НАСТРОЙКА ПРИЛОЖЕНИЙ БАЗ ДАННЫХ

Б. А. НОВИКОВ, Г. Р. ДОМБРОВСКАЯ

Функциональность и настройка системы

Модель приложения и схема базы данных: логическая структура данных и производительность, выбор структур хранения

Короткие запросы: использование индексов, выбор критериев селекции

Настройка больших запросов: управление полным просмотром

Транзакции и производительность

Параллельные системы: критерии качества, уровни параллелизма, размещение данных

Настройка сервера баз данных

bhv

Б. А. Новиков
Г. Р. Домбровская

НАСТРОЙКА ПРИЛОЖЕНИЙ БАЗ ДАННЫХ

Рекомендовано УМО по образованию в области инновационных междисциплинарных образовательных программ в качестве учебного пособия для студентов вузов, обучающихся по специальности "Математическое обеспечение и администрирование информационных систем"

Санкт-Петербург
«БХВ-Петербург»
2006

УДК 681.3.06(075.8)
ББК 32.973.26-018.2я73
Н73

Новиков Б. А., Домбровская Г. Р.

Н73 Настройка приложений баз данных. — СПб.: БХВ-Петербург, 2006. — 240 с.: ил.

ISBN 5-94157-840-7

Описываются методы настройки баз данных и приложений, применяемые при создании высокоэффективных систем. Показывается, как решения, принимаемые на различных этапах создания системы, начиная с определения требований и выбора архитектуры, влияют на итоговый продукт. Детально обсуждаются методы и приемы, обеспечивающие получение высокой эффективности приложения и базы данных, разработки запросов, а также действия по настройке базы данных во время эксплуатации системы.

Для системных аналитиков, проектировщиков и разработчиков прикладных информационных систем, студентов старших курсов технических вузов

УДК 681.3.06(075.8)
ББК 32.973.26-018.2я73

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Татьяна Лапина*
Зав. редакцией	*Григорий Добин*
Редактор	*Наталья Сержантова*
Компьютерная верстка	*Татьяны Олоновой*
Корректор	*Наталия Першакова*
Дизайн обложки	*Инны Тачиной*
Зав. производством	*Николай Тверских*

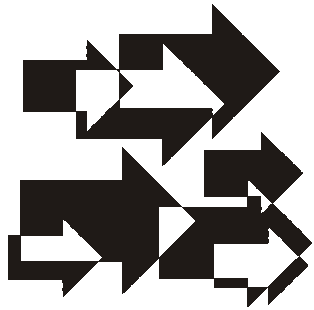
Лицензия ИД № 02429 от 24.07.00. Подписано в печать 27.01.06.
Формат 70×100$^{1}/_{16}$. Печать офсетная. Усл. печ. л. 19,35.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

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

ISBN 5-94157-840-7

Оглавление



Глава 1

Введение

Эффективное использование баз данных является исключительно важным фактором, существенно влияющим на качество прикладной информационной системы, поскольку практически любая такая система использует какую-либо систему управления базами данных для организации хранения информации.

Этот факт, очевидный для специалистов в области баз данных и тех, кто занимается сопровождением работающих систем, часто недооценивается разработчиками (как приложений, так и инструментальных программных средств).

1.1. Как возникают проблемы производительности?

Проблемы производительности возникают, прежде всего, потому, что проектирование и разработка сколько-нибудь сложных информационных систем всегда оказывается довольно трудоемким делом. Обычно проектировщикам требуется значительное время для того, чтобы освоить предметную область, собрать и проанализировать требования к системе и так далее в соответствии с принятой для проекта методологией проектирования и разработки. Точное выполнение функциональных требований занимает значительную часть всего времени, выделенного на проект, и поэтому требования по надежности и производительности часто рассматриваются как менее приоритетные.

Хорошо известно, что проекты такого типа часто затягиваются. Это может приводить к тому, что неизбежные компромиссы между качеством системы и временем, затрачиваемым на ее разработку, смещаются в пользу более быстро реализуемых решений.

Стремительный рост мощности вычислительной техники создает иллюзию неограниченных возможностей аппаратуры. Конечно, профессионалы, которым адресована эта книга, хорошо знают, что сложность алгоритмов не всегда линейная и поэтому далеко не всегда рост объемов данных может быть компенсирован увеличением производительности аппаратуры, но этот факт обычно трудно учитывать, когда приближаются сроки сдачи проекта. Более того, в начальный период эксплуатации системы объемы данных, скорее всего, невелики и поэтому неэффективность реализации никак не проявляется.

Одна из основных причин, по которым производительность системы может оказаться неудовлетворительной, связана с тем, что современные языки программирования используют способ представления алгоритмов, принципиально отличающийся от того, на котором основаны языки запросов баз данных. Эта проблема, известная под метафорическим названием "несоответствие импеданса" (impedance mismatch), обсуждается в *главе 4*. Разработчики могут недооценивать или просто быть не в состоянии учесть сложности, вызванные этим несоответствием.

Наряду с этими субъективными факторами имеются и объективные.

Технологии баз данных развиты в такой степени, что для небольших по размеру и сложности баз данных удовлетворительная производительность достигается автоматически, без каких-либо дополнительных усилий со стороны разработчиков, в особенности при наличии избыточной мощности аппаратных вычислительных средств. Иначе говоря, даже не очень эффективное проектирование и реализация взаимодействия с базой данных не приводят к существенному ухудшению эксплуатационных характеристик информационной системы, и с ростом производительности вычислительных систем абсолютные размеры небольшой системы постепенно растут.

Более 15 лет назад ведущие исследователи и разработчики систем управления базами данных утверждали: "Программистам придется принять тот факт, что оптимизатор СУБД может запрограммировать и выполнить запрос лучше, чем они, точно также, как компилятор языка программирования вырабатывает лучший код" [12]. Время оптимизаторов высокого качества наступило: сегодня мы имеем возможность использовать СУБД, способные выполнять большинство запросов весьма эффективно и, безусловно, лучше чем это может сделать программист среднего класса.

Тем не менее, производительность прикладных информационных систем очень часто оказывается неудовлетворительной и, несмотря на все достижения технологий проектирования приложений и технологий систем управления базами данных, настройка, т. е. вмешательство с целью улучшения характеристик производительности зачастую оказывается необходимой.

1.2. Что такое настройка?

Мы называем настройкой прикладной системы совокупность разнообразных мер, направленных на приведение системы в соответствие с требованиями к производительности без изменения функциональности системы.

Как правило, термин "настройка" применяется по отношению к системам управления базами данных. Это вызвано следующими факторами:

- система управления базами данных всегда является одним из основных компонентов прикладной системы и обычно несет основную нагрузку по выполнению функций этой системы;
- управление сервером базы данных реализуется с помощью высокоуровневых средств, как правило, только косвенно влияющих на его поведение.

Мы используем термин "настройка" в более широком смысле, чем только настройка сервера базы данных, потому что во многих случаях получить удовлетворительные результаты, ограничиваясь только настройкой базы данных, не удается, и требуется внесение изменений или учет требований производительности на уровне различных компонентов системы.

Методы настройки можно условно разделить на два класса: глобальные и локальные. Цель глобальной настройки — улучшение эксплуатационных характеристик системы в целом, в то время как целью локальной настройки является улучшение характеристик отдельных запросов или классов запросов. При этом важно отметить, что в том и другом случае эффект от настройки достигается небольшими (локальными) изменениями. Переписывание всей системы заново не может считаться настройкой как по смыслу самого слова "настройка", так и потому, что полное переписывание, скорее всего, не приведет к достижению целей настройки.

Никакие действия по настройке не должны изменять функциональные свойства или надежность настраиваемой системы. Например, если для ускорения работы системы разработчики отказываются от создания резервных копий или ведения журналов, то такое действие не может считаться элементом настройки, потому что оно снижает надежность системы. Если система оказывается не в состоянии обеспечить приемлемое время ответа при немедленном обновлении и поэтому в системе применяется отложенное обновление, то это также не может считаться элементом настройки, потому что в этом случае пользователь может видеть устаревшие данные, что является изменением функциональности.

Сказанное не означает, конечно, что подобные решения не должны применяться. Компромиссы при создании любой системы неизбежны. Функциональность системы должна быть сбалансирована с возможностями аппаратных средств, а требования к производительности должны быть

реализуемыми, при этом функциональность неизбежно ограничивается. Однако изменение функциональности не может рассматриваться как элемент настройки системы.

По причинам, упомянутым выше, настройка обычно необходима только для достаточно больших или сложных систем. Более того, во многих случаях значительная часть запросов выполняется достаточно эффективно в результате работы оптимизатора запросов СУБД и поэтому тоже не требует каких-либо усилий по настройке. Иначе говоря, как правило, требуется локальная настройка, а не глобальная.

Приведенное выше определение предполагает, что некоторые действия по настройке должны выполняться, начиная с самых ранних фаз создания системы и далее на всех этапах ее проектирования и разработки. Эта точка зрения не совпадает с широко распространенным представлением о том, что настройка может выполняться только в период эксплуатации системы, и отражает идеальную картину, а не реальность. Часто бывает трудно заранее убедить заказчика в том, что настройка является частью нормального процесса создания прикладной системы. Изменить это положение вряд ли возможно, однако в *главе 3* обсуждаются действия по настройке системы на различных фазах проектирования и разработки.

Конечно, необходимость настройки в период эксплуатации готовой системы совсем не обязательно является признаком низкого качества или ошибок при ее создании. Необходимость настройки может возникать и по причинам, которые не могли быть известны до начала эксплуатации.

Хотя мы не ограничиваем настройку только уровнем базы данных, центральное место в этой книге занимают приемы локальной настройки серверов баз данных и отдельных запросов. Авторам приходилось применять подобные методы для настройки систем, уже находящихся в эксплуатации, однако во многих случаях знание этих приемов может существенно улучшить характеристики системы еще на фазе разработки.

1.3. Чем может помочь эта книга?

Несмотря на то, что количество публикаций по системам управления базами данных огромно, вопросы настройки обсуждаются очень редко и недостаточно полно. Возможно, одной из причин этого является то, что работа по настройке баз данных весьма трудоемка и требует очень высокой квалификации. Кроме этого, довольно часто необходимость настройки может быть выявлена только в период эксплуатации системы, когда база данных выросла до достаточно больших объемов. Дорогостоящая настройка системы в этот период плохо вписывается в стандартные жизненные циклы разработки.

Среди немногих публикаций, уделяющих серьезное внимание настройке систем управления базами данных, необходимо выделить книгу [11], в которой на большом экспериментальном материале анализируются и проверяются различные методы настройки, в основном глобальные, хотя и достигаемые локальными изменениями конфигурации системы.

Большинство руководств по системам управления базами данных также уделяет основное внимание глобальной настройке. Возможно, причина этого состоит в том, что для настройки сервера базы данных не требуется внесения каких-либо изменений в программный код приложения. Возможности такой настройки, однако, ограничены и обычно их воздействие недостаточно локально. Поэтому необходимость изменения кода запросов возникает достаточно часто, а иногда появляется необходимость и в изменении программного кода приложения.

Как и в любой быстро развивающейся области знаний и технологий, в программировании вообще, и в настройке баз данных в частности, существует огромное количество мифов, т. е. распространенных мнений, принимаемых большинством разработчиков на веру, без какого-либо анализа их справедливости и применимости в конкретных условиях. Довольно часто носители подобных мифов склонны защищать их с религиозным фанатизмом. Некоторые вопросы вызывали, как известно, даже "религиозные войны", к счастью, без применения оружия.

Механизмы возникновения подобных мифов достаточно просты: ярко сформулированные идеи легко запоминаются и передаются, однако при передаче утрачивается контекст, в котором эти идеи были первоначально предложены, обоснованы и привели к получению хороших результатов. После многократной передачи и фольклорных изменений восстановить исходный контекст без кропотливого исследования литературы уже не представляется возможным. Утраченное понимание технических аргументов подменяется ссылками на авторитет рынка. В результате хорошая идея превращается в миф, истинность которого в отрыве от контекста оказывается весьма сомнительной.

Довольно часто мифы попадают в книги, и, таким образом, вера в них поддерживается априорным авторитетом печатного слова. Одним из примеров таких изданий является книга [14], представляющая методологию настройки, основанную почти исключительно на мифах, большая часть которых неверна или неприменима в контексте современных высокопроизводительных СУБД. Несмотря на это, книга [14], безусловно, понравится читателю, который надеется освоить материал, не затрачивая много времени на его изучение и глубокий анализ.

Авторы надеются, что приведенные в этой книге рекомендации не станут основой для новых мифов. При обсуждении приемов настройки мы показываем, почему эти приемы могут оказаться полезными, и в каком контексте

они хорошо себя показали в нашей практике. Это, безусловно, не означает, что те же приемы окажутся полезными