятельства сразу бросаются в глаза:

- Сходство диаграмм увеличения производительности, особенно в случае Oracle и SQL Server.
- Подход, ограничивающий усовершенствования только индексированием, представленный столбцами с подписью *SecondExample* с индексом по одному столбцу или по двум столбцам, дает очень маленький прирост производительности. Реальное ускорение было получено в других вариантах программы, хотя случай с MySQL интересен тем, что присутствие индекса серьезно снижает производительность (по сравнению с тем, как должно быть), как вы можете видеть для варианта с *SixthExample*.

Несомненно, лучший результат с MySQL достигнут, как и с остальными продуктами, с единственным запросом и без дополнительных индексов. Однако нужно заметить не только то, что в этой версии оптимизатор иногда может попытаться использовать индексы, даже когда они вредны, но и то, что он очень чувствителен к тому, как написан запрос. Сравнение *FifthExample* и *SixthExample* указывает на преимущество соединения перед (логически эквивалентными) вложенными запросами.

Oracle и SQL Server в этом примере демонстрируют, что их оптимизатор явно нечувствителен к вариациям синтаксиса (даже если SQL Server отмечает, в противоположность MySQL, небольшое преимущество вложенных запросов перед объединениями), и они достаточно интеллектуальны в этом случае, чтобы не использовать индексы, когда последние не ускоряют запрос. Оптимизаторы, к сожалению, могут вести себя не так идеально, когда операторы значительно сложнее, чем в этом простом примере, вот почему я посвятил пятую главу рефакторингу операторов. И Oracle, и SQL Server являются надежными рабочими лошадками в корпоративном мире, где многие информационные процессы состоят из пакетных процессов и массированного просмотра таблиц. Когда вы рассматриваете производительность Oracle с исходным запросом, три минуты можно считать очень приличным временем для выполнения

нескольких сотен полных просмотров таблицы с двумя миллионами строк (на машине не самого последнего поколения). Однако не забывайте, что небольшая переработка снизила время, требуемое для выполнения того же процесса (как в «требованиях бизнеса») до чуть меньше двух секунд. Иногда прекрасная производительность при выполнении полных просмотров означает, что время отклика будет посредственным, но не ужасающе долгим, и что серьезные дефекты кода остались незамеченными. Для этого процесса требуется только один полный просмотр таблицы транзакций. Возможно, никаких сигналов тревоги не возникло бы, если бы программа выполнила десять полных просмотров вместо 252, но от этого она не стала бы совершеннее.

Выбор среди различных подходов

Как я уже указывал, два различных подхода, которые я использовал для рефакторинга тестового кода, несовместимы: в одном случае я сконцентрировал свои усилия на усовершенствовании функций, которые в другом случае просто удалил. Из рис. 1.1, 1.2 и 1.3 совершенно очевидно, что лучшим подходом ко всем продуктам оказалось использование «единственного запроса», что делает создание нового индекса ненужным. Тот факт, что никакие дополнительные индексы не нужны, становится понятным, когда вы учитываете, что одно значение `areaid` определяет периметр, который представляет значительное подмножество в таблице. Извлечение многих строк с индексом дороже, чем просмотр их (более подробно на эту тему мы поговорим в следующей главе). Индекс необходим только тогда, когда у нас есть один запрос, возвращающий значения `accountid`, и другой запрос для получения данных транзакций, поскольку диапазон дат является выборочным для одного значения `accountid`, но не для всего набора счетов. Использование индексов (включая создание соответствующих дополнительных индексов), которое часто ассоциируется в умах с традиционным подходом к настройке SQL, может оказаться менее важным, когда вы находите метод рефакторинга.

Я не утверждаю с определенностью, что индексы не важны. Они могут быть очень нужны, в частности в средах оперативной обработки транзакций (OLTP). Но вопреки популярному мнению, значение их не всеобъемлюще; индексы являются только одним фактором из нескольких, и во многих случаях они оказываются не самым важным элементом из тех, которые надо принимать во внимание, когда вы пытаетесь увеличить производительность.

Самое важное, что добавление нового индекса приводит к риску появления проблем где-то еще. Кроме дополнительных требований к хранилищу, которые иногда могут быть весьма высокими, индекс добавляет накладные расходы ко всем вставкам и удалениям из таблицы, а также к обновлениям в индексируемых столбцах – индексы должны обновляться вдоль всей таблицы. Это может быть мелочью, если основной

проблемой является производительность запросов, и если у нас много времени для загрузки данных.

Есть, однако, еще более печальный факт. Просто рассмотрите на рис. 1.1 влияние индекса на производительность программы *SixthExample.java*: он превратил очень быстрый запрос в сравнительно медленный. Что если у нас уже есть запросы, написанные по тому же образцу, что и запрос в *SixthExample.java*? Я могу разрешить один вопрос, но при этом создать проблемы там, где их не было. Индексирование очень важно, и я буду обсуждать эту тему в следующей главе, но когда что-то хорошо работает, затрагивание индексов всегда рискованно[1]. То же самое верно в отношении изменений, которые оказывают глобальное влияние на базу данных, в частности изменений параметров, которые влияют даже больше на запросы, чем на индекс.

Однако могут быть и другие соображения, которые надо принимать во внимание. В зависимости от достоинств и недостатков членов команды разработчиков, не говоря уже о капризах руководства, оптимизацию функций просмотра и добавление индекса можно воспринимать как меньший риск, чем переосмысливание базового запроса. Предыдущий пример прост, и базовый запрос, хоть и не является совершенно тривиальным, представляет собой запрос средней сложности. В отдельных случаях написание удовлетворительного запроса может либо превышать квалификацию разработчиков, либо оказаться невозможным из-за неудачного дизайна базы данных, который нельзя изменить.

Несмотря на меньшее увеличение производительности и тщательные нерегрессивные тесты, требуемые в случае таких изменений в структуре базы данных, как дополнительный индекс, отдельное усовершенствование функций и главного запроса может быть иногда более приятным решением для вашего руководства, чем то, что я бы назвал большим рефакторингом. В конце концов, адекватное индексирование привело к увеличению производительности от 16 до 35 раз в случае *ThirdExample.java*, что нельзя назвать ничтожным.

Иногда разумно остановиться просто на хорошем, даже если отличный результат не за горами – вы всегда можете упомянуть отличное решение как последний вариант.

На каком бы решении и почему именно вы в конце концов ни остановились, вы должны понимать, что обоими подходами к рефакторингу движет одна и та же идея: минимизация количества обращений к серверу базы данных и, в частности, уменьшение слишком большого количества запросов, генерируемых функцией `AboveThreshold()`, которое было в первоначальной версии кода.

[1] Даже если, в худшем случае, прекращение использования индекса (или превращение его в невидимый в Oracle 11 или более поздней версии) является операцией, которую можно выполнить относительно быстро.

Оценка возможных выигрышей

Самая большая проблема, когда берешься за рефакторинг, – это, несомненно, реальная оценка возможных результатов.

Когда вы рассматриваете альтернативный вариант добавления оборудования для разрешения проблем с производительностью, вы погружаетесь в море цифр: количество и рабочая частота процессоров, память, скорость записи данных на диск… и, конечно, цена оборудования. Не забывайте, что зачастую приращение производительности после добавления оборудования может оказаться смехотворным, а в некоторых случаях и просто отрицательным[1] (когда желателен целый ряд возможных усовершенствований).

В подсознании высших руководителей, отвечающих за информационные технологии, глубоко укоренилась вера в то, что удвоение вычислительной мощности означает лучшую производительность – пусть не вдвое, но хотя бы около того. Если вы, предлагая рефакторинг, противопоставляете его варианту обновления оборудования, вы вступаете в тяжелую битву – у вас должны быть козыри, по меньшей мере не уступающие козырям сторонников аппаратного решения.

Действия по методу проб и ошибок в течение неопределенного времени, попытки случайных изменений в надежде на удачу не являются эффективными и не гарантируют успеха. Если после оценки того, что же нужно делать, вы не можете дать серьезные обоснования оценки времени, требуемого для реализации изменений, и ожидаемых положительных результатов, вам не удастся доказать свою правоту – разве что нового оборудования не окажется в наличии.

Оценка того, насколько вы сможете улучшить данную программу, – очень непростая задача. Во-первых, вы должны определить, в каких единицах вы собираетесь измерять это «насколько». Несомненно, пользователи (или руководители информационных подразделений) предпочитают услышать: «Мы можем уменьшить время, требуемое для выполнения этого процесса на 50%» или что-то подобное. Однако аргументы, касающиеся времени работы, очень опасны. Когда вы рассматриваете аппаратный вариант, вы принимаете в расчет дополнительную вычислительную мощность. В этом случае самой безопасной стратегией будет попытка оценить, насколько вы сможете сэкономить мощность благодаря более эффективной обработке данных и сколько времени вы сможете сберечь, устраняя некоторые процессы – например, повторение запроса тысячи или миллионы раз там, где надо запустить запрос только один раз. Поэтому главное – не хвастаться гипотетическим приростом производительности, который очень трудно предсказать, а доказать, во-первых, что в текущем коде есть неэффективные фрагменты, а во-вторых, что эти фрагменты несложно исправить.

[1] Из-за несовместимых требований к оборудованию. Это происходит не так часто, как незначительное улучшение, но случается.

Лучший способ доказать, что рефакторинг окупится, это, вероятно, покопаться немного глубже в трассировочном файле, полученном в Oracle для исходной программы (нет надобности говорить, что анализ статистики выполнения SQL Server даст сходный результат).

Трассировочный файл Oracle дает подробные данные о времени использования процессора и общем времени, затраченном на различные фазы (синтаксический анализ, выполнение и, для операторов `select`, извлечение кода) выполнения каждого оператора, а также различные «события ожидания» и время, затраченное СУБД на ожидание доступности ресурса. На рис. 1.4 вы можете увидеть, как в Oracle распределялось время исполнения операторов SQL в исходной версии примера из этой главы.

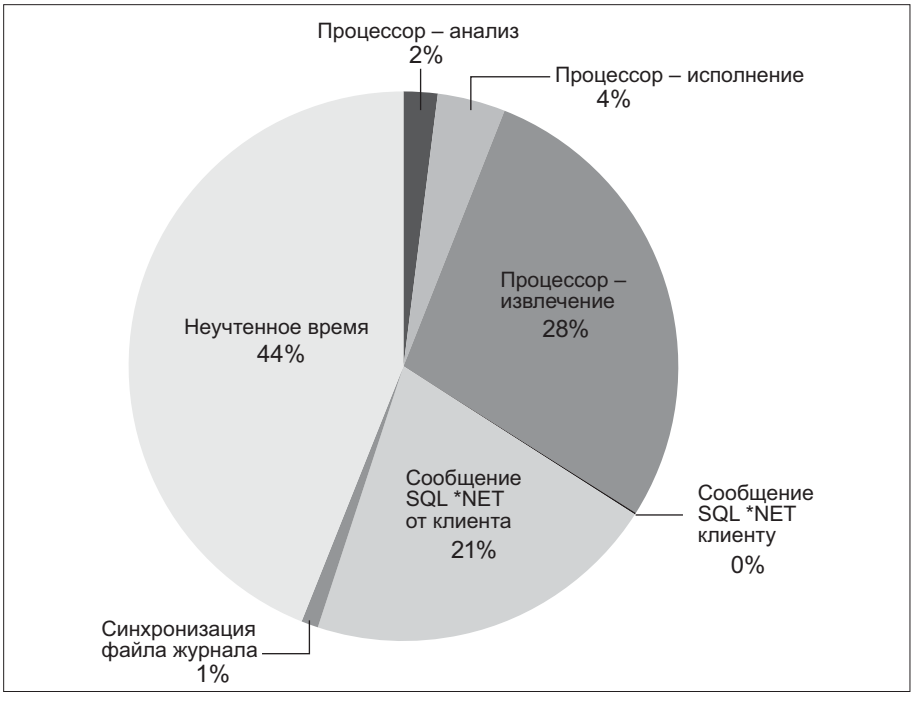

Рис. 1.4. *Как в Oracle распределялось время в первой версии*

Как видите, 128 секунд, записанных в файле трассировки, можно приблизительно разделить на три части:

- Процессорное время, когда Oracle обрабатывал запросы: его можно в свою очередь разделить на время синтаксического разбора операторов, время, требуемое на исполнение операторов, и время, требуемое на извлечение строк. Синтаксический разбор сводится к анализу операторов и выбору плана исполнения. В процессе исполнения время требуется на поиск первой строки результирующего набора для оператора `select` (может потребоваться время на сортировку этого

результирующего набора перед определением первой строки) и реальную модификацию таблицы для операторов, изменяющих данные. Вы также можете увидеть рекурсивные операторы, которые являются операторами по отношению к словарю данных, который получен из операторов программы, либо на стадии синтаксического разбора, либо, например, для выполнения процесса выделения пространства для вставки данных. Благодаря использованию мной подготовленных операторов и отсутствию массивной сортировки, основная часть этой секции занята извлечением строк. В случае жестко закодированных операторов каждый оператор появляется как совершенно новый запрос к SQL-машине, что означает получение информации из словаря данных для анализа и идентификации наилучшего плана исполнения; аналогично, операции сортировки обычно требуют динамического выделения временного пространства для хранения, что тоже означает запись данных выделения в словарь данных.

- Время ожидания, в течение которого СУБД либо простаивает (например, `SQL*Net message from client`, то есть время, когда Oracle просто ждет получения оператора SQL для обработки), либо ждет освобождения ресурса или окончания операции, например операций ввода-вывода, отмеченных двумя событиями `db file` (`db file sequential file` в первую очередь указывает на обращения к индексу и `db file scattered read` для просмотры таблицы, о чем трудно догадаться тому, кто не знает Oracle), каждое из которых полностью отсутствует здесь (все данные были загружены в память предыдущими статистическими вычислениями таблиц и индексов). Фактически, единственной операцией ввода-вывода, которую мы видим, является запись в файлы журналов, благодаря режиму автоматической фиксации изменений JDBC. Теперь вы понимаете, почему отключение автоматической фиксации изменений дало в этом случае очень мало изменений, поскольку на это ушел только 1% времени базы данных.
- Неучтенное время в результате различных системных ошибок, а также тот факт, что точность не может быть выше, чем точность часов, ошибок округления, внутренних операций Oracle и т. п.

Если бы я основывал мой анализ на соотношениях из рис. 1.4, чтобы попытаться предсказать, насколько процесс может быть усовершенствован, я не смог бы дать никаких надежных предположений о степени улучшения. Это тот случай, когда у вас возникает искушение последовать совету Сэмюэля Голдвина:

Никогда не делайте предсказаний, особенно о будущем.

Прежде всего, большинство периодов ожидания представляет собой ожидание работы (хотя факт, что СУБД ожидает работы, должен вызвать тревогу у опытного практика). Операции ввода-вывода не являются проблемой, несмотря на отсутствующий индекс. Вы можете ожидать, что индекс ускорит извлечение данных, но предыдущие эксперименты доказали, что вызванные появлением индекса улучшения были далеки

от глобальных. Если вы наивно полагали, что существует возможность избавиться от всех ожиданий, включая неучтенное время, то вы могли бы не менее наивно предполагать, что лучшее, чего вы можете добиться, это уменьшить время исполнения в три, а то и (при некоторой удаче) в четыре раза, в то время как я энергичным рефакторингом уменьшил это время в сто раз. Несомненно, лучше получить стократное ускорение, надеясь на трехкратное, чем наоборот. Однако по-прежнему не похоже, что вы знаете, что делаете.

Как я получил стократное ускорение, объяснить несложно (после того как дело сделано): я больше не извлекал строки и, сократив процесс практически до единственного оператора, также устранил периоды ожидания ввода от приложения (когда нужно выполнить много операторов SQL). Однако периоды ожидания сами по себе не дали мне полезной информации о том, где сосредоточить усилия. Лучшее, что можно получить из файлов трассировки и анализа периодов ожидания, это уверенность в том, что некоторые из самых популярных рецептов настройки базы данных если и дадут эффект, то очень маленький.

Периоды ожидания полезны, когда ваши изменения сильно ограниченны, а именно так бывает, когда вы настраиваете систему: они сообщают вам, где время тратится впустую и где вам нужно попробовать увеличить производительность любыми имеющимися в вашем распоряжении средствами. Так или иначе, продолжительность периодов ожидания также фиксирует верхнюю границу улучшения, которую вы можете ожидать. Они по-прежнему могут быть полезны, когда вы хотите выполнить рефакторинг кода, как индикатор слабости текущей версии (хотя это не единственный способ обнаружить недостатки). К сожалению, пользы от них мало, когда вы делаете попытку предсказать производительность после пересмотра кода. Ожидание ввода для приложения и большое количество неучтенного времени (когда сумма ошибок округления велика, это означает, что у вас много базовых операций) – и то и другое симптомы «болтливого» приложения. Однако чтобы понять, почему приложение так «болтливо» и выяснить, можно ли сделать его менее «болтливым» и более эффективным (иным способом, чем низкоуровневая настройка параметров TCP), мне нужно знать, не чего ждет СУБД, а что создает ее загруженность. В определении, что загружает СУБД, вы обычно находите множество операций, без которых после некоторых размышлений можно просто обойтись – вместо того, чтобы ускорять их.

Настройка – это попытка сделать ту же вещь быстрее; рефакторинг – это попытка добиться того же результата быстрее. Если вы сравните, что делает база данных, с тем, что она должна была бы или могла бы делать, вы сможете получить некоторые надежные и заслуживающие доверия цифры, завернутые в подходящие словесные предостережения. Как я уже упоминал, что действительно интересно в трассировке Oracle, так это неполный просмотр таблицы из двух миллионов строк. Если я проанализирую тот же самый файл трассировки другим образом, то смогу

создать табл. 1.7 (обратите внимание, что время работы программы меньше, чем процессорное время для третьего и четвертого операторов; это не опечатка, а то, что показывает файл трассировки, – просто результат ошибок округления).

Таблица 1.7. Сообщения файла трассировки Oracle о работе СУБД

Оператор	Количество исполнений	Процессор	Время работы	Строк
`select accountid from area_ accounts where areaid=:1`	1	0.01	0.04	270
`select txid, amount, curr from transactions ...`	270	38.14	39.67	31 029
`select threshold from thresholds where iso=:1`	31 029	3.51	3.38	30 301
`select :1 * rate from currency_ rates ...`	61	0.02	0.01	61
`insert into check_ log(...)`	61	0.01	0.01	61

Глядя на табл. 1.7, вы можете заметить следующее:

- Первая заметная деталь в табл. 1.7 заключается в том, что количество строк, возвращаемых одним оператором, чаще всего равно количеству исполнений следующего оператора: очевидный знак, что мы просто передаем результат одного запроса в следующий запрос вместо выполнения соединений.
- Вторая бросающаяся в глаза подробность заключается в том, что все время работы на стороне СУБД – это процессорное время. Таблица из двух миллионов строк большей частью кэширована в памяти и просматривается в ней. Полный просмотр таблицы не обязательно означает операции ввода-вывода.
- Мы делаем запросы к таблице пределов свыше 30 000 раз, при этом в большинстве случаев возвращается одна строка. Эта таблица содержит 20 строк. Это означает, что каждое отдельное значение в среднем извлекается 1 500 раз.
- Oracle сообщает время работы около 43 секунд. Измеренное время выполнения при этом запуске равно 128 секундам. Поскольку не было никаких операций ввода-вывода, о которых можно было бы говорить, различие проистекает только из времени исполнения кода Java и из «диалога» между программой Java и сервером СУБД. Если мы уменьшим количество исполнений, мы можем ожидать, что время ожидания, пока СУБД ответит на вызовы JDBC, уменьшится пропорционально.

У нас уже есть отличное объяснение, что же не так: мы тратим много времени на извлечение данных из таблицы transactions, мы запрашиваем пределы в 1500 раз чаще, чем надо, и мы не используем соединения, таким образом умножая обмены между приложением и сервером.

Но вопрос в том, на что мы можем надеяться. Из цифр совершенно очевидно, что если мы извлекаем данные только 20 раз (или даже 100 раз) из таблицы transactions, время, в общей сложности затраченное на запросы, упадет почти до нуля. Сложнее оценить время, которое нам нужно для выполнения запросов к таблице transactions, и мы можем сделать это, представив наихудший случай. Нет необходимости возвращать единственную транзакцию несколько раз; поэтому худший случай, который я могу себе представить, заключается в полном просмотре таблицы и возвращении примерно 1/64 (2 000 000, разделенные на 31 000) ее строк. Я легко могу написать запрос, который благодаря псевдостолбцу Oracle с номерами строк в том порядке, в котором они возвращены (с другой СУБД я мог бы использовать переменную сеанса), возвратит каждую 65-ю строку, и запустить этот запрос с SQL*Plus следующим образом:

```
SQL> set timing on
SQL> set autotrace traceonly
SQL> select *
  2 from (select rownum rn, t.*
  3       from transactions t)
  4 where mod(rn, 65) = 0
  5 /

30769 rows selected.

Elapsed: 00:00:03.67

Execution Plan
----------------------------------------------------------
Plan hash value: 3537505522

--------------------------------------------------------------------------------
| Id | Operation        | Name       | Rows | Bytes| Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|  0 | SELECT STATEMENT |            | 2000K|  125M| 2495    (2)| 00:00:30 |
|* 1 | VIEW             |            | 2000K|  125M| 2495    (2)| 00:00:30 |
|  2 | COUNT            |            |      |      |            |          |
|  3 | TABLE ACCESS FULL|TRANSACTIONS| 2000K|   47M| 2495    (2)| 00:00:30 |
--------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

  1 - filter(MOD("RN",65)=0)
```

```
Statistics
----------------------------------------------------------
      1 recursive calls
      0 db block gets
  10622 consistent gets
   8572 physical reads
      0 redo size
1246889 bytes sent via SQL*Net to client
  22980 bytes received via SQL*Net from client
   2053 SQL*Net roundtrips to/from client
      0 sorts (memory)
      0 sorts (disk)
  30769 rows processed
```

При полном просмотре возвращается примерно столько же строк, как и в моей программе – это заняло меньше четырех секунд, хотя найти данные в памяти не удавалось 8 раз из 10 (отношение `physical reads` к `consistent gets + db block gets`, что является количеством логических ссылок к блокам Oracle, содержащим данные). Этот простой тест является доказательством того, что нет необходимости тратить почти 40 секунд на извлечение информации из таблицы `transactions`.

Кроме переписывания исходной программы, нет больше ничего, что мы могли бы сделать на этом этапе. Но у нас есть хорошее подтверждение концепции, достаточно убедительно демонстрирующей, что правильное переписывание программы может заставить ее использовать меньше пяти секунд процессорного времени вместо сорока. Я выделяю здесь процессорное время, а не время выполнения, поскольку время выполнения включает периоды ожидания, которое, как я знаю, станет ничтожным, но для которого у меня нет надежных цифр. Говоря о процессорном времени, я могу сравнить результат переписывания кода с покупкой машины, в восемь раз более быстрой, чем нынешняя, и стоящей гораздо дороже. Отсюда вы можете оценить, сколько времени займет рефакторинг программы, сколько времени потребуется для тестирования, понадобится ли вам сторонняя консультация – все вещи, которые зависят от вашей среды, объема кода, который нужно пересмотреть, и силы команды разработчиков. Посчитайте, сколько это будет вам стоить, и сравните с ценой сервера, в восемь раз более мощного (как в этом примере), чем ваш нынешний сервер, если такой вообще существует. Вы, вероятно, признаете свою позицию очень убедительной.

Поэтому процесс оценки, можете ли вы возлагать надежды на рефакторинг, требует двух шагов:

1. Выяснить, что база данных делает. Только когда вы знаете, что делает ваш сервер, вы можете обнаружить проблемные запросы. Затем вы обнаруживаете, что несколько дорогостоящих операций можно сгруппировать и выполнить за один проход и поэтому данный конкретный запрос не нужно выполнять столько раз, сколько это происходит сейчас, или что использование результата запроса в цикле для выполнения другого запроса можно заменить на более эффективное соединение.

2. Проанализируйте текущую активность критическим взглядом и создайте простые, обоснованные доказательства концепции, которая убедительно продемонстрирует, что код может быть более эффективным.

Большая часть этой книги посвящена описанию шаблонов, которые, вероятно, будут хорошей базой для рефакторинга и – как следствие дадут вам идеи, что может быть доказательством концепции. Но прежде чем мы обсудим эти шаблоны, давайте посмотрим, как вы можете идентифицировать операторы SQL, которые исполняются.

Выяснение, что делает база данных

У вас есть несколько способов отслеживать активность SQL на вашей базе данных. Какие-то варианты доступны не на всех СУБД, а некоторые методы больше подходят для тестовых баз данных, чем для баз данных, участвующих в реальных бизнес-процессах. К тому же одни методы позволяют вам отслеживать все, что происходит, а другие позволяют сфокусироваться на конкретных сессиях. Каждый из этих способов имеет свое применение.

Запрос динамических представлений

В шестой версии Oracle появилось то, что ныне стало популярной тенденцией: динамические представления, реляционные (табличные) представления структур памяти, которые система обновляет в режиме реального времени и к которым вы можете делать запросы с помощью SQL. Некоторые представления, например `v$session` в Oracle, `sys.dm_exec_requests` в SQL Server и `information_schema.processlist` в MySQL, позволяют вам видеть, что выполняется в данный момент (или только что было исполнено). Однако отбор этих представлений не будет убедительным, если вы не знаете либо скорости, с которой эти запросы выполняются, либо их продолжительности. Действительно интересны те динамические представления, которые отображают кэш оператора, со счетчиками числа исполнений каждого оператора, процессорным временем, затраченным каждым оператором, количеством базовых единиц данных (именуемых блоками (*block*) в Oracle и страницами (*page*) в других СУБД), связанных с оператором, и т. п. Количество страниц данных, к которым был осуществлен доступ при выполнении запроса, обычно называется *logical read* и является хорошим индикатором объема работы, требуемой для выполнения оператора. Релевантными динамическими представлениями являются `v$sqlstats`[1] в Oracle и `sys.dm_exec_query_stats` в SQL Server. Как вы можете заключить из названий, запрос к этим представлениям может сделать не каждый пользователь, только администратор базы данных может разрешить вам это. Эти представления также являются основой для утилит мониторинга (на которые иногда нужна отдельная

[1] Доступно начиная с Oracle 10, но информация была доступна в `v$sql` в предыдущих версиях.

лицензия), например Automatic Workload Repository (AWR) для Oracle и других продуктов.

Динамические представления являются отличным источником информации, но с ними связаны три проблемы:

- Они дают мгновенные снимки, даже если счетчики являются кумулятивными. Если вы хотите иметь ясную картину того, что происходит в течение дня или просто за несколько часов, вам придется делать периодические снимки и сохранять их где-то. Иначе вы рискуете сфокусироваться на большом ресурсоемком запросе, запущенном в два часа ночи, когда никто не следит за базой данных, и упустить реальные проблемы. Получение регулярных снимков – это именно то, что делает утилита Statspack СУБД Oracle, а также продукт Oracle ASH (что означает Active Session History – история активного сеанса), который требует отдельного лицензирования (то есть лишних расходов).
- Запрос к структурам в памяти, которые показывают существенную и крайне изменчивую информацию, занимает время, которое можно сравнить с многими запросами к базе данных, но может нанести ущерб всей системе. Это утверждение нельзя в равной степени отнести ко всем динамическим представлениям (некоторые более чувствительны, чем другие), но чтобы вернуть подходящую информацию, процесс, который возвращает данные из самой глубины СУБД, должен каким-то образом заблокировать структуры на то время, пока он выполняет считывание из них: другими словами, когда вы считываете информацию, указывающую, что один конкретный оператор был исполнен 1 768 934 раза, процесс, который только что исполнил его снова и хочет выполнить реальную запись, должен ждать обновления счетчика. Это не занимает много времени, но на системе с очень большим кэшем и многими операторами, возможно, что ваш запрос `select` к представлению, который за кулисами представляет собой обход некоторой сложной структуры в памяти, затормозит не один процесс. Вы не должны забывать, что вам придется сохранить возвращенный снимок, что обязательно затормозит ваш запрос. Другими словами, если вы будете обращаться к динамическим представлениям с большой частотой, вряд ли остальные пользователи смогут работать нормально.
- В результате есть определенный риск что-нибудь потерять. Проблема в том, что счетчики связаны с кэшированным оператором. С какой бы точки зрения вы не смотрели бы на проблему, кэш имеет ограниченный размер, и если вы выполняете очень большое количество различных операторов, в какой-то момент СУБД будет вынуждена заново использовать кэш, в котором до этого хранился старый оператор. Если вы исполните совсем большое количество различных операторов, обновление может быть быстрым, и выполняется SQL со счетчиками. Исполнить множество различных операторов несложно; вам нужно только выполнять жесткое кодирование и избегать подготовленных операторов и хранимых процедур.

Зная о последних двух проблемах, некоторые сторонние разработчики создали программы, которые привязываются к структурам памяти СУБД и считывают их напрямую, без дополнительно нагрузки на слой SQL. Это позволяет им сканировать память значительно быстрее с минимальным воздействием на основную работу СУБД. Их способность сканировать память несколько раз в секунду может удивить, например, многих успешных администраторов баз данных Oracle, делающих снимки активности каждые полчаса.

В сущности, потеря оператора не является чем-то очень важным, когда вы пытаетесь оценить, чем загружен сервер базы данных. Вам просто нужно отлавливать важные операторы, которые вносят серьезный вклад в загрузку машины или имеют слишком большое время отклика. Попросту говоря, есть две значительные категории операторов, заслуживающих настройки:

- Большие, ужасные, медленные операторы SQL, которые хотят настроить все (и которые трудно пропустить).
- Операторы, которые сами по себе не являются ресурсоемкими, но которые выполняются столько раз, что их кумулятивный эффект очень велик.

В тестовой программе в начале этой главы мы встретились с примерами обеих категорий: полный просмотр таблицы из двух миллионов строк, который можно оценить как медленный запрос, и сверхбыстрые, часто используемые функции просмотра.

Если мы будем проверять содержимое кэша операторов с постоянными интервалами, мы не найдем операторов, которые были использованы первыми и затем вытеснены из кэша между двумя проверками. Я никогда не встречал приложения, которое бы выполняло очень большое количество разнотипных операторов. Но печально то, что СУБД полагает два оператора идентичными только в том случае, если их текст идентичен с точностью до байта:

- Если операторы являются подготовленными и используют переменные или если мы вызываем хранимые процедуры. Даже когда мы выполняем опрос с небольшой частотой, мы получим картину, которая может быть неполной, но которая является хорошим представлением, что же происходит, поскольку операторы будут выполняться снова и снова и будут оставаться в кэше.
- Если вместо использования подготовленных операторов вы построите их динамически, связывая с константами, которые никогда не будут одинаковыми, то ваши операторы на практике могут быть идентичными, но SQL-машина увидит множество различных выражений, – каждое из которых будет очень быстрым и выполняться только один раз, – которые будут попадать в кэш и молниеносно вытесняться оттуда, чтобы освободить место для других подобных выражений. Если вы проверяете содержимое кэша мимоходом, вы поймаете только небольшую часть того, что было выполнено в действительности,

и получите неправильное представление о том, что происходит, поскольку несколько операторов, которые могут правильно использовать параметры, будут иметь непропорционально большой «вес» по сравнению с жестко закодированными операторами, которые могут быть (и скорее всего являются) действительной проблемой. В случае SQL Server, в котором хранимые процедуры получают режим предпочтения в том, что касается нахождения в кэше, операторы SQL, исполненные вне хранимых процедур, поскольку они более изменчивы, могут выглядеть менее опасными, чем они есть на самом деле.

Чтобы точно обрисовать, как это может работать на практике, предположим, что у нас есть запрос (или хранимая процедура), использующий параметры, цену которого мы установим в 200 единиц, и короткий жестко закодированный запрос, цену которого мы установим в три единицы. Вычислим, что между двумя запросами к кэшу операторов более дорогой запрос выполнится пять раз, а короткий запрос 4000 раз – похоже на 4000 отдельных операторов. Более того, предположим, что наш кэш операторов может хранить не более 100 запросов (это значительно меньше типичных реальных значений). Если вы проверите верхние пять запросов, более дорогой запрос будет иметь общую стоимость 1000 (5200), а за ним будут следовать четыре запроса со сравнительно смешной ценой 3 (1×3 в каждом случае). Кроме того, если вы попытаетесь выразить относительную важность каждого запроса, сравнивая цену каждого из них с общей ценой кэшированных операторов, вы можете неправильно вычислить общее значение для кэша как $1 \times 1000 + 99 \times 3 = 1297$ (основываясь на текущих значениях в кэше), в которых дорогой запрос составляет 77%. Использование текущего значения кэша отдельно значительно преуменьшает общую стоимость быстрого запроса. Если применить единицы стоимости из текущего кэша к общему количеству операторов, выполненных за этот период, реальная полная цена за период будет равна $5 \times 200 + 4000 \times 3 = 13\,000$. Это означает, что «дорогой» запрос представляет только 7% от реальной цены, а «быстрый» запрос – 93%.

Поэтому вы должны собирать цифры не только по операторам, но и глобальные цифры, которые подскажут вам, какая доля реальной нагрузки объясняется операторами, которые вы собрали. Поэтому, если вы решили сделать запрос `v$sqlstats` в Oracle (или `sys.dm_exec_query_stats` в SQL Server), вам нужно также найти глобальные счетчики. В Oracle вы их легко найдете в `v$sysstat`: общее количество исполненных операторов, затраченное процессорное время, количество логических считываний и число физических операций ввода-вывода с момента запуска базы данных. В SQL Server многие ключевые цифры находятся в переменных, например `@@CPU_BUSY`, `@@TIMETICKS`, `@@TOTAL_READ` и `@@TOTAL_WRITE`. Однако числом, которое, вероятно, представляет собой лучший критерий работы, выполненной любой СУБД, является количество логических считываний, и его можно найти в `sys.dm_os_performance_counters` как значение, связанное с не совсем удачно названным счетчиком `Page lookups/sec` (это кумулятивное значение, а не частота).

Теперь о том, что же делать, если вы осознали, что, проверяя кэш операторов, скажем, каждые десять 10 минут, вы теряете множество операторов? Например, в моем предыдущем примере я объяснил стоимость 1297 из 13 000. Это само по себе довольно плохой знак, и как будет видно в следующей главе, мы, вероятно, можем ожидать довольно значительной выгоды просто от использования подготовленных операторов. Решение может заключаться в увеличении частоты опросов, но, как я уже объяснял, проверка может быть дорогой. Слишком частая проверка может привести к проблемам в системе, здоровье которой и так под вопросом.

Когда вы теряете множество операторов, есть два решения для получения менее искаженного представления, что же делает база данных: одно заключается в переходе на анализ журнальных файлов (см. следующий раздел), а другое – базовый анализ плана исполнения вместо текста операторов.

Область памяти в кэше операторов, связанная с каждым оператором, содержит также ссылку на план исполнения, связанный с этим оператором (`plan_hash_value` в `v$sqlstats` в Oracle, `plan_handle` в `sys.dm_exec_query_stats` в SQL Server; обратите внимание, что в Oracle каждый план указывает на адрес «родительского курсора» в отличие от SQL Server, где оператор указывает на план). Связанную с планом статистику можно найти соответственно в `v$sql_plan_statistics` (с чрезвычайной детализацией и когда для сбора статистики для базы данных установлен высокий уровень, с почти таким же объемом информации, что и в файлах трассировки) и в `sys.dm_exec_cached_plans`.

Планы исполнения – это не то, чем мы интересуемся; в конце концов, если мы запустим *FirstExample.java* с дополнительным индексом, мы получим очень хорошие планы исполнения, но жалкую производительность. Однако жестко закодированные операторы, которые отличаются только константами, будут большей частью использовать один и тот же план исполнения. Поэтому объединение статистики операторов по связанным с ними планами исполнения, а не по их тексту, даст значительно более правдивое представление о том, что же делает база данных в действительности. Не вызывает сомнения, что планы исполнения не являются безупречными индикаторами. Некоторые изменения на уровне сеанса могут привести к различным планам исполнения с одним и тем же текстом оператора в двух сеансах (у нас также могут быть синонимы, превращающиеся в совершенно различные объекты); и любой простой запрос к неиндексированной таблице приведет к тому же самому просмотру таблицы, какие бы столбцы вы ни использовали для фильтрации строк. Но даже если нет однозначного соответствия между шаблоном оператора и планом исполнения, планы исполнения (со связанным примером запроса) определенно позволят вам понять, на какой тип операторов нужно обратить внимание, лучше, чем один текст оператора, когда числа не суммируются с оператором.

Добавление операторов в файл трассировки

Во всех СУБД есть возможность вести журнал исполненных операторов SQL. Ведение журнала основано на перехвате операторов до (или после) их выполнения и записи их в файл. Перехват может осуществляться в нескольких местах: там, где оператор выполняется (на сервере базы данных), или там, где он выпущен (на стороне приложения), или где-то посередине.

Ведение журнала на сервере. Ведение журнала на сервере – это то, о чем думает большинство. Диапазоном может быть или вся система, или текущий сеанс. Например, когда вы регистрируете медленные запросы в MySQL, запуская сервер с параметром `--log-slow-queries`, этот параметр влияет на каждый сеанс. Препятствие в том, что если вы хотите собрать интересующую информацию, объем данных, которые вам нужно зарегистрировать, часто бывает весьма большим. Например, если мы запустим программу *FirstExample.java* с сервером MySQL, запущенным с параметром `--log-slow-queries`, в журнале регистрации мы ничего не найдем, если у нас есть дополнительный индекс в таблице `transactions`: здесь нет медленных запросов (медленным является весь процесс), тем не менее можно добиться ускорения в 15 или 20 раз. Регистрация этих медленных запросов может быть полезна администраторам для идентификации больших ошибок и отвратительных запросов, но она ничего вам не скажет о том, что база данных делает на самом деле, поскольку каждый модуль работает достаточно быстро.

Поэтому, если вы действительно хотите выяснить, в чем проблема, вам нужно регистрировать каждый оператор, который выполняет СУБД (запуская сервер MySQL с параметром `--log`, присваивая глобальной переменной `general_log` значение `ON` и присваивая параметру Oracle `sql_trace` значение `true` или активизируя событие 10046; если вы используете SQL Server, вызовите `sp_trace_create` для определения вашего файла трассировки, затем `sp_trace_setevent` несколько раз для указания, что же вы хотите отслеживать, а затем `sp_trace_setstatus` для запуска трассировки). На практике это означает, что если вы хотите перевести загруженный сервер в режим трассировки, то вы получите гигантские файлы журналов (иногда их будет много: Oracle создает один файл трассировки на серверный процесс, и их количество может измеряться сотнями, а в SQL Server новый файл трассировки может создаваться каждый раз, когда текущий файл становится слишком большим). На рабочем сервере вы постараетесь избежать такой ситуации, особенно если уже есть признаки, что сервер на последнем издыхании. В дополнение к накладным расходам, вызванным хронометражем, отловом и записью операторов, возникает риск переполнения диска, что очень дискомфортно для СУБД. Однако это может быть полезным, если вы уверены в системе или можете создать нагрузку, эмулирующую реальный производственный процесс.

Тем не менее когда вы хотите собрать данные на рабочей системе, обычно безопаснее отслеживать отдельный сеанс, что вы можете сделать

в MySQL, присвоив переменной сеанса `sql_log_off` значение `ON`, а в Oracle запустив:

```
alter session set timed_statistics=true,
alter session set max_dump_file_size=unlimited;
alter session se tracefile_identifier='something meaningful';
alter session set events '10046 trace name context forever, level 8';
```

Параметр `level` для события определяет, сколько информации собирать. Возможными значениями являются:

- 1 (базовая информация трассировки);
- 4 (связанные переменные, значения, которые передаются);
- 8 (статистика ожиданий);
- 12 (статистика ожиданий и связанные переменные).

В SQL Server вам нужно вызвать `sp_trace_filter`, чтобы указать Service Profile Identifier (SPID) или регистрационное имя (в зависимости от того, что удобнее), что вы хотите отслеживать, прежде чем включать трассировку.

Поскольку не всегда удобно модифицировать существующую программу для добавления такого типа операторов, вы можете попросить администратора базы данных либо включать трассировку вашего сеанса, когда он запускается (администраторы баз данных иногда могут это), либо, что более удобно, создать специальную учетную запись пользователя, чтобы режим трассировки включался, когда вы подключаетесь к базе данных под этой учетной записью.

Ведение журнала на клиентской стороне. К сожалению, у ведения журнала на сервере есть свои недостатки:

- Поскольку регистрация происходит на сервере, это требует ресурсов, которые могут быть дефицитными, что может привести к падению общей производительности, даже если вы отслеживаете один сеанс.
- Отслеживание отдельного сеанса может оказаться сложным, когда вы используете сервер приложений, который опрашивает соединения с базой данных: больше нет однозначной связи между сеансом конечного пользователя и сеансом базы данных. В результате вам может потребоваться отслеживать больше или меньше, чем вы предполагали первоначально, и будет трудно получить ясную картину активности, не попавшую в файлы трассировки, без соответствующих инструментов.

Когда вы являетесь разработчиком, вход на сервер рабочей базы данных вам часто запрещен. Файлы трассировки будут созданы под учетными записями, к которым вы не имеете доступа, и вы будете зависеть от администраторов, чтобы получить и обработать сгенерированные вами файлы трассировки. Зависимость от людей, которые часто бывают очень заняты, может быть не очень удобна, особенно если вы хотите повторять операции трассировки много раз.

Альтернативной возможностью является отслеживание операторов SQL там, где они выпускаются: на стороне сервера приложений. Идеальным случаем является, конечно, когда само приложение снабжено правильными инструментами. Но могут быть другие возможности:

- Если вы используете такой сервер приложений, как *JBoss*, *WebLogic* или *WebSphere* – или просто JDBC, – вы можете использовать бесплатный трассировщик *p6spy*, который будет хронометрировать каждый исполненный оператор.
- С JDBC вы иногда можете включать ведение журнала в драйвере JDBC (например, вы можете сделать это с драйвером JDBC MySQL).

Промежуточный вариант ведения журнала. Возможны также гибридные решения. Примером для SQL Server является SQL Profiler, который активизирует трассировку на серверной стороне, но является инструментом клиентской стороны, получающим данные трассировки с сервера. Хотя использовать SQL Profiler намного приятнее, чем системные процедуры T-SQL, трафик данных в сторону инструмента весьма велик, и он не так эффективен, как «чистая» трассировка на стороне сервера.

Я должен упомянуть, что вы можете перехватывать и хронометрировать операторы SQL и другими способами, даже если они требуют хорошего объема программирования; вы можете обнаружить эти приемы в некоторых продуктах сторонних разработчиков. Когда вы «беседуете» с СУБД, вы фактически посылаете ваши операторы SQL на порт TCP/IP на сервере. Существуют программы, которые могут прослушивать порт, отличный от порта, который прослушивает программа-приемник СУБД; эта программа будет перехватывать и записывать все, что вы ей посылаете, прежде чем передать СУБД. Иногда поставщик СУБД официально допускает этот метод и предоставляет true proxy (например, MySQL Proxy или Sybase Open Server). Техника использования прокси подобна реализации веб-фильтрации через прокси-сервер.

Использование файлов трассировки

С файлами трассировки вы не упустите никаких операторов, а это означает, что вы получите огромное количество данных. Я уже объяснял, что если операторы жестко закодированы, вы упустите многие из них и получите, вообще говоря, искаженное представление о том, что происходит, при просмотре кэша операторов. С файлами трассировки проблема несколько другая: вы получите в вашем файле так много операторов, что он может оказаться бесполезным, даже после обработки сырых файлов журнала с помощью *tkprof* для Oracle или *mysqlsla* [1] для MySQL.

Хорошим способом сохранять файлы является их загрузка в базу данных и выполнение запроса к ним. Программа *tkprof* компании Oracle давно способна генерировать операторы `insert` вместо текстовых отчетов, а инструменты SQL Server и функция `fn_trace_gettable()` делают

[1] http://hackmysql.com

загрузку файла трассировки в таблицу очень простой операцией. Есть только одно препятствие: вам нужна база данных, в которую вы будете загружать файлы. Если в вашем распоряжении нет базы данных для целей разработки или вы являетесь консультантом, который старается четко разделять свою работу и работу клиентов, подумайте о SQLite[1]: вы можете использовать ее для создания файла, в котором данные хранятся в виде таблиц и запросов на языке SQL, даже если загрузка файла трассировки требует некоторого программирования.

Проблема в том, что даже при наличии базы данных в вашем распоряжении объем анализируемых данных может обескуражить. Я однажды делал трассировку в Oracle (зная, что запрос `v$sqlstats` должен мне ничего не дать, поскольку большинство операторов было жестко запрограммировано) программы, которой требовалось около 80 минут для генерации 70-страничного отчета. Это был сервер Sun корпоративного уровня с операционной системой Solaris; текстовый отчет, полученный как вывод программы `tkprof`, был столь велик, что мне не удалось открыть его с помощью `vi` и я выполнил анализ с помощью таких команд, как `grep`, `sed`, `awk` и `wc` (если вы не очень знакомы с Unix-подобными операционными системами, это мощные, но весьма низкоуровневые утилиты командной строки). Цель была не в рефакторинге кода (не было исходного кода, были доступны только исполнимые модули), а в выяснении, почему клиент-серверное приложение так медленно работает на мощном сервере, когда для создания такого же отчета с теми же данными на мобильном компьютере технического консультанта поставщика (который дал тот же совет купить более мощный сервер) требовалось значительно меньше времени. Тем не менее файл трассировки дал мне ответ: в файле было около 600 000 операторов; их среднее время выполнения 80 минут / 600 000 = 8 миллисекунд. Фактически Oracle простаивал 90% времени, и большую часть времени работы программы составляли сетевые задержки. Я был счастлив иметь возможность сделать рефакторинг этой программы. Было несложно сократить время выполнения программы на несколько порядков.

Вы должны обрабатывать файлы трассировки (или данные трассировки), если хотите что-то получить от них. Если операторы жестко запрограммированы, лучше с помощью регулярных выражений заменить все строковые константы на что-нибудь фиксированное (например, *constant*), а все числовые константы на ноль, к примеру. Это ваш единственный шанс свести операторы, идентичные с точки зрения приложения, но не с точки зрения СУБД. Вам может потребоваться сжать файл трассировки с тем, чтобы присвоить каждой строке одну и ту же метку времени (единственный способ точно узнать, когда был выполнен каждый оператор). После того как это сделано, вам нужно агрегировать всю хронометрическую информацию, которая у вас есть, и сосчитать, сколько у вас есть идентичных шаблонов. После этого у вас будет серьезный материал для анализа того, что вы можете сделать для увеличения производительности.

[1] http://www.sqlite.org

Анализ собранного материала

Теперь давайте посмотрим, как мы можем проанализировать данные, либо собранные проверкой содержимого кэша, либо имеющиеся в журналах, и что мы узнаем о возможностях увеличения производительности.

Во всех случаях я обнаружил, что большая часть загрузки базы данных в значительные периоды дня (или ночи) была результатом менее чем десяти запросов или шаблонов запросов, и вы обычно можете сузить количество таких запросов до четырех или пяти.

Большинство из этих запросов будут очень большими или быстрыми, но выполняемыми очень часто. Как мы уже видели, даже если мы не можем игнорировать большие запросы, фокусируясь на отдельных запросах, мы можем упустить большие приросты производительности. Нам нужно вернуться на шаг назад, сопоставить активность SQL с деловой активностью и проверить, нельзя ли сделать так, чтобы очень часто выполняемые действия стали менее частыми или вообще не происходили. Если нам удастся идентифицировать запросы, которые мы могли бы выполнять реже, то нам нужно определить, какое процессорное время они используют в настоящее время, и вычислить возможную экономию. Сделать это проще и безопаснее, чем пытаться предсказать время отклика. Когда вы покупаете более мощный сервер, все, что вы знаете наверняка, это то, что фактически вы покупаете процессорное время, а собственные тесты поставщика позволяют вам выяснить соотношение мощностей между вашим нынешним сервером и новым. Если вы убедительно продемонстрируете, что рефакторинг кода может уменьшить потребление процессорного времени на 20% в периоды наибольшей нагрузки, то для всех практических целей это означает, что предполагаемый рефакторинг даст тот же результат, что и покупка нового сервера с мощностью на 20% большей, минус затраты на миграцию.

Фактически рефакторинг дает больше, чем просто уменьшение используемого процессорного времени на сервере, и я надеюсь, что рис. 1.5 даст вам хорошее представление о реальном ожидаемом улучшении. Во многих плохо написанных приложениях происходит непрерывный обмен между клиентской и серверной частью приложения. Предположим, у нас есть цикл курсора – некоторый оператор `select`, возвращающий какое-то количество строк и выполняющий, например, оператор `update` в цикле. Приложение произведет некоторое количество вызовов базы данных:

- Первым будет вызов `execute`, которые проанализирует оператор `select`, сделает его синтаксический разбор, при необходимости определит план исполнения и сделает все необходимое (включая сортировки) для обнаружения первой возвращаемой строки.
- Затем приложение будет повторять вызовы `fetch`, которые будут возвращать строки либо одну за другой, либо, в некоторых случаях, пакетами.

- До выхода из цикла другой вызов `execute` будет выполнять оператор `update`. Поскольку эта операция изменяет базу данных, для этого оператора после `execute` не будет никаких других вызовов.

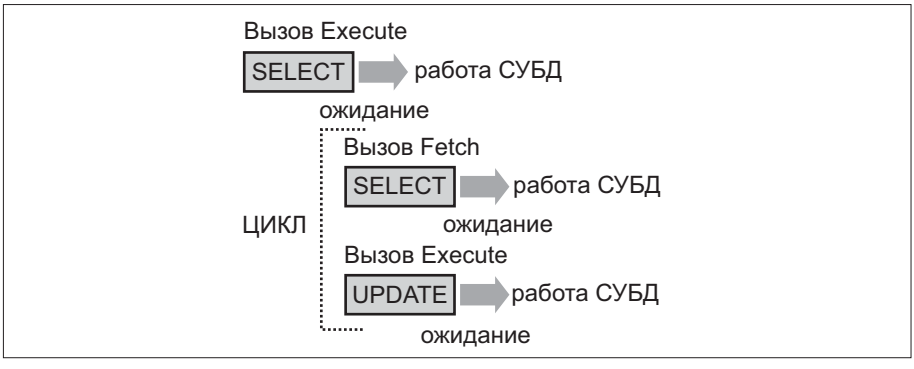

***Рис. 1.5.** «Разговорчивое» приложение SQL*

Основные вызовы базы данных являются синхронными, это означает, что большинство приложений приостанавливаются и ждут, пока СУБД выполнит работу на своей стороне. Время ожидания фактически состоит из нескольких компонентов:

- времени передачи, которое требуется, чтобы оператор достиг сервера;
- времени, которое требуется серверу для вычисления контрольной суммы вашего оператора и выяснения, нет ли уже соответствующего плана исполнения в кэше;
- в случае отсутствия плана исполнения в кэше – времени на синтаксический разбор оператора и определение плана исполнения;
- реального времени выполнения оператора, включающего процессорное время и, возможно, время ожидания ввода-вывода или время ожидания ресурса, который нельзя использовать совместно;
- времени на получение данных от сервера или получения информации о выполнении, например, кода возврата и количества обработанных строк.

Путем настройки операторов на единичной основе вы улучшаете только четвертый компонент. Если мы стараемся избавиться от ненужных вызовов, как мы делали в более раннем примере, улучшая функции просмотра, мы можем удалить в цикле выполнение пяти компонентов значительное число раз. Если мы избавляемся от цикла, мы избавляемся сразу от всех накладных расходов.

Теперь, когда мы знаем главные операторы, тормозящие нашу СУБД, нам нужно обратить внимание на несколько следующих моментов:

- Если операторы, которые являются (глобально) самыми тормозящими, представляют собой быстрые операторы, выполняемые очень

большое количество раз, у вас есть большие шансы на успех рефакторинга, и оценка возможных улучшений не очень сложна. Если у вас есть статическая таблица, например `reference lookup table`, для которой среднее количество запросов за активный сеанс больше, чем количество строк в таблице, в среднем каждая строка вызывается несколько раз. Очень часто слово *несколько* означает не два или три раза, а сотни раз и более. Вы можете быть уверены, что сможете уменьшить количество вызовов и соответственно сократить процессорное время, связанное с каждым исполнением. Вы обычно находите вызовы этого типа, когда выполняете преобразования, конвертирование валют или календарные вычисления, а также некоторые другие операции. Как минимизировать количество вызовов, мы обсудим более детально в третьей главе.

- Если у вас есть какие-то очень ресурсоемкие операторы, предсказать экономию за счет переписывания оператора лучшим способом значительно сложнее; если стратегия индексирования, которую мы будем обсуждать во второй главе, правильна и если, как вы увидите в третьей главе, нет сложных представлений, которые можно упростить, ваш успех здесь сильно зависит от вашей квалификации в SQL. Я надеюсь, что после того как вы прочитаете пятую главу, ваши шансы на значительное усовершенствование плохо написанных операторов станут выше на 40–60%.
- Когда вы видите несколько сходных обновлений, применяемых к одним и тем же таблицам, и при этом неплохо знаете SQL, у вас тоже есть очень хорошие шансы объединения некоторых из них, что мы будем обсуждать в шестой главе. Делая в одном операторе то, что раньше делали два оператора, вы запросто увеличите производительность вдвое. К сожалению, иногда вы не можете сделать этого. Уверенно можно предположить возможность улучшения лишь в 20–25% случаев.

Когда вы можете описать, какие операторы загружают сервер, и можете для каждого из них предположить вероятность улучшения в терминах процессорного времени, у вас будут цифры, которые вы сможете сравнить с параметрами более быстрого оборудования, – но никогда не обсуждайте время отклика! Процессорное время является только одним компонентом времени отклика, и вы можете значительно уменьшить время отклика, устранив все задержки, связанные с взаимодействием приложения и СУБД. Теперь проблемой для вас станет определение того, какой ценой вы получите эти улучшения. Следующие главы помогут вам понять, что именно вы получите и какой ценой.