

СТЕФАН ФАРО

РЕФАКТОРИНГ

SQL приложений



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®

H I G H T E C H

Рефакторинг SQL-приложений

Стефан Фаро и Паскаль Лерми



Санкт-Петербург — Москва
2009

Серия «High tech»
Стефан Фаро и Паскаль Лерми
Рефакторинг SQL-приложений

Перевод Ф. Гороховского

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

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 эффективных 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 ЛПН N 000054 от 25.12.98.

Подписано в печать 31.03.2009. Формат 70×100¹/16. 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	9
Предисловие	10
1. Оценка	20
Простой пример	21
Настройка SQL, традиционный способ	25
Припудривание кода	28
Настройка SQL заново	30
Рефакторинг, первая точка зрения	30
Рефакторинг, вторая точка зрения	35
Сравнение и комментарии	37
Выбор среди различных подходов	40
Оценка возможных выигрышей	42
Выяснение, что делает база данных	49
Запрос динамических представлений	49
Добавление операторов в файл трассировки	54
Использование файлов трассировки	56
Анализ собранного материала	58
2. Проверка работоспособности	61
Статистика и проблемы с данными	62
Доступная статистика	62
Ловушки для оптимизатора	67
Экстремальные значения	67
Временные таблицы	69
Обзор индексирования	69
Краткий обзор типа индексации	71
Детальное исследование	73
Индексы, которые нарушают правила	79
Индексы на основе битовых карт	79
Кластерные индексы	80
Индексы по выражениям	81
Синтаксический разбор и связующие переменные	81
Как определить проблемы синтаксического разбора	82

Оценка потерь производительности из-за синтаксического разбора	84
Разрешение проблем синтаксического анализа	88
Что если одно значение должно быть привязано несколько раз?	90
Разрешение проблем синтаксического анализа для ленивых	91
Правильный подход к разрешению проблем синтаксического анализа . . .	92
Обработка списков в подготовленных операторах	94
Передача списка как одной переменной	95
Пакетная обработка списков.	98
Использование временной таблицы	99
Групповые операции	100
Управление транзакциями	102
3. Пользовательские функции и представления	105
Пользовательские функции	106
Усовершенствование чисто вычислительных функций	107
Дальнейшее усовершенствование функций.	110
Усовершенствование функций поиска.	118
Пример 1: календарная функция	119
Пример 2: функция конвертирования валют	127
Усовершенствование функций против переписывания операторов.	135
Представления.	136
Для чего нужны представления	136
Сравнение производительности со сложными представлениями и без них	137
Рефакторинг представлений.	144
4. Концепция тестирования	148
Генерирование тестовых данных	149
Размножение строк	150
Использование функций генерирования случайных значений	151
SQL Server и функции генерирования случайных значений	152
Подгонка под существующие распределения	156
Генерирование большого числа строк	160
Целостность на уровне ссылок	164
Генерирование случайного текста	165
Сравнение альтернативных версий	167
Блочное тестирование	167
Приближенное сравнение	168
Сравнение таблиц и результатов	169
Что сравнивать	169
Примитивные способы сравнения	170
Сравнение SQL, версия из учебника.	172
Сравнение SQL, версия получше	174

Сравнение контрольных сумм в SQL	175
Ограничения сравнения	182
5. Рефакторинг операторов.	183
Планы исполнения и директивы оптимизатора	184
Анализ медленного запроса	190
Идентификация базового запроса	190
Приведение в порядок фразы from	192
Рефакторинг базового запроса	195
Анализ составных частей	196
Устранение повторяющихся шаблонов	196
Игры с подзапросами.	203
Подзапросы в списке select	203
Подзапросы во фразе from	206
Подзапросы во фразе where	206
Ранняя активизация фильтров.	209
Упрощение условий.	211
Другие направления оптимизации.	214
Упрощение агрегатов	214
Использование фразы with	214
Комбинирование операторов объединения	215
Перестроение исходного запроса	216
Вложенные циклы.	216
Соединение слиянием и хеш-соединение.	217
6. Рефакторинг задач	219
SQL-мышление	220
Использование SQL там, где SQL работает лучше	221
Рассчитывайте на успех	222
Реструктуризация кода.	225
Объединение операторов.	226
Введение управляющих структур в SQL	226
Использование агрегатов	227
Использование функции coalesce() вместо if ... is null	228
Использование исключений	229
Извлечение всех нужных данных за один прием	234
Изменение логики	235
Избавление от функции count()	236
Избегайте излишеств.	244
Избавляйтесь от циклов	245
Причины использования циклов	248
Анализ циклов.	250
Сомнительные циклы	251

7. Рефакторинг потоков и баз данных	255
Реорганизация обработки	256
Борьба за ресурсы	257
Время обслуживания и интенсивность входного потока	258
Усиление параллелизма	259
Размножение поставщиков услуг на уровне приложения	261
Укорачивание критических разделов	269
Изолирование опасных зон	270
Работа с несколькими очередями	271
Параллелизм вашей программы и СУБД	278
Потрясая основы	281
Сортировка строк	283
Разбиение таблиц	286
Изменение столбцов	288
Изменение содержимого	288
Разбиение столбцов	289
Добавление столбцов	290
Материализация представлений	291
8. Как это работает: практика рефакторинга	294
Можете ли вы взглянуть на базу данных?	294
«Мертвые» запросы	296
Все эти быстрые запросы	298
Не бывает явно «плохих» запросов	300
Пора заканчивать	301
A. Сценарии и примеры программ	302
B. Инструменты	312
Программы mklipsum и lipsum	312
Как собрать программы mklipsum и lipsum	312
Как использовать программы mklipsum и lipsum	312
Roughbench	315
Как собрать программу Roughbench	315
Как использовать программу Roughbench	315
Файл roughbench.properties	315
Указание параметров	316
Генерирование переменных	317
Генерирование целых чисел или чисел с плавающей запятой	317
Генерирование дат	317
Вывод	318
Алфавитный указатель	320

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 приступить 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 Фаулер¹ – 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 самозванных 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 запираяться 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, используя 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 А «Сценарии и примеры программ», и приложение В «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 бесплатный 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*:

```
1  try {
2      long txid;
3      long accountid;
4      float amount;
5      String curr;
6      float conv_amount;
7
8      PreparedStatement st1 = con.prepareStatement("select accountid"
9          + " from area_accounts"
10         + " where areaid = ?");
11      ResultSet rs1;
12      PreparedStatement st2 = con.prepareStatement("select txid,amount,curr"
13          + " from transactions"
14          + " where accountid=?"
15          + " and txdate >= date_sub(?, interval 30 day)"
16          + " order by txdate");
17      ResultSet rs2 = null;
18      PreparedStatement st3 = con.prepareStatement("insert into check_log(txid,"
19          + " conv_amount)"
20          + " values(?,?)");
21
22      st1.setInt(1, areaid);
23      rs1 = st1.executeQuery();
24      while (rs1.next()) {
25          accountid = rs1.getLong(1);
26          st2.setLong(1, accountid);
27          st2.setDate(2, somedate);
28          rs2 = st2.executeQuery();
29          while (rs2.next()) {
30              txid = rs2.getLong(1);
31              amount = rs2.getFloat(2);
32              curr = rs2.getString(3);
33              if (AboveThreshold(amount, curr)) {
34                  // Конвертация
35                  conv_amount = Convert(amount, curr, valuationdate);
36                  st3.setLong(1, txid);
37                  st3.setFloat(2, conv_amount);
38                  dummy = st3.executeUpdate();
39              }
40          }
41      }
42      rs1.close();
43      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 = ?");

        ResultSetsrs;
        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 MySQL¹ 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 Oracle¹.

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
Execute	252	0.00	0.01	0	0	0
Fetch	11903	32.21	32.16	0	2163420	117676
total	12156	32.22	32.18	0	2163420	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 Waited	Max. Wait	Total Wait
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 Wait
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 требуется 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, сообщаящий, 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.areaaid = ?"
+ "          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*.

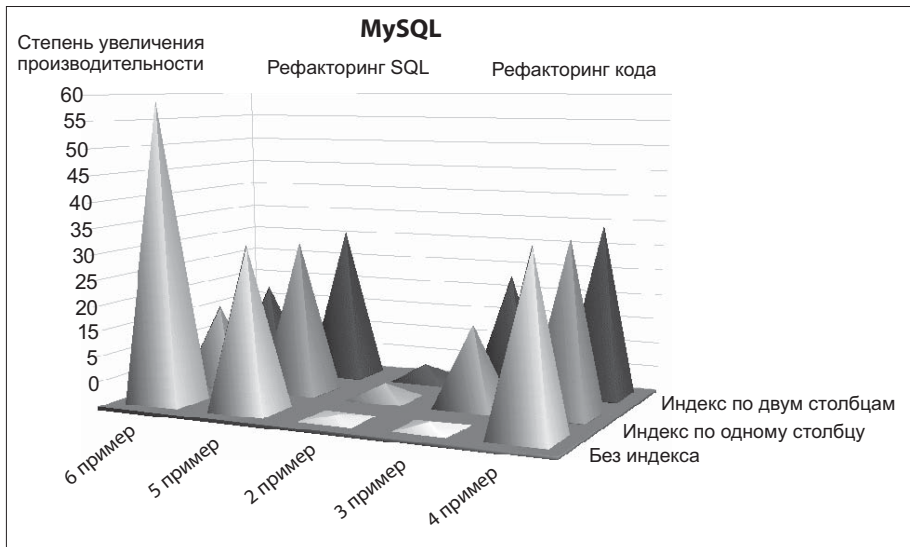


Рис. 1.1. Результаты рефакторинга для MySQL

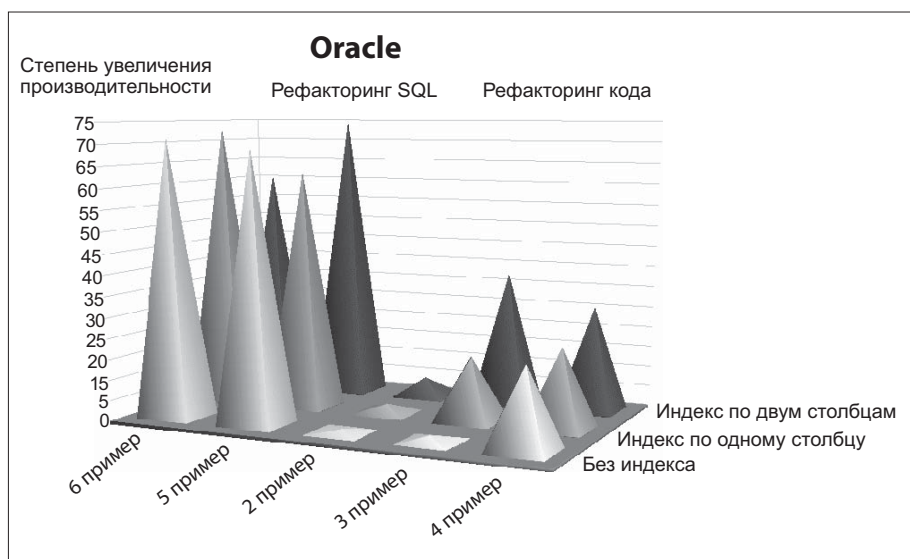


Рис. 1.2. Результаты рефакторинга для Oracle

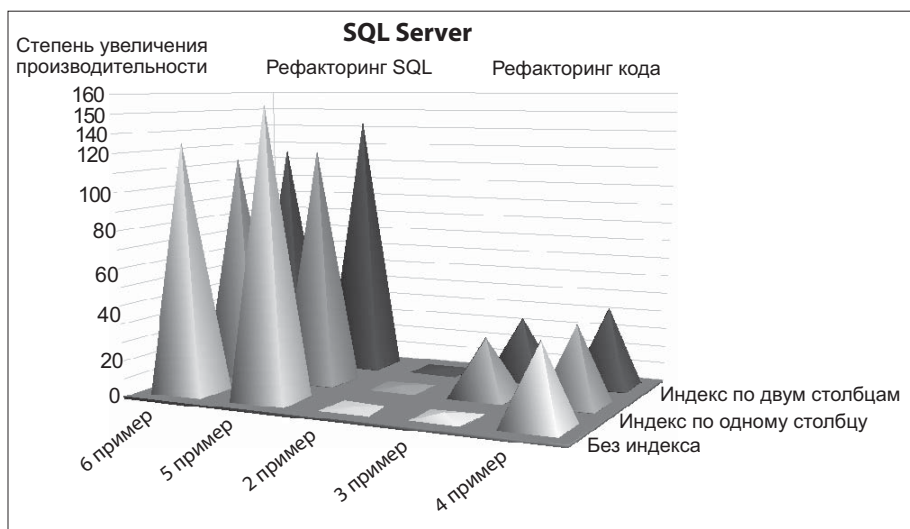


Рис. 1.3. Результаты рефакторинга для SQL Server

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 рискованно¹. 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 отрицательным¹ (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.

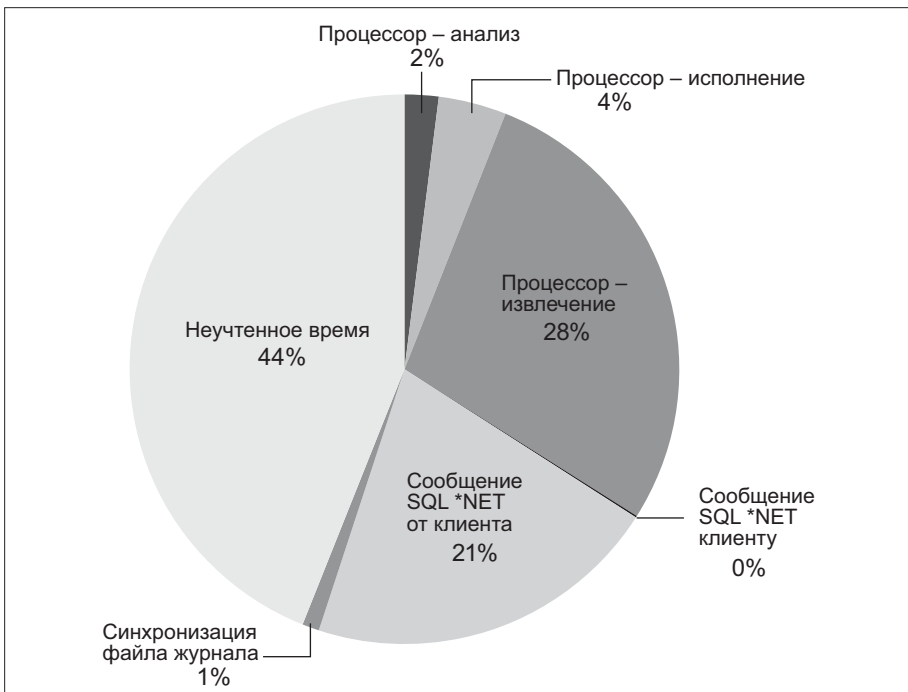


Рис. 1.4. Как в Oracle распределялось время в первой версии

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

результатирующего 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 словарию 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 просмотра 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 ограничены, 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 «болтливости» 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 thresh- olds where iso=:1	31 029	3.51	3.38	30 301
select :1 * rate from cur- rency_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$sqlsession` в Oracle, `sys.dm_exec_requests` в SQL Server и `information_schema.processlist` в MySQL, позволяют вам видеть, что выполняется в данный момент (или только что было исполнено). Однако отбор этих представлений не будет убедительным, если вы не знаете либо скорости, с которой эти запросы выполняются, либо их продолжительности. Действительно интересны те динамические представления, которые отображают кэш оператора, со счетчиками числа исполнений каждого оператора, процессорным временем, затраченным каждым оператором, количеством базовых единиц данных (именуемых блоками (*block*) в Oracle и страницами (*page*) в других СУБД), связанных с оператором, и т. п. Количество страниц данных, к которым был осуществлен доступ при выполнении запроса, обычно называется *logical read* и является хорошим индикатором объема работы, требуемой для выполнения оператора. Релевантными динамическими представлениями являются `v$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 построить 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 позволяют 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 set 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 *rbspy*, 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 проху (например, MySQL Proху 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 *mysqsla*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 SQLite¹: 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.

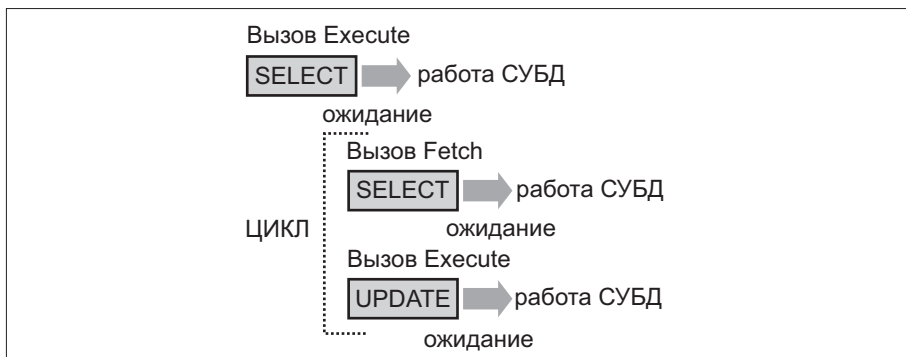


Рис. 1.5. «Разговорчивое» приложение SQL

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.

2

Проверка работоспособности

*И вам не удастся разузнать,
Зачем он распаляет эту смуту,
Терзающую дни его покоя
Таким тревожным и опасным бредом?*

Вильям Шекспир (1564–1616)
Гамлет, III акт, сцена 1, перевод М. Л. Лозинского

Прежде чем приступать к серьезной работе по рефакторингу, вам нужно уточнить несколько моментов, особенно если имеется значительное количество больших, ресурсоемких запросов. Иногда грубые ошибки, которые легко исправить, остаются незамеченными из-за недостатка времени, знаний или просто непонимания того, как работает СУБД. В этой главе мы будем обсуждать несколько моментов, за которыми вы должны следить и при необходимости исправлять, прежде чем вы начнете измерять производительность для рефакторинга. При определенной удаче (а я, например, довольно ленив) рефакторинг может впоследствии показаться не столь ужасным процессом.

Одним из самых критических аспектов производительности СУБД, касающихся отдельных запросов, является эффективность оптимизатора запросов (к сожалению, оптимизатор мало что может сделать с плохими алгоритмами). Поиск оптимизатором лучшего плана исполнения обычно базируется на информации о существующих индексах и статистике (имеющейся в словаре данных), сбор которой должен быть постоянной частью администрирования базы данных.

Есть две вещи, которые нужно проверить прежде всего, когда вы обнаруживаете проблемы с производительностью на конкретных запросах: является ли статистика достаточно свежей и детальной и все ли в порядке

с индексированием. Это два необходимых условия, чтобы оптимизатор мог правильно выполнить свою работу (выполняет ли оптимизатор свою работу правильно, когда индекс и статистика не соответствуют требованиям, – это другой вопрос). Сначала мы рассмотрим тему статистики, включая статистику индексов, затем обсудим сами индексы. Порядок может удивить вас, поскольку множество людей ставит знак равенства между хорошей производительностью **SQL** и **правильным использованием индексов**. Только хорошая статистика может дать оптимизатору возможность принять решение использовать правильный индекс или, наоборот, не использовать плохой. В некотором смысле статистика – обратная сторона индексов.

Статистика и проблемы с данными

Словом «статистика» обозначается общий термин, относящийся к множеству данных, которые сохраняет СУБД в зависимости от того, что в ней хранится. Пространство, требуемое для хранения статистики, ничтожно, чего не скажешь о времени, которое нужно для ее накопления. Сбор статистики может быть либо автоматизирован, либо запускаться администратором. Поскольку сбор детальной статистики может быть очень ресурсоемкой операцией (в худшем случае вам придется сканировать всю вашу базу данных), вы можете собирать выборочную статистику, вычисленную или оценочную. Разные СУБД собирают несколько различающуюся информацию.

Доступная статистика

Было бы довольно скучно рассматривать детально всю статистику, собираемую различными продуктами, которая может оказаться обширной в случае **Oracle** и **SQL Server**. **Давайте просто обсудим, какая информация может оказаться полезной для оптимизатора при рассмотрении альтернативных планов исполнения.**

Во-первых, вы должны понимать, что SQL-машина должна проделать большую работу с физическим хранилищем данных. Когда СУБД сохраняет данные, она выделяет хранилище для таблицы, а это процесс не непрерывный, а дискретный: большой фрагмент файла резервируется для конкретной таблицы, и когда этот участок заполняется, выделяется новый. Выделение пространства происходит не при каждом добавлении строки. Как следствие, данные, которые принадлежат таблице, физически разбиты на кластеры. Их размер зависит от носителя, на котором находится ваше хранилище (если вы используете дисковые массивы, таблица не будет так фрагментирована, как в случае единственного раздела на одном диске). Но когда СУБД сканирует таблицу, она считывает за один прием большой объем данных, поскольку велика вероятность, что в том же фрагменте данных, где хранится требуемая строка, находятся и строки, которые потребуются следующими.

Напротив, индекс связан по ключу со строками, не обязательно расположенными рядом друг с другом (если это не кластерный индекс SQL Server, индекс-таблицы в Oracle или подобный тип хранилища, с которым СУБД умеет работать). Когда СУБД извлекает данные, адресуемой единицей является страница (в Oracle именуемая блоком) размером в несколько килобайт. Каждый элемент индекса, связанный со значением ключа, сообщает нам, что в одной странице есть строка, соответствующая ключу, но весьма вероятно, что этому ключу не будет соответствовать больше ни одна строка. Выборки будут происходить построчно значительно чаще, чем в случае полного сканирования. Капля индексов будет хороша для относительно малого числа строк, но не обязательно так же хороша, как море полных сканирований для большого количества строк.

В результате при фильтрации строк и применении оператора `where` главной заботой оптимизатора при принятии решения, может ли индекс быть полезен, является избирательность – другими словами, какая доля строк удовлетворяет данному критерию. Это требует знания двух чисел: количества строк в таблице и количества строк, удовлетворяющих критерию поиска.

Например, условие типа этого можно оценить более или менее легко:

```
where my_column = некоторое значение
```

Когда столбец `my_column` проиндексирован, и если индекс создан как уникальный, нам не нужна статистика: условие будет удовлетворено либо одной строкой, либо отсутствием строк. Несколько интереснее случай, когда столбец проиндексирован неуникальным индексом. Чтобы оценить избирательность, нам приходится опираться на такую информацию, как количество различных ключей в индексе. Например, у китайцев фамилия будет одной из приблизительно трех тысяч, поскольку в Китае, одну фамилию носят, в среднем, 400 000 человек. Если вы думаете, что 1:3000 является избирательным соотношением, вы можете сравнить китайские фамилии с французскими, которых примерно 1,3 миллиона (в том числе иностранного происхождения) при населении около 60 миллионов человек, или 46 человек на фамилию в среднем.

Избирательность – это серьезная проблема. Когда избирательность низка, преимущества доступа к индексам путем сканирования простой таблицы сомнительны. Чтобы проиллюстрировать этот момент, я создал таблицу из пяти миллионов строк с пятью столбцами – C0, C1, C2, C3 и C4 – и заполнил столбцы случайными значениями соответственно от 1 до 5, от 1 до 10, от 1 до 100, от 1 до 1000 и от 1 до 5000. Затем я сделал запросы на выборку строк с одним конкретным значением для каждого столбца, каждый из которых вернул от 1 000 000 до 1000 строк в зависимости от того, какой столбец был выбран в качестве критерия поиска. Я запускал эти запросы несколько раз: первый раз на неиндек-

сированной таблице, а затем после того, как проиндексировал каждый столбец¹. Кроме того, я отключил вычисление статистики.

В табл. 2.1 показан рост производительности, который я получил с помощью индексов. Значение представляет собой соотношение времени работы запроса при неиндексированном столбце ко времени работы запроса при индексированном столбце; значение меньше единицы указывает на то, что извлечение строк при наличии индекса происходило дольше, чем при его отсутствии.

Таблица 2.1. Рост производительности благодаря индексированию

Строк возвращено	MySQL	Oracle	SQL Server
~ 1 000 000	x0.6	x1.1	x1.7
~ 500 000	x0.8	x1.0	x1.5
~ 50 000	x4.3	x1.0	x2.1
~ 5 000	x39.2	x1.1	x20.1
~ 1 000	x222.3	x20.8	x257.2

Единственным продуктом в этом примере, для которого поиск по индексу всегда был быстрее, чем полное сканирование, оказался SQL Server. Для MySQL в этом примере почувствовать позитивный эффект от индексирования можно только тогда, когда запрос возвращает меньше 1% строк. Если запрос возвращает 10% строк или больше, эффект индексирования отрицательный. Для Oracle наличие индекса или его отсутствие – почти несущественно до тех пор, пока количество возвращаемых строк не снизится до 0,02%. Даже с SQL Server мы не достигаем ускорения на порядок (десятикратного), пока количество возвращаемых строк не уменьшится до очень скромной доли от размера таблицы.

Не существует магического процента, который можно было бы вставить сам по себе в четкое и краткое правило, типа «в Oracle, если будет возвращено количество строк, меньшее, чем этот *магический процент*, должен быть индекс». Влиять могут многие факторы, например относительный порядок строк в таблице (индекс по столбцу, содержащему значения, возрастающие по мере вставки строк, например, более эффективен для запроса, чем индекс подобной избирательности по столбцу со случайными значениями).

Избирательность важна, но, к сожалению, избирательность – это еще не все. Я заполнил мою таблицу из пяти миллионов строк случайно сгенерированными значениями, а генераторы случайных значений создают равномерно распределенные числа. В реальной жизни распределения не столь равномерны. Предыдущее сравнение распространенных фамилий в Китае

¹ Только обычные индексы, никаких кластерных индексов.

и во Франции может служить примером: даже во Франции за средним значением прячутся впечатляющие диспропорции. Для француза по фамилии Мартэн найдется свыше 200 000 соотечественников-однофамильцев (не считая англосаксов); даже фамилию Мерсье носят больше 50 000 французов. Конечно, это скромные числа по сравнению с фамилиями Ли, Ван и Джан, которые вместе представляют свыше 250 миллионов китайцев.

То, что вы не можете быть уверенными в равномерном распределении значений, означает, что когда вы ищете какое-то одно значение, индекс приведет вас прямо к тому результирующему набору, который вам нужен, а когда вы ищете другие значения, он будет не настолько эффективен. Оценка эффективности индекса по отношению к определенному значению особенно важна, когда у вас есть несколько критериев и несколько подходящих индексов, из которых ни один нельзя назвать явным фаворитом в условиях средней избирательности. Чтобы помочь оптимизатору сделать обоснованный выбор, вам может потребоваться собрать гистограммы. Чтобы вычислить гистограммы, СУБД делит диапазон значений для каждого столбца на несколько категорий (одна категория для каждого значения, если у вас не более пары сотен различных значений, – в противном случае фиксированное число) и подсчитывает, сколько значений попадает в каждую категорию. Таким образом, оптимизатор знает, что когда вы ищете людей с определенной фамилией, родившихся в промежутке между двумя датами, он скорее будет использовать индекс по фамилии (если она редкая), а когда вы будете искать другую фамилию при том же диапазоне дат, он будет использовать индекс по дате, поскольку фамилия широко распространена. Вот, в сущности, идея гистограмм. Позже из этой главы вы узнаете, впрочем, что гистограммы сами могут косвенно создавать проблемы с производительностью.

Кроме количества строк в таблице, средней избирательности индексов и того, как значения распределены, могут быть очень полезны и другие сведения. Я уже упоминал связь между порядком индексных ключей и порядком строк в таблице (которая не может быть сильнее, чем при наличии кластерного индекса в SQL Server или MySQL). Если эта связь сильна, индекс будет эффективным при поиске по диапазону ключей, поскольку соответствующие строки будут кластеризованы. Многие другие сведения статистической природы могут быть полезны при рассмотрении оптимизатором альтернативных планов исполнения и оценке количества строк (или *кардинального числа*), остающихся после каждого последовательного выполнения условий отбора. Вот список некоторых таких значений:

Количество нулевых значений

Когда нулевые значения в индексе не сохраняются (как в случае Oracle), информация о том, что большинство значений в столбце являются нулевыми¹, говорит оптимизатору: когда столбец упоминается

¹ Что может указывать на неправильное проектирование таблицы.

как имеющий данное значение, условие, вероятно, очень избирательно, даже если в индексе всего несколько различных значений ключа.

Диапазон значений

Знание того, в каком диапазоне расположены значения в столбце, очень полезно при попытке оценить, сколько строк может быть возвращено при использовании операторов `>=`, `<=` или `between`, особенно при отсутствии гистограмм. Условие равенства значения, которое выпадает из известного диапазона, скорее всего также очень избирательно. Однако когда столбцы проиндексированы, минимальное и максимальное значение легко можно выяснить из индекса.

Проверка присутствия статистики не представляет сложности. Например, в Oracle вы можете запустить под SQL*Plus примерно такой сценарий:

```
col "TABLE" format A18
col "INDEX" like "TABLE"
col "COLUMN" like "TABLE"
col "BUCKETS" format 999990
break on "TABLE" on "ANALYZED" on sample_size on "ROWS" on "INDEX" on "KEYS"
select t.table_name "TABLE",
       to_char(t.last_analyzed, 'DD-MON') "ANALYZED",
       t.sample_size,
       t.num_rows "ROWS",
       i.index_name "INDEX",
       i.distinct_keys "KEYS",
       i.column_name "COLUMN",
       i.num_distinct "VALUES",
       i.num_nulls "NULLS",
       i.num_buckets "BUCKETS"
from user_tables t
     left outer join (select i.table_name,
                           i.index_name,
                           i.distinct_keys,
                           substr(ic.column_name, 1, 30) column_name,
                           ic.column_position pos,
                           c.num_distinct,
                           c.num_nulls,
                           c.num_buckets
                       from user_indexes i
                           inner join user_ind_columns ic
                                on ic.table_name = i.table_name
                                and ic.index_name = i.index_name
                           inner join user_tab_columns c
                                on c.table_name = ic.table_name
                                and c.column_name = ic.column_name) i
    on i.table_name = t.table_name
order by t.table_name,
       i.index_name,
       i.pos
/
```

Или в MySQL (где проверять нужно значительно меньше) вы можете запустить следующий сценарий:

```
select t.table_name,
       t.table_rows,
       s.index_name,
       s.column_name,
       s.cardinality
from information_schema.tables t
     left outer join information_schema.statistics s
       on s.table_schema = t.table_schema
        and s.table_name = t.table_name
where t.table_schema = schema()
order by t.table_name,
         s.index_name,
         s.seq_in_index;
```

В SQL Server процесс значительно более неуклюжий, там вы должны сначала вызвать `sp_helpstats` с именем вашей таблицы и 'ALL', а затем запустить `dbcc show statistics` для каждой статистики, имя которой было возвращено предыдущей хранимой процедурой.

Но вот что может быть интересно – проверить, нет ли аномалий, которые могут заставить оптимизатор запросов ошибиться.

Ловушки для оптимизатора

Неравномерное распределение значений в столбце – причина, по которой оптимизатор вычисляет стоимости исполнения из неверных предпосылок и, следовательно, выбирает неправильный план исполнения. Но могут быть и другие, менее очевидные случаи, когда оптимизатор сбивается с пути.

Экстремальные значения

Бывает полезным проверить диапазон значений в индексированном столбце, что легко сделать следующим образом:

```
select min(column_value), max(column_value)
from my_table;
```

Когда индекс есть, максимальное и минимальное значения могут быть считаны из индекса очень быстро. Такой запрос может привести к некоторым интересным открытиям. В Oracle, например, мне чаще, чем хотелось бы, доводится находить в столбцах с датами значения, ссылающиеся на первый век от Рождества Христова, поскольку люди по ошибке вводят даты в формате ММ/ДД/YY, когда Oracle ожидает ввода дат в формате ММ/ДД/YYYY (эта проблема значительно менее вероятна в SQL Server, для которого в типе `datetime` исчисление начинается с 1753 года, или в MySQL, который принимает даты, начинающиеся с 1 января 1000 года). В числовых столбцах могут встречаться значения типа 99999999, используемые для того, чтобы обойти то обстоятельство,

что столбец обязательный, а значение неизвестно (очевидная проблема проектирования) и т. п. Имейте в виду, что когда недоступна никакая другая информация, оптимизатор предполагает, что значения распределены равномерно между минимальным и максимальным, и может сильно недооценить или переоценить количество строк, возвращаемых при сканировании диапазона (условие неравенства). В результате он может принять решение использовать индекс в тех случаях, когда этого делать не следовало бы, или совершить противоположную ошибку. Коррекция бессмысленных значений (или неверных данных) является первым шагом, который может вернуть оптимизатор на путь истинный.

Что верно для минимальных значений, верно и для максимальных. Нередко очень далекие даты используют в столбце с датами для обозначения «не истекшей» или «текущей» даты. Имеет смысл также проверить аномалии, о которых экстремальные значения ничего не скажут. Например, два программиста могут использовать разные стандарты для «далеких дней»: один использует 31 декабря 2999 года, а другой, смотрящий значительно дальше, использует 9999 год. Если вы когда-нибудь встретитесь с таким случаем, ваш оптимизатор сойдет с ума от такого диапазона дат. Даже с гистограммами два различных пика далеких дат могут дать оптимизатору неверные представления о распределении. Когда оптимизатор встречается значение, на которое нет точной ссылки в гистограмме, он должен экстраполировать его распределение с помощью непрерывной функции. А очень неравномерное распределение может привести к экстраполяции, выходящей за границы. Нет необходимости говорить, что ошибки в программе просто ждут, когда один разработчик использует одно значение, а другой – другое. Вы легко можете протестировать такой случай, запустив что-нибудь подобное (к сожалению, это ресурсоемкий запрос, который стоит запускать только на копии рабочих данных):

```
select extract(year from date_column), count(*)
  from my_table
 group by extract(year from date_column)
```

Даже если в вашей базе данных используется только одна дата из далекого будущего, использование такой удаленной даты для обозначения текущего значения требует гистограмм. Иначе, как только данные накопятся и прошедшие исторические значения будут иметь большой вес в вашей базе данных, каждое сканирование диапазона прошедших значений – то есть запрос, содержащий примерно следующее:

```
where expired_date < now()
```

при отсутствии гистограммы будет приводить к сканированию базы данных, как если бы он возвращал, скажем, десятилетие в тысяче или в двух тысячах лет от нашего времени. Если бы у вас действительно были даты, отстоящие более чем на тысячу лет, десятилетие представляло бы порядка 1% от всех строк, когда вы на самом деле хотите вернуть, возможно, 99% от всех строк вашей таблицы. Оптимизатор

может решить использовать индекс, когда полное сканирование таблицы могло бы быть более действенным, или выбрать индекс по столбцу даты вместо другого, более эффективного, индекса.

Временные таблицы

Стоит также подчеркнуть тот факт, что временные таблицы, содержание которых, по определению, очень изменчиво, также вносят свой вклад в усложнение задачи оптимизатора. Это особенно справедливо, когда единственный признак их временной природы – использование суффикса или префикса в их именах и когда они создаются не как временные таблицы, но (важный нюанс) используются как временные. К сожалению, полезные намеки в имени полностью теряются в оптимизаторе. Как следствие, любая таблица, созданная как обычная, и рассматривается как обычная таблица, вне зависимости от ее имени, и к ней применяются основные правила. Если, например, статистика была вычислена, когда таблица была пустой, оптимизатор будет думать, что в таблице очень мало или совсем нет строк, вне зависимости от того, что происходит. С этим можно бороться, указав СУБД держать таблицу под постоянным наблюдением, как это можно сделать в Oracle, где вы указываете динамическое квантование, или в SQL Server с автоматической статистикой, но это вызывает очевидные накладные расходы.

В некоторых продуктах временные таблицы видны только из сеанса, в котором они были созданы; следовательно, простое присутствие таблицы, содержащей в своем имени TMP или TEMP, в `information_schema.tables` заслуживает исследования (фактически, использование временных таблиц само по себе заслуживает исследования; это использование можно оправдать, но иногда оно маскирует недостатки SQL).

Обзор индексирования

Проверка того, правильно ли проиндексированы таблицы, – это, вероятно, один из первых шагов, которые вы должны выполнить. Результаты, приведенные в табл. 2.1, должны были убедить вас отбросить простое правило «если производительность низка, нужно добавить индексы». Фактически (и это может оказаться сюрпризом!) индексы чаще избыточны, чем недостаточны.

Обычно первая задача для улучшения производительности приложения – приведение в порядок индексов. В этом разделе я приведу несколько примеров, которые помогут вам лучше понять, как работают индексы и как вы можете определить, что они требуют пересмотра.

Во-первых, существуют два типа индексов: индексы, которые существуют как результат проектирования базы данных, и индексы производительности.

Индексы, возникшие при проектировании базы данных, усиливают семантику – это индексы по первичным ключам и столбцам с уникальными

значениями. Когда вы определяете в таблице ограничение, что один набор столбцов однозначно идентифицирует строку в столбце, вы приказываете СУБД при каждой вставке строк проверять, что строка с таким ключом еще не существует, и возвращать ошибку, если ключ дублирует уже существующий. Проверка присутствия ключа в дереве в процессе вставки строки очень эффективна, и случается, что индексы оказываются древовидными структурами. Поскольку столбцы, которые однозначно идентифицируют строку, часто используются для поиска и возвращения строки, такие индексы еще и выполняют роль весьма удобных индексов производительности, что к лучшему. Некоторые продукты (например, InnoDB-машина в MySQL) также требуют наличия индекса по внешним ключам, чего ни в Oracle, ни в SQL Server нет. Хотя индексирование внешних ключей обычно рекомендуется как «хорошая практика», я не разделяю точку зрения, что все внешние ключи нужно обязательно индексировать. Прежде всего, во многих случаях внешний ключ является первым столбцом в первичном ключе, и он уже проиндексирован индексом первичного ключа. Но реальный вопрос состоит в том, какая таблица должна быть ведущей в запросе – таблица, содержащая внешний ключ, или ссылающаяся таблица, – поскольку требования к индексированию в каждом случае разные. Запросы мы будем обсуждать значительно подробнее в пятой главе. Пока что просто скажу, что любой индекс по внешнему ключу, не являющийся обязательным, должен считаться индексом производительности.

Индексы производительности – это индексы, которые создаются для ускорения поиска; они включают обычные индексы на основе В-дерева¹, и индексы других типов, например индексы на основе битовых карт и хеш-индексы, которые вы можете встретить повсюду (я не включил в это обсуждение полнотекстовые индексы, которые могут быть очень интересны с точки зрения производительности, но находятся несколько вне ядра СУБД). Для упрощения я буду впредь говорить об обычных индексах.

Мы видели, что обычные индексы иногда ускоряют поиск, иногда замедляют, а иногда никак не влияют на скорость. Поскольку они обрабатываются в режиме реального времени и поскольку каждая вставка строки, удаление строки или обновление индексированного столбца вызывает вставку, удаление или обновление индексных ключей, любой индекс, который не ускоряет поиск, фактически ведет к уменьшению общей производительности. Потери становятся более серьезными, когда есть сильная конкуренция во время вставки, поскольку не так легко распределить конкурентные вставки по древовидному индексу, как по таблице; иногда запись в индексы становится реальным узким местом.

¹ Я не хочу вдаваться в детали структуры индекса; достаточно сказать, что В-деревья представляют собой древовидные структуры, подобные любой иерархической структуре, которая придет вам в голову, за исключением того, что они исходно спроектированы так, что могут реорганизовывать сами себя и расти медленно в глубину. Более того, время поиска одинаково для всех ключей.

Некоторые индексы необходимы, но индексировать каждый столбец просто так не стоит. На практике плохое индексирование встречается значительно чаще, чем хотелось бы.

Краткий обзор типа индексации

Проверка того, как проиндексированы таблицы в схеме, занимает несколько секунд и может дать ключ к существующим или ожидаемым проблемам с производительностью. Когда в схеме нет тысяч таблиц, запуск следующего запроса может оказаться очень информативным:

```
select t.table_name,
       t.table_rows,
       count(distinct s.index_name) indexes,
       case
         when min(s.unicity) is null then 'N'
         when min(s.unicity) = 0 then 'Y'
         else 'N'
       end unique_index,
       sum(case s.columns
              when 1 then 1
              else 0
            end) single_column,
       sum(case
              when s.columns is null then 0
              when s.columns = 1 then 0
              else 1
            end) multi_column
from information_schema.tables t
left outer join (select table_schema,
                       table_name,
                       index_name,
                       max(seq_in_index) columns,
                       min(non_unique) unicity
                  from information_schema.statistics
                  where table_schema = schema()
                  group by table_schema,
                           table_name,
                           index_name) s
on s.table_schema = t.table_schema
and s.table_name = t.table_name
where t.table_schema = schema()
group by t.table_name,
         t.table_rows
order by 3, 1;
```

Если вы работаете с Oracle, вы можете предпочесть следующий запрос:

```
select t.table_name,
       t.num_rows table_rows,
       count(distinct s.index_name) indexes,
       case
```



```

        when min(s.unicity) is null then 'N'
        when min(s.unicity) = 'U' then 'Y'
        else 'N'
    end unique_index,
    sum(case s.columns
        when 1 then 1
        else 0
    end) single_column,
    sum(case
        when s.columns is null then 0
        when s.columns = 1 then 0
        else 1
    end) multi_column
from user_tables t
left outer join (select ic.table_name,
                        ic.index_name,
                        max(ic.column_position) columns,
                        min(substr(i.uniqueness, 1, 1)) unicity
                  from user_ind_columns ic,
                       user_indexes i
                  where i.table_name = ic.table_name
                        and i.index_name = ic.index_name
                  group by ic.table_name,
                           ic.index_name) s
on s.table_name = t.table_name
group by t.table_name,
         t.num_rows
order by 3, 1;

```

Этот запрос показывает для каждой таблицы количество строк, количество индексов, наличие уникального индекса, количество индексов по одному столбцу и количество многостолбцовых композитных индексов. Такой запрос не выявит все ошибки и не даст замечательных идей о том, как можно волшебным образом увеличить производительность, но может указать на некоторые возможные проблемы:

Неиндексированные таблицы

Таблицы со значительным количеством строк и без индекса обнаружить легко. У таких таблиц есть свое применение, в частности, при загрузке базы данных или крупных операциях с преобразованием данных, но они также могут таить большие ошибки.

Таблицы без уникального индекса

Таблицы без уникального индекса обычно не имеют ограничений на первичный ключ¹. Отсутствие первичного ключа может быть негативно для данных, поскольку СУБД не будет делать ничего для предотвращения вставки дублирующих строк.

¹ В некоторых редких случаях ограничения первичных ключей могут быть наложены путем использования неуникального индекса.

Таблицы с единственным индексом

Обратите внимание, что наличие только одного уникального индекса в таблице не является само по себе свидетельством хорошего проектирования. Если индекс по первичному ключу создан по одному столбцу, и в этом столбце хранятся автоматически генерируемые системой числа, наличие только одного индекса почти так же плохо, как его полное отсутствие. СУБД никоим образом не будет препятствовать вставке дублирующих строк, поскольку когда вы будете вставлять те же данные снова, она сгенерирует новое число, которое сделает новую строку отличающейся от всех остальных. Использование сгенерированного системой числа является удобным способом создания ключа для использования в качестве ссылки на строку. Однако в 99% случаев это лучше реализовать как второй уникальный индекс, основанный на столбцах, которые реально (как в «реальной жизни») позволяют вам указать, что одна строка – особенная и отличается от других.

Таблицы со многими индексами

Очень большое количество индексов редко является хорошим знаком, за исключением хранилищ данных, где это принято как стандартная практика. В обычной рабочей базе данных (в противоположность тем, которые предназначены для принятия решений) наличие индексов почти в таком же количестве, как и количество столбцов в таблице, не нужно. Скорее всего, по крайней мере некоторые из индексов являются в лучшем случае бесполезными, а в худшем – вредными. Данная ситуация требует более тщательного рассмотрения.

Таблицы с более чем тремя индексами, каждый из которых составлен по одному столбцу

Число «три» в данном случае приблизительно; если хотите, можете заменить его на «четыре». Но суть в том, что наличие только индексов по одному столбцу может быть признаком несколько примитивного подхода к индексированию. Это требует детального рассмотрения.

Нет надобности говорить, что наиболее внимательно вам нужно рассматривать те таблицы, которые снова и снова появляются в запросах, создающих значительную нагрузку на систему.

Детальное исследование

В моей практике помимо просто бесполезных индексов наиболее частые ошибки индексирования связаны с композитными индексами. Композитные индексы – это индексы, созданные сразу по нескольким столбцам. Чтобы создать индекс, СУБД сканирует таблицу, конкатенирует значения столбцов, которые вы хотите проиндексировать, называет результат «ключом», связывает с этим ключом физический адрес строки (подобно номеру страницы в предметном указателе книги), сортирует по

ключу и создает по полученному отсортированному списку дерево. Это позволяет быстро и легко получить доступ к данному ключу.

Когда два критерия всегда используются вместе, мало смысла использовать отдельные индексы, даже если оба критерия приемлемо избираемы: наличие двух индексов означает, что оптимизатору придется либо выбирать, какой индекс лучше использовать, либо использовать каждый индекс отдельно и объединять результаты, то есть возвращать только строки, связанные с обоими индексами. В любом случае это будет сложнее и, следовательно, медленнее, чем просто поиск по одному индексу (обработка индексов при операциях вставки и удаления также будет менее ресурсоемкой при меньшем количестве индексов).

Чтобы сравнить композитные индексы с индексами по одному столбцу, предположим, что мы хотим сделать запрос к таблице, в которой перечислены английские пьесы конца XVI и начала XVII веков. Эта таблица содержит такую информацию, как название произведения, его жанр (комедия, трагедия и т. п.) и год создания. Столбец с жанром может содержать нулевые значения, поскольку некоторые пьесы никогда не печатались, были утеряны и известны только из бухгалтерских книг или из дневников современников. Кроме того, во времена английского ренессанса многие пьесы были просто пересказами народных историй. Теперь предположим, что мы хотим выбрать пьесы, которые были в нашей базе данных отмечены как комедии и созданы между 1590 и 1595 годами, и что у нас есть индекс по жанру и индекс по году создания.

Выборка может выглядеть так:

	->	The Fancies Chaste и Noble	Ford
	->	A Challenge for Beautie	Heywood
	->	The Lady's Trial	Ford
comedy	->	A Woman is a Weathercock	Field
comedy	->	Amends for Ladies	Field
comedy	->	Friar Bacon и Friar Bungay	Greene
comedy	->	The Jew of Malta	Marlowe
comedy	->	Mother Bombie	Lyly
comedy	->	The Two Gentlemen of Verona	Shakespeare
comedy	->	Summer's Last Will и Testament	Nashe
... много записей ...			
comedy	->	The Spanish Curate	Massinger, Fletcher
comedy	->	Rule a Wife и Have a Wife	Fletcher
comedy	->	The Elder Brother	Massinger, Fletcher
comedy	->	The Staple of News	Jonson
comedy	->	The New Inn	Jonson
comedy	->	The Magnetic Lady	Jonson
comedy	->	A Maiden-Head Well Lost	Heywood
history	->	Richard III	Shakespeare
history	->	Edward II	Marlowe
history	->	Henry VI, part 1	Nashe, Shakespeare
....			

Этот индекс указывает на 87 спектаклей в таблице, которые помечены как комедии. Между тем индекс по году создания выглядит примерно так:

```

...
1589 -> A Looking Glass for London и England      Greene
1589 -> Titus Andronicus                          Shakespeare
1590 -> Mother Bombie                             Lyly
1591 -> The Two Gentlemen of Verona                Shakespeare
1591 -> Richard III                               Shakespeare
1592 -> Summer's Last Will и Testament             Nashe
1592 -> Edward II                                 Marlowe
1592 -> Henry VI, part 1                          Nashe, Shakespeare
1592 -> Henry VI, part 2                          Nashe, Shakespeare
1592 -> Henry VI, part 3                          Shakespeare, Nashe
1593 -> The Massacre at Paris                      Marlowe
1594 -> The Comedy of Errors                       Shakespeare
1594 -> The Taming of the Shrew                   Shakespeare
1594 -> The History of Orlando Furioso            Greene
1595 -> A Midsummer Night's Dream                 Shakespeare
1595 -> A Pleasant Conceited Comedy of George a Green Greene
1595 -> Love's Labour's Lost                      Shakespeare
1595 -> Richard II                               Shakespeare
1595 -> The Tragedy of Romeo и Juliet             Shakespeare
1596 -> A Tale of a Tub                           Jonson
...

```

Здесь у нас только 17 спектаклей, созданных между 1590 и 1595 годами, но даже беглый взгляд на названия позволяет определить, что многие из них вовсе не комедии!

СУБД может найти (из индекса по жанру) 87 ссылок на комедии, отдельно найти 17 ссылок на спектакли, созданные между 1590 и 1595 годами, и оставить только общие для обоих наборов ссылки. «Ценой» будет считывание 87 плюс 17 элементов индекса, а затем сортировка их для поиска общего подмножества.

Мы можем сравнить эту цену с ценой доступа к индексу по жанру и году создания. Поскольку год создания значительно избирательнее, чем жанр, у нас может возникнуть искушение построить композитный индекс по (creation_year, genre) в таком порядке:

```

...
1589|morality -> A Looking Glass for London и England      Greene
1589|tragedy  -> Titus Andronicus                          Shakespeare
1590|comedy    -> Mother Bombie                             Lyly
1591|comedy    -> The Two Gentlemen of Verona                Shakespeare
1591|history   -> Richard III                               Shakespeare
1592|comedy    -> Summer's Last Will и Testament             Nashe
1592|history   -> Edward II                                 Marlowe
1592|history   -> Henry VI, part 1                          Nashe, Shakespeare
1592|history   -> Henry VI, part 2                          Nashe, Shakespeare

```

1592 history	-> Henry VI, part 3	Shakespeare, Nashe
1593 history	-> The Massacre at Paris	Marlowe
1594 comedy	-> The Comedy of Errors	Shakespeare
1594 comedy	-> The Taming of the Shrew	Shakespeare
1594 tragicomedy	-> The History of Orlando Furioso	Greene
1595 comedy	-> A Midsummer Night's Dream	Shakespeare
1595 comedy	-> A Pleasant Conceited Comedy of George a Green	Greene
1595 comedy	-> Love's Labour's Lost	Shakespeare
1595 history	-> Richard II	Shakespeare
1595 tragedy	-> The Tragedy of Romeo и Juliet	Shakespeare
1596 comedy	-> A Tale of a Tub	Jonson
1596 comedy	-> The Merchant of Venice	Shakespeare
...		

При считывании этого единственного индекса СУБД знает, что вернуть, а что отбросить. Однако в данном конкретном запросе этот индекс не является самым эффективным, поскольку первый столбец представляет для всех практических применений главный ключ сортировки записей, а второй столбец является второстепенным ключом. Когда СУБД считывает индекс, дерево приводит ее к первой комедии, которая, как мы знаем, была создана в 1590 году, *Mother Bombie*. Отсюда она начинает считывать все записи, будь то комедии, трагедии или иные произведения, пока не натолкнется на первую запись с 1595 годом, которая не является комедией.

Если же мы проиндексируем по (genre, creation_year), СУБД может начать снова с *Mother Bombie*, но может закончить считывание элементов индекса, дойдя до первой комедии 1596 года. Вместо считывания 15 элементов индекса, она считает только 8 записей, которые действительно указывают на те строки, которые нам надо получить из таблицы:

comedy <null>	-> Amends for Ladies	Field
comedy 1589	-> Friar Bacon и Friar Bungay	Greene
comedy 1589	-> The Jew of Malta	Marlowe
comedy 1590	-> Mother Bombie	Lyly
comedy 1591	-> The Two Gentlemen of Verona	Shakespeare
comedy 1592	-> Summer's Last Will и Testament	Nashe
comedy 1594	-> The Comedy of Errors	Shakespeare
comedy 1594	-> The Taming of the Shrew	Shakespeare
comedy 1595	-> A Midsummer Night's Dream	Shakespeare
comedy 1595	-> A Pleasant Conceited Comedy of George a Green	Greene
comedy 1595	-> Love's Labour's Lost	Shakespeare
comedy 1596	-> A Tale of a Tub	Jonson
comedy 1596	-> The Merchant of Venice	Shakespeare
comedy 1597	-> An Humorous Day's Mirth	Chapman
comedy 1597	-> The Case is Altered	Jonson
comedy 1597	-> The Merry Wives of Windsor	Shakespeare
comedy 1597	-> The Two Angry Women of Abington	Porter
..		

Конечно, этот пример не очень показателен, но в некоторых очень больших таблицах факт, что один индекс прямо указывает на нужную запись, а другой долго думает, будет замечен. Когда в условии указан диапазон значений (например, `creation_year` между 1590 и 1595) или просто неравенство в столбце, который входит в композитный индекс, поиск происходит быстрее, если этот столбец появляется в индексе после столбца, для которого есть условие равенства (как в `genre = 'comedy'`).

Плохо то, что мы не можем делать с индексами все, что хотим. Если у нас есть индекс по (`genre`, `creation_year`) и часто выполняются запросы типа «Какие спектакли были созданы в 1597 году?», использование нашего индекса не слишком удачно. СУБД может решить просканировать индекс и перескакивать от жанра к жанру (включая неизвестный или неопределенный жанр), чтобы найти диапазон записей, содержащих 1597 год. Или благодаря более сложной структуре индексов СУБД может сделать менее ресурсоемким и поэтому более быстрым сканирование таблицы.

Поскольку данные такого типа не предполагают активного обновления, имеет смысл иметь два индекса: с одной стороны, индекс по (`genre`, `creation_year`), а с другой стороны, индекс по одному столбцу `creation_year`, чтобы быстро выполнялись все запросы – как включающие жанр, так и без него.

Основной проблемой с композитными индексами всегда является порядок столбцов. Предположим, у нас есть индекс по столбцам C1, C2 и C3, именно в таком порядке. С помощью условия следующего типа:

```
where C1 = value1
and C2 = value2
and C3 = value3
```

можно найти в словаре все слова, где первой буквой будет *к*, второй *р*, а третьей *и* – будут получены все слова до (не включая) слова *кров*. Теперь предположим, что условие такое:

```
where C2 = value2
and C3 = value3
```

Эквивалентом будет поиск в словаре всех слов с *р* и *и* на второй и третьей позициях. Такой поиск осуществить сложнее, поскольку теперь мы должны проверить все разделы от А до Я и поискать соответствующие слова.

Как последний пример, предположим такое условие:

```
where C1 = value1
and C3 = value3
```

Это то же самое, что сказать «хочу увидеть все слова, начинающиеся с буквы *к*, третьей буквой в которых является *и*». Зная первую букву, мы можем обоснованно использовать индекс, но в этом случае мы должны просмотреть все слова, начинающиеся на *к*, каждый раз проверяя третью букву. А если бы у нас были даны первая и вторая буквы, поиск был бы значительно эффективнее.

Композитный индекс подобен знанию первых букв слова для поиска в словаре: если вы не найдете значение, соответствующее первому столбцу, ваш индекс будет не слишком полезен. Если вам даны значения первого столбца, то даже если вы не знаете значений во всех столбцах, вы можете использовать ваш индекс достаточно эффективно. Степень этой эффективности зависит от избирательности столбцов, для которых есть значение.

Но пока ссылки на столбец в операторе `where` нет, вы не можете использовать столбцы, следующие за ним в индексе, даже когда на них есть ссылка в столбце `where` с условиями равенства. Многие недостатки использования индексов происходят из-за неправильного порядка, когда столбцы, содержащие важный критерий, «закопаны» под столбцами, поиск по которым происходит редко.

Поэтому при пересмотре индексов вам нужно проверить несколько вещей:

- Одностолбцовые индексы по столбцам, которые также появляются в первой позиции композитного индекса, являются избыточными, поскольку можно использовать композитный индекс — возможно, чуть менее эффективно, но все равно эффективно. Избыточное индексирование часто можно обнаружить в случаях, подобных классическому примеру учета занятий студентов: один студент может посещать несколько курсов, а несколько студентов могут заниматься на одних курсах. Следовательно, таблица учета занятий будет связывать `studentid` с `courseid`; первичным ключом для этой таблицы будет `studentid` плюс `courseid`, плюс, вероятно, семестр. Но `studentid` и `courseid` являются также и внешними ключами, соответственно указывающими на таблицы студентов или курсов. Если вы методично проиндексируете все внешние ключи, то создадите индекс по `courseid` (хорошо) и индекс по `studentid` (избыточен с первичным ключом, который начинается с этого столбца). Потерь производительности при запросах не будет, но они будут при вставке строк.
- Для создания индексов (особенно если они не определены как уникальные!), являющихся супермножеством первичного ключа, должны быть серьезные обоснования (нет оснований не делать его уникальным). Иногда такие индексы создают с тем, чтобы найти все данные, требуемые в запросе, в индексе и избавиться от доступа к самой таблице.
- Столбцы, которые *всегда* появляются в операторе `where` одновременно, должны быть проиндексированы в композитном индексе, а не в отдельных индексах по одному столбцу.
- Когда столбцы *часто* появляются вместе, то столбцы, которые могут появиться без других, в композитном индексе должны стоять первыми при условии, что они достаточно избирательны, чтобы сделать поиск по индексу более эффективным, чем полное сканирование таблицы. Если все столбцы могут появляться отдельно и являются приемлемо избирательными, имеет смысл отдельно

проиндексировать отдельные столбцы, которые не стоят на первой позиции в композитном индексе.

- Вам также нужно убедиться, что если есть условие диапазона по столбцу, которое всегда или очень часто связано с условиями равенства по другим столбцам, эти другие столбцы в композитном индексе будут первыми.
- Индексы на основе В-дерева по столбцам с малым количеством элементов бессмысленны, если значения распределены равномерно или нет гистограммы значений (поскольку тогда оптимизатор предполагает, что значения распределены равномерно). Например, если в таблице студентов есть столбец с указанием пола, обычно его нет смысла индексировать, поскольку количество студентов обоего пола будет приблизительно одинаковым; вряд ли вы захотите использовать индекс, извлекающий 50% строк. Тем не менее может быть другая ситуация – в военном училище, где женщины составляют малую часть всех курсантов, это может быть избирательным критерием. Тогда для СУБД потребуется гистограмма значений.

Индексы, которые нарушают правила

Как всегда, в общих правилах существуют исключения, в частности для определенных типов индексов.

Индексы на основе битовых карт

В Oracle, например, разрешены индексы на основе битовых карт, которые больше подходят для систем принятия решений. Индексы на основе битовых карт полезны в первую очередь для данных, которые могут масшированно вставляться, но не обновляются вообще или это происходит редко (их слабой стороной является то, что они плохо управляют конкуренцией между параллельными сеансами). Во вторую очередь – для столбцов с малым количеством различных значений. Идея индексов на основе битовых карт состоит в том, что если у вас есть несколько условий по нескольким столбцам с малым количеством значений, объединение индексов на основе битовых карт с помощью логических операторов OR и AND уменьшит количество строк-кандидатов очень быстро и весьма эффективно. Хотя реализации различаются, основной принцип индексов на основе битовых карт ближе к операциям, которые вы можете выполнять с полнотекстовыми индексами¹, чем к обычным индексам на основе В-дерева, за исключением того, что, как и с обычными индексами на основе В-дерева, верно следующее:

- Индексы на основе битовых карт правильно индексируют значения столбцов, но не части столбца, как полнотекстовые индексы.

¹ Полнотекстовые индексы являются неотъемлемой частью SQL Server и MySQL, и они доступны, хотя и не так тесно интегрированы, в Oracle.

- Индексы на основе битовых карт полностью участвуют в реляционных операциях типа объединений.
- Индексы на основе битовых карт обрабатываются (с ранее упомянутыми ограничениями) в режиме реального времени.

Если вы когда-нибудь встретите индексы на основе битовых карт, следуйте таким правилам:

- Если система транзакционная, не используйте индексы на основе битовых карт в активно обновляемых таблицах.
- Не изолируйте индексы на основе битовых карт. Сила индексов на основе битовых карт в том, как вы их объединяете. Если у вас есть один индекс на основе битовых карт среди нескольких обычных индексов на основе В-дерева, оптимизатору будет трудно использовать одиночный индекс на основе битовых карт эффективно.

Кластерные индексы

Вы можете встретить другой основной тип исключений в кластерных индексах, которые я упоминал уже не раз. Задача кластерных индексов состоит в том, чтобы обеспечить сохранение строк в таблице в том же порядке, что и ключи в индексе. Кластерные индексы являются эффективным механизмом как SQL Server, так и MySQL (с таким механизмом хранения, как InnoDB), и у них есть близкие родственники в Oracle в виде индекс-таблиц. В кластерных индексах ключи ведут вас прямо к данным, а не к адресу данных, поскольку таблица и индекс объединены, что вдобавок экономит дисковое пространство. Поскольку обновление кластерного ключа означает физическое перемещение всей строки на хранение в новое место, кластерный индекс (очевидно, что он может быть только один в таблице) будет обычно первичным ключом (не говорите мне, что вы обновляете ваш первичный ключ!) или кандидатом в первичные ключи – то есть комбинацией обязательных и уникальных столбцов.

Есть одна вещь, которую нужно проверить в случае кластерных индексов: является ли порядок строк подходящим для того, что вы хотите сделать? Три преимущества кластерных индексов заключаются в том, что вы переходите непосредственно к строкам, строки сгруппированы вместе (как видно из названия) для непрерывных значений ключевых столбцов, и они заранее отсортированы, что экономит СУБД много работы в следующих ситуациях:

- при использовании операторов `order by`;
- при использовании операторов `group by`;
- при использовании таких функций ранжирования, как `rank()` или `row_number()`.

Они могут также оказаться полезными в объединениях. Если столбцы, с которыми ищется соответствие, находятся в двух таблицах в одном порядке, что может произойти, если первичный ключ первой таблицы является первым столбцом в композитном первичном ключе второй таблицы, нахождение соответствий сложности не представляет.

Заметьте также, что если вы систематически используете в качестве первичного ключа не несущий практического смысла столбец идентификации (столбец с автоматическим приращением для пользователей MySQL), который никогда не используется ни как ключ сортировки, ни как критерий для группировки или объединения, ваша кластерная таблица не будет иметь преимуществ перед обычными таблицами.

Индексы по выражениям

В заключение несколько слов об индексах по вычисляемым столбцам (иногда называемыми индексами, основанными на функциях). Вспоминая пример поиска в словаре, обратите внимание, что когда вы ищете не слово, а его преобразование (например, слова, начинающиеся с какой-то буквы), словарь будет использоваться только как список слов, который надо просканировать, без всякой надежды на эффективность поиска. Если вы примените функцию к столбцу и сравните ее с константой, вы сделаете индекс по этому столбцу непригодным, вот почему лучше вместо этого применять обратную функцию к константе. Но если вы хотите искать рифмы, есть одно решение: использование словаря рифм (боюсь, что ваша поэзия не будет пользоваться успехом). В словаре рифм элементы отсортированы не в стандартном алфавитном порядке. Индексы по вычисленным столбцам соотносятся с индексами по исходным столбцам так же, как словари рифм с обычными словарями. Однако есть ограничение: индексированное выражение должно быть детерминированным – то есть при получении одних и тех же значений аргумента они должны всегда возвращать одно и то же значение. Прежде чем бросаться переопределять все функции, которые снижают производительность, как детерминированные, прочитайте следующую главу. Могут возникать интересные побочные эффекты. Но индексы по вычисленным столбцам или (действительно) детерминированным функциям спасли не одно плохо спроектированное и написанное приложение.

Синтаксический разбор и связующие переменные

Кроме статистики и индексов – и то и другое зависит от кода приложения – есть еще один момент, имеющий непосредственное отношение к коду приложения, который надо проверить на самых ранних этапах: как закодированы операторы. Существуют жестко закодированные операторы, в которых все константы являются неотъемлемой частью оператора, и гибко закодированные операторы, в которые константы, значения которых могут меняться, передаются как аргументы. Я писал в первой главе, что жестко закодированные операторы затрудняют понимание того, что загружает сервер СУБД, и поэтому могут привлечь наше внимание не к тем аспектам, где кроется корень проблем, если мы будем недостаточно внимательны.

Они могут также привести к серьезному падению производительности (это особенно справедливо в случае Oracle). Чтобы понять, что загружает

сервер, мы должны понимать, что происходит на каждом этапе, и сначала нужно описать, что происходит, когда мы запускаем оператор SQL. С точки зрения разработки мы пишем оператор SQL в строковую переменную. Этот оператор может быть константой или динамически создаваться приложением. Затем мы вызываем функцию, которая выполняет оператор. Мы проверяем код ошибки, и если ошибок нет, а оператором был `select`, мы в цикле извлекаем строки с помощью другой функции.

Но что происходит на стороне сервера?

SQL является специализированным языком для сохранения, извлечения и модификации данных. Как и во всех языках, операторы преобразуются в машинный код или по крайней мере в какой-то промежуточный, базовый код. Однако преобразование является ресурсоемкой операцией в большинстве языков. Учтите перевод синонимов, управление правами доступа к таблицам, интерпретацию символа `*` в операторе `select` и т. п. Все эти операции требуют от приложения рекурсивных запросов к словарю данных. Что еще хуже, придется принимать во внимание роль, которую играет оптимизатор, и выбор лучшего плана исполнения: идентификацию индексов, проверку, пригодны ли и полезны ли они, вычисления, в каком порядке обращаться к многочисленным таблицам в сложном связывании, включение представлений в уже пугающий запрос и т. д. Синтаксический разбор запроса SQL является очень сложной и поэтому ресурсоемкой операцией.

Как определить проблемы синтаксического разбора

Есть сравнительно простой способ проверить наличие проблем синтаксического разбора при условии, что у вас есть права на выполнение запросов к нужным системным представлениям. В Oracle `v$$sysstat` сообщит вам все, что нужно, включая процессорное время, которое использовала СУБД и которое потребовалось на выполнение синтаксического разбора с момента запуска:

```
select sum(case name
            when 'execute count' then value
            else 0
            end) executions,
       sum(case name
            when 'parse count (hard)' then value
            else 0
            end) hard_parse,
round(100 * sum(case name
                when 'parse count (hard)' then value
                else 0
                end) /
      sum(case name
            when 'execute count' then value
            else 0
            end), 1) pct_hardcoded,
round(100 * sum(case name
```

```

        when 'parse time cpu' then value
        else 0
    end) /
sum(case name
    when 'CPU used by this session' then value
    else 0
end), 1) pct_cpu
from v$sysstat
where name in ('parse time cpu',
'parse count (hard)',
'CPU used by this session',
'execute count');
```

В MySQL `information_schema.global_status` даст вам очень похожую информацию, кроме использования процессорного времени:

```

select x.queries_and_dml,
       x.queries_and_dml - x.executed_prepared_stmt hard_coded,
       x.prepared_stmt,
       round(100 * (x.queries_and_dml - x.executed_prepared_stmt)
            / x.queries_and_dml, 1) pct_hardcoded
from (select sum(case variable_name
    when 'COM_STMT_EXECUTE' then 0
    when 'COM_STMT_PREPARE' then 0
    else cast(variable_value as unsigned)
end) as queries_and_dml,
       sum(case variable_name
    when 'COM_STMT_PREPARE' then cast(variable_value as unsigned)
    else 0
end) as prepared_stmt,
       sum(case variable_name
    when 'COM_STMT_EXECUTE' then cast(variable_value as unsigned)
    else 0
end) as executed_prepared_stmt
from information_schema.global_status
where variable_name in ('COM_INSERT',
                        'COM_DELETE',
                        'COM_DELETE_MULTI',
                        'COM_INSERT_SELECT',
                        'COM_REPLACE',
                        'COM_REPLACE_SELECT',
                        'COM_SELECT',
                        'COM_STMT_EXECUTE',
                        'COM_STMT_PREPARE',
                        'COM_UPDATE',
                        'COM_UPDATE_MULTI')) x;
```

Обратите внимание на то, что поскольку все значения являются кумулятивными, то если вы хотите контролировать конкретный фрагмент программы, вам нужно делать снимки до и после, а затем пересчитывать отношения.

В SQL Server можно непосредственно получить значения некоторых счетчиков повторяемости синтаксического разбора. Чтобы дать вам представление об ожидаемых цифрах, скажу, что в корпоративных системах в Oracle я всегда встречал соотношение между синтаксическим разбором и общим временем выполнения в районе 3–4%, когда этот аспект кодирования был удовлетворительным. У меня не было сомнений в необходимости анализа при 8–9%, но, конечно, реальный вопрос заключался в том, сколько этот анализ стоит нам (хотя данные о процессорном времени, которые сообщает Oracle, являются очень хорошим индикатором) и на какой выигрыш в результате коррекции кода мы можем надеяться.

Оценка потерь производительности из-за синтаксического разбора

Чтобы оценить потери из-за синтаксического разбора, я запустил очень простой тест в Oracle, SQL Server и MySQL. Я запустил с одной из таблиц из первой главы тестовую программу JDBC под названием *HardCoded.java*, важнейший фрагмент которой приведен здесь:

```
start = System.currentTimeMillis();
try {
    long txid;
    float amount;
    Statement st = con.createStatement();
    ResultSet rs;
    String txt;

    for (txid = 1000; txid <= 101000; txid++){
        txt = "select amount"
            + " from transactions"
            + " where txid=" + Long.toString(txid);
        rs = st.executeQuery(txt);
        if (rs.next()) {
            amount = rs.getFloat(1);
        }
    }
    rs.close();
    st.close();
} catch(SQLException ex){
    System.err.println("==> SQLException: ");
    while (ex != null) {
        System.out.println("Message: " + ex.getMessage());
        System.out.println("SQLState: " + ex.getSQLState());
        System.out.println("ErrorCode: " + ex.getErrorCode());
        ex = ex.getNextException();
        System.out.println("");
    }
}
stop = System.currentTimeMillis();
System.out.println("HardCoded - Elapsed (ms)\t" + (stop - start));
```

Эта программа выполняет в цикле 100 000 раз простой запрос, который извлекает значение из единственной строки, связанной со значением первичного ключа. Вы можете заметить, что счетчик цикла преобразован в строку и затем просто вставлен в текст запроса, прежде чем оператор передан на сервер СУБД для исполнения.

Если вы выполняете запрос многократно, имеет смысл написать его так, как вы написали бы функцию: вам нужен фрагмент кода, который получает параметры и выполняется каждый раз с разными значениями аргумента (здесь нет ничего необычного для SQL; мы можем сказать то же самое о любом языке). Каждый раз, когда вы запускаете подобный оператор с константами, меняющимися при каждом исполнении, вам стоит не жестко кодировать эти константы, а выполнить три шага:

1. Подготовить операторы с меткой-заполнителем для каждой константы.
2. Привязать значения к операторам, т. е. создать указатели на переменные, которыми будут заменены во время исполнения метки-заполнители.
3. Выполнить оператор.

Когда вам потребуется выполнить оператор с новыми значениями, все, что вам нужно будет сделать, это присвоить новые значения переменным и повторить шаг 3.

Поэтому я переписал программу *HardCoded.java*, используя подготовленный оператор вместо динамически генерируемого жестко закодированного, но двумя различными способами:

- В программе *FirmCoded.java* в каждом повторении цикла создается новый подготовленный оператор с меткой-заполнителем для *txid*, текущее значение *txid* привязывается к оператору, оператор исполняется, значение суммы извлекается и перед увеличением *txid* оператор закрывается.
- В программе *SoftCoded.java* подготовленный оператор создается один раз, а перед следующим исполнением оператора значение параметра просто меняется, после чего значение суммы извлекается в цикле.

На рис. 2.1 показана относительная производительность программ *FirmCoded* и *SoftCoded* по сравнению с программой *HardCoded* для каждой рассматриваемой СУБД.

Различия очень интересны, особенно в случае *FirmCoded.java*: в Oracle наблюдается значительное увеличение производительности, а в MySQL – наоборот, явное падение. Почему? В документации по MySQL Connector/J (драйверу JDBC) указано, что подготовленные операторы реализуются драйвером, и в той же документации проводится различие между клиентскими подготовленными операторами и серверными подготовленными операторами. Это на самом деле не вопрос языка или драйвера: я переписал все три программы на C, и хоть они работали быстрее, чем на Java, соотношение

производительности между *HardCoded*, *FirmCoded* и *SoftCoded* осталось таким же. Различие между MySQL и Oracle заключается в том, как сервер кэширует операторы. MySQL кэширует результаты запросов: если идентичные запросы, возвращающие несколько строк, исполняются часто (это обычно тот случай, который мы видим на популярном веб-сайте, использующем систему управления содержанием, основанную на MySQL), то запрос выполняется один раз, его результат кэшируется, и пока таблица не модифицирована – результат, который позже возвращается из кэша, это тот же запрос, который передается серверу. В Oracle также есть кэш запросов, хоть он появился поздно (в Oracle 11g); однако он десятилетиями использовал кэш операторов, где хранятся планы исполнения вместо результатов. Если подобный (байт за байтом) запрос уже исполнялся с этими же таблицами и до сих пор находится в кэше, он будет использован даже другим сеансом, если среда идентична¹.

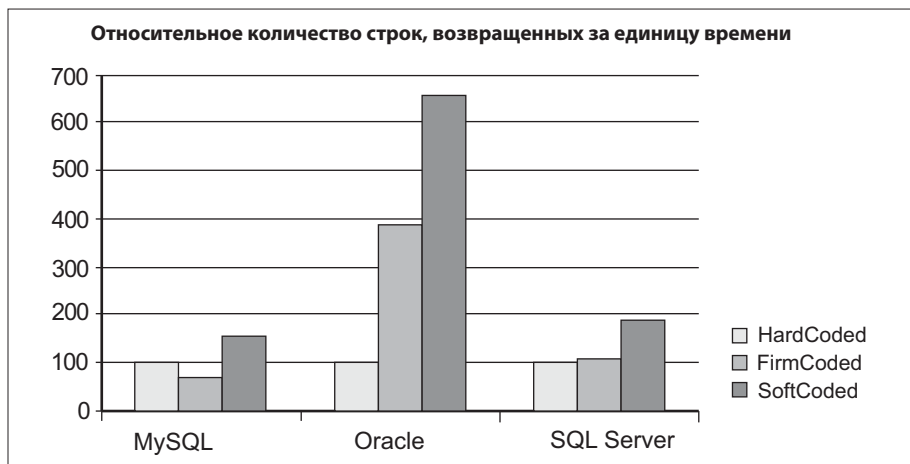


Рис. 2.1. Потери производительности в результате жесткого кодирования

В Oracle каждый раз, когда новый запрос передается серверу, вычисляется контрольная сумма запроса, проверяется контекст и осуществляется поиск в кэше. Если подобный запрос найден, просто происходит этот «мягкий разбор» и запрос исполняется. В противном случае выполняется реальный синтаксический анализ, то есть мы имеем ресурсоемкий «жесткий анализ». В *FirmCoded* мы заменяем жесткий разбор на мягкий разбор, а в *SoftCoded* мы полностью избавляемся от мягкого разбора. Я пропустил различные программы через команду Unix *time*, результаты показаны на рис. 2.2. Можно видеть, что различие в общем времени работы полностью на стороне сервера, поскольку процессорное время на стороне клиента (сумма *sys* и *user*) остается более или менее одинаковым.

¹ Сеанс может локально изменять некоторые параметры, которые влияют на способ исполнения оператора.

Поскольку *FirmCoded* несколько раз выполняет выделение объекта подготовленного оператора, это тот вариант программы, который требует наибольшего количества ресурсов на клиентской стороне.

В MySQL, когда мы подготавливаем оператор, мы связываем его с конкретным планом, но если кэш запросов активен (по умолчанию это не так), то после того, как параметры подставлены, происходит проверка контрольной суммы, чтобы выяснить, можно ли извлечь результат из кэша. Если связующие переменные различаются, оператор, который «подготовлен», является жестко закодированным. В результате в *FirmCoded*, где операторы подготовлены для исполнения только один раз, вообще не будет выигрыша в производительности. Мы просто видим падение производительности из-за более ресурсоемкой клиентской части. Если оператор выполняется повторно с различными значениями, как в случае *SoftCoded*, связь между запускаемым оператором и его планом исполнения сохраняется, и мы видим значительное повышение производительности – хотя и не такое заметное, как в Oracle.

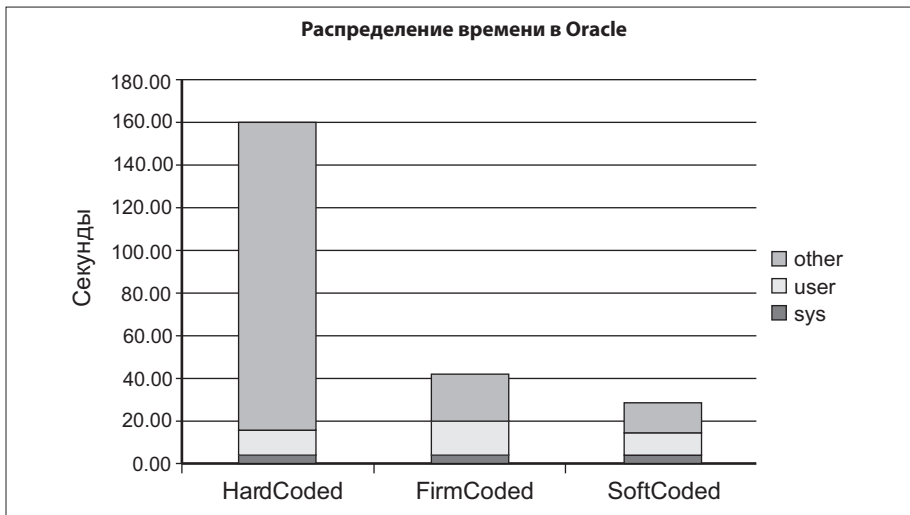


Рис. 2.2. Что говорит команда *Unix time* о затраченном времени

SQL Server находится где-то посередине, и даже если в свете результатов он находится ближе к MySQL, чем к Oracle, его внутренний механизм на самом деле значительно ближе к Oracle (планы исполнения могут использоваться совместно). Картину смазывает тот факт, что даже в простом запросе из моего примера SQL Server выполняет простую параметризацию жестко закодированного оператора. Когда он анализирует жестко закодированный оператор, то если он выглядит достаточно надежным (с первичным ключом это действительно так), СУБД просто вырезает константу и передает ее как параметр. В случае SQL Server жестко закодированные операторы были через внутренние механизмы СУБД превращены

в нечто подобное подготовленным операторам *FirmCoded.java*; в результате исходная производительность была фактически значительно выше, чем в Oracle. Программа *FirmCoded.java* была, однако, несколько лучше, поскольку каждый раз, когда мы подготавливали оператор, мы сэкономили эту дополнительную работу по вырезанию значений констант до проверки того, исполнялся ли уже оператор. Однако я должен подчеркнуть, что простая параметризация производится не активно, а скорее наоборот, и поэтому «работает» только в простых случаях.

Первое (промежуточное) заключение состоит в том, что жестко закодированные операторы губительны в случае Oracle, их нужно избегать в SQL Server, и они вызывают серьезное падение производительности в любой СУБД, когда в отдельном сеансе повторно запускаются операторы, имеющие один и тот же шаблон.

Так или иначе, использование подготовленных операторов в Oracle и в SQL Server – это альтруизм: если все сеансы используют подготовленные операторы, каждый выигрывает, поскольку жесткая фаза будет только в одном сеансе. Напротив, в MySQL ситуация более эгоистичная: один сеанс получит значительную выгоду (50% в моем примере), если он заново исполняет тот же запрос, но остальные сеансы этой выгоды не получают. Имейте в виду, что я говорю здесь о сеансах базы данных – если вы используете сервер приложений, который объединяет несколько пользовательских сеансов, и вы заставляете их совместно использовать небольшое количество сеансов базы данных, пользовательские сеансы могут получать совместную выгоду.

Разрешение проблем синтаксического анализа

Даже разработчики с немалым опытом часто прибегают к жестко закодированным операторам, когда их операторы динамически генерируются «на лету», – просто конкатенировать значения после конкатенации фрагментов кода SQL кажется естественным. Однако здесь есть некоторые подводные камни.

Во-первых, когда вы конкатенируете значения констант, в частности строки, введенные в форме в веб-приложении, возникает немалый риск, если вы недостаточно внимательны.

Во-вторых, многие разработчики, кажется, верят, что жесткое кодирование оператора не дает большой разницы, когда «все так динамично», особенно когда есть практическое ограничение на количество подготовленных операторов. Но даже когда пользователи могут выбрать любое количество критериев среди множества вариантов, количество комбинаций может быть большим (это два в степени, равной количеству критериев), но количество возможных комбинаций обычно ограничивается тем обстоятельством, что некоторые из них чрезвычайно популярны, а другие используются очень редко. В результате реальное многообразие вызывается значениями, которые введены для конкретного критерия, в значительно большей степени, чем выбором самого критерия.

Переход от жестко закодированных к мягко закодированным операторам обычно не требует больших усилий. При использовании подготовленных операторов предыдущий фрагмент кода для жестко закодированного оператора будет выглядеть так:

```
start = System.currentTimeMillis();
try {
    long txid;
    float amount;
    PreparedStatement st = con.prepareStatement("select amount"
        + " from transactions"
        + " where txid=?");
    ResultSet rs;

    for (txid = 1000; txid <= 101000; txid++){
        st.setLong(1, txid);
        rs = st.executeQuery();
        if (rs.next()) {
            amount = rs.getFloat(1);
        }
    }
    rs.close();
    st.close();
} catch(SQLException ex){
    System.err.println("=> SQLException: ");
    while (ex != null) {
        System.out.println("Message: " + ex.getMessage ());
        System.out.println("SQLState: " + ex.getSQLState ());
        System.out.println("ErrorCode: " + ex.getErrorCode ());
        ex = ex.getNextException();
        System.out.println("");
    }
}
stop = System.currentTimeMillis();
System.out.println("SoftCoded - Elapsed (ms)\t" + (stop - start));
```

Модифицировать код очень легко, когда, как в нашем случае, количество значений для «параметризации» постоянно. Однако сложность заключается в том, что когда операторы генерируются динамически, количество условий обычно меняется, и соответственно меняется количество параметров. Это может быть очень удобно, если вы хотите конкатенировать фрагмент кода SQL, содержащий маркер-заполнитель, привязать соответствующую переменную и продолжить дальше. К сожалению, это работает не так (и в любом случае у вас возникнут сложности при повторном исполнении оператора): оператор сначала подготавливается, а затем значения привязываются к нему – но вы не можете подготовить оператор до того, как он полностью построен.

Однако поскольку построение оператора обычно зависит от полей, в которые пользователь ввел данные, не очень сложно пройти через список полей дважды. Один раз – чтобы построить оператор с маркерами-заполнителями, а второй раз – привязать к маркерам реальные значения.

Что если одно значение должно быть привязано несколько раз?

Некоторые языки позволяют использовать именованный маркер (например, в C# или Visual Basic вы можете вызывать маркер-заполнитель @name). Если вы в операторе несколько раз ссылаетесь на одну переменную, вам нужно в контексте .NET вызвать метод SqlCommand.Parameters.AddWithValue("@name", ...) только один раз, чтобы связать одно и то же значение со всеми вхождениями @name в операторе. Однако некоторые языки допускают только групповой маркер-заполнитель (например, ?), и позиция каждого маркера-заполнителя определяется по номеру его появления. Когда каждый параметр появляется только один раз, разница с именованными маркерами-заполнителями невелика. Но когда вам нужно сослаться на один параметр несколько раз, привязывание его столько раз, сколько он появляется, не слишком удобно. Например, если вы хотите получить значения на конкретную дату, вы можете сгенерировать оператор, выглядящий подобным образом:

```
select whatever
from list of tables
where ...
    and effective_date <= ?
    and until_date > ?
...
```

где маркер-заполнитель ? должен принимать в обоих случаях одно и то же значение.

Можно вставить внутрь запроса формальный запрос, позволяющий вам сослаться на то, что он возвращает. Например, в Oracle это будет выглядеть так:

```
select whatever
from list of tables
    , (select ? as val from dual) as mydate
where ...
    and effective_date <= mydate.val
    and until_date > mydate.val
...
```

Замена жестко закодированных операторов мягко закодированными не требует сверхчеловеческих усилий, но это в любом случае не та работа, которую можно делать механически, – вот почему люди так часто поддаются искушению возложить всю работу на СУБД.

Разрешение проблем синтаксического анализа для ленивых

Вы уже видели, что SQL Server пытается разрешить проблемы синтаксического анализа по-своему. Это довольно робкая попытка, и вы можете сделать ее более смелой, присвоив параметру базы данных `PARAMETERIZATION` значение `FORCED`. В Oracle эквивалентную возможность дает параметр `CURSOR_SHARING`, значением по умолчанию которого является `EXACT` (то есть планы исполнения используются только тогда, когда операторы строго идентичны), но ему можно присвоить значение `SIMILAR` (что очень близко к значению по умолчанию `SIMPLE` параметра SQL Server `PARAMETERIZATION`) или `FORCE`. Однако есть несколько огорчительных различий: например, в Oracle параметр может быть установлен как на уровне базы данных¹, так и в области действия сессии. Oracle также, когда его об этом попросишь, ведет себя несколько энергичнее, чем SQL Server. Например, SQL Server может оставить следующий фрагмент кода нетронутым:

```
and char_column like 'pattern%'
```

в то время как Oracle превратит его в:

```
and char_column like :''SYS_B_n'' || '%'
```

Так же SQL Server сохранит оператор, который уже содержит по крайней мере один параметр, в то время как Oracle будет параметризовать его дальше. Но в целом, идея та же самая: заставить СУБД выполнить то, что разработчики должны были сделать в первую очередь.

Я запустил мои тесты во второй раз с Oracle после выполнения (с правами системного администратора) следующей команды:

```
alter system set cursor_sharing=similar;
```

Как я говорил, такая настройка заставляет Oracle вести себя, как SQL Server. И если я заменю цифры, полученные перед этим с Oracle, на новый результат, я получу коэффициент ускорения, который значительно больше схож с тем, что мы получили с SQL Server – поскольку и в SQL Server, и в Oracle, если операторы являются по-прежнему жестко закодированы в программе в *HardCoded.java*, они больше не являются жестко закодированными во время исполнения (рис. 2.3).

С практической точки зрения, если мы запускаем программу с Oracle и замечаем проблемы синтаксического анализа, одной из первых проверок готовности к работе будет выяснение значения параметра `cursor_sharing` и, если оно равно `exact`, присвоение ему значения `similar` и проверка, увеличится ли производительность.

¹ Не забывайте, что в SQL Server и MySQL термин *база данных* означает не одно и то же, когда мы говорим о сервере.

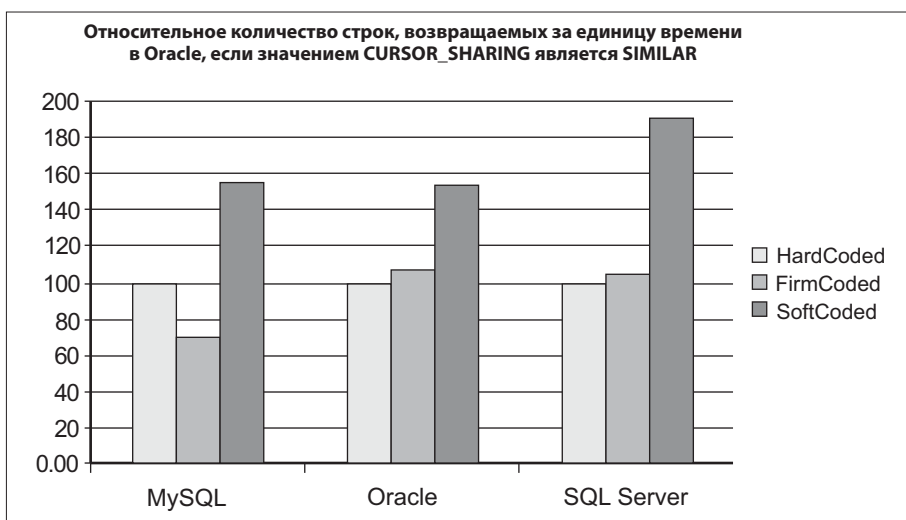


Рис. 2.3. Падение производительности со включенным фоновым мягким разбором

Правильный подход к разрешению проблем синтаксического анализа

Зачем заботиться об использовании подготовленных операторов, если СУБД может сделать это за нас? Во-первых, мы можем видеть в примерах с Oracle и SQL Server, что если процесс автоматизированной параметризации лишь немного менее эффективен, чем подготовленные операторы, то операторы, которые мы подготовили один раз для многократного исполнения, работают по крайней мере в полтора раза быстрее, чем те, параметризация которых происходит перед самым исполнением. Во-вторых, наличие пункта *Автозамена* в меню *Сервис* текстового редактора не означает, что вам можно не обращать внимание на грамматику и позволять системе автозамены делать неправильные исправления. В-третьих, менее активный тип системной параметризации выдает ошибочные предупреждения и будет с реальными приложениями смягчать только часть проблемы; более активный тип может свести вас с ума.

Вам нужно будет позаботиться не только о том, чтобы заменить каждый маркер-заполнитель на константу. Когда у вас есть столбец с малым количеством различных значений, передача значений во время исполнения может оказаться вредна, если значения не распределены равномерно; проще говоря, вас могут подстерегать неприятные сюрпризы, если одно значение встречается очень часто, а другие – относительно редко (частый случай со столбцами, в которых записано состояние). Случай, когда столбец с небольшим количеством значений проиндексирован, может привести к проблемам, поскольку редкие значения чрезвычайно избирательны.

Когда оптимизатор решает, какой план исполнения считать лучшим, то, встретив при анализе оператора что-нибудь подобное:

```
and status = ?
```

и при этом столбец `status` проиндексирован, он вынужден выбирать, использовать индекс (возможно, отдавая ему предпочтение перед другим индексом) или нет. В случае соединения это может повлиять на решение посетить таблицу, к которой относится столбец `status`, перед другими. Целью оптимизатора является выдача результирующего набора как можно быстрее, что означает возможно более раннюю фильтрацию строк, которые не могут принадлежать результирующему набору.

Проблема с маркерами-заменителями состоит в том, что они не содержат никаких намеков на то, какое реальное значение будет передано в момент исполнения. На этом этапе есть несколько возможностей, которые зависят от того, что оптимизатор знает о данных (статистика, которую часто собирают в плановом порядке как часть администрирования базы данных):

- Никой статистики не собирали вообще, и оптимизатор ничего не знает о содержимом столбцов. В этом случае единственное, что он знает, это то, что индекс не уникальный. Если у оптимизатора есть выбор между неуникальным индексом и уникальным, он предпочтет уникальный индекс, поскольку одно значение ключа может вернуть не более одной строки при уникальном индексе. Если у оптимизатора есть выбор только между неуникальными индексами, можно надеяться разве что на удачу¹. Вот довод в пользу статистики.
- Статистика не особо точна, и все, что оптимизатор знает, – столбец содержит сильно различающиеся значения. В таком случае он обычно игнорирует индекс, что может оказаться очень неудачным, если условие по столбцу `status` окажется наиболее избирательным.
- Статистика точна, и оптимизатор также знает из частотной гистограммы, что некоторые значения очень избирательны, а другие не избирательны вообще. Встретив маркер-заполнитель, оптимизатор вынужден будет выбирать из следующего:
 - Необузданный оптимизм: предположение, что передаваемое значение всегда будет очень избирательным, и поэтому оптимизатор предпочтет план исполнения, который даст преимущество условию по столбцу `status`. Вряд ли вы встретите оптимистичные оптимизаторы, поскольку оптимизм – штука ненадежная.
 - Осторожность: игнорирование индекса, поскольку статистически его использование сомнительно. Мы возвращаемся к случаю, где оптимизатор знает только то, что есть небольшое количество различных значений, а больше ничего – и снова это может быть плохой тактикой, когда условие по столбцу `status`

¹ Удачу иногда удается подтолкнуть.

оказывается наиболее избирательным в запросе.

- **Сообразительность:** опора на значение, которое было передано первым (в Oracle это известно как *bind variable peeking*). Если значение избирательно, индекс будет использован; если нет, индекс будет проигнорирован. Это все очень хорошо, но что если утром мы используем избирательные значения, а затем в течение дня шаблон запросов меняется и в конце идут запросы с самым распространенным значением? Или если происходит обратное? Что если по каким-то причинам происходит повторный синтаксический разбор запроса со значениями, приводящими к совершенно другому плану исполнения, который будет применяться к большому количеству выполнений? В этом случае пользователи окажутся свидетелями внешне случайного поведения запросов – иногда быстрого, а иногда медленного. У пользователей такое поведение запросов редко будет пользоваться популярностью.
- Еще большая сообразительность: зная, что может быть проблема с одним столбцом, оптимизатор будет проверять значение столбца каждый раз (на практике это означает такое поведение, как будто бы значение было жестко закодировано).

Как видите, маркеры-заполнители являются большой проблемой для оптимизаторов, когда они занимают место значений, которые должны сравниваться с индексированными столбцами, содержащими малое количество неравномерно распределенных значений. Иногда оптимизаторы делают правильный выбор, но очень часто, даже когда они пытаются быть сообразительными, их выбор оказывается неверным.

Для столбцов с малым количеством различных неравномерно распределенных значений лучшим способом кодирования, если распределение данных оказывается приемлемо статичным, является, несомненно, жесткое кодирование значений, с которыми происходит сравнение в операторе *where*. Наоборот, все неравномерно распределенные значения, которые изменяются между последовательными исполнениями, *должны* передаваться как аргументы.

Обработка списков в подготовленных операторах

К сожалению, есть проблема, которая возникает часто (особенно в динамически генерируемых операторах) и которую особенно сложно обрабатывать, когда вы серьезно относитесь к правильной подготовке операторов: фразы *in (...)*, где ... представляет собой разделенный запятыми список, а не подзапрос.

В некоторых случаях фраза *in* сложностей не создает:

- Когда мы проверяем столбец *status* или любой столбец, который может принимать одно из нескольких возможных значений, фраза *in* должна оставаться жестко закодированной по той же причине, что

и выше.

- В случае высокой изменчивости значений при постоянном количестве элементов в списке мы можем легко использовать подготовленные операторы обычным образом.

Сложности возникают, когда и значения изменяются в широком диапазоне, и количество элементов в списке тоже меняется сильно. Даже если вы используете подготовленные выражения и осмотрительно используете маркеры для каждого значения в списке, два оператора, отличающиеся только количеством элементов в списке, — это различные операторы, для каждого из которых требуется свой синтаксический разбор. В случае подготовленных операторов о «переменном списке аргументов», который разрешен в некоторых языках, ничего не известно.

Со списками не было бы больших проблем, если бы у вас было единственное условие во фразе `where` с критерием `in`, а значения брались бы из выпадающего списка в форме HTML. Даже если вы можете выбрать несколько элементов, общее их количество в списке ограничено, и наличие в кэше, скажем, десяти различных операторов проблемы не представляет: один для списка с одним элементом, второй для списка с двумя элементами, и так далее до списка с десятью элементами.

Но теперь предположим, что этот список является только одним из пяти различных критериев, которые могут быть динамически добавлены в процессе построения запроса. Вы должны помнить, что наличие пяти возможных критериев означает, что мы можем иметь 32 (два в пятой степени) комбинации, и наличие 32 различных подготовленных операторов вполне приемлемо. Но если один из этих пяти критериев представляет собой список, который может иметь любое количество значений от одного до десяти, то такой список сам по себе эквивалентен десяти критериям, и наши пять критериев превращаются в 14. Возведя два в четырнадцатую степень, получим 16 384 — число совсем другого порядка. И конечно, чем больше максимально возможное количество элементов в списке, тем больше количество комбинаций. В результате выгода от применения подготовленных операторов исчезает.

Учитывая все это, можно обрабатывать списки в подготовленных операторах тремя способами, которые мы обсудим в следующих подразделах.

Передача списка как одной переменной

С помощью некоторых приемов SQL вы можете разбить строку и сделать ее подобной таблице, из которой можно производить выборку или которую вы можете позже соединить с другими таблицами. Для дальнейшего объяснения я буду использовать простой пример для MySQL, который вы легко сможете адаптировать к другому диалекту SQL. Предположим, наша программа вызывается из формы HTML, а в ней присутствует список, из которого пользователь может выбрать несколько значений. Вы знаете только то, что из списка может быть возвращено не более десяти значений.

Первой операцией является конкатенация всех выбранных значений в строку (разделителем в данном случае является запятая) и добавление в начале и конце списка того же разделителя, как в этом примере:

```
`, Doc, Grumpy, Happy, Sneezy, Bashful, Sleepy, Dopey, `
```

Эта строка будет параметром. Теперь я размножу эту строку столько раз, сколько максимально ожидается элементов, используя картезианское объединение следующим образом:

```
select pivot.n, list.val
from (select 1 as n
      union all
      select 2 as n
      union all
      select 3 as n
      union all
      select 4 as n
      union all
      select 5 as n
      union all
      select 6 as n
      union all
      select 7 as n
      union all
      select 8 as n
      union all
      select 9 as n
      union all
      select 10 as n) pivot,
(select ? val) as list;
```

Есть несколько способов получить что-то похожее на `pivot`. Например, я могу использовать специальную таблицу для большего количества строк. Если я подставляю строку на место маркера-заполнителя и запускаю запрос, то получу следующий результат:

```
mysql> \. list0.sql
+----+-----+
| n | val |
+----+-----+
| 1 | , Doc, Grumpy, Happy, Sneezy, Bashful, Sleepy, Dopey, |
| 2 | , Doc, Grumpy, Happy, Sneezy, Bashful, Sleepy, Dopey, |
| 3 | , Doc, Grumpy, Happy, Sneezy, Bashful, Sleepy, Dopey, |
| 4 | , Doc, Grumpy, Happy, Sneezy, Bashful, Sleepy, Dopey, |
| 5 | , Doc, Grumpy, Happy, Sneezy, Bashful, Sleepy, Dopey, |
| 6 | , Doc, Grumpy, Happy, Sneezy, Bashful, Sleepy, Dopey, |
| 7 | , Doc, Grumpy, Happy, Sneezy, Bashful, Sleepy, Dopey, |
| 8 | , Doc, Grumpy, Happy, Sneezy, Bashful, Sleepy, Dopey, |
| 9 | , Doc, Grumpy, Happy, Sneezy, Bashful, Sleepy, Dopey, |
| 10 | , Doc, Grumpy, Happy, Sneezy, Bashful, Sleepy, Dopey, |
+----+-----+
10 rows in set (0.00 sec)
```

Однако в моем списке только семь элементов. Поэтому я должен ограничить число строк количеством элементов. Простейший способ посчитать элементы – это вычислить длину списка, удалить разделители, вычислить новую длину, а разница покажет количество разделителей. Поскольку количество разделителей на единицу превышает число элементов, я легко могу ограничить вывод правильным количеством строк, добавив в свой запрос следующее условие:

```
where pivot.n < length(list.val) - length(replace(list.val, ',', ''));
```

Теперь я хочу, чтобы первый элемент был в первой строке, второй элемент – во второй строке и т. д. Если я перепишу мой запрос следующим образом:

```
select pivot.n, substring_index(list.val, ',', 1 + pivot.n)
from (select 1 as n
      union all
      select 2 as n
      union all
      select 3 as n
      union all
      select 4 as n
      union all
      select 5 as n
      union all
      select 6 as n
      union all
      select 7 as n
      union all
      select 8 as n
      union all
      select 9 as n
      union all
      select 10 as n) pivot,
      (select ? val) as list
where pivot.n < length(list.val) - length(replace(list.val, ',', ''));
```

я получу с моей тестовой строкой следующий результат:

```
mysql> \. list2.sql
+---+-----+
| n | substring_index(list.val, ',', 1 + pivot.n) |
+---+-----+
| 1 | ,Doc                                         |
| 2 | ,Doc,Grumpy                                |
| 3 | ,Doc,Grumpy,Happy                           |
| 4 | ,Doc,Grumpy,Happy,Sneezy                    |
| 5 | ,Doc,Grumpy,Happy,Sneezy,Bashful             |
| 6 | ,Doc,Grumpy,Happy,Sneezy,Bashful,Sleepy      |
| 7 | ,Doc,Grumpy,Happy,Sneezy,Bashful,Sleepy,Dopey |
+---+-----+
7 rows in set (0.01 sec)
```

Я уже очень близок к конечному результату, который получу, применив `substring_index()` во второй раз. Я хочу, чтобы функция `substring_index()` теперь возвращала последний элемент в каждой строке, для чего заменю первую строку в моем запросе на:

```
select pivot.n,
       substring_index(substring_index(list.val, ',', 1 + pivot.n), ',', -1)
```

Это даст мне следующий результат:

```
mysql> \. list3.sql
+---+-----+
| n | substring_index(substring_index(list.val, ',', 1 + pivot.n), ',', -1) |
+---+-----+
| 1 | Doc |
| 2 | Grumpy |
| 3 | Happy |
| 4 | Sneezzy |
| 5 | Bashful |
| 6 | Sleepy |
| 7 | Dopey |
+---+-----+
7 rows in set (0.00 sec)
mysql>
```

Как можно заметить, хотя запрос сложнее, чем средний запрос из учебника по SQL, он работает, как показано здесь, с любым списком, содержащим не более десяти элементов, а его переделка для использования с более длинными списками сложности не представляет. Текст оператора остается идентичным, поэтому новый синтаксический разбор его происходить не будет, даже если количество элементов в списке изменится.

Передача значений списка как простой строки является очень хорошим и эффективным решением для маленьких списков. Его ограничением является длина строки в языке оболочки или в SQL (самое жесткое ограничение в Oracle – 4000 символов). Если ваш список содержит несколько сотен элементов, более подходящим будет другое решение.

Пакетная обработка списков

Такое решение – пакетная обработка. Длина списков ограничивается фиксированным количеством элементов. Этот метод позволяет нам использовать то обстоятельство, что `in()` игнорирует дубликаты в списке, будь это явный список или результат запроса. Предположим, мы получаем наши параметры в массиве, который, как в предыдущем примере, может содержать до десяти значений. Если я попытаюсь выполнить мягкое кодирование этого оператора, то напишу что-то подобное:

```
my_condition = "where name in"
loop for i in 1 to count(myarray)
  if i > 1 then my_condition = my_condition + "("
```

```
        else my_condition = my_condition + ","
    my_condition = my_condition + "?"
end loop
my_condition = my_condition + ")"
```

Затем я в новом цикле свяжу каждое значение с его маркером-заполнителем и получу столько различных операторов, сколько элементов может быть в списке. Вместо этого я могу сгенерировать оператор, который сможет обрабатывать до десяти элементов в списке:

```
my_condition = "where name in (?, ?, ?, ?, ?, ?, ?, ?, ?)"
```

Затем я могу подготовить оператор, выполнив соответствующий вызов к базе данных, и последовательно привязать десять параметров, повторяя некоторые из фактических параметров, чтобы заполнить десять элементов:

```
loop for i in 1 to 10
    j = 1 + modulo(i, count(array))1
    bind(stmt_handler, i, array(j))
end loop
```

Если возникнет потребность, мы можем установить несколько размеров списков, при этом один оператор будет предназначен для списков размером, скажем, от 1 до 20 элементов, другой оператор – для списков длиной от 21 до 40 элементов и т. д. Таким образом, если у нас будут списки, содержащие любое количество элементов, не превышающее 100, мы будем уверены, что в кэше хранится не более 5 различных операторов вместо 100.

Использование временной таблицы

В случае очень больших списков простейшим решением будет, вероятно, использование временной таблицы и замена явных списков подзапросом, который осуществляет выборку из этой временной таблицы – и здесь текст запроса останется тем же, вне зависимости от количества строк во временной таблице. Недостатком этого метода оказывается стоимость вставки: взаимодействие с базой данных усиливается. Однако действительный вопрос касается происхождения данных, которые используются для заполнения списка. В случае огромных списков нет ничего необычного в том, что мы получаем их с помощью запросов. В таком случае операция `insert ... select ...` во временную таблицу будет происходить полностью на сервере, и это избавит прикладную программу от извлечения данных. Вы можете решить, что временные таблицы не нужны, а простой подзапрос или объединение выполнит ту же задачу значительно эффективнее. Если данные извлекаются из временного файла и этот файл можно послать на сервер, на котором находится база

¹ Первое значение не будет первым элементом массива, но, тем не менее, все прекрасно работает.

данных, то вы можете почти безболезненно использовать различные утилиты для загрузки временной таблицы (и внешние таблицы Oracle, и оператор MySQL `load data infile`, и SQL Server Integration Services предоставляют разработчикам инструменты, позволяющие значительно уменьшить объем программирования).

Групповые операции

Кроме индексирования, статистики и скорости синтаксического разбора – основных факторов, влияющих на одно конкретное приложение или все приложения, получающие доступ к базе данных, – есть и другие типы процессов, которые нужно проверить, если производительность вас не устраивает. Например, фактором, снижающим производительность при извлечении или вставке большого количества строк, является обмен информацией между сервером приложений и базой данных по локальной сети.

К сожалению, эта тема связана не только с СУБД, но и с языком, используемым для доступа к СУБД. Поэтому я дам широкий обзор возможных проблем и приведу несколько примеров. Принцип очень прост: предположим, вам нужно переместить кучу песка. Обработка строки за строкой подобна перемещению этой кучи с помощью чайной ложки. Вероятно, использование лопаты или тачки будет более эффективным.

Реализация может быть самой разной. Для некоторых вариантов может потребоваться больше фантазии, чем для других. Но в любом случае, когда вы извлекаете данные с сервера в курсорном цикле, вы посылаете команду «давай еще данные». Если говорить о самом низком уровне, вы посылаете по сети пакет, в котором содержится эта команда. В ответ СУБД посылает вам обратный пакет, в котором содержатся либо требуемые данные, либо сообщение «готово». Каждый раз происходит обмен информацией, вызывающий некоторые сетевые задержки. Обмен происходит и тогда, когда вы вставляете строки. Пакеты имеют фиксированный размер, и отправка полного пакета несколько не дороже и не медленнее, чем отправка почти пустого пакета. Обмен более полными пакетами означает меньшее количество пакетов, требуемых для того же объема данных. Когда вы вставляете или извлекаете большие объемы данных – например, для создания файла, который потом будет отправлен куда-то еще для обработки в другой системе, – вряд ли есть смысл посылать на сервер строку за строкой или получать данные с сервера построчно, даже когда процедурная часть вашей программы действительно осуществляет обработку по одной строке. Некоторые продукты и среды программирования позволяют вам запускать пакетные операции; примерами являются T-SQL, JDBC и SQLJ, позволяющие вам группировать несколько операторов `insert` перед отправкой их на сервер, или объект `SqlPipe` в .NET Framework. Другие позволяют осуществлять вставку данных в массив и извлечение из массива, как в случае Oracle и PL/SQL или интерфейса OCI C/C++. В MySQL реализован несколько другой

подход с потоками; идет непрерывный поток данных без явного запроса на их отправку. Если вы используете библиотеки С или РНР, например, вы сможете получать целый результирующий набор за один прием, что хорошо для малых и средних результирующих наборов, но может привести к нехватке памяти, когда потребуется вернуть большой объем данных. В этом случае вам придется переключиться в другой режим.

В некоторых случаях пакетная передача полностью выполняется клиентской стороной вашего приложения. Например:

```
while fetch_one_row_from_database()
do something
```

и за вашей спиной клиентская сторона сделает следующее:

```
while fetch_one_row_from_array()
do something

fetch_one_row_from_array {
    increase index;
    if index > array_size
        fetch_array_size_rows_at_once_from_database
        set index to point to first array entry
    if buffer[index] is empty
        return done
    else
        return buffer(index)
```

Такое обычно происходит в программах JDBC, в функциях интерфейса с базами данных РНР или в некоторых клиентских библиотеках С/С++. Немногие разработчики помнят, что размер массива иногда бывает под их контролем и что значение по умолчанию не всегда оказывается самым подходящим для их задач.

На рис. 2.4 показано, как на производительность влияет установка «размера выборки» в случае программы JDBC, которая направляет содержимое таблицы `transactions` из первой главы в файл CSV. Не пытайтесь сравнивать различные продукты друг с другом, поскольку тесты запускались на разных машинах и я стандартизировал количество строк, возвращаемых за единицу времени, в виде «индекса производительности». MySQL и SQL Server (то есть их драйверы JDBC) явно игнорировали настройки, но Oracle очень чувствителен к параметру. Увеличение размера выборки от значения по умолчанию, равного 10, до 100 удвоило производительность в этом примере с очень небольшими усилиями (и программа JDBC, и база данных были запущены на одной машине). Когда я утверждал, что MySQL игнорирует настройку, это было не совсем точно: я мог увеличить производительность MySQL приблизительно на 15%, создав оператор типа *forward only* и *read only* и установив размер выборки равным `Integer.MIN_VALUE`, что привело бы к переключению в более быстрый (в данном случае) потоковый режим (что зададо бы также поддержку протокола SQL Server TDS).

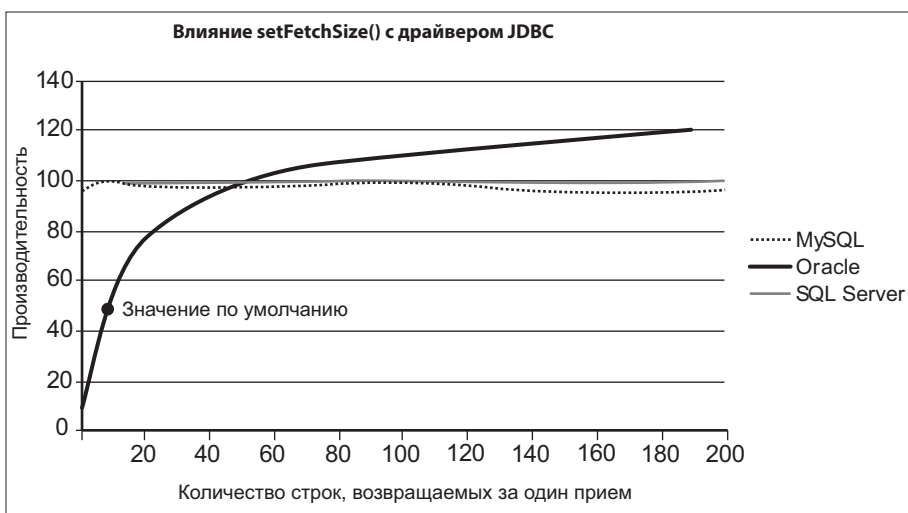


Рис. 2.4. Изменение размера выборки в программе JDBC

Во всех случаях, которые я упоминал до сих пор, использовался довольно консервативный подход к групповым операциям, большей частью путем буферизации данных для оптимизации передачи. Вы можете встретить групповые операции, которые могут быть интересны для масштабированных процессов: например, групповые операции копирования с помощью SQL Native Client в SQL Server или прямые функции *path-loading* интерфейса вызовов C в Oracle. Но в этом случае мы очень далеки от простых и быстрых изменений.

Управление транзакциями

Еще одним важным моментом, который надо проверить, является управление транзакциями. Под транзакцией подразумевается нераздельный набор операторов. Она открывается либо неявно первым оператором, изменяющим базу данных, либо явно и заканчивается либо фиксацией, сохраняющим изменения на случай сбоя сервера, либо откатом, который отменяет все изменения в базе данных с момента начала транзакции (или, в некоторых случаях, точки промежуточного сохранения). В момент окончания транзакции снимаются все наложенные сеансом блокировки измененной таблицы, страницы или строки, в зависимости от уровня блокировки. На практике, если сервер выходит из строя в середине транзакции, все изменения теряются, поскольку могли быть не записаны на диск (если же были записаны, при запуске базы данных будет произведен их откат).

Фиксация изменений занимает время, поскольку единственный способ гарантировать, что изменения станут необратимыми, даже в случае выхода базы данных из строя, состоит в записи всех обновлений в файл

в энергонезависимой памяти с подтверждением, что процесс записи данных успешно завершился. Если фиксировать изменения очень часто, вы заметите, что значительное время будет занято ожиданиями фиксации.

Фиксация часто является критичной в оперативной обработке транзакций, поскольку если блокировки занимают слишком много времени, конкурентные транзакции ставятся в очередь, пока предыдущая транзакция не закончится и блокировка с ресурса не будет снята. В случае ночных пакетных обновлений (когда нет конкуренции) один процесс может выполнять масштабные операции вставки, очистки и изменения данных. Если вы не знаете, насколько часто ваша программа пакетной загрузки и обновления выполняет фиксацию в ходе работы, это нужно обязательно выяснить. В некоторых языках по умолчанию (привет, JDBC!) используется режим автоматической фиксации, то есть фиксация происходит после каждого изменения в базе данных, что, как можно видеть на рис. 2.5, очень болезненно для всех СУБД (я стандартизовал количество строк, обновляемых за единицу времени, в массивованном обновлении двух миллионов строк).

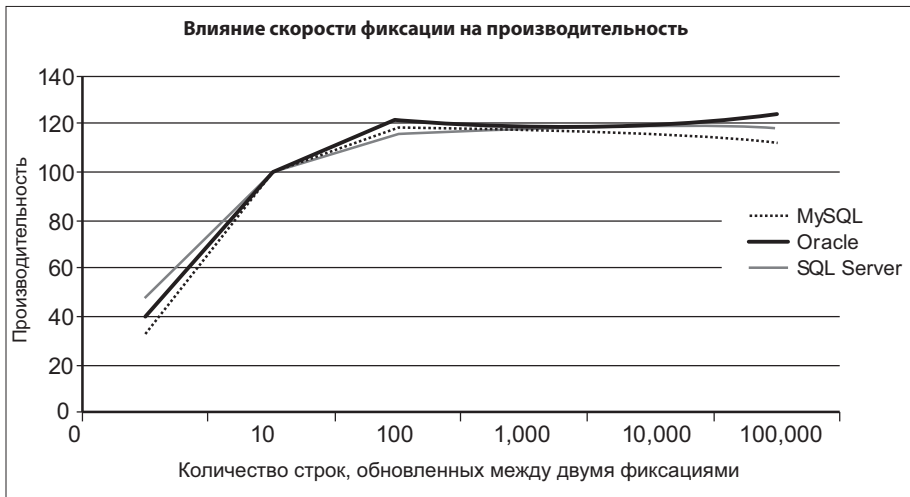


Рис. 2.5. Влияние частоты фиксации на производительность операции обновления

Хочу добавить, что когда включен режим автоматической фиксации, не только серьезно снижается производительность пакетных операций, но и нарушается концепция транзакций, поскольку логической единицей обработки становится оператор SQL. Это может привести к некоторым интересным ситуациям. Предположим, вы хотите перенести деньги с чекового счета на сберегательный счет. Сначала вы обновляете ваш чековый счет, вычитая сумму, которую хотите перевести, а затем обновляете сберегательный счет, добавляя к нему ту же сумму. А теперь

представьте, что будет, если компьютер выйдет из строя между двумя обновлениями. Транзакции были изобретены именно для этого: если два обновления входят в состав одной транзакции, первое обновление будет отменено. А если она не будет отменена, вам придется звонить в банк!

Мы вернемся к транзакциям и частоте обновления в шестой главе, но поскольку речь идет о «быстрых победах», пока мы на этом остановимся. Теперь пришла пора взглянуть на представления (views) и хранимые функции (stored functions).

3

Пользовательские функции и представления

*Когда инструментов мало,
а задач много, приходится жульничать.*

Генри В. Фаулер (1858–1933)
и Фрэнсис Г. Фаулер (1871–1918)
Королевский Английский¹, глава IV

Хранимые объекты в базе данных являются главными мишенями рефакторинга по двум причинам. Во-первых, они часто встречаются при рассмотрении серьезных проблем с производительностью. Во-вторых, то, что их исходный код находится внутри базы данных, часто позволяет получить к ним более быстрый доступ, чем к процедурному коду, который разбросан среди бесчисленных исходных файлов. Как и в случае индексов и статистики, вы зачастую можете значительно повысить производительность, выполнив рефакторинг написанных пользователем функций и представлений, не касаясь кода, даже если функции и представления на самом деле являются частью кода.

Хранимые функции и представления служат цели сосредоточения в одном месте часто вызываемого кода SQL. Если вам удастся значительно усовершенствовать часто используемый хранимый объект, полученное увеличение производительности распространится далеко за пределы первоначального проблемного процесса. И наоборот, вы должны

¹ The King's English (Oxford Language Classics Series) – известный труд братьев Фаулер, опубликованный впервые в 1906 году и посвященный использованию английского языка. – *Примеч. науч. ред.*

отнестись к рефакторингу хранимых объектов со всей тщательностью, пропорционально его возможным побочным эффектам.

Разработчики обычно начинают с написания функций для часто исполняемых операций (включая проверку правильности ввода, которую часто можно реализовать путем объявления ограничений целостности). Я не случайно включил две служебные функции в пример из первой главы: функции поиска используются часто и нередко являются причиной низкой производительности. Систематическое применение того, что является хорошей практикой в процедурных языках, часто приводит к падению производительности кода SQL. Язык SQL не процедурный, а скорее декларативный – он в первую очередь работает с таблицами, а не со строками. В мире процедурного программирования вы используете функции для записи часто используемых операций, имея в виду две задачи:

- Обеспечить использование всеми разработчиками одного и того же тщательно отлаженного кода вместо размножения, иногда неправильного, строк кода, выполняющего те же задачи.
- Упростить поддержку путем концентрации всего кода в одном месте, чтобы все изменения вносились только один раз.

Если вы хотите записать сложный оператор SQL, выполнив те же требования, которым удовлетворяют функции в мире процедурного программирования, вам нужно использовать представления (views), которые являются настоящими «функциями» языка SQL. Как вы увидите в этой главе, нельзя сказать, что представления никогда не влияют на производительность. Но когда при программировании доступа к базам данных у вас есть выбор, то слово «представление» должно возникать в вашей голове раньше, чем понятие «пользовательская функция» (user-written function).

Пользовательские функции

Если вы привыкли к процедурным языкам, у вас наверняка возникнет искушение широко использовать пользовательские функции. Немногие разработчики способны сопротивляться этому искушению. Давайте условно разделим пользовательские функции и процедуры на три категории:

Процедуры изменения базы данных

Последовательные операции изменения базы данных (то есть в основном вставка, удаление и изменение, плюс разрозненные выборки), выполняющие одиночные бизнес-задачи.

Чисто вычислительные функции

Они могут включать в себя условные операторы и различные операции.

Функции поиска

Выполнение запросов к базе данных.

Процедуры изменения базы данных часто обладают теми же недостатками, что и процедурный код любого типа: несколько операторов там, где достаточно одного, циклы и т. п. Другими словами, они не являются плохими по своей природе, но иногда просто плохо реализованы. Я не буду тратить время на них, поскольку то, что мы будем обсуждать в следующих главах по поводу программирования, в общем применимо и к этому типу хранимых процедур.

Усовершенствование чисто вычислительных функций

Чисто вычислительные функции обычно весьма безопасны. В SQL наиболее распространенным местом для сосредоточения не слишком сложных вычислений¹ являются вычисляемые столбцы в представлениях. При вызове функции переключение контекста замедляет исполнение, но если ваша функция написана не слишком плохо (к базам данных это никакого отношения не имеет), вы не получите большого увеличения производительности после модификации ее кода. Единственный случай, когда модификация кода может дать значительное ускорение, – лучшее использование встроенных функций-примитивов. Чаще всего в своей практике я видел плохое использование встроенных функций работы со строками или датами, особенно при наличии циклов.

Приведу пример пользовательской функции в Oracle, которая работает значительно эффективнее при широком использовании встроенных функций: подсчет строковых шаблонов. Подсчет повторяющихся шаблонов – это операция, которая может оказаться полезной во многих случаях, и вы найдете пример ее применения в конце главы. Давайте сформулируем задачу так: у нас есть строка *haystack* и шаблон *needle*, а мы хотим выяснить, сколько раз шаблон *needle* попадает в строке *haystack*.

Один из вариантов этой функции (`function1`):

```
create or replace function function1(p_needle in varchar2,
                                     p_haystack in varchar2)
return number
is
    i    number := 1;
    cnt number := 0;
begin
    while (i <= length(p_haystack))
    loop
        if (substr(p_haystack, i, length(p_needle)) = p_needle)
        then
            cnt := cnt + 1;
```

¹ В SQL Server сложные вычисления обычно программируются в управляющем коде, или, другими словами, на языке .NET, и вызываются из базы данных.

```

        end if;
        i := i + 1;
    end loop;
    return cnt;
end;
/

```

Это стандартный код, в котором используются две встроенных функции, `length()` и `substr()`. Мы можем немного усовершенствовать его, если не хотим подсчитывать перекрывающиеся шаблоны, увеличивая индекс на длину шаблона `needle` вместо единицы каждый раз, когда находим соответствие.

Однако мы можем написать функцию по-другому (`function2`), используя функцию Oracle `instr()`, которая возвращает позицию подстроки внутри строки. Функция `instr()` принимает два дополнительных параметра: начальная позиция в строке и номер вхождения (первое, второе и т. д.) искомой подстроки. Функция `function2` использует второй параметр, увеличивая его до тех пор, пока поиск не завершится неудачей:

```

create or replace function function2(p_needle in varchar2,
                                     p_haystack in varchar2)

return number
is
    pos          number;
    occurrence number := 1;
begin
    loop
        pos := instr(p_haystack, p_needle, 1, occurrence);
        exit when pos = 0;
        occurrence := occurrence + 1;
    end loop;
    return occurrence - 1;
end;
/

```

В функции `function3` я удалил цикл и теперь сравниваю длину исходной строки `haystack`, в которой осуществляется поиск, с длиной строки после замены всех вхождений шаблона `needle` на пустую строку. Теперь, чтобы выяснить количество вхождений шаблона `needle`, остается просто разделить разницу на длину шаблона. В отличие от предыдущих функций, реализация больше не является специфической для Oracle (я использовал этот метод во второй главе для подсчета разделителей в строке со списком значений).

```

create or replace function function3(p_needle in varchar2,
                                     p_haystack in varchar2)

return number
is
begin
    return (length(p_haystack)
           - length(replace(p_haystack, p_needle, '')))

```

```

        /length(p_needle) ;
    end;
/

```

Чтобы сравнить функции, я создал таблицу и заполнил ее 50 тысячами строк из случайных букв в нижнем регистре, каждая длиной от 5 до 500 символов:

```

create table test_table(id number,
                        text varchar2(500))
/
begin
    СУБД_random.seed(1234);
    for i in 1 .. 50000
    loop
        insert into test_table
            values(i, СУБД_random.string('L', СУБД_random.value(5, 500)));
    end loop;
    commit;
end;
/

```

Наконец, я сравнил все три функции, посчитав в тестовых таблицах, сколько строк содержит больше десяти вхождений букв *s*, *q* и *l* соответственно:

```

select count(*)
from test_table
where functioni('s', text) > 10
/
select count(*)
from test_table
where functioni('q', text) > 10
/
select count(*)
from test_table
where functioni('l', text) > 10
/

```

Я запустил этот тест с каждой из трех функций, просуммировал время выполнения (во всех трех случаях результат оказался практически одинаковым) и разделил 150 000 (общее количество просканированных строк) на общее время выполнения, чтобы сравнить количество строк, просканированных за одну секунду, во всех трех случаях. Результаты эксперимента показаны на рис. 3.1.

Исходная функция с циклом по строкам приблизительно в 10 раз менее эффективна, чем функция с `replace()`, а вариант с использованием `instr()` – где-то посередине между ними. Имеет смысл внимательно прочитать документацию по встроенным функциям вашей СУБД (изобретение велосипеда – занятие неблагодарное). Но усовершенствование функции средней сложности может значительно увеличить производительность в пакетной программе.

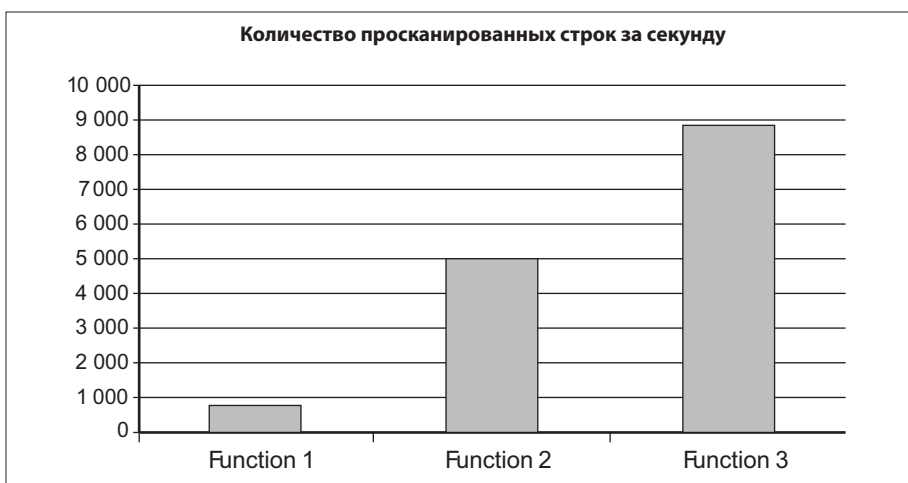


Рис. 3.1. Три различных способа реализации подсчета вхождения шаблона в Oracle

Дальнейшее усовершенствование функций

Оптимизаторы в Oracle и SQL Server знают один специфический тип функций – детерминированные. Функция является детерминированной, если она, будучи вызванной с одними и теми же параметрами, всегда возвращает одно и то же значение. SQL Server сам определяет, является ли функция детерминированной. Oracle считает, что функция детерминированная, если в объявлении типа возвращаемого функцией значения присутствует ключевое слово `deterministic` (MySQL 5.1 знает это ключевое слово, но оптимизатор игнорирует его). Если функция детерминированная, СУБД «запоминает» некоторое количество ассоциаций между параметрами и возвращаемым значением и возвращает результат без реального вызова функций, если параметры вызова были те же самые. Такое кэширование результатов функции может значительно увеличить производительность, но это имеет смысл только тогда, когда мы вызываем функцию с относительно небольшим количеством различных значений параметров. В предыдущем примере функции поиска шаблона объявление функции детерминированной (даже если она таковой является) не принесло бы пользы, поскольку один из параметров – текстовая строка, в которой осуществляется поиск, – каждый раз другой. Однако во многих случаях одни и те же параметры используются снова и снова, особенно при применении функций к датам, поскольку очень часто в процессах обрабатывается очень узкий диапазон дат: одна и та же дата появляется много раз.

С детерминизмом надо быть очень осторожным. Иногда функция не является детерминированной по совершенно неожиданным причинам. Типичный пример – функция, возвращающая число, представляющее

день недели, например когда вы применяете к вашему столбцу с датой функции `to_char(date_column, 'D')` в Oracle, `datepart(dw, date_column)` в SQL Server или `dayofweek(date_column)` в MySQL. За исключением случая MySQL, где функция возвращает номер дня недели по стандарту ISO, функция является недетерминированной, поскольку зависит от региональных настроек и соглашений. Как видно из следующего примера для Oracle, соглашения могут быть различными даже в географически близко расположенных странах:

```
SQL> alter session set nls_territory=spain;
Session altered.
SQL> select to_char(to_date('1970/01/01', 'YYYY/MM/DD'), 'D')
       2 from dual;
```

```
TO_CHAR(TO_DATE('1970/01/01', 'YYYY/MM/DD'), 'D')
-----
4
```

```
SQL> alter session set nls_territory=portugal;
Session altered.
SQL> select to_char(to_date('1970/01/01', 'YYYY/MM/DD'), 'D')
       2 from dual;
```

```
TO_CHAR(TO_DATE('1970/01/01', 'YYYY/MM/DD'), 'D')
-----
5
```

```
SQL> alter session set nls_territory=morocco;
Session altered.
SQL> select to_char(to_date('1970/01/01', 'YYYY/MM/DD'), 'D')
       2 from dual;
TO_CHAR(TO_DATE('1970/01/01', 'YYYY/MM/DD'), 'D')
-----
6
```

В Oracle можно обойти эту проблему путем преобразования числа, соответствующего дню, в его *название*, передав необязательный параметр, указывающий язык для возвращаемого значения:

```
ORACLE-SQL> alter session set nls_language=american;

Session altered.

SQL> select to_char(to_date('1970/01/01', 'YYYY/MM/DD'),
       2 'DAY', 'NLS_DATE_LANGUAGE=ITALIAN')
       3 from dual;

TO_CHAR(TO_DATE('1970/01/01', 'YYYY/MM/DD'), 'DAY', 'NLS_DATE_LANGUAGE=ITALIAN')
-----
GIOVEDI

SQL> select to_char(to_date('1970/01/01', 'YYYY/MM/DD'), 'DAY')
       2 from dual;
```



```
TO_CHAR(TO_DATE('1970/01/01', 'YYYY/MM/DD'), 'DAY')
```

```
-----
```

```
THURSDAY
```

```
SQL> alter session set nls_language=german;
```

```
Session altered.
```

```
SQL> select to_char(to_date('1970/01/01', 'YYYY/MM/DD'),
 2 'DAY', 'NLS_DATE_LANGUAGE=ITALIAN')
 3 from dual;
```

```
TO_CHAR(TO_DATE('1970/01/01', 'YYYY/MM/DD'), 'DAY', 'NLS_DATE_LANGUAGE=ITALIAN')
```

```
-----
```

```
GIOVEDI
```

```
SQL> select to_char(to_date('1970/01/01', 'YYYY/MM/DD'), 'DAY')
 2 from dual;
```

```
TO_CHAR(TO_DATE('1970/01/01', 'YYYY/MM/DD'), 'DAY')
```

```
-----
```

```
DONNERSTAG
```

В SQL Server для того, чтобы указать, какой день недели будет принят за первый, и, соответственно, определить, что же вернет функция `datepart(dw, ...)`, нужно сделать вызов `set datefirst`, а этого нельзя сделать в функции. Однако существует выход: сравнение возвращаемого функцией значения с тем, что возвращает *та же функция при передаче ей известной даты*. Из результатов, полученных в Oracle, мы знаем, что первое января 1970 года было четвергом. Детерминированная функция должна вернуть субботу для 3 января 1970 года и т. д.

Зная, как обеспечить в Oracle действительно детерминированную идентификацию дней, я теперь могу написать функцию, возвращающую единицу, если дата соответствует выходному дню (то есть субботе или воскресенью), а в противном случае – ноль. Чтобы понять преимущества декларирования в Oracle функции как детерминированной, я создам две идентичные функции: одна из них будет объявлена детерминированной, а другая – нет:

```
SQL> create or replace function weekend_day(p_date in date)
 2 return number
 3 is
 4   wday char(3);
 5 begin
 6   wday := substr(to_char(p_date, 'DAY',
 7                     'NLS_DATE_LANGUAGE=AMERICAN'), 1, 3);
 8   if (wday = 'SAT') or (wday = 'SUN')
 9   then
10     return 1;
11   else
12     return 0;
13   end if;
```

```

14 end;
15 /

```

Function created.

Elapsed: 00:00:00.04

```

SQL> create or replace function weekend_day_2(p_date in date)
2   return number
3   deterministic
4   is
5     wday char(3);
6   begin
7     wday := substr(to_char(p_date, 'DAY',
8                       'NLS_DATE_LANGUAGE=AMERICAN'), 1, 3);
9     if (wday = 'SAT') or (wday = 'SUN')
10    then
11      return 1;
12    else
13      return 0;
14    end if;
15 end;
16 /

```

Function created.

Elapsed: 00:00:00.02

Теперь предположим, что у нас есть таблица с записями продаж, где для каждой продажи указаны дата и сумма, и мы хотим сравнить общую сумму продаж за выходные с общей суммой за будние дни в течение прошедшего месяца. Давайте запустим запрос дважды, сначала с недетерминированной функцией, а затем с детерминированной:

```

SQL> select sum(case weekend_day(sale_date)
2             when 1 then 0
3             else sale_amount
4             end) week_sales,
5             sum(case weekend_day(sale_date)
6             when 0 then 0
7             else sale_amount
8             end) week_end_sales
9   from sales
10  where sale_date >= add_months(trunc(sysdate), -1)
11 /
WEEK_SALES WEEK_END_SALES
-----
191815253      73131546.8

```

Elapsed: 00:00:11.27

```

SQL> select sum(case weekend_day_2(sale_date)
2             when 1 then 0
3             else sale_amount

```

```

4         end) week_sales,
5         sum(case weekend_day_2(sale_date)
6             when 0 then 0
7             else sale_amount
8         end) week_end_sales
9     from sales
10    where sale_date >= add_months(trunc(sysdate), -1)
11    /

```

```
WEEK_SALES WEEK_END_SALES
```

```
-----
191815253      73131546.8
```

Elapsed: 00:00:11.24

Результаты неубедительные. Почему? Фактически, мой запрос нарушает одно из условий, которые я сформулировал для эффективности детерминированных функций: многократный вызов с одними и теми же параметрами. Тип date в Oracle эквивалентен типу datetime в других СУБД: он содержит элемент времени с точностью до одной секунды. Нет надобности говорить, что при генерировании тестовых данных я позаботился о том, чтобы создать значения sale_date в разные периоды дня. Давайте скроем элемент времени, применив к данным функцию trunc(), что сделает значение времени равным 00:00:00, и попробуем снова:

```

SQL> select sum(case weekend_day(trunc(sale_date))
2         when 1 then 0
3         else sale_amount
4     end) week_sales,
5         sum(case weekend_day(trunc(sale_date))
6             when 0 then 0
7             else sale_amount
8         end) week_end_sales
9     from sales
10    where sale_date >= add_months(trunc(sysdate), -1)
11    /

```

```
WEEK_SALES WEEK_END_SALES
```

```
-----
191815253      73131546.8
```

Elapsed: 00:00:12.69

```

SQL> select sum(case weekend_day_2(trunc(sale_date))
2         when 1 then 0
3         else sale_amount
4     end) week_sales,
5         sum(case weekend_day_2(trunc(sale_date))
6             when 0 then 0
7             else sale_amount
8         end) week_end_sales
9     from sales

```

```

10 where sale_date >= add_months(trunc(sysdate), -1)
11 /

```

```

WEEK_SALES WEEK_END_SALES
-----
191815253      73131546.8

```

Elapsed: 00:00:02.58

При подходящих условиях детерминированная функция позволила ускорить выполнение запроса в пять раз.

Имейте в виду, что к объявлению функции детерминированной в Oracle нельзя подходить безответственно.

Чтобы понять это, представьте, что у нас есть таблица служащих и что каждый из них может быть включен в проекты с помощью таблицы `project_assignment`, которая связывает идентификатор служащего с идентификаторов проекта. Мы можем запустить следующий запрос, чтобы выяснить, в каких проектах принимают участие люди с фамилией *Sharp*:

```

SQL> select e.lastname, e.firstname, p.project_name, e.empno
2   from employees e,
3        project_assignment pa,
4        projects p
5  where e.lastname = 'SHARP'
6        and e.empno = pa.empno
7        and pa.project_id = p.project_id
8        and pa.from_date < sysdate
9        and (pa.to_date >= sysdate or pa.to_date is null)
10 /

```

LASTNAME	FIRSTNAME	PROJECT_NAME	EMPNO
SHARP	REBECCA	SISYPHUS	2501
SHARP	REBECCA	DANAIDS	2501
SHARP	MELISSA	DANAIDS	7643
SHARP	ERIC	SKUNK	7797

Следующий запрос вернет только единственную запись для сотрудника *Crawley* и проекта, в котором он участвует:

```

SQL> select e.lastname, e.firstname, p.project_name
2   from employees e,
3        project_assignment pa,
4        projects p
5  where e.lastname = 'CRAWLEY'
6        and e.empno = pa.empno
7        and pa.project_id = p.project_id
8        and pa.from_date < sysdate
9        and (pa.to_date >= sysdate or pa.to_date is null)
10 /

```

LASTNAME	FIRSTNAME	PROJECT_NAME
-----	-----	-----
CRAWLEY	RAWDON	SISYPHUS

Теперь предположим, что по некоторым неизвестным причинам некто написал функцию просмотра, возвращающую имя, связанное с номером сотрудника, написанную более сложным образом, с учетом записанных в таблицах имен в верхнем регистре:

```
SQL> create or replace function NameByEmpno(p_empno in number)
2  return varchar2
3  is
4      v_lastname varchar2(30);
5  begin
6      select initcap(lastname)
7      into v_lastname
8      from employees
9      where empno = p_empno;
10     return v_lastname;
11 exception
12     when no_data_found then
13         return '*** UNKNOWN ***';
14 end;
15 /
```

А кто-то другой еще решил использовать эту функцию для возвращения связи «служащий–проект» без явного связывания таблицы employees:

```
SQL> select p.project_name, pa.empno
2  from projects p,
3       project_assignment pa
4  where pa.project_id = p.project_id
5        and namebyempno(pa.empno) = 'Sharp'
6        and pa.from_date < sysdate
7        and (pa.to_date >= sysdate or pa.to_date is null)
8  /
```

PROJECT_NAME	EMPNO
-----	-----
SKUNK	7797
SISYPHUS	2501
DANAIDS	7643
DANAIDS	2501

Анализ показывает, что индексирование по функции может ускорить работу запроса:

```
SQL> create index my_own_index on project_assignment(namebyempno(empno))
2  /
create index my_own_index on project_assignment(namebyempno(empno))
*
```

```
ERROR at line 1:
ORA-30553: The function is not deterministic
```

Наш разработчик модифицирует функцию, добавляет магическое ключевое слово `deterministic` и успешно создает индекс.

В один прекрасный день мистер Crawley сделал мисс Sharp предложение, которое было с благосклонностью принято. После свадьбы сотрудник отдела кадров выполнил следующий запрос на изменение фамилии:

```
SQL> update employees set lastname = 'CRAWLEY' where empno = 2501;

1 row updated.

SQL> commit;

Commit complete.
```

Что теперь происходит при запросе к проекту? Соединение трех таблиц больше не видит сотрудницу по имени Rebecca Sharp, а видит Rebecca Crawley, как и ожидалось:

```
SQL> select e.lastname, e.firstname, p.project_name
2   from employees e,
3        project_assignment pa,
4        projects p
5  where e.lastname = 'SHARP'
6        and e.empno = pa.empno
7        and pa.project_id = p.project_id
8        and pa.from_date < sysdate
9        and (pa.to_date >= sysdate or pa.to_date is null)
10 /
```

```
LASTNAME FIRSTNAME PROJECT_NAME
```

```
-----
SHARP    MELISSA  DANAIDS
SHARP    ERIC     SKUNK
```

```
SQL> select e.lastname, e.firstname, p.project_name
2   from employees e,
3        project_assignment pa,
4        projects p
5  where e.lastname = 'CRAWLEY'
6        and e.empno = pa.empno
7        and pa.project_id = p.project_id
8        and pa.from_date < sysdate
9        and (pa.to_date >= sysdate or pa.to_date is null)
10 /
```

```
LASTNAME          FIRSTNAME          PROJECT_NAME
-----
CRAWLEY           REBECCA           SISYPHUS
CRAWLEY           RAWDON            SISYPHUS
CRAWLEY           REBECCA           DANAIDS
```

Для запроса, использующего функцию и основанного на функции индекса, ничего не изменилось:

```
SQL> select p.project_name, pa.empno
2   from projects p,
3        project_assignment pa
4  where pa.project_id = p.project_id
5        and namebyempno(pa.empno) = 'Crawley'
6        and pa.from_date < sysdate
7        and (pa.to_date >= sysdate or pa.to_date is null)
8  /
```

PROJECT_NAME	EMPNO
SISYPHUS	2503

```
SQL> select p.project_name, pa.empno
2   from projects p,
3        project_assignment pa
4  where pa.project_id = p.project_id
5        and namebyempno(pa.empno) = 'Sharp'
6        and pa.from_date < sysdate
7        and (pa.to_date >= sysdate or pa.to_date is null)
8  /
```

PROJECT_NAME	EMPNO
SKUNK	7797
SISYPHUS	2501
DANAIDS	7643
DANAIDS	2501

Причина проста: в индексах хранятся значения ключей и адреса. Они дублируют часть информации, и СУБД возвращает данные из индексов, когда вся необходимая информация из таблицы в индексах имеется. Обновление коснулось таблицы, но не индекса. Наш текущий ключ хранится в индексе. Мы уже говорили, что функция является детерминированной и что она всегда возвращает одно и то же значение для одного и того же набора параметров. Это означает, что результат никогда не изменится. СУБД верит этому утверждению и при изменении основной таблицы не пытается перестроить индекс или счесть его недействительным (что в любом случае было бы непрактичным) и возвращает неправильные результаты.

Очевидно, что предыдущая функция не может быть детерминированной, поскольку это функция просмотра, выполняющая запрос к базе данных. К счастью, у нас есть и другие средства для усовершенствования функций просмотра. Их мы сейчас и рассмотрим.

Усовершенствование функций поиска

Функции поиска – это благодатная почва для эффективных усовершенствований. Внедряя доступ к базе данных в заранее откомпилированную функцию, творческий разработчик превращает функцию в строительный блок. Можно провести и другую аналогию: черный ящик (для

оптимизатора она выглядит именно так). Когда вы вызываете функцию поиска внутри запроса, то запросы, запущенные внутри этой функции, изолированы от того, что происходит снаружи, даже если они обращаются к тем же таблицам, что и вызвавший их оператор. В частности, если вы будете недостаточно внимательны, запросы, скрытые внутри функции, будут исполняться при каждом ее вызове.

Оценка того, сколько раз функция может быть вызвана, является ключом к оценке ее вклада в общее падение производительности. Это справедливо для любой функции, а особенно для функций просмотра, поскольку даже быстрый доступ к базе данных значительно дороже, чем вычисление математических или строковых выражений. И здесь снова надо понимать разницу между двумя случаями, а именно:

- Функции, на которые есть ссылка внутри списка `select` и которые поэтому вызываются для каждой строки, принадлежащей результирующему набору.
- Функции, на которые есть ссылка внутри фразы `where` и которые могут быть вызваны любое количество раз между общим количеством возвращаемых строк (в лучшем случае) и общим количеством строк, в которых проверяются условия, указанные во фразе `where` (в худшем случае). Реальное количество строк зависит от эффективности других критериев во фразе `where`, отсекающих строки до того, как приходится вычислять функцию для более точного поиска.

Если мы рассмотрим самый плохой случай, когда функция просмотра является единственным критерием во фразе `where`, мы придем, в сущности, к вложенному циклу: таблица будет просканирована и для каждой строки будет вызвана функция, обращающаяся к другой таблице. Даже если запрос SQL в функции очень быстрый, производительность все равно упадет очень сильно. У оптимизатора не будет шанса выбрать другой план исполнения – например, соединение двух таблиц (одной, к которой применяется функция, и другой, запрос к которой происходит из функции) с помощью слияния или хеширования. Хуже того, во многих случаях функция просмотра будет всегда вызываться с теми же параметрами и возвращать те же значения, будучи не действительно детерминированной, а «локально детерминированной» в пределах пользовательского сеанса или пакетного запуска.

Теперь я использую два различных примера функций просмотра и продемонстрирую, как можно их усовершенствовать в некоторых случаях – даже когда они выглядят весьма простыми.

Пример 1: календарная функция

Первая функция называется `NextBusinessDay()`, она получает дату в качестве единственного параметра и возвращает дату, вычисленную следующим образом:

- Если параметр является пятницей, мы добавляем к нему три дня (в стране, где суббота и воскресенье являются нерабочими днями).

- Если параметр является субботой, мы добавляем к нему два дня.
- В остальных случаях мы добавляем один день.
- Затем мы ищем полученный результат в таблице, содержащей все даты общегосударственных праздников. Если мы находим общегосударственный праздник, соответствующий найденной нами дате, мы повторяем алгоритм. В противном случае результат получен.

Вот как я могу запрограммировать эту функцию (не забывая тему номера дня недели) – сначала для Oracle:

```
create or replace function NextBusinessDay(this_day in date)
return date
as
    done boolean := false;
    dow char(3);
    dummy number;
    nextdate date;
begin
    nextdate := this_day;
    while not done
    loop
        dow := substr(to_char(nextdate, 'DAY',
            'NLS_DATE_LANGUAGE=AMERICAN'), 1, 3);
        if (dow = 'FRI')
        then nextdate := nextdate + 3;
        elsif (dow = 'SAT')
        then nextdate := nextdate + 2;
        else nextdate := nextdate + 1;
        end if;
        begin
            select 1
            into dummy
            from public_holidays
            where day_off = nextdate;
        exception
            when no_data_found then
                done := true;
        end;
    end loop;
    return nextdate;
end;
/
```

Теперь – для SQL Server:

```
create function NextBusinessDay(@this_day date)
returns date
as
begin
    declare @done bit;
    declare @dow char(1);
    declare @dummy int;
    declare @nextdate date;
```

```

set @nextdate = @this_day;
set @done = 0;
while @done = 0
begin
    set @dow = datepart(dw, @nextdate);
    if @dow = datepart(dw, convert(date, '01/02/1970', 101))
        set @nextdate = @nextdate + 3;
    else
        if @dow = datepart(dw, convert(date, '01/03/1970', 101))
            set @nextdate = @nextdate + 2;
        else
            set @nextdate = @nextdate + 1;
    set @dummy = (select 1
        from public_holidays
        where day_off = @nextdate);
    if coalesce(@dummy, 0) = 0
    begin
        set @done = 1;
    end;
    return @nextdate;
end;
end;

```

И наконец – для MySQL (здесь мне не нужно беспокоиться о том, что вернет функция dayofweek()):

```

delimiter //
create function NextBusinessDay(this_day datetime)
returns date
reads sql data
begin
    declare done boolean default 0;
    declare dow smallint;
    declare dummy smallint;
    declare nextdate date;
    declare continue handler for not found set done = 1;
    set nextdate = date(this_day);
    while not done do
        set dow = dayofweek(nextdate);
        case dow
            when 6 then set nextdate = date_add(nextdate, interval 3 day);
            when 7 then set nextdate = date_add(nextdate, interval 2 day);
            else set nextdate = date_add(nextdate, interval 1 day);
        end case;
        select 1
        into dummy
        from public_holidays
        where day_off = nextdate;
    end while;
    return nextdate;
end;
//
delimiter ;

```

Я использую всю ту же таблицу с миллионом строк (содержащую даты продаж), что и в примере с подсчетом продаж в выходные и будние дни. Теперь я просто запущу `select NextBusinessDay(sale_date)...` для всех строк моей таблицы.

По понятным причинам (вы их тоже сейчас поймете) для моих тестов я в основном использовал Oracle.

На моей машине простой тест, возвращающий весь миллион строк, занял около двух с половиной минут, то есть 6400 строк в секунду. Не самый худший результат. Но статистика, которую предоставляет Oracle¹, сообщает о двух миллионах рекурсивных вызовов. Сюда входят вызовы функции и запросы внутри функции. Это огромное число, если учесть, что только количество разных дат в таблице составляет 121. Поскольку строки вводятся, скорее всего, в хронологическом порядке², в большинстве случаев результат функции, примененной к строке, оказывается таким же, как и результат для предыдущей строки.

Как я могу увеличить производительность, не трогая код вызова функции? Один из вариантов, если СУБД поддерживает его, заключается в том, чтобы попросить SQL-машину кэшировать результат функции. Кэширование результата функции не столь категорично, как определение функции детерминированной. Оно просто означает, что пока таблицы, с которыми связана функция, остаются немодифицированными, функция может кэшировать результаты и снова возвращать их, если она будет вызвана с теми же параметрами. Такая функция не позволит нам, например, создать индекс по ее результату, но она не будет выполняться без надобности.

В Oracle кэширование результатов функции реализовано начиная с версии Oracle 11g, MySQL 5.1 не кэширует результат запросов, вызванных в функции, а SQL Server 2008 не допускает кэширования, за исключением информации, что функция возвращает `null`, если на входе `null` (что в некоторых случаях может ускорить вычисления). По этой причине я создал свою функцию заново в Oracle, просто модифицировав ее заголовок следующим образом:

```
create or replace function NextBusinessDay(this_day in date)
return date
result_cache relies_on(public_holidays)
as ...
```

Необязательная фраза `relies_on` просто сообщает Oracle, что кэшированный результат нужно будет удалить, если таблица `public_holidays` будет модифицирована.

¹ `set autotrace traceonly stat` в SQL*Plus.

² Я сгенерировал случайные данные и затем отсортировал их по дате для создания реалистичной картины.

Запуск того же теста выдал еще более худший результат: производительность снизилась примерно на 30%, до 4400 строк в секунду. Причина та же, что и в случае детерминированной функции: поскольку в датах хранится также и время, которое каждый раз иное, функция почти никогда не вызывается с одним и тем же параметром. Кэширование не принесло нам выгоды, но мы несем накладные расходы на его реализацию.

Если я не хочу модифицировать исходный код, я могу и «смошенничать». Я могу переименовать мою предыдущую кэшированную функцию в `Cached_NextBusinessDay()` и переписать функцию, вызываемую в операторе, следующим образом:

```
create or replace function NextBusinessDay(this_day in date)
return date
as
begin
    return(Cached_NextBusinessDay(trunc(this_day)));
end;
```

При запуске того же запроса сканирование моей таблицы заняло чуть больше 30 секунд, то есть производительность возросла до 31 700 строк в секунду – почти в пять раз по сравнению с исходной. Это соотношение близко к полученному после объявления функции детерминированной – без всех ограничений, связанных с реально детерминированной функцией. Если вы хотите усовершенствовать функцию просмотра, кэширование ее результатов может помочь.

Но что делать, если используемая СУБД не поддерживает кэширование результатов функции? Есть еще несколько приемов, позволяющих использовать то, что строки хранятся в более или менее хронологическом порядке. Например, каждая версия Oracle начиная с Oracle 7 (выпущенной в начале 1990-х) поддерживает использование переменных в пакетах для кэширования предыдущих результатов. Пакетные переменные видны только в пределах одного сеанса. Сначала вы объявляете вашу функцию внутри пакета:

```
create or replace package calendar
is
    function NextBusinessDay(this_day in date)
    return date;
end;
/
```

Теперь надо создать тело пакета, в котором в двух переменных, предварительно инициализированных, хранятся последний входной параметр и последний результат. Не забывайте, что время жизни кэша ограничено сеансом. В некоторых случаях вам потребуется создать механизм его продления. Функция сравнивает текущий входной параметр с предыдущим и возвращает предыдущий результат, если эти параметры идентичны. В противном случае запускается запрос, который является ресурсоемкой частью функции:

```

create or replace package body calendar
is
    g_lastdate date := to_date('01/01/0001', 'DD/MM/YYYY');
    g_lastresult date := NULL;
    function NextBusinessDay(this_day in date)
    return date
    is
        done boolean := false;
        dow char(3);
        dummy number;
    begin
        if (this_day <> g_lastdate)
        then
            g_lastdate := this_day;
            g_lastresult := this_day;
            while not done
            loop
                dow := substr(to_char(g_lastresult, 'DAY',
                                     'NLS_DATE_LANGUAGE=AMERICAN'), 1, 3);
                if (dow = 'FRI')
                then g_lastresult := g_lastresult + 3;
                elsif (dow = 'SAT')
                then g_lastresult := g_lastresult + 2;
                else g_lastresult := g_lastresult + 1;
                end if;
                begin
                    select 1
                    into dummy
                    from public_holidays
                    where day_off = g_lastresult;
                exception
                    when no_data_found then
                        done := true;
                end;
            end loop;
        end if;
        return g_lastresult;
    end;
end;

```

Наконец, чтобы сохранить интерфейс таким же, я переписал исходную функцию в виде простой оболочки, которая после удаления части, содержащей время, вызывает функцию в пакете:

```

create or replace function NextBusinessDay(this_day in date)
return date
as
begin
    return calendar.NextBusinessDay(trunc(this_day));
end;

```

Для тех, кто использует более старые версии Oracle, хорошей новостью будет, что в этом случае (то есть при обработке дат в более или менее хронологическом порядке) результат оказывается даже лучше, чем когда я позволяю Oracle управлять кэшем: мой тест оказался почти на 10% быстрее, достигнув производительности 34 868 строк в секунду. Однако не забывайте, что кэш в Oracle не зависит от порядка дат, но количество дат ограничено. На рис. 3.2 показана производительность разных вариантов функции `NextBusinessDay()`.

Интересно, что фраза `result_cache` может как значительно снизить производительность, так и поднять ее, в зависимости от того, в каких обстоятельствах она используется.

Можем ли мы применить какие-либо методы усовершенствования, работающие с Oracle, к другим продуктам? Смотря по обстоятельствам. Возможность явного указания сохранять результат функции в кэше на момент написания была особенностью Oracle как пакета. К сожалению, SQL Server не предоставляет способа ручного кэширования результатов функций, написанных на T-SQL, поскольку он знает лишь о локальных переменных, которые существуют только во время исполнения вызова функции, и поскольку на временные таблицы, которые ближе всего к глобальным переменным, из функции ссылаться нельзя.

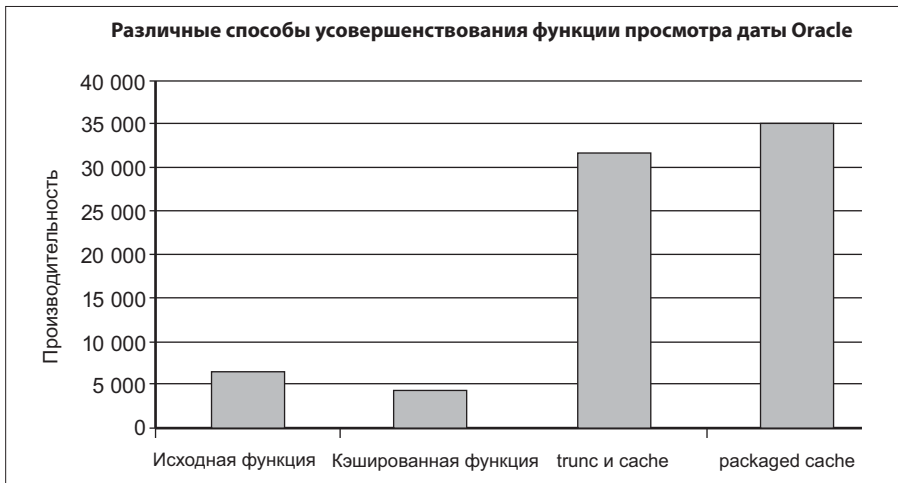


Рис. 3.2. Как меняется производительность одних и тех же функций просмотра Oracle

Хотя MySQL 5.1 может кэшировать результаты запросов, эта СУБД не может кэшировать результаты запросов, запущенных из функции или процедуры. Однако, в противоположность T-SQL, MySQL позволяет ссылаться на глобальные переменные сеанса из функции, а переменные сеанса можно использовать для буферизации результатов. Единственное затруднение заключается в том, что если пакетные

переменные Oracle являются частными, когда они объявлены в теле пакета и не видны за пределами пакета, то переменные сеанса MySQL доступны любой функции. Присвоение им стандартных имен (например, @var1) чревато проблемами, так что в целях безопасности лучше использовать длинные имена, снижающие риск случайной путаницы. В следующем примере я постоянно использую `<function name>$` как префикс:

```
delimiter //
create function NextBusinessDay(this_day datetime)
returns date
reads sql data
begin
    declare done        boolean default 0;
    declare dow          smallint;
    declare dummy        smallint;
    declare continue handler for not found set done = 1;
    if (ifnull(@NextBusinessDay$last_date, '1769-08-15') <> date(this_day))
    then
        set @NextBusinessDay$last_date = date(this_day);
        set @NextBusinessDay$last_business_day = date(this_day);
        while not done do
            set dow = dayofweek(@NextBusinessDay$last_business_day);
            case dow
                when 6 then
                    set @NextBusinessDay$last_business_day =
                        date_add(@NextBusinessDay$last_business_day, interval
3 day);
                when 7 then
                    set @NextBusinessDay$last_business_day =
                        date_add(@NextBusinessDay$last_business_day,
interval 2 day);
                else set @NextBusinessDay$last_business_day =
                    date_add(@NextBusinessDay$last_business_day,
interval 1 day);
            end case;
            select 1
            into dummy
            from public_holidays
            where day_off = @NextBusinessDay$last_business_day;
        end while;
    end if;
    return @NextBusinessDay$last_business_day;
end;
//
delimiter ;
```

Как вы можете видеть на рис. 3.3, реализация в MySQL тех же идей, что и в Oracle, тоже дает значительный прирост производительности при сканировании строк, расположенных в хронологическом порядке.

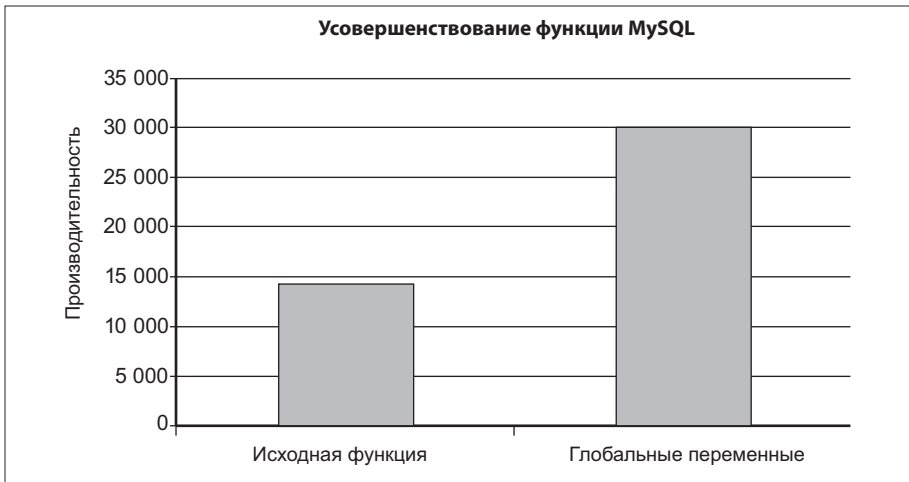


Рис. 3.3. Использование переменных для кэширования результатов функции в MySQL

Пример 2: функция конвертирования валют

Вторая функция, `FxConvert()`¹, принимает два параметра – сумму и код валюты и возвращает сумму, сконвертированную в заранее определенную валюту по самому свежему курсу. Здесь я использую таблицы примера из первой главы. Вы можете использовать функцию `FxConvert()` как прототип для функций преобразования.

Вот исходный код для Oracle:

```
create or replace function FxConvert(p_amount in number,
                                     p_currency in varchar2)
return number
is
    n_converted number;
begin
    select p_amount * a.rate
    into n_converted
    from currency_rates a
    where (a.iso, a.rate_date) in
        (select iso, max(rate_date) last_date
        from currency_rates
        where iso = p_currency
        group by iso);
    return n_converted;
exception
    when no_data_found then
        return null;
```

¹ FX является стандартной аббревиатурой для Foreign eXchange, рынка валют.


```
end;
/
```

Вот код для SQL Server:

```
create function fx_convert(@amount float,
    @currency char(3))
returns float
as
begin
    declare @rate float;
    set @rate = (select a.rate
        from currency_rates a
        inner join (select iso,
            max(rate_date) last_date
        from currency_rates
        where iso = @currency
        group by iso) b
        on a.iso = b.iso
        and a.rate_date = b.last_date);
    return coalesce(@rate, 0) * @amount;
end;
```

А здесь код для MySQL:

```
delimiter //
create function FxConvert(p_amount float, p_currency char(3))
returns float
reads sql data
begin
    declare converted_amount float;
    declare continue handler for not found set converted_amount = 0;
    select a.rate * p_amount
    into converted_amount
    from currency_rates a
    inner join (select iso, max(rate_date) last_date
        from currency_rates
        where iso = p_currency
        group by iso) b
    on a.iso = b.iso
    and a.rate_date = b.last_date;
    return converted_amount;
end;
//
delimiter ;
```

Давайте начнем, как и прежде, с Oracle. У нас есть, по сути, два способа усовершенствования функции: либо использовать ключевое слово `result_cache` для Oracle 11g или более поздней версии, либо использовать пакетные переменные и «вручную» реализовывать кэширование.

Если я хочу извлечь пользу из `result_cache`, потребуется модифицировать функцию так, чтобы количество входных параметров было ограничено. Поскольку может быть большое количество различных сумм, значение суммы передаваться функции не будет. Поэтому я переопределил ее как оболочку для реальной функции, результаты которой будут кэшироваться:

```
create or replace function FxRate(p_currency in varchar2)
return number
result_cache relies_on(currency_rates)
is
    n_rate number;
begin
    select rate
    into n_rate
    from (select rate
          from currency_rates
          where iso = p_currency
          order by rate_date desc)
    where rownum = 1;
    return n_rate;
exception
    when no_data_found then
        return null;
end;
/
create or replace function FxConvert(p_amount in number,
                                     p_currency in varchar2)
return number
is
begin
    return p_amount * FxRate(p_currency);
end;
/
```

Если я хочу реализовать кэширование с помощью пакетных переменных, нельзя использовать две переменные, как я делал в примере `NextBusinessDay()`. Я полностью полагался на факт, что функция была успешно вычислена для той же самой даты. В этом примере, даже при наличии ограниченного количества валют, их коды будут чередоваться (в отличие от ситуации в предыдущем примере), и я не смогу получить выгоды от «запоминания» предыдущего значения.

Вместо простых переменных я использую ассоциативный массив PL/SQL, проиндексированный по коду валюты:

```
create or replace package fx
as
    function rate(p_currency in varchar2)
    return number;
end;
```

```

/
create or replace package body fx
as
    type t_rates is table of number
        index by varchar2(3);
    a_rates t_rates;
    n_rate number;
    function rate(p_currency in varchar2)
    return number
    is
        n_rate number;
    begin
        begin
            n_rate := a_rates(p_currency);
        exception
            when no_data_found then
                begin
                    select a.rate
                    into a_rates(p_currency)
                    from currency_rates a
                    where (a.iso, a.rate_date) in
                        (select iso, max(rate_date) last_date
                        from currency_rates
                        where iso = p_currency
                        group by iso);
                    n_rate := a_rates(p_currency);
                end;
            end;
        return n_rate;
    end;
end;
/
create or replace function FxConvert(p_amount in number,
p_currency in varchar2)
return number
is
begin
    return p_amount * fx.rate(p_currency);
end;
/

```

Я протестировал альтернативные версии функции, запустив следующий запрос с таблицей transactions, содержащей два миллиона строк. Исходная функция сканировала 100 строк в секунду:

```

select sum(FxConvert(amount, curr))
from transactions;

```

Результат показан на рис. 3.4.

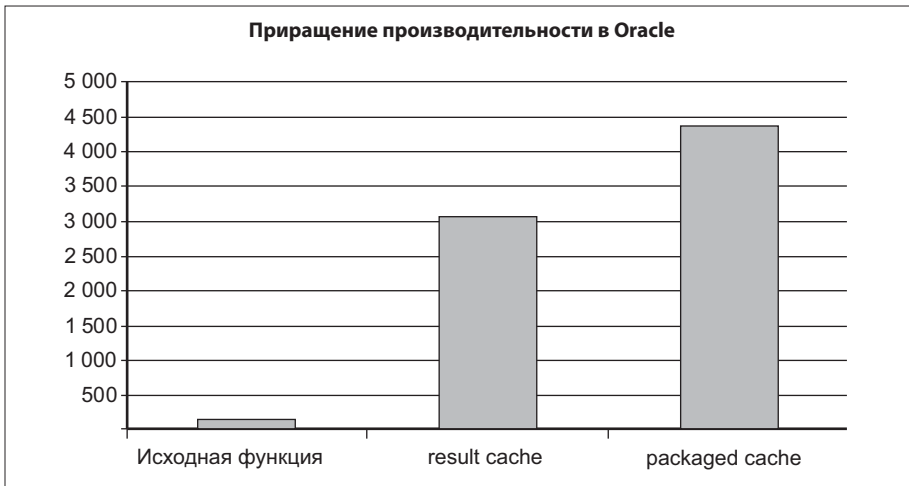


Рис. 3.4. Сравнение альтернативных способов написания функции преобразования валюты в Oracle

Когда нет доступа к коду, изменения, которые я только что продемонстрировал, приведут к впечатляющему ускорению: простое разбиение функции для использования кэша результатов увеличило скорость в 30 раз на моей машине, а более продуманное использование кэшированного массива внутри пакета увеличило скорость в 45 раз. Но не забывайте, что функция является черным ящиком, и если вы действительно хотите добиться впечатляющего ускорения в Oracle, у вас нет другого выхода, кроме как переписать тестовый запрос следующим образом:

```
select sum(t.amount * r.rate)
from transactions t
  inner join
    (select a.iso, a.rate
     from (select iso,
                  max(rate_date) last_date
           from currency_rates
          group by iso) b
     inner join currency_rates a
       on a.iso = b.iso
       and a.rate_date = b.last_date) r
on r.iso = t.curr;
```

Вы, вероятно, сочтете, что этот запрос труднее для восприятия, чем тот, в котором используется функция, но на самом деле подзапрос, который следует после первой фразы `inner join`, — это не что иное, как подзапрос из функции. Я использую `inner join`, поскольку функция игнорирует валюты, для которых курс неизвестен, поэтому нам не нужно беспокоиться о нескольких строках, ссылающихся на очень экзотические валюты. Рис. 3.5, который отличается от рис. 3.4 только заменой вызова функции на соединение, вероятно, комментариев не требует.

А как обстоит дело с другими продуктами?

Мы уже видели, что варианты ограничены функциями T-SQL. Я упоминал, что T-SQL не знает о переменных сеанса. Вам придется прибегнуть к хитрости и использовать управляемый код (подпрограмму языка .NET, вызываемую SQL-машиной). Например, вы можете использовать статические переменные в C#. Но лучше сразу выбросить такую идею из головы. Разработчики Microsoft решили сделать в интерфейсах между T-SQL и кодом на языке .NET SAFE значением по умолчанию для PERMISSION_SET.

Поэтому, если вы попытаетесь использовать статические функции в функции с управляемым кодом, то получите следующее сообщение об ошибке:

```
CREATE ASSEMBLY failed because type '...' in safe assembly '...' has  
a static field '...'. Attributes of static fields in safe assemblies  
must be marked readonly in Visual C#, ReadOnly in Visual Basic, or  
initonly in Visual C++ and intermediate language.
```

Вы можете использовать статические переменные. Но проблема в DLL, они совместно используются всеми сеансами. При конкуренции между запросами может произойти все, что угодно.

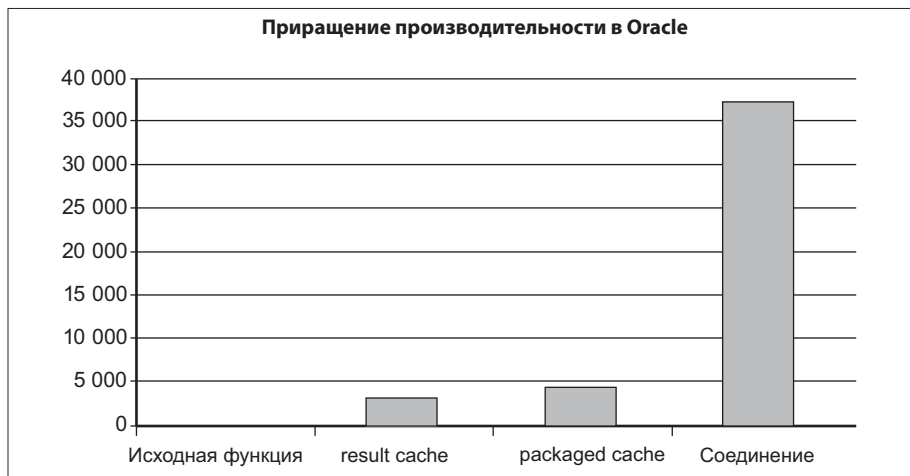


Рис. 3.5. Сравнение различных вариантов замены функции на простое соединение в Oracle

Чтобы продемонстрировать это (на этот раз я не буду приводить код), я протестировал функцию на языке C#, конвертирующую валюты с помощью двух статических переменных и «запоминающую» курсы валют и последнюю встретившуюся валюту (в данном случае этот метод неэффективен, но эффективность сейчас не является моей целью). Я суммировал проконвертированные суммы из моей таблицы transactions с двумя миллионами строк.

Если вы запустите запрос, когда ваш сеанс является единственным, вы получите, но с меньшей скоростью, тот же результат, что и с функцией T-SQL:

```
782780436669.151
```

Теперь, запустив тот же запрос с функцией на C# со статическими переменными при тех же данных в трех конкурентных сеансах, вы получите следующие результаты:

```
782815864624.758
```

```
782797493847.697
```

```
782816529963.717
```

Что произошло? Каждый сеанс просто переписывал статические переменные, путая результат, – вот пример нескоординированного конкурентного доступа к одной и той же области памяти.

Если рефакторинг функции, причем только одной функции, не является вариантом в случае SQL Server, часто возможна замена функции просмотра на соединение. Сравнение производительности запроса, вызывающего функцию, с запросом, использующим соединение, хоть и не столь впечатляюще, как в случае с Oracle, тем не менее дает надежду на увеличение скорости при использовании функции, которая может быть объединена внутри запроса. Результат моих тестов показан на рис. 3.6.

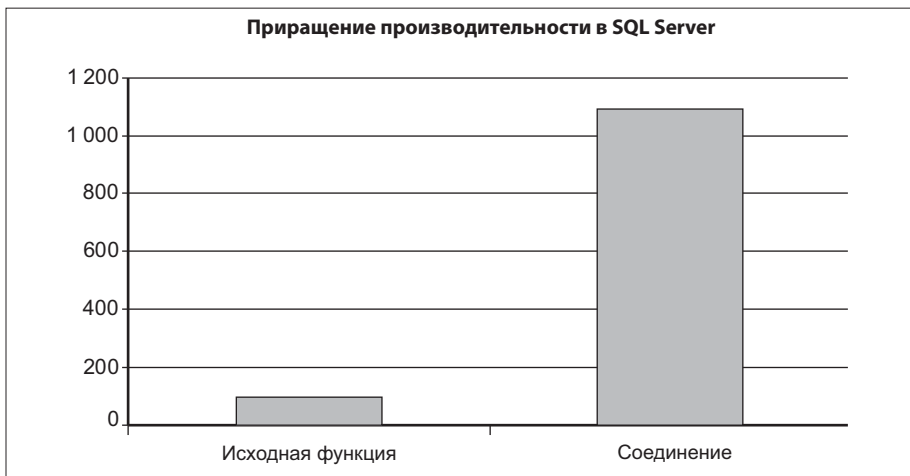


Рис. 3.6. Сравнение производительности SQL Server при последовательном вызове простой функции просмотра и при использовании соединения

Как мы уже видели, с MySQL вы можете использовать переменные сеанса. Однако в случае валют, когда нам нужно запоминать несколько их курсов, придется проявить некоторую фантазию, поскольку MySQL не знает переменных-массивов. Вместо того чтобы использовать массивы, я конкатенировал коды валют в одну строку, а соответствующие курсы

валют – в другую. Встроенные функции позволяют мне идентифицировать позицию нужного кода в первой строке и найти соответствующий курс валют, преобразовав строку в значение с плавающей точкой, добавив 0.0. Для безопасности я определил максимальную длину строк:

```
delimiter //
create function FxConvert(p_amount float, p_currency char(3))
returns float
reads sql data
begin
    declare pos smallint;
    declare rate float;
    declare continue handler for not found set rate = 0;
    set pos = ifnull(find_in_set(p_currency,
                                @FxConvert$currency_list), 0);

    if pos = 0
    then
        select a.rate
        into rate
        from currency_rates a
             inner join (select iso, max(rate_date) last_date
                        from currency_rates
                        where iso = p_currency
                        group by iso) b
                        on a.iso = b.iso
                        and a.rate_date = b.last_date;
        if (ifnull(length(@FxConvert$rate_list), 0) < 2000
            and ifnull(length(@FxConvert$currency_list), 0) < 2000)
        then
            set @FxConvert$currency_list = concat_ws(',',
                @FxConvert$currency_list,
                p_currency);
            set @FxConvert$rate_list = concat_ws('|',
                @FxConvert$rate_list,
                rate);
        end if;
    else
        set rate = 0.0 + substring_index(substring_index(@FxConvert$rate_list,
            '|', pos), '|', -1);
    end if;
    return p_amount * rate;
end;
//
delimiter ;
```

Я не претендую на то, чтобы считать получившийся код элегантным, и не утверждаю, что ручная работа с ассоциативными массивами – это просто. Но рост производительности налицо. И что интересно, переписанная функция работает даже немного лучше, чем соединение, как вы можете видеть на рис. 3.7.

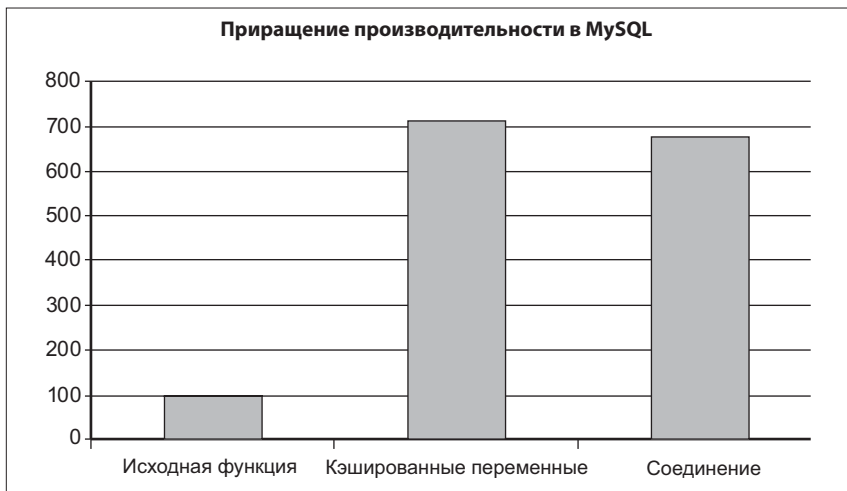


Рис. 3.7. Значительно переписанная функция конвертирования для MySQL

Усовершенствование функций против переписывания операторов

Как показал предыдущий пример с конвертированием валют, иногда оказывается значительно эффективнее переписать оператор, чтобы полностью избавиться от функции. Вы можете переделать доступ к базе данных, прежде реализованный в виде функции, с использованием соединений или подзапросов – это легко реализовать в случае простой функции, но сложнее, если функция содержит процедурную логику (как перенести процедурную логику в операторы, мы будем обсуждать в шестой главе). Вот два главных преимущества устранения функций:

- Вам придется модифицировать единственный запрос. Изменения локализованы, что упрощает проверку правильности изменений и тестирование производительности.
- Добавляя вспомогательные операторы SQL в главный оператор, вы превращаете оптимизатор базы данных в вашего союзника. У оптимизатора будет полное представление о том, чего вы хотите достичь, и он попытается выполнить глобальную оптимизацию вместо независимой оптимизации каждой части.

А вот два главных недостатка:

- Внесенные усовершенствования принесут выгоду только этому конкретному процессу, даже если функция используется во многих местах.
- Вы нарушаете исходную задачу функции – централизацию кода для упрощения обслуживания.

Ваш выбор иногда может быть ограничен политическими причинами: если вам заказали рефакторинг приложения А, и вы понимаете, что можете очень серьезно увеличить производительность, переработав функцию F, которая осуществляет запрос к таблицам из другой функциональной области, у вас может не оказаться прямого доступа к этим таблицам. И если функция F является краеугольным камнем в другой функциональной области, приложений В и С, в которых проблем с производительностью на этот момент не обнаруживалось, лучше не трогать функцию F, чтобы не вызвать личных проблем. В этом случае добавление специальной функции-оболочки FA, вызывающей исходную функцию F, часто оказывается наилучшим решением, если вы сможете ограничить в функции FA количество вызовов функции F.

Я уже указывал в начале этой главы, что представления являются в SQL эквивалентом функций. Во многих случаях (хорошим примером служит вышеприведенное конвертирование валют) создание представления на основе соединения для замены функции может предоставить простой и эффективный интерфейс. Но даже представления могут иногда негативно повлиять на производительность, как вы вскоре убедитесь.

Представления

Я часто встречаю ситуации, когда представления (views) вовлечены в проблемы с производительностью. Повторю, что представления являются в SQL эквивалентом функций. Но в случаях глубоко вложенных функций и бестолковой передачи параметров они могут повлиять на производительность. Во многих случаях анализ проблем с производительностью приводит к представлениям.

Для чего нужны представления

Представления могут служить нескольким целям. Простейшей (и самой безобидной) их ролью является сужение видимости данных до нескольких столбцов или строк или комбинации столбцов и строк. Поскольку во всех СУБД есть информационные функции, извлекающие данные текущего конкретного пользователя (например, его регистрационное или системное имя), можно создать общее представление, которое будет возвращать данные в зависимости от прав доступа пользователя. В таком представлении к основной таблице просто добавлены условия фильтрации – сколько-то заметного влияния на производительность они не оказывают.

Но представления могут быть также и хранилищами для сложных, многократно используемых операций SQL. Тогда, если мы забываем, что имеем дело не с основной таблицей, а с представлением, в некоторых случаях могут возникать серьезные проблемы с производительностью.

В этом разделе я покажу, как представление даже средней сложности может повлиять на производительность, с помощью двух примеров, обращающихся к двум таблицам из первой главы: таблице `transactions`, содержащей суммы множества транзакций в различных валютах, и таблице `currency_rates`, хранящей курсы обмена валют на различные даты.

Сравнение производительности со сложными представлениями и без них

Сначала давайте создадим «сложное» представление, которое является не более чем суммой транзакций по валютам (в реальном мире, вероятно, будут дополнительные условия на даты, но отсутствие такого условия в данном примере ничего не меняет):

```
create view v_amount_by_currency
as select curr, round(sum(amount), 0) amount
   from transactions
  group by curr;
```

Я не стал выбирать представление, в котором значения группируются случайным образом. Проблема с группировкой и фильтрацией заключается в том, что вы делаете сначала: либо группируете большой объем данных, а потом осуществляете фильтрацию результата, либо вы сначала фильтруете, а потом группируете значительно меньший объем данных. Иногда выбора у вас нет (например, когда вы хотите отфильтровать результат группировки с помощью фразы `having`). Но когда выбор есть, то от того, что вы сделаете в первую очередь, производительность зависит очень сильно.

Вопрос заключается в том, что происходит, когда вы запускаете запрос, подобный следующему:

```
select *
from v_amount_by_currency
where curr = 'JPY';
```

У вас есть два варианта. Вы можете выполнить фильтрацию после группировки:

```
select curr, round(sum(amount), 0) amount
from transactions
group by curr
having curr = 'JPY';
```

Или вы можете вставить фильтрацию внутрь представления:

```
select curr, round(sum(amount), 0) amount
from transactions
where curr = 'JPY'
group by curr;
```

Назовем последний запрос, который я буду использовать в роли эталона, «оптимальным».

Что происходит, когда мы запрашиваем данные с помощью представления, фактически зависит от СУБД (и, конечно, версии СУБД). MySQL 5.1 запускает оператор представления и затем применяет фильтр, как показывает команда explain:

```
mysql> explain select * from v_amount_by_currency
-> where curr = 'JPY';
+----+-----+-----+-----+-----+-----+
| id | select_type | table      | //      |      |      |
+----+-----+-----+-----+-----+-----+
| 1  | PRIMARY    | <derived2> | //      | 170  | Using where |
| 2  | DERIVED    | transactions | //2000421 |      | Using temporary; Using filesort |
+----+-----+-----+-----+-----+-----+
2 rows in set (3.20 sec)
```

В MySQL существует расширение оператора create view, algorithm=merge, которое вы можете использовать, чтобы заставить MySQL объединить представление внутри оператора, но group by делает его неэффективным.

В Oracle план исполнения показывает, что оптимизатор достаточно интеллектен, чтобы сочетать представление и условие фильтрации, и что он фильтрует таблицу по мере сканирования до сортировки результата и выполнения группировки:

```
-----+-----+-----+-----+-----+-----+
| Id | Operation          | Name          | Rows | Bytes | Cost (%CPU)|...
-----+-----+-----+-----+-----+-----+
| 0  | SELECT STATEMENT   |               |      |      |           |...
| 1  | SORT GROUP BY NOSORT|               |      |      |           |...
|* 2 | TABLE ACCESS FULL | TRANSACTIONS  | 276K | 3237K | 2528 (3)   |...
-----+-----+-----+-----+-----+-----+
Predicate Information (identified by operation id):
-----+-----+-----+-----+-----+
2 - filter("CURR"='JPY')
```

Совсем по-другому ведет себя Oracle, когда встречается фраза having в обычном операторе. «Оптимистичный оптимизатор» Oracle предполагает, что разработчики знают, что они пишут, и фраза having всегда применяется после группировки. Подобно оптимизатору Oracle, оптимизатор SQL Server помещает условие where, применяемое к представлению, внутрь представления. Однако будучи менее уверенным «в человеческой природе», чем Oracle, оптимизатор SQL Server способен также вставить условие, которое может быть применено до группировки, внутрь фразы where, когда оно появляется во фразе having.

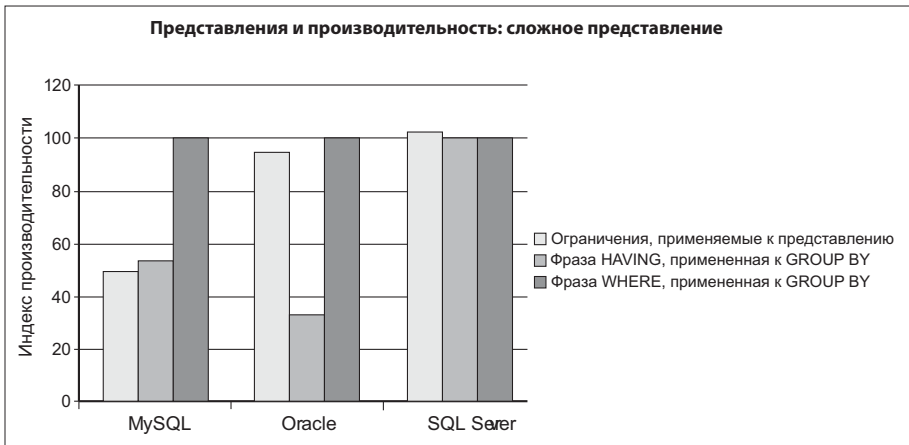


Рис. 3.8. Как запрос к сложному представлению можно сравнить с запросом к таблице

На рис. 3.8 показано, как работает представление по сравнению с «оптимальным» запросом, который осуществляет фильтрацию перед группированием, и запросом, в котором используется фраза `having` (хотя делать этого не стоило). Серьезное падение производительности происходит в MySQL как при использовании представления, так и при неверном использовании фразы `having`. В Oracle сильное падение производительности наблюдается при неправильной фильтрации с использованием фразы `having`, но нет почти никаких потерь при использовании представления. С SQL Server производительность одинакова во всех трех случаях.

Однако обратите внимание, что причиной падения производительности в MySQL является также применение условия к столбцу, управляющему группировкой: столбцу валют. У меня не было (почти) потерь, если условие применялось к агрегированной сумме, поскольку в этом случае фраза `where`, которая применялась к представлению, была строго эквивалентна фразе `having`.

Сложность предыдущего представления была довольно средней. Я создал «более сложное» представление, ссылающееся на вторичное служебное представление. Давайте начнем с этого вторичного представления, возвращающего для каждой валюты самый последний курс:

```
create view v_last_rate
as
select a.iso, a.rate
from currency_rates a,
     (select iso, max(rate_date) last_date
      from currency_rates
      group by iso) b
where a.iso= b.iso
and a.rate_date = b.last_date;
```

Теперь я могу построить более сложное представление, которое возвращает конвертированную сумму для каждой валюты, но отделяет валюты, которые наиболее важны для работы банка, а все валюты, представляющие малую часть оборотов, группирует вместе как OTHER:

```
create view v_amount_main_currencies
as
select case r.iso
  when 'EUR' then r.iso
  when 'USD' then r.iso
  when 'JPY' then r.iso
  when 'GBP' then r.iso
  when 'CHF' then r.iso
  when 'HKD' then r.iso
  when 'SEK' then r.iso
  else 'OTHER'
end currency,
round(sum(t.amount*r.rate), 0) amount
from transactions t,
v_last_rate r
where r.iso = t.curr
group by case r.iso
  when 'EUR' then r.iso
  when 'USD' then r.iso
  when 'JPY' then r.iso
  when 'GBP' then r.iso
  when 'CHF' then r.iso
  when 'HKD' then r.iso
  when 'SEK' then r.iso
  else 'OTHER'
end;
```

Теперь я запускаю тот же самый тест, что и с более простой группировкой, сравнивая производительность следующего фрагмента кода с производительностью двух функционально эквивалентных запросов, в которых используется служебное представление:

```
select currency, amount
from v_amount_main_currencies
where currency = 'JPY';
```

Сначала я помещаю условие по валюте во фразу having:

```
select t.curr, round(sum(t.amount) * r.rate, 0) amount
from transactions t,
v_last_rate r
where r.iso = t.curr
group by t.curr, r.rate
having t.curr = 'JPY';
```

Затем я помещаю условие по валюте во фразу where:

```
select t.curr, round(sum(t.amount) * r.rate, 0) amount
from transactions t,
```

```

v_last_rate r
where r.iso = t.curr
and t.curr = 'JPY'
group by t.curr, r.rate;

```

Результаты вы можете видеть на рис. 3.9. Как можно было ожидать, MySQL вычисляет группы в представлении, а затем производит фильтрацию (так же, как с фразой *having*). Производительность по сравнению с запросом, делающим выборку из таблицы с условием фильтрации, находящимся там, где и нужно было, весьма низкая.

Но самое интересное в этом неудачном случае то, что даже оптимизаторы Oracle и SQL Server не могут обеспечить такую же производительность запроса с представлением, которая может быть получена при корректном запросе к таблице. Выполнение запроса к представлению требует почти вдвое больше времени, чем оптимальный запрос к таблице.

Эти примеры представлений по-прежнему можно считать простыми – по сравнению с некоторыми встречавшимися мне весьма необычными представлениями. Даже самый интеллектуальный оптимизатор запутается, если представление становится слишком сложным или представления нагромождаются друг на друга.

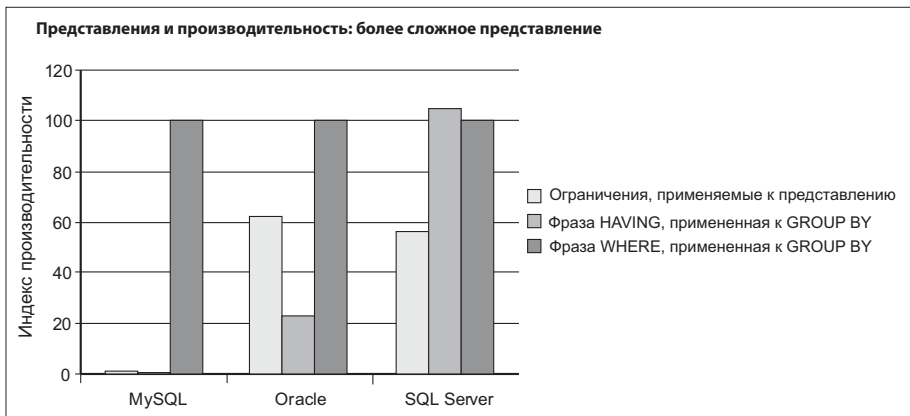


Рис. 3.9. Сравнение производительности с более сложным представлением

Не стоит упрекать оптимизатор, иногда он просто не может решить, что сделать для увеличения производительности.

Например, предположим, что у нас есть представление, определенное как соединение:

```

create view v(c1, c2, c3, c4, c5) as
select a.c1, a.c2, a.c3, b.c4, b.c5
from a, b
where a.c6 = b.c6

```

Теперь предположим, что в запросе нам нужны только столбцы *c1* и *c2* из таблицы *a*.

Нужно ли нам соединение с таблицей *b*? Ответ может быть только «да» или «нет»:

Нет

Нам не нужно соединение с таблицей *b*, если столбец *c6* одновременно является и обязательным столбцом, и внешним ключом, который должен всегда иметь соответствие в таблице *b*. Например, можно представить, что таблица *a* содержит закупки, в столбце *c6* находятся коды клиентов, а в столбцах *c4* и *c5* хранятся имена и города проживания клиентов.

Да

Нам нужно соединение с таблицей *b*, если столбец *c6* может содержать значение *null* или он не должен обязательно ссылаться на строку из таблицы *b*, поскольку в таком случае соединение не только предоставляет дополнительную информацию (*c4* и *c5*), но и служит неявным дополнительным критерием фильтрации, требующим, чтобы значения *a.c6* не были равны *null* (если бы они были равны *null*, соответствия быть не могло бы, поскольку *null* никогда не может быть равным чему-либо, даже *null*) и чтобы оно соответствовало существующей строке в таблице *b*.

Другими словами, иногда соединение просто дает дополнительные столбцы, где:

```
select c1, c2 from v
```

вернет в точности те же строки, что и:

```
select c1, c2 from a
```

А в других случаях запрос к представлению вернет меньше строк, чем оператор *select* к таблице, поскольку соединение добавляет условие фильтрации. Иногда человек в состоянии определить, что нужно, а что нет для получения нужного результирующего набора, а оптимизатор (очень часто из-за отсутствия соответствующей информации) не сможет сделать этого. При наличии сомнений СУБД будет всегда выбирать самый безопасный вариант. Так, если вы ссылаетесь на представление, оптимизатор будет предполагать (если не сможет принять другое решение), что представление может возвращать результирующий набор, причем отличающийся от результирующего набора, возвращаемого более простым запросом, и будет использовать представление.

Чем сложнее представление, тем больше в нем неоднозначностей. Даже самый совершенный оптимизатор не во всех случаях сможет решить, как лучше всего объединять представление с запросом. Кроме того, сложный запрос означает очень большое количество вариантов, которые оптимизатор не сможет исследовать за ограниченный период времени. Всегда существует порог, за которым оптимизатор сдается и СУБД исполняет представление «как есть» внутри другого запроса.

Ранее я упоминал соединения, которые возвращают столбцы, не используемые в запросе, что в некоторых условиях приводит к бесполезной дополнительной работе. Есть много и других случаев, особенно в случае соединения представления с запросом к таблице, на которую уже есть ссылка из представления.

Например, предположим, что запрос построен на основе внешнего соединения:

```
select a.c1, a.c2, a.c3, b.c4, b.c5
from a
      left outer join b
        on b.c6 = a.c3
```

Дальше представим, что это представление соединено (с помощью внутреннего соединения) внутри запроса с таблицей *b*, чтобы получить из таблицы *b* другую информацию, отсутствующую в представлении. Внутреннее соединение в запросе является более сильным условием, чем внешнее – в представлении. В контексте запроса внешнее соединение в запросе могло бы быть и внутренним: строки, для которых нет соответствия между *a* и *b*, будут возвращены представлением, но отфильтрованы запросом. Каждая дополнительная строка, возвращенная внешним соединением, является потерей времени в контексте запроса, и внутреннее соединение оказывается более эффективным, чем внешнее.

Можно найти подобные случаи с представлениями, построенными на объединении нескольких запросов. Они постоянно используются в запросах с такими критериями фильтрации, что результирующий набор может содержать только строки, полученные из очень малой части результирующего набора представления, и мы могли бы избавиться от нескольких потенциально дорогих операторов *select*.

Есть и другой, не менее неприятный, эффект представлений. Он возникает тогда, когда ничего не подозревающий программист в качестве основного метода использует поиск или соединение по столбцам, вычисляемым «на лету». Предположим, например, что один столбец в таблице содержит (по историческим причинам) метку времени Unix (количество секунд, прошедших после полуночи 1 января 1970 года). Это единственная таблица в схеме, которая содержит дату в формате ином, чем обычная дата SQL, поэтому разработчик решил создать простое представление, ссылающееся на эту таблицу и преобразующее дату в обычный формат SQL.

Если доверчивый программист наивно напишет условие, подобное нижеприведенному:

```
where view_date_column = convert_to_date('date string')
```

реально будет выполнено следующее:

```
where convert_from_unix_timestamp(timestamp) = convert_to_date('date string')
```


Это прекрасный пример опасности того, что функция применяется к индексированному столбцу – и это препятствует использованию индекса. Если отсутствует вычисленный индекс (что могут позволить себе Oracle и SQL Server) по конвертированной метке времени¹, то при любом условии, связанном с датой и примененном к представлению, преимущества существующего индекса по исходной метке времени будут потеряны.

Некоторые разработчики пытаются скрыть недостатки плохого проектирования баз данных за представлениями и сделать вид, будто схема более нормализованная, чем это есть на самом деле, разбивая, например, на части один столбец, который содержит несколько фрагментов информации. Такие попытки приводят к тому, что фильтры и соединения будут основаны не на реальных и проиндексированных, а на производных столбцах.

Рефакторинг представлений

Многие представления безобидны, но если анализ показывает, что некоторые из проблемных запросов используют сложные представления, на них нужно обратить внимание в первую очередь. А поскольку сложные представления с низкой производительностью часто являются комбинацией нескольких представлений, имеет смысл переписать те из них, что построены на базе других – если они появляются в проблемных запросах. Во многих случаях вам удастся увеличить производительность запросов, не затрагивая их самих, а просто переопределив представления, на которые они ссылаются. Сложные представления нужно максимально упростить, чтобы оптимизатор мог устранить «тяжелые» или ненужные операции. Переработка представлений, конечно, требует тех же навыков, что и переработка запросов, – об этом мы будем говорить в пятой главе.

Если у вас есть сомнения насчет представлений, запрос к словарю данных может помочь определить те из них, которые нужно переписать. В MySQL мы можем поискать в текстах представлений имена других представлений из той же схемы и некоторые ключевые слова, которые указывают на их сложность. Следующий запрос подсчитывает для каждого представления, сколько представлений оно содержит, а также количество подзапросов – `union`, `distinct` или `group by`:

```
select y.table_schema,
       y.table_name,
       y.view_len,
       y.referenced_views views,
       cast((y.view_len - y.wo_from) / 4 - 1 as unsigned) subqueries,
       cast((y.view_len - y.wo_union) / 5 as unsigned) unions,
       cast((y.view_len - y.wo_distinct) / 8 as unsigned) distincts,
       cast((y.view_len - y.wo_group) / 5 as unsigned) groups
from (select x.table_schema,
            x.table_name,
```

¹ Не забывайте, что функция преобразования обязательно должна быть детерминированной и что в метке времени Unix отсутствует информация о часовом поясе.

```

        x.view_len,
        cast(x.referenced_views as unsigned) referenced_views,
        length(replace(upper(x.view_definition), 'FROM', '')) wo_from,
        length(replace(upper(x.view_definition), 'UNION', '')) wo_union,
        length(replace(upper(x.view_definition), 'DISTINCT', '')) wo_
distinct,
        length(replace(upper(x.view_definition), 'GROUP', '')) wo_group
from (select v1.table_schema,
        v1.table_name,
        v1.view_definition,
        length(v1.view_definition) view_len,
        sum(case
            when v2.table_name is not null
            then (length(v1.view_definition)
                - length(replace(v1.view_definition,
                                v2.table_name, '')))
                /length(v2.table_name)
            else 0
        end) referenced_views
from information_schema.views v1
left outer join information_schema.views v2
on v1.table_schema = v2.table_schema
where v1.table_name <> v2.table_name
group by v1.table_schema,
        v1.table_name,
        v1.view_definition) x
group by x.table_schema,
        x.table_name) y
order by 1, 2;

```

Тот факт, что в Oracle текст представлений хранится в столбце типа long, не позволяет нам напрямую искать ключевые слова внутри определения представления. Тут может помочь хранимая процедура PL/SQL или преобразование в более подходящий тип clob или очень большой тип varchar2. Однако мы можем получить очень интересный индикатор сложности представления благодаря представлению user_dependencies, которое можно вызывать рекурсивно, когда одни представления зависят от других:

```

col "REFERENCES" format A35
col name format A40
select d.padded_name name,
        v.text_length,
        d."REFERENCES"
from (select name,
        lpad(name, level + length(name)) padded_name,
        referenced_name || ' (' || lower(referenced_type) || ')' "REFERENCES"
from user_dependencies
where referenced_type <> 'VIEW'
connect by prior referenced_type = type
        and prior referenced_name = name

```

```

start with type = 'VIEW') d
left outer join user_views v
on v.view_name = name;

```

В SQL Server есть две возможности: запрос `information_schema.views` или перечисление иерархии зависимостей с помощью `sys.sql_dependencies`. Однако удаление дублирующих строк и приведение списка зависимостей к удобочитаемому виду требует изрядных усилий. Следующий запрос, вызывающий список зависимостей, предназначен только для опытных людей:

```

with recursive_query(level,
                    schemaid,
                    name,
                    refschemaid,
                    refname,
                    reftype,
                    object_id,
                    ancestry) as
(select 1 as level,
    x.schema_id as schemaid,
    x.name,
    o.schema_id as refschemaid,
    o.name as refname,
    o.type_desc as reftype,
    o.object_id,
    cast(x.rn_parent + '.' as
        + cast(dense_rank() over (partition by x.rn_parent
                                order by o.object_id) as varchar(5))
    as varchar(50)) as ancestry
from (select distinct cast(v.rn_parent as varchar(5)) as rn_parent,
    v.name,
    v.schema_id,
    v.object_id,
    d.referenced_major_id,
    cast(dense_rank() over (partition by v.rn_parent
                            order by d.object_id)
    as varchar(5)) as rn_child
from (select row_number() over (partition by schema_id
                                order by name) as rn_parent,
    schema_id,
    name,
    object_id
from sys.views) v
inner join sys.sql_dependencies d
on v.object_id = d.object_id) x
inner join sys.objects o
on o.object_id = x.referenced_major_id
union all
select parent.level + 1 as level,
    parent.refschemaid as schemaid,
    parent.refname as name,

```

```

        o.schema_id as refschemaid,
        o.name as refname,
        o.type_desc as reftype,
        o.object_id,
        cast(parent.ancestry + ','
              + cast(dense_rank() over (partition by parent.object_id
                                         order by parent.refname) as varchar(5))
              as varchar(50)) as ancestry
from sys.objects o
  inner join sys.sql_dependencies d
        on d.referenced_major_id = o.object_id
  inner join recursive_query parent
        on d.object_id = parent.object_id)
select a.name,
       len(v.view_definition) view_length,
       a.refname,
       a.reftype
from (select distinct space((level - 1) * 2) + name as name,
                    name as real_name,
                    schemaid,
                    refname,
                    lower(reftype) as reftype,
                    ancestry
      from recursive_query) as a
  inner join sys.schemas s
        on s.schema_id = a.schemaid
 left outer join information_schema.views v
        on v.table_schema = s.name
        and v.table_name = a.real_name
order by a.ancestry;

```

Запросы, исследующие зависимости между объектами, сообщают также о том, какие представления зависят от пользовательских функций.

Различные возможности, которые мы исследовали в этой и предыдущей главах, — это, вероятно, все, что можно сделать без реорганизации базы данных или затрагивания кода приложения. Если на этом этапе производительность нас по-прежнему не устраивает, пора начинать переработку кода. Но сначала давайте рассмотрим несколько инструментов, которые помогут нам получить достоверную информацию.

4

Концепция тестирования

*Произношение я ей уже поставил;
но с этой девушкой приходится думать
не только о том, как она произносит,
но и о том, что она произносит.*

Джордж Бернард Шоу (1856–1950)
Пигмалион, акт III, пер. Е. Калашниковой

Прежде чем я перейду к теме хирургического вмешательства в операторы и программы SQL, стоит затронуть тему, имеющую большое практическое значение. Как определить концепцию тестирования, как проверить, что изменения, приводящие к ускорению работы, не изменяют функциональности? Можно было бы доказать математически, что реляционная конструкция эквивалентна другой конструкции. На самом деле многие черты языка SQL не являются реляционными в строгом смысле (можно начать с оператора `order by`), и на практике причины дизайна баз данных приводят к тому, что некоторые запросы оказываются жалким подобием тех, какими они были бы в идеальном мире. В результате вам придется прибегать к обычной тактике разработки программ: сравнению полученного результата с тем, чего вы ожидали в максимально возможном числе случаев. Правда, такое сравнение в случае таблиц значительно сложнее, чем в случае скалярных величин или даже массивов.

Поскольку рефакторинг по своей сути является итеративным процессом (последовательным внесением небольших изменений до тех пор, пока производительность не станет приемлемой), нам нужны инструменты для наиболее быстрого выполнения итераций. В этой главе мы сначала поговорим о генерировании тестовых данных, а затем о проверке правильности изменений.

Генерирование тестовых данных

Среди важных этапов рефакторинга генерирование тестовых данных является, вероятно, самым недооцениваемым. Я встречался со случаями, когда генерирование качественных данных занимало значительно больше времени, чем исправление и тестирование операций SQL. Проблема с SQL заключается в том, что как бы ни был плох код, запрос к таблице из десяти строк всегда будет выполняться мгновенно, а в большинстве случаев вы увидите результат на экране сразу после того, как нажмете клавишу Enter, даже в случае таблицы с 10 000 строк. Падение производительности обычно начинается только после того, как вы накопите достаточное количество данных. Хотя очень часто можно сравнить относительную производительность альтернативных вариантов с таблицами среднего размера, довольно трудно предсказать, сумеете ли вы добиться приемлемой производительности без повторного тестирования целевых объемов.

Есть несколько случаев, когда вам потребуется генерирование тестовых данных:

- Если вы работаете дистанционно и объем рабочих данных слишком велик, чтобы его можно было переслать по сети или даже прислать на дисках DVD.
- Когда рабочие данные содержат важную информацию: например, профессиональные секреты или личную медицинскую или финансовую информацию, к которой у вас нет доступа.
- Когда целью рефакторинга является предупреждение возможных будущих проблем при росте объемов данных значительно выше текущих величин.
- И даже когда по различным причинам вам просто нужно небольшое согласованное подмножество рабочих данных, которые вы не можете получить от заказчиков¹. В таком случае создание собственных данных часто является самым простым и быстрым способом подготовить все, что вам нужно, перед началом работы.

Даже когда инструменты существуют, их приобретение может не окупиться, если потребность в них возникает у вас очень редко. И даже если ваши руководители полностью убеждены в пользе этих инструментов, сложная схема закупок в больших корпорациях зачастую не позволяет уложиться в плотное расписание работ по факторингу. Очень часто приходится создавать собственные тестовые данные.

¹ Коммерческие продукты умеют извлекать наборы согласованных данных из рабочих баз данных, но, к сожалению, если вы являетесь консультантом (как я), не приходится ожидать, что все ваши заказчики купят такой продукт. Ограничение целостности на уровне ссылок может быть серьезным ограничением возможностей.

Генерирование любого нужного количества строк для набора таблиц – это операция, которую нельзя выполнять бездумно. В этом разделе я покажу вам несколько способов, которыми вы можете сгенерировать около 50 000 строк в классической тестовой таблице Oracle, emp, которая включает в себя следующие столбцы:

empno

Первичный ключ, представляющий собой целое значение

ename

Фамилия сотрудника

job

Описание должности

mgr

Идентификатор менеджера текущего сотрудника

hiredate

Дата приема на работу, всегда в прошлом

sal

Зарплата, число с плавающей точкой

comm

Другое число с плавающей точкой, составляющее комиссию для агентов по продажам, для большинства сотрудников значение null

deptno

Целочисленный идентификатор подразделения, в котором работает сотрудник

Размножение строк

Исходная таблица Oracle emp содержит 14 строк. Простейшим способом размножения строк является использование декартова произведения: создайте представление или таблицу ten_rows, при запросе к которой будут просто возвращены значения от 0 до 9 как столбец num.

Поэтому, если вы запустите следующий фрагмент кода без какого-либо условия соединения, вы добавите в таблицу emp 56 000 новых строк (14 строк \times 10 \times 10 \times 10 \times 4):

```
insert into emp
select e.*
from emp e,
     ten_rows t1,
     ten_rows t2,
     ten_rows t3,
     (select num
      from ten_rows
      where num < 4) t4
```

Очевидно, что первичный ключ будет явно против такого метода, поскольку каждый из идентификаторов сотрудников, которые по своей сути являются уникальными, должен быть размножен 4000 раз. Поэтому вам нужно выбрать одно из следующих решений:

- Отключить все ограничения уникальности (на практике – удалить все уникальные индексы, реализующие эти ограничения), размножить данные, а затем снова включить ограничения.
- Попыаться в процессе вставки сгенерировать случайные значения. Вспоминая, что `ten_rows` возвращает числа от 0 до 9 и что все исходные имена сотрудников находятся в пределах восьмой тысячи, что-то подобное поможет нам выполнить задачу:

```
insert into emp(empno, ename, job, mgr, hiredate, sal, comm, deptno)
select 8000 + t1.num * 1000 + t2.num * 100 + t3.num * 10 + t4.num,
       e.ename, e.job, e.mgr, e.hiredate, e.sal, e.comm, e.deptno
from emp e,
     ten_rows t1,
     ten_rows t2,
     ten_rows t3,
     (select num
      from ten_rows
      where num < 4) t4
```

- В качестве альтернативы вы можете использовать последовательность Oracle или вычисления с помощью переменной в MySQL или SQL Server (если столбец не был определен как столбец идентификатора с автоматическим приращением).

Ваше ограничение первичного ключа будет удовлетворено, но данные вы получите неважные. Во второй главе я настаивал на необходимости наличия свежей статистики. Здесь, если вы будете умножать столбцы, то получите только 14 различных имен для более чем 50 000 сотрудников и 14 различных дат приема на работу, что не имеет никакого отношения к реальности (представьте себе 4000 человек, борющихся за единственную должность начальника). Во второй главе вы видели отношение между избирательностью индекса и производительностью. Если вы, например, создадите индекс по столбцу `ename`, он вряд ли будет полезен – хотя с реальными данными он может быть очень эффективным. Для получения реальных измерений производительности вам нужны правдоподобные тестовые данные.

Использование функций генерирования случайных значений

Использование функций генерирования случайных значений для обеспечения разнообразия данных – значительно лучший способ, чем многократное копирование малой выборки. К сожалению, для генерирования осмысленных данных требуется нечто большее, чем просто случайные последовательности.

Во всех СУБД есть хотя бы одна функция, генерирующая случайные числа, обычно вещественные числа между 0 и 1. Чтобы получить значения в любом диапазоне, нужно просто умножить случайные числа на разницу между максимальным и минимальным значениями в диапазоне и добавить к результату минимальное значение. Расширив этот метод, вы можете получать числовые значения любых типов и даже любой диапазон дат – если примените их арифметику.

Единственная проблема заключается в том, что функция генерирования случайных значений возвращает равномерно распределенные числа, а это означает, что вероятность получения значения в любой части диапазона одинаковая. В реальной жизни распределения редко бывают равномерными. Например, зарплаты колеблются вокруг среднего значения – известная куполообразная гауссова кривая (так называемого «нормального» распределения). Однако даты приема на работу могут быть распределены иначе: в растущей компании многие сотрудники пришли недавно, и если просматривать их список в обратном хронологическом порядке, количество сотрудников, принятых на работу раньше какой-то даты, резко уменьшается. Статистики называют это экспоненциальным распределением.

Такое значение, как величина зарплаты, вообще говоря, вряд ли будет проиндексировано, как зачастую и даты. Когда оптимизатор оценивает пользу от использования индекса, он принимает во внимание распределение данных. Неправильный тип распределения может заставить оптимизатор выбрать способ исполнения запроса, соответствующий тестовым данным, но вовсе не обязательно реальным рабочим данным. В результате вы можете в тестах встретиться с проблемами, которых не будет на рабочих данных, и наоборот.

SQL Server и функции генерирования случайных значений

SQL Server нужно уговаривать, чтобы он сгенерировал случайные значения, по причинам, непосредственно связанным с кэшированием и оптимизацией функций. Если вы запустите следующий код, вы получите значение между нулем и единицей:

```
select rand();
```

Однако если вы запустите следующий фрагмент кода, то получите одно и то же случайное значение, повторенное десять раз, а это не совсем то, что вам нужно:

```
select rand()  
from ten_rows;
```

На самом деле SQL Server «кэширует» результат функции `rand()` и использует одно и то же значение для каждой строки. Такое поведение может удивить вас, ведь функция `rand()` исходно является недетерминированной и должна вычисляться при каждом ее вызове (хотя серьезные статистики считают генерируемые на компьютере «случайные» числа детерминированными)¹.

Вы можете обойти это неудобство, поместив функцию T-SQL внутрь вашей собственной пользовательской функции. Но здесь вы можете наткнуться на новую сложность: если непосредственно сослаться на функцию `rand()` внутри своей функции, SQL Server внезапно вспомнит, что функция детерминированной не является, – и «пожалуется» (complains). Вторым уровнем обхода является скрытие функции `rand()` внутри представления, после чего можно делать запрос к представлению изнутри функции.

Давайте сделаем это. Сначала создадим представление:

```
create view random(value)
as select rand();
```

Затем создадим функцию, осуществляющую запрос к представлению:

```
create function randuniform()
returns real
begin
    declare @v    real;

    set @v = (select value from random);
    return @v;
end;
```

А затем запустим запрос, вызывающий функцию:

```
select dbo.randuniform() rnd
from ten_rows;
```

И все волшебным образом работает!

К счастью, генерирование неравномерно распределенных случайных данных из генератора равномерно распределенных данных, имеющегося

¹ Один из родоначальников компьютерной науки и автор одного из методов генерирования случайных чисел Джон фон Нейман однажды в шутку назвал людей, пытающихся создать случайности из среднего арифметического, грешниками.

в вашей любимой СУБД, – хорошо изученная математическая операция. Например, следующая функция MySQL сгенерирует данные, распределенные в виде куполообразной кривой вокруг среднего значения μ со стандартным отклонением σ ¹ (чем больше *сигма*, тем более пологим будет купол):

```
delimiter //
create function randgauss(mu double,
                        sigma double)
returns double
not deterministic
begin
    declare v1 double;
    declare v2 double;
    declare radsq double;
    declare v3 double;
    if (coalesce(@randgauss$flag, 0) = 0) then
        repeat
            set v1 := 2.0 * rand() - 1.0;
            set v2 := 2.0 * rand() - 1.0;
            set radsq := v1 * v1 + v2 * v2;
        until radsq < 1 and radsq > 0
        end repeat;
        set v3 := sqrt(-2.0 * log(radsq)/radsq);
        set @randgauss$saved := v1 * v3;
        set @randgauss$flag := 1;
        return sigma * v2 * v3 + mu;
    else
        set @randgauss$flag := 0;
        return sigma * @randgauss$saved + mu;
    end if;
end;
//
delimiter ;
```

Следующая функция SQL Server сгенерирует экспоненциально распределенные значения; параметр *lambda* управляет тем, насколько быстро число значений уменьшается. Если значение *lambda* велико, вы получите меньше больших значений и намного больше значений, близких к нулю. Например, вычитание из текущей даты количества дней, равного произведению 1600 на `randexp(2)` даст реалистичное на вид распределение дат приема на работу:

```
create function randexp(@lambda real)
returns real
begin
    declare @v real;
```

¹ Мю (μ) и сигма (Σ) – буквы греческого алфавита, традиционно используемые для представления среднего значения и стандартного отклонения (меры распределения вокруг среднего значения).

```

if (@lambda = 0) return null;
set @v = 0;
while @v = 0
begin
    set @v = (select value from random);
end;
return -1 * log(@v) / @lambda;
end;

```

Вспомните, что порядок строк относительно индексных ключей тоже может иметь некоторое значение. Если ваши данные не хранятся в самоорганизующейся системе типа кластерного индекса, вам может понадобиться отсортировать их в соответствии с ключом, который соответствует порядку вставки строк, чтобы корректно эмулировать реальные данные.

Однако при генерировании случайных символьных строк мы столкнемся с проблемами.

Oracle поставляется с пакетом PL/SQL, `dbms_random`, который содержит несколько функций, включая одну, генерирующую случайные строки из букв, с цифрами или без, с возможностью добавления экзотических печатных символов, в нижнем, верхнем или смешанном регистре. Например:

```
SQL> select dbms_random.string('U', 10) from dual;
```

```
СУБД_RANDOM.STRING('U', 10)
```

```
-----
PGTWRFYMKB
```

Или, если вы хотите получить случайную строку случайной длины от 8 до 12 символов:

```
SQL> select dbms_random.string('U',
2                               8 + round(dbms_random.value(0, 4)))
3 from dual;
```

```
СУБД_RANDOM.STRING('U', 8+ROUND(СУБД_RANDOM.VALUE(0, 4)))
```

```
-----
RLTRWPLQHEB
```

Написание подобной функции для MySQL или SQL Server сложности не представляет. Если вам просто нужна строка длиной не больше 250 символов в верхнем регистре, вы можете написать что-то подобное:

```

create function randstring(@len int)
returns varchar(250)
begin
    declare @output varchar(250);
    declare @i int;
    declare @j int;

    if @len > 250 set @len = 250;
    set @i = 0;
    set @output = '';

```

```
while @i < @len
begin
    set @j = (select round(26 * value, 0) from random);
    set @output = @output + char(ascii('A') + @j);
    set @i = @i + 1;
end;
return @output;
end;
```

Вы легко можете адаптировать этот пример для SQL Server с использованием нижнего или смешанного регистра.

Но если вы хотите генерировать строки, которые (подобно названию должности) должны выбираться из ограниченного количества значений в нужных пропорциях, вам придется опираться на нечто иное, чем случайная генерация строк: распределение тестовых данных должно основываться на данных существующих.

Подгонка под существующие распределения

Даже когда данные являются совершенно конфиденциальными (например, медицинские записи), статистическая информация о них засекречивается очень редко. Люди, отвечающие за данные, скорее всего, с радостью пойдут вам навстречу и предоставят результаты запроса `group by`, особенно если вы напишете запрос за них (и попросите их запустить запрос в период низкой загрузки базы данных в случае больших таблиц).

Предположим, мы хотим сгенерировать реалистическое распределение должностей. Если мы запустим запрос `group by` к нашей тестовой таблице `emp`, мы получим следующий результат:

```
SQL> select job, count(*)
      2 from emp
      3 group by job;
```

JOB	COUNT(*)
CLERK	4
SALESMAN	4
PRESIDENT	1
MANAGER	3
ANALYST	2

Если бы у вас было намного больше строк, на которых можно основывать распределение тестовых данных, вы могли бы прямо использовать результат этого запроса. При наличии всего 14 строк предыдущий результат можно использовать только как исходную точку. Не будем рассматривать желанную для многих должность президента — одну строку мы сможем позже вставить вручную. Я хочу получить распределение, показанное на рис. 4.1.

Создание такого распределения будет несложным, если мы свяжем каждое название должности с диапазоном значений между 0 и 100: все, что нам нужно будет сделать, это сгенерировать случайные числа, равномерно распределенные в этом интервале. Если число попадет в интервал от 0 до 35, будет возвращено значение CLERK; если оно попадет в интервал от 35 до 70, будет возвращено значение SALESMAN; от 70 до 80 – MANAGER и от 80 до 100 – ANALYST. С практической точки зрения все, что нам нужно сделать, это вычислить накопленные частоты.

Теперь я детально покажу вам, как создать более сложный список, чем названия должностей: фамилии. Конечно, я мог бы сгенерировать случайные строки, но DXGBBRTYU – фамилия явно не самая удачная. Как же создать правдоподобный список фамилий? Ответ, как часто бывает, можно найти в Интернете. Например, если вы хотите генерировать американские фамилии, хорошей исходной точкой будет раздел *Genealogy* на сайте <http://www.census.gov>¹. На этом сайте есть список самых распространенных американских фамилий, ранжированный по распространенности и с указанием числа владельцев.



Рис. 4.1. Целевое распределение должностей для тестовых данных

Если вы хотите сгенерировать сравнительно малое количество разных фамилий, вы можете загрузить файл *Top1000.xls*, содержащий тысячу самых распространенных американских фамилий согласно последней

¹ В Wikipedia есть статья о наиболее распространенных фамилиях (в том числе русских) в других странах: http://en.wikipedia.org/wiki/Common_surnames.

переписи (доступен и файл с более полным списком). Загрузите его и сохраните как файл `.csv`. Вот первые строки из файла:

```
name;rank;count;prop100k;cum_prop100k;pctwhite;pctblack;pctapi;\.
.. (продолжение первой строки) ..\pctaian;pct2prace;pcthispanic
SMITH;1;2376206;1727,02;1727,02;73,35;22,22;0,40;0,85;1,63;1,56
JOHNSON;2;1857160;1349,78;3076,79;61,55;33,80;0,42;0,91;1,82;1,50
WILLIAMS;3;1534042;1114,94;4191,73;48,52;46,72;0,37;0,78;2,01;1,60
BROWN;4;1380145;1003,08;5194,81;60,71;34,54;0,41;0,83;1,86;1,64
JONES;5;1362755;990,44;6185,26;57,69;37,73;0,35;0,94;1,85;1,44
MILLER;6;1127803;819,68;7004,94;85,81;10,41;0,42;0,63;1,31;1,43
DAVIS;7;1072335;779,37;7784,31;64,73;30,77;0,40;0,79;1,73;1,58
GARCIA;8;858289;623,80;8408,11;6,17;0,49;1,43;0,58;0,51;90,81
RODRIGUEZ;9;804240;584,52;8992,63;5,52;0,54;0,58;0,24;0,41;92,70
WILSON;10;783051;569,12;9561,75;69,72;25,32;0,46;1,03;1,74;1,73
...
```

Хотя нам нужны имя и накопленная пропорция (присутствующая в пятом поле, `cum_prop100k`, или накопленная пропорция на 100 000), я буду использовать первые три поля только для иллюстрации того, как я могу обрабатывать данные для получения нужного результата.

Сначала я создам таблицу `name_ref` следующим образом:

```
create table name_ref(name varchar(30),
rank int,
headcount bigint,
cumfreq bigint,
constraint name_ref_pk primary key(name));
```

Ключевым столбцом для выбора имени является `cumfreq`, представляющий собой накопленную частоту. Для данных в этом столбце я решил использовать целые значения. Если бы эти значения были процентными, целые числа не предоставляли бы необходимой точности и мне пришлось бы использовать десятичные значения. Но я буду, как американское правительство, использовать значения «1 на 100 000» («per-100,000») вместо значений «1 на 100», и это даст мне достаточную точность, чтобы я мог работать с целыми числами.

На этом этапе я могу загрузить данные в свою таблицу с помощью стандартной команды СУБД: `BULK INSERT` в SQL Server, `LOAD` в MySQL или утилиты `SQL*Loader` (или внешних таблиц) в Oracle (более точные команды для каждой СУБД вы найдете в загружаемых примерах кода, описанных в приложении А).

После того как исходные данные загружены, я сначала должен вычислить накопленные значения, нормализовать их, а затем преобразовать накопленные значения «на 100 000». Поскольку мое подмножество ограничено в этом примере тысячей самых распространенных фамилий в США, я получу значительно более высокие накопленные частоты, чем реальные значения. Эту системную ошибку легко минимизировать с помощью большей выборки (списка фамилий, которые носят свыше ста человек,

имеющегося на том же сайте). Я очень легко могу вычислить нормализованные накопленные частоты с помощью двух операторов в MySQL:

```
set @freq = 0;
update name_ref
set cumfreq = greatest(0, @freq := @freq + headcount)
order by rank;
update name_ref
set cumfreq = round(cumfreq * 100000 / @freq);
```

Два других оператора выполняют ту же операцию в SQL Server:

```
declare @maxfreq real;
set @maxfreq = (select sum(headcount) from name_ref);
update name_ref
set cumfreq = (select round(sum(n2.headcount)
                        * 100000 / @maxfreq, 0)
              from name_ref n2
              where name_ref.rank >= n2.rank);
```

А в Oracle для той же задачи хватит одного оператора:

```
update name_ref r1
set r1.cumfreq = (select round(100000 * r2.cumul / r2.total)
                 from (select name,
                             sum(headcount) over (order by rank
                                                    range unbounded preceding) cumul,
                             sum(headcount) over () total
                             from name_ref) r2
                 where r2.name = r1.name);
```

Преыдущие фрагменты кода представляют собой хорошие иллюстрации синтаксических вариаций, созданных на основе самых основных запросов SQL. Некоторые из этих операций (особенно в случае SQL Server) не очень эффективны, но поскольку эту операцию предполагается выполнить только один раз на тестовой базе данных — на это можно не обращать внимания.

Теперь все, что остается сделать, это создать индекс по столбцу cumfreq, после чего мы готовы генерировать статистически правдоподобные данные (это, конечно, не совсем статистически корректные данные, поскольку я проигнорировал все фамилии, кроме тысячи самых распространенных). Для каждой строки нужно сгенерировать случайное число между 0 и 100 000 и вернуть фамилию, связанную с самым маленьким значением cumfreq, не меньшим, чем это случайное число.

Чтобы вернуть случайную фамилию в SQL Server, я могу запустить следующий код (с помощью представления, вызывающего функцию rand()):

```
select top 1 n.name
from name_ref n,
     random r
where n.cumfreq >= round(r.value * 100000, 0)
order by n.cumfreq;
```


Вы можете применить эти приемы для генерирования данных почти любого типа. Все, что вам нужно, это найти списки элементов, для которых указаны весовые коэффициенты (иногда вы можете найти ранги, но можно создать весовые коэффициенты, присвоив больший вес первому рангу, 0,8 от этого веса – второму рангу, 0,8 от этого веса – третьему рангу и т. д.). Очевидно, что любая фраза `group by` к существующим данным позволит вам заполнить тестовые данные в соответствии с существующими соотношениями.

Генерирование большого числа строк

Когда мы хотим сгенерировать тестовые данные, нам обычно нужно большое количество значений. Вставка множества случайных значений иногда может оказаться труднее, чем кажется. В MySQL мы получим ожидаемый результат, вызвав запрос, возвращающий одну случайную фамилию из результата запроса, возвращающего несколько строк:

```
mysql> select (select name
->             from name_ref
->             where cumfreq >= round(100000 * rand())
->             limit 1) name
-> from ten_rows;
```

```
+-----+
```

```
| name |
```

```
+-----+
```

```
| JONES |
```

```
| MARTINEZ |
```

```
| RODRIGUEZ |
```

```
| WILSON |
```

```
| THOMPSON |
```

```
| BROWN |
```

```
| ANDERSON |
```

```
| SMITH |
```

```
| JACKSON |
```

```
| JONES |
```

```
+-----+
```

```
10 rows in set (0.00 sec)
```

```
mysql>
```

К сожалению, SQL Server возвращает одну и ту же фамилию десять раз, даже с нашей специальной оболочкой для функции `rand()`:

```
select (select top 1 name
from name_ref
where cumfreq >= round(dbo.randuniform() * 100000, 0)
order by cumfreq) name
from ten_rows;
```

Следующий запрос в Oracle ведет себя подобно запросу SQL Server¹:

¹ По крайней мере, в Oracle 11g.

```
select (select name
      from (select name
            from name_ref
            where cumfreq >= round(dbms_random.value(0, 100000)))
      where rownum = 1) name
from ten_rows;
```

В обоих случаях SQL-машина пытается оптимизировать обработку запроса путем кэширования результата подзапроса, блокируя таким образом создание нового случайного числа для каждой строки. Вы можете исправить это поведение в SQL Server, принудительно привязав подзапрос к каждой новой возвращаемой строке:

```
select (select top 1 name
      from name_ref
      where cumfreq >= round(x.rnd * 100000, 0)
      order by cumfreq) name
from (select dbo.randuniform() rnd
      from ten_rows) x;
```

Однако для Oracle этого недостаточно. Во-первых, вы, вероятно, обратили внимание, что в Oracle у нас два уровня подзапросов в списке `select`, и нам нужно ссылаться на случайное значение в самом «глубоком» подзапросе. Синтаксис Oracle допускает, чтобы подзапрос ссылался на текущие значения, возвращаемые запросом прямо над ним, – но не дальше, и если мы сошлемся в самом «глубоком» запросе на случайное значение, связанное с каждой строкой, возвращенной запросом `ten_rows`, мы получим ошибку. Есть ли абсолютная необходимость в двух уровнях подзапросов? Особенность Oracle заключается в том, что псевдостолбец `rownum` вычисляется по мере того, как строки извлекаются из базы данных (или из результирующего набора, полученного вложенным запросом) во время идентификации строк, которые образуют результирующий набор, возвращаемый текущим запросом. Если вы выполните фильтрацию по `rownum` после окончания сортировки, то, конечно, извлечете n первых или последних строк, в зависимости от того, в каком направлении сортировка – по возрастанию или по убыванию. Если вы выполните фильтрацию по `rownum` в том же запросе, где выполняется сортировка, то получите n строк (первые n строк из базы данных) и отсортируете их, что обычно приводит к совершенно другому результату. Конечно, может случиться, что строки будут извлечены в правильном порядке (поскольку Oracle либо будет использовать индекс, ссылающийся на строки в порядке значений ключа, либо будет сканировать таблицу, в которой строки были вставлены по уменьшению частоты). Но тогда это будет просто случайностью.

Если мы хотим принудить Oracle заново исполнять для каждой строки запрос, возвращающий случайную фамилию, а не возвращать то, что лежит у него в кэше, мы должны также привязать подзапрос из списка `select` к главному запросу, который возвращает определенное количество строк, но другим способом, например, как в следующем запросе:

```
SQL> select (select name
2      from (select name
3      from name_ref
4      where cumfreq >= round(dbms_random.value(0, 100000)))
5      where rownum = 1
6      and -1 < ten_rows.num) ename
7 from ten_rows
8 /
```

ENAME

PENA
 RUSSELL
 MIRANDA
 SWEENEY
 COWAN
 WAGNER
 HARRIS
 RODRIGUEZ
 JENSEN
 NOBLE

10 rows selected.

Условие, что число из `ten_rows` больше `-1`, выполняется всегда. Но Oracle этого не знает, и ему приходится каждый раз снова выполнять запрос и, соответственно, каждый раз возвращать новую фамилию.

Но это не последняя из наших проблем с Oracle. Сначала, если мы хотим сгенерировать большое количество строк и несколько раз соединить с таблицей `ten_rows`, нам придется добавить в список `select` фиктивную ссылку на *каждое* вхождение таблицы `ten_rows` – иначе мы получим многочисленные последовательности повторяющихся имен. Это приведет к очень громоздкому коду SQL и усложнит изменения количества генерируемых строк. Но есть и худшее следствие: когда вы попытаетесь сгенерировать все столбцы с помощью единственного запроса Oracle, то получите данные с сомнительной случайностью.

Например, я попытался сгенерировать просто 100 строк с помощью следующего запроса:

```
SQL> select empno,
2      ename,
3      job,
4      hiredate,
5      sal,
6      case job
7      when 'SALESMAN' then round(rand_pack.randgauss(500, 100))
8      else null
9      end comm,
10     deptno
11 from (select 1000 + rownum * 10 empno,
12      (select name
```

```

13      from (select name
14              from name_ref
15              where cumfreq >= round(100000
16                                     * dbms_random.value(0, 1)))
17      where rownum = 1
18             and -1 < t2.num
19             and -1 < t1.num) ename,
20      (select job
21          from (select job
22                  from job_ref
23                  where cumfreq >= round(100000
24                                           * dbms_random.value(0, 1)))
25          where rownum = 1
26                 and -1 < t2.num
27                 and -1 < t1.num) job,
28      round(dbms_random.value(100, 200)) * 10 mgr,
29      sysdate - rand_pack.randexp(0.6) * 100 hiredate,
30      round(rand_pack.randgauss(2500, 500)) sal,
31      (select deptno
32          from (select deptno
33                  from dept_ref
34                  where cumfreq >= round(100000
35                                           * dbms_random.value(0, 1)))
36          where rownum = 1
37                 and -1 < t2.num
38                 and -1 < t1.num) deptno
39      from ten_rows t1,
40           ten_rows t2)
41 /

```

EMPNO	ENAME	JOB	HIREDATE	SAL	COMM	DEPTNO
1010	NELSON	CLERK	04-SEP-07	1767		30
1020	CAMPBELL	CLERK	15-NOV-07	2491		30
1030	BULLOCK	SALESMAN	11-NOV-07	2059	655	30
1040	CARDENAS	CLERK	13-JAN-08	3239		30
1050	PEARSON	CLERK	03-OCT-06	1866		30
1060	GLENN	CLERK	09-SEP-07	2323		30
...						
1930	GRIMES	SALESMAN	05-OCT-07	2209	421	30
1940	WOODS	SALESMAN	17-NOV-07	1620	347	30
1950	MARTINEZ	SALESMAN	03-DEC-07	3108	457	30
1960	CHANEY	SALESMAN	15-AUG-07	1314	620	30
1970	LARSON	CLERK	04-SEP-07	2881		30
1980	RAMIREZ	CLERK	07-OCT-07	1504		30
1990	JACOBS	CLERK	17-FEB-08	3046		30
2000	BURNS	CLERK	19-NOV-07	2140		30

100 rows selected.

Хотя фамилии выглядят достаточно случайными, как и даты поступления, зарплаты и комиссионные, мне ни разу не удалось получить,

несмотря на многократные попытки, ничего иного, кроме как SALESMAN или CLERK для должности, а все сотрудники поголовно оказывались в 30 отделе.

Однако когда я написал процедуру PL/SQL, последовательно генерирующую каждый столбец и вставляющую строку за строкой, таких проблем не возникло, и я получил правильно распределенные, вполне случайные на вид данные. У меня нет ясного объяснения такого поведения. То, что при использовании процедурного подхода я получил правильные данные, доказывает, что генерирование случайных чисел тут ни при чем. Скорее проблема в кэшировании функции и различных механизмах оптимизации, которые в этом (по-видимому, частном) случае препятствовали моим намерениям.

Из предыдущих глав вы могли предположить (эту тему мы еще будем обсуждать в дальнейшем), что я рассматриваю процедурную обработку как некую аномалию в реляционном контексте. Но верно и обратное: когда мы генерируем последовательность случайных чисел, то находимся в процедурной среде, чуждой работе с базами данных. Мы должны выбрать для работы правильный инструмент, а генерирование случайных данных представляет собой процесс с вызовами процедур.

Оборотной стороной связывания процедурных языков с базами данных является их относительная медлительность. Создание миллионов строк не будет мгновенной операцией. Поэтому разумно сохранять таблицы, которые вы заполнили случайными данными, в файлах, если вы хотите сравнить с одними и теми же данными несколько операций, изменяющих базу данных. Это ускорит перезагрузку. В качестве альтернативы вы можете использовать «обычный» язык типа C++ или Java или даже мощный язык сценариев вроде Perl и сгенерировать плоский файл, который можно будет загружать в базу данных с большой скоростью. Генерирование файлов независимо от базы данных позволит вам воспользоваться преимуществами таких библиотек, как Gnu Statistical Library (GSL), для изолированного генерирования случайных значений. Главная сложность заключается в том, чтобы генерируемые данные соответствовали предопределенным распределениям, а для этого при больших объемах данных (как в случае с фамилиями) база данных и язык SQL удобнее. Но с помощью небольшой программы вы сможете извлечь данные этого типа из вашей базы данных или из файла SQLite¹ или Access.

Целостность на уровне ссылок

Генерирование случайных данных с сохранением их целостности может оказаться непростой задачей. Больших сложностей с обеспечением целостности данных между несколькими таблицами нет. Вам просто нужно сначала заполнить справочные таблицы (во многих случаях даже

¹ SQLite является одним из моих любимых инструментов, когда нужно организовать персональные данные, будь то справочные данные, как в этом случае, или собранные данные о производительности.

не потребуется генерировать данные, можно будет вместо этого использовать реальные таблицы), а затем таблицы, которые на них ссылаются. Необходимо лишь быть внимательными с генерированием правильных пропорций каждого внешнего ключа с помощью вышеприведенных примеров. Реальные сложности возникают с зависимостями внутри таблицы. В такой таблице, как `emp`, есть несколько зависимостей:

`comm`

Существование комиссии зависит от того, занимает ли сотрудник должность `SALESMAN`. С этой зависимостью проблем нет.

`mgr`

Идентификатор менеджера, `mgr` (или `null` для президента), должен быть существующим идентификатором сотрудника. Относительно легкое решение, если вы генерируете данные с помощью процедуры, — указать, что идентификатор менеджера может быть только одним из (например) первых 10% сотрудников. Поскольку идентификаторы сотрудников генерируются последовательно, мы знаем, в каком диапазоне они будут считаться идентификаторами менеджеров. Тогда остается единственная трудность, заключающаяся в самих менеджерах, для которых мы можем установить, что их менеджером может быть только сотрудник, чья запись была сгенерирована раньше, чем его собственная. Однако такой процесс обязательно приведет к множеству несогласованностей (например, у кого-нибудь, имеющего должность `MANAGER`, будет менеджер с должностью `CLERK` в другом отделе).

Вообще, чем хуже спроектирована база данных, тем больше вы встретите избыточной информации и скрытых зависимостей — и тем больше шансов, что при заполнении таблиц вы будете плодить несогласованности. Для примера возьмите таблицу `emp`. Если идентификатор менеджера является атрибутом данной должности в данном отделе, то есть атрибутом функции (нет смысла обновлять записи сотрудников, остающихся на своих должностях, если их менеджер переместился на другую должность), а не атрибутом конкретной персоны, большинство сложностей исчезает. К сожалению, в процессе рефакторинга приходится работать с тем, что есть, а полная переработка проекта базы данных (как бы ни была она частястую необходима) — случай редкий. В результате вам приходится определять несогласованности данных, которые, вероятно, происходят из генерирования случайных значений, и рассматривать, не приведут ли они к искажению того, что вы собираетесь измерять. Надо быть готовыми в некоторых случаях запустить несколько обновлений SQL для того, чтобы сделать генерируемые данные более согласованными.

Генерирование случайного текста

Прежде чем закрыть тему генерирования данных, нужно сказать несколько слов об одном типе данных — случайный текст. Очень часто в тестовых столбцах баз данных хранятся не имена или короткие метки,

а данные XML, длинные комментарии или статьи, как в системах управления содержимым сайтов (CMS) или в Википедии, построенной на основе базы данных. Иногда бывает полезным сгенерировать реалистичные тестовые данные и для этих столбцов: например, вам может потребоваться выяснить, является ли like '%pattern%' приемлемым решением для поиска, требуется ли полнотекстовое индексирование или лучше реализовать поиск по ключевым словам.

Создание случайных данных XML вряд ли требует больше труда, чем генерирование чисел, дат или символьных строк: заполнение таблиц, которые более или менее соответствуют определению типа документа XML (DTD) с последующими запросами к таблицам для добавления к значениям соответствующих тегов, сложности не представляет. Значительно сложнее создать реальный неструктурированный текст. Вариант генерирования случайных символьных строк, где пробелы и знаки пунктуации являются такими же символами, как и остальные, всегда в вашем распоряжении. Однако такой текст совсем не будет похож на реальный, что затруднит тестирование поиска по тексту: у вас не будет реальных слов, по которым можно будет осуществлять поиск. Вот почему псевдослучайный реальный текст гораздо предпочтительнее.

В примерах проектирования вы могли встретить абзацы, заполненные латинским текстом, который очень часто начинается со слов *lorem ipsum*¹. Словами *lorem ipsum* (сокращенно *lipsum*) обычно называют заполняющий текст – абсолютно бессмысленный, но правдоподобно выглядящий (он хоть и похож на латинский текст, но не имеет никакого смысла), используемый полиграфистами с XVI века. Хороший способ создать такой текст – это начать с реального текста, разметить его (включая знаки препинания, которые рассматриваются при этом как однобуквенные слова), сохранить их все в таблице SQL и вычислить частоты. Однако генерирование текста несколько сложнее, чем случайный выбор слов согласно частотам, как мы делали с фамилиями. Генерация случайного текста обычно основана на *цепях Маркова*, то есть случайных событиях, которые зависят от предыдущих случайных событий. На практике выбор слова основан на его длине и на длине двух предыдущих сгенерированных слов. Для выбора нового слова генерируется случайное число, а на основе длины его и двух предыдущих слов выбирается длина нового слова. После этого среди слов с выбранной длиной выбирается слово в соответствии с частотой.

Я написал две программы, которые вы можете найти на сайте этой книги (<http://www.oreilly.com/catalog/9780596514976>). Как их использовать, написано в приложении В. Эти программы используют SQLite. Первая, *mklipsum*, анализирует текст и генерирует файл данных SQLite. Этот

¹ Подобные примеры можно найти на сайте <http://www.oswd.org>, где предлагаются варианты дизайна сайтов с открытым кодом.

файл используется во второй программе, *lipsum*, которая и генерирует случайный текст.

Вы можете использовать в качестве основы для файла данных любой текст по вашему выбору. Лучше всего, очевидно, выбрать текст, похожий на тот, который должен храниться в вашей базе данных. Вы можете придать вашему тексту шикарный классический вид, используя латынь. Можно найти немало латинских текстов в Интернете, но я бы порекомендовал *De finibus bonorum et malorum* (О разграничении добра и зла) Цицерона, из которого растут корни *lorem ipsum*. Неплохим выбором также является текст, который уже выглядит случайным (не будучи таковым на самом деле), например подростковый блог или маркетинговый пресс-релиз. Наконец, вы можете использовать книги (на разных языках) с сайта <http://www.gutenberg.org>. Учтите, что специфическая лексика и необычные имена героев произведений могут наложить определенный отпечаток на получившийся результат.

На этом я закрою тему генерирования тестовых данных. Но не забывайте про этот этап в оценке предполагаемого времени на выполнение работы.

Сравнение альтернативных версий

Составной частью рефакторинга может стать переписывание кода. Если проверки и усовершенствования, описанные во второй и третьей главах, окажутся недостаточными, вам потребуются методы сравнения альтернативных версий и выбора лучшей. Но прежде чем сравнивать производительность альтернативных версий, нужно убедиться, что переписанные нами варианты полностью эквивалентны исходной, медленной версии¹.

Блочное тестирование

Блочное тестирование – очень популярный метод, часто связываемый с рефакторингом. Это один из краеугольных камней экстремального программирования. Блоком называют самую маленькую часть программы, которую можно протестировать (например, процедуру или метод). В сущности, идея заключается в том, чтобы установить контекст так, чтобы тесты можно было повторять, а затем запустить набор тестов, сравнивающих фактический результат операции с ожидаемым и прекращающихся, если какой-нибудь из тестов потерпит неудачу.

¹ Мне встречались случаи, когда при переписывании обнаруживались незамеченные ранее ошибки, и поэтому новая версия была не полностью идентична старой. Но мы не будем рассматривать такие необычные случаи.

Использование такой концепции очень удобно в случае процедурных или объектных языков. С SQL все обстоит по-другому¹, за исключением случаев, когда вы хотите проверить только поведение оператора `select`, возвращающего единственное значение. Например, когда вы выполняете оператор `update`, количество строк, на которые он может повлиять, варьируется от нуля до общего количества строк в таблице. Поэтому вы не можете сравнить «одно значение» с «одним ожидаемым значением», но можете сравнить «одно состояние таблицы» с «одним ожидаемым состоянием таблицы», где состояние определено не значением, ограниченным количеством атрибутов (как в случае экземпляра объекта), а скорее значениями набора строк, – возможно, очень большого.

Нам может потребоваться сравнить таблицу с другой таблицей, которая представляет ожидаемое состояние. Возможно, это окажется не самой удобной из операций, но такой процесс надо рассмотреть.

Приближенное сравнение

Если вы переписываете запрос, который возвращает очень мало строк, беглого взгляда на результат бывает достаточно для оценки его правильности. Недостаток этого метода состоит в том, что «беглый взгляд» несколько проблематично внедрить в автоматизированные наборы тестов и нельзя применить ни к запросам, возвращающим большое количество строк, ни к операторам, модифицирующим данные.

Вторым простейшим функциональным сравнением для двух альтернативных версий является количество строк либо возвращенных оператором `select`, либо измененных оператором `insert`, `delete` или `update`. Количество строк, обработанных оператором, – это стандартное значение, возвращаемое любым интерфейсом SQL с СУБД. Его можно выяснить либо из какой-то системной переменной, либо как значение, возвращенное функцией, вызывающей исполнение оператора.

Конечно, считать равенство количества обработанных строк доказательством функциональной эквивалентности не слишком надежно. Тем не менее для первого приближения этот метод может быть достаточно эффективен – если вы сравниваете *один* оператор с *одним* же оператором (имеется в виду, что вы не заменяете несколько операторов SQL одним). Однако основывать сравнение только на количестве обработанных строк рискованно. Если использовать группировку в качестве примера, суммы могут быть неправильными, когда вы забыли условие соединения и суммируете несколько дублирующих строк, – тем не менее запрос может возвращать то же количество строк, что и правильный. Также и обновление средней сложности, в котором столбцам присваиваются значения подзапросов, может сообщить о правильном

¹ Стоит отметить, что большинство концепций блочного тестирования, имеющих для баз данных, на самом деле разработаны для хранимых процедур – другими словами, процедурного расширения языка.

количестве обновленных строк – и все же быть неверным, если в подзапрос вкралась ошибка.

Если переписанный оператор, примененный к тем же данным, обрабатывает другое количество строк, чем исходный, – вы можете быть уверены, что один из этих операторов неверен (необязательно новый), но равенство количества обработанных строк является очень слабым подтверждением правильности.

В других случаях сравнение количества обработанных строк вообще неприменимо: если вы сумели заменить несколько последовательных обновлений таблицы одним, может не быть простой взаимосвязи между количеством строк, обработанных каждым из исходных обновлений, и количеством строк, обработанных вашей версией.

Сравнение таблиц и результатов

Можно сравнивать количество операций, но что действительно имеет значение – это данные. Нам может потребоваться сравнить таблицы (например, как на одни и те же исходные данные повлияли исходный процесс и переработанная версия) или результирующие наборы (например, сравнить вывод двух запросов). Поскольку результирующие наборы могут быть видны как виртуальные таблицы (аналогично представлениям), я буду рассматривать, как можно сравнивать таблицы, подразумевая, что когда во фразе `from` я ссылаюсь на имя – это может быть и таблица, и подзапрос.

Что сравнивать

Прежде чем описывать приемы сравнения, я хотел бы подчеркнуть, что во многих случаях не все столбцы таблицы подходят для сопоставления, особенно когда мы сравниваем различные способы изменения данных. Очень часто в таблицу записывается такая информация, как временная метка или идентификатор пользователя, который последним изменял данные в таблице. Если вы последовательно запускали исходный процесс и переработанный, временные метки будут различаться, поэтому их надо будет игнорировать. Однако вы можете встретить и более сложные случаи, когда приходится иметь дело с генерируемыми базой данных числовыми ключами (последовательности в Oracle, идентификационные столбцы в SQL Server или столбцы с автоматическим приращением в MySQL).

Представьте, что вы хотите проверить, дает ли переписанный процесс изменения данных тот же результат, что и исходный медленный процесс. Предположим, процесс применен к таблице `T`. Вы можете начать с создания таблицы `T_COPY`, точной копии (включая индексы и все остальное) таблицы `T`. Затем вы запускаете исходный процесс и переименовываете обновленную таблицу `T` в `T_RESULT`. Вы переименовываете таблицу `T_COPY` в `T`, запускаете переработанный процесс и пытаетесь сравнить новое состояние таблицы `T` с таблицей `T_RESULT`. Если вы

используете базу данных Oracle и в процессе участвуют какие-то последовательные номера, то при втором запуске вы получите последовательные номера, следующие за теми, которые были сгенерированы при первом запуске. Конечно, вы можете перед запуском переработанного процесса сбросить все использовавшиеся последовательности и создать их заново, начиная с наибольшего числа, которое было до запуска исходного процесса. Однако это серьезно усложнит дело, и (что еще важнее) вы должны помнить о несоответствии физического порядка строк в таблице. Это также означает, что обработка строк в различном порядке вполне допустима, поскольку конечный результат идентичен. Если вы обрабатываете строки в различном порядке, ваш более быстрый процесс может во всех случаях присваивать той же строке другой последовательный номер. Эти же соображения касаются идентификационных столбцов (с автоматическим приращением). Строго говоря, строки будут различными, но, как и было в случае с временными метками, логически они будут идентичными.

Если вы не хотите тратить время на поиск различий, не имеющих отношения к бизнес-логике, вы должны помнить, что столбцы с автоматическим присвоением значений (если это не постоянное значение по умолчанию) значения не имеют. Тем не менее, существует огромное количество разработчиков (и многочисленные инструменты проектирования), искренне верящих, что проект базы данных хорош, если в каждой таблице есть уникальный столбец с генерируемыми базой данных числами, именуемый *id*. Нужно понимать, что такой целенаправленно построенный первичный ключ есть не что иное, как простая замена осмысленного ключа (часто состоящего из нескольких столбцов), уникально идентифицирующего строку. Поэтому учтите, что:

- Когда я использую в следующих разделах символ *, я ссылаюсь не на все столбцы в таблице, а на все относящиеся к предметной области столбцы.
- Когда я ссылаюсь на столбцы с первичным ключом, я не обязательно ссылаюсь на столбец (или столбцы), определенный как «первичный ключ» в базе данных, а на столбцы, которые, с точки зрения предметной области, уникально идентифицируют каждую строку в таблице.

Примитивные способы сравнения

Простой, но эффективный способ узнать, содержат ли две таблицы идентичные данные – вывести их в файлы операционной системы с последующим использованием системных утилит для выявления различий.

Теперь надо определить, как выводить в файлы:

Утилиты СУБД

Во всех СУБД есть инструменты для вывода данных из базы данных, но эти инструменты редко удовлетворяют нашим требованиям.

Например, в MySQL утилита `mysqldump` выводит полные таблицы. Как вы уже видели, во многих случаях могут быть различные значения для технических столбцов, которые будут функционально правильными. В результате сравнение вывода утилиты `mysqldump` для одних и тех же данных после двух различных, но функционально эквивалентных процессов будет сообщать о различиях, не имеющих значения для дела.

Утилита `bcpr` в SQL Server не имеет таких ограничений, как `mysqldump`, поскольку с параметром `queryout` она может выводить результат запроса. По этой причине утилита `bcpr` подходит для наших целей.

Oracle до версии 10.2 относился к той же категории, что и MySQL, поскольку его утилита экспорта `exp` умела выводить только все столбцы в таблице. В отличие от `mysqldump`, единственным форматом вывода был собственный двоичный формат, который не мог помочь в поиске различий. Более поздние версии неявно помогали нам выводить результат запроса с помощью утилиты `expdp` или путем создания внешней таблицы типа `oracle_datapump` как результата запроса. Однако с этими технологиями есть проблемы: формат по-прежнему двоичный, а файл создается на сервере, куда у разработчика может не быть доступа.

Подводя итог, если исключить утилиту `bcpr`, и в некоторых случаях `mysqldump`, утилиты СУБД не очень подходят для наших целей (впрочем, они и создавались не для этого).

Инструменты командной строки СУБД

Запуск произвольного запроса с помощью инструмента командной строки (`sqlplus`, `mysql` или `sqlcmd`) является, вероятно, простейшим способом вывести результирующий набор в файл. Создайте файл, содержащий запрос, который возвращает те данные, что вы хотите проверить, запустите его с помощью утилиты командной строки, и вы получите снимок данных.

Загвоздка со снимками данных в том, что они могут быть очень велики. Обойти проблему большого объема можно, пропустив данные через утилиту подсчета контрольной суммы – такую, как `md5sum`. Например, для вычисления контрольной суммы в запросе MySQL вы можете использовать сценарий, подобный нижеприведенному:

```
#
# Маленький bash-сценарий для вычисления контрольной суммы
# результата запроса к базе данных mysql.
#
# Это упрощенная версия, информация о соединении
# должна быть указана как <user>/<password>:<database>
#
# Дополнительно можно указать метку (-l "label")
# и временную метку (-t), что может быть полезно,
# результаты добавляются в тот же файл.
#
usage="Usage: $0 [-t][-l label] connect_string sql_script"
```

```
#
ts=''
while getopts "tl:h" options
do
    case $options in
        t ) ts=$(date +%Y%m%d%H%M%S);;
        l ) label=$OPTARG;;
        h ) echo $usage
            exit 0;;
        \? ) echo $usage
            exit 0;;
        * ) echo $usage
            exit 1;;
    esac
done
shift $((OPTIND - 1))
user=$(echo $1 | cut -f1 -d/)
password=$(echo $1 | cut -f2 -d/ | cut -f1 -d:)
database=$(echo $1 | cut -f2 -d:)
mysum=$(mysql -B --disable-auto-rehash -u${user} -p${password} \
-D${database} < $2 | md5sum | cut -f1 -d' ',)
echo „$ts $label $mysum“
```

Имейте в виду, что контрольные суммы MD5 чувствительны к порядку строк. Если у таблицы нет кластерного индекса, переработанный код может привести к тому, что тот же набор данных будет сохранен в таблице с другим порядком строк. Имеет смысл добавить к запросу, с помощью которого вы получаете снимок, фразу `order by`, даже если это приведет к некоторым накладным расходам.

У вычисления контрольной суммы есть очевидное слабое место: если вы выясните, что результаты не идентичны, то не будете знать, где находятся различия. Это означает, что вы окажетесь в одной из двух ситуаций:

- Вы очень сильны в «формальном SQL» и сумеете разобраться, как два логически близких оператора могут возвращать различные результаты, не запуская сами эти операторы (в 9 случаях из 10 причиной окажутся затаившиеся значения `null`).
- Ваших знаний SQL не хватает (или просто сегодня не ваш день), и вам нужно реально «видеть» различие результатов, чтобы понять, что происходит. Тогда вывод данных в файл может помочь, если контрольные суммы различаются (конечно, вам придется еще раз запустить медленный запрос).

В общем, в разных ситуациях нужны разные подходы.

Сравнение SQL, версия из учебника

Использование утилиты командной строки либо для вывода в файл, либо для подсчета контрольной суммы содержимого таблицы или результата запроса предполагает, что тестируемым блоком является целая программа. В некоторых случаях вам захочется проверить изнутри

переработанного кода, являются ли данные по-прежнему идентичными тому, что можно получить из исходной версии. Желательно иметь возможность проверять данные из программы с помощью либо хранимой процедуры, либо оператора SQL.

Классический способ в SQL сравнивать содержимое двух таблиц заключается в использовании оператора `except` (`minus` в Oracle). Различие между таблицами A и B состоит в строках таблицы A, которые отсутствуют в таблице B, плюс строки в таблице B, отсутствующие в таблице A:

```
(select * from A
  except
  select * from B)
union
(select * from B
  except
  select * from A)
```

Обратите внимание, что в понятие «отсутствующая» входит понятие «отличающаяся». Логически каждая строка состоит из первичного ключа, однозначно идентифицирующего строку, и некоторого количества атрибутов.

Строка действительно отсутствует, если отсутствует первичный ключ. Однако поскольку в предыдущем фрагменте кода SQL мы подразумевали под словом *все* только столбцы, имеющие отношение к предметной области, то строки с одинаковым первичным ключом в обеих таблицах, но с разными атрибутами, будут одновременно появляться как отсутствующие (с одним набором атрибутов) в одной таблице и (с другим набором атрибутов) в другой таблице.

Если ваш диалект SQL не знает об операторах `except` или `minus` (как в случае MySQL¹), лучшей реализацией этого алгоритма, вероятно, будут два внутренних соединения, что требует знания имен столбцов первичного ключа (ниже они названы `pk1`, `pk2`, ... `pkn`):

```
select A.*
from A
      left outer join B
        on A.pk1 = B.pk1
      ...
      and A.pkn = B.pkn
where B.pk1 is null
union
select B.*
from B
      left outer join A
        on A.pk1 = B.pk1
      ...
      and A.pkn = B.pkn
where A.pk1 is null
```

¹ Хотя в MaxDB того же производителя они реализованы.

Теперь, если вы посмотрите на оба примера кода критическим взглядом, вы заметите, что запросы могут оказаться относительно дорогими. Сравнение всех строк требует сканирования обеих таблиц – мы не можем избежать этого. К сожалению, запросы делают больше, чем просто сканируют таблицы. Например, когда вы используете `except`, происходит неявная сортировка данных (это логично: сравнение сортированных наборов сложности не представляет, так что перед сравнением имеет смысл отсортировать обе таблицы). Поэтому код SQL, в котором используется оператор `except`, фактически сканирует и сортирует каждую таблицу *дважды*. Поскольку таблицы содержат (как мы надеемся) одни и те же данные, то при больших объемах данных это неэффективно. Кто-то мог бы заметить, что, строго говоря, вместо фразы `union` в запросе (которая неявно приводит к удалению дубликатов и выполнению сортировки) можно использовать `union all`, поскольку запрос таков, что две части фразы `union` не могут вернуть одну и ту же строку. Однако поскольку мы не ожидаем большого числа различий (и надеемся на их полное отсутствие), это не сильно ускорит запрос, и отсортированный результат может оказаться удобнее.

Версия кода, явным образом ссылающаяся на столбцы первичного ключа, работает не лучше. Как вы убедились во второй главе, использование индекса (даже индекса по первичному ключу) для извлечения строк по одной при работе с большим количеством строк значительно менее эффективно, чем сканирование. Поэтому, даже если мы написали порозному, реальная обработка в обоих случаях будет сходной.

Сравнение SQL, версия получше

На популярном среди разработчиков для Oracle сайте (<http://asktom.oracle.com/>, созданном Томом Кайтом), Марко Стефанетти предложил решение, которое ограничивает количество сканирований таблиц одним сканированием каждой таблицы, но с применением оператора `group by` к объединению. Следующий запрос является результатом совместной работы Марка и Тома:

```
select X.pk1, X.pk2, ..., X.pkn,
       count(X.src1) as cnt1,
       count(X.src2) as cnt2
from (select A.*,
            1 as src1,
            to_number(null) as src2
      from A
      union all
      select B.*,
            to_number(null) as src1,
            2 as src2
      from B) as X
group by X.pk1, X.pk2, ..., X.pkn
having count(X.src1) <> count(X.src2)
```

Хотя в предыдущем разделе использование оператора `union` вместо `union all` не вносило больших отличий – там он был применен к результатам (вероятно, очень маленьким) двух сравнений, здесь это имеет значение. В данном случае мы применяем `union` к двум сканированиям таблицы. У нас не может быть дубликатов, и если таблицы велики, не стоит выполнять ненужную сортировку.

Метод Стефанетти–Кайта масштабируется значительно лучше, чем классический метод. Чтобы протестировать его, я запускал оба запроса с двумя таблицами, различавшимися только одной строкой. Когда таблицы содержали только тысячу строк (1000 и 999, если быть точным), обе версии работали быстро. Однако по мере роста количества строк производительность классического метода сравнения быстро падала, как вы можете видеть на рис. 4.2.



Рис. 4.2. Как масштабируются два метода сравнения таблицы

Метод Стефанетти–Кайта ведет себя значительно лучше даже при большом количестве строк.

Сравнение контрольных сумм в SQL

Подходят ли эти методы сравнения для работ по рефакторингу? Боюсь, что они полезнее при однократном сравнении, чем при итеративном тестировании.

Предположим, что вы переписываете медленный запрос, возвращающий много строк. Если вы хотите применить предыдущие методы к результирующему набору, а не к обычной таблице, вы можете попробовать вставить проблемный запрос как подзапрос во фразу `from`. Если вы не гений, то вряд ли с первой попытки найдете правильный вариант запроса. А если вам каждый раз придется заново выполнять медленный подзапрос, то работа будет идти медленно.

Поэтому имеет смысл создать таблицу `result_ref`, в которой будет храниться целевой результирующий набор:

```
create table result_ref
as (slow query)
```

В качестве альтернативы (особенно если ожидается большой результирующий набор, а с дисковым пространством есть проблемы) вы можете выполнить подсчет контрольной суммы этого результирующего набора из SQL. В MySQL есть команда `checksum table`, но это команда для проверки целостности данных, которая применяется к таблицам, а не к запросам, и в любом случае в ней невозможно исключить какие-либо столбцы. Однако во всех СУБД есть функции для вычисления контрольной суммы данных: В MySQL это функция `md5()`, в Oracle – `dbms_crypto.hash()`, а в SQL Server – `hashbytes()`, две последние позволяют использовать либо алгоритм MD5, либо SHA11. В некоторых продуктах есть функции для генерирования хеш-значений (в SQL Server также есть `checksum()`, а в Oracle – `dbms_utility.get_hash_value()`), но будьте осторожны с использованием функций. Различные функции хеширования имеют различные свойства: функция типа `checksum()` в первую очередь предназначена для построения хеш-таблиц, но при этом допустимы коллизии – то есть для нескольких строк хеширование приведет к одному результату. Когда вы строите хеш-таблицы, наибольшее значение имеет скорость функции. Однако если мы хотим сравнить данные, вероятность того, что для различных данных хеширование приведет к одинаковому результату, должна быть нулевой – то есть нам нужно использовать более сильный алгоритм MD5 или SHA1¹.

Тут есть загвоздка: все эти функции (кроме функции SQL Server `checksum_aggr()`, про которую говорят, что это просто двоичное исключение или операция [XOR]) являются скалярными функциями: они получают строку, для которой вычисляется контрольная сумма, как аргумент и возвращают хеш-значение. Они не оперируют наборами строк, и вы не можете перенаправить в них результат запроса, как в случае `md5-sum`, например. Что было бы действительно полезно, так это функция, которая принимает в качестве аргумента текст произвольного запроса и возвращает контрольную сумму набора данных, возвращенного этим набором, – например, функция, которая позволила бы нам написать следующее:

```
set mychecksum = qrysum('select * from transactions');
```

Нужно, чтобы результат можно было сравнить с предыдущей контрольной суммой, а длинный процесс прервать, как только обнаружено различие. Для этой цели нам необходимо написать функцию, позволяющую делать следующее:

¹ SHA1 больше подходит для криптографии, но мы не пытаемся шифровать данные, мы просто хотим вычислить контрольную сумму, а MD5 выполняет эту задачу очень хорошо.

- Выполнять произвольный запрос.
- Вычислять контрольную сумму строки, возвращенной запросом.
- Объединять все контрольные суммы строк в общую контрольную сумму.

Первый пункт указывает на динамический SQL. В Oracle все требуемые средства имеются в пакете `dbms_sql`. SQL Server и MySQL снабжены несколько хуже. В SQL Server есть `sp_executesql`, а также такие хранимые процедуры, как `sp_describe_cursor_columns`, работающая с курсорами, которые на практике являются обработчиками, связанными с запросами. Но в MySQL, хоть мы и можем подготавливать операторы из случайной строки и выполнять их, но если не использовать для хранения результатов временные таблицы, мы не сможем выполнять цикл по результирующему набору, возвращенному подготовленным оператором. Можно выполнять цикл с курсором, но курсоры нельзя связать напрямую с произвольным оператором, переданным в виде строки.

Поэтому я покажу вам, как написать хранимую процедуру, которая вычисляет контрольные суммы результатов произвольного запроса. Поскольку я предполагаю, что вам интересен один (максимум – два) продукта из трех рассматриваемых, а мне надоели книги с длинными фрагментами кода, которые я не могу применить напрямую, я опишу принципы и конкретные проблемы для каждого продукта. Вы можете загрузить полный текст программ с комментариями с сайта этой книги и прочитать описание в приложении А.

Давайте начнем написание функции подсчета контрольной суммы с Oracle, поскольку здесь есть все, что требуется для динамического построения запроса. Последовательность операций следующая:

1. Создать курсор.
2. Связать текст запроса с курсором и сделать его синтаксический анализ.
3. Добавить в структуру описания столбцов или выражений, возвращаемых запросом.
4. Выделить структуры для хранения данных, возвращаемых запросом.

На этом этапе мы встречаемся с первой сложностью: PL/SQL – это не Java или C#, и динамическое выделение памяти осуществляется с помощью «коллекций». В результате код становится громоздким, по крайней мере, с моей точки зрения. Данные возвращаются как символы и конкатенируются в буфер. Вторая сложность заключается в том, что строки PL/SQL имеют ограничение размера 32 кбайт. Если мы превышаем это ограничение, нам приходится переходить к LOB-данным. Вывод всех данных в гигантский LOB с последующим вычислением контрольной суммы MD5 при росте результирующего набора приведет к появлению проблем с дисковым пространством. Данные нужно где-то хранить до передачи функции подсчета контрольной суммы. К сожалению, выбор использования контрольных сумм вместо копирования таблицы или

результатирующего набора часто вызван в первую очередь беспокойством о нехватке дискового пространства.

Более реалистичный вариант, чем использование LOB-данных, – вывод данных в буфер кусками по 32 кбайт и вычисление контрольной суммы для каждого куска. Но как тогда объединить контрольные суммы?

В контрольной сумме MD5 используется 128 бит (16 байт). Мы можем использовать другой буфер размером 32 кбайт для конкатенации всех вычисленных контрольных сумм, пока их не наберется 2000, – в этот момент буфер заполнится. Теперь мы можем «сжать» буфер контрольной суммы, вычислив контрольную сумму его содержимого, и продолжить. Когда все данные будут извлечены, содержимое буфера контрольной суммы надо будет пропустить через алгоритм MD5 последний раз, что даст нам значение, которое функция возвратит.

Вычисление контрольных сумм для других контрольных сумм произвольное количество раз вызывает вопросы насчет действительности конечного результата. В конце концов, сжатие файла, который уже был сжат, иногда приводит к увеличению размера файла. В качестве примера я запустил следующий сценарий, вычисляющий контрольную сумму текста Цицерона на латыни, а затем 10 000 раз вычисляющий контрольную сумму предыдущей итерации. На втором этапе сценарий применяет тот же процесс к тому же тексту, просто я удалил из него имя человека, которому посвящена книга, – Брута.

```
checksum=$(echo "Non eram nescius, Brute, cum, quae summis ingeniis
exquisitaque doctrina philosophi Graeco sermone tractavissent, ea
Latinis litteris mandaremus, fore ut hic noster labor in varias
reprehensiones incurreret. nam quibusdam, et iis quidem non admodum
indoctis, totum hoc displicet philosophari. quidam autem non tam id
reprehendunt, si remissius agatur, sed tantum studium tamque multam
operam ponendam in eo non arbitrantur. erunt etiam, et ii quidem eruditi
Graecis litteris, contemnentes Latinas, qui se dicant in Graecis legendis
operam malle consumere. postremo aliquos futuros suspicor, qui me ad alias
litteras vocent, genus hoc scribendi, etsi sit elegans, personae tamen et
dignitatis esse negent." | md5sum | cut -f1 -d' ')
echo "Initial checksum : $checksum"
i=0
while [ $i -lt 10000 ]
do
    checksum=$(echo $checksum | md5sum | cut -f1 -d' ')
    i=$(( $i + 1 ))
done
echo "After 10000 iterations: $checksum"
checksum=$(echo "Non eram nescius, cum, quae summis ingeniis
exquisitaque doctrina philosophi Graeco sermone tractavissent, ea
Latinis litteris mandaremus, fore ut hic noster labor in varias
reprehensiones incurreret. nam quibusdam, et iis quidem non admodum
indoctis, totum hoc displicet philosophari. quidam autem non tam id
reprehendunt, si remissius agatur, sed tantum studium tamque multam
operam ponendam in eo non arbitrantur. erunt etiam, et ii quidem eruditi
```

```

Graecis litteris, contemnentes Latinas, qui se dicant in Graecis legendis
operam malle consumere. postremo aliquos futuros suspicor, qui me ad alias
litteras vocent, genus hoc scribendi, etsi sit elegans, personae tamen et
dignitatis esse negent." | md5sum | cut -f1 -d' ')
echo "Second checksum : $checksum"
i=0
while [ $i -lt 10000 ]
do
checksum=$(echo $checksum | md5sum | cut -f1 -d' ')
i=$(( $i + 1 ))
done
echo "After 10000 iterations: $checksum"

```

Результат воодушевляет:

```

$ ./test_md5
Initial checksum : c89f630fd4727b595bf255a5ca762fed
After 10000 iterations: 1a6b606bb3cec62c0615c812e1f14c96
Second checksum : 0394c011c73b47be36455a04daff5af9
After 10000 iterations: f0d61feebae415233fd7cebc1a412084

```

Первые контрольные суммы различаются так же, как и контрольные суммы для контрольных сумм, а это именно то, что нам нужно.

Есть код этой функции для Oracle, описанный в приложении А. Я привел ее не только в качестве курьеза и примера кода PL/SQL, но и чтобы показать главную проблему этой функции: хотя при работе с запросом, возвращающим несколько тысяч строк, она выдавала результат быстро, вычисление ею контрольной суммы для следующего запроса к таблице из первой главы, содержащей два миллиона строк, заняло около часа:

```
select * from transactions
```

Моей целью является достаточно быстрое выполнение итеративных тестов с переработанными операторами. Поэтому я изменил эту функцию, учитывая то, что выполнение происходит быстрее всего на сервере, и не просто на сервере, а еще и внутри SQL-машины.

Одним из тормозящих факторов в моей функции является то, что я возвращаю данные внутри кода PL/SQL и затем вычисляю их контрольную сумму — поэтому я пытаюсь заполнить 32 кбайт буфера данными до начала вычисления контрольной суммы. Есть другая возможность, которая значительно упрощает код: возвращать контрольную сумму непосредственно для каждой строки.

Не забывайте, что после синтаксического разбора запроса я вызывал функцию, чтобы описать различные столбцы, возвращаемые запросом. Вместо того чтобы использовать это описание для подготовки буферов к приему данных и вычисления, сколько байт может вернуть каждая строка, я построил новый запрос, в котором преобразовал каждый столбец в символьную строку, конкатенировал их значения и применил функ-

цию подсчета контрольной суммы к полученной строке внутри запроса. Для фразы `from` мне потребовалось всего лишь написать следующее:

```
... || ' from (' || original_query || ') as x'
```

На этом этапе вместо того, чтобы обрабатывать внутри функции каждый столбец для каждой строки, возвращаемой запросом, я обхожусь только одной 16-байтной контрольной суммой на строку.

Но я могу продолжить эту логику еще дальше: что если вместо агрегирования контрольных сумм в моей программе поместить процесс агрегирования в запрос SQL? Агрегирование данных SQL делает хорошо. Все, что мне нужно сделать, это создать свою собственную функцию агрегирования для объединения контрольных сумм произвольного количества строк в общую контрольную сумму. Я мог бы использовать только что представленный мной механизм, конкатенирование контрольных сумм и сжатие получившейся строки каждые 2000 строк. Я создал функцию агрегирования, выполняющую операцию XOR со всеми контрольными суммами. Это несложная функция. Вычисления в ней не назовешь идеальными, но поскольку моей задачей является быстрая идентификация различий в результирующих наборах, а не зашифровывание номера моей кредитной карты, для моих потребностей ее достаточно. Важнее то, что она достаточно быстра и нечувствительна к порядку строк в результирующем наборе (но чувствительна к порядку столбцов в списке `select` в запросе).

Когда я применил эту функцию к тому же запросу:

```
select * from transactions
```

я получил контрольную сумму примерно через 100 секунд. Коэффициент ускорения по сравнению с версией, которая получает все данные и вычисляет контрольную сумму внутри программы, оказался равен 36. Это еще один хороший пример того, как можно эффектно усовершенствовать исходно неплохую программу, сведя к минимуму взаимодействие программы с SQL-машиной, даже когда эта программа является хранимой процедурой. Этот пример также показывает, что тестирование с малыми объемами данных может ввести в заблуждение, поскольку обе версии, когда применялись к результирующему набору из 5 000 строк, возвращали результат мгновенно – с небольшим преимуществом функции, которая на деле является значительно более медленной.

Разобравшись с Oracle, я могу использовать те же принципы и для SQL Server, и для MySQL, хотя в этих СУБД нет той гибкости обработки динамических запросов внутри хранимой процедуры, которую дает пакет `dbms_sql` в Oracle. Различие заключается в том, что в обоих продуктах больше правил, чем в Oracle, в отношении того, что функция может выполнять, а динамический SQL не очень продуман. Поэтому вместо функции я буду писать процедуры с параметром *out*, чтобы достичь тех же целей, что в случае функции.

В SQL Server я сначала должен связать текст запроса с курсором. К сожалению, при объявлении курсора текст оператора, связанного с курсором, нельзя передать как переменную. Для обхода этой проблемы оператор `declare cursor` сам строится динамически, что позволяет нам получить описания различных столбцов запроса, передаваемого в переменной `@query`.

```
--
-- Сначала связываем запрос, который передается курсору
--
set @cursor = N'declare c0 cursor for ' + @query;
execute sp_executesql @cursor;
--
-- Исполняем sp_describe_cursor_columns в переменную курсора.
--
execute sp_describe_cursor_columns
    @cursor_return = @desc output,
    @cursor_source = N'global',
    @cursor_identity = N'c0';
```

Следующая трудность возникает со встроенной функцией `checksum_agg()`. Эта функция оперирует четырехбайтными целыми числами, в результате 16-байтный результат функции `hashbytes()` (при выборе алгоритма MD5) значительно превышает ее разрядность. Поскольку XOR действует побитно, я ничего не изменю в результате, если разобью контрольную сумму MD5, вычисленную для каждой строки, на четыре четырехбайтных куска, к каждому куску применю функцию `checksum_agg()` и, наконец, конкатенирую четыре агрегата. Ничего сложного.

В MySQL, как я уже упоминал, возможности динамического SQL ограничены по сравнению с Oracle или SQL Server. Тем не менее написание процедуры, вычисляющей контрольную сумму произвольного запроса не так сложно, как может показаться. Мы можем обойти отсутствие поддержки динамического описания столбцов, возвращаемых запросом, создав таблицу, в которой находится структура результирующего набора (я предполагаю, что все вычисляемые столбцы имеют псевдонимы):

```
create table some_unlikely_name as
select * from (original query)
limit 0
```

Обратите внимание, что это `create table`, а не `create temporary table`. Поскольку для MySQL таблица является постоянной, я могу извлечь описание столбцов, возвращаемых запросом, из `information_schema`.

Зная имена столбцов, мы можем с легкостью построить запрос, который конкатенирует их и применяет к результату функцию `md5()`. Как и в случае SQL Server, 16-байтная контрольная сумма MD5 слишком велика для встроенной функции агрегирования `bit_xor()`: она принимает как аргумент и возвращает 64-битное целое число. Как и для SQL

Server, контрольные суммы MD5 разбиваются (по две за раз), и к каждой части отдельно применяется функция `bit_xor()`, а затем производится склеивание.

Ограничения сравнения

Как видите, есть несколько способов убедиться, что переработанный процесс дает тот же результат, что и исходная версия. Выбор варианта зависит от того, переписываете вы отдельные запросы или целые процессы, от объема обрабатываемых данных, от скорости, с которой вы хотите осуществлять итерации, и от уровня детальности, необходимого для исправления потенциальных различий. Чаще всего вы будете использовать комбинацию методов, в одном месте проверяя количество строк, в другом контрольные суммы, а где-то еще сравнивая данные.

Однако не забывайте, что даже самое тщательное сравнение данных не может заменить логический анализ кода. Поскольку язык SQL основан на логических концепциях, исключениях, пересечениях, комбинациях условий `and` и `or` плюс странном значении `null` (которое зловредно отказывается быть либо равным чему-то еще, либо отличаться от чего-нибудь), то доказательство правильности лежит в самом запросе. Случается даже, что правильность не обязательно означает идентичность результатов.

Один заказчик как-то спросил меня, после того как я представил значительно более быстрый вариант первоначального запроса, могу ли я доказать, что он дает тот же результат. Оказалось, что мой запрос, как назло, вернул не совсем тот же результат, что и исходный. Этот запрос был первым этапом двухэтапного процесса, и его задачей было идентифицировать маленький набор возможных кандидатов при нечетком поиске. Поскольку второй этап, который действительно идентифицировал конечный результат, был сравнительно быстрым, я выбрал на первом этапе несколько неточную, но значительно более быструю фильтрацию. Ну что ж, мне пришлось пересмотреть идентичность результатов на первом этапе.

Теперь, когда у нас есть тестовые данные и различные способы протестировать эквивалентность обработки данных переработанными версиями, давайте сосредоточим наше внимание на главной теме книги: усовершенствовании существующих приложений SQL. Сначала рассмотрим переработку медленных запросов – этим мы займемся в следующей главе.

5

Рефакторинг операторов

*Нужно решительно двигаться к победе,
иначе наше дело будет проиграно.*

Улисс Симпсон Грант (1822–1885)
Воспоминания, глава XXXVII

В ситуации, когда еще остается широкий простор для дальнейших действий, даже частичные усовершенствования большого, плохо написанного запроса, резко снижающего производительность программы, оказывают очевидный психологический эффект. Если вы хотите, чтобы на вас смотрели с восхищением, лучше сразу внести изменения в такой запрос, чем обещать привести программу в порядок в ближайшие месяцы.

Хотя эффективная обработка данных даже в случае плохо написанных операторов и является основной функцией оптимизаторов, эта функция может не выполняться, даже если индексы в порядке и оптимизатор имеет доступ ко всей требуемой информации. Так часто случается и в жизни: если оптимизатор срабатывает в 99 случаях из 100, все заметят единственный неудачный случай. Если оптимизатору доступна вся необходимая статистика, сбои могут быть вызваны следующими причинами:

- в самой программе-оптимизаторе допущена ошибка (такое случается!);
- запрос настолько сложен, что за отведенное время оптимизатор сумел обработать только небольшую часть возможных вариантов и не смог подобрать более эффективный способ обработки.

Хочу заметить, что ошибки в оптимизаторах чаще всего возникают в ситуациях почти неразрешимой сложности и что правильное написание запроса является наиболее эффективным способом решения проблемы.

Более короткий и простой запрос требует от оптимизатора меньшего объема работы, следовательно, эффективный план исполнения будет найден быстрее. Умение хорошо писать код SQL можно сравнить с грамотной речью: если есть сомнения в ее правильности, вас могут не понять. Соответственно, если вы просто латаете и объединения, и условия `where`, и подзапросы, то оптимизатор может потерять нить того, что является наиболее критичным для достижения желаемого результата, для идентификации результирующего набора, который должен быть получен в итоге, и выбрать неверный путь.

Поскольку под выражением «медленный запрос» обычно понимается такой запрос, который извлекает строки дольше, чем можно было бы ожидать, мы можем сказать, что медленным является запрос с «медлительной фразой `where`». Фраза `where` используется в нескольких операторах SQL. В этой главе я буду использовать в качестве основного примера оператор `select`, но следует понимать, что рассматриваемые правила могут быть применены и к фразе `where` операторов `update` и `delete`, которые тоже извлекают строки.

Планы исполнения и директивы оптимизатора

Прежде всего нужно сделать небольшое замечание: многие люди связывают переработку запроса с анализом плана исполнения, но мне это не кажется правильным. Я могу прочесть план исполнения, но он ничего мне не скажет. План исполнения всего лишь сообщает, что произошло или произойдет. Само по себе это не плохо и не хорошо: оценить план исполнения можно только путем сравнения его с тем, что вы сами считаете образцовым планом исполнения. Например: «По-моему, здесь было бы лучше использовать этот индекс, почему же он делает полное сканирование таблицы?» Возможно, некоторые люди просто привыкли думать в терминах планов исполнения. Мне значительно легче думать в терминах запросов. План исполнения сложного запроса может занимать много страниц. Мне доводилось видеть и тексты запросов длиной в несколько страниц, но запрос обычно короче, чем план его исполнения, а потому значительно понятнее. Действительно, когда я читаю запрос, то более или менее ясно «вижу», что SQL-машина будет делать, и такое видение можно рассматривать как очень индивидуальный тип плана исполнения. Но я никогда не извлекал из анализа плана исполнения ничего действительно полезного, такого, чего я не мог бы, зная время исполнения, понять при внимательном чтении операторов и обращении к словарю данных.

С моей точки зрения, планы исполнения приносят не больше пользы, чем чрезмерно подробные инструкции типа «проехать 427 метров, затем повернуть налево на трассу 96 и проехать 2230 метров...». Мне намного удобнее пользоваться визуальными подсказками, которые я с легкостью смогу идентифицировать (например, указанием на заправки, рестораны или другие ориентиры), чем точными дистанциями, вычисленными

с помощью карт, или номерами дорог и названиями улиц, которые на ходу трудно разглядеть. Таков и мой подход к SQL. Ориентирами являются определенные знания: какие таблицы являются действительно большими, критерии действительно избирательными, а индексы – надежными. Представим, что оператор SQL рассказывает историю и имеет некую карту. Как в настоящем путешествии, вы отправляетесь из точки А (условия фильтрации во фразе *where*) в точку В (нужный вам результирующий набор). Если по дороге выясняется, что дорога от точки А до точки В оказалась слишком длинной, детальный анализ пройденного пути не обязательно поможет вам определить, почему путешествие оказалось таким долгим. Подсчет сигналов светофора и времени ожидания перед каждым светофором может быть полезен для выяснения, почему дорога заняла столько времени, но он ничего не скажет вам как водителю о том, где пролегает оптимальный маршрут. Методичное изучение дорожных условий очень полезно для выявления мест, где возникают пробки, и опасных участков дорог. При поиске наиболее короткого и удобного пути из точки А в точку В мне нужно всего лишь определить места, которых стоит избегать, изучить карту и заново продумать маршрут. Так же и в программировании: существуют определенные подходы к настройке, более подходящие для администраторов баз данных, чем для разработчиков, просто потому, что каждая группа предпочитает собственные методы работы. Вы можете разделять или не разделять мои взгляды, у каждого из нас свои особенности восприятия, и одни подходы нам ближе, чем другие. Так или иначе, очень часто использование разных подходов приводит к одному и тому же результату, хотя не всегда с одинаковой скоростью. В этой главе я описываю переработку операторов так, как я ее понимаю, и рассматриваю те подходы к ней, которые применяю на практике.

Если выбор способа достижения высокой скорости работы запроса можно считать делом вкуса, то по поводу директив оптимизатора, которые многие считают основным инструментом настройки, у меня есть сомнения. Директивы оптимизатора действительно работают. Я использовал их раньше и, вероятно, буду использовать в будущем. Тем не менее использование директив кажется мне спорной практикой, которую лучше избегать.

Теперь с помощью нескольких рисунков я попробую объяснить, почему так не люблю директивы оптимизатора. Язык SQL является декларативным языком. Другими словами, вы должны указать, что вы хотите получить, а не то, как база данных должна это делать. То же относится к запросам, для которых, по вашему мнению, существует «естественный способ» исполнения, который определяется способом написания запроса. Например, у вас есть следующая фраза *from*:

```
from table1
  inner join table2
    on ...
  inner join table3
    on ...
```

Вы можете предположить, что естественный способ выглядит так: сначала производится поиск необходимых данных в таблице `table1`, затем в таблице `table2`, затем в `table3` (с использованием условий, указанных во фразе `where`, для того чтобы связать таблицы в некоторое подобие цепи). Конечно, это слишком наивный подход к исполнению запроса. Задачами оптимизатора запросов являются проверка различных условий и оценка эффективности фильтрации данных и конечного результата. Оптимизатор пытается решить, следует ли начать с таблицы `table2`, затем перейти к таблице `table3`, затем к `table1` или другая комбинация окажется более эффективной. Точно так же он пытается определить, будет использование индекса полезно или вредно. После рассмотрения ряда возможностей оптимизатор вырабатывает план исполнения, соответствующий «естественному способу» выполнения другого функционально эквивалентного запроса. На рис. 5.1 показана моя модель работы оптимизатора и способ, при помощи которого он преобразует исходный запрос в какой-либо другой.

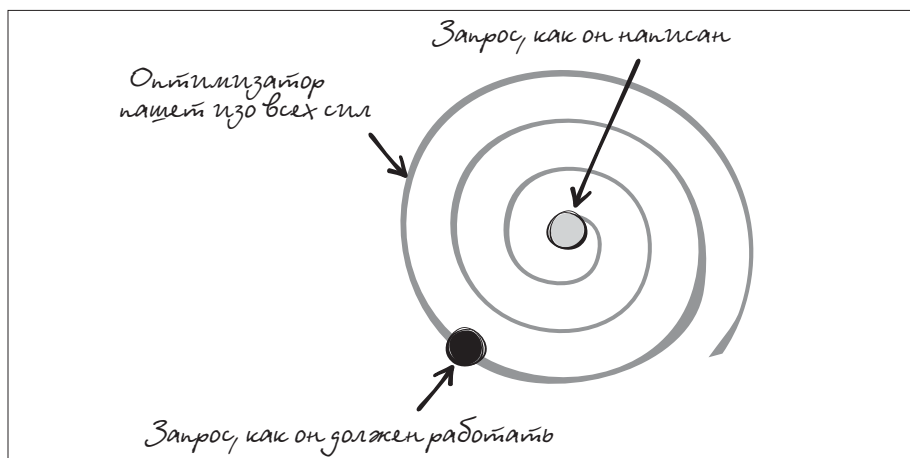


Рис. 5.1. Оптимизатор запросов за работой (аллегория)

Сложности работы оптимизатора имеют два аспекта. Во-первых, на выполнение работы ему отведено ограниченное время, а в случае сложного запроса он не может проверить все возможные комбинации. Например, в случае Oracle параметр `optimizer_max_permutations` определяет максимальное количество проверяемых комбинаций. В какой-то момент оптимизатор прекращает поиск вариантов и может принять решение выполнить глубоко вложенный подзапрос без оптимизации, поскольку времени на продолжение анализа уже не остается. Во-вторых, чем сложнее запрос, тем менее точны оценки стоимостей, которые сравнивает оптимизатор. Когда оптимизатор планирует различные шаги, которые он может предпринять, решающим фактором является то, насколько каждый шаг будет способствовать улучшению результата и насколько он

поможет сосредоточиться на конечном результирующем наборе. Например: «Если у меня есть определенное количество строк на входе, сколько строк я получу на выходе?» Даже когда вы используете в запросе для извлечения данных первичный или уникальный ключ, не всегда можно заранее определить результат: вы можете получить одну строку или не получить ни одной. Разумеется, если критерию может соответствовать много строк, ситуация сложнее. Каждая определяемая оптимизатором стоимость является оценочной. Когда запрос очень сложен, количество ошибок оценки возрастает; иногда они компенсируют друг друга, иногда накапливаются. Возрастает риск неправильной оценки, которую может сделать оптимизатор.

В результате, если сложный запрос написан очень плохо, оптимизатор оказывается не в состоянии найти более точный эквивалентный запрос. В ситуации, подобной той, что показана на рис. 5.2, оптимизатор выберет эквивалентный запрос, который, согласно его оценке, будет лучше исходного запроса, но, возможно, ненамного; оптимальный же план исполнения найден не будет.

В подобных случаях многие прибегают к директивам оптимизатора. Если вы имеете некоторое представление о том, каким должен быть план исполнения (возможно, перед обновлением программного обеспечения, или потому, что вы попали в другую среду или обнаружили план, попробовав ряд директив), то вы можете добавить специфичные для конкретной СУБД директивы, которые определенным образом направляют усилия оптимизатора, как показано на рис. 5.3.

Директивы постоянно используются, если вы пытаетесь добиться *стабильности плана*, предоставляемой некоторыми продуктами возможности записывать план исполнения, который считается удовлетворительным и будет использоваться при следующем исполнении того же запроса.

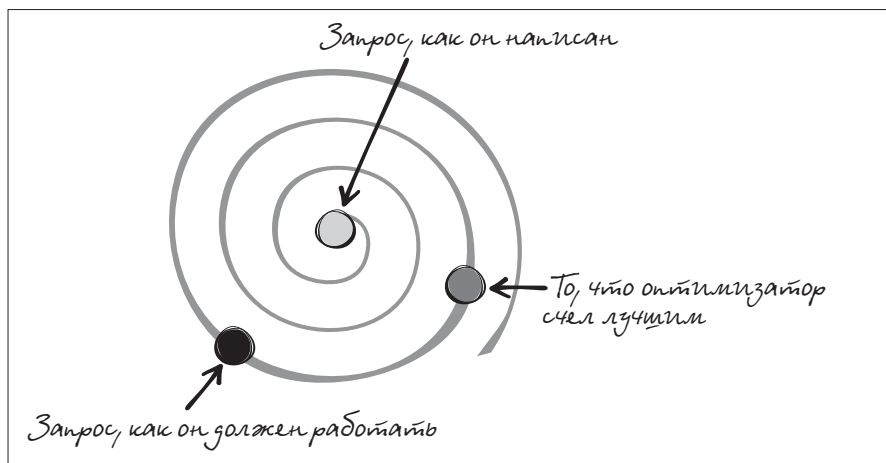


Рис. 5.2. Когда оптимизатор не находит лучший план исполнения

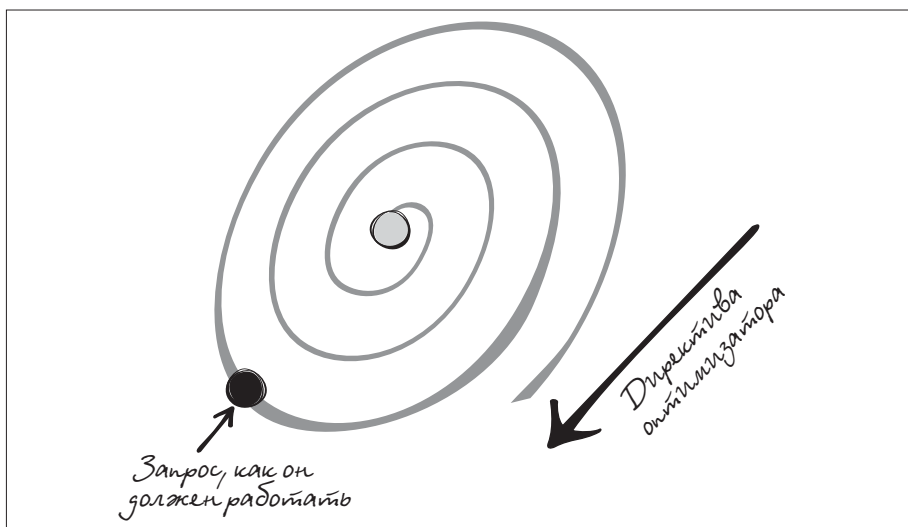


Рис. 5.3. Направление усилий оптимизатора с помощью директив

Мои сомнения, касающиеся директив и стабильности плана, основаны на том, что мы живем не в статичном мире. Как заметил Гераклит 2500 лет назад, нельзя войти в одну и ту же реку дважды: и вода уже не та, и человек не тот. Создавайте запас на перспективу, количество клиентов компании может значительно увеличиться. Прибыльный в прошлом году продукт может сдать свои позиции, а тот, который только начали производить, может быстро стать успешным. Одной из задач оптимизаторов является адаптация планов исполнения к изменившимся условиям, будь то объем данных, распределение данных или даже загрузка машины. Возвращаясь к аналогии с водителем, можно сказать, что вы можете выбирать разные маршруты в зависимости от того, едете ли вы в час пик или в более спокойное время.

Если вы запишете планы исполнения (и если это не временная мера), то рано или поздно возникнет ситуация, показанная на рис. 5.4: то, что когда-то казалось лучшим планом исполнения, перестанет быть таковым, а связанный такой «смирительной рубашкой» оптимизатор не сможет выбрать правильный план исполнения.

Даже в стабильной бизнес-обстановке вы можете столкнуться с неприятными сюрпризами: оптимизаторы являются динамической частью СУБД, и даже если вы редко видите упоминания об оптимизаторах в маркетинговых пресс-релизах, они совершенствуются от версии к версии, у них появляются новые возможности и в них исправляются ошибки. Директивы могут интерпретироваться несколько иначе, чем в предыдущих версиях. В результате серьезные обновления программного обеспечения могут стать своего рода кошмаром для приложений, в которых активно используются директивы оптимизатора.

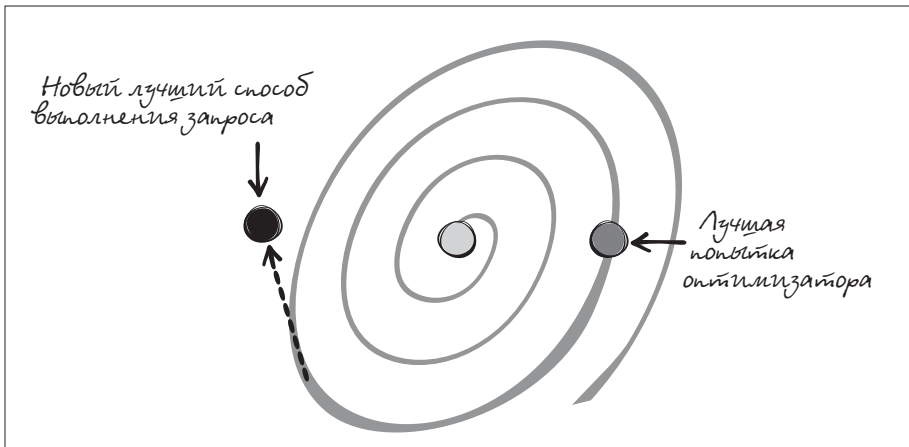


Рис. 5.4. Когда директивы оптимизатора сбивают его с толку

Надеюсь, что теперь вы поняли причины моих сомнений по поводу директив оптимизатора. Моя задача при переработке запроса – сделать оптимизатор своим союзником, а не доказывать, что я умнее его. Поэтому мне нужно переработать запрос до простейшего вида, чтобы не сбивать с толку оптимизатор, и максимально приблизиться, как на рис. 5.5, к наилучшему в данной ситуации варианту запроса. С моей точки зрения, наилучшим вариантом запроса является тот, который больше всего соответствует предполагаемым действиям SQL-машины. Возможно, это не лучший вариант, и со временем наиболее быстрый план исполнения может оказаться далек от написанного мной. Но тогда оптимизатор будет в состоянии сыграть свою роль и обеспечить стабильную производительность, а не стабильные планы исполнения.

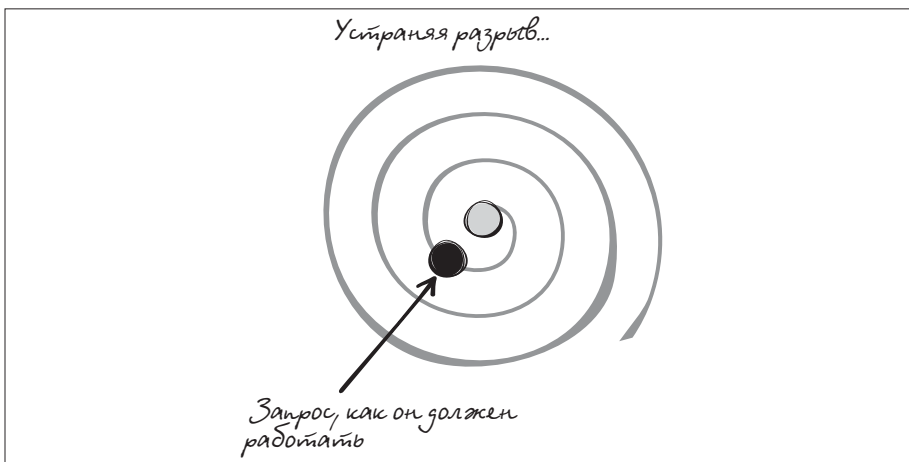


Рис. 5.5. Приближаемся к тому, что SQL-машина должна делать

Теперь, когда цель сформулирована, давайте посмотрим, как она может быть достигнута.

Анализ медленного запроса

Первое, что нужно сделать перед началом анализа медленного запроса, — это подготовиться к тому, что запрос покажется слишком сложным. И если вы еще не убежали в ужасе, то можно сказать, что половина победы уже в ваших руках. Однако если даже при свежей статистике и приемлемых индексах оптимизатор не справляется со своей задачей, возможно, проблема связана с оператором (если проблема не очевидна, это, скорее всего, означает, что она скрыта глубоко внутри представления или пользовательской функции). Если запрос ссылается на одно или несколько представлений, можно попытаться найти тексты представлений и ввести их в запрос как подзапросы (этот процесс может быть рекурсивным). Хотя запрос после этого не станет выглядеть лучше, общее представление о нем вы получите. Зачастую проще анализировать один устрашающе сложный оператор SQL, чем мешанину вызовов процедур. В конце концов, сложность, как и красоту, определяет зритель. С функциями дело обстоит несколько сложнее, чем с представлениями, о которых шла речь в предыдущей главе. Если функция ссылается на таблицы, которые не появляются в запросе, то, вероятно, на первом этапе следует оставить таблицы на месте и обратить внимание на частоту их вызовов. Если функции ссылаются на таблицы, уже присутствующие в запросе, и если логика этих таблиц не слишком сложна, возможно, стоит попытаться переместить их в запрос, по крайней мере, в качестве эксперимента.

При анализе запроса я всегда замечаяю, что самой сложной частью анализа является поиск ответа на вопрос: что пытался сделать создатель запроса?

Совет человека, который одинаково хорошо знает и функциональность, и SQL, чрезвычайно полезен, но часто возникают ситуации, когда вам нужно заняться очень старой частью кода, которую уже никто не понимает, и вы остаетесь наедине с проблемой. Если вы хотите понять что-то действительно сложное, есть только один вариант: разбить запрос на несколько более понятных частей. Первый шаг в этом направлении — выяснить, что является наиболее существенным в запросе.

Идентификация базового запроса

Когда вы работаете над сложным запросом, важно попытаться его максимально «очистить». Базовый запрос (query core) включает минимальное количество столбцов и таблиц, необходимое для идентификации конечного результирующего набора. Рассматривайте его как каркас запроса. Количество строк, которое возвращает базовый запрос, должно быть равным количеству строк, которое возвращает исходный запрос.

Количество столбцов, возвращаемых базовым запросом, вероятно, будет значительно меньше.

Можно разбить столбцы, которые предполагается обработать, на две категории: базовые столбцы, необходимые для идентификации результирующего набора (или строк, на которые вы хотите воздействовать), и «косметические» столбцы, возвращающие информацию, которая извлекается на основе содержимого базовых столбцов. Базовые столбцы являются для вашего запроса тем же, чем является первичный ключ для таблицы, но столбцы первичного ключа не обязательно являются базовыми. Внутри таблицы столбец, который является базовым для одного запроса, для другого запроса может оказаться косметическим. Например, если вы хотите вывести на экран имя заказчика при вводе его кода, столбец с именем будет косметическим, а столбец с кодом – базовым. Но если вы запустите запрос, в котором само имя является критерием выбора, столбец с именем станет базовым, а код может оказаться косметическим, если он не включен в связь, которая дополнительно ограничивает результирующий набор. Таким образом, базовыми столбцами являются:

- столбцы, к которым применяется фильтрация для определения результирующего набора;
- ключевые столбцы (как в первичном/внешнем ключе), которые позволяют связывать вместе таблицы, к которым принадлежат предыдущие столбцы.

Иногда ни один столбец таблицы, из которой вы возвращаете данные, не является базовым. Посмотрите, например, на следующий запрос, где столбец `finclsref` является первичным ключом таблицы `tdsfincbhcls`:

```
select count(*),
       coalesce(sum(t1.boughtamount)
                - sum(t1.soldamount), 0)
from tdsfincbhcls t1,
     fincbhcls t2,
     finrdvcls t3
where t1.finclsref = t2.finclsref
      and t2.finstatus = 'MA'
      and t2.finpayable = 'Y'
      and t2.rdvcode = t3.rdvcode
      and t3.valdate = ?
      and t3.ratecode = ?
```

Хотя запрос возвращает информацию только из таблицы `tdsfincbhcls`, имеющей псевдоним `t1`, в действительности результирующий набор обуславливается тем, что происходит на границе `t2/t3`, поскольку для получения всех первичных ключей (из таблицы `t1`), определяющих результирующий набор, вам нужны только таблицы `t2` и `t3`. Разумеется, если в следующем запросе подзапрос возвращает уникальные значения (в противном случае можно предположить, что в исходном запросе была ошибка), мы можем написать альтернативный вариант:


```
select count(*),
       coalesce(sum(t1.boughtamount)
               - sum(t1.soldamount), 0)
from tdsfincbhcls t1
where t1.finclsref in
      (select t2.finclsref
       from fincbhcls t2,
            finrdvcls t3
       where t2.finstatus = 'MA'
            and t2.finpayable = 'Y'
            and t2.rdvcode = t3.rdvcode
            and t3.valdate = ?
            and t3.ratecode = ?
            and t2.finclsref is not null)
```

При усовершенствовании запроса прежде всего необходимо обеспечить, чтобы подзапрос, который действительно является базовым для исходного запроса, выполнялся быстро. Потом нам потребуется выбрать нужный тип соединения, но высокая скорость выполнения подзапроса все равно останется обязательным условием для обеспечения хорошей производительности.

Приведение в порядок фразы from

Как показывает предыдущий пример, при выяснении того, что же является наиболее существенным для формирования результирующего запроса, вам следует обратить внимание на роль таблиц, появляющихся во фразе `from`. Существует три основных типа таблиц:

- таблицы, из которых извлекаются данные и к столбцам которых могут применяться или не применяться какие-либо условия (критерии поиска);
- таблицы, из которых данные не извлекаются, но к которым могут быть применены условия. Критерии, применяемые к таблицам второго типа, становятся непрямыми условиями для таблиц первого типа;
- таблицы, которые выступают только в роли «клея» между таблицами первых двух типов, позволяя связывать их с помощью соединений.

На рис. 5.6 показано, как может выглядеть типичный запрос при разных типах участвующих в нем таблиц. Таблицы представлены прямоугольниками, а стрелками показаны соединения. Возвращаемые данные извлекаются из таблиц A и D, но условие применяется также к столбцам таблицы E. Наконец, у нас есть две таблицы (B и C), из которых ничего не возвращается, а их столбцы появляются только в условиях соединения.

В результате мы получаем цепочку таблиц, связанных друг с другом с помощью соединений. В некоторых ситуациях у нас могут быть различные ветвления, и в этом случае следить надо за таблицами на концах ветвей. Рассмотрим простой вариант – таблицу E. Таблица E принадлежит к ядру запроса, поскольку к ней применяется условие (за исключением

случаев, когда условие всегда истинно), однако к фразе `from` она не принадлежит, поскольку никакие данные из нее не извлекаются. Задачей таблицы `E` является предоставление условия существования (или несуществования); ссылка на таблицу `E` говорит: «...и мы хотим получить те строки, для которых существует соответствующая строка в таблице `E`, удовлетворяющая этому условию». Логическое место, где мы можем найти ссылку на таблицу `E`, находится в подзапросе.

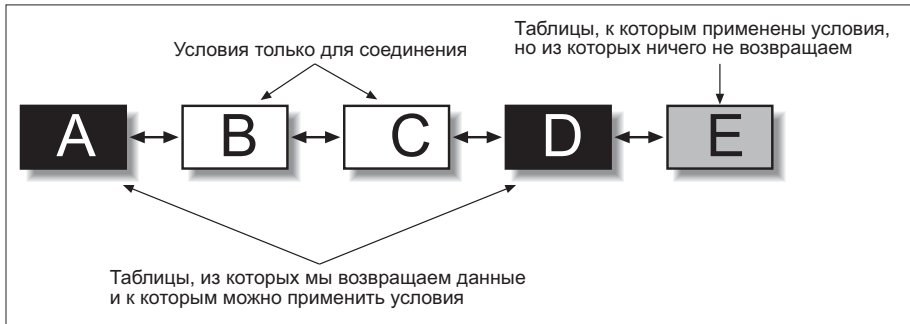


Рис. 5.6. Возможная классификация таблиц в типичном запросе

Существует несколько способов написания подзапросов (ниже в этой главе мы рассмотрим лучший из них). Кроме того, мы можем вставлять подзапросы в различные части главного запроса, в список `select`, фразу `from` и фразу `where`. На этом этапе мы не будем усложнять написание. Подчеркну тот факт, что я всего лишь пытаюсь до начала «борьбы за скорость» сделать запрос более простым и понятным. Я могу убрать таблицу `E` из фразы `from` и вместо этого написать следующее:

```
and D.join_column in (select join_column from E where <условия>)
```

Если речь идет о таблице `A`, то запрос, который одновременно возвращает данные из таблицы и применяет критерии поиска к ее столбцам, безусловно, является базовым.

Однако если какие-то данные извлекаются из таблицы `A`, но к ней не применяется никакой конкретный критерий поиска, кроме условия соединения, этот запрос нужно изучить более тщательно. «Участь» таблицы `A` в таком случае зависит от ответов на два вопроса:

1. *Может ли столбец из таблицы `B`, используемый для соединения, иметь значение `null`?*

Про значение `null` нельзя сказать, чтобы оно было равно чему-либо, в том числе и другому значению `null`. Если вы удалите соединение между таблицами `A` и `B`, строка из таблицы `B` будет принадлежать результирующему набору, и если все остальные условия удовлетворены, то столбец, участвующий в соединении, будет иметь значение `null`. Если вы сохраните соединение, строка не будет принадлежать

результатирующему набору. Другими словами, если столбец из таблицы В может иметь значение `null`, то соединение с таблицей А будет участвовать в формировании результирующего набора и станет частью базового запроса.

2. Если столбец из таблицы В, используемый для соединения, является обязательным, станет ли соединение неявной проверкой существования?

Чтобы ответить на этот вопрос, нужно понимать, что соединение может служить одной из двух целей:

- извлечь связанные данные из другой таблицы: например, получить имя и адрес заказчика при соединении по идентификатору клиента. Чаще всего это случай отношения по внешнему ключу. Имя заказчика всегда связано с его идентификатором. Мы хотим получить «косметическую» информацию. Такой тип соединения не принадлежит к базовым запросам;
- извлечь данные и проверить, существует ли соответствующая строка, если подразумевается, что в некоторых случаях ее может и не быть. В качестве примера можно привести извлечение информации о заказе для уже отгруженных заказов. Такое соединение участвует в определении результирующего набора. В этом случае следует поместить соединение внутрь базового запроса.

Провести различия между соединениями, которые только добавляют данные для уже определенного результирующего набора, и соединениями, которые участвуют в уменьшении этого результирующего набора, – один из ключевых этапов идентификации проблем. Вам нужно рассмотреть все фразы `distinct` и определить их полезность. После того как вы перенесете в подзапросы обращения к таблицам, реализующие проверку условий существования, вы поймете, что многие фразы `distinct` можно удалить без каких-либо последствий для результата запроса. Удалить же фразу `distinct` – значит не просто напечатать на восемь символов меньше. Если я высыплю перед вами содержимое моего кошелька и спрошу, сколько различных монет в нем было, сначала вам придется рассортировать монеты, что займет некоторое время. Если я дам вам только различные монеты, вы ответите немедленно.

Как и вам, SQL-машине нужно провести сортировку для устранения дублирования, а это расход ресурсов – процессора, памяти и времени.

Иногда можно заметить, что во время процесса очистки некоторые из соединений совершенно бесполезны¹. Когда я писал эту книгу, меня попросили проверить производительность приложения, которое до меня уже дважды проверяли на тот же предмет. Обе предыдущие проверки указали на запросы, построенные на приведенном ниже шаблоне, в котором

¹ Это происходит значительно чаще, чем должно происходить, особенно если запрос унаследован из другого существующего запроса, а не написан «с нуля».

используется специфичный для Oracle синтаксис (+) для указания внешнего соединения (таблица, с которой создано внешнее соединение, находится на стороне (+)). Работа всех этих запросов заняла 30 секунд.

```
select count(paymentdetail.paymentdetailid)
from paymentdetail,
     payment,
     paymentgrp
where paymentdetail.paymentid = payment.paymentid (+)
and payment.paymentgroupid = paymentgrp.paymentgroupid (+)
and paymentdetail.alive = 'Y'
and paymentdetail.errorid = 0
and paymentdetail.mode in ('CARD', 'CHECK', 'TRANSFER')
and paymentdetail.payed_by like 'something%'
```

Рекомендации содержали обычный бессистемный набор предложений: создать индекс, изменить параметры базы данных, а также, поскольку все три «таблицы» во фразе `from` в действительности были представлениями (простыми объединениями), использовать вместо них материализованные представления. Я был, вероятно, первым, кто обнаружил, что все три критерия были применены к `paymentdetail` и что удаление и `payment`, и `paymentgrp` из запроса ничего не изменило в результате, за исключением того, что запрос теперь выполнялся за три секунды (без каких-либо других изменений). Добавлю, что вообще не вижу какого-либо смысла в таком подсчете (подробнее см. в шестой главе).

На этой ранней стадии рефакторинга запроса я всего лишь пытался максимально упростить запрос и получить как можно более короткий результат. Я бы вернул только значения первичного ключа, если такое было бы возможно. Работа над очень сжатым запросом особенно важна, когда в вашем запросе присутствует агрегирование или когда форма запроса подразумевает, что агрегат типа вычисления минимальных значений может предложить изящное решение. Причина заключается том, что большинство агрегатов подразумевает сортировку. Если вы сортируете только по минимальному количеству требуемых столбцов, объем данных будет намного меньше, чем если вы будете сортировать по строкам с большим количеством информации. Вы сможете высвободить большой объем памяти и использовать меньше ресурсов, что приведет к значительно более быстрой обработке.

Теперь можно перейти к вопросу о структурных модификациях.

Рефакторинг базового запроса

Сосредоточим усилия на полученном базовом запросе. Запрос, возвращающий минимум информации, не обязательно быстр. С таким запросом легко работать, но если он производится слишком медленно, его следует переписать. Рассмотрим различные усовершенствования, которые можно внести в базовый запрос для его ускорения.

Анализ составных частей

Сложный запрос обычно состоит из комбинации более простых запросов: различных операторов `select` в сочетании с операторами объединения типа `union` или подзапросами. Выполнение этих простых запросов может следовать неявному алгоритму, согласно которому определенные части должны быть выполнены раньше других из-за наличия зависимостей. Нужно упомянуть, что оптимизатор может редактировать код SQL и фактически изменять порядок зависимостей. В некоторых случаях СУБД может даже разбить запрос и выполнять независимые части одновременно. Но, в любом случае, несомненно одно: большой медленный запрос обычно является комбинацией мелких запросов. Нет необходимости говорить, что времени для выполнения полного запроса никогда не может требоваться меньше, чем времени для выполнения одного, даже самого длинного шага.

Удачное решение можно найти быстрее, если изобразить имеющийся запрос «в разрезе» и проверить «силу» различных критериев. Это поможет вам увидеть, где следует сосредоточить основные усилия, а также понять, каких результатов можно добиться.

Устранение повторяющихся шаблонов

Теперь нужно найти таблицы, на которые есть несколько ссылок. Мы можем обнаружить ссылки на одну и ту же таблицу в нескольких местах:

- в различных частях запроса, где используются такие операторы объединения, как `union`;
- в различных частях подзапросов, например в списке `select`, фразе `from`, фразе `where` и т. д.;
- во фразе `from` (в различных формах:
 - как соединение таблицы с самой собой через различные столбцы, принадлежащие одному и тому же функциональному домену. В качестве примера можно привести запрос, который показывает имя сотрудника и имя его менеджера, если идентификатор менеджера как сотрудника является атрибутом строки, описывающей сотрудника;
 - через промежуточные таблицы, устанавливающие связь между двумя различными строками данной таблицы. Например, в генеалогической базе данных таблица браков устанавливает связь между различными строками в таблице `персон`).

Если вы обнаруживаете таблицу, которая неоднократно появляется в запросе, вам следует задать себе два вопроса (именно в таком порядке):

1. Могу ли я извлечь все необходимые данные из этой таблицы, обратившись к ней только один раз?
2. Если я могу получить все данные за один проход, стоит ли это делать?

В качестве примера приведу запрос, в котором я обнаружил следующий критерий, где *x* является псевдонимом для таблицы во фразе *from* (не показанной):

```
... and ((select min(o.hierlevel)
         from projvisibility p,
              org o
         where p.projversion = x.version
              and p.id = x.id
              and p.accessid = o.parentid
              and o.id = ?)
        = (select min(o.hierlevel)
         from projvisibility p,
              org o
         where p.projversion = x.version
              and p.id = x.id
              and p.accessid = o.parentid
              and o.id = ?
         and p.readallowed = 'Y'))
```

Перед нами очевидный случай повторяющихся шаблонов, поскольку после того, как мы убедились, что два значения на месте двух маркеро-заполнителей *?* идентичны, единственным различием между двумя подзапросами оказывается последняя строка предыдущего фрагмента кода. На что воздействует эта часть запроса? В сущности, мы обрабатываем набор значений, определенных тремя значениями, два из которых унаследованы из внешнего запроса (*x.version* и *x.id*), а третье передано как параметр. Обозначим такой набор значений как *S*.

Мы сравниваем минимальное значение для *S* с минимальным значением поднабора *S*, которое определено значением *Y* для столбца *readallowed* таблицы *projvisibility*. Назовем этот поднабор *SR*. Когда SQL-машина приступает к поиску минимального значения в *S*, она обязательно проверяет и значения *SR*. Следовательно, второй подзапрос избыточен.

Магической конструкцией SQL для сравнения поднаборов внутри набора часто является *case*. Найти два минимальных значения одновременно не слишком сложно. Для этого нужно написать следующее:

```
select min(o.hierlevel) absolute_min,
       min(case p.readallowed
            when 'Y' then o.hierlevel
            else null
          end) relative_min
from ...
```

Значение *null* такими функциями, как *min()*, игнорируется, и мы получаем два значения за один проход. Нужно только несколько модифицировать исходный запрос:

```
... and exists (select null
               from projvisibility p,
                    org o
```

```

where p.projversion = x.version
      and p.id = x.id
      and p.accessid = o.parentid
      and o.id = ?
having min(o.hierlevel) = min(case p.readallowed
                              when 'Y' then o.hierlevel
                              else null
                              end))

```

Как мы видим, устранено повторение шаблонов, а следовательно, не совершается избыточная работа (обратите внимание, что из фразы `select` удалены два агрегата, поскольку они нужны только во фразе `having`). Запрос стал проще и (по крайней мере, для меня) понятнее.

Рассмотрим другой, более сложный пример:

```

select distinct
  cons.id,
  coalesce(cons.definite_code, cons.provisional_code) dc,
  cons.name,
  cons.supplier_code,
  cons.registration_date,
  col.rco_name
from weighing w
  inner join production prod
    on prod.id = w.id
    inner join process_status prst
      on prst.prst_id = prod.prst_id
  left outer join composition comp
    on comp.formula_id = w.formula_id
  inner join constituent cons
    on cons.id = w.id
  left outer join cons_color col
    on col.rcolor_id = cons.rcolor_id
where prod.usr_id = :userid
      and prst.prst_code = 'PENDING'
union
select distinct
  cons.id,
  coalesce(cons.definite_code, cons.provisional) dc,
  cons.name,
  cons.supplier_code,
  cons.registration_date,
  cons.flash_point,
  col.rco_name
from weighing w
  inner join production prod
    on prod.id = w.id
    inner join process_status prst
      on prst.prst_id = prod.prst_id
  left outer join composition comp
    on comp.formula_id = w.formula_id

```

```
inner join constituent cons
  on cons.id = comp.cons_id
  left outer join cons_color col
    on col.rcolor_id = cons.rcolor_id
where prod.usr_id = :userid
and prst.prst_code = 'PENDING'
```

В этом запросе много повторяющихся шаблонов: в обеих частях `union` одни и те же таблицы, а списки `select` и фразы `where` идентичны. Различия обнаруживаются только в соединениях, при помощи которых таблица `cons` в первой части запроса непосредственно связана с `w`, а во второй – с таблицей `comp`, которая тоже имеет внешнее соединение с `w`.

Интересно, что возвращаемые данные извлекаются только из двух таблиц: `cons` и `col`.

Начнем со схематического представления двух частей запроса, чтобы идентифицировать его базовый запрос. Я буду использовать обозначения, которые представлены и на рис. 5.6. Таким образом, таблицы, из которых данные возвращаются обязательно, представлены в виде черных прямоугольников, а таблицы, к которым применяются условия фильтрации, но никакие данные при этом не извлекаются, показаны как серые прямоугольники. Соединительные таблицы окрашены белым цветом, соединения показаны стрелками, а внешние соединения – пунктирными стрелками. На рис. 5.7 представлена первая часть соединения.

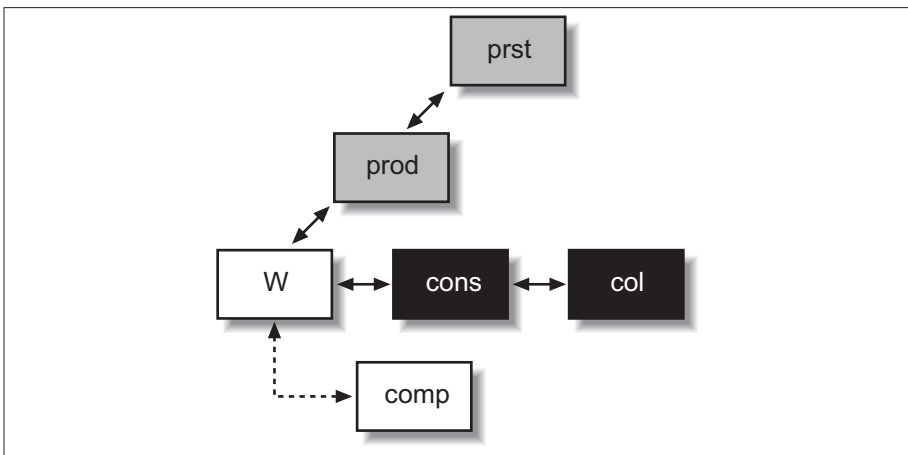


Рис. 5.7. Первая часть соединения

В этой части запроса есть два интересных момента. Во-первых, `prod` и `prst` логически принадлежат подзапросу, а поскольку `w` представляет собой не что иное, как соединительную таблицу, то и она оказывается вместе с ними.

Второй интересный момент заключается в том, что таблица `comp` имеет внешнее соединение, но из нее ничего не возвращается, а к ее столбцам не применяется никаких условий (кроме условия соединения). Это означает, что таблица `comp` не принадлежит не только базовому запросу (то есть она не принимает участия в формировании результирующего набора), но и запросу вообще. Очень часто ошибки такого типа возникают из-за использования «цельнотянутых» фрагментов кода. Соединение с таблицей `comp` в этой части запроса совершенно бесполезно. Единственное, что она может сделать, – увеличить количество возвращаемых строк (факт, коварно скрытый использованием фразы `distinct`), поскольку `union` также удаляет все дублирующие строки.

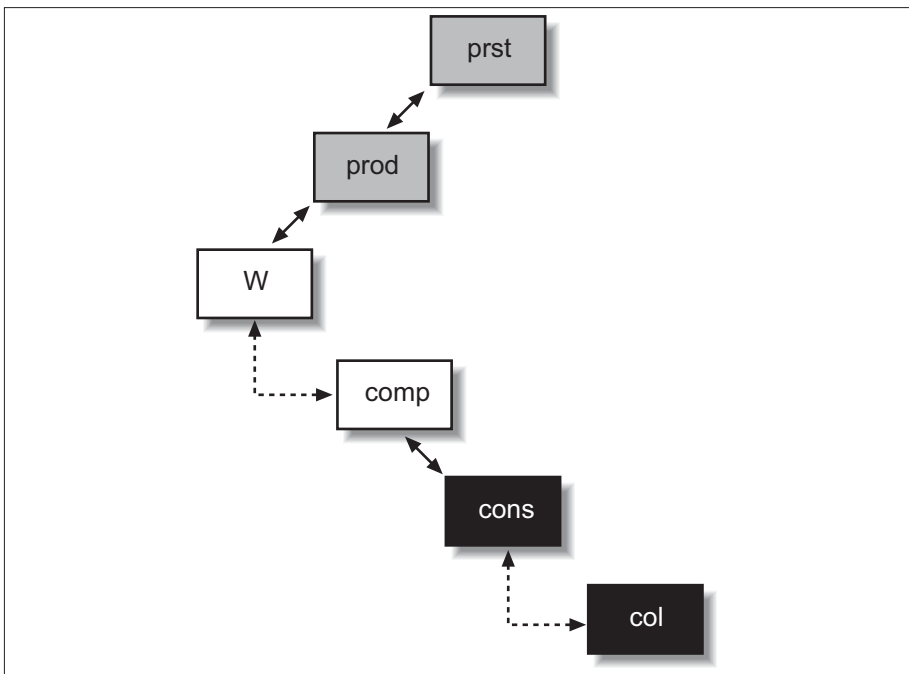


Рис. 5.8. Вторая часть объединения

Вторая часть объединения `union`, которая представлена на рис. 5.8, рассказывает другую историю. Как и в первой части `union`, таблицы `prst`, `prod` и `w` можно отнести к запросу. Однако в этом случае таблица `comp` необходима, поскольку можно сказать, что она связывает возвращаемые данные (из таблиц `cons` и `col`) с таблицами, которые позволяют точно определить, какие строки из таблиц `cons` и `col` нам нужны. Но в соединении есть что-то странное: внешнее соединение между таблицами `comp` и `w` показывает, что SQL-машина «заполнит то, что будет возвращено из таблицы `w`, значениями `null` вместо столбцов из таблицы `comp`, если вы не сможете найти соответствие в `comp`». Пока все хорошо, за исключением

того, что таблица `comp` также связана – причем обычным, внутренним соединением – с таблицей `cons` на другой стороне. Сомневаться в любых конструкциях, которые не попадают в категорию «простых и основных», – дело полезное, а внешние соединения не являются ни простыми, ни основными. Давайте предположим, что для одной строки из `w` мы не находим соответствия в таблице `comp`. Тогда для столбца `comp.cons_id`, который используется для соединения с таблицей `cons`, будет возвращено значение `null`. Значение `null` никогда не бывает равным чему-либо другому, и ни одна строка из таблицы `cons` не будет ему соответствовать. Строки, которые будут возвращены в этой части запроса, могут соответствовать только случаям, когда установлено соответствие между таблицами `w` и `comp`. Отсюда мы можем сделать вывод: соединение между таблицами `w` и `comp` должно быть внутренним, а не внешним. Внешнее соединение не добавляет значения. Но поскольку это потенциально более толерантное соединение, чем обычное внутреннее, оно может послать оптимизатору неправильный сигнал и снизить производительность.

Теперь все таблицы `prst`, `prod`, `w` и `comp` можно перенести в подзапрос, как в первой части `union`. Запрос можно переписать следующим образом:

```
select cons.id,
       coalesce(cons.definite_code, cons.provisional_code) dc,
       cons.name,
       cons.supplier_code,
       cons.registration_date,
       col.rco_name
from constituent cons
  left outer join cons_color col
    on col.rcolor_id = cons.rcolor_id
where cons.id in (select w.id
                  from weighing w
                    inner join production prod
                      on prod.id = w.id
                    inner join process_status prst
                      on prst.prst_id = prod.prst_id
                  where prod.usr_id = :userid
                  and prst.prst_code = 'PENDING')

union
select cons.id,
       coalesce(cons.definite_code, cons.provisional) dc,
       cons.name,
       cons.supplier_code,
       cons.registration_date,
       cons.flash_point,
       col.rco_name
from constituent cons
  left outer join cons_color col
    on col.rcolor_id = cons.rcolor_id
where cons.id in (select comp.cons_id
                  from composition comp
                    inner join weighing w
```

```

        on w.formula_id = comp.formula_id
    inner join production prod
        on prod.id = w.id
    inner join process_status prst
        on prst.prst_id = prod.prst_id
    where prod.usr_id = :userid
    and prst.prst_code = 'PENDING')

```

Теперь, если мы будем рассматривать подзапрос как целое вместо применения операции union к двум мощным соединениям, то сможем перенести объединение в подзапрос:

```

select cons.id,
       coalesce(cons.definite_code, cons.provisional_code) dc,
       cons.name,
       cons.supplier_code,
       cons.registration_date,
       col.rco_name
from constituent cons
     left outer join cons_color col
         on col.rcolor_id = cons.rcolor_id
where cons.id in (select w.id
                  from weighing w
                     inner join production prod
                         on prod.id = w.id
                     inner join process_status prst
                         on prst.prst_id = prod.prst_id
                  where prod.usr_id = :userid
                  and prst.prst_code = 'PENDING'
                 union
                 select comp.cons_id
                 from composition comp
                     inner join weighing w
                         on w.formula_id = comp.formula_id
                     inner join production prod
                         on prod.id = w.id
                     inner join process_status prst
                         on prst.prst_id = prod.prst_id
                 where prod.usr_id = :userid
                 and prst.prst_code = 'PENDING')

```

В реальной ситуации, из которой взят этот пример, производительность после указанных действий сильно возросла (время выполнения уменьшилось с более чем минуты до приблизительно 0,4 секунды), и я на этом остановился. В запросе по-прежнему оставались повторяющиеся шаблоны (соединение между w, prod и prst). В первой части нового объединения из таблицы w возвращался столбец id, а во второй части столбец formula_id из той же таблицы был использован для соединения с таблицей comp.

Но если бы производительность все еще оставалась неудовлетворительной, то в Oracle или SQL Server я мог бы попытаться использовать фразу with для повторяющегося запроса:

```
with q as (select w.id, w.formula_id
  from weighing w
 inner join production prod
      on prod.id = w.id
 inner join process_status prst
      on prst.prst_id = prod.prst_id
 where prod.usr_id = :userid
      and prst.prst_code = 'PENDING')
```

В таком случае объединение приняло бы следующий вид:

```
select q.id
from q
union
select comp.cond_id
from composition comp
  inner join q
      on q.formula_id = comp.formula_id
```

Хотелось бы подчеркнуть, что на этом этапе все модификации, которые я применял к запросу, были чисто логическими: в расчет не принимались размер таблиц, их структура или индексы. Иногда логического анализа оказывается достаточно. Однако часто после того, как логическая структура запроса ясно определена, наступает время внимательнее взглянуть на данные, чтобы реконструировать запрос. Проблема SQL заключается в том, что многие конструкции являются логически эквивалентными (или почти логически эквивалентными, как вы увидите). Выбор варианта может иметь принципиальное значение, поскольку разные СУБД могут обрабатывать одинаковые конструкции по-разному. Особенно это справедливо в отношении подзапросов.

Игры с подзапросами

Вы можете использовать подзапросы во многих местах, поэтому важно полностью понимать смысл альтернативных вариантов, чтобы правильно их использовать и избегать ошибок. В этом разделе я сделаю обзор наиболее значимых ситуаций, в которых можно использовать подзапросы.

Подзапросы в списке select

Первым местом, где могут быть находиться подзапросы, является список select. Например:

```
select a.c1, a.c2, (select some_col from b where b.id = a.c3) as sub
from a ...
```

Такой подзапрос имеет ограничение: он не может возвращать больше одной строки (подзапросы такого типа иногда называют скалярными). Однако он может вообще не возвращать строк, и в таком случае столбец sub в результате будет представлен как значение null. Как правило, подзапросы в списке select сопоставляются с внешним запросом:

в предыдущем примере они ссылаются на текущую строку главного запроса с помощью равенства `b.id = a.c3`.

Если оптимизатор не попытается переписать весь запрос или кэшировать промежуточные результаты, подзапрос будет выполняться для каждой возвращаемой строки. Такое решение может оказаться удачным для запроса, который возвращает не больше нескольких сотен строк, но для запроса, возвращающего миллионы строк, оно станет фатальным.

Даже когда количество строк, возвращаемых запросом, не слишком велико, наличие нескольких запросов, обращающихся к одной и той же таблице, нежелательно. Например (Случай 1):

```
select a.c1, a.c2,
       (select some_col from b where b.id = a.c3) as sub1,
       (select some_other_col from b where b.id = a.c3) as sub2,
       ...
from a ...
```

В этой ситуации два запроса могут быть заменены внешним соединением:

```
select a.c1, a.c2,
       b.some_col as sub1,
       b.some_other_col as sub2,
       ...
from a
      left outer join b
        on b.id = a.c3
```

Соединение должно быть внешним, чтобы получать значения `null`, когда ни одна строка из таблицы `b` не соответствует текущей строке из таблицы `a`.

Однако в некоторых (редких) случаях добавить новое внешнее соединение к фразе `from` нельзя, поскольку тогда одна (основная) таблица будет иметь внешнее соединение с двумя разными таблицами. В таком случае единственным выходом становится использование подобных запросов в списке `select`.

Случай 1 можно считать простым, поскольку оба подзапроса адресовались к одной и той же строке из таблицы `b`. Случай 2 несколько интереснее:

```
select a.c1, a.c2,
       (select some_col
        from b
        where b.id = a.c3
              and b.attr = 'ATTR1') as sub1,
       (select some_other_col
        from b
        where b.id = a.c3
              and b.attr = 'ATTR2') as sub2,
       ...
from a ...
```

В отличие от Случая 1, в Случае 2 подзапросы обращаются к двум разным строкам таблицы *b*, но эти две строки должны быть связаны с одной и той же строкой таблицы *a*. Мы, конечно, можем перенести оба подзапроса во фразу *from*, дав таблице *b* псевдонимы *b1* и *b2*, в результате чего получим два внешних соединения. Но можно принять нестандартное решение: слить подзапросы.

Чтобы преобразовать два подзапроса в один, мы должны, как в простом случае с идентичными условиями *where*, вернуть *some_col* и *some_other_col* одновременно. Но нужно учесть одну особенность: поскольку условия различны, нам потребуются *or* или *in*, и мы получим не одну, а две строки. Если мы хотим вернуться к результату с одной строкой, как в Случае 1, необходимо вместить две строки в одну, используя комбинирование *case* с агрегатом, как в этой главе уже упоминалось раньше:

```
select id,
       max(case attr
            when 'ATTR1' then some_col
            else null
            end) some_col,
       max(case attr
            when 'ATTR2' then some_other_col
            else null
            end) some_other_col
from b
where attr in ('ATTR1', 'ATTR2')
group by id
```

Вставив этот запрос как подзапрос, но теперь уже во фразу *from*, вы получите запрос, эквивалентный исходному:

```
select a.c1, a.c2,
       b2.some_col,
       b2.some_other_col,
       ...
from a
left outer join (select id,
                      max(case attr
                           when 'ATTR1' then some_col
                           else null
                           end) some_col,
                      max(case attr
                           when 'ATTR2' then some_other_col
                           else null
                           end) some_other_col
                    from b
                    where attr in ('ATTR1', 'ATTR2')
                    group by id) b2
on b2.id = a.c3
```

Как и в *case 1*, внешнее соединение необходимо для обработки отсутствия соответствующего значения. Если в исходном запросе одно из

значений может быть равно `null`, но не обоим значениям одновременно, обычное внутреннее соединение может решить проблему.

Какой вариант запроса можно считать наиболее эффективным? Это зависит от объема данных и от существующих индексов.

Первый шаг при переработке исходного запроса с двумя скалярными подзапросами – убедиться, что таблица `b` правильно проиндексирована. В данном случае я ожидаю индекс по `(id, attr)`. Но если заменить два скалярных подзапроса агрегатом, соединение между `a.c3` и `b.id` внезапно переходит на следующий этап: теперь я ожидаю, что сначала произойдет агрегирование, а затем соединение. Если рассматривать агрегирование отдельно, индекс по `(id, attr)` окажется малополезен или вообще бесполезен, а индекс по `(attr, id)` может быть более полезным, поскольку он выполнит вычисление агрегата более эффективно.

Теперь возникает вопрос о соотношении количества различных элементов в столбцах `id` и `attr`. Если в столбце `attr` больше значений, чем в `id`, то использование индекса, в котором `attr` появится в первой позиции, будет иметь смысл. В противном случае нам придется сравнивать производительность однократного полного сканирования таблицы `b` в сочетании с агрегированием и производительность быстрого запроса по индексу, исполненного (в данном примере) дважды для каждой возвращенной строки. Если таблица `b` очень велика, а количество строк, возвращаемых глобальным запросом, очень мало, скалярные подзапросы могут быть быстрее. Во всех остальных случаях запрос, использующий скалярные подзапросы, вероятно, будет выполняться медленнее запроса, в котором используется подзапрос с фразой `from`.

Подзапросы во фразе `from`

В противоположность запросам в списке `select`, подзапросы во фразе `from` никогда не согласуются и могут исполняться независимо друг от друга. Подзапросы во фразе `from` служат двум целям:

- выполнение операций, которые требуют сортировки. Например, `distinct` или `group by` с меньшим объемом, чем все, что возвращается внешним запросом (см. предыдущий пример);
- передача сигнала оптимизатору: «это совершенно независимая часть работы».

Подзапросы во фразе `from` удачно определены в литературе по Oracle как вложенные представления: они ведут себя (и часто исполняются) как представления, продолжительность жизни которых не превышает времени, необходимого для исполнения запроса.

Подзапросы во фразе `where`

В противоположность подзапросам в списке `select` или во фразе `from`, подзапросы во фразе `where` могут быть как согласованными с внешним запросом, так и несогласованными. В первом случае они ссылаются на «текущую строку» внешнего запроса (типичным примером согласованного

подзапроса является `and exists()`). Во втором случае подзапросы могут исполняться независимо от внешнего запроса (типичным примером является `and column_name in()`). С логической точки зрения они взаимозаменяемы. Если, например, вы хотите идентифицировать всех зарегистрированных пользователей веб-сайта, которые недавно оставляли сообщения на форуме, можно написать примерно следующее:

```
select m.member_name, ...
from members m
where exists (select null
              from forum f
              where f.post_date >= ...
                 and f.posted_by = m.member_id)
```

или:

```
select m.member_name, ...
from members m
where m.member_id in (select f.posted_by
                      from forum f
                      where f.post_date >= ...)
```

Но между двумя предыдущими фрагментами кода есть два существенных различия:

- согласованный подзапрос выполняется каждый раз, когда изменяется значение от внешнего запроса, а несогласованный подзапрос должен быть запущен только один раз;
- возможно, особое значение имеет следующий факт: если запрос согласованный, то исполнением управляет внешний запрос. Несогласованный запрос может управлять внешним запросом (в зависимости от других критериев).

Из-за этих различий несогласованный подзапрос можно перемещать во фразу `from` почти без изменений. *Почти* – потому что `in()` выполняет неявный `distinct`, который должен стать явным во фразе `from`. Следовательно, мы можем написать предыдущий пример и так:

```
select m.member_name ...
from members m
      inner join (select distinct f.posted_by
                  from forum f
                  where f.post_date >= ...) q
      on q.posted_by = m.member_id
```

В первой главе уже шла речь о том, что различные СУБД демонстрируют различную реакцию на альтернативные варианты. В результате этого мы можем убрать таблицу из фразы `from` на первом этапе нашего анализа, поскольку выполняется только проверка существования, и поместить ее обратно во фразу `from`, если СУБД предпочитает соединение вложенных представлений (`joining inline views`) больше, чем `in()`.

Нужно хорошо понимать различия между двумя вариантами. В предыдущем запросе `distinct` применялся к списку идентификаторов членов,

найденных в таблице `forum`, а не ко всем данным, возвращенным из таблицы `members`. У нас есть, в первую очередь, подзапрос.

Выбрать согласованный или несогласованный подзапрос не очень трудно, и некоторые оптимизаторы могут сделать это за вас. Однако многие забывают о том, что предположения, касающиеся индексирования при использовании согласованных и несогласованных подзапросов, различаются, поскольку это уже не та часть запроса, которая «вытащит» остальное. Оптимизатор будет добиваться максимальной эффективности с существующими индексами и не будет пытаться создавать индексы (хотя мастера настройки, имеющиеся в некоторых продуктах, могут предложить это сделать).

Таким образом, все сводится к объемам данных. Если ваш запрос возвращает мало строк, а подзапрос используется как дополнительное, в определенной степени вспомогательное, условие, выбор согласованного подзапроса будет оправданным. Он также может быть оправданным, если эквивалентный подзапрос `in()` будет возвращать сотни тысяч строк, поскольку написание подобного имеет мало смысла:

```
and my_primary_key in (subquery that returns plenty of rows)
```

Если вы действительно хотите получать доступ к множеству строк, но не хотите использовать индекс, даже уникальный, у вас есть две возможности:

- если вы ожидаете получить действительно большой результирующий набор, то лучше всего использовать хеш-соединения (подробнее о соединениях ниже). В таком случае я бы переместил подзапрос во фразу `from`;
- если вы ожидаете получить небольшой результирующий набор, первый способ использовать не следует. Предположим, что по каким-то причинам оптимизатор не смог правильно оценить количество возвращаемых подзапросом строк и ожидает, что оно будет невелико. В этом случае он может проигнорировать остальные условия (некоторые из них должны быть очень избирательными) и сосредоточиться на ссылке на первичный ключ, но это действие будет неправильным. В такой ситуации вам лучше отказаться от согласованного подзапроса.

Во время написания этой книги я столкнулся с очень интересным случаем. Вызывавший проблемы запрос входил в состав программного пакета, а непосредственного способа модифицировать код не было. Однако при мониторинге исполняемых операторов SQL я нашел причину задержки в условии, которое использовало несогласованный подзапрос:

```
...
and workset.worksetid in
    (select tasks.worksetid
     from tasks
     where tasks.projectversion = ?)
...
```

Таблица `tasks` содержала 200 000 строк. Индексирование столбца `projectversion` было явно плохой идеей, поскольку количество значений в столбце неизбежно было малым, а оставшаяся часть запроса ясно показывала, что нам были неинтересны редкие номера версий проекта. Хотя я и не ожидал, что смогу модифицировать запрос, я переписал его и заменил первоначальное условие следующим:

```
and exists (select null
            from tasks
            where tasks.worksetid = workset.worksetid
               and tasks.projectversion = ?)
```

Этого оказалось достаточно, чтобы понять: столбец `worksetid` таблицы `tasks` не был проиндексирован. При текущей схеме индексирования согласованный подзапрос и должен был быть медленным, значительно более медленным, чем несогласованный подзапрос, поскольку SQL-машине приходилось сканировать таблицу `tasks` каждый раз, когда значение `worksetid` во внешнем запросе изменялось. Администратор базы данных тут же создал индекс по столбцам (`worksetid`, `projectversion`) таблицы `tasks` в тестовой базе данных, после чего, без всякого изменения текста запроса, время исполнения внезапно уменьшилось (с приблизительно 30 секунд до сотых долей секунды). Произошло следующее: оптимизатор произвел те же преобразования, что и я. Однако теперь, когда индексирование сделало согласованный подзапрос эффективным, оптимизатор может смириться с этим типом подзапроса, что задает правильный путь выполнения запроса. Следовательно, даже если у вас нет прав на модификацию запроса, то переписав его, вы сможете понять, почему он работает медленно, и найти решение.

Ранняя активизация фильтров

Другой важный принцип, фильтрация, должен быть введен в процесс как можно раньше. Это означает, что вам нужно удалить строки до операций соединения, сортировки или агрегирования, а не после них. Я уже упоминал классический случай `where` против `having`, когда при удалении строк до агрегирования, если условия не переносятся на результаты агрегирования, разница может оказаться огромной (особенно если оптимизатор не исправляет ваши ошибки). Но есть и другие распространенные случаи. Возьмите, например, схему, подобную приведенной на рис. 5.9, где главная таблица содержит базовую информацию о количестве элементов, а периферийные таблицы – об атрибутах этих элементов, которые могут как иметь те или иные значения, так и вообще не иметь значений. Все таблицы содержат идентификатор элемента и два столбца, указывающих, когда строка была создана (обязательный столбец) или последний раз модифицирована (где может быть значение `null`, если строка с момента создания ни разу не модифицировалась). Задачей является получение списка всех элементов, которые были модифицированы в конкретный день.

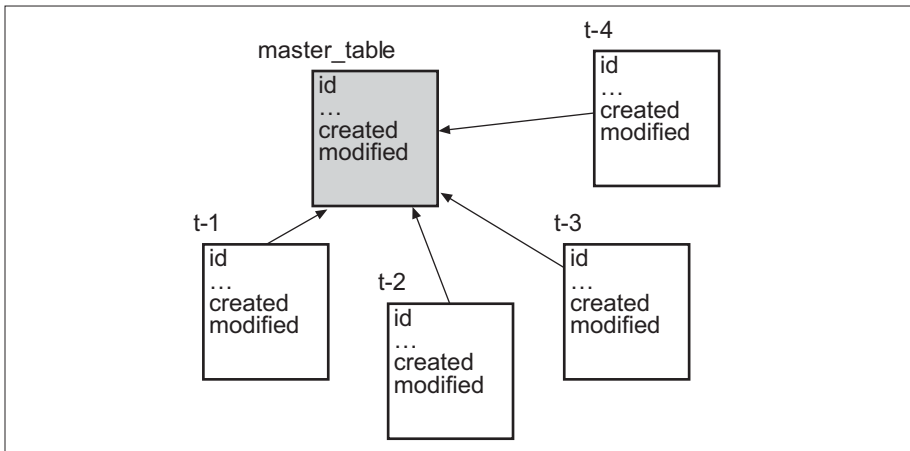


Рис. 5.9. Перечисление объектов, один атрибут которых недавно был модифицирован

Однажды я видел такой код для выполнения этой задачи¹:

```

select ...
from master_table m
  left outer join t1
    on t1.id = master_table.id
  left outer join t2
    on t2.id = master_table.id
  left outer join t3
    on t3.id = master_table.id
  left outer join t4
    on t4.id = master_table.id
where (case
  when coalesce(m.modified, m.created) > some_date then 1
  when coalesce(t1.modified, t1.created) > some_date then 1
  when coalesce(t2.modified, t2.created) > some_date then 1
  when coalesce(t3.modified, t3.created) > some_date then 1
  when coalesce(t4.modified, t4.created) > some_date then 1
  else 0
end) = 1
  
```

Участвующие в запросе таблицы не были огромными. Все они содержали от тысяч до десятков тысяч строк. Однако запрос был очень медленным, а в конечном итоге возвращал несколько строк. Маленький результирующий набор означал, что за рассматриваемый период было модифицировано только несколько строк и что условие по дате было очень избирательным. Но чтобы иметь возможность работать со всеми датами

¹ В реальном запросе использовалась функция *greatest()*, существующая в MySQL и в Oracle, но отсутствующая в SQL Server. Я переписал запрос с целью улучшения переносимости.

создания и модификации в одном выражении, SQL-машина должна сначала выполнить все соединения, а эта работа оказывается большой при наличии пяти таблиц такого размера, если перед соединением не было успешно применено условие фильтрации.

Решением в таком случае является идентификация базового запроса как (маленького) набора идентификаторов, которые относятся к недавно модифицированным строкам. Мы можем переписать базовый запрос следующим образом:

```
select id
from master_table
where coalesce(m.modified, m.created) > some_date
union
select id
from t1
where coalesce(t1.modified, t1.created) > some_date
union
select id
from t2
where coalesce(t2.modified, t2.created) > some_date
union
select id
from t3
where coalesce(t3.modified, t3.created) > some_date
union
select id
from t4
where coalesce(t4.modified, t4.created) > some_date
```

Даже когда каждый оператор `select` преобразуется в сканирование таблицы, в результате сканирования мы получаем сотни тысяч строк – другими словами, не очень много по сравнению с фильтрацией после соединения. Неявный `distinct` и последующая сортировка, которую вызывает `union`, будут действовать на малое количество идентификаторов, и вы очень быстро получите короткий список идентификаторов. В результате вам удастся осуществить отбор до соединений, после чего можно приступить к извлечению и соединению нескольких интересующих вас строк, что будет почти мгновенным процессом.

Упрощение условий

После того как вы удостоверились, что ни к одному столбцу не применяется функция, препятствующая использованию эффективного индекса по этому столбцу, вы уже не можете сделать чего-либо серьезного для улучшения обычных условий равенства или неравенства. Но мы еще не закончили рассматривать подзапросы. Даже когда мы уже выбрали, будет ли подзапрос согласованным или несогласованным, мы можем решить, что у нас много условий, которые теперь вычисляются как подзапросы (возможно, в результате приведения в порядок фразы `from`).

Поэтому следующим шагом становится минимизация запросов. Одним из основных принципов должно быть как можно более редкое обращение к каждой таблице (если вы изучали исследование операций, вспомните задачу о коммивояжере¹). Вы можете обнаружить несколько шаблонов, при которых легко добиться однократного обращения к каждой таблице:

- если несколько подзапросов ссылаются на одну и ту же таблицу, проверьте, нельзя ли всю информацию собрать за один проход. Этот тип переработки обычно получается проще в несогласованных подзапросах. Поэтому, когда у нас есть что-либо подобное:

```
and c1 in (select col1 from t1 where ...)
and c2 in (select col2 from t1 where ...)
```

(где c1 и c2 не обязательно ссылаются на столбцы одной и той же таблицы во внешнем запросе), иногда возможно, в зависимости от «совместимости» двух фраз `where`, обойтись одним запросом, который возвращает и col1, и col2. Затем можно вставить подзапрос во фразу `from` как вложенное представление (не забывая при необходимости добавить `distinct`) и создать соединение с ним;

- если, наоборот, один и тот же столбец из внешнего запроса относится к нескольким подзапросам, обращающимся к различным таблицам, как показано здесь:

```
and c1 in (select col1 from t1 where ...)
and c1 in (select col2 from t2 where ...)
```

можно будет объединить два подзапроса оператором объединения `intersect` или простым соединением:

```
and c1 in (select col1
           from t1
           inner join t2
             on t2.col1 = t1.col1
           where ...)
```

- другим относительно распространенным шаблоном являются подзапросы, которые реализуют сортировку по двум ключам, то есть «я хочу получить текст, связанный с идентификатором, для самой последней даты, для которой есть текст, соответствующий идентификатору, а если подходящих строк на эту дату окажется несколько, то текст с самым большим номером версии». Такое условие часто кодируется следующим образом:

```
select t.text
from t
where t.textdate = (select max(x.textdate)
```

¹ Зная количество городов и расстояния между ними, необходимо найти кратчайший маршрут, при котором каждый город будет посещен коммивояжером только один раз до возвращения в город, откуда начиналась поездка.

```

        from t as x
        where x.id = t.id)
and t.versionnumber = (select max(y.versionnumber)
                      from t as y
                      where y.id = t.id
                      and y.textdate = t.textdate)

order by t.id

```

Два таких подзапроса могут быть объединены с внешним запросом. В Oracle или в SQL Server я использую функцию ранжирования:

```

select x.text
from (select id,
            text,
            rank() over (partition by id
                        order by textdate desc,
                        versionnumber desc) as rnk
      from t) x
where x.rnk = 1
order by x.id

```

В MySQL я могу выполнить сортировку по `id` и, как в предыдущем случае, по уменьшению `textdate` и `versionnumber`. Чтобы получить только самую последнюю версию текста для каждого `id`, я могу использовать локальную переменную `@id` для запоминания предыдущего значения `id` и показывать только те строки, для которых значение `id` не равно предыдущему. Благодаря сортировке по убыванию это будет самая последняя версия.

Действительно сложно одновременно сравнивать текущее значение `id` со значением `@id` и присваивать это текущее значение переменной для следующей строки. Я делаю это с помощью функции `greatest()`, которая должна перед возвращением вычислить все свои аргументы слева направо. Первый аргумент функции `greatest()` – логическая функция `if()`, которая сравнивает значение столбца с переменной, – возвращает 0, если они равны (в этом случае СУБД не должна возвращать строку), и 1, если они различаются. Вторым аргументом функции `greatest()` является вызов функции `least()`, которой я передаю сначала 0, который будет возвращен и не повлияет на результат функции `greatest()`, а затем выражение, которое присваивает текущее значение `id` переменной `@id`. После вызова функции `least()` переменная принимает правильное значение, и я готов к обработке следующей строки. В качестве заключительного штриха фиктивный оператор `select` инициализирует переменную. Вот что получается после соединения всех компонентов:

```

select x.text
from (select a.text,
            greatest(if(@id = id, 0, 1), least(0, @id := id)) must_show
      from t a,
            (select @id := null) b
      order by a.id,

```

```
a.textdate desc,  
a.versionnumber desc) x  
where x.must_show = 1
```

Разумеется, перед нами случай, когда можно поставить под сомнение утверждение о том, что SQL позволяет вам сформулировать, чего именно вы хотите, без указания на то, как этого добиться, и счесть, что слово *упрощение* в заголовке этого раздела притянута за уши. Тем не менее существуют случаи, когда этот единственный запрос позволяет добиться значительного ускорения по сравнению с запросом, в котором есть несколько подзапросов.

Другие направления оптимизации

Существуют и другие направления оптимизации. Хотя я объединил их в группу под названием «небольшие усовершенствования», в некоторых случаях они могут буквально спасти медленные запросы. Но все они являются вариациями других идей и принципов, изложенных в этой главе.

Упрощение агрегатов

Как вы уже видели, агрегаты, которые вычисляются на основе одной и той же таблицы в разных подзапросах, часто могут быть преобразованы в однопроводные операции. Вам нужно запустить запрос с объединением результирующих наборов, возвращенных различными подзапросами, и с осторожностью использовать конструкцию *case* для вычисления различных сумм, подсчетов и т. п.

Использование фразы *with*

И Oracle, и SQL Server знают фразу *with*, позволяющую присваивать имя запросу, который может быть запущен только один раз, если на него есть несколько ссылок в главном запросе. Нередко в большом операторе *union* можно найти один и тот же подзапрос в различных операторах *select* объединения. Если в начале оператора вы присвоите этому подзапросу имя, как показано здесь:

```
with my_subquery as (select distinct blah from ...)
```

вы сможете изменить

```
where somecol in (select blah from ...)
```

таким образом:

```
where somecol = my_subquery.blah
```

SQL-машина может решить (но не обязательно это сделает) выполнить *my_subquery* отдельно, сохранить его результирующий набор в каком-то временном хранилище и использовать этот результирующий набор каждый раз, когда встречается ссылка на него.

Поскольку это обычно работает только с несогласованными подзапросами, использование фразы `with` может означать предварительное преобразование некоторых согласованных подзапросов в несогласованные.

Комбинирование операторов объединения

Чаще всего шаблоны SQL повторяются с операторами объединения типа `union`. Когда вы встречаете что-нибудь подобное:

```
select ...
from a, b, t1
where ...
union
select ...
from a, b, t2
where ...
```

вы можете задаться вопросом, нельзя ли минимизировать количество посещений таблиц `a` и `b`, присутствующих в обеих частях оператора, и можете предположить следующую конструкцию:

```
select ...
from a, b,
    (select ...
     from t1
     where ...
     union
     select ...
     from t2
     where ...)
where ...
```

Последний вариант не всегда возможен – например, если таблица `b` в первой части оператора связана по одному столбцу с таблицей `t1`, а во второй части – по другому столбцу с таблицей `t2`. Но не следует забывать, что есть и другие промежуточные возможности, например:

```
select ...
from a,
    (select ...
     from b, t1
     where ...
     union
     select
     from b, t2
     where ...)
where ...
```

Без тщательного изучения различных критериев, входящих в запрос, индексирования таблиц и размера промежуточных результирующих наборов невозможно предугадать, какой альтернативный вариант даст лучшие результаты. Иногда запрос, содержащий повторяющиеся шаблоны и обращающийся к различным таблицам несколько раз, демонстрирует

прекрасную производительность, поскольку есть определенное усиление различных критериев к различным таблицам, что делает каждый `select` в операторе очень избирательным и очень быстрым. Но если вы в процессе выяснения причин медленного выполнения запроса обнаружите эти повторяющиеся шаблоны, не забудьте рассмотреть и другие комбинации.

Перестроение исходного запроса

После улучшения базового запроса единственное, что вы должны сделать, это вернуть на место ссылки на те таблицы, которые были удалены на первом этапе в связи с их непричастностью к идентификации результирующего набора. В технических терминах это означает устанавливать соединения.

SQL-машина может выполнять соединения двумя способами¹. Соединения, в общем, являются операциями приведения в соответствие значений обычных столбцов (или, в некоторых случаях, нескольких столбцов). Обратите внимание на следующий пример:

```
select a.c2, b.col2
from a, b
where a.c3 = <некоторое значение>
      and a.c1 = b.col1
      and b.col3 = <некоторое другое значение>
```

Вложенные циклы

Одним из методов, приходящих на ум, является использование вложенных циклов: сканирование первой таблицы (или результирующего набора, определенного ограничивающими условиями для этой таблицы), получение для каждой строки значения в столбце соединения, поиск во второй таблице строк с соответствующим значением и возможная проверка того, удовлетворяют ли эти строки каким-либо другим условиям. В предыдущем примере SQL-машина могла бы:

- просканировать таблицу в поисках строк, удовлетворяющих условию по столбцу `c3`;
- проверить значение в столбце `c1`;
- поискать в столбце `col1` таблицы `b` строки, соответствующие этому значению;
- прежде чем вернуть столбцы в список `select`, проверить, удовлетворяет ли столбец `col3` другому условию.

¹ Я не рассматривал некоторые приемы сохранения, которые позволяют нам сохранять как единый физический объект две логически различные таблицы. Такие реализации имеют много недостатков при обычном использовании, и поэтому подходят только для очень специфических случаев.

Обратите внимание: у оптимизатора SQL есть несколько вариантов, зависящих от индексирования и от того, что он знает об избирательности различных условий: если столбец `b.col1` не проиндексирован, а `a.c1` проиндексирован, то будет значительно эффективнее начать с таблицы `b`, а не с таблицы `a`, поскольку первую таблицу придется сканировать в любом случае. Относительное качество условий по столбцам `c3` и `col3` может также свидетельствовать в пользу одной из таблиц, и оптимизатор может, например, произвести отбор по столбцу `c3` или `col3` до или после `c1` и `col1`.

Соединение слиянием и хеш-соединение

Есть и другой метод, состоящий в подготовке двух таблиц, участвующих в соединении, перед поиском соответствия.

Предположим, что у вас есть монеты, которые нужно положить в ящик, при этом монеты каждого типа должны лежать в отдельном лотке. Вы поочередно берете монеты и кладете каждую в соответствующий лоток. Это хороший метод, если монет немного, и он очень похож на вложенный цикл, поскольку вы проверяете монеты одну за другой, аналогично тому, как программа сканирует строки таблицы. Но если монет много, можно сначала рассортировать монеты, а затем положить сразу все монеты одного достоинства в нужный лоток. Именно так работает соединение слиянием. Однако если вам не хватает места, чтобы разложить сортируемые монеты, вы можете выбрать промежуточный вариант: рассортировать монеты по цвету, а не по достоинству, а затем, положить монеты в лоток, выполнить окончательную сортировку. Приблизительно так работает хеш-соединение.

Для предыдущего примера соединение таким методом будет означать следующий порядок действий SQL-машины:

- сначала отфильтровать таблицу `a` по столбцу `c3`;
- затем либо отсортировать по столбцу `c1`, либо построить хеш-таблицу, которая связывает результат вычисления `c1` со строкой, которая дала этот конкретный результат;
- затем (или одновременно с предыдущим действием) отфильтровать таблицу `b` по столбцу `col3`;
- затем отсортировать полученный набор строк по столбцу `col1` или вычислить хеш-значения по этому столбцу;
- и, наконец, сопоставить два столбца либо слиянием двух отсортированных результирующих наборов, либо с помощью хеш-таблицы.

Иногда вложенные циклы показывают большую производительность, чем хеш-соединения, иногда наоборот. Это зависит от конкретных обстоятельств. Можно заметить большое сходство между вложенным циклом и согласованным подзапросом и между соединением слиянием или хеш-соединением и несогласованным подзапросом. Если вы выполняете согласованный подзапрос так, как он написан, то получается не что иное, как вложенный цикл. Вложенные циклы и согласованные

подзапросы хороши, если объемы данных малы, а индексы избирательны. При очень больших объемах данных преимущество имеют хеш-соединения и несогласованные подзапросы, а индексы либо неэффективны, либо отсутствуют.

Следует понимать, что, например, используя во фразе `from` вложенные представления, вы посылаете оптимизатору очень сильный сигнал «хеш-соединение». Если оптимизатор не переработает запрос, когда SQL-машина исполняет вложенное представление, это закончится промежуточным результирующим набором, который нужно где-то сохранить, или в памяти, или во временном хранилище. Если этот промежуточный результирующий набор мал и если он должен быть соединен с хорошо проиндексированными таблицами, выбирать следует вложенный цикл. Если промежуточный результирующий набор велик или если он должен быть соединен с другим промежуточным результатом, для которого индекса может и не быть, единственным возможным вариантом является соединение слиянием или хеш-соединение. Поэтому вы можете писать запросы, не используя плохо управляемые директивы, но, тем не менее, явно высказывая оптимизатору ваши представления о том, как он должен исполнять запрос (при этом оставляя ему право не согласиться с вами).

6

Рефакторинг задач

*После изменений мы остаемся
более или менее прежними.*

Пол Саймон (род. 1941),
песня «Боксер»

Переработка запросов часто рассматривается как основной способ усовершенствования программ, обращающихся к базам данных. Однако это не самый интересный и не самый эффективный способ рефакторинга программ. Наверняка вы иногда встречаете запросы, серьезное редактирование которых приводит к неожиданному и значительному увеличению скорости. Но по мере того как оптимизаторы становятся более совершенными, а сбор статистики – более автоматизированным и более точным, настройка запросов вырастает в особый вид ваших навыков, и все это способствует вашему социальному самоутверждению.

В реальной жизни настройка отдельных операторов SQL уже не имеет такой важности, как раньше. Важно совершенствовать не те навыки работы с SQL, которые позволяют улучшать существующие операторы, а те, которые позволяют вам обходиться минимальным доступом к данным. Конечно, у этих навыков много общего. В пятой главе уже было продемонстрировано, что эффективная переработка операторов, как и эффективное проектирование SQL-приложения, позволяет уменьшить количество обращений к таблицам. Но вы должны увидеть общую картину и подумать не только о том, что вы можете добавить в запрос и что он будет после этого возвращать, но и о том, откуда берутся условия фильтрации и что вы будете делать с полученными данными. Когда у вас в голове сложится ясное представление об этом, возможно, вы

сможете выполнить сложный процесс за меньшее число шагов, упростив программу и используя всю мощность СУБД. Почти каждый может спроектировать последовательность относительно эффективных (не требующих дальнейшей настройки) единичных операторов SQL. Но переосмыслить всю задачу в целом – это трудное и захватывающее дело, и это то, что оптимизатор никогда не делает за вас.

Следует упомянуть, что я регулярно встречаю адвокатов «легкой поддержки», утверждающих, что простой код, который легко поддерживать, имеет преимущество перед более совершенным, но сложным кодом. Я считаю, что этот аргумент распространяется в очевидных интересах людей, хорошо известных мне в связи с профессиональной деятельностью, – менеджеров проектов ИТ-компаний, пишущих приложения, – которые защищают свою работу и стараются переложить вину за низкую производительность на службы технической поддержки, администраторов СУБД, саму СУБД и оборудование.

Я, вероятно, пристрастен, но я не уверен, что множество вызовов, вложенных циклов и ветвлений действительно приводит к улучшению сложного запроса SQL в середине довольно простой программы, особенно если я вижу различие в производительности. Кривая изучения в случае SQL, вероятно, круче, чем в случае процедурных языков, которые молодые разработчики обычно изучают в вузах, но я не верю, что снижение стандартов программирования приведет к хорошим результатам в долгосрочной перспективе. Я хочу только показать, что вы можете сделать, какую выгоду получить, хочу сравнить сложность процедурных конструкций со сложностью SQL, но решать, что делать, вы будете сами.

SQL-мышление

Любой специалист по SQL скажет, что если вы пишете операторы SQL, думать нужно не так, как если вы пишете процедурную или объектно-ориентированную программу. Вы работаете не с переменными, а со строками, которых могут быть миллионы. Если, как многие, вы представляете таблицу SQL как разновидность массива записей или файла, то вы, скорее всего, будете обрабатывать ее как массив или файл в процедурной программе. Ссылка на «оператор» SQL может быть обманчивой для программистов, имеющих опыт разработки на других языках. Операторы SQL являются операторами ровно в той степени, в которой они указывают, какие данные они хотят получить и модифицировать, но они не являются операторами в том смысле, который вкладывается в понятие процедурных операторов, являющихся просто шагами программы. Оператор SQL сам по себе фактически является программой, генерирующей много низкоуровневых операций, что, как я надеюсь, было ясно показано в пятой главе. Трудность состоит в том, что операторы SQL (и, в частности, операторы `select`) не слишком хорошо состыкуются с процедурными языками: для извлечения данных вам приходится обрабатывать результирующий набор в цикле, поэтому курсоры

и циклы являются составной частью подавляющего большинства приложений SQL.

Использование SQL там, где SQL работает лучше

Хотя циклы в некоторых местах программ SQL, вероятно, неизбежны, в программах встречается очень много ненужных конструкций. Объявление в начале процедуры десятков переменных для хранения результатов, полученных из базы данных, часто оказывается дурным знаком. При виде обработки результирующего набора в цикле я задаюсь вопросом: что же можно будет делать с полученными из базы данными? Если вы показываете данные пользователю, или записываете в файл, или пересылаете по сети, то против использования циклов нет никаких возражений. Но если извлеченные данные просто передаются в серию каскадных операторов или если они дальше обрабатываются в программе для получения результатов, которые СУБД могла бы вернуть непосредственно, возникает подозрение, что SQL используется неправильно.

Можно поставить вопрос и по-другому: какую стоимость добавляет ваш код в обработку данных? Я однажды столкнулся со случаем, который, не являясь типичным, может служить примером неправильного использования SQL. Псевдокод выглядел примерно так:

```
открыть курсор, связанный с запросом
loop
    извлечь данные из курсора
    выйти из цикла, если данные не найдены
    if (какие-то данные, возвращенные запросом, равны константе) then
        ...
        обработка
        ...
    end if
end loop
```

В конструкции `if` не было части `else`. Если мы хотим обработать только подмножество того, что возвращает курсор, зачем возвращать все? Значительно проще было бы добавить условие во фразу `where` запроса. Конструкция `if` не добавляет стоимости. Извлечение большого количества строк, чем необходимо, означает, что по сети будут переданы лишние данные и что часть извлеченных данных вообще не будет использована.

Один раз меня вызвали спасти программу, написанную в 8000 миль от места, где была создана программа из предыдущего примера. Проблема состояла не только в том, что программа работала медленно, сколько в том, что она регулярно выходила из строя после исчерпания памяти. Я обнаружил цикл по результату запроса (как обычно), извлекавшего данные в структуру под названием `accrFacNotPaid_Struct`, а внутри этого цикла – следующий фрагмент кода¹:

¹ Идентификаторы, как обычно, изменены.

```

TripleKey tripleKey = new TripleKey(accrFacNotPaid_Struct.pid_deal,
    accrFacNotPaid_Struct.cycle_rid,
    accrFacNotPaid_Struct.expense_code);
if (tripleKeyValues.containsKey(tripleKey)) {
    MapFacIntAccrNotPaidCache.FacIntStr
        existFacNotPaid_Struct =
        (MapFacIntAccrNotPaidCache.FacIntStr)tripleKeyValues.get(tripleKey);
    existFacNotPaid_Struct.accrual_amount += accrFacNotPaid_Struct.accrual_
amount;
}
else
{
    tripleKeyValues.put(tripleKey, accrFacNotPaid_Struct);
    mapFacIntAccrNotPaidCache.addAccrNotPaid(accrFacNotPaid_Struct);
    mapFacIntAccrNotPaidCache.addOutstdngAccrNotPaid(accrFacNotPaid_Struct);
}

```

Как мне кажется, дублирование данных, полученных из базы данных, в динамические структуры – не самая удачная идея. С помощью следующего небольшого фрагмента кода я добился того, чего хотел раз-работчик:

```

select pid_deal, cycle_rid, expense_code, sum(accrual_amount)
from ...
group by pid_deal, cycle_rid, expense_code

```

А как виртуозно использовались коллекции Java!

Такие операции легко выполнить в SQL, поэтому нет необходимости делать это в коде. SQL обрабатывает данные быстрее, именно для этого он и был создан.

Рассчитывайте на успех

Еще одно различие между процедурным подходом и философией SQL обнаруживается в процессе обработки ошибок и исключений. Когда вы пишете процедурную программу, вам приходится (*прежде* чем делать что-нибудь еще) проверять, имеют ли переменные смысл в контексте, в котором они используются. Если вы встречаете процедурную программу, которая делает то же самое при взаимодействии с базой данных, не сомневайтесь, что ее можно значительно улучшить.

Многие разработчики предполагают, что ресурсоемкими взаимодействиями с базой данных являются только те, что приводят к операциям физического ввода-вывода на сервере базы данных. В действительности любое взаимодействие с базой данных ресурсоемко: самые быстрые операторы SQL выполняют множество машинных инструкций, даже если не принимать во внимание взаимодействие между программой и сервером базы данных.

Если, например, при работе с C++ паранойя может считаться почти достоинством, то она превратится в недостаток, если каждую отдельную

проверку реализовывать с помощью запроса SQL. Если вы программируете на языке SQL, вам нужно думать не о «контроле», а о «целостности данных» или, если угодно, «безопасной операции». Оператор SQL должен делать только то, для чего он предназначен, и ничего больше. Но вместо того, чтобы пытаться защитить исполнение вашего оператора колючей проволокой и рвами с водой, вам следует встроить меры безопасности в оператор.

Рассмотрим пример. Предположим, спецификации вашего проекта таковы:

У нас есть несколько учетных записей участников, которые заблокированы за неуплату членских взносов. Для каждого идентификатора учетной записи процедура должна проверить, делает ли сумма платежей, полученных за предыдущую неделю, остаток на счете положительным. Если остаток положителен, а учетная запись заблокирована, ее нужно разблокировать.

Если учетной записи не существует, остаток отрицателен или учетная запись не заблокирована, должно появиться соответствующее сообщение об ошибке.

Логика кода может выглядеть так:

```
// Проверить, что учетная запись существует
select count(*)
from accounts
where accountid = :id
if count = 0 => "Счет не существует"
// Проверить, что остаток положительный
//(group by, поскольку может быть несколько транзакций)
select count(x.*)
from (select a.accountid
      from accounts a, payment b
      where b.payment_date >= <today minus seven days>
        and a.accountid = b.accountid
      group by a.accountid
      having sum(b.amount) + a.balance >= 0) as x
if count = 0 => "Недостаточная сумма платежей"
// Проверить, что учетная запись заблокирована
select count(*)
from accounts
where accountid = :id
  and locked = 'Y'
if count = 0 => "Учетная запись не заблокирована"
// Разблокировать учетную запись
update accounts
set locked = 'N'
where accountid = :id
```

Это прямая реализация спецификации, и ее нельзя считать удачной в контексте SQL. В данном случае правильным будет убедиться, что при

несоблюдении указанных условий `update` ничего не будет делать. Если учетной записи не существует, ничего страшного не произойдет. Нам нужно всего лишь выяснить, не является ли остаток на счете отрицательным и не заблокирована ли учетная запись в данный момент.

Поэтому мы можем написать безопасный оператор `update` следующим образом:

```
update accounts a
set a.locked = 'N'
where a.accountid = :id
    and a.locked = 'Y'
    and exists (select null
                from payments p
                where p.accountid = a.accountid
                    and p.paymentdate >= <сегодня минус семь дней>
                group by p.accountid
                having sum(p.amount) >= a.balance)
```

Такой оператор нанесет базе данных не больше ущерба, чем даже сверхзащищенный. Разумеется, в процессе переработки оператора мы потеряли возможность проверки наличия ошибок. Но так ли это? Все интерфейсы баз данных предоставляют либо функцию, либо переменную среды, которая сообщает нам, сколько строк было обновлено последним оператором. Предположим, мы говорим о переменной `$$PROCESSED`. Тогда можно заменить отбор с вытеснением на криминалистический анализ:

```
if ($$PROCESSED = 0) then
/* Это не то, чего мы ожидали */
select locked
from accounts
where a.accountid = :id
if данные не найдены => учетная запись не существует
else if locked = 'N' then учетная запись не заблокирована
else недостаточная сумма платежей
```

Эти два алгоритма реализованы здесь как функции Visual Basic (всю программу, описание которой приведено в приложении А, вы можете загрузить с сайта этой книги). Функция `unlocked_1` реализует неэффективный алгоритм внутри приложения, которое сначала проверяет условия, а после получения сигнала «все чисто» выполняет обновление. Функция `unlocked_2` реализует правильный алгоритм SQL, сначала выполняющий запрос и потом проверяющий, было ли все выполнено так, как ожидалось. На рис. 6.1 показан результат выполнения обеих процедур приблизительно с 40 000 учетных записей и с переменной долей строк, обработка которых закончилась неудачей.

Понятно, какой метод эффективнее: мы можем ожидать сближения, если обработка оказалась неудачной в большинстве случаев, но в случае разумной доли ошибок «оптимистичная» версия, реализованная

в `unlocked_2`, демонстрирует вдвое большую производительность, чем «пессимистичная» версия функции `unlocked_1`.



Рис. 6.1. Функция, которая проверяет и делает, в сравнении с функцией, которая делает и проверяет (SQL Server/Visual Basic)

Каждый тест, который получает доступ к базе данных и предшествует оператору, выполняющему реальное действие, имеет более высокую стоимость, чем остальная часть кода. Если вы сначала делаете, а потом (при необходимости) проверяете, накладные расходы будут только у операций, которые первыми закончились неудачей. Даже если проверка происходит очень быстро, скажем, за 0,001 секунды процессорного времени, то если вы повторите ее два миллиона раз, это займет 33 минуты процессорного времени, что может оказаться неприемлемым в пиковые периоды, когда процессор становится особенно важным ресурсом.

Реструктуризация кода

Как было показано в пятой главе, один из ведущих принципов рефакторинга операторов заключается в том, чтобы попытаться уменьшить количество «посещений» каждой таблицы и (особенно) каждой строки. Для рефакторинга задач вы должны применять те же принципы: если одна и та же таблица принимает участие в двух последовательных операторах, нужно выяснить, действительно ли то, что делает программа

между двумя операторами, оправдывает лишнее обращение к серверу базы данных. Определить, можно ли объединить два последовательных оператора для одного «похода» на сервер базы данных, не очень сложно. Программы, логика которых запутанна, сложно разбить на десяток эффективных операторов. Но язык SQL позволяет сделать значительную часть логики доступной для данных, как будет продемонстрировано ниже.

Объединение операторов

Очень часто в программах можно встретить последовательности операторов, которые можно объединить. Например, я видел несколько сценариев, в которых либо несколько последовательных обновлений было сделано в одной и той же таблице (при этом некоторые из них легко можно было бы реализовать в одном операторе), либо данные, вставленные одним оператором, обновлялись следующим, хотя конечный результат можно было вставить за один прием. Такое разбиение операций очень часто встречается в сценариях, образующих системы принятия решений и осуществляющих миграцию данных перед обновлением программного обеспечения. Приходит на ум последовательность операций, которые хороши, но написаны одинаково плохо. Наличие `or` во фразе `where` и разумного использования фразы `case` часто оказывается достаточно для уменьшения времени исполнения в два-три раза (а иногда и более) в зависимости от количества операторов, которые вам удастся объединить.

Введение управляющих структур в SQL

Несколько последовательных запросов, адресованных одной и той же таблице, должны вас насторожить, даже если они встроены в конструкции `if ... then ... else`. Взгляните на следующий пример, содержащий три запроса, но всего с двумя ссылками на таблицы (идентификаторы, начинающиеся с `p_`, обозначают параметры, передаваемые запросам):

```
select count(*)
into nAny
from t1
where chaid = p_chaid
      and tneid = p_tneid
      and levelname = 'LEVEL1';
if nAny > 0 then
  [ assignments 1 ]
else
  select count(*)
  into nAny
  from t1
  where chaid = p_chaid
        and tneid = p_tneid
        and levelname = 'LEVEL2';
  if nAny > 0 then
```

```

        [ assignments 2 ]
    else
        [ assignment 3 ]
        if p_taccode = 'SOME VALUE' then
            select count (*)
            into nAny
            from t2
            where taccode = p_taccode
                and tpicode = p_tpicode
                and tneid = p_tneid
                and ttelevelname = 'LEVEL2';
            if nAny > 0
                [ assignment 4 ]
            else
                [ assignment 5 ]
            end if;
        else
            [ assignment 6 ]
        end if;
    end if;
end if;

```

Первые два запроса обращаются к одной и той же таблице. Хуже всего то, что в них применяются одни и те же условия, за исключением значения одной константы. Конечно, у нас могут быть гистограммы, индекс по столбцу levelname и сильно различающиеся распределения значений LEVEL1 и LEVEL2. Прежде чем утверждать, что можно добиться значительного увеличения производительности, нужно проверить эти возможности. Тот факт, что два столбца из трех во фразе where имеют суффиксы id, заставляет меня предположить, что они принадлежат составному первичному ключу. Если мое предположение верно, скорость исполнения двух запросов будет приблизительно одинаковой, но если нам удастся объединить их, получившийся запрос должен выполняться значительно быстрее.

Использование агрегатов

Агрегаты часто оказываются удобными для упрощения конструкций if ... then ... else. Они позволяют нам вместо последовательной проверки различных условий получить за один проход несколько результатов, которые позже можно будет проверить без доступа к базе данных. Предыдущий пример будет хорошей основой для демонстрации этого приема. Если я заменю первые два оператора на следующий фрагмент:

```

select sum(case levelname
            when 'LEVEL1' then 1
            else 0
            end) as count1,
       sum(case levelname
            when 'LEVEL2' then 1
            else 0

```

```

        end) as count2
into nAny1, nAny2
from t1
where chaid = p_chaid
    and tneid = p_tneid
    and levelname in ('LEVEL1', 'LEVEL2');
```

я вряд ли сделаю первый запрос значительно более долгим, и мне больше не придется запускать второй запрос: нужно будет всего лишь последовательно проверить значения `nAny1` и `nAny2`. Коэффициент ускорения будет, разумеется, зависеть от того, сколько раз я получу ноль в `nAny1` и в `nAny2`. Один или два запроса помогут нам сразу же оценить ускорение.

Использование функции `coalesce()` вместо `if ... is null`

Некоторые тесты могут потребовать определенных замен. Следующий фрагмент короче, но интереснее предыдущего. Его задача состоит в определении значения, которое надо присвоить переменной `s_phacode`:

```

select max(dphorder)
into n_dphorder
from filphase
where filid = p_filid
    and phacode || stepcode != 'RTERROR'
    and phacode || substr(stepcode,1,3) != 'RTSUS'
    and dphdteffect <= p_date;
if n_dphorder is null then
    s_phacode := 'N/A'
else
    select max(phacode)
    into s_phacode
    from filphase
    where filid = p_filid
    and dphorder = n_dphorder;
end if;
```

Нельзя сказать, что сравнение конкатенированных столбцов с константами мне нравится. Но тот факт, что условиями являются неравенства, означает, что индексы, которые предпочитают равенства, вряд ли могут нам помочь. Я предпочел бы видеть следующее:

```

where (phacode != 'RT'
    or (stepcode != 'ERROR' and stepcode not like 'SUS%'))
```

Мы обсуждаем детали, и это не должно снизить производительность, поскольку негативные условия вряд ли будут выборочными.

Поскольку единственным назначением переменной `n_dphorder` является извлечение значения, которое будет использовано дальше, первый шаг состоит в замене переменной во втором запросе на первый запрос:

```

select coalesce(max(phacode), 'N/A')
into s_phacode
from filphase
```

```
where filid = p_filid
and dphorder = (select max(dphorder)
                from filphase
                where filid = p_filid
                and phacode || stepcode != 'RTERROR'
                and phacode || substr(stepcode,1,3) != 'RTSUS'
                and dphdteffect <= p_date);
```

Если подзапрос вернет null, главный запрос тоже вернет null, поскольку невозможна ситуация, при которой условие по dphorder было бы удовлетворено. Я предпочитаю использовать функцию coalesce() вместо проверки в вызывающем коде, так что вне зависимости от того, что произойдет, я получу то, что нужно, в одном запросе.

У меня по-прежнему есть две ссылки на таблицу filphase, но они находятся внутри одного запроса. Поскольку ожидается, что в большинстве случаев первый исходный запрос вернет определенное значение, должно произойти только одно обращение к SQL-машине. В противном случае обращений было бы два. Важно то, что SQL-машина получает запрос, который дает оптимизатору больше возможностей для улучшения, чем два отдельных запроса. Конечно, мы также можем попробовать еще больше переработать запрос и облегчить оптимизатору работу (об этом шла речь в предыдущей главе). В запросе в его нынешнем виде нам придется проверить несколько вещей. Если (filid, dphorder) окажется первичным ключом (что вполне возможно), то ограничение следующего запроса первой возвращенной строкой даст особые возможности, при условии, что вы готовы иметь дело со случаем, когда никакие данные не найдены:

```
select phacode
from filphase
where filid = p_filid
      and dphdteffect <= p_date
      and (phacode != 'RT'
           or (stepcode != 'ERROR' and stepcode not like 'SUS%'))
order by dphorder desc, phacode desc
```

Использование исключений

Иногда оказывается невозможным эффективно объединить или переписать операторы, но небольшое расследование обычно позволяет обнаружить много процедурных процессов, которые могут быть написаны лучше, особенно в хранимых функциях и процедурах. «Служебные» функции, которые обычно вызываются снова и снова, являются отличной основой для рефакторинга. Когда вы отслеживаете операторы, обращающиеся к базе данных, и ищите относительно короткие операторы, которые исполняются так часто, что сильно загружают сервер, загляните в служебные функции: функции просмотра, функции преобразования и т. п.

Можно избавиться от определенных пользователем функций, которые выполняют запросы к базе данных, и заменить эти функции (если такая

функциональность действительно вам нужна) на представления. Не нужно бояться кажущейся сложности этих функций – очень часто сложность просто является результатом неправильной обработки SQL.

Вот еще один реальный пример. В приложении Oracle есть несколько отдельных фрагментов кода. Все фрагменты кода идентифицированы именами и перечислены в таблице `codes`. Каждому фрагменту присвоен номер в таблице `codes`, а приложение является интернациональным, поэтому с каждой комбинацией *код + поддерживаемый язык* связано сообщение. Переведенные сообщения хранятся в таблице `code_values`. Код функции таков:

```

/* Function getCodeValue
Desc : Returns, in the language specified, the value of a code provided
      in a code series provided.
Parameters : 1. p_code_series      - Name of the code series
              2. p_code            - Code to be searched for
              3. p_language        - Language to be used for searching
              4. p_ret_desc (boolean) - True, if function should return
                                      the decoded text
                                      False, if function should return
                                      the no. of records returned from
                                      the code_values table
                                      (mostly this helps to verify, if
                                      it returns 1 or 0 rows)

*/
function getCodeValue( p_code_series in codes.code_series%type
                      , p_code in code_values.code%type
                      , p_language in code_values.language%type
                      , p_ret_desc in boolean :=true
                      -- true = Description , false=count
                      ) return varchar2 is
cursor c_GetCodeText ( v_code_series in codes.code_series%type
                      , v_code in code_values.code%type
                      , v_language in code_values.language%type
                      ) is
select cv.text
from code_values cv,
     codes c
where c.code_series = cv.code_series
     and c.code_series = v_code_series
     and cv.code = v_code
     and cv.language = v_language;
v_retvalue code_values.code_value_text%type;
v_temp number;
begin
select count(1)
into v_temp
from code_values cv,
     codes c
where c.code_series = cv.code_series
     and c.code_series = p_code_series

```

```

        and cv.code = p_code
        and cv.language = p_language;

    if p_ret_desc then -- return the Decoded text
        /* check if it returns multiple values */
        if coalesce(v_temp,0) = 0 then
            /* no value returned */
            v_RetValue := '';
        elsif coalesce(v_temp,0) = 1 then
            /* if it returns one record, then find the decoded text and
               return */
            open c_GetCodeText(p_code_series, p_code, p_language);
            fetch c_GetCodeText into v_retvalue;
            close c_GetCodeText;
        else
            /* if it returns multiple values, then also return only first
               value. We can modify the code here to return NULL or process
               it differently if needed.
            */
            open c_GetCodeText(p_code_series, p_code, p_language);
            fetch c_GetCodeText into v_retvalue;
            close c_GetCodeText;
        end if;
    else -- return only the count for the code
        v_retvalue := v_temp;
    end if;
    return v_retvalue;
end getCodeValue;

```

Теперь все правильно. Перед нами два очень похожих. Один из них проверяет, есть ли соответствие, а другой (запрос с курсором) выполняет работу. Рассмотрим запрос повнимательнее:

```

select cv.text
from code_values cv,
     codes c
where c.code_series = cv.code_series
     and c.code_series = v_code_series
     and cv.code = v_code
     and cv.language = v_language;

```

Параметр `v_code_series` используется для указания «семейства фрагментов», а значение `code_series`, соответствующее этому параметру, используется, в свою очередь, для поиска соответствующей строки в `code_values`. Как вы видели в предыдущей главе, единственным обоснованием для такого действия является необходимость проверить, действительно ли в таблице `code` есть соответствующая строка. Oracle может предотвратить появление несоответствий между `codes` и `code_values`. Беспольный доступ к таблице `codes` с помощью ее первичного ключа вряд ли существенно увеличит стоимость запроса. Но можно ожидать, что триплет (`code_series`, `code`, `language`) окажется первичным

ключом для таблицы `code_values`, и при выполнении двух обращений к первичному ключу там, где достаточно одного, эта стоимость увеличится вдвое. В некоторых случаях эта функция используется интенсивно и может иметь значение.

Теперь обратим внимание на функцию. Одна и та же ошибка сделана дважды: и в запросе, который осуществляет подсчет, и в запросе, который возвращает значение. Но зачем подсчитывать, если нам действительно нужны данные? Попробуем проанализировать, что происходит после того, как количество соответствующих значений подсчитано:

1. Сначала проверяется `coalesce(v_temp, 0)`. Нет надобности быть параноиком. При подсчете вы можете получить либо 0, либо значение, большее нуля. Получить значение `null` невозможно. Могло быть и хуже; мы, по крайней мере, избежали `coalesce(abs(v_temp), 0)`. Конечно, это очень досадная ошибка, но она не окажет реального влияния на производительность. Однако наличие конструкций такого типа можно считать сигналом, который требует, чтобы вы продолжили поиск более серьезных ошибок в коде. Как пошутил один из первых рецензентов этой книги Дэвид Нур, глупые незначительные ошибки частенько оказываются рядом с не менее глупыми, но значительными ошибками.
2. Затем нужно проверить, сколько кодов найдено – один, ни одного или более одного. Если кодов не больше одного, то предварительный подсчет не требуется: нужно только извлечь значение и проверить состояние ошибки, генерируемое, если извлекать нечего (в PL/SQL это реализуется с помощью блока исключений, который будет искать predefinedное исключение `no_data_found`). Для другого состояния ошибки, когда найдено несколько кодов, есть другое predefinedное исключение – `too_many_rows`. Но, как я уже сказал, предполагаю, что `(code_series, code, language)` является первичным ключом таблицы `code_values`. Если первичные ключи определены правильно (а я не думаю, что требую слишком многого), такое не может происходить чаще, чем `count(*)` будет возвращать `null`. Из длинного комментария мы можем понять, что разработчик добросовестно старался продумать свою работу. Но он просто не думал в терминах SQL.

То, как написан код (с использованием `if ... elsif ... else ... end if` вместо нескольких блоков `if ... end if`), и размер комментария доказывает, что у разработчика есть определенный опыт и что он хорошо владеет процедурными языками. Но есть некоторые основания полагать, что разработчик имеет довольно смутное представление об SQL и реляционных базах данных.

Структура рассматриваемой функции должна быть такой:

```
begin
  select text
  into v_text
  from code_values
```

```
where code_series = p_code_series
      and code = p_code
      and language = p_language;
if p_ret_desc then
  return v_text;
else
  return 1;
end if;
exception
  when no_data_found then
    if p_ret_desc then
      return '';
    else
      return 0;
    end if;
end;
```

Все остальные проверки в коде становятся бесполезными после:

```
alter table codes
add constraint codes_pk primary key(code_series);
alter table code_values
add constraint code_values_pk primary key(code_series, code, language);
alter table code_values
add constraint code_values_fk foreign key (code_series)
references codes(code_series);
```

Если нужно оценить, насколько проведенная работа улучшила исходную функцию, я могу считать, что удаление бесполезных соединений в запросе, который был исполнен дважды, увеличило производительность примерно вдвое.

Мы можем сделать функцию (в том виде, в котором я переписал ее раньше), немного более эффективной, просто проверив существование сообщения. Если я хочу видеть возвращаемый текст, SQL-машина будет искать индекс по первичному ключу и найдет адрес соответствующей строки (за исключением случаев, когда индекс является кластерным, но в случае Oracle этого не будет). Получение самой строки потребует одного дополнительного доступа к другому блоку в памяти, что окажется бесполезным, если я всего лишь хочу проверить существование элемента индекса. Поэтому, если проверка существования действительно важна (что маловероятно), мы можем улучшить функцию, исполнив `select 1`, когда текст не нужен. Но необходимость варианта, который только подсчитывает, для меня не вполне очевидна. Проверить, что возвращено, более чем достаточно.

Из этого примера можно вынести урок: если вы хотите создать действительно эффективную программу, очень важно определить границу между тем, что относится к базе данных, и тем, что относится к «оболочке». Любой разработчик старается делать то, что он умеет лучше всего, и это естественно. Делать слишком много на процедурной стороне, — несомненно, самый распространенный подход, но я встречал

и противоположный случай – людей, предпочитающих выполнять все процессы внутри базы данных (что тоже часто бывает абсурдом). Если вы хотите действовать эффективно, молоток не должен быть вашим единственным инструментом.

Конечно, если функция становится совсем простой, ее смысл можно поставить под сомнение. Если она вызывается из оператора SQL, внешнее соединение с `code_values` будет значительно более эффективным.

Извлечение всех нужных данных за один прием

Если вам нужно получить что-либо из базы данных, то значительно эффективнее будет написать запрос так, чтобы он всегда возвращал то, что вам нужно в конечном итоге, не требуя подчиненного запроса. Это правило становится еще более важным, когда есть вероятность того, что запросов будет несколько. Представьте, например, сообщения на различных языках, которые хранятся в базе данных. Часто бывает так, что когда вы хотите вернуть какой-то текст на данном языке, вы не уверены, все ли строки переведены. Но обычно есть по крайней мере один язык, на котором все сообщения существуют, и этот язык определяют как «язык по умолчанию»: если нельзя найти сообщение на родном языке пользователя, возвращают сообщение на языке по умолчанию (если для пользователя оно бесполезно, то пригодится хотя бы службе технической поддержки).

Для наших целей языком по умолчанию будет, предположим, Волапюк (код `vo`).

Вы можете написать следующий код:

```
select message_string
from text_table
where msgnum = ?
      and lang = ?
```

Если запрос не вернет ни одной строки, то, чтобы получить сообщение на Волапюке, вы можете запустить такой запрос:

```
select message_string
from text_table
where msgnum = ?
      and lang = 'vo'
```

В качестве альтернативы вы можете написать один запрос:

```
select message_string
from (select message_string,
             case lang
               when 'vo' then 2
               else 1
             end k
from text_table
where msgnum = ?
      and lang in (?, 'vo'))
```

```
union all
select '???' message_string,
      3 k
from dual
order by 2) as msg
```

и ограничить вывод первой возвращенной строкой, используя синтаксис вашего диалекта SQL (`top 1`, `limit 1` или `where rownum = 1`). Сортирующая фраза гарантирует, что эта первая строка будет сообщением на предпочтительном языке, если таковой имеется. Такой запрос несколько сложнее базового. Но вызывающая программа будет намного проще, поскольку ей больше не надо будет беспокоиться о запросе, не возвращающем ни одной строки. Окажется ли она быстрее? Неизвестно. Если у вас есть энергичная команда переводчиков, обеспечивающая максимально полный перевод всех сообщений, производительность с более сложным запросом, вероятно, будет немного ниже (разница вряд ли будет явно заметна). Однако если многие переводы отсутствуют, предполагаю, что один запрос будет работать быстрее.

Но, как в примере с `code_values`, реальной наградой не всегда будет то, что останется один запрос вместо другого (изредка двух): теперь, когда у нас есть один оператор, мы можем превратить этот оператор в подзапрос и непосредственно перенести в другой запрос. Это, в свою очередь, может дать нам возможность избавиться от циклов и обойтись правильными процессами SQL.

Изменение логики

Чтобы закончить обсуждение вариантов переноса логики в операторы SQL, упомяну также случаи, когда, даже если вам не удастся обойтись единственным оператором, небольшая модификация может принести значительное улучшение.

Предположим, у нас есть следующий код:

```
select from A where ...
проверка результата запроса
  if какое-то условие
    update table B
  else
    update table C
```

Если второй случай – обновление таблицы C – встречается значительно чаще, то что-либо подобное нижеприведенному псевдокоду окажется в среднем быстрее, если выполнение обновления, которое включает в себя первый запрос, будет производиться быстрее, чем выполнение запроса с последующим обновлением:

```
update table C where (includes condition on A)
if nothing is updated then
  update table B
```

Избавление от функции `count()`

В предыдущих примерах вы несколько раз видели случаи бесполезных вызовов функции `count()`. Количество разработчиков, критикующих функцию `count()`, бесконечно. Нет необходимости говорить, что если вам нужно опубликовать складской отчет, который сообщает о количестве товара на складе, функция `count()` нужна. Вообще говоря, использование функции `count()`, связанной с фразой `group by`, имеет под собой основания: в большинстве случаев использование функции `count()` является, как в одном из предыдущих примеров, проверкой существования.

Мы снова сталкиваемся с ситуацией «параноидального программирования», которое так любят процедурные разработчики, мало знакомые с SQL. Нет необходимости заблаговременно проверять, существуют ли данные. Когда вы хотите извлечь данные, просто сделайте это и обработайте событие, если данные не будут найдены. Если вы хотите обновить или удалить данные, тоже сделайте это, а затем сравните количество обработанных строк с тем количеством, которое вы ожидали. Если числа отличаются, придется разбираться. Но в 99% случаев, когда все работает правильно, нет необходимости исполнять два запроса вместо одного. Если вы исполняете два запроса для каждой транзакции, вы сможете выполнить на том же оборудовании только половину от количества транзакций, которые можно было бы выполнить с одним запросом.

В настоящий момент я могу назвать только один случай, имеющий приемлемое обоснование для предварительного подсчета: вывод общего количества обнаруженных строк при поиске, возвращающем много страниц с результатами. Если количество строк слишком велико, пользователь может сузить критерии поиска, а программа может отменить поиск. Это разумное требование обычно реализуется худшим из возможных способов: вместо `count(*)` вставляют список столбцов после `select` в реальном запросе, не обращая внимания на базовый запрос.

В пятой главе я упоминал случай, в котором функция `count()` могла бы работать в десять раз быстрее, если бы мы просто удалили два внешних соединения, бесполезных для ограничения результирующего набора. Я также упоминал, что помещение запроса в начало было бессмысленным. Это была целая история: проблема заключалась в том, что время от времени пользователь запускал (медленный) запрос, который возвращал около 200 000 строк, и это приводило к блокировкам по времени сервера приложений. Предварительный подсчет количества строк был предназначен для отмены выполнения запроса, если результирующий набор содержал более 5000 строк. В этом случае пользователю приходилось уточнять критерий поиска, чтобы получить более контролируемый результат.

Моя рекомендация состояла в том, чтобы ограничить размер результирующего набора 5001 строкой. Единственной трудностью было получение количества строк одновременно с извлечением строк без запуска дополнительного запроса. И MySQL, и Oracle, и SQL Server дают такую возможность, хотя делают это разными способами.

Например, в MySQL решение такое:

```
select ...  
limit 5001;
```

с последующим:

```
select found_rows();
```

И Oracle, и SQL Server позволяют нам возвращать общее количество строк в результирующем наборе в каждую строку с помощью конструкции `count(*) over ()`, которая получает данные и итог в одной операции. Кроме уменьшения количества обращений к базе данных, польза заключается еще и в том, что обычный оператор `count distinct` выполняется на сервере. Очень часто SQL-машине нет необходимости выполнять дополнительный подсчет: не нужно забывать, что когда возвращаемые данные отсортированы, что является обычным случаем, то перед возвращением самой первой строки результирующего набора SQL-машине приходится сравнивать ее с каждой другой строкой того же результирующего набора. Полный результирующий набор (а следовательно, и количество строк в нем) становится известен, когда вы получаете первую строку отсортированного результирующего набора.

Учитывая вышесказанное, в случае MySQL функция `found_rows()` не может быть вызвана раньше, чем вы извлечете данные из результирующего набора, который нужно оценить. Это нам не подходит. В крайнем случае, вы можете извлечь результирующий набор, сосчитать строки и получить результат без необходимости вызывать функцию `found_rows()`. Используя таблицу `transactions` (см. первую главу) и ограничив набор 500 строками, я попытался схитрить. Я выполнил оператор `select`, а затем вызвал функцию `found_rows()` перед извлечением результата из запроса в нижеприведенной программе *sample_bad.php*:

```
<?php  
require('config.php');  
/* Use prepared statements */  
$mycnt = 0;  
$stmt = $db->stmt_init();  
$stmt2 = $db->stmt_init();  
if ($stmt->prepare(„select accountid, txdate, amount“  
    . „ from (select accountid, txdate, amount“  
    . „ from transactions“  
    . „ where curr = ‚GBP‘“  
    . „ limit 501) x“  
    . „ order by accountid, txdate“))  
    && $stmt2->prepare(„select found_rows()“) {  
    $stmt->execute();  
    $stmt->bind_result($accountid, $txdate, $amount);  
    $stmt2->execute();  
    $stmt2->bind_result($cnt);  
    if ($stmt2->fetch()) {  
        printf(„cnt : %d\n“, $cnt);
```

```

    } else {
        printf(„cnt not found\n");
    }
    while ($stmt->fetch()) {
        $mycnt++;
    }
    $stmt->close();
    $stmt2->close();
    } else {
        echo $stmt->error;
    }
    $db->close();
?>

```

После запуска этой программы я получил следующий результат:

```

$ php sample_bad.php
cnt not found

```

Чтобы заставить функцию `found_rows()` что-нибудь вернуть, вам нужно извлечь данные (даже с модификатором `SQL_CALC_FOUND_ROWS`). Если вы хотите узнать количество возвращаемых строк до начала извлечения данных, вам придется пойти на еще большую хитрость. Вот что я сделал в программе *sample_ok.php*:

```

<?php
require('config.php');
/* создание подготовленного оператора */
$stmt = $db->stmt_init();
if ($stmt->prepare("select counter, accountid, txdate, amount"
    . " from ((select 1 k, counter, accountid, txdate, amount"
    . "         from (select 502 counter,"
    . "                 accountid,"
    . "                 txdate,"
    . "                 amount "
    . "         from (select accountid, txdate, amount"
    . "                 from transactions"
    . "                 where curr = 'GBP'"
    . "                 limit 501) a"
    . "         order by accountid, txdate) c"
    . "         union all"
    . "         (select 2 k, found_rows(), null, null, null)"
    . "         order by k) x)"
    . " order by counter, accountid, txdate")) {
    $stmt->execute();
    $stmt->bind_result($counter, $accountid, $txdate, $amount);
    // Извлечение первой строки, чтобы узнать количество строк в результи-
    рующем наборе
    if ($stmt->fetch()) {
        if ($counter == 501) {
            printf("Query returns more than 500 rows.\n");
        } else {
            while ($stmt->fetch()) {

```

```
        printf("%6d %15s %10.2f\n", $accountid, $txdate, $amount);
    }
}
} else {
    echo $stmt->error;
}
$stmt->close();
} else {
    echo $stmt->error;
}
$db->close();
?>
```

С помощью `union all` я соединил вызов функции `found_rows()` с исполнением запроса, от которого я хотел получить количество строк. Затем я добавил первую фразу `order by`, чтобы гарантировать, что результат функции `found_rows()` будет показан после данных, и убедиться, что он подсчитает количество строк в моем результирующем наборе. Затем я отсортировал результат в обратном порядке, написав мой запрос так, чтобы количество строк всегда возвращалось первым, а оставшаяся часть была отсортирована по моему желанию.

Разумеется, полный результирующий набор был извлечен на сервере. Если максимальное количество строк невелико, двойная сортировка не представляет проблемы. Большим преимуществом было то, что я узнавал, имело ли смысл извлекать оставшиеся данные, после извлечения даже одной строки. Это сильно упрощало клиентскую сторону моей программы.

Хотелось бы обратить ваше внимание на оптимизацию. При выводе на экран результатов поиска принято использовать фразу `order by`. Например, если вы напишете:

```
select ...
order by ...
limit 501
```

будет возвращена максимум 501 строка, но полный результирующий набор запроса (который был бы возвращен без ограничивающей фразы) должен быть отсортирован. Если вы напишете:

```
select *
from (select ...
      limit 501) as x
order by ...
```

то сначала получите 501 строку, возвращенную запросом, а затем отсортируете эти строки, что может быть достаточно быстрой операцией. Если срабатывает ограничивающая фраза, – то есть если запрос возвращает больше 501 строки, – запросы вернут различные результаты, и второй результат окажется бессмысленным. Но если ваша цель состоит в том, чтобы отбросить любой запрос, содержащий свыше 500 строк, то для вас не имеет значения, будет ли 501 возвращенная строка в числе первых или

нет. Если результирующий набор содержит не более 500 строк, результирующие наборы и скорости в обоих вариантах будут эквивалентными.

С SQL Server или Oracle задача намного проще, здесь вы можете использовать функцию `count()` с фразой `over()` и возвращать общее количество строк в каждой строке результирующего набора. В SQL Server запрос такой:

```
select accountid, txdate, amount, count(*) over () counter
from (select top 501
      accountid, txdate, amount
      from transactions
      where curr = 'GBP') as tx
order by accountid, txdate
```

Если этот запрос находит, скажем, 432 строки, он вернет число 432 в последнем столбце каждой возвращаемой строки.

Замечу еще раз, что я в первую очередь забочусь об ограничении количества возвращаемых строк и только затем о сортировке. Поскольку я знаю, что аннулирую запрос, если получу 501 строку, меня не беспокоит бессмысленность результата в этом случае.

Однако вопрос заключается в том, имеет ли смысл замена столь простого выражения, как `count(*)`, на одно сложное выражение, как в MySQL. Большинство людей уверены, что простой SQL быстрее, но ответ всегда должен быть обоснован.

Для объяснения я написал три простые программы на PHP на основе предыдущего примера MySQL. Я сделал цикл по всем 170 валютам в моей таблице `currencies`. Для каждой я извлекал транзакции, если их было меньше 500. В противном случае я просто выводил сообщение, что количество транзакций превышает 500. Чтобы обеспечить оптимальные условия теста, я проиндексировал таблицу `transactions` по столбцу `curr`, содержащему код валюты.

В первой программе, которая приведена ниже, есть два оператора, `$stmt` и `$stmt2`. Первый считает количество транзакций, а второй действительно извлекает данные, когда количество транзакций для валюты меньше 500, из моей таблицы, содержащей два миллиона строк:

```
<?php
require('config.php');
/* создание двух подготовленных операторов */
$stmt = $db->stmt_init();
$stmt2 = $db->stmt_init();
if ($stmt->prepare(„select count(*) from transactions where curr = ?“)
    && $stmt2->prepare(„select accountid, txdate, amount“
        . „ from transactions“
        . „ where curr = ?“
        . „ order by accountid, txdate“)) {
    if ($result = $db->query(„select iso from currencies“)) {
        while ($row = $result->fetch_row()) {
            /* связывание с оператором count */
```

```

        $curr = $row[0];
        $counter = 0;
        $stmt->bind_param('s', $curr);
        $stmt->execute();
        $stmt->bind_result($counter);
        while ($stmt->fetch()) {
        }
        if ($counter > 500) {
            printf("%s : more than 500 rows\n", $curr);
        } else {
            if ($counter > 0) {
                /* Fetch */
                $stmt2->bind_param('s', $curr);
                $stmt2->execute();
                $stmt2->bind_result($accountid, $txdate, $amount);
                /* мы просто посчитали */
                $counter2 = 0;
                while ($stmt2->fetch()) {
                    $counter2++;
                }
                printf("%s : %d rows\n", $curr, $counter2);
            } else {
                printf("%s : no transaction\n", $curr);
            }
        }
    }
}
$result->close();
}
}
$stmt->close();
$stmt2->close();
$db->close();
?>

```

Во второй программе я сделал только одно усовершенствование: я хотел выяснить, будет ли у меня менее или более 500 транзакций. С точки зрения моей программы наличие 501 транзакции значит то же, что и наличие 1 345 789 транзакций. Однако «невинная» SQL-машина этого не знает. Вы приказываете ей считать, и она честно считает — до самой последней транзакции. Поэтому мое усовершенствование заключается в замене

```

select count(*)
from transactions
where curr = ?

```

на

```

select count(*)
from (select 1
      from transactions
      where curr = ?
      limit 501) x

```

В третьей программе есть только такое «усложненное» выражение:

```
<?php
require('config.php');
require('sqlerror.php');
$stmt = $db->stmt_init();
if ($stmt->prepare("select counter, accountid, txdate, amount"
    . " from ((select 1 k, counter, accountid, txdate, amount"
    . "         from (select 502 counter,"
    . "                 accountid,"
    . "                 txdate,"
    . "                 amount "
    . "         from (select accountid, txdate, amount"
    . "                 from transactions"
    . "                 where curr = ?"
    . "                 limit 501) a"
    . "         order by accountid, txdate) c"
    . "         union all"
    . "         (select 2 k, found_rows(), null, null, null)"
    . "         order by k) x)"
    . " order by counter, accountid, txdate")) {
    if ($result = $db->query("select iso from currencies")) {
        while ($row = $result->fetch_row()) {
            /* связывание с оператором count */
            $curr = $row[0];
            $counter = 0;
            $stmt->bind_param('s', $curr);
            $stmt->execute();
            $stmt->bind_result($counter, $accountid, $txdate,
                $amount);

            if ($stmt->fetch()) {
                if ($counter > 500) {
                    printf("%s : more than 500 rows\n", $curr);
                    $stmt->reset();
                } else {
                    $counter2 = 0;
                    while ($stmt->fetch()) {
                        $counter2++;
                    }
                    printf("%s : %d rows\n", $curr, $counter2);
                }
            } else {
                printf("%s : no transaction\n", $curr);
            }
        }
        $result->close();
    }
}
$stmt->close();
$db->close();
?>
```

Я последовательно запускал каждую программу (по несколько раз) и засекал время. Результаты приведены на рис. 6.2. На моей машине среднее время обработки 170 валют с помощью первой программы и полный подсчет в сумме составили 7,38 секунд. Когда я ограничил подсчет тем, что мне было действительно нужно, время сократилось до 6,09 секунд. В случае единственного запроса время было 6,28 секунд.

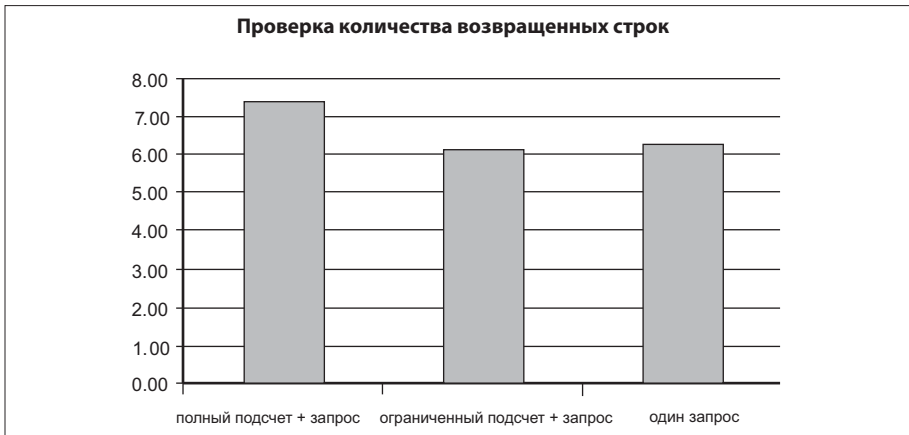


Рис. 6.2. Сравнение различных стратегий проверки количества возвращаемых строк до извлечения данных (MySQL/PHP)

Различие в секунду с небольшим при обработке 170 элементов я считаю внушительным. Можете представить разницу при повторении этих операций десятки тысяч раз – и подумайте о стоимости машины, на 15% более быстрой, чем ваша нынешняя. Несмотря на то что в третьей программе есть сложное выражение SQL, двукратная сортировка и множество других вещей, она работает вполне приемлемо, хотя и не настолько быстро, как вторая программа.

В этом случае идентификация результирующего набора оказалась очень быстрой. Чтобы проверить, что происходит, когда получение результирующего набора занимает больше времени, я запустил тесты с Oracle, используя аналитическую версию функции `count()`¹, возвращающую общее количество строк в каждой строке результирующего набора. Я запустил запрос с индексом по валютам, что дало мне «быструю фразу where», и без индекса по валютам, что дало мне «медленную фразу where». Как можно догадаться, время обработки отличалось более чем на порядок. Чтобы упростить сравнение, я принял время выполнения исходной программы (полный подсчет, затем запрос) за 100 и выразил остальные результаты в сравнении с этим числом. Полную картину вы можете увидеть на рис. 6.3. С быстрой фразой `where` результат отразил то, что я полу-

¹ То есть `count(*) over ()`, как в предыдущем примере для SQL Server.

чил в MySQL: в сущности, запрос так быстр, что накладные расходы, вызванные аналитической функцией, «съедают» преимущества единственного обращения к базе данных. Но в случае медленной фразы `where` баланс нарушается. Поскольку теперь требуется время для идентификации строк, поиск и подсчет их в единой операции дает очень значительный выигрыш.

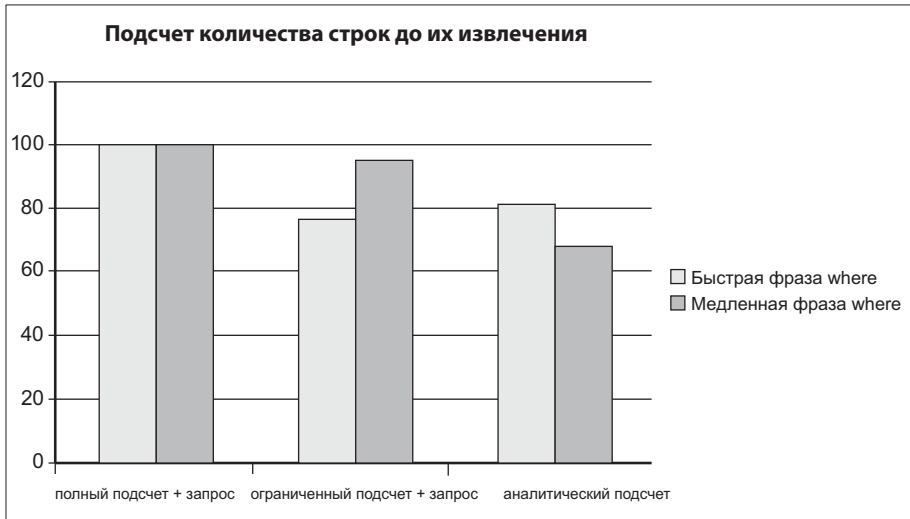


Рис. 6.3. Сравнение различных методов в зависимости от скорости идентификации результирующего набора (Oracle, PL/SQL)

Урок, который надо извлечь, заключается в том, что удаление предварительного подсчета путем переноса итога в запрос, реально возвращающий данные (с относительно элегантным использованием `count() over ()` или с чем-либо другим), приносит больше пользы, поскольку идентификация результирующего набора медленна и сложна. Поэтому этот прием при правильном использовании может принести очень значительное ускорение.

Случай, когда мы не готовы аннулировать результаты поиска, возвращающего очень большое количество строк, отличается от предыдущего не очень сильно, за исключением того, что вы не можете использовать методы оптимизации, которые состоят в уменьшении результирующего набора перед сортировкой. В этой ситуации вы не можете допустить бесполезной сортировки. Но в ином случае, даже если вы выполняете пакетную обработку с использованием обычного приема возвращения последовательных «страниц» данных, основные идеи остаются примерно теми же.

Избегайте излишеств

Выше я настаивал на важности уменьшения количества операторов SQL и на необходимости попытаться получить как можно больше от одного

оператора. Важно помнить, что получение одного огромного оператора SQL не является целью. Цель состоит в использовании минимально возможного количества операторов в одной ветви исполнения, а это несколько другое. Мне иногда встречаются операторы с подобными условиями:

```
where 1 = ?  
and ...
```

Нет смысла отправлять оператор SQL-машине, заставляя ее анализировать оператор и позволяя СУБД решать, исполнять оператор или нет, в зависимости от значения параметра. Решение о выполнении или невыполнении оператора должна принимать вызывающая программа. Интереснее те случаи, где ветвление зависит от результатов запросов, в которых есть конструкции, подобные следующей:

```
select c1, ...  
into var1, ...  
from T  
where ...  
if (var1 = ...)   
then  
    Новая операция с базой данных  
[else]  
end if;
```

В этой ситуации вы можете, в зависимости от природы условий, использовать различные приемы из тех, что я описывал выше.

Избавляйтесь от циклов

Избавление от некоторых операторов и передача работы с условиями в SQL во многих случаях полезны, но настоящий «рай для рефакторинга» заключается в удалении циклов, содержащих операторы SQL. В большинстве случаев эти операторы сами выполняют циклы по результирующим наборам запроса (циклы курсора). Удаление циклов увеличивает не только производительность, но и понятность, «читабельность» кода, что очень полезно для упрощения поддержки.

Если вы исполняете операторы в цикле, вы посылаете СУБД идентичные (за исключением отдельных параметров, которые могут меняться) операторы в быстрой последовательности. Вы множите переключения контекста между программой-оболочкой и SQL-машиной и даете оптимизатору слишком мало пищи для размышлений, а это заканчивается тем, что работа программы часто оказывается медленнее, чем могла бы быть, зачастую более чем на порядок.

Приведу очень простой пример: я использовал emp и dept, тестовые таблицы, описывающие сотрудников и подразделения, традиционно используемые в Oracle, для получения имен сотрудников и названий подразделений, где они работают. Сначала я использовал цикл по таблице emp, затем на основе полученного номера подразделения извлекал название подразделения. Затем я использовал прямое соединение. Я использовал

обычные тестовые таблицы с 14 и 4 строками. Единственным усовершенствованием было то, что я объявил `empno` первичным ключом таблицы `emp`, а `deptno` – первичным ключом таблицы `dept`, и собрал статистику по обеим таблицам. Чтобы увеличить различия, я запустил тест 20 000 раз. Перед вами конечный результат двух последовательных запусков:

```
ORACLE-SQL> declare
2     v_ename emp.ename%type;
3     v_dname dept.dname%type;
4     n_deptno dept.deptno%type;
5     cursor c is select ename, deptno
6                   from emp;
7 begin
8     for i in 1 .. 20000
9     loop
10         open c;
11         loop
12             fetch c into v_ename, n_deptno;
13             exit when c%notfound;
14             select dname
15                 into v_dname
16                 from dept
17                 where deptno = n_deptno;
18         end loop;
19         close c;
20     end loop;
21 end;
22 /
```

PL/SQL procedure successfully completed.

Elapsed: 00:00:27.46

```
ORACLE-SQL> declare
2     v_ename emp.ename%type;
3     v_dname dept.dname%type;
4     cursor c is select e.ename, d.dname
5                   from emp e
6                   inner join dept d
7                   on d.deptno = e.deptno;
8 begin
9     for i in 1 .. 20000
10    loop
11        open c;
12        loop
13            fetch c into v_ename, v_dname;
14            exit when c%notfound;
15        end loop;
16        close c;
17    end loop;
18 end;
19 /
```

```
PL/SQL procedure successfully completed.  
Elapsed: 00:00:09.01
```

Без курсорного цикла мой код работает в три раза быстрее. Однако у меня самые благоприятные условия: происходит поиск по первичному ключу, все находится в памяти, выполняется запуск процедуры, которая не является хранимой, но ведет себя как хранимая процедура (в СУБД передается единый блок PL/SQL). Мысль о том, что в случаях, когда таблицы несколько более объемны, а операции более просты, это «скромное» соотношение 3:1 станет значительно более впечатляющим, не кажется притянутой за уши.

Много раз попытки придумать вариант без циклов приводили к полному переосмыслению процессов и повышению производительности на несколько порядков. Рассмотрите, например, следующий шаблон, который встречается довольно часто, когда вас интересует самое последнее событие (в самом широком смысле слова), связанное с элементом:

```
loop  
  select ...  
  where date_col = (select max(date_col)  
                    from ...  
                    where <условие по чему-то, возвращенному циклом>  
                    and ....)
```

Подзапрос использует ту же «таблицу истории», что и внешний запрос, и зависит от того, что вернул цикл. Очень часто этот подзапрос также является согласованным и зависит от внешнего запроса. В результате количество его исполнений, вероятно, будет огромным. Если заменить целый цикл чем-нибудь подобным:

```
select ..  
from t1  
  inner join (select c1, max(date_col) date_col  
              from ...  
              where ...  
              group by c1) x  
  on x.c1 = t1....  
  and x.date_col = t1.date_col  
where ...
```

вы за один раз выполните вычисления для всех интересующих дат, все рассогласуете и придете к значительно более эффективному запросу, который ваш друг-оптимизатор обработает со скоростью света, поскольку теперь ему будет доступна вся картина.

Возможна только одна сложность, связанная с удалением циклов: иногда курсорные циклы оправданны. Поэтому прежде чем приниматься за рефакторинг программы, нужно попытаться понять, почему предыдущие разработчики использовали цикл. Возможно, это происходило потому, что они не знали, как сделать лучше. Но это мог быть и осознанный технический способ реализации стоящих перед ними задач.

Причины использования циклов

Разработчики исполняют операторы в циклах по одной из следующих причин:

Наличие логики If ... then ... else

Как вы только что видели, часто существует возможность вставить логику if ... then ... else внутри запросов. Наличие единственного запроса внутри цикла устраняет потребность в цикле, а целый блок обычно можно заменить на соединение запроса в курсоре с запросом внутри блока. Речь фактически идет о реальной пользе замены процедурной логики if ... then ... else на запрос SQL.

Необходимость изменить операции, которые последовательно влияют на несколько таблиц

Иногда данные считывают из временной таблицы, загруженной из потока XML, а затем вставляют в несколько таблиц в процессе сохранения иерархической структуры XML в чисто реляционную структуру.

Ситуация, когда разработчик хочет, чтобы изменения регулярно фиксировались внутри цикла, имея в виду одну или несколько целей

Освобождение блокировок

Блокировки, в зависимости от их жесткости, могут значительно ухудшать параллелизм. Если какое-либо незафиксированное изменение блокирует либо пользователей, которые хотят подсчитать измененные данные, либо пользователей, которые хотят изменить какие-либо другие данные в той же таблице, блокировки нужно освобождать как можно быстрее. Единственным способом освободить блокировку является окончание транзакции либо путем фиксации, либо путем отката изменений. При выполнении единого длительного обновления не существует способа фиксации до окончания обновления. Обработка данных небольшими порциями внутри цикла обеспечивает лучший контроль над блокировками.

Не генерировать слишком много откатов

Вы, вероятно, знакомы с тем фактом, что если вы делаете обработку в транзакционном режиме, то каждый раз, когда вы изменяете какие-то данные, СУБД сохраняет исходное значение. На этом основывается возможность отмены изменений в любой момент и возвращения к состоянию, про которое вы знаете, что данные в нем согласованы. На практике это означает, что вам необходимо некоторое дисковое пространство для хранения «истории отката» до того момента, пока вы не зафиксируете изменения и не сделаете их окончательными. В зависимости от продукта информация для отката будет записываться в журнал транзакции или в выделенную область хранилища. При массированном обновлении потребности в пространстве могут быть большими:

некоторые «долгоиграющие» транзакции могут приводить к исчерпанию свободного пространства. Гораздо хуже то, что когда длительная транзакция оканчивается неудачей, возврат к прежнему состоянию занимает примерно столько же времени, сколько ее выполнение, делая не самую простую ситуацию еще хуже. А после сбоя всей системы это может привести к удлинению периода недоступности, поскольку когда база данных запускается после сбоя, она должна откатить все незаконченные транзакции, прежде чем пользователи смогут нормально работать. Это еще один случай, который служит мотивом для обработки данных небольшими частями с регулярной фиксацией изменений.

Обработка ошибок

Если вы выполняете массированное обновление данных (например, если вы обновляете данные из временных таблиц), одного неправильного фрагмента данных (например, нарушающих ограничения целостности данных) достаточно для неудачного завершения всего обновления. В такой ситуации вам нужно сначала идентифицировать фрагмент данных, приведший к ошибке, что не всегда просто сделать, а после исправления данных заново запустить весь процесс. Многие разработчики предпочитают последовательную обработку небольших частей: то, что благополучно обновлено, можно зафиксировать сразу после того, как вы убедитесь, что данные загружены правильно. Неправильные данные при этом можно идентифицировать сразу же после того, как они обнаружены.

Пока мы говорим о транзакциях, я должен заметить, что операторы фиксации значительно сильнее влияют на производительность, чем думают многие разработчики. По окончании вызова, выполняющего фиксацию, вы получаете неявную гарантию того, что ваши изменения записаны и будут сохранены вне зависимости от дальнейших событий. СУБД, возвращая управление вашей программе после успешной фиксации, «торжественно клянется», что изменения останутся на месте, даже если аварийное завершение работы системы произойдет через 0,00001 секунд после фиксации. Существует только один способ гарантировать сохранность обновлений: записать их диск. Операторы `commit` резко меняют темп процесса. Просто представьте себе бешеную активность в памяти – программы сканируют страницы, перескакивая от страницы индекса к страницам соответствующей таблицы, изменяя данные тут и там, иногда ожидая, пока страницы будут загружены в память, чтобы их сканировать или модифицировать. В это же время в фоновом режиме происходит асинхронная запись в файлы базы данных некоторых измененных страниц. Внезапно ваша программа выполняет фиксацию изменений. Операции ввода-вывода становятся синхронными. Для вашей программы все замедляется до тех пор, пока все не будет должным образом записано в файл журнала, что обеспечивает целостность вашей базы данных. По окончании вызова ваша программа снова получает свободу.

Вывод таков: если вы выполняете фиксацию слишком часто, программа будет проводить много времени в состоянии ожидания ввода-вывода. Если вы выполняете фиксацию недостаточно часто, программа может заблокировать несколько параллельных транзакций (или оказаться заблокированной столь же «эгоистичной» программой). Имейте это в виду при анализе содержимого цикла.

Анализ циклов

Когда вы имеете дело с обработкой SQL внутри цикла, вам обязательно нужно задать себе следующие вопросы, поскольку ненужные циклы встречаются часто:

Только ли операторы select имеются внутри цикла или же есть также операторы insert, update или delete?

Если у вас есть только операторы select, забот о блокировках и транзакциях у вас значительно меньше, если только программа не запущена на высоком уровне изоляции¹ (я расскажу об уровнях изоляции в следующей главе). Если внутри цикла в запутанной логике if ... then ... else есть ссылки на много различных таблиц, от циклов, вероятно, будет трудно избавиться. Иногда дизайн базы данных попадает в «интересную» категорию и написание чистого SQL оказывается невозможным, но если вы сумеете уменьшить внутреннюю цикла до одного оператора, то сможете избавиться от цикла.

Если у нас есть операторы insert, вставляем ли мы данные во временные таблицы или в конечные таблицы?

Некоторые разработчики питают слабость к временным таблицам, которые я редко считаю оправданными, хотя иногда они находят применение. Многие разработчики, кажется, игнорируют insert ... select Я регулярно сталкиваюсь со случаями, когда сложный процесс с промежуточным заполнением временных таблиц можно было бы заменить более зрелым кодом SQL и вставить результат непосредственно в запрос. В подобных случаях всегда достигается заметное увеличение производительности.

Если содержимое базы данных внутри цикла изменяется, фиксируются ли изменения внутри или вне цикла?

Проверка участка, где происходит фиксация, должна быть первым действием при просмотре кода, поскольку управление транзакциями является важнейшим из обоснований выполнения операторов SQL внутри циклов. Если внутри цикла нет фиксации, аргументация рассыпается: вместо улучшения параллелизма вы снижаете конкуренцию, поскольку есть вероятность, что ваш цикл окажется медленнее, чем один оператор, и тогда ваши транзакции будут дольше удерживать блокировку.

¹ Это в большей степени относится к SQL Server и MySQL, чем к Oracle.

Если внутри цикла нет оператора `commit`, вы можете выбрать одно из двух противоположных действий: сделать фиксацию внутри цикла или попытаться избавиться от цикла. Если вы наблюдаете серьезные проблемы с блокировкой, вам, вероятно, следует осуществлять фиксацию внутри цикла. В противном случае лучше попытаться избавиться от цикла.

Сомнительные циклы

Как было сказано выше, во многих случаях можно упростить сложную процедурную логику и перенести ее в операторы SQL. Когда вы достигаете этапа, на котором внутри курсорного цикла нет ничего, кроме присвоения значения переменной и одного запроса, это обычно означает победу – до замены целого цикла единственным запросом остается совсем немного. Но даже тогда, когда в процесс вовлечены операции, изменяющие содержимое базы данных, иногда можно изменить ход цикла:

Последовательные операции изменения

Случай множественных операций изменения, последовательно применяемых к нескольким таблицам (например, вставка данных, возвращенных курсорным циклом, в две различных таблицы из одного источника), является более сложным. Во-первых, похоже, слишком мало людей знает, что иногда возможны операции с несколькими таблицами (в Oracle оператор `insert` для нескольких таблиц появился в девятой версии). Но даже без них вы можете выяснить, есть ли возможность заменить цикл последовательными операторами – например, заменить это:

```
loop on select from table A
  insert data into table B;
  insert data into table C;
```

на это:

```
insert into table B
select data from table A;
insert into table C
select data from table A;
```

Даже если на уровне ссылок между таблицами B и C есть ограничения целостности, скорее всего, это будет работать отлично (сложности могут возникнуть в случае, если ссылочные ограничения чрезмерно сложны. Конечно, производительность будет зависеть от скорости запроса (или запросов, поскольку запросы не обязательно должны быть полностью идентичными), возвращающего данные из таблицы A. Если он безнадежно медленный, то, с учетом всех обстоятельств, цикл может оказаться быстрее. Однако если чтение пятой главы принесло вам какую-то пользу и вы научились ускорять запросы, два последовательных оператора `insert`, вероятно, окажутся быстрее, особенно если результирующий набор, по которому мы выполняем цикл, содержит много строк.

Вы, несомненно, заметили, что я отступил от одного из собственных базовых принципов – не обращаться к одним и тем же данным повторно. Однако я заменил его другим принципом: делать в моей программе то, что может быть сделано внутри SQL-машины, бессмысленно, даже если программа является хранимой процедурой, выполняемой на сервере.

Генерирование отката

Необходимость генерирования отката является другим разумным поводом. Я редко встречал случаи, когда частота фиксаций изменений была определена на основе изучения того, сколько откатов было сгенерировано, сколько времени занимал весь процесс отката транзакции в случае сбоя как раз перед фиксацией, или на основе сравнения различных частот. Слишком часто количество операций между фиксациями оказывалось случайным числом между 100 и 5 000.

Когда вы вставляете данные, у СУБД нет необходимости записывать большое количество информации для отката на случай сбоя (даже если вы вставляете очень много данных). Я видел, как СУБД ежедневно «заглатывает» восемь миллионов строк, вставляемых единственным оператором `insert ... select ...` без «икоты», даже несмотря на то, что оператор исполнялся в течение часа (что значительно быстрее, чем альтернативный цикл). То, насколько быстро вы сможете сделать откат в случае сбоя, зависит от того, вставляете ли вы данные в конец таблицы (в этом случае отбросить все, что добавил ваш процесс, будет легко), или же СУБД будет использовать ваши данные для заполнения «дыр», оставшихся от предыдущих удалений.

Когда вы удаляете данные, дело обстоит по-другому. В процессе удаления вам, вероятно, придется сохранить большой объем данных, и откат в случае сбоя на практике означает вставку всех этих данных обратно. Но если вам действительно приходится удалять большое количество данных, `truncate`, вероятно, является именно той командой, которая вам нужна. Если невозможно очистить всю таблицу или раздел, вам стоит предусмотреть сохранение нужной информации во временную (на этот раз) таблицу, очистку и последующую вставку, что в некоторых случаях может значительно ускорить процессы.

Обновления можно рассматривать как промежуточный вариант. Попробуйте различные частоты фиксации. Попросите администраторов базы данных отслеживать, сколько откатов генерируют ваши транзакции. Возможно, вы в конце концов обнаружите, что нет никаких реальных технических ограничений, заставляющих вас выполнять фиксацию изменений часто, и что некоторые циклы можно удалить.

Транзакции

Самыми сильными аргументами против некоторых типов переработки, таких как использование нескольких операторов `insert ... select ...`, является обработка транзакций и ошибочных ситуаций.

В предыдущем примере цикла:

```
loop on select from table A
  insert data into table B;
  insert data into table C;
```

я могу фиксировать изменения либо по окончании вставки данных в таблицы B и C, либо после каждой итерации, либо после любого произвольного количества итераций. С двумя операторами `insert ... select ...` есть вероятность, что единственный момент, когда я смогу фиксировать изменения, появится только по окончании работы обоих операторов `insert`.

Я меня нет четкого и простого возражения на эти очень важные аргументы¹. Все очень часто зависит от дополнительных подробностей.

Когда пакетные процессы иницируются, например, получением файла, я обычно делаю выбор в пользу скорости, удаляю все циклы, какие можно, и как можно меньше фиксирую изменения. Если что-нибудь происходит не так, у нас по-прежнему есть исходный файл (предполагается, что источник данных только один), и исправление и перезапуск будут означать всего лишь задержку. Поскольку очень часто программы без использования циклов работают значительно быстрее, чем процедурные программы, последовательность «запуск–сбой–исправление–перезапуск» займет значительно меньше времени, чем программа, которая содержит циклы и часто осуществляет фиксацию. Небольшой риск при выполнении допустим, если нет риска для данных.

Когда источником данных является сетевой поток, все зависит от того, работаем ли мы в режиме «отменить и заменить» или же все должно отслеживаться. Если все должно отслеживаться, регулярные операторы `commit` необходимы. Как обычно, все зависит от вашей готовности платить за время и инфраструктуру при таком уровне риска. Таким образом, решения лежат в области эксплуатации и финансов, а не техники.

Я не уверен, что вопрос блокировок является самым важным при выборе между процедурным и более реляционным подходами. Освобождение блокировок на регулярной основе – не очень правильная стратегия в случае массированных обновлений. Когда вы выполняете оператор `commit` внутри цикла, вы всего лишь освобождаете блокировки, чтобы захватить другие блокировки в следующей итерации. Вы можете оказаться в одной из двух ситуаций:

- Если что-то случится, ваша программа будет работать автономно, блокировки станут проблемой и вы сможете сосредоточиться на достижении максимальной производительности.

¹ Тем не менее я могу указать на пакетную обработку ошибочных ситуаций Oracle.

Лучшего способа увеличения производительности, чем устранение как можно большего количества процедурных элементов, не существует.

- Ваша программа работает одновременно с другой программой, которая пытается захватить ту же самую блокировку. Если ваша программа захватывает блокировку на краткий период и отпускает ее, вам ничего не остается делать, как позволить конкурирующему процессу захватывать блокировку в очень короткие промежутки времени, когда ваша программа не владеет ею, в результате чего заблокированной оказывается ваша программа. Конкурирующая программа может оказаться не столь «благородной», как ваша, и держать блокировку долго. Вся концепция конкуренции основана на идее, что программам не нужен один и тот же ресурс в одно и то же время. Если ваша программа заблокирована, когда работает другая программа, то лучше заблокировать нужные ресурсы раз и навсегда, в результате чего вы минимизируете общее время блокирования ресурсов, а по окончании работы программы освободите ресурсы для другой программы. Иногда последовательное исполнение значительно быстрее, чем параллельное.

Случай блокировки строк является несколько более сложным, и успешное решение будет зависеть от того, как различные программы получают доступ к строкам. Но мы уже коснулись вопросов, которые относятся к потоку операций, теме следующей главы.

7

Рефакторинг потоков и баз данных

*Отчаянный недуг
Врачуют лишь отчаянные средства
Иль никакие.*

Уильям Шекспир (1564–1616)
Гамлет, акт IV, сцена 3¹

До настоящего момента я обращал внимание в первую очередь на усовершенствование кода и удачные варианты использования СУБД с «единичной» точки зрения. Единичной не в смысле «одной строки», а в смысле «одного экземпляра программы». Программы редко имеют эксклюзивный доступ к базе данных (возможно, за исключением некоторых важных пакетных программ). Значительно чаще несколько экземпляров одной и той же программы или различных программ конкурируют между собой при осуществлении запросов и обновлении данных. Для совместной работы программ нужно обеспечить, чтобы каждая из них была эффективной и достаточно простой, тонкой в смысле времени и бюджета. Вы никогда не соберете хороший оркестр из плохих музыкантов, часто даже одного плохого музыканта достаточно, чтобы разрушить усилия всего оркестра. Собрать вместе виртуозов – необходимая часть дела, но этого недостаточно. Особое значение имеют вопросы масштабируемости, и обеспечить здесь гармонию важнее, чем увеличить производительность процессов, работающих автономно, поскольку в этом случае появится больше точек соприкосновения. В предыдущих главах я уже упоминал вопрос о конкуренции и блокировках, но я мало писал о среде с интенсивной конкуренцией между пользователями. В такой среде политика невмешательства часто приводит к тому, что в «узких местах»

¹ Пер. М. Л. Лозинского.

появляются «заторы», и когда конкуренция сильно снижает производительность, вам приходится «регулировать потоки».

В свою очередь, для уменьшения конкуренции требуется внести изменения либо в физическую, либо в логическую структуру базы данных. Одним из главных факторов, определяющих производительность, несомненно, является дизайн базы данных. Но рефакторинг – это деятельность, основанная на идее «последним вошел – первым вышел»: вы начинаете с того, что было сделано в процессе разработки последним, а заканчиваете тем, что делалось в начале. К сожалению, к тому моменту, когда вас приглашают для рефакторинга, базы данных эксплуатируются уже давно и в них содержится много данных. Реорганизация – не всегда подходящий вариант, даже когда логическая структура остается нетронутой, просто потому, что вы не можете позволить себе приостановку деятельности на перенос данных, перестройку индексов и отключение ограничений с последующей их активизацией (это может занимать дни), или потому, что руководство не хочет рисковать. Еще хуже, когда физические изменения происходят одновременно с логическими, что требует изменения программ. При строительстве дома всегда лучше предусмотреть подземный гараж, тем более, что стоимость такого гаража после окончания строительства будет одинаковой как для одноэтажного дома, так и для небоскреба. Иногда снос и постройка «с нуля» может оказаться дешевле. С базами данных такая простая операция, как добавление к большой таблице нового столбца со значением по умолчанию, может занимать часы, поскольку данные организованы построчно (по крайней мере в самых известных СУБД) и добавление данных к каждой строке требует переноса огромного объема данных¹.

С учетом сказанного выше, в случае, когда нет времени для организации процесса, пересмотр организации базы данных может оказаться вашей последней картой. В этой главе будут затронуты темы конкурентного доступа и физической реорганизации.

Реорганизация обработки

В первой главе я советовал вам попробовать отслеживать запросы, которые исполняет СУБД. Обнаруженное может вас удивить, и вы можете увидеть, что запросы, которые были совершенно безвредными, когда выполнялись по отдельности, в совокупности оказываются совсем не такими. Например, вы можете обнаружить дорогие жестко закодированные запросы, которые исполняются несколько раз в различных сеансах за короткий промежуток времени и вряд ли возвращают каждый раз разные результаты. Или вы можете обнаружить, что приложение загружает (с помощью последовательности операторов `select * from ...`, без всякой фразы `where`) всю таблицу, хранящую данные сотрудников, и всю таблицу

¹ Что подобно стихийному бедствию.

со ссылками на задачи каждый раз, когда кто-нибудь подключается, чтобы отчитаться о проделанной за неделю работе¹. Приложение не спрашивает вас, действительно ли запрос должен быть запущен такое количество раз, а если да, то можно ли хранить результирующий набор в какой-то временной области, чтобы дорогой запрос исполнялся только в первом сеансе, а остальные сеансы извлекали результат из кэша. Или вместо загрузки всех необходимых данных одновременно было бы лучше ограничить загрузку данными, которые требуются всегда, и внедрить функцию загрузки по требованию. Люди, которые пишут функциональные спецификации, обычно думают только об одном экземпляре программы. Увеличивая количество вопросов и предлагая альтернативные возможности людям, которые отлично знают ее функциональную сторону, вы не обязательно найдете подходящее решение сами, но, возможно, дадите толчок обсуждению, из которого почерпнете много полезных идей.

Борьба за ресурсы

Если вы имеете дело с базами данных и конкуренцией, лучшей моделью является один или несколько поставщиков услуг и какое-то количество клиентов, пытающихся получить услугу, как показано на рис. 7.1. Проблема в том, что во время обслуживания одного клиента остальные вынуждены ждать.

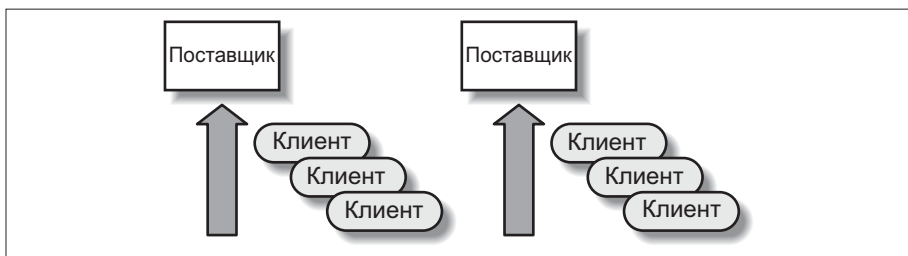


Рис. 7.1. Как должна выглядеть СУБД для вас

Эта модель прекрасно применяется в следующих случаях:

- низкоуровневые ресурсы, такие как использование процессора, операции ввода-вывода или области памяти, куда вы хотите записывать данные или считывать их;
- такие высокоуровневые ресурсы, как заблокированные строки или таблицы.

Модель поставщиков услуг, обслуживающих клиентов, которые выстраиваются в очередь, возникла из широко известной «теории очередей»,

¹ Реальная история. В следующей версии эта проблема была устранена не добавлением фразы `where` (размечтались!), а путем использования локального кэша на компьютерах пользователей.

считающейся одной из основ программирования. Она математически описывает ситуации, постоянно происходящие в реальной жизни. Если вам приходится ждать освобождения процессора и вы уверены, что вам потребуется большинство циклов процессора, нет сомнений в необходимости обновлять машину. Но часто, когда накапливаются очереди клиентов, ожидающих одного сервиса, другие поставщики услуг на той же машине простаивают. Например, такое может произойти, если ваша машина проводит много времени в ожидании окончания операций ввода-вывода. Сложность заключается в необходимости регулирования потоков и распределения нагрузки между гетерогенными ресурсами.

Время обслуживания и интенсивность входного потока

Рассмотрим случай, когда есть только один поставщик услуг. В системе, основанной на концепции поставщиков услуг и клиентов, действительно важны два значения: насколько долго поставщик услуг выполняет свою работу (время обслуживания) и как часто приходят клиенты (интенсивность входного потока). Для нас важнее всего соотношение между этими двумя значениями.

Когда интенсивность входного потока, даже на короткий период, намного превышает время обслуживания, мы получаем удлиняющуюся очередь клиентов, ожидающих, когда их обслужат.

Посмотрите, например, что происходит в пункте обмена валюты, где работает только один кассир. Если люди, желающие обменять валюту, приходят редко и у кассира хватает времени обслужить одного клиента до прихода другого, все относительно хорошо. Проблемы начинаются тогда, когда клиенты приходят быстрее, чем кассир успевает их обслуживать: в результате образуется очередь. Представьте себе, что будет, если эта касса расположена в аэропорту, где только что приземлился аэробус А-380...

Способов увеличить пропускную способность такой системы немного.

На месте поставщика услуг вы можете предпринять одно из следующих действий:

- попытаться обслуживать клиентов быстрее, совершая прежние действия в ускоренном темпе;
- нанять еще нескольких сотрудников, которые будут выполнять те же функции. Этот метод работает отлично, если между сотрудниками нет взаимодействия – например, если у каждого сотрудника есть пачка банкнот в различных валютах и он может обменивать деньги независимо от остальных. Но если всем сотрудникам потребуется личный доступ к общему ресурсу, этот метод будет работать очень плохо – представим, что сотрудники обязаны делать ксерокопии паспорта, а копировальный аппарат только один. В этом случае «узким местом» окажется доступ к копировальному аппарату. Клиенты будут думать, что они ждут сотрудника, хотя в действительности они будут ждать освобождения копировального аппарата. Однако наличие большего количества сотрудников может быть полезным в том случае, когда

действия сотрудников будут организованными: если один сотрудник делает ксерокопии паспортов, другой собирает паспорта у новых клиентов, третий считает банкноты, а четвертый заполняет квитанции, общая производительность будет значительно выше, чем при наличии только одного сотрудника. Тем не менее возможности увеличения производительности скоро будут исчерпаны, и добавление новых сотрудников будет всего лишь означать большее количество сотрудников, скользящих в очереди к копировальному аппарату.

На месте клиента услуг вы можете предпринять одно из таких действий:

- лучше объяснить свои потребности, чтобы получить ответ быстрее;
- попытаться организовать свою деятельность так, чтобы избегать длинных очередей. Например, если вы ведете своих детей в парк развлечений и обнаруживаете огромную очередь на один из аттракционов, вы можете пойти куда-нибудь еще и вернуться, когда очередь уменьшится.

Если клиентом является запрос SQL, достижение более быстрого отклика на тот же запрос требует настройки, а если настройка не дает нужного результата, то более быстрого оборудования. Наиболее удачное формулирование запроса (и избавление от ненужных запросов, например, итогов) было темой двух предыдущих глав. Два других направления – увеличение параллелизма и попытки распределить нагрузки в основном во времени, а не в пространстве, – могут быть применены и в рефакторинге. К сожалению, при таких сценариях у вас обычно значительно меньше пространства, чем при рефакторинге запросов или даже процессов. Редко бывает так, что не удастся ускорить медленный запрос или процесс. Но не каждый процесс можно распараллелить: у девяти беременных женщин не получится родить одного ребенка через месяц.

В сущности, увеличение производительности в результате внедрения параллельных процессов происходит не из-за разделения одной задачи между несколькими процессами, а скорее от наложения нескольких сходных задач. Вернувшись в Испанию в 1522 году после кругосветного путешествия, соратники Магеллана рассказали, что правитель одного из Молуккских островов имел 600 детей. Для того чтобы стать их отцом, ему не потребовалось 5400 месяцев (то есть 450 лет), достаточно было иметь 200 жен.

Усиление параллелизма

Есть несколько способов (прямых или косвенных) увеличить параллелизм: если программа выполняет простой процесс, запуск большего количества экземпляров этой программы может быть способом увеличения параллелизма (пример приведен ниже); увеличение количества процессоров на компьютере, на котором установлен сервер базы данных или сервер приложений, служит той же цели, но другим способом. Если несколько рычагов при определенных условиях могут помочь вам

увеличить производительность, то достичь этого можно, потянув за разные «ниточки»:

- если вы запускаете больше экземпляров одной и той же программы на одном и том же компьютере, то делаете это обычно потому, что низкоуровневые ресурсы (процессор, операции ввода-вывода) не загружены и вы надеетесь эффективнее использовать ваш компьютер, распределяя ресурсы между несколькими экземплярами программы. Если вы обнаруживаете, что одна из ваших программ проводит много времени в ожидании ввода-вывода и для уменьшения этого времени ничего нельзя сделать, имеет смысл запустить один или несколько других экземпляров программы, которые будут в это время использовать процессор;
- если вы увеличиваете количество процессоров или каналов ввода-вывода, это обычно означает (или должно означать), что по крайней мере один из низкоуровневых ресурсов перегружен. Ваша задача состоит прежде всего в том, чтобы решить локальные проблемы.

Вам нужно быть очень внимательно относиться к тому, как различные экземпляры получают доступ к таблицам и строкам, и убедиться, что никакая часть работы не выполняется по несколько раз несколькими конкурирующими экземплярами программы, которые ничего не знают друг о друге, и что необходимая координация программ не приведет к результату, напоминающему канкан, в котором все синхронно взмахивают ногами. Если все экземпляры программы одновременно ждут ввода-вывода, параллельное исполнение только ухудшит производительность.

Одним из условий, необходимых для эффективности параллельной работы, являются быстрые и успешные транзакции, которые монополизируют неразделяемые ресурсы на минимально возможное время. Я упоминал в предыдущей главе, что замена процедурного процесса на единственный запрос SQL не только уменьшает количество обменов между приложением и сервером, но и передает оптимизатору базы данных задачу наиболее быстрого выполнения запроса. Вы видели в пятой главе, что существует фактически два способа соединить две таблицы: с помощью вложенных циклов и с помощью хеш-соединений¹. Вложенные циклы по своей природе являются последовательными и похожи на то, что вы делаете при использовании курсорных циклов. Хеш-соединения требуют некоторых подготовительных шагов, которые иногда можно выполнить в параллельных потоках. Конечно, выбор варианта, который в конце концов сделает оптимизатор, будет зависеть от многих факторов, в том числе от текущей версии СУБД. Но, давая оптимизатору некоторый контроль над процессом, вы оставляете все варианты открытыми и санкционируете параллельную работу, но запрещаете код, если он изобилует циклами.

¹ Соединения слиянием по духу очень близки хеш-соединениям.

Размножение поставщиков услуг на уровне приложения

Я придумал простой контрольный пример, показывающий, как вы можете попытаться распараллелить обработку, увеличив количество «поставщиков услуг» внутри приложения. Для этого примера я использовал только MySQL (поскольку в ней разрешены механизмы хранения с различными минимальными единицами блокировки) и создал таблицу MyISAM. Поскольку к таблицам MyISAM нельзя применить блокировку на уровне строк, это можно считать худшим сценарием.

Во многих системах вы можете встретить таблицы, предназначенные в первую очередь для взаимодействия между процессами: какие-то «процессы-издатели» записывают в таблицу задачи, которые должны быть исполнены, а «процессы-исполнители» читают эту таблицу, выполняющую роль списка задач, который функционировать по принципу «первым вошел – первым вышел», выполняют задачу и (обычно после исполнения задачи или сбоя при выполнении) записывают в таблицу код окончания. Я частично эмулировал такой механизм, сосредоточив внимание исключительно на стороне «исполнителей». Я создал таблицу под названием `fifo` примерно такого вида:

```
mysql> desc fifo;
```

Field	Type	Null	Key	Default	Extra
seqnum	bigint(20) unsigned	NO	PRI	NULL	auto_increment
producer_pid	bigint(20)	NO			
produced	time	NO			
state	char(1)	NO			
consumed	time	YES		NULL	
counter	int(11)	NO			

6 rows in set (0.00 sec)

Я вставил в эту таблицу 300 строк с состоянием R (*Ожидание*) и значением 0 в столбце `counter`. Исполнители получают значение столбца `seqnum` для самой старой строки в состоянии R. Если ничего не найдено, программа заканчивает работу. В реальных условиях при постоянно активных процессах-издателях через несколько секунд таблица будет вызвана снова. Затем исполнитель обновляет строку и устанавливает значение в столбце `state` равным P (*Обработка*), используя уже найденное значение `seqnum`. Я предпочел бы выполнять единственный оператор `update`, но, поскольку мне нужно значение `seqnum` для обновления строки по окончании выполнения задачи, приходится исполнять два запроса¹. Затем я эмулирую некоторую обработку, посылая программе команду «заснуть» на случайное число секунд (в среднем – на две секунды). Затем программа обновляет строку, устанавливая в столбце `state`

¹ Обратите внимание, что в Oracle и некоторых интерфейсах есть возможность выполнять обновление и возвращать значение за один прием.

значение `D` (*Закончено*), вставляет метку времени окончания и увеличивает счетчик на единицу.

Стоит добавить, что у таблицы `fifo` есть только один индекс первичного ключа по столбцу `seqnum`. В частности, нет индекса по столбцу `state`, а это означает, что поиск самой старой необработанной строки превращается в сканирование таблицы. Такое индексирование в данном случае неудивительно. Обычно таблицы, используемые для обмена сообщениями между процессами, в каждый конкретный момент времени содержат очень мало строк: если система настроена правильно, издатели и исполнители работают приблизительно с одинаковой скоростью, а программа «уборки мусора» регулярно удаляет обработанные строки. Индексирование столбца `state` приведет к дополнительным накладным расходам на обслуживание индекса, когда издатели вставляют строки, исполнители изменяют содержимое столбца или происходит очистка строк. Без интенсивного тестирования трудно определить, принесет ли пользу индекс по столбцу `state`. В данном случае этого индекса нет.

Первая версия программы (*consumer_naive.c*) выполняет следующие операторы SQL в цикле:

```
select min(seqnum) from fifo
where state = 'R';
update fifo
set state = 'P'
where seqnum = ?; /* Использует значение, возвращаемое первым запросом */
```

...маленький намек...

```
update fifo
set state = 'D', consumed = curtime(), counter = counter + 1
where seqnum = ?;
```

В программе по умолчанию используется режим автоматической фиксации изменений MySQL C API, при этом каждый оператор `update` неявно превращается в транзакцию, которая фиксируется сразу после ее выполнения.

При среднем времени обработки строки около двух секунд, на обработку 300 строк таблицы `fifo` требуется около десяти минут. Могу ли я увеличить скорость путем увеличения количества исполнителей, чтобы обработка проводилась параллельно? Это был бы способ решения стандартной проблемы в архитектуре такого типа, поскольку издатели редко добавляют строки с равномерной скоростью. Бывают всплески активности, и если у вас недостаточно исполнителей, задачи накапливаются. Как показывает этот пример, избавление от очереди может занять некоторое время.

Когда я запускаю двух исполнителей, для обработки всех задач требуется пять минут, при пяти исполнителях – две минуты, при десяти – одна, при двадцати – около тридцати пяти секунд, а с пятьюдесятью процессами – около пятнадцати секунд. Есть только одна загвоздка: когда

я проверяю значение счетчика, обнаруживается некоторое количество строк, для которых значение больше единицы. Это означает, что строка была обновлена более чем одним процессом (вспомните, ведь процесс по окончании своей работы увеличивает значение счетчика на единицу). Я запускал тест пять раз с различным количеством параллельных исполнителей. На рис. 7.2 показаны средние результаты.



Рис. 7.2. Когда конкуренция приводит к печальным последствиям

Многократное выполнение задач явно не является желательным результатом. Если я увижу в банковской выписке, что одно и то же списание с моего счета было сделано дважды, я вряд ли обрадуюсь, как не обрадуюсь, узнав, что мой билет на концерт был продан еще двум или трем людям.

Параллельная работа требует определенной синхронизации, в SQL для этого служат два элемента:

- уровень изоляции;
- блокировка.

Уровень изоляции определяет информированность сеанса о том, что делают другие конкурирующие сеансы. Стандарт ISO, который более или менее поддерживают все продукты, определяет четыре уровня, из которых наибольший практический интерес представляют два:

Read uncommitted (уровень чтения незафиксированных данных)

На этом уровне все сеансы немедленно извещаются о том, что делают другие. Когда один сеанс модифицирует какие-либо данные, все остальные сеансы видят изменения вне зависимости от того,

зафиксированы они или нет. Это чрезвычайно опасная ситуация, поскольку транзакция может опираться в своих вычислениях на изменения, которые потом окажутся отмененными путем отката транзакции. Это легко может привести к несоответствиям данных, которые потом будет очень сложно найти. Такой уровень безопасен только при условии, что вы используете режим автоматической фиксации. Вы не сможете читать незафиксированные данные, поскольку изменения фиксируются автоматически. Однако режим автоматической фиксации можно использовать только в очень простых процессах.

Read committed (уровень чтения только зафиксированных данных)

На этом уровне сеанс не видит изменений, внесенных другим сеансом, до момента их фиксации. Это похоже на то, что происходит при использовании программ контроля версий: когда файл помечен как изъятый и кто-то другой модифицирует его, вы можете видеть только последнюю версию, которая была возвращена, и не можете модифицировать файл, пока он заблокирован. Только когда файл будет возвращен (эквивалент фиксации), вы увидите новую версию. Такой уровень изоляции используется по умолчанию в SQL Server.

Его можно рассматривать как относительно безопасный, но проблемы могут возникнуть, если у вас есть «долгоиграющие» запросы к таблице, которые исполняются конкурентно с другими быстрыми запросами к той же таблице. Представьте такой случай: в конце месяца в банке запущен запрос, суммирующий остатки на счетах всех клиентов. Клиентов много, у большинства есть по несколько счетов, а таблица `accounts` содержит десятки миллионов строк. Суммирование такого количества строк не является мгновенной операцией. Пока запрос работает, бизнес функционирует в обычном режиме. Предположим, что у банка есть старый клиент, данные текущего счета которого хранятся «в начале таблицы» (что бы это ни значило). Этот клиент недавно открыл сберегательный счет, данные которого располагаются ближе к концу таблицы. Когда ваш запрос начал выполняться, на текущем счете остаток был 1000, а на новом сберегательном счете – 200. Запрос суммирует значения, считывает 1000 и продолжает работу. Вскоре после того, как программа считала остаток текущего счета, клиент решил перевести 750 с текущего счета на сберегательный. Доступ к счетам происходит по их номерам (которые являются первичными ключами), транзакция выполняется очень быстро и немедленно фиксируется, в результате остаток на сберегательном счете увеличивается на 750. Но ваша программа генерирования отчета все еще работает со счетами других клиентов, находящихся в таблице далеко от ее конца. Когда программа отчета наконец добирается до сберегательного счета этого конкретного клиента, то, поскольку транзакция уже давно зафиксирована, программа считывает 950. В результате общая сумма отчета оказывается неверной.

Существует способ избежать проблем такого рода: потребовать от СУБД ставить в начале запроса метку времени или функционально

эквивалентный последовательный номер и возвращаться к той версии данных, которая была текущей на момент запуска запроса, каждый раз, когда запрос встречает строку, обновленную после записанного времени. Этот откат ко времени, когда запрос был запущен, происходит вне зависимости от того, было ли изменение зафиксировано. В предыдущем примере программа могла бы по-прежнему считывать значение остатка на сберегательном счете, равное 200, поскольку именно это значение было текущим, когда запрос начал свою работу. Такой механизм отслеживания версий строк под названием *row versioning* имеется в SQL Server и реализуется с помощью параметра базы данных `READ_COMMITTED_SNAPSHOT`. Этот режим можно задать по умолчанию в Oracle, где он именуется *read consistency* и его можно рассматривать как вариант стандартного уровня чтения только зафиксированных данных. Механизм хранилища InnoDB в MySQL также работает на уровне чтения только зафиксированных данных (но не является уровнем по умолчанию для InnoDB).

Конечно, ничего не бывает бесплатным, и использование механизма отслеживания версий строк означает, что старые значения данных должны храниться как можно дольше, а не удаляться сразу после того, как транзакция зафиксирована. В некоторых средах, объединяющих медленные запросы и многочисленные быстрые транзакции, хранение длинной истории может привести к переполнению временного хранилища и ошибкам времени исполнения. Обычно существует предел объема хранилища, используемого для записи прежних значений. Незафиксированные значения хранятся до тех пор, пока хранилище буквально не лопнет по швам. Зафиксированные значения хранятся столько, сколько возможно. Когда свободное пространство почти исчерпано, на их место записываются значения, предшествовавшие последним изменениям. Если оператору `select` требуется версия данных, которая была стерта при освобождении места, его работа завершится ошибкой (в Oracle это неприятная ошибка «snapshot too old»). Перезапуск оператора приведет к изменению метки начала работы запроса и в большинстве случаев – к успешному исполнению. Но если ошибки такого типа происходят слишком часто, первое, что нужно проверить – нельзя ли ускорить исполнение этого оператора. Если нельзя, то следует проверить, не происходят ли фиксации слишком часто. Если и теперь не удастся решить проблему, нужно постараться увеличить пространство для задачи.

Repeatable read (воспроизводимое чтение)

Уровень изоляции *воспроизводимое чтение* является расширением отслеживания версий строк: вместо хранения данных в течение работы запроса диапазоном становится вся транзакция. Это уровень по умолчанию для InnoDB-машины в MySQL. В предыдущем примере с остатком на счете, если бы вы запустили запрос во второй раз на уровне воспроизводимого чтения, программа отчета увидела бы остаток 1000 на текущем счете и 200 – на сберегательном (на уровне чтения только

зафиксированных данных программа отчета при втором запуске считала бы 250 и 950, если эти два счета больше никто не трогал). Пока сеанс не зафиксирует изменения (или не сделает откат), программа будет иметь то же представление о данных, что и при первом считывании. Как вы можете догадаться, возникновение проблем, связанных с пространством для хранения истории данных, здесь более вероятно, чем на уровне чтения только зафиксированных данных.

Однако могут появиться новые строки, вставленные после первого оператора `select`. Чтение является воспроизводимым по отношению к данным, которые уже были считаны, но не к таблице как единому целому. Если вы хотите, чтобы вне зависимости от того, что происходит, таблица выглядела одинаково между последовательными операторами `select`, нужно переключиться на следующий уровень изоляции.

Serializable (сериализуемые транзакции)

Этот уровень изоляции представляет собой режим, в котором транзакции игнорируют все изменения, внесенные в базу данных другими транзакциями. Только когда транзакция заканчивается, сеанс видит изменения, внесенные в базу данных другими сеансами после начала транзакции. Таким образом, фиксация выполняет еще и роль кнопки обновления.

В реальной жизни в подавляющем большинстве случаев применяются только уровень чтения зафиксированных данных (с отслеживанием версий строк или без него) и воспроизводимое чтение. Эти два уровня обеспечивают хорошо согласованное представление данных без перенапряжения системы в отношении блокировок и обслуживания истории изменений.

Я упомянул блокировки в связи с различными механизмами синхронизации, которые система поддерживает для обеспечения согласованности данных. Когда вы определяете уровень изоляции, вы по умолчанию определяете шаблон блокировки. Но вы можете также принудительно заблокировать таблицы, чтобы сообщить системе, что хотите обеспечить эксклюзивную запись или (иногда) доступ к чтению¹ одной или нескольких таблиц и что каждый, кому доступ, конфликтующий с вашим, потребует после того, как вы захватили блокировку, должен будет ждать окончания вашей транзакции. И наоборот: если кто-нибудь уже получил доступ в режиме, несовместимом с вашим запросом, когда вы пытаетесь заблокировать таблицу, ваш сеанс вынужден будет ждать², пока другой сеанс не зафиксирует или не откатит свои изменения.

Однако давайте вернемся к нашей программе MySQL/MyISAM. Мы работаем в режиме автоматической фиксации, и MyISAM в каждый момент

¹ Oracle никогда не блокирует других читателей.

² Хотя Oracle позволяет указать, что если команда `lock` не может быть выполнена немедленно, она должна возвращать ошибку.

времени позволяет обновлять таблицу только одному процессу. Имеющаяся у нас проблема (конкурентные процессы «атакуют» одну и ту же строку) появляется в той фазе, когда мы пытаемся захватить строку для обработки, а именно когда мы выполняем следующее:

```
select min(seqnum) from fifo
where state = 'R';
update fifo
set state = 'P'
where seqnum = ?;
```

Происходит вот что: один процесс получает идентификатор следующей обрабатываемой строки, а прежде чем он успевает обновить таблицу, один или несколько других процессов получают тот же самый идентификатор. Нам явно нужно обеспечить, чтобы после того, как мы идентифицировали строку как кандидата на обработку (например, после того, как оператор `select` возвратил идентификатор), ни один иной процесс не мог бы с ней работать. Не забывайте, что мы не можем объединить эти два оператора, поскольку нам нужен идентификатор для обновления строки, когда мы закончим. Отдельный оператор мог бы подойти, поскольку внутренние механизмы СУБД обеспечивает так называемое свойство ACID (Atomicity, Consistency, Isolation, Durability) отдельного оператора `update` (проще говоря, вам не нужно беспокоиться об отдельном операторе, поскольку СУБД обеспечивает непротиворечивость данных).

Если мы хотим запустить несколько экземпляров программы и получить эффект от их параллельной работы, нам нужно еще раз сделать рефакторинг кода. Один из способов сделать это – выключить режим автоматической фиксации, чтобы получить лучший контроль над транзакциями, а затем в главном цикле заблокировать таблицу для нашего эксклюзивного использования, прежде чем мы будем искать следующую строку для обработки, обновить таблицу и, наконец, освободить блокировку. Тогда у нас будет гарантия того, что ни один параллельный поток не получит идентификатор, и мы будем чувствовать себя свободно, зная, что этот идентификатор будет обработан только один раз. Я использовал эту стратегию в моей программе *consumer_lock.c*.

Другой способ таков:

```
установить количество обновленных строк равным нулю
loop
    select min(seqnum) from fifo
    where state = 'R';
    /* Выйти, если ничего не найдено */
    update fifo
    set state = 'P'
    where seqnum = ?
        and state = 'R';
until количество обновленных строк равно 1;
```

Когда строка обновлена, программа проверяет, находится ли она в том состоянии, в котором должна быть. Если предположение, что строка еще

находится в состоянии *Готова*, неверно, это означает, что другой процесс оказался быстрее и захватил ее, а нам просто придется получить другую строку. Алгоритм очень близок к исходной программе, за исключением того, что оператор `update` стал безопасным и я добавил меры предосторожности на случай сбоя. Я назвал эту программу *consumer_not_so_naive.c*.

Я написал еще одну программу, использующую стратегию «безопасного обновления», но так же предполагающую использование нижележащего механизма хранилища InnoDB и доступность блокировки на уровне строк. В третьей программе добавлена фраза `for update` к оператору `select`, идентифицирующему следующую обрабатываемую строку, в результате чего выбранная строка блокируется. Эта программа называется *consumer_inno.c*. Я запустил все три программы с фиктивным отставанием, которое надо устранить, в 200 строк, средним временем простоя 0,4 секунды и переменным количеством параллельных процессов от 1 до 50. Результаты показаны на рис. 7.3. Интереснее всего на рисунке, вероятно, то, что все три кривые почти совпадают. Все три решения масштабируются очень хорошо, и время, которое требуется n процессам для устранения отставания, равно времени, которое требуется для этой задачи одному процессу, деленному на n .

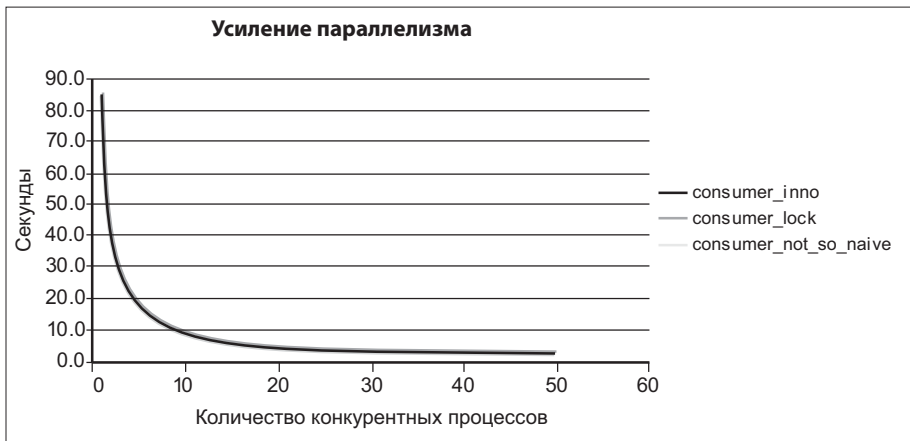


Рис. 7.3. Запуск параллельных процессов после коррекции

Вы можете удивиться, почему не имеет значения, используем мы блокировку таблицы или блокировку строки. Не забывайте, что «поставщики услуг» с точки зрения приложения являются инициаторами запроса к базе данных. В свете этого есть два очень простых аргумента, доказывающие, что блокировка строк работает не лучше, чем блокировка таблиц:

- все процессы «охотятся» за одной и той же самой старой необработанной строкой. Для каждого процесса в счет идет только одна строка, которая становится критичным ресурсом. Неважно, застряли вы

в ожидании этой строки или таблицы, – блокировка строки в данной ситуации не приносит никакой заметной прибыли;

- даже если нам придется сканировать таблицу `fifo`, небольшое количество строк в ней делает работу оператора `select` почти мгновенной. Даже когда мы блокируем таблицу на время выполнения операторов `select` и `update`, блокировка удерживается столь малое время относительно «скорости поступления», что очереди не образуется.

В противоречие тому, что вы могли бы подумать, характер блокировок не обязательно является решающим фактором в вопросе производительности на относительно малых таблицах (до тысяч или десятков тысяч строк). Если доступ к данным быстрый, конкуренция умеренная, а блокировки держатся доли секунды, у вас, вероятно, не возникнет проблем с производительностью. Но когда задействованы тысячи конкурирующих процессов, нацеленных на различные строки в больших таблицах, и обновления, которые нельзя назвать мгновенными, то блокировка строк дает преимущества за счет некоторого «перекрытия» во время модификации строк.

Укорачивание критических разделов

В конкурентном программировании критическим разделом является часть кода, к которому может получить доступ только один процесс или поток. Когда СУБД устанавливает блокировку, код, который выполняется до конца транзакции (будь то операторы SQL или код из основного приложения, исполняющего вызовы SQL), по вашему явному запросу или неявно становится потенциально критическим разделом. Все зависит от блокировки другого процесса. Будьте внимательны при блокировке строк: блокировки, которые действуют на диапазоны строк, легко могут заблокировать другие процессы. Еще легче ожидания блокировки могут возникать в СУБД, которые расширяют блокировки (подобно SQL Server или DB2). Это означает, что если много единичных ресурсов блокируется одним и тем же процессом, многочисленные блокировки автоматически заменяются одной объединенной блокировкой. Вы можете переключиться, например, на блокировку всей страницы от многочисленных блокировок строк на странице.

Если вы хотите, чтобы ваши конкурентные процессы мирно сосуществовали, нужно постараться укоротить критические разделы.

Нужно помнить один важный момент, заключающийся в том, что при обновлении или удалении строк и вставке результата запроса в таблицу идентификация строки для удаления, обновления или вставки занимает часть времени отклика оператора SQL. В случае оператора `insert ... select ...` это очевидно. В других случаях неявный оператор `select` является результатом фразы `where`. Если этот субоператор работает медленно, он может оказаться самой длительной частью всего периода отклика.

Эффективным способом улучшения конкуренции при использовании блокировок таблиц является рефакторинг кода с разбиением на два шага (может показаться, что я противоречу тому, что писал в предыдущей главе,

но читайте дальше). Если вы действуете в режиме блокировки таблиц, то таблица `t` блокируется, как только СУБД начинает исполнять оператор `update t set ...`. Все зависит от отношения времени, требуемого для поиска строк, ко времени, требуемому для их обновления. Если поиск строк является быстрым процессом, а обновление занимает длительное время (поскольку вы обновляете столбец, который присутствует во многих индексах), вы мало что можете сделать, за исключением удаления некоторых индексов, польза от которых сомнительна. Но если идентификация результирующего набора занимает значительную часть времени, имеет смысл исполнять две фазы отдельно, поскольку при поиске строк интенсивность блокировок значительно ниже, чем при их обновлении. Если вы хотите обновить единственную строку, сначала получите для одной или нескольких переменных лучшие идентификаторы из возможных: либо столбцы, которые образуют первичный ключ, либо, что еще лучше, идентификатор, подобный `rowid` (адрес строки) в Oracle. Это не требует двухшагового обновления, поскольку блокировка будет реализована на уровне строки. Если вы хотите обновить много строк, можно также использовать временную таблицу¹. Затем вы применяете изменения, которые блокируют всю таблицу, но при использовании этих идентификаторов – и немедленной фиксации по завершении – вы минимизируете период времени, в течение которого сохраняете монополию на таблицу.

Изолирование опасных зон

Усиление параллелизма обычно приводит к усилению конкуренции, и производительность часто оказывается ограниченной «трением» в одном месте. Типичным случаем, который я встречал неоднократно, является таблица, хранящая значения «следующего идентификатора» для одного или нескольких суррогатных ключей, то есть внутренних идентификаторов, используемых в качестве удобных псевдонимов для громоздких композитных первичных ключей. Примером может служить значение `orderid`, которое будет присвоено следующему создаваемому заказу. Нет необходимости использовать такую таблицу, поскольку в MySQL есть столбцы с автоматическим приращением, в SQL Server – столбцы идентификации, а в Oracle (как и в Postgres или DB2) существуют последовательности. Даже при различной реализации их функциональность будет одинаковой, а различные реализованные механизмы – оптимизированными для СУБД, в которой они поддерживаются. Более того, все системы баз данных содержат простые средства, позволяющие получать последние сгенерированные значения для дальнейшего использования.

Причиной, по которой многие разработчики используют суррогатные ключи, является желание сохранить независимость от используемой СУБД в «донкихотских» поисках переносимости. В действительности ограничением переносимости SQL являются самые распространенные

¹ Я не большой любитель временных таблиц. Но, как и любые другие возможности, они имеют свою область применения.

функции, например функции арифметики дат, сильно зависящие от СУБД. Даже очень распространенная функция вычисления длины строки называется `length()` в Oracle и MySQL, но `len()` в T-SQL. Если вы не планируете ограничиваться только простыми запросами (в таком случае пользы от этой книги для вас будет очень мало), перенос кода SQL с одной СУБД на другую будет означать некоторую переработку – не обязательно сложную, но, тем не менее, переработку (многие примеры из данной книги это подтверждают). Если вашей целью является приложение с одинаково низкой производительностью на широком спектре баз данных, использование суррогатных ключей приемлемо. Если вы беспокоитесь о производительности, наилучшей стратегией будет идентификация и сохранение всех частей кода, предназначенных для конкретной системы, на уровне абстракции или в центральном месте, и использование того, что подходит для конкретной СУБД.

Избавление от таблицы, хранящей счетчики, часто требует довольно серьезной переработки и даже более серьезного тестирования, поскольку обычно запросы и обновления такой таблицы происходят из многих мест вашего кода. Существование выделенной функции ничего не меняет, поскольку даже с последовательностью Oracle нет оснований запускать

```
select имя_последовательности.nextval
into my_next_id
from dual;
```

перед оператором `insert`, когда вы можете непосредственно вставить `имя_последовательности.nextval`. Выбор значения из последовательности или его автоматическое генерирование вместо извлечения из таблицы является значительно более серьезным изменением, чем поиск значений, возвращенных функцией. Тем не менее проблемы конкуренции для таблицы, хранящей «следующие номера», могут сильно снизить производительность, а если мониторинг показывает, что значительная часть времени тратится на работу с этой таблицей, то вам стоит заменить ее на генератор этой СУБД.

Во многих случаях проблема заключается не в плохом проектировании, а, скорее, в логике обработки, заставляющей несколько процессов действовать в одной и той же «области» базы данных и потому пересекаться друг с другом. Например, если несколько процессов конкурентно вставляют строки в одну и ту же таблицу, вам нужно выяснить, как таблицы хранятся физически. Если вы используете параметры по умолчанию, все процессы будут ждать возможности добавления строки к таблице, и после того как будет вставлена последняя строка, возникнет перегрузка: все процессы не могут осуществлять запись с одинаковым смещением на одну и ту же страницу памяти в одно и то же время, и даже в случае очень гранулированных блокировок должен иметь место некоторый род сериализации.

Работа с несколькими очередями

Для случая, при котором процессы удаляли задачи из таблицы `fifo`, я привел пример, где к приложению было добавлено параллельное исполнение,

а совместно используемым ресурсом была таблица `fifo`. Но, как я отмечал, модель очередей применялась на многих уровнях, от верхнего до нижнего. Попробуем рассмотреть простейший пример: что происходит, когда мы вставляем одну строку в таблицу. Как показано на рис. 7.4, у нас есть несколько потенциальных очередей:

1. Нам может потребоваться захватить блокировку таблицы, но если таблица уже заблокирована в несовместимом или эксклюзивном режиме, нам придется ждать.
2. Затем мы будем либо искать свободное место в таблице для вставки данных, либо добавлять строку в конец таблицы. Если несколько процессов пытаются одновременно выполнить вставку данных, то пока один записывает байты в память, другим приходится ждать. Может также происходить рекурсивная деятельность самой базы данных для расширения пространства, выделенного таблице.
3. Затем, вероятно, надо будет обработать один или несколько индексов: нам придется добавлять адрес вновь вставленной строки, связанный со значением ключа. Кроме того, возможно применить операцию выделения пространства и, если несколько процессов пытаются обновить одно и то же значение ключа или очень близкие значения ключа, они будут пытаться выполнить запись в одну и ту же область индекса и, как и в случае таблицы, потребуется сериализация.
4. Фиксация изменений будет означать синхронную запись в файл журнала. Несколько процессов не могут одновременно записывать в файл журнала, так что придется ждать подтверждения от системы ввода-вывода.

Этот список не является исчерпывающим. Возможно, потребуются операции ввода-вывода для загрузки в память страниц, относящихся к таблице или ее индексам, или нужно будет проверить ограничения внешнего ключа, чтобы гарантировать, что вставляемые данные действительно соответствуют правильным значениям. Могут сработать триггеры и выполняться другие операции, которые будут транслированы в последовательность очередей. Если компьютер перегружен, может также образоваться очередь процессора.

Тем не менее достаточно посмотреть на рис. 7.4, чтобы понять: одной очереди, ожидать в которой приходится долго, достаточно, чтобы убить производительность, и когда есть несколько мест, где возможно ожидание, нужно быть особенно внимательными, чтобы не начать исправлять не ту проблему¹. Чтобы проиллюстрировать это, я написал и запустил

¹ По этой теме я не могу сделать ничего лучше, чем посоветовать пользователям Oracle книгу «Optimizing Oracle Performance», написанную Cary Millsap и Jeff Holt (издательство O'Reilly), в которой подробно объясняется, как идентифицировать места, где действительно происходят потери времени, и как избежать различных «ловушек». В книге есть также глава, посвященная теории очередей для тех, кто работает с базами данных.

простой пример с помощью инструмента под названием Roughbench (его можно бесплатно скачать по адресу <http://www.roughsea.com> или из раздела, посвященного этой книге на <http://www.oreilly.com/catalog/9780596514976>). Программа Roughbench представляет собой простую оболочку JDBC, которая берет файл SQL и многократно запускает его содержимое либо фиксированное количество раз, либо в течение фиксированного периода времени. Этот инструмент может запускать переменное количество потоков, которые будут исполнять один и тот же оператор, и может генерировать случайные данные.

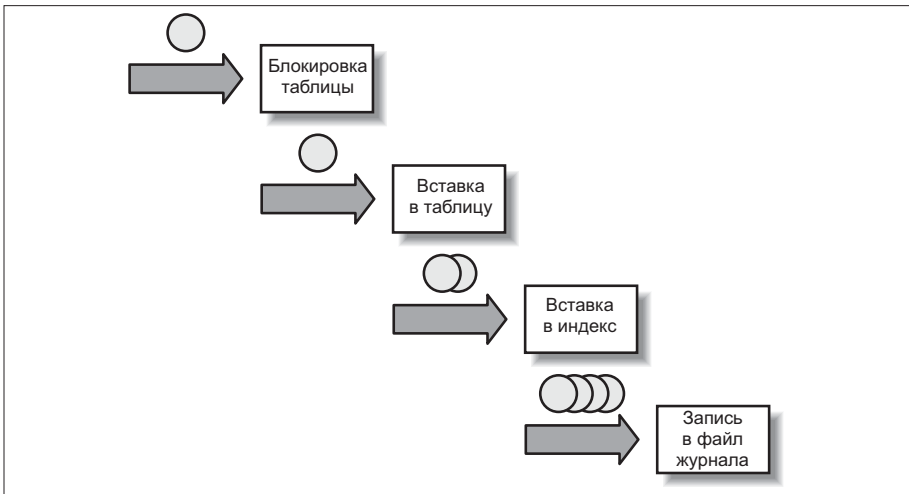


Рис. 7.4. Несколько очередей: где затор?

Я создал следующую тестовую таблицу:

```
create table concurrent_insert(id          bigint not null,
                               some_num    bigint,
                               some_string  varchar(50));
```

В другой серии запусков я сделал `id` столбцом с автоматическим приращением и объявил его первичным ключом (неявно создав таким образом уникальный индекс по нему). Наконец, в последней серии запусков я оставил `id` первичным ключом и разбил таблицу на пять разделов по этому столбцу.

В течение одной минуты я вставлял строки в таблицу, используя последовательно от одного до пяти потоков, и проверял, сколько строк мне удастся вставить.

Варианты тестовой таблицы не были выбраны случайным образом. Таблица без индексов является всего лишь основой. В правильно спроектированной базе данных я мог бы ожидать, что каждая таблица имеет какой-то первичный ключ или, по крайней мере, уникальный столбец. Но это позволяет нам измерить влияние как при единственном потоке,

так и в конкурентной среде наличия (некластеризованного) первичного ключа, основанного на столбце с автоматическим приращением. Я логически ожидаю меньшей производительности с индексом, чем без него, основываясь на случае, где у меня был только один поток, поскольку управление индексом добавляет накладные расходы к управлению таблицей. Когда я запускаю больше одного потока, меня интересует наклон кривой по мере добавления новых процессов: если наклон не меняется, мой индекс не добавляет трения. Если наклон снижается, для конкуренции и масштабируемости это означает, что индекс плох. Поскольку осуществлено разбиение на разделы, СУБД приходится для каждой строки вычислять, в какой раздел она должна попасть, что делается (в данном случае) путем хеширования значения `id`. Поэтому разбиение делает единственный поток несколько менее эффективным, чем он был при отсутствии разбиения. Но я также ожидаю, что процессы будут мешать друг другу по мере роста конкуренции. Я создал количество разделов, равное максимальному количеству процессов, с целью распределения конкурентных вставок по таблице и индексу — что не поможет, если вся таблица заблокирована во время вставки. В табл. 7.1 приведены относительные результаты, которые я получил, выполняя фиксацию после каждой вставки для хранилищ `MyISAM` и `InnoDB`¹. На деле они не соответствуют тому, что мы ожидали.

Таблица 7.1. Относительная скорость вставки, фиксация после каждой строки

	Хранилище	Количество потоков				
		1	2	3	4	5
Без индекса	MyISAM	1,00	1,39	1,45	1,59	1,50
Автоприращение	MyISAM	0,78	1,19	1,21	1,36	1,31
Разбиение	MyISAM	0,75	1,12	1,13	1,26	1,23
Без индекса	InnoDB	0,31	0,49	0,58	0,63	0,68
Автоприращение	InnoDB	0,32	0,47	0,59	0,63	0,67
Разбиение	InnoDB	0,30	0,46	0,56	0,61	0,65

Таблица 7.1 требует некоторых пояснений:

- Во-первых, масштабируемость очень плоха. Конечно, умножение на пять количества строк, вставленных в пять раз большим количеством

¹ Поскольку результаты, полученные на моей относительно скромной двухъядерной двухпроцессорной тестовой машине имеют небольшую универсальную ценность, я выразил все результаты по отношению к количеству строк, которые мне удалось вставить с одним потоком в неиндексированную таблицу `MyISAM`. К вашему сведению, это было около 1500 строк в секунду.

процессов, является хорошим сюжетом для сказок и маркетинговых пресс-релизов. Но все-таки, если машина подошла к полной загрузке процессора при пяти активных потоках, но недостаточно близко, нужно попытаться сделать лучше.

- Обнаруживается очень значительное снижение производительности с хранилищем InnoDB, что само по себе не поражает. Вы должны помнить, что в InnoDB есть много механизмов, которые недоступны в MyISAM, и ссылочная целостность – не последний из них. Встроенные механизмы этого хранилища позволяют избежать дополнительных операторов и сетевых обращений к серверу за подтверждением. Если производительность при вставке оказывается ниже по сравнению с MyISAM, часть этой разницы, вероятно, окупится еще где-то, и различие в целом будет значительно меньше.

Замечу в скобках, что если вы действительно хотите добиться впечатляющей производительности, существует очень быстрое хранилище под названием ЧЕРНАЯ ДЫРА (эквивалент файла UNIX/*dev/null*...).

- Пик результатов с MyISAM на четырех потоках (на машине с эквивалентом четырех процессоров). InnoDB начинает со значительно более низкой базы, но общее количество вставленных строк равномерно растет, и даже если масштабируемость далека от выдающейся, она значительно лучше, чем у MyISAM. С InnoDB количество строк, вставленных за одну минуту при пяти процессах, приблизительно в 2,2 раза больше, чем при одном.
- Наиболее заметной характеристикой, вероятно, является то, что если с MyISAM первичный ключ снижает производительность и разбиение не дает ожидаемого улучшения при росте конкуренции, то с InnoDB нет разницы между различными вариантами. Равномерная производительность InnoDB произведет на вас большее впечатление, когда вы поместите графики на основе цифр из табл. 7.1 рядом друг с другом, как я сделал на рис. 7.5 (на рис. 7.5 не очевидно только то, что соотношение между наибольшей и наименьшей скоростями вставки в моем тесте с InnoDB оказалось значительно лучше, чем с MyISAM).

Причина, по которой таблицы InnoDB не показывают разницы, проста: блокирующей все очередь является очередь, которая пишет в файл журнала. На практике идентификацию точки, которая тормозит всю цепь, с SQL Server или Oracle произвести легче, чем с той версией MySQL-машины, с которой я выполнял свои тесты. В этой версии в представлении `information_schema.global_status` ожидания были несколько короче, и обширный вывод команды `show engine innodb status` не очень легко интерпретировать, если вы не знаете точно, что ищете.

Тем не менее когда я фиксирую изменения каждые 5000 строк, разница очевидна. Выбор числа 5000 – случайный. Я сохранил ту же самую ссылку в табл. 7.1, которая представляет собой количество строк, вставленных в режиме автоматической фиксации в неиндексированную таблицу MyISAM.

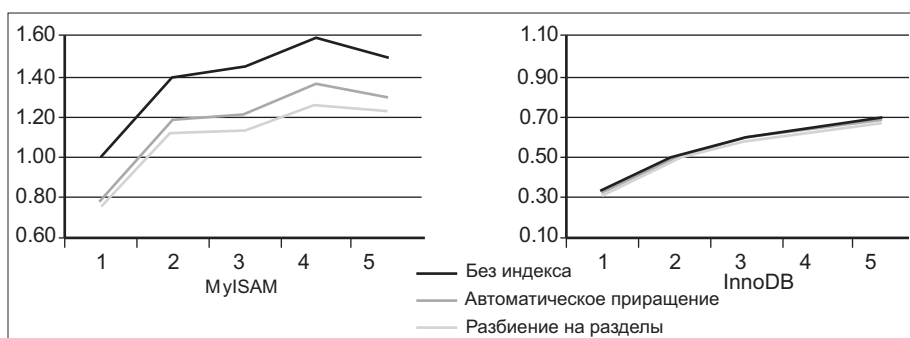


Рис. 7.5. График результатов из таблицы 7.1

Таблица 7.2 рассказывает несколько иную историю, чем табл. 7.1, – с одним потоком. Если добавление индекса не дает ощутимой разницы в InnoDB, что несколько неожиданно, то разбиение снижает производительность еще больше. Но, наблюдая эволюцию по мере добавления новых потоков, вы можете заметить, что если значительно более редкая фиксация изменений просто поднимает кривые MyISAM, то удаление узкого места при записи в журнал для InnoDB дает скачок.

Разница в поведении в случае InnoDB, возможно, лучше видна на рис. 7.6: во-первых, если мы проигнорируем нереальный случай неиндексированной таблицы с тремя и более потоками, производительность при разбиении на разделы будет выше на InnoDB. Во-вторых, мы также можем увидеть, что конкуренция в индексе по первичному ключу ниже, чем в таблице, поскольку производительность с индексом растет значительно медленнее, чем без него. В-третьих, разбиение устраняет конкуренцию и в таблице, и в индексе, и в сочетании с блокировкой строки позволяет нам получить такую же скорость вставки, что и при отсутствии индекса.

Таблица 7.2. Относительная скорость вставки, фиксация после 5000 строк

	Хранилище	Потоков				
		1	2	3	4	5
Без индекса	MyISAM	1.49	2.21	2.15	2.40	2.47
Автоприращение	MyISAM	1.26	1.80	1.74	1.97	1.78
Разбиение	MyISAM	1.19	1.86	1.60	1.74	1.58
Без индекса	InnoDB	1.18	1.83	1.95	2.17	2.01
Автоприращение	InnoDB	1.19	1.69	1.76	1.85	1.72
Разбиение	InnoDB	1.11	1.68	1.86	2.11	2.01

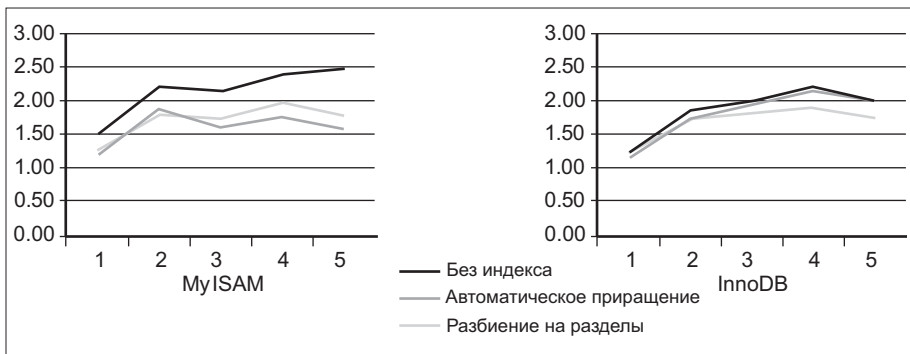


Рис. 7.6. Когда больше нет блокировки при записи в файлы журнала

Моя не слишком мощная машина не позволила мне продолжить тесты, добавляя новые потоки, и производительность достигла максимума при четырех потоках. Из этого простого примера можно извлечь несколько уроков:

Вы не должны начинать с попыток улучшить неправильную очередь

Исходной проблемой в этом примере была скорость фиксации. При фиксации после каждой вставки разбивать таблицу для увеличения производительности было бессмысленно. Не забывайте, что разбиение таблицы, которая уже содержит десятки миллионов строк, является операцией, требующей планирования и времени, и она не свободна от риска. Если вы неправильно идентифицируете главное «узкое место», вы можете мобилизовать многих людей на решение бесполезной задачи и дать надежды, которые не сбудутся, даже если новая физическая структура будет правильным выбором при правильных условиях.

Сдвиг «узких мест»

В зависимости от конкуренции, индексирования, частоты фиксации, объема данных и других факторов очередь, в которой скорость поступления и время обслуживания вступают в противоречие, не всегда остается одной и той же. В идеале вы должны пытаться прогнозировать, основываясь на предыдущем опыте.

Удаление «узких мест» является итеративным процессом

После того как вы внесете исправления в том месте, где очередь удлинится, клиенты станут перемещаться в следующую очередь быстрее, таким образом увеличивая скорость поступления для следующей очереди и, возможно, делая ее «непроходимой» для поставщика услуг. Вы не должны удивляться, если одно улучшение производительности внезапно приводит к появлению новой проблемы, которая до того момента пряталась в тени только что решенной проблемы.

Вы должны обращать внимание не только на производительность, но и на масштабируемость

Производительность и масштабируемость – это не всегда одно и то же, и иногда имеет смысл выбрать решение, которое на данный момент является не самым эффективным, но гарантирует ровный и относительно предсказуемый рост. Это особенно справедливо, если, допустим, какое-то обновление оборудования уже запланировано на следующий квартал, и вы хотите получить максимальную прибыль от обновления. Это может быть эффективное хранилище, или большее количество процессоров, или более мощный процессор – улучшение, которое может избавить от одного «узкого места» и ничего не сделать с другими. Поэтому вам нужно продумать пути усовершенствования, которые вы наметили, и выбрать тот, который с наибольшей долей вероятности принесет удовлетворительные результаты.

Чтобы убедиться в том, что производительность и масштабируемость в одних условиях не являются одним и тем же, взгляните на рис. 7.7, где я поместил рядом как графики измерений производительности, полученные мной с четырьмя потоками при различных конфигурациях, так и графики соотношений производительности, полученные при изменении количества потоков от одного до четырех (это соотношение обозначено на рисунке как «Масштабируемость»). На этом очень ограниченном примере вы можете увидеть, что комбинация разбиения InnoDB-машины дает хорошую производительность (лучшую, если рассматривать только случаи, когда определен первичный ключ) и скромную масштабируемость. Если вы ожидаете долгосрочного роста, то в данном примере это, вероятно, лучший вариант. С другой стороны, если единственным варьированием нагрузки, которого вы ожидаете, является внезапный всплеск, который происходит время от времени, выбор простого решения с MyISAM не кажется неудачным. Но «MyISAM плюс разбиение» является решением, которое (применительно к этому примеру) приносит больше вреда, чем пользы (я говорю «к этому примеру», поскольку у вас могут быть другие заботы, а не только смягчение конкуренции для использования разбитой на разделы таблицы). Упрощение архивирования и очистки является хорошим аргументом, при котором ничего не надо делать с конкуренцией и блокировками.

Параллелизм вашей программы и СУБД

Увеличение количества клиентов и поставщиков услуг и попытки минимизировать блокировки и конкуренцию при обеспечении согласованности данных – это приемы, которые не всегда удается успешно реализовать, но которые известны и понятны многим. Значительно реже я встречал людей, пытающихся распараллелить свою программу и работу СУБД. Хотя даже не очень мощная машина может сейчас обрабатывать несколько задач действительно параллельно, мы все еще используем языки программирования, являющиеся наследниками

Фортрана и Кобола, из тех времен, когда один процессор последовательно выполнял инструкции одну за другой. Учтите также, что многозадачность не является естественной для человеческого разума (по крайней мере, для меня), и вы поймете, что нет ничего удивительного в том, что процессы часто проектируются как исключительно линейные по своей сути.



Рис. 7.7. Сравнение производительности и масштабируемости с четырьмя процессами

Когда вы выполняете операторы `select` внутри программы, последовательность операций всегда одинакова, независимо от СУБД и языка, при помощи которого осуществляется вызов:

1. Вы получаете обработчик для исполнения оператора.
2. Вы выполняете вызов для синтаксического разбора оператора.
3. Вы привязываете некоторые параметры ввода к вашему оператору (необязательный шаг).
4. Вы определяете области памяти, куда вы хотите поместить данные, возвращаемые из базы данных.

5. Вы исполняете запрос.
6. Как только код возврата указывает на удачное завершение, вы извлекаете строки.

Иногда составные API объединяют некоторые из вышеописанных шагов в единый вызов функции. Но даже если вы получаете в конечном счете массив значений, все шаги остаются прежними.

Большинство людей не отдают себе отчета в том, что почти все вызовы базы данных являются синхронными, то есть когда вы выполняете вызов базы данных, ваша программа ничего не делает до тех пор, пока вызов не будет возвращен. Это не является проблемой в транзакционной среде с высокой конкуренцией, поскольку (если только все конкурирующие процессы не находятся в очереди ожидания на стороне базы данных) пока ваш сеанс простаивает, другой сеанс полностью загружает процессор. Однако в некоторых средах вы можете увеличить производительность, заставив клиентскую сторону работать, когда СУБД тоже работает. Хорошим примером является загрузка хранилищ данных, пакетная операция, часто обозначаемая как процесс ETL (Extract/Transform/Load). Если, как часто бывает, вы извлекаете данные из одной базы данных для загрузки в другую, ваша программа будет ждать дважды: когда вы извлекаете данные и когда вы их вставляете.

Если функции API СУБД позволяют (что бывает не всегда), вы можете, особенно если ваш процесс преобразований требует некоторого процессорного времени, увеличить производительность, сделав следующее (см. рис. 7.8):

- Фаза извлечения возвращает первый пакет данных из исходной базы данных. Пока вы мало что можете сделать.
- Вместо последовательной обработки данных и последующей вставки их в целевую базу данных вы копируете только что полученные данные в другую область памяти (в том виде, в котором они получены) или привязываете другую область памяти к вашему оператору `select`, прежде чем извлекать следующий пакет.
- Пока второй пакет извлекается, другой поток преобразует то, что вернул первый пакет, и записывает эти данные в третью область памяти. Закончив, он ждет, пока завершится извлечение второго пакета.
- В это же время третий поток берет преобразованный набор из третьего буфера, копирует его в четвертый и загружает содержимое этого четвертого буфера в целевую базу данных.

В крейсерском режиме у вас работают параллельно три потока: один извлекает пакет данных № n , в это время второй пакет преобразовывает пакет данных № $n - 1$, а третий загружает пакет № $n - 2$.

Нет надобности говорить, что синхронизация может оказаться сложной, поскольку ни один поток не должен осуществлять запись в область памяти, которая еще не обработана до конца следующим потоком в очереди, и не должен начинать считывание своего ввода, если предыдущий

поток еще не выполнил свою задачу. Если вы не в совершенстве владеете семафорами и многопоточностью, я бы не советовал использовать механизм такого типа.

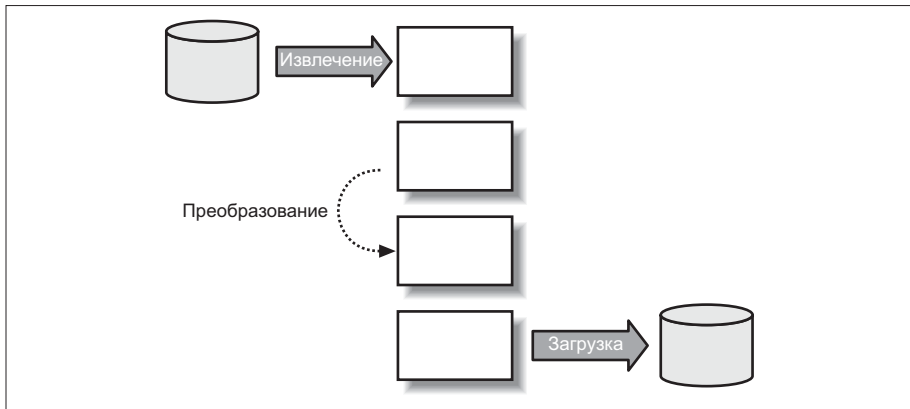


Рис. 7.8. Распараллеливание процессов ETL

Вы можете применить ту же самую идею в контексте приложения, которое будет работать через глобальную сеть (WAN). Я видел несколько приложений, которые осуществляли доступ к базам данных, расположенным на другом континенте. Поскольку обновление скорости света пока не выполнено, вы вряд ли сможете избежать полусекундной сетевой задержки между Нью-Йорком и Гонконгом, поэтому имеет смысл использовать кэш памяти для локального хранения данных только для чтения. Проблемой обычно является загрузка в кэш, которая выполняется либо при запуске (или перезапуске) сервера приложений, либо в момент соединения, если кэш является локальным для сеанса. За полсекунды сетевой задержки компьютер может сделать много. Если вместо извлечения данных, заполнения кэша, извлечения следующего пакета данных, заполнения кэша и т. д. у вас есть один поток, заполняющий кэш, выделяющий память и привязывающий структуры данных, в то время как другой поток извлекает остающиеся данные, вы, вероятно, сможете значительно укоротить периоды запуска или соединения.

Потрясающие основы

В этой главе я показал, что иногда разбиение таблицы является эффективным способом ограничить конкуренцию, когда несколько конкурирующих потоков обращаются к базе данных. Я не знаю, насколько модификация физического дизайна базы данных относится к рефакторингу, но поскольку дизайн может оказывать значительное влияние на производительность, я представляю вам на рассмотрение несколько изменений и кратко объясню, когда и почему они могут быть полезны.

К сожалению, физические изменения редко являются благом для всех операций, и это не позволяет рекомендовать какое-либо конкретное решение без множества предостережений. Вы должны тщательно протестировать изменения, которые хотите внести, составить некоторое подобие ведомости прибылей и убытков и «взвесить» каждое увеличение или уменьшение производительности с точки зрения его важности для рабочего процесса. Как я уже указывал, разбиение большой таблицы, которая не была разбита с самого начала, может оказаться длительным мероприятием, и, как любое длительное мероприятие, представлять опасность для рабочего процесса. Я никогда не видел, чтобы изменения физического дизайна предлагали для маленьких баз данных. Реорганизация же большой базы данных является весьма разрушительной операцией, которая требует недель тщательного планирования, безупречной организации и безошибочного исполнения (добавлю, что никто не оценит ваших усилий, если все пройдет благополучно, а вот в случае неудачи вы точно заработаете плохую репутацию). Не удивляйтесь, если администраторы базы данных не воспримут вашу идею с энтузиазмом. Поэтому вы должны быть абсолютно уверены, что такие изменения являются реальным решением, а не заменой старых проблем на новые.

Разбиение на разделы – это только один, хотя и, вероятно, наиболее достойный одобрения, тип физических изменений, которые вы можете внести в таблицу. Вы можете также предложить изменения, которые повлияют как на логическую, так и на физическую структуру базы данных. Это могут быть простые изменения типа добавления столбца в таблицу в пользу какого-то конкретного процесса, более сложные (например, изменение типа данных), или такие сложные, как реструктуризация таблиц или перемещение части данных в удаленную базу данных.

Изменение физической структуры базы данных для увеличения производительности является очень популярным подходом среди некоторых людей (но не среди администраторов баз данных), которые видят в физической реорганизации способ форсирования работы приложений без необходимости решения слишком большого количества вопросов в этих приложениях.

Вы должны хорошо понимать, что физическая реорганизация любого рода – это не волшебство, и таким образом редко удастся получить ускорение того же порядка, что и при модификации доступа к базе данных (см. пятую и шестую главы). Любое изменение базы данных должно иметь серьезные основания:

- Если вы неправильно идентифицировали причину медленной работы, вы можете затратить большие усилия впустую. Предыдущий пример с конкурентными вставками иллюстрирует именно этот случай: при фиксации изменений после каждой вставки разбиение на разделы не приносит заметной пользы, поскольку «узким местом» является конкуренция при записи в файл журнала, а не

конкуренция при записи в таблицу или в индекс по первичному ключу. Только после того, как решена проблема с файлом журнала, можно выяснить, не сместилось ли «узкое место» к конкурирующим вставкам в таблицу и (в большинстве случаев) в индекс, и может ли разбиение помочь в увеличении производительности.

- Другим важным моментом является то, что преимущества, которые вы можете извлечь из физической реорганизации, обычно бывают двух разных типов: либо вы пытаетесь кластеризовать данные, чтобы все данные, которые вы ищете, оказывались в небольшом количестве страниц, тем самым минимизируя объем памяти, который приходится сканировать, и количество операций ввода-вывода, либо вы пытаетесь распределить данные так, чтобы уменьшить конкуренцию и направить различных «клиентов» в различные «очереди». Вероятно, любое разбиение неблагоприятно повлияет на некоторые части кода. Если вы улучшаете критическую часть и ухудшаете вспомогательную, это хорошо. Если вы заменяете одну проблему другой – это может быть неудачным решением.

Теперь посмотрим, как мы можем ускорить обработку.

Сортировка строк

Как мы только что видели, разбиение является лучшим способом уменьшения конкуренции благодаря тому, что конкурирующие процессы обращаются к разным физическим областям единой логической таблицы. Еще одним полезным способом реорганизации является группировка строк, которые обычно принадлежат к одному и тому же результирующему набору. Такая организация особенно важна для строк, которые извлекаются с помощью сканирования диапазона в индексе, то есть с помощью подобного условия:

```
and some_indexed_column between <min value> and <max value>
```

Во второй главе я упоминал связь между порядком ключей в индексе и порядком соответствующих строк. Эта связь нигде не бывает столь важной, как в случае сканирования диапазона. В худшем случае может произойти следующее (рис. 7.9): когда вы сканируете индекс, все последовательные ключи, которые вам встречаются, ссылаются на разные страницы таблицы. Такой запрос, скорее всего, приведет к большому количеству операций ввода-вывода, поскольку только некоторые из страниц уже будут в памяти, когда они понадобятся.

К этому вы можете добавить простой процессора, поскольку нужно будет получить доступ к каждой странице в памяти для извлечения только одной строки.

Конечно, идеальным вариантом было бы найти все строки, адресуемые набором ключей, собранным на одной странице или на минимально возможном количестве страниц: это позволило бы минимизировать операции ввода-вывода, и сканирование страниц было бы действительно

полезно. Если мы хотим создать такую идеальную ситуацию для всех возможных диапазонов ключей, у нас есть только один вариант решения: строки в таблице должны находиться в том же порядке, что и ключи в индексе. Чтобы достичь этой цели, мы либо заново создаем таблицу и вставляем строки после сортировки по индексному ключу (с риском столкнуться с необходимостью выполнять ту же операцию через регулярные интервалы, если строки регулярно вставляются, удаляются и обновляются), либо определяем таблицу с самоорганизующейся структурой, отвечающей нашим потребностям. Мы можем достичь цели, либо определив индекс как кластерный индекс в SQL Server или в MySQL и InnoDB-машине, либо определив таблицу как индекс-таблицу (эквивалент Oracle)¹.

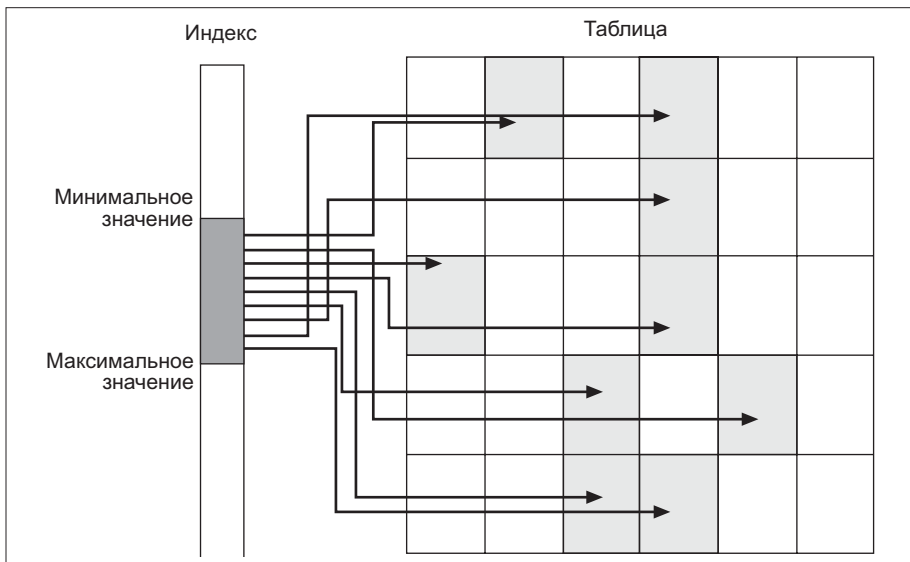


Рис. 7.9. Неудачное сканирование диапазона индекса

Мы должны помнить о нескольких условиях:

- строки упорядочиваются либо по индексу `idx1`, либо по индексу `idx2`, но не по обоим сразу, за исключением случаев, когда есть очень сильная связь между столбцами, проиндексированными в `idx1`, и столбцами, проиндексированными в `idx2` (например, последовательные номера с автоматическим приращением и даты вставки). Один индекс, а следовательно, одна категория поиска в диапазонах будет иметь несправедливое преимущество перед остальными;

¹ Есть и другие возможности (например, добавление в индекс столбцов), чтобы находить всю информацию в индексе, что позволяет избежать непрямого доступа к таблице.

- индексом, который структурирует таблицу, обычно должен быть индекс по первичному ключу. Может случиться так, что если первичный ключ состоит из нескольких столбцов, порядок столбцов в первичном ключе будет не вполне соответствовать нужному нам порядку. В такой ситуации мы должны будем переопределить первичный ключ, что, в свою очередь, означает необходимость переопределить все внешние ключи, которые указывают на этот первичный ключ, и, возможно, также на некоторые индексы – все это может завести нас довольно далеко;
- вставки будут дороже, поскольку вместо простого добавления новой строки вам придется вставлять ее в определенное место. Вы не захотите обновлять первичный ключ, а следовательно, обновления не являются проблемой;
- широкие (то есть содержащие много столбцов) таблицы делают вставку значительно более трудной, и такая организация принесет им мало пользы: если строки очень длинные, в любом случае на одну страницу попадет очень мало строк, и их правильный порядок не принесет значительной разницы;
- конкуренция будет действительно сильной, если у нас будут конкурентные вставки.

Когда некоторые из этих условий не могут быть удовлетворены, на помощь может прийти повторное разбиение. Если вы сделаете разбиение по диапазону ключей, данные будут упорядочены не так жестко, как в предыдущем случае, но будут в определенной степени собраны вместе: вы будете уверены, что весь результирующий набор находится в разделе, содержащем минимальное значение и все следующие разделы, вплоть до раздела, содержащего максимальное значение для вашего сканирования. В зависимости от различных условий, приведенных в запросе, возможно даже, что индекс не потребуется и будет достаточно прямого сканирования соответствующих разделов.

Как и в случае кластерных индексов, такое использование разделов может скорее увеличить, а не уменьшить конкуренцию. Типичным случаем является разбиение по неделям дат создания строки в таблице, в которой хранятся данные за год: в любой конкретный момент наиболее активной частью таблицы будет раздел, соответствующий текущей неделе, где все будут хотеть вставлять данные немедленно. Если вставка данных является процессом с серьезной конкуренцией, вы должны выбрать между разбиением, распределяющим данные по всему пространству, и разбиением с кластеризацией данных. Кроме того, вы можете попытаться использовать подразбиение. Предположим, таблица содержит заказы на поставку. Вы можете сделать разбиение по диапазонам дат заказа, а внутри каждого раздела сделать дальнейшее разбиение, допустим, по идентификаторам клиентов. Конкурентные процессы, которые вставляют заказы на поставку, будут, конечно, обращаться к одному и тому же разделу, но поскольку все процессы, скорее всего, будут иметь дело

с разными клиентами, они (конкурентные процессы) будут обращаться к различным подразделам.

Подразбиение (использование которого я встречал редко) может быть хорошим способом соблюсти баланс между противоположными целями. С учетом того, что я говорил, оно не сильно поможет, если вы будете запускать много запросов, при помощи которых хотите извлекать все заказы одного клиента за год, поскольку тогда эти заказы будут разбросаны по 52 подразделам. Разбиения и подразбиения являются отличными решениями для организации ваших данных, но вы должны тщательно изучить все тонкости, чтобы гарантировать, что общая прибыль стоит усилий. При рефакторинге у вас не будет второго шанса, и вам нужно будет заручиться поддержкой всех сопричастных людей, особенно администраторов базы данных. Когда я встречаю таблицу с более чем миллионом строк, в моем списке задач автоматически появляется разбиение – с вопросительным знаком. Но решать, делать ли разбиение, в каждом случае нужно отдельно. Я обычно с большим недоверием отношусь к таблице, которая содержит несколько сотен миллионов строк и не разбита при этом на разделы. Однако некоторые таблицы в диапазоне от миллиона до десятков миллионов строк иногда лучше оставлять неразбитыми.

Разбиение таблиц

Используете ли вы кластерный индекс, превращаете ли вашу таблицу в индекс-таблицу или разбиваете ее на разделы и подразделы, – в любом случае, вы «просто» изменяете физическую структуру базы данных. Для ваших программ таблица остается той же самой, и даже если вас возненавидит несколько человек, которые из-за вас вынуждены работать по выходным, вы, по крайней мере, сможете сделать вид, что для программ это будет прозрачным (тем не менее вам может потребоваться пересмотреть несколько запросов, если вы помните пятую главу). Некоторые другие физические изменения базы данных могут принести значительную пользу, но потребуют небольшой коррекции запросов.

По ряду причин (рис. 7.9) при плохой производительности существуют таблицы с очень широкими строками: при сканировании диапазона в индексе происходит обращение в основном к различным блокам. В этом случае причиной является не порядок строк, поэтому реорганизация таблицы, как мы уже видели, нам не поможет. Причина в том, что, поскольку строки широки, каждая страница или блок хранит очень мало строк. Эта ситуация часто возникает, когда таблица содержит много длинных столбцов типа `varchar`, текстовых столбцов¹, или больших объектов (LOB), особенно если они часто обновляются, вызывая фрагментацию блоков данных. Когда объекты LOB очень велики, они хранятся на специальных страницах, а строки содержат только указатели на эти страницы. Но, поскольку использование преобразования

¹ Столбцы `Clob` в Oracle.

логических адресов в физические может значительно снизить производительность, не очень большие объекты LOB часто хранятся в строке (до 4000 байт в Oracle и 8000 байт в SQL Server; в случае MySQL размер зависит от механизма хранилища). Не нужно иметь в строке много заполненных столбцов по 4000 знаков, чтобы плотность строк в странице таблицы стала очень низкой и оказывала соответствующее влияние на скорость сканирования диапазона.

Возможным решением может стать разбиение широкой таблицы *t* на две таблицы, как показано на рис. 7.10 (круги обозначают столбцы первичного ключа). Идея заключается в том, что длинные столбцы редко принимают участие в запросах, возвращающих много строк, а соединения и дополнительные извлечения добавляют к маленьким результирующим наборам очень небольшие накладные расходы. В таком случае вы можете попытаться выделить в таблицу, которую я переименовал в *t_head*, все узкие проиндексированные столбцы, а также несколько других столбцов, которые могут потребоваться для идентификации результирующего набора. В результате отбора столбцов строки в таблице *t_head* окажутся короткими, плотность – высокой, и мы даже сможем использовать кластерный индекс или разбить таблицу, гарантируя таким образом, что сканирование по индексному диапазону (или полное сканирование таблицы) будет происходить с обращением к небольшому количеству страниц, полных информации.

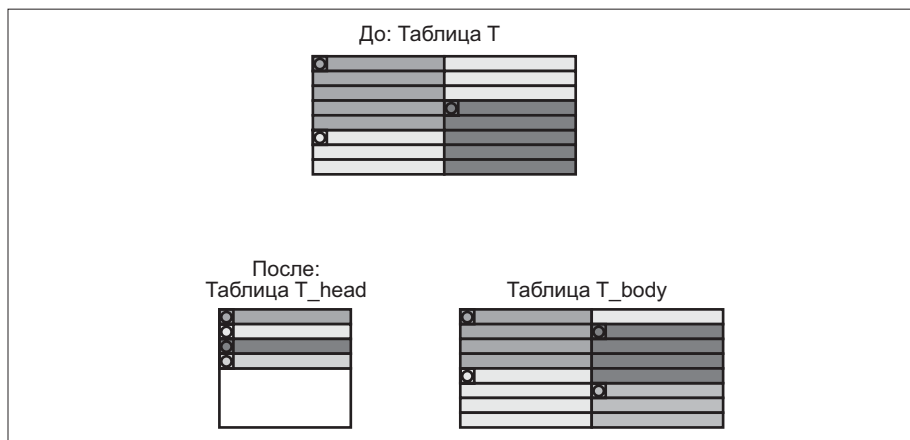


Рис. 7.10. Разбиение таблиц

Все длинные столбцы, а также продублированные столбцы первичного ключа перенесены во вторую таблицу под названием *t_body*. Эта вторая таблица, являющаяся вторичным хранилищем определенного рода, «проиндексирована» по тому же первичному ключу, что и таблица *t_head*.

В этом случае в программу должны быть внесены некоторые изменения, поскольку первичная таблица исчезла:

- все операторы `select` и некоторые операторы `update`, которые обращаются только к столбцам, принадлежащим теперь таблице `t_head` или `t_body`, должны ссылаться на новую таблицу, чтобы мы получили преимущество от новой структуры. Для операторов `update` это может быть слишком сложно. Все зависит от того, как таблицы были первоначально заполнены. Может оказаться так, что при вставке новой строки вы будете присваивать значения только столбцам, которые теперь принадлежат таблице `t_head`. В этом случае нет смысла вставлять в таблицу `t_body` соответствующую строку, которая пуста, за исключением столбцов первичного ключа. Но тогда первое обновление, применяемое к столбцу таблицы `t_body`, должно быть преобразовано в оператор `insert` – путем перехвата исключения, а не предварительной проверки, существует ли строка!;
- все операторы `select`, ссылающиеся на столбцы, которые теперь разнесены между таблицами `t_head` и `t_body`, должны ссылаться на представление, соединяющее таблицу `t_head` с таблицей `t_body` так, чтобы это выглядело как обращение к исходной таблице. Может потребоваться, чтобы это соединение было внешним, поскольку таблица `t_body` может теперь не содержать соответствующей строки для каждой строки в таблице `t_head`;
- как я уже объяснял, в случае вставки оператора `insert` в таблицу `t` будут происходить либо последовательные вставки в таблицы `t_head` и `t_body`, либо немедленные вставки в таблицу `t_head` с последующими (значительно позже) вставками в таблицу `t_body`;
- для операторов `delete` ситуация проще: вам всего лишь придется выполнить удаления из двух таблиц в одной транзакции.

Изменение столбцов

Вы можете внести несколько изменений в столбцы: «поиграть» с их содержимым, разбить столбцы или добавить новые. Как любые физические изменения, внести их не слишком легко, а поскольку логическая структура таблиц меняется, должны быть сделаны параллельные изменения в программах.

Изменение содержимого

Можно поэкспериментировать с атрибутом столбца `null/not null`. Как правило, любой столбец, который должен всегда содержать данные, нужно объявлять как `not null` (в Oracle `null` является значением по умолчанию). Оптимизатору важно знать, что у столбца всегда есть определенное значение, поскольку это влияет на стратегии соединения, а также на оценку подзапросов и количества уникальных элементов. Даже когда столбец может быть незаполненным, имеет смысл использовать значение по умолчанию. Если у вас есть индекс по одному столбцу, который может содержать значения `null`, некоторые продукты, в том числе Oracle, не индексируют значения `null`, а условие, подобное нижеприведенному,

будет обязательно преобразовано (если нет других условий) в полное сканирование, даже если большинство строк содержат значения:

```
and column_name is null
```

Использование значений по умолчанию способствует эффективности поиска по индексу¹.

Иногда успешной оказывается противоположная стратегия: если столбец не содержит ничего, кроме флага Y/N, а вас интересуют только строки, в которых значение равно Y (меньшинство), замена значения, важность которого не очень велика (например, N), на null может сделать индексы и строки более компактными (некоторые СУБД не хранят значения null), а поиск несколько более эффективным (поскольку приходится считывать меньше блоков). Но в большинстве случаев разница окажется столь незначительной, что не будет стоить затраченных усилий. Кроме того, в некоторых системах баз данных отсутствуют механизмы сжатия, что делает такую оптимизацию спорной.

Разбиение столбцов

Многие проблемы, связанные с производительностью, возникают непосредственно из-за отсутствия нормализации баз данных – особенно из-за несоблюдения условий *первой нормальной формы*, которая устанавливает, что столбцы содержат элементарные величины для поиска и что условия должны применяться к полному столбцу, а не к подстроке или к какому-либо подобному разбиению столбца. Конечно, частичные попытки нормализации имеют «привкус отчаяния»: очень часто правильная нормализация означает полную переработку проекта базы данных. Такая серьезная реконструкция может оказаться необходимой, но вряд ли она обрадует тех, кто распоряжается деньгами. К сожалению, половинчатые попытки исправления явных недостатков редко значительно улучшают ситуацию. Предположим, у вас есть столбец под названием name, который содержит имена, отчества и фамилии. Столбцы такого типа превращают даже простой поиск в сложнейшую операцию. Забудьте о сортировке по фамилии (даже если фамилии находятся на первом месте, то наверняка в списке найдутся люди с именами иностранного происхождения, у которых на месте фамилии будет стоять имя), а поиск будет осуществляться по сложным условиям, для которых индексы совершенно бесполезны. Например:

```
where upper(name) like '%DOE%'
```

Несложно догадаться, что в этом случае содержимое столбца должно быть разбито на управляемые, элементарные фрагменты информации,

¹ Если говорить о датах и, как правило, обо всех столбцах, по которым производится поиск по диапазону значений, фиктивные значения, которые заменяют столбцы null, могут дать оптимизатору неверное представление о диапазоне значений, а следовательно, о том, какую часть таблицы придется просканировать.

и, возможно, стоит реализовать принудительный ввод данных в нижнем или верхнем регистре, даже если визуализация потребует новых преобразований. Но тогда возникает вопрос: нужно ли дублировать информацию, сохраняя старые столбцы с целью минимизации изменений в существующих программах, или от старых столбцов нужно избавиться полностью. Большой проблемой при сохранении старого столбца является поддержка данных. Если вы не хотите модифицировать существующие операторы `insert`, то вы либо заполните новые столбцы с помощью триггера, либо определите их (в SQL Server) как вычисляемые столбцы, которые можно индексировать. Не имеет большого значения, вызываете ли вы функцию, которая разрезает строку, из триггера или ее вызов заложен в определении вычисляемого столбца. Вызываемая функция может делать вставки и обновления (относительно) медленно, а автоматическая нормализация строк, вполне возможно, приведет к большому количеству ошибок, если имена не соответствуют весьма простым шаблонам. В результате вы, вероятно, потратите значительно больше времени на попытки исправить функцию преобразования, чем на модификацию операторов `insert` и `update`. Возможно, безопаснее будет не пытаться поддерживать «режим совместимости» при явно неудачной реализации.

Добавление столбцов

В противоположность попыткам нормализации данных существует более популярный вариант действий, заключающийся в денормализации данных. Обычно основным поводом для денормализации является либо устранение соединений (путем распространения через последовательность таблиц ключей, допускающих ярлыки), либо устранение агрегатов путем вычисления значений для мгновенных отчетов. Например, вы можете добавить к таблице `orders` столбец `order_amount`, который будет обновляться суммой каждой строки, ссылающейся на заказ в таблице `order_details`, где будут перечислены все входящие в заказ продукты.

Денормализация может представлять угрозу целостности данных. Рано или поздно ключи, которые дублируются ради избежания соединений, ставят вас в сложное положение, поскольку поддержание ссылочной целостности становится почти невозможным, и у вас появляются строки, которые нельзя ни с чем связать. Мгновенные вычисления безопаснее, но здесь возникает вопрос поддержки: сохранять агрегат в актуальном состоянии при постоянных вставках, обновлениях и удалениях непросто, а искать ошибки в агрегатах сложнее, чем где-либо еще. Более того, если такой случай был пригоден для этого типа денормализации на заре развития баз данных SQL, такие расширения, как добавление фразы `over ()` к стандартным функциям агрегирования, делают его значительно менее полезным.

Наиболее убедительные примеры применения денормализации, которые я встречал в последнее время, были связаны с разбиениями: в больших таблицах был избыточный столбец, который использовался как ключ

раздела. Это интересный гибрид двух приемов, делающий хранилище данных более совместимым с существующими запросами.

Часто денормализация означает попытку совместить две нормализованные схемы. Нормализация и, в частности, сама идея элементарных данных, на которой основана первая нормальная форма, относится к тому, как мы используем данные. Представьте ситуацию, когда по причинам, связанным с производительностью, вы хотите сохранить и агрегаты, и детали. Если вас интересуют и детали, и агрегаты, нет причин поддерживать агрегаты: вы можете вычислять их почти без дополнительных накладных расходов при извлечении деталей. Агрегаты хранят в тех случаях, когда идентифицирован целый класс запросов, для которых нужны только агрегаты. Другими словами, класс запросов, где элементарной частицей является агрегат. Это справедливо не всегда (например, заранее вычисленный агрегат может оказаться очень удобным для замены фразы *having*, которую очень тяжело вычислять сразу), но довольно часто. Денормализация является симптомом противоречия между операционными и отчетными требованиями. Часто лучше признать это и, даже если выделенная база данных поддержки решений¹ избыточна, создать выделенную схему, основанную на материализованных представлениях, а не на с трудом поддерживаемых денормализованных столбцах в транзакционной базе данных.

Материализация представлений

Материализованные представления являются, по сути, не чем иным, как репликацией данных: обычно в представлении хранится тест запроса, который выполняется при каждом запросе к представлению. Создание материализованного представления на практике эквивалентно следующему:

```
create table materialized_view_name
as
select ....
```

В SQL Server представление является материализованным, когда по нему создан уникальный кластерный индекс. После этого все модификации базовых таблиц распространяются на копию. В Oracle есть специальный оператор `create materialized view`, в котором вы среди прочего указываете, как представление должно обновляться: автоматически, то есть каждый раз, когда фиксируется транзакция с участием таблицы, или асинхронно, то есть с помощью задания «обновления», которое СУБД выполняет по расписанию. В последнем случае материализованное представление может быть обновлено либо полностью (вариант режима «отменить и заменить»), либо с приращением, что требует создания таблиц журнала, связанных с базовыми таблицами. Обновление,

¹ Которая может использовать другую технологию, например OLAP cube, или обработку на основе столбцов, как IQ компании Sybase.

которое происходит при фиксации транзакции, требует журнала. MySQL до версии 6.0 включительно не поддерживает материализованные представления, но механизмы, подобные реализованным в Oracle, могут быть разработаны с помощью триггеров (триггеры, которые Oracle использует для управления материализованными представлениями, встроены в ядро Oracle 8i, но в предыдущих версиях использовались обычные триггеры). Вы можете автоматически генерировать эти триггеры, запуская запросы к представлениям `information_schema`.

За все приходится платить, и существование материализованных представлений означает, что все операции, изменяющие базовые таблицы, будут стоить больше и выполняться дольше из-за операций, необходимых для записи, и применения изменений к материализованным представлениям при фиксации, – если только вы не выбрали полное обновление, что может быть жизнеспособным решением, если допустим некоторый временной интервал между данными в материализованном представлении и данными в базовых таблицах и если обновления являются частью ночных пакетных процессов.

Материализованные представления чаще всего применяются в двух случаях:

- Первый случай, который я только что рассмотрел, является вариантом легкого процесса ETL. Если накладные расходы для базовых таблиц приемлемы – и если вам удастся изолировать часть вашего приложения, которая относится в основном к отчетам или поддержке принятия решений, – материализованные представления, которые могут быть проиндексированы для повышения производительности, станут вашими союзниками. Oracle и SQL Server могут даже распознавать запрос, который был использован для создания материализованного представления, и обращаться к заранее созданному результирующему набору вместо того, чтобы исполнять запрос;
- Второй случай – когда один или несколько ваших запросов ссылается на удаленные таблицы на другом сервере. Если вам не требуется синхронное отображение данных, получение локальной копии даст оптимизатору больше свободы в выборе правильного пути. Когда вы обращаетесь к удаленной таблице, вложенные циклы обычно не являются вариантом, поскольку каждая итерация требует обмена данными по сети. Возможно, SQL-машина будет жадно «поглощать» данные через сеть, сохранять их локально (неиндексированными) и наилучшим образом выполнять запрос. Используя материализованное представление, вы можете создать локальную копию, индексировать ее по своему желанию и использовать вложенные циклы, если они окажутся наиболее эффективным способом исполнения запроса.

Разработчики слишком часто рассматривают материализованные представления как панацею. Однажды я посетил встречу с десятком опытных администраторов баз данных, в ходе которой разработчики предлагали

создать для многомерного анализа тринадцать материализованных представлений для таблицы, которая должна была хранить около миллиарда строк. Администраторы баз данных чуть не лишились чувств. Несколько позже я видел, как разработчик использовал материализованные представления для решения проблем, связанных с производительностью запросов, которые ссылались на самую позднюю дату в таблице, предназначенной для хранения данных за два года. После группы материализованных представлений следовал такой шаблон:

```
select *  
from t  
where date_column = (select max(date_column) from t)
```

При ближайшем рассмотрении оказалось, что (большие) таблицы не были разбиты на разделы, а единственными индексами были индексы первичного ключа – по суррогатным ключам.

Я надеюсь, что эта книга продемонстрировала: создание материализованных представлений, несмотря на их несомненную полезность в некоторых случаях, не является первым шагом, который вы должны предпринять при попытках увеличения производительности.

8

Как это работает: практика рефакторинга

*Следи за тем, как здесь мой шаг ведет
К познанию истин, для тебя бесценных,
Чтоб знать потом, где пролегает брод.*

Данте Алигьери (1265–1321)
Божественная комедия,
Рай, песнь вторая, 124–126¹

В этой книге я попытался рассказать о различных способах рефакторинга приложений SQL с низкой производительностью. В этой короткой главе я кратко опишу, как обычно действую, когда меня просят исправить приложение баз данных, не оправдывающее возложенных на него надежд. Я попытаюсь писать конкретно и абстрактно одновременно: конкретно – чтобы показать практику рефакторинга, а абстрактно – чтобы описать суть процессов. Нет необходимости говорить, что подробные обоснования и детальные объяснения вы найдете в предыдущих главах.

Можете ли вы взглянуть на базу данных?

Когда вас просят привести в порядок программу, очень часто оказывается, что никто в точности не знает сути проблемы.

- Если вам повезет, то наиболее серьезные проблемы, связанные с производительностью, уже будут сведены к одному процессу или задаче, а поведение программы с тестовой базой данных будет точно таким же,

¹ Пер. М. Л. Лозинского.

как и с главной рабочей базой данных. Если вам очень повезет, то вам будет помогать человек, хорошо знакомый с функциональной стороной приложения.

В таком случае, вероятно, лучшим решением будет включить трассировку и запустить слишком медленную задачу. От администратора базы данных вам нужно будет получить права, позволяющие адресовать запросы не только таблицам приложения, но и словарным данным таблиц, динамическим представлениям (dynamic performance views) и, возможно, некоторым системным хранимым процедурам. Убедитесь, что файлы трассировки генерируются и что у вас есть свободный доступ к ним. Попытайтесь собрать как можно больше данных: запускаящиеся операторы SQL, время исполнения, процессорное время, количество исполнений, количество обработанных строк и информацию о периодах ожидания, если она доступна. Также постарайтесь получить доступ к коду приложения. Даже если оно написано на языке, с которым вы никогда не работали, логика программы расскажет вам, как комбинируются различные операторы SQL, которые вы собираетесь отслеживать.

- Если информация о проблемах дошла до вас через «испорченный телефон», то есть через нескольких людей в иерархической структуре компании, а проблему удастся быстро воспроизвести на тестовой базе данных, вы должны обратиться к рабочей базе данных. Для этого нужно получить необходимые привилегии. Не нужно требовать полных прав администратора базы данных, но следует заранее подготовить список необходимых прав:

Обязательные привилегии

Право делать запросы к динамическим представлениям и подмножеству словаря данных, описывающему объекты, используемые в приложении.

Крайне желательные привилегии

Право делать запросы к таблицам приложения.

Вам не нужны права на обновление таблиц приложения или на удаление из них строк, даже если медленными являются операторы `update` или `delete`. Вы можете изучить поведение этих операторов, заменив их на операторы `select`. Часто можно диагностировать многие проблемы вообще без запросов к таблицам приложения, причем вы даже сможете предложить некоторые возможные решения для увеличения производительности. Но даже если данные являются конфиденциальными и доступа к ним лично вам не дадут, вполне вероятно, что вам охотно предоставят статистическую информацию о распределениях (и тем охотнее, чем острее стоит проблема). Это позволит вам сгенерировать данные для тестовой базы данных (см. главу 4) и протестировать различные варианты решений, которые пришли вам в голову.

Когда вы не знаете, в чем именно заключается проблема, вам нужно получить как можно больше информации из кэша базы данных. Если

вам доступны средства мониторинга, используйте их. Если специализированной программы нет или она не предоставляет нужной информации, создайте собственную программу. Прочитайте о программе *snarmon.sql* в разделе *Глава 1* приложения А. Если вы предполагаете, что проводить анализ производительности придется несколько раз, постарайтесь создать инструмент, основанный на этом типе запросов. Нет необходимости в изящном интерфейсе пользователя – он может быть любым. И не имеет особого значения, будет ли получен на выходе простой текст, файлы CSV, файлы HTML (которые я обычно использую) или сложные графики. А вот что вам действительно нужно, так это хронологический список операторов SQL, активнее всего использовавших ресурсы в тот период, который вас интересует. Отсортируйте операторы по (общему) времени работы в убывающем порядке, а если хронологическая информация недоступна, то по количеству логических операций ввода-вывода. Если известно процессорное время, его будет интересно сравнить с общим временем. При осуществлении мониторинга базы данных я обычно делаю снимки кэша сервера базы данных каждые десять минут и группирую их по часам. MySQL требует более частого опроса. Когда инструменты мониторинга показывают, что запросы на получение снимка входят в число самых ресурсоемких, это означает, что вы осуществляете слишком активный мониторинг.

Не забывайте собирать и просматривать глобальную статистику. Сравнивая результаты, полученные вашими собственными методами, с общими цифрами, вы сможете оценить достоверность вашего мониторинга. Если вы упускаете свыше 20% или 30% логических операций ввода-вывода, есть опасность, что вы преследуете ложную цель.

Пока идет процесс мониторинга, вам не стоит тратить время впустую. Постарайтесь ознакомиться с приложением. Проверьте схему индексирования, как описано во второй главе. Изучите статистику. Просмотрите информацию о блокировках и проблемах, связанных с конкуренцией. Также может оказаться полезным сравнить статистическую информацию об относительном весе (в виде количества исполнений) и потреблении ресурсов операторами *select* и операторами, изменяющих данные.

«Мертвые» запросы

Случается, что проблемные запросы уже идентифицированы или вы обнаруживаете среди самых частых запросов очень ресурсоемкие. В этом случае вам нужно обратиться к пятой главе.

Проблемные запросы чрезвычайно сложны. Но для паники нет оснований.

- Проверьте реальную природу таблиц, появляющихся в запросе. Если это представления, переместите их в оператор как подзапросы (этот процесс может оказаться рекурсивным) и удалите из этих подзапросов все, что не относится к текущему запросу (глава 3). Опасайтесь

также распределенных запросов: вложенные циклические запросы между далекими таблицами могут убить любой запрос.

- Если у вас не получается модифицировать программу, возможно, вы сможете увеличить производительность, просто переписав представления, которые ссылаются на другие представления, так, чтобы они ссылались только на исходные таблицы.
- Проверьте, не являются ли операторы `update`, `insert` и `delete` медленными из-за блокировок, захваченных другими, более медленными операторами.
- Для операторов `update`, `insert` и `delete` проверьте также, не сработали ли триггеры. Проблема может заключаться именно в этом.
- Удалите все директивы оптимизатора и сократите запрос до его ядра. Каковы входные данные запроса? Где ограничения во фразе `where`? Какие критерии являются действительно избирательными, позволяющими резко уменьшить количество обрабатываемых строк? Если вы не знаете, запустите несколько операторов `group by` (или проверьте статистику, если таблицы слишком велики). Нет ли проблем с распределением данных? Получает ли оптимизатор всю требуемую информацию? Превратите оптимизатор в вашего союзника.
- Хранится ли среди возвращаемых столбцов информация в одном столбце для всех таблиц или столбец является столбцом соединения, общим для нескольких таблиц? Изменится ли что-нибудь, если вернуть информацию из другой таблицы?
- Идентифицируйте таблицы, которые вам нужны для получения данных, таблицы, которые вам нужны для указания условий, и таблицы, которые вам нужны только для соединений. Переместите таблицы, которые вам нужны только для указания условий, в подзапросы. Если они очень велики, а ожидаемый результирующий набор мал, сделайте эти запросы согласованными. В противном случае продумайте применение `in ()` или соединение с подзапросом во фразе `from`. Опасайтесь, что `in (subquery)` добавит неявный `distinct`, который должен стать явным, если подзапрос находится во фразе `from`.
- При работе с подзапросами опасайтесь значений `null`, которые не могут быть ни равны чему-либо еще, ни отличаться от чего-то иного, ни даже быть равными другим значениям `null`. Не забывайте указывать условия `is not null` для столбцов, которые не являются обязательными.
- Будьте осторожны: согласованные и несогласованные подзапросы не должны опираться на различные условия индексирования.
- Какой критерий должен управлять запросом, быть его исходной точкой? Если среди возможных «кандидатов» есть несколько критериев, применяемых к разным таблицам, будет ли возможность изолировать строки-кандидаты каждого запроса, а затем объединить их с помощью соединений или операций над множествами типа `intersect`?

- Имеются ли какие-нибудь определенные пользователем функции? Часто ли они вызываются? Вызываются ли они снова и снова с одними и теми же параметрами? Если они могут вызываться часто, могут ли их результаты быть кэшированными (глава 3)? Каков исходный код этих функций? Можно ли перенести их логику в один запрос (глава 6)? Можете ли вы использовать этот единственный запрос в соединении?
- Есть ли функции, которые могли бы препятствовать использованию индекса? Опасайтесь неявных преобразований типов. Убедитесь, что столбцы и значения на обеих сторонах условий имеют один и тот же тип.
- Находятся ли столбцы композитных индексов в нужном порядке (глава 2)? Не может ли индекс, который выглядит как имеющий неправильный порядок, быть нужен кому-то еще? Вряд ли вы захотите испортить другим квартальный отчет.
- Не забывайте, что обращения к базе данных эффективнее тогда, когда столбец композитного индекса, к которому применяются условия неравенства, находится после столбца, к которому применяются условия равенства.
- Будьте очень внимательны с индексами, поскольку они могут добавить большие накладные расходы к операциям вставки и удаления. Если требуется новый индекс, попытайтесь идентифицировать тот индекс, без которого можно обойтись, поскольку он избыточен или недостаточно избирателен. Постарайтесь сохранять количество индексов постоянным.
- Есть ли какие-либо шаблоны, повторяющиеся на протяжении всего запроса? Есть ли определенные таблицы, обращение к которым происходит несколько раз? Можете ли вы обойтись только одним обращением?
- Характеризуют ли подзапросы особый порядок исполнения запроса через зависимости? Например, зависит ли согласованный подзапрос от значения столбца, что требует исполнения другого подзапроса или соединения? Можно ли сделать этот порядок исполнения менее жестким и дать больше свободы оптимизатору?
- Если в запросе есть операции над множествами, можно ли их вынести за скобки?
- Можно ли заменить какие-либо рефлексивные соединения на функции ранжирования?

Все эти быстрые запросы

Большие, сложные запросы встречаются часто, но еще чаще высокая стоимость связана с большим числом исполнений при низкой единичной стоимости. Важно то, что общая стоимость (единичная стоимость,

умноженная на количество исполнений) значительно больше единичной стоимости. Когда единичные стоимости низки, вопрос заключается не столько в настройке запроса SQL, сколько в алгоритме, как вы видели в шестой главе.

- Каково соотношение между количеством обрабатываемых «бизнес-единиц» (будь то генерирование счетов, покупки, регистрации или что-то иное) и количеством запросов? Является ли оно разумным? Можно ли его уменьшить?
- Каковы аргументы в пользу циклов?
- Можете ли вы заменить «параноидальные» предварительные проверки на криминалистический анализ? Попробуйте упорядочить операторы SQL по количеству исполнений. Близкое количество исполнений часто дает очень хороший ключ к логике, даже если вы не видите кода. В программе SQL нужно сначала делать, а потом проверять, почему что-то не сработало. Если в большинстве случаев проверка проходит успешно, нет необходимости проводить предварительную проверку.
- Каков результат умножения количества запросов к таблице на количество активных сеансов и последующего деления на общее количество строк в таблице? Является ли он удовлетворительным? Сделайте все, что возможно сделать в одном операторе.
- Есть ли что-либо похожее на запрос из функции просмотра, что исполняется слишком часто?
- Когда критерием ввода являются суррогатные ключи (генерируемые системой последовательные числа), есть вероятность, что проблема кроется в алгоритмах, а не в самом запросе, даже если запрос медленный. Где-то должен быть запрос, который возвращает эти суррогатные ключи на основе реального критерия, предоставленного пользователем. Попробуйте соединить оба запроса. Не используйте курсорный цикл.
- Среди быстрых запросов иногда встречаются рекурсивные запросы к словарю данных. Попробуйте выяснить, к какому типу внутренних операций они относятся и нельзя ли избежать этих операций путем оптимизации размера таблицы, индекса или других подобных действий. Если запросы, ссылающиеся на внутренние операции, производятся медленно, работайте в тандеме с администратором базы данных.
- Попробуйте найти запросы `count(*)`. Скорее всего, вы сможете обойтись без них.
- Ищите запросы, исполняющие только простые операции или возвращающие значения, которые можно получить и другими способами. Запрос, который не возвращает данные из таблицы, бесполезен. Например, если приложению нужно узнать время, установленное на сервере, не стоит выделять для этой цели отдельный запрос. Время можно вернуть вместе с «полезными» данными.

- Низкая единичная стоимость не всегда означает минимальную единичную стоимость. Если вы можете модифицировать часто запускаемый запрос так, чтобы он выполнял ту же работу, обращаясь к трем блокам вместо пяти, это приведет к 40%-ному ускорению, что значительно снизит нагрузку на сервер и высвободит ресурсы. Добиться усовершенствований такого рода можно, избегая лишних физических обращений при индексном поиске одним из следующих способов:
 - использовать кластерный индекс или индекс-таблицу, в которой проход по дереву индекса приводит вас к данным строки, а не к индексу строки;
 - поместить все необходимые данные в индекс, избежав таким образом необходимости обращаться к таблице.

Не бывает явно «плохих» запросов

Случай, когда вы можете сосредоточить свое внимание на небольшом количестве «тяжеловесов» SQL, будь то дорогие запросы, исполняемые несколько раз, или простые запросы, исполняемые миллионы раз, можно считать простым. Сложнее ситуация, когда нет одного дорогого запроса, но есть много запросов, каждый из которых потребляет среднее количество ресурсов.

Иногда такая ситуация просто показывает, что вам вряд ли удастся добиться многого (никто не может выигрывать всегда). Но вы можете также столкнуться с ситуацией, когда операторы создаются программой динамически и являются жестко закодированными.

- Проверьте частоту фиксаций и ее связь с количеством бизнес-транзакций, то есть логически независимых единиц работы.
- Возможно, если вы заставите СУБД заменять параметры на константы, это поможет. Однако это не освобождает вас от обязанности указать разработчикам на плохую практику.
- Одним из худших случаев является генерирование фраз `in ()` программой. Можно ли заменить список запросом, который возвращает различные элементы, если такой запрос существует? Если нет, возможно ли использовать какой-либо из методов, описанных во второй главе, для привязывания списка значений?
- Будет ли ситуация иной, если сосредоточиться на планах исполнения, а не на операторах? Найдите для каждого отдельного плана самый дорогой запрос как образец.
- Существуют ли проблемы, связанные с конкуренцией, которые можно было бы решить разбиением на разделы?
- Система, а особенно система баз данных, является последовательностью очередей, которые предоставляют ресурсы конкурирующим клиентам. Если клиентам не требуются одни и те же ресурсы в одно и то же время, увеличение параллелизма увеличит

производительность. Когда одна очередь внутри системы удлинняется, производительность падает.

- Не пытайтесь решать неправильно поставленные проблемы. Снижение конкуренции в неправильном месте не окажет никакого воздействия на производительность.
- На уровне физического хранилища вы можете сделать только две вещи: кластеризовать строки или записать их с разбиением данных. И то и другое может быть полезным, но для противоположных классов проблем. Вы должны решить, какая проблема для вас важнее.
- Физическая реорганизация ваших таблиц может ускорить некоторые запросы путем минимизации количества просматриваемых страниц, но нужно убедиться, что улучшение важного процесса не приведет к ухудшению другого, возможно, еще более важного.

Замечание

Повышение производительности является итеративным процессом. Очень часто за проблемами, которые вы решаете, скрываются другие проблемы.

Пора заканчивать

Иногда у меня возникает смутное ощущение, что попытки улучшения кода SQL – сизифов труд. Сделать нужно много, а растущие базы данных быстро выявляют все недостатки кода, который успешно работал какое-то время. Как я пытался показать, есть много вариантов и много путей, которые вы можете выбрать. Однако лучшие результаты даст изучение и анализ кода. Не полагайтесь на настройку, хотя она тоже имеет свою нишу, а еще меньше полагайтесь на «мастеров» и «ассистентов», предназначенных для автоматизированной оптимизации ваших запросов. Если бы все было так просто, проблем с производительностью не возникало бы никогда. Рефакторинг – это не таинственное искусство, а методика работы. Понимая исходные требования бизнеса и замыслы разработчиков, применяя некоторые простые принципы и навыки работы с SQL, вы действительно можете поднять приложения на впечатляющий уровень. Кроме того, вы вполне можете получать удовольствие от этого процесса.

*Мы могли бы стать всем,
Еще не поздно все изменить.
Я с удовольствием поделюсь своими идеями –
Согласись, мы должны это сделать!*

Пол Уильямс
Песня из мюзикла «Багси Мэллоун»



Сценарии и примеры программ

В тексте книги приведено много примеров кода и фрагментов. Многие из них я взял из модифицированных мной программ (слегка изменив по соображениям конфиденциальности). Поскольку эти примеры не работают без соответствующего массива таблиц и индексов на основе не подлежащих разглашению данных, приводить их бессмысленно.

В эту книгу включены также многие другие программы и примеры, которые я написал специально для нее, а также служебные сценарии для общего использования, осуществляющие запросы к словарю базы данных. Вы можете загрузить эти программы и сценарии с сайта компании O'Reilly, посвященного этой книге, <http://www.oreilly.com/catalog/9780596514976>. В этом приложении описываются вышеупомянутые загружаемые файлы и даются дополнительные комментарии, которые, как я надеюсь, будут вам интересны.

Поскольку большинству читателей интересна только одна конкретная СУБД, я сгруппировал примеры кода по системам, по одному архивному файлу для каждой СУБД. Внутри каждого такого файла я организовал примеры кода по главам (один каталог на главу).

Обратите внимание, что некоторые из программ я написал специально для одной СУБД и их невозможно использовать с другими (например, все примеры седьмой главы написаны исключительно для MySQL). В нескольких случаях вы сможете адаптировать эти примеры для других продуктов с небольшими изменениями. Поэтому не поленитесь посмотреть код для других продуктов, если вас заинтересует программа, которая для вашей СУБД не представлена.

В нижеприведенных описаниях для программ, версии которых для MySQL, Oracle или SQL Server не приведены, я указал в скобках, для какой СУБД они предназначены. Другими словами, если ничего не указано, это означает, что версия программы или сценария имеется для всех трех

продуктов. Обратите внимание, что в случае если сценарий делает запрос к словарю данных, сходство имени не означает, что сценарии вернут совершенно одинаковую информацию, но только один *тип* информации.

Глава 1

Сначала я использовал четыре сценария для создания таблиц в этой главе (эти таблицы появятся снова в некоторых примерах из следующих глав). Если вы хотите запустить программы на вашей собственной тестовой базе данных, вам придется начать с генерирования данных для примеров.

Следующие два сценария создают таблицы и индексы по первичному ключу:

refactoring_big_tables.sql

refactoring_small_tables.sql (включает данные)

А следующие два сценария создают дополнительные индексы. Второй сценарий создает индекс, который заменяет индекс, созданный первым сценарием (подробности в первой главе):

additional_index.sql

additional_index_alternate.sql

Большая таблица `transactions` заполняется с помощью скрипта *GenerateData.java*, требующего наличия файла *database.properties*, который вы должны модифицировать, указав, как подключаться к вашей базе данных.

После того как таблицы созданы и заполнены, вы можете запустить следующие программы:

FirstExample.java

SecondExample.java

ThirdExample.java (требуется Java 1.5 или более поздней версии)

FourthExample.java

FifthExample.java

SixthExample.java

Все эти скрипты используют один и тот же файл *database.properties* как программу генерации данных.

Следующие сценарии зависят от продукта:

dba_analysis.sql (SQL Server) представляют собой запрос, который администратор SQL Server запускает для диагностики проблем (подобно всем административным сценариям, он требует некоторых системных привилегий).

profiler_analysis.sql (SQL Server) представляют собой запрос, который вы можете запустить, если вы сохраняете ту информацию, которую подпрограмма протоколирования SQL Server собирает в таблицу. Он включает в себя функцию T-SQL под названием *Normalize* для нормализации текста оператора.

snapmon.sql (Oracle, SQL Server) является моим базовым инструментом мониторинга. Он собирает статистическую информацию обо всех запросах, которые на данный момент хранятся в кэше, а также о глобальных значениях, которые возвращаются, как если бы они были связаны с пустым запросом. Я разработал инструменты мониторинга на различных языках программирования (сценарий заполнения включен), которые все базируются на этом запросе или небольших его вариациях. Каждый запрос идентифицируется с помощью идентификатора, а конкретное значение присваивается глобальным статистическим данным, что позволяет вам проверить, собрали ли вы осмысленную информацию или потеряли часть операторов. В последнем случае я предполагаю, что вы запускаете модифицированную версию этого запроса, в котором вы идентифицируете запросы по идентификатору в их рабочем проекте.

Значения, которые возвращает *snapmon.sql*, являются кумулятивными. Если вы хотите получить значения в соответствии с интервалами времени, вам придется при каждом запросе вычислять различие от предыдущих результатов.

Вы можете делать это несколькими способами. Можно записывать результаты запросов в файлы, объединять и сортировать два последовательных файла или творчески использовать языки программирования *awk* или *perl* для получения нужных результатов. Я советую вам использовать SQLite (<http://www.sqlite.org>), который представляет API (интерфейс прикладного программирования) для большинства языков программирования и языков сценариев, и выполнить следующие действия:

1. Создать две временные таблицы под именами *snaptmp1* и *snaptmp2*. Эти таблицы должны иметь структуру, соответствующую выводу сценария *snapmon.sql*. Вы должны последовательно вставить в *snaptmp1* и *snaptmp2*. Обратите внимание, что вам не нужно делать запросы к базе данных очень часто. В течение нескольких минут вы обычно сможете получить достаточно ясное представление, что делает ваша база данных.
2. После второго прохода начните вставлять данные в постоянную таблицу с той же структурой, выполняя запрос по такому образцу:

```
insert into ...
select <identifier>, abs(sum(<col1>)), ..., abs(sum(<coln>))
from (select <identifier>, <col1>, ... <coln>
      from snaptmp1
      union all
      select identifier, -1 * <col1>, ..., -1 * <coln>
      from snaptmp2) x
```

```
group by <identifier>
having sum(...) <> 0
```

Вы можете внести дальнейшие усовершенствования, выбрав только верхние запросы (к которым обязательно должен принадлежать пустой «глобальный запрос»). Вы можете также создать таблицу для хранения текста операторов. При появлении новых идентификаторов вы должны получить текст соответствующего оператора из базы данных. Я не хочу подавлять вашу фантазию, предлагая слишком много вариантов, но вы можете собирать и другую информацию (например, периоды ожидания) и создавать графики, которые покажут загрузку.

На тот момент, когда писались эти строки, для MySQL не существовало ничего подобного, так что для этого продукта могут быть более подходящими другие средства мониторинга. Тем не менее, в зависимости от используемого хранилища, некоторая полезная информация может быть извлечена из `information_schema.global_status` (параметры `COM_xxx` и `INNODB_xxx`), а также из таблиц `falcon_xxx`, которые дадут вам по крайней мере общее представление. Закончить можно оператором `select` к `information_schema.processlist`, эти запросы надо делать чаще, чем к таблицам Oracle и SQL Server, поскольку он показывает только текущую информацию.

Глава 2

Для создания таблиц во второй главе использованы следующие сценарии:

IndexSelectivity.java проверяет влияние селективности на производительность доступа к индексу.

stats.sql (MySQL, Oracle) показывает некоторые статистические значения, касающиеся таблиц.

indexing.sql перечисляет все таблицы в текущей схеме и сообщает, сколько у них есть индексов, а также другую информацию.

checkparse.sql представляет собой сценарий, сообщающий, много ли операторов анализируется в вашей базе данных, что всегда свидетельствует о жестко закодированных операторах (хотя обратное не совсем верно, поскольку СУБД иногда заменяет параметры на константы, что вы могли увидеть в главе).

Несколько других программ *.java* снабжены обязательным файлом *database.properties*. Следующие три программы тестируют падение производительности при анализе:

HardCoded.java

FirmCoded.java

SoftCoded.java

Четвыре программы пошагово иллюстрируют, как список значений может быть передан в виде единого параметра:

list0.sql

list1.sql

list2.sql

list3.sql

Следующие три программы выбирают строки из таблицы `transactions`. Первая использует размер выборки по умолчанию, вторая принимает размер выборки в виде аргумента командной строки, а третья активизирует потоковый режим MySQL:

DumpTx_default.java

DumpTx.java

DumpTx_Stream.java (только MySQL)

И наконец, следующая программа проверяет влияние фиксации транзакции на производительность:

UpdateTx.java

Глава 3

Для Oracle существуют три различных способа написать функцию, которая подсчитывает `patterns`. Существует также сценарий, который создает тестовую таблицу и последовательно применяет к этой таблице три функции.

function1.sql (Oracle)

function2.sql (Oracle)

function3.sql (Oracle)

test_search.sql (Oracle)

Для Oracle также имеются три функции, проверяющие, соответствует ли дата дню недели. Первая – как недетерминированная функция, вторая – как детерминированная функция, а третья – как детерминированная функция, которая содержит детерминированную функцию после удаления из даты времени:

weekend_day1.sql (Oracle)

weekend_day2.sql (Oracle)

weekend_day3.sql (Oracle)

Простая функция *NextBusinessDay.sql* предназначена для вычисления следующего рабочего дня. Она использует таблицу, созданную сценарием *public_holidays.sql*.

Также для Oracle (в частности) и для MySQL приведены несколько попыток рефакторинга этой функции. Для Oracle это *NextBusinessDay2.sql* и *NextBusinessDay3.sql*, оба этих файла требуют версии Oracle11g или более поздней, чего не требуется для *NextBusinessDay4.sql*:

public_holidays.sql
NextBusinessDay.sql
NextBusinessDay2.sql (MySQL, Oracle)
NextBusinessDay3.sql (Oracle)
NextBusinessDay4.sql (Oracle)

Аналогично, есть простая функция конвертирования валют под названием *fx_convert.sql* и, для Oracle и MySQL, альтернативные версии (для файла Oracle *fx_convert2.sql* требуется Oracle11g или более поздняя версия):

fx_convert.sql
fx_convert2.sql (MySQL, Oracle)
fx_convert3.sql (MySQL, Oracle)

Наконец, один сценарий заменяет функцию простым соединением:

fxjoin.sql

Вот несколько различных режимов представления таблиц, созданных сценариями и программами из первой главы:

v_amount_by_currency.sql
v_amount_main_currencies.sql
v_amount_other_currencies.sql
v_last_rate.sql

Следующий запрос пытается идентифицировать сложные режимы представления, которые могут являться причиной значительного снижения производительности:

complex_views.sql

Глава 4

Для четвертой главы написана функция определения режима просмотра, возвращающая 10 строк, пронумерованных от 0 до 9:

ten_rows.sql

Для SQL Server разработан режим представления, требуемый для последующего генерирования случайных данных, и функция, которая использует этот режим:

random_view.sql (SQL Server)

randuniform.sql (SQL Server)

Две функции, возвращающие случайные числа, не распределенные равномерно в заданном интервале, находятся в этих сценариях:

randexp.sql

randgauss.sql

А если вы хотите посмотреть, на что похоже распределение, вот маленький сценарий, который рисует гистограмму:

histogram.sql

Чтобы сгенерировать строку случайных символов (что может быть полезным для генерирования паролей), служит этот сценарий:

randstring.sql

Сценарий *job_ref.sql* показывает, как сгенерировать имена, соответствующие заранее заданному распределению.

Чтобы сгенерировать имена, которые будут более реалистичными, чем те, которые люди обычно используют в сообщениях электронной почты, чтобы сбить вам различные лекарства и поддельные швейцарские часы, я предлагаю вам посетить страницу <http://www.census.gov/genealogy/www/> и щелкнуть на ссылке, которая приведет вас к списку самых распространенных фамилий. Там должна быть электронная таблица, содержащая тысячу самых распространенных американских фамилий. Загрузите ее и сохраните как файл с расширением *.csv*. В моих сценариях подразумевается, что в качестве разделителя используется точка с запятой (;).

После этого создайте таблицу для загрузки этих данных:

name_ref.sql

Сценарий для MySQL создает таблицу, загружает данные и выполняет их нормализацию и создает индекс. То же самое делает сценарий для SQL Server (использующий служебный файл *.xml*).

Сценарий для Oracle *name_ref.sql* просто создает таблицу. Загрузку данных вы должны сделать с помощью управляющего файла *name_ref.ctl*, который вы можете использовать с SQL*Loader следующим образом:

```
sqlldr <username>/<password> control=name_ref.ctl
```

Затем (по-прежнему с Oracle) запустите *name_ref2.sql* для нормализации данных и создания индекса таблицы.

Сценарий *get_ten_names.sql* генерирует десять случайных американских фамилий.

Для Oracle имеются три дополнительных сценария: *dept_ref.sql* позволяет вам случайным образом выбрать номер подразделения для таблицы emp; *gen_emp.sql* является генератором строк для таблицы emp на чистом

SQL, не дающем действительно случайных данных, а *gen_emp_pl.sql* представляет собой отлично работающую версию на PL/SQL.

Для генерирования случайного текста обратитесь к приложению В.

Сценарий *check* (MySQL и Oracle) является сценарием для оболочки *bash*, который пропускает вывод произвольного запроса через программу *md5sum*.

Сценарий *grysum.sql* является функцией (Oracle) или процедурой (MySQL и SQL Server), вычисляющей контрольную сумму для вывода произвольного запроса. Сценарий *grysum_complicated.sql* делает то же самое для Oracle, но применяет последовательные контрольные суммы к буферам. Он особенно интересен как пример динамического SQL, написанный на PL/SQL.

grysum.sql

grysum_complicated.sql (Oracle)

Глава 5

Для этой главы нет примеров сценариев. Примеры в этой главе взяты из реальных ситуаций.

Глава 6

Большинство примеров этой главы также взяты из реальных ситуаций, но некоторые я написал для этой книги.

Каталог SQL Server содержит файлы для примера разблокировки счетов после платежа. Две функции были написаны на Visual Basic – одна, точно следующая спецификациям, и другая, выполняющая ту же самую задачу с помощью более толкового использования языка SQL:

create_tables.sql (SQL Server)

procs.vb (SQL Server)

create_vb_procs.sql (SQL Server)

Есть также не совсем доделанный пример на PHP, написанный для MySQL, и файл конфигурации, в котором нужно изменить данными о соединении. Первые два примера показывают неправильное и правильное использование *found_rows()* (очень специфично для MySQL):

config.php (MySQL)

sample_bad.php (MySQL)

sample_ok.php (MySQL)

Другой набор программ на PHP показывает, как можно избежать подсчета перед загрузкой данных, когда есть ограничение сверху на количество показываемых строк:

with_count.php (MySQL)

with_count2.php (MySQL)

without_count.php (MySQL)

Наиболее интересным запросом в последней программе является запрос, возвращающий общее количество строк результирующего набора в первой строке. Его версии (как запрос к таблице `transactions`, возвращающий больше 500 строк) есть для всех СУБД (включая MySQL).

top500GBP.sql

Для пользователей Oracle очень простой пример, использующий `emp` и `dept`. Показывает, почему я не люблю `cursor loops`:

cursor.sql (Oracle)

Глава 7 (MySQL)

Для седьмой главы пример кода есть только для MySQL (однако пользователи SQL Server и Oracle не должны забывать про инструмент `Roughbench`, описанный в приложении В).

Таблица, используемая в этих примерах, создана с помощью *fifo.sql*.

Различным программам, имитирующим обработку сообщений, нужны следующие исходные тексты и сопутствующий файл *makefile*:

common.c

common.h

consumer_inno.c

consumer_lock.c

consumer_naive.c

consumer_not_so_naive.c

makefile

Для тестирования одновременных вставок есть два подкаталога, по одному для каждого используемого хранилища. Сценарии, имена которых заканчиваются на `_ai`, используют ключ с автоматическим увеличением, а сценарии, имена которых заканчиваются на `_p`, используют секционированную таблицу. Имеются также сценарии для создания таблиц и сценарии для вставки данных, которые вы должны запустить с `Roughbench`.

InnoDB

- *create_concurrent_insert.sql*
- *create_concurrent_insert_ai.sql*
- *create_concurrent_insert_p.sql*
- *insert_concurrent_insert.sql*
- *insert_concurrent_insert_ai.sql*
- *insert_concurrent_insert_p.sql*

MyISAM

- *create_concurrent_insert.sql*
- *create_concurrent_insert_ai.sql*
- *create_concurrent_insert_p.sql*
- *insert_concurrent_insert.sql*
- *insert_concurrent_insert_ai.sql*
- *insert_concurrent_insert_p.sql*

В

Инструменты

В этом приложении описываются два инструмента, исходные коды которых вы можете загрузить, и которые, как я надеюсь, будут вам полезны. Первый инструмент представляет собой пару программ, `mklipsum` и `lipsum`, которые я вкратце описал в четвертой главе. Вторым инструментом, `Roughbench`, написанный на языке Java, я часто использую в моих тестах, в частности в седьмой главе.

Эти инструменты выпущены по лицензии GPL и поставляются на стандартных условиях.

Программы `mklipsum` и `lipsum`

Программы `mklipsum` и `lipsum` представляют собой инструменты для генерирования случайного текста. Исходный код написан на языке C, я подготовил его с помощью GNU Autotools (хорошо, насколько мог). Исполнимые файлы для Windows также присутствуют (они были портированы с помощью Mingwin). Для обеих программ требуется SQLite3, которые можно загрузить по адресу <http://www.sqlite.org>. Хочу заметить, что программа `mklipsum` содержит нетривиальный код SQL.

Как собрать программы `mklipsum` и `lipsum`

На компьютере *nix после распаковки и разархивирования файлов для сборки программ введите: `./configure`, затем `make` и `make install` (прочитайте файл `INSTALL` с дополнительной информацией и объяснением, как изменить настройки по умолчанию).

Как использовать программы `mklipsum` и `lipsum`

Использование программ `mklipsum` и `lipsum` не представляет никаких сложностей. Программа `mklipsum` подготавливает файл SQLite для

использования программой `lipsum` из текста, который она размечает и анализирует. Текст считывается из стандартного ввода. В файле SQLite хранятся слова в кодировке UTF8, и именно этот набор символов должен быть использован при вводе. Если на вашей системе имеется библиотека *libiconv* для преобразования наборов символов, программа `mklipsum` сможет осуществить конвертирование наборов символов. Доступность или недоступность возможности преобразования наборов символов будет показана на экране после запуска программы.

Программа `mklipsum` может принимать один необязательный параметр (в случае доступности преобразования наборов символов этих параметров может быть два).

Первый параметр является идентификатором файла, который можно использовать для идентификации языка исходного файла. По умолчанию создаваемый файл SQLite называется *lipsum.db*.

Если вы запустите:

```
mklipsum xx < sample_text_file
```

файл SQLite получит имя *lipsumxx.db*. Эта возможность предназначена для генерирования случайного текста со словами из разных языков. Вы можете использовать код ISO языка для идентификации различных файлов.

Второй параметр, если преобразование наборов символов доступно, указывает набор символов, используемый в исходном файле. Обратите внимание, что вы не можете указать только набор символов, он обязательно должен следовать за идентификатором файла.

Я тестировал `mklipsum` и `lipsum` с различными языками, использующими латинский алфавит (с расширениями). Возможно, для языков, основанных на кириллице, потребуется внести некоторые изменения в программу, и, скорее всего, для таких языков, как арабский или иврит, не говоря уже о китайском или японском, программу потребуется частично переписать (программы пытаются сохранить исходный регистр символов, но следуют западным правилам и, например, делают прописными первые буквы слов, которые стоят после точки).

Вот пример, в котором используются несколько глав из «Левиафана» Томаса Гоббса:

```
$ ./mklipsum < leviathan_sample.txt
mklipsum $Revision$ with charset conversion
-- Reading input ...
-- Loading vocabulary ...
--- 770 words of length 1 analyzed
--- 761 words of length 2 analyzed
--- 800 words of length 3 analyzed
--- 614 words of length 4 analyzed
--- 474 words of length 5 analyzed
--- 312 words of length 6 analyzed
--- 237 words of length 7 analyzed
--- 152 words of length 8 analyzed
```

```

--- 119 words of length 9 analyzed
--- 72 words of length 10 analyzed
--- 43 words of length 11 analyzed
--- 22 words of length 12 analyzed
--- 21 words of length 13 analyzed
--- 5 words of length 14 analyzed
--- 1 words of length 15 analyzed
--- 1 words of length 16 analyzed
--- 1 words of length 19 analyzed
-- Vocabulary: 1029 distinct words loaded
-- Loading lengths relationships ...
--- .....
--- .....
-- Indexing
$

```

Чтобы сгенерировать случайный текст, вам нужно запустить программу `lipsum`, которая может принимать в качестве параметра идентификатор файла, сообщающий ей, какой файл SQLite нужно использовать.

```
$ ./lipsum
```

Speech Thy Eye up with serves which no Businesse; can of is therefore up can false a Joynts happen, he late Marcus, up any motion But Eyes of hath Species; up my Cold which my How First Voyces my are up them also exercise which a Death are any of parts if Circumstances; Image whereas els waking is any must, Sees For office is am cease. Ever say. Prognostiques many course,one cause I Speech mind of For removed of though. Give not onely Anger horse can them, well sight.

Вот пример с использованием другого файла, содержащего текст Эммануила Канта на немецком языке:

```
$ ./lipsum de
```

Sein daß Sitz Begriff Synthesis als, von priori Satz kann vorher vielleicht. Lassen, in komme eine bewußt, des Kreis die ist. Um allen selbst natürlicher a dieser die Prädikat hergibt Zufälligkeit bewußt, priori Verstand a fassen leicht, er Begriffe. Hat wir wird denn desjenigen der empfangen sollten. Seine in der einander ist mithin man korrespondiert, als.

Программа `lipsum` может принимать несколько флагов:

```
$ ./lipsum -?
```

Usage : ./lipsum [flags] [suffix]
This program looks in the current directory for an sqlite file named lipsum<suffix>.db that must have previously been generated by the mklipsum utility.

Flags:

- h, -? :Display this and exit.
- n <count> :Number of paragraphs to generate (default 1)
- w <len>[,<dev>]:Approximate number of words per paragraph.

The program generates paragraphs averaging

```

        <len> words, with a standard deviation
        <dev>. By default, <dev> is 20% of <len>.
        Default for <len>: 500.
        This flag is exclusive of -c.
    -c <len>[,<dev>] :Same as -c, but based on character count
                     instead of word count.
    -l <len>          :Insert a carriage return as soon as possible
                     after having output <len> characters.
                     Default: 65.

```

Roughbench

Программа Roughbench принимает в качестве параметра имя простого сценария SQL, считывает и исполняет его некоторое количество раз или в течение определенного периода времени, при этом может запустить несколько параллельных потоков, выполняющих один и тот же сценарий.

По окончании будет показана статистическая информация.

Как собрать программу Roughbench

Для компилирования программы вам потребуется J2SE Development Kit (JDK) версии 1.5 или более поздней. Загрузить JDK можно по адресу <http://java.sun.com/j2se>.

Как использовать программу Roughbench

Использование программы Roughbench не представляет сложности:

1. Укажите параметры JDBC в файле под названием *roughbench.properties*.
2. Убедитесь, что переменная среды CLASSPATH указана правильно.
3. Подготовьте файл, содержащий (единственный) оператор, который вы хотите запустить. Этот оператор может содержать символы ? для подстановки значений, сгенерированных программой Roughbench.
4. Укажите несколько параметров и как генерировать значения в командной строке.

Командная строка выглядит подобным образом:

```
java [options] roughbench <sqlfile> [generators ... ]
```

Файл roughbench.properties

Файл *roughbench.properties* должен содержать по крайней мере следующие два значения:

```

CONNECT=<JDBC URL>
DRIVER=<Driver package name>

```

Кроме того, если отсутствует принудительная аутентификация и внутри JDBC URL не указано никакого метода аутентификации, вы должны указать:

```
USERNAME=<...>
```

и:

```
PASSWORD=<...>
```

Например, предположим, что мы хотим подключиться как пользователь `bench` с паролем `possward` к базе данных, именуемой `TEST`, запущенной на том же сервере. Тогда мы можем использовать следующее:

Файл `roughbench.properties` для Oracle:

```
CONNECT=jdbc:oracle:thin:@localhost:1521:TEST
USERNAME=bench
PASSWORD=possward
DRIVER=oracle.jdbc.OracleDriver
```

Для MySQL мы можем указать имя пользователя и пароль в URL:

```
CONNECT=jdbc:mysql://localhost:3306/TEST?user=bench&password=possward
DRIVER=com.mysql.jdbc.Driver
```

Указание параметров

Параметры, перечисленные в табл. В.1, указываются в командной строке в соответствии со следующим синтаксисом:

```
-D<option name>=<value>
```

Таблица В.1. Параметры программы *Roughbench*

Название параметра	Комментарий	Значение по умолчанию
COMM	Количество выполнений между двумя успешными фиксациями (игнорируется для операторов <code>select</code>).	1 (фиксация после каждого оператора DML)
LOOPS	Сколько раз должен быть запущен оператор SQL. Без <code>-DTIME=...</code>	1
RATE	Количество выполнений в минуту. Частота является средней; интервал между двумя последовательными выполнениями является случайной величиной.	
TAG	Идентификатор запуска.	Метка времени, формат MonDD-hh:mn
THR	Количество запускаемых параллельных потоков. Все потоки выполняют один и тот же оператор SQL.	1
TIME	Количество минут, в течение которых оператор SQL должен быть запущен. Без <code>-DLOOPS=...</code>	

Генерирование переменных

Вы можете генерировать случайные переменные для привязки их к операторам, указывая в командной строке столько генераторов, сколько имеется символов подстановки в операторе. Генераторы указываются, как показано в табл. В.2.

Таблица В.2. Генераторы

Генераторы	Значение
C<constant>	Постоянное значение; <constant> может быть числовым или строковым значением.
Ra,b,...,c	Стандартное случайное значение из списка (один и тот же элемент может появляться в списке несколько раз, чтобы придать ему больший вес). Значения могут быть любого типа.
Ra-b	Случайное значение в диапазоне от a до b, включая границы (количество символов).
N<avg>,<stddev>	Нормальное (Гауссово) распределение, со средним <avg> и стандартным отклонением <stddev>.
I[<start> [,<step>]]	Значение с приращением. <start> по умолчанию 1, <step> – 1.
S<min>[-<max>]	Строка случайных букв длиной от <min> до <max>, включая границы; <max> по умолчанию имеет то же значение, что и <min>.

Генерирование целых чисел или чисел с плавающей запятой

Генерирование целых чисел или чисел с плавающей запятой зависит от типа первого числа в спецификации генератора. Например:

- R1-250000 сгенерирует целые числа от 1 до 250 000.
- R1.0-250000 сгенерирует числа с плавающей запятой в том же диапазоне.

Генерирование дат

Для генерирования дат вы должны использовать функции вашей СУБД и сгенерировать целые числа, которые будут использованы в функциях работы с датами и в арифметике дат.

Например, вы можете использовать в различных диалектах SQL следующие выражения, связанные с генератором чисел с приращением для генерирования дат в хронологическом порядке за последние триста дней. Поскольку 300 дней равны почти 26 миллионам секунд, вы можете сгенерировать нужное количество дней в интервале до текущей даты.

В MySQL:

```
date_add(date_sub(curdate(), INTERVAL 300 DAY), INTERVAL ? SECOND)
```

В Oracle:

```
sysdate - 300 + ?/86400
```

В T-SQL (SQL Server/Sybase):

```
dateadd(ss, ?, dateadd(dd, -300, getdate()))
```

и т. д.

Вывод

Программа Roughbench выводит информационные сообщения о стандартных ошибках и результаты в стандартный вывод, что облегчает перенаправление результатов в файл.

выводимая информация содержит следующее:

- название программы, версию и сведения об авторских правах и лицензировании;
- справку о параметрах командной строки;
- текст запущенного запроса.

Результаты выводятся следующим образом:

```
tag<tab>filename<space>thread#<tab>tenth of second<tab>count
```

tag представляет собой значение, указанное в параметре `-DTAG=...` командной строки. Значением по умолчанию является метка времени, например Jun11-15:42.

filename представляет собой имя файла, содержащего оператор SQL.

thread# является числом (начиная с 0), идентифицирующим конкретный поток.

tenth of second и *count* сообщают, сколько операторов было выполнено за какое время. Значение 0 для *tenth of second* означает меньше 0,1 секунды, значение 1 соответствует интервалу между 0,1 и 0,2 секунды и т. д.

Например, следующий вывод означает, что при запуске 23 октября в 4 часа дня поток 3 выполнил сценарий `insert.sql` 15 874 раз, из которых пять раз время выполнения было между 0,1 и 0,2 секунды, а в остальных случаях меньше 0,1 секунды:

```
Oct23-16:00 insert.sql 3 0 15869
Oct23-16:00 insert.sql 3 1 5
```

На экран могут быть выведены и дополнительные строки. Например, при выполнении в циклическом режиме общее время выполнения требуемого числа повторений цикла будет показано следующим образом:

```
tag<tab>filename<space>thread#<tab>elapsed (ms)<tab>time
```

Если в некоторых случаях выполнение оператора окажется неудачным, количество успешных и неудачных выполнений будет показано следующим образом:

```
tag<tab>filename<space>thread#<tab>OK<tab>count
```

```
tag<tab>filename<space>thread#<tab>KO<tab>count
```


Алфавитный указатель

Символы

@@CPU_BUSY, 52
@@TIMETICKS, 52
@@TOTAL_READ, 52
@@TOTAL_WRITE, 52

А

AboveThreshold(), функция, 22, 23, 30, 31, 32, 33, 41
ACID, свойство, 267
ASH, 50
Automatic Workload Repository (AWR), 50
AWR (Automatic Workload Repository), 50

В

bcp (SQL Server), 171
bit_xor(), функция, 181
BULK INSERT, команда (SQL Server), 158

С

checksum_agg(), функция, 176
checksum table, команда (MySQL), 176
checksum(), функция, 176
CLASSPATH, переменная среды, 315
coalesce(), функция, 228
commit, оператор, 249
Convert(), функция, 22, 23, 24, 33
count(), функция, 236, 299
create table, оператор, 181
create view, оператор, 136, 137, 139, 141
CURSOR_SHARING, параметр (Oracle), 91

Д

datetime, тип данных, 67, 114
date, тип данных (Oracle), 111
dba_analysis.sql, сценарий, 303

dbcc show statistics, команда, 67
dbms_crypto, пакет (Oracle), 176
dbms_sql, пакет (Oracle), 177
dbms_utility, пакет (Oracle), 176
delete, оператор
 обсуждение рефакторинга, 297
 разбиение таблиц, 287
 функциональное сравнение, 168
dept_ref.sql, сценарий, 308
DTD (document type definition), 166

Е

ETL (Extract/Transform/Load), 280, 292
event 10046 (Oracle), 26
except, оператор (SQL), 173, 174
executeQuery(),
 функция, 22, 23, 24, 33, 34
executeUpdate(), функция, 23, 36
explain, команда, 137
Extract/Transform/Load (ETL), 280, 292

Ф

fifo.sql, сценарий, 310
fn_trace_gettable(), функция (SQL Server), 56
found_rows(), функция, 237
from, фраза
 внешние соединения, 204
 обсуждение рефакторинга, 297
 переработка запросов, 176, 180
 повторяющиеся шаблоны, 196
 подзапросы, 206
 приведение в порядок, 192
FxConvert(), функция
 преобразования, 127

Г

gen_emp_pl.sql, сценарий, 308
gen_emp.sql, сценарий, 308
general_log, переменная (MySQL), 54
GenerateData.java, сценарий, 303

get_hash_value(), функция, 176
GNU Autotools, 312
Gnu Statistical Library (GSL), 164
greatest(), функция, 213
group by, фраза, 80, 297
GSL (Gnu Statistical Library), 164

Н

hashbytes(), функция, 176
HashMap, класс, 33
hash(), функция, 176
having, фраза, 138

I

IndexSelectivity.java, сценарий, 305
information_schema.global_status
(MySQL), 83
information_schema.processlist
(MySQL), 49
insert, оператор
 группировка, 100
 обсуждение рефакторинга, 297
 разбиение таблиц, 287
 функциональное сравнение, 168
instr(), функция, 108, 109
in, фраза, 94

J

J2SE Development Kit (JDK), 315
Jboss, сервер приложений, 56
JDBC
 подготовленные операторы, 23
 поддержка трассировщика рбспу, 56
JDK (J2SE Development Kit), 315

L

least(), функция, 213
length(), функция, 107, 108, 109, 134,
 144, 145
lipsum, инструмент, 312, 313, 314
load data infile, оператор, 100
LOAD, команда (MySQL), 158
LOB (большие объекты), 287

M

MD5, 171, 175
md5(), функция, 176
minus, оператор (Oracle), 173, 174

min(), функция, 197
mklipsum, инструмент, 312, 313, 314
MySQL
 count(), функция, 236
 date, значения, 67
 генерирование случайных
 данных, 155
 генерирование строк, 160
 детерминированные функции, 111
 динамические представления, 49
 исходная ситуация для примера, 25
 кластерные индексы, 80
 контрольные суммы, 181
 материализованные
 представления, 291
 механизм хранения данных
 InnoDB, 265
 мониторинг баз данных, 296
 определение проблем синтаксического
 разбора, 88
 переменные сеанса, 133
 поддержка LOB, 287
 пример календарной функции, 121
 пример функции преобразования, 128
 рефакторинг представлений, 144
 сравнение увеличения
 скорости, 28, 34, 35, 37
 традиционная настройка SQL, 25
 уровень изоляции repeatable read, 265
 фильтрующие представления, 139
mysqldump, инструмент, 171
MySQL Proxy, 56
mysqslsl, инструмент, 56

N

NextBusinessDay(), календарная
 функция, 119

O

optimizer_max_permutations, параметр
 (Oracle), 186
Oracle
 count(), функция, 236
 date, значения, 67
 event 10046 level 8, 26
 order by, фраза, 81
 snapshot too old, ошибка, 265
 генерирование строк, 160
 детерминированные функции, 111
 запросы к динамическим

- представлениям, 49
- индексы на основе битовых карт, 79
- исходная ситуация для примера, 25
- материализованные
 - представления, 291
- определение возможных
 - улучшений, 43
- определение вопросов синтаксического
 - разбора, 81
- определение проблем синтаксического
 - разбора, 88
- поддержка LOB, 287
- поиск по индексу, 64
- пример календарной функции, 119
- пример функции преобразования, 127
- рефакторинг представлений, 145
- сравнение увеличения
 - скорости, 28, 34, 35, 37
- фильтрующие представления, 139

P

- pbspy, трассировщик, 56
- PARAMETERIZATION, параметр (SQL Server), 91
- Parameters, метод (SqlCommand), 90
- profiler_analysis.sql, сценарий, 303

Q

- qrysum.sql, сценарий, 309

R

- rand(), функция, 152
- rank(), функция, 80
- READ_COMMITTED_SNAPSHOT,
 - параметр базы данных, 265
- Roughbench,
 - инструмент, 310, 315, 316, 318
- rowid (Oracle), 270
- row_number(), функция, 80

S

- select, оператор
 - подзапросы, 204
 - разбиение таблиц, 287
 - фильтрация, 211
 - функциональное сравнение, 168
- Service Profile Identifier (SPID) (SQL Server), 55
- setDate(), функция, 22, 23, 24, 34, 36

- setInt(), функция, 22, 23, 36
- setLong(), функция, 22, 23
- SHA1, 176
- snapmon.sql, сценарий, 296, 304
- snapshot too old, ошибка, 265
- sp_describe_cursor_columns, хранящая
 - процедура (SQL Server), 177
- sp_executesql, хранящая процедура
 - (SQLServer), 177
- sp_helpstats, хранящая процедура
 - (SQLServer), 67
- SPID (Service Profile Identifier)
 - (SQLServer), 55
- sp_trace_filter, хранящая процедура
 - (SQLServer), 55
- SqlCommand, класс, 90
- SQLite, 57, 164, 304, 312, 313, 314
- SQL*Loader, утилита, 158
- sql_log_off, переменная сеанса, 55
- SqlPipe, объект, 100
- SQL*Plus, утилита, 66
- SQL Profiler (SQL Server), 27, 56
- SQL Server
 - count(), функция, 236
 - date, значения, 67
 - SQL Profiler, 27, 56
 - версии строк, 264
 - генерирование строк, 160
 - глобальные счетчики, 52
 - детерминированные функции, 111
 - динамические представления, 49
 - исходная ситуация для примера, 25
 - кластерные индексы, 80
 - контрольные суммы, 181
 - материализованные
 - представления, 291
 - определение проблем синтаксического
 - разбора, 88
 - поддержка LOB, 287
 - поиск по индексу, 64
 - пример календарной функции, 120
 - пример функции преобразования, 127
 - проблемы синтаксического разбора, 88
 - рефакторинг представлений, 145
 - сравнение увеличения
 - скорости, 28, 34, 35, 37
 - фильтрующие представления, 139
 - функции генерирования случайных
 - значений, 152, 155
- SQL Server Integration Services, 100
- Statspack, 50

stats.sql, сценарий, 305
substr(), функция, 107, 108, 112, 113, 119, 123
Sybase Open Server, 56
sys.dm_exec_cached_plans (SQL Server), 53
sys.dm_exec_query_stats (SQL Server), 27, 34, 49, 52, 53
sys.dm_exec_requests (SQL Server), 49
sys.dm_os_performance_counters (SQL Server), 52

T

tkprof, инструмент (Oracle), 56, 57
trunc(), функция, 113, 114, 122, 124
T-SQL, 131

U

update, оператор
 обсуждение рефакторинга, 297
 разбиение таблиц, 287
 функциональное сравнение, 168

V

v\$session (Oracle), 49
v\$sql_plan_statistics (Oracle), 53
v\$sqlstats (Oracle), 49, 52, 53, 57
v\$ssystat (Oracle), 82

W

WebLogic, сервер приложений, 56
WebSphere, сервер приложений, 56
where, фраза
 медленные операторы, 184
 обсуждение рефакторинга, 297
 повторяющиеся шаблоны, 196
 подзапросы, 206
 представления, 139
with, фраза, 214

A

агрегаты
 денормализация, 291
 контрольные суммы, 180
 упрощение, 214
 упрощение конструкций, 225
алгоритм
 MD5, 171, 175

SHA1, 176
анализ составных частей, 196
атрибут null/not null, 288

Б

базовые столбцы, 191
базовый запрос
 with, фраза, 214
 анализ составных частей, 196
 идентификация, 190
 подзапросы, 204
 ранняя активизация фильтрации, 209
 соединение операторов
 объединения, 215
 упрощение агрегатов, 214
 упрощение условий, 211
 устранение повторяющихся шаблонов, 196
базы данных
 параллелизм, 280
 рефакторинг доступа, 13, 31, 295
блоки (Oracle), 49, 63
блокировки
 конкуренция за ресурсы, 253, 256
 множественные очереди, 271
 уровни изоляции, 263, 265
блочное тестирование, 167
большие объекты (LOB), 287

В

версии строк, 265
вложенные циклы, 216
внешние ключи
 базовые столбцы, 191
 индексирование, 70
внешние соединения
 значения null, 204, 206
 представления, 142
внутренние соединения, 142
вопросительные знаки, 23
временные таблицы
 изменчивость, 69
 списки, 99
время обслуживания, 256, 258
время ожидания, 44
выделение пространства
 производительность, 62
 сериализация, 272
выражения, 81
вычисляемые столбцы, 81, 107

Г

генерирование случайного текста, 166
генерирование случайных чисел, 64, 152
генник, Джонатан, 10
глобальные счетчики, 52

Д

данные XML, 166
денормализация, 290
детерминированная идентификация даты, 111
детерминированные функции
определенные, 110
предосторожности
при использовании, 111
динамические представления, 49
динамический SQL, 177
директивы оптимизатора, 184, 297
древовидные структуры, 70

Ж

жестко закодированные операторы
замена на мягко закодированные, 89
обработка SQL-машиной, 44
определенные, 81
проблемы синтаксического разбора, 88

З

запросы
анализ, 190
вложенные циклы, 216
обсуждение рефакторинга, 296
перестроение, 216
приведение в порядок фразы
from, 192
соединения слиянием, 217
хеш-соединения, 217
значения null
внешние соединения, 204, 206
игнорирование функциями, 197
индексы, 288
обсуждение рефакторинга, 298
оптимизаторы, 65

И

избирательность, 63

извлечение

всех данных за один прием, 234
рассмотрение
производительности, 100

индексы

значения null, 288
избирательность, 63
кластерные, 80, 284, 301
композитные, 73
обзор, 69
обсуждение рефакторинга, 298
одностолбцовые, 73
первичный ключ, 285
по выражениям, 81
по вычисляемым столбцам, 81
порядок строк, 283
проверка адекватности, 62
производные от дизайна базы данных, 69
рассмотрение доступа, 63
рассмотрение
производительности, 64, 70
сканирование диапазона, 30, 68
составные, 298
таблицы в схемах, 71
типы, 69
в виде В-дерева, 70
на основе битовых карт, 79
на основе функций, 81
по первичному ключу, 285
производительности, 70

инструменты

lipsum, 312, 313, 314
mklipsum, 312, 313, 314
Roughbench, 310, 315, 316, 318

исполнение, 43**К**

Кайт, Том, 174
календарная функция, 119
кластерные индексы, 80, 284, 301
кодировка UTF8, 313
композитные индексы
обсуждение рефакторинга, 298
проблемы, 73
конкурентный доступ
уменьшение критических частей, 270
уровни изоляции, 263
конкуренция
как узкое место, 282
параллелизм, 270, 278

- первичные ключи, 274
- разбиение на разделы, 274, 280, 282
- структура базы данных, 256
- счетчики в таблицах, 271
- контрольные суммы, 171, 175, 180
- концепция тестирования
 - блочное тестирование, 167
 - генерирование случайного текста, 166
 - генерирование строк, 160
 - генерирование тестовых данных, 149
 - ограничения сравнения, 182
 - подгонка под существующие распределения, 156
 - приближенное сравнение, 168
 - размножение строк, 150
 - сравнение таблиц и результатов, 169
 - функции генерирования случайных значений, 152
 - целостность на уровне ссылок, 164
- косметические столбцы, 191

Л

- логические считывания
 - измерение рабочей производительности, 52

М

- масштабируемость, 278
- материализованные представления, 291
- медленные запросы, 184
 - анализ, 190
- механизм хранения данных
 - InnoDB, 265, 274
- множественные очереди, 271
- мягко закодированные операторы
 - замена жестко закодированных, 89
 - определенные, 89

Н

- несогласованные подзапросы, 206, 297
- нормальная форма, 289

О

- обработка
 - изолирование проблемных областей, 270
 - конкуренция за ресурсы, 253, 257
 - на стороне клиента, 55
 - на стороне сервера, 54, 82

- параллелизм, 280
- исключений
- SQL-мышление, 223
- реструктуризация кода, 229
- регистрация на стороне клиента, 55
- операторы SQL, 82
 - ошибка snapshot too old, 265
 - SQL-мышление, 223
 - причины использования циклов, 248
- объединения
 - повторяющиеся шаблоны, 196
 - представления, построенные как, 143
 - сложные запросы, 196
 - соединение, 215
- одностолбцовые индексы, 73
- операторы
 - жестко закодированные, 44, 81, 88
 - медленные, 184
 - мягко закодированные, 89
 - подготовленные, 23, 85
 - рекурсивные, 44
- операторы SQL
 - вывод в файлы трассировки, 53
 - избавление от циклов, 245
 - категории, заслуживающие настройки, 51
 - настройка, 219
 - обработка на стороне сервера, 82
 - объединение, 224
 - переработка, 135
 - планы исполнения, 53
 - принципы написания, 220
 - проверка данных, 176
 - сравнение контрольных сумм, 171, 175
 - уменьшение количества, 244
 - управляющие структуры, 224
 - функциональное сравнение, 168
 - хранимые процедуры, 50
- операторы объединения
 - обсуждение рефакторинга, 298
 - повторяющиеся шаблоны, 196
 - сложные запросы, 196
 - соединение, 215
- оптимизаторы
 - диапазон значений, 65
 - значения, 65
 - основы поиска, 62
 - планы исполнения, 187
 - проблемы синтаксического разбора, 88
 - проблемы с производительностью, 183

- рассмотрение информации, 63
- сложности, 186
- функциональность, 186
- оценка
 - анализ собранного материала, 58
 - выбор подхода, 40
 - вывод операторов в файлы
 - трассировки, 53
 - запросы к динамическим представлениям, 49
 - использование файлов
 - трассировки, 56
 - обзор примера, 21
 - определение возможных улучшений, 42
 - пример рефакторинга, 31
 - промежуточный вариант ведения журнала, 56
 - регистрация на стороне клиента, 55
 - регистрация на стороне сервера, 54
 - сравнение решений, 37
 - традиционная настройка SQL, 25
- очереди
 - множественные, 271

П

- пакетные операции, 100
- параллелизм
 - изолирование проблемных областей, 270
 - конкуренция, 270, 278
 - обсуждение рефакторинга, 300
 - синхронизация, 264
 - СУБД, 280
 - увеличение количества поставщиков услуг, 261
- первая нормальная форма, 289
- первичные ключи
 - базовые столбцы, 191
 - индексирование, 69
 - конкуренция, 274
 - ограничения, 69
 - суррогатные ключи, 270
 - таблицы MyISAM, 274
- переменные
 - массив, 133
 - передача списков, 96
 - сеанса, 131
- переменные-массивы, 133
- переменные сеанса
 - ограничения T-SQL, 131

- поддержка в MySQL, 133
- планы исполнения
 - директивы оптимизатора, 184
 - кэш операторов, 53
 - определение, 184
 - стабильность плана, 187
- подготовленные операторы
 - JDBC, 23
- подзапросы
 - from, фраза, 206
 - select, оператор, 204
 - where, фраза, 206
 - минимизация, 211
 - написание, 193
 - несогласованные, 206, 298
 - обсуждение рефакторинга, 296, 298
 - повторяющиеся шаблоны, 196
 - скалярные, 203
 - согласованные, 206, 298
- подход с единственным запросом, 39
- пользовательские функции
 - категории, 106
 - обсуждение рефакторинга, 298
 - реструктуризация кода, 229
- пороговые значения (проверка транзакций)
 - код генерирования транзакций, 28
 - проверка транзакций, 39
 - рефакторинг, 31
 - традиционная настройка SQL, 25
- поставщики услуг
 - реорганизация обработки, 257
 - увеличение количества внутри приложения, 261
- представления
 - динамические, 49
 - материализованные, 291
 - рассмотрение производительности, 136
 - рефакторинг, 144
 - фильтрация, 138
 - функциональность, 136
 - хранимые процедуры, 105
- привилегии, 295
- программное обеспечение контроля версий, 265
- процедуры, 106
- процедуры изменения базы данных, 106

Р

- разбиение на разделы
 - MyISAM, 278

- денормализация, 290
- конкуренция, 274, 280
- опасности, 282
- распределение чисел, 64, 65
 - равномерное, 64
- регистрация
 - на стороне сервера, 54
 - операторов в файлы трассировки, 54
 - промежуточный вариант, 56
- режим автоматической фиксации, 103, 264
- рекурсивные операторы
 - обсуждение рефакторинга, 299
 - определенные, 44
- реорганизация обработки
 - изолирование проблемных областей, 270
 - конкуренция за ресурсы, 257
 - множественные очереди, 271
 - параллелизм, 280
 - польза, 282
- репликация данных, 291
- реструктуризация кода
 - coalesce(), функция, 228
 - count(), функция, 236
 - избавление от циклов, 245
 - извлечение всех данных за один прием, 234
 - изменение логики, 235
 - служебные функции, 229
 - уменьшение количества операторов, 244
 - упрощение с помощью агрегатов, 225
- ресурсы
 - конкуренция, 253, 257
- рефакторинг
 - доступа к базам данных, 295, 296
 - доступа к базе данных, 31
 - доступ к базам данных, 13
 - подход с единственным запросом, 39
 - польза, 14
 - представления, 144
 - пример пороговых значений, 31
 - причины, 11
 - проблемы очередей, 300
 - служебные функции, 229
 - функций, 298

С

- связанные переменные, 90
- синтаксический разбор

- вопросы исправления, 91
- определение проблем, 88
- потери производительности, 88
- синхронизация
 - вызовы базы данных, 280
 - материализованные представления, 292
 - параллелизм, 264
 - сериализация, 272
- скалярные подзапросы, 203
- сканирование диапазона
 - для индексов, 68
- скорость получения данных, 256, 258
- согласованные подзапросы, 206, 297
- соединение слиянием, 217
- соединения
 - внешние, 142
 - внутренние, 142
 - обсуждение рефакторинга, 298
 - рассмотрение запросов, 217
 - слиянием, 217
 - хеш-соединения, 217
- списки
 - временные таблицы, 99
 - передача как переменных, 95
- среднее значение, 154
- стабильность плана, 187
- стандарт ISO, 263
- стандартное отклонение, 154
- статистика
 - определенная, 63
 - проверка, 62
- Стефанетти, Марко, 174
- столбцы
 - вычисляемые, 81, 107
 - изменение содержания, 288
 - обсуждение рефакторинга, 297
 - разбиение, 289
- страницы
 - извлечение данных в, 63
 - ссылки на операторы, 49
- строки
 - подсчет шаблонов, 107
 - разбиение, 96
- суррогатные ключи, 270, 299
- схемы, 71

Т

- таблицы
 - без уникальных индексов, 72
 - вложенные циклы, 216
 - временные, 69, 99

- индексирование схемы, 71
- конкуренция, 271
- неиндексированные, 72
- обсуждение рефакторинга, 298
- приведение в порядок фразы `from`, 192
- причины использования циклов, 248
- разбиение, 287
- с единственным индексом, 72
- с многими индексами, 72
- сравнение, 169
- строки по шаблону, 96
- типы, 192
- таблицы MyISAM, 261, 274
- транзакции
 - откат, 252
 - рассмотрение
 - производительности, 102
 - циклы, 253

У

- узкие места, 277, 282
- управляющие структуры
 - `coalesce()`, функция, 228
 - извлечение, 234
 - изменение логики, 235
 - исключения, 229
 - упрощение с помощью агрегатов, 225
- уровни изоляции, 263
 - `read committed`, 264
 - `read uncommitted`, 263
 - `repeatable read`, 265
 - `serializable`, 266

Ф

- файлы трассировки
 - вывод операторов, 54
 - использование, 56
- Фаулер, Мартин, 10
- фильтрация
 - базовые столбцы, 191
 - представления, 138
 - ранняя активизация, 209
- фрагментация блоков данных, 286
- функции
 - детерминированные, 110, 111
 - пользовательские, 106
 - улучшение, 135
 - чисто вычислительные, 107
- функции поиска
 - извлечение имен сотрудников, 114

- определенные, 107
- пример календарной функции, 119
- пример функции преобразования, 127
- рассмотрение производительности, 116
- функции преобразования, 127
- функция
 - `AboveThreshold()`, 22, 23, 30, 31, 32, 33, 41
 - `bit_xor()`, 181
 - `checksum()`, 176
 - `checksum_agg()`, 176
 - `coalesce()`, 228
 - `Convert()`, 22, 23, 24, 33
 - `count()`, 236, 299
 - `executeQuery()`, 22, 23, 24, 33, 34
 - `executeUpdate()`, 23, 36
 - `fn_trace_gettable()` (SQL Server), 56
 - `found_rows()`, 237
 - `get_hash_value()`, 176
 - `greatest()`, 213
 - `hash()`, 176
 - `hashbytes()`, 176
 - `least()`, 213
 - `md5()`, 176
 - `min()`, 197
 - `rand()`, 152
 - `rank()`, 80
 - `row_number()`, 80
 - `setDate()`, 22, 23, 24, 34, 36
 - `setInt()`, 22, 23, 36
 - `setLong()`, 22, 23
 - пользовательская, 229
 - служебная, 229

Х

- хеш-соединения, 217
- хранимые процедуры
 - выполнение оператора, 50
 - контрольные суммы, 177
 - представления, 105
 - проверка данных, 177

Ц

- целостность на уровне ссылок
 - денормализация, 290
 - случайные данные, 164
 - таблицы MyISAM, 274
- цепи Маркова, 166
- циклы
 - анализ, 250

вложенные, 216
избавление, 245
причины использования, 248

Ч

частота фиксации, 300
чисто вычислительные функции, 107

Ш

шаблоны
обсуждение рефакторинга, 298
подсчет в строках, 107
устранение повторяющихся, 196

Э

экспоненциальное распределение, 152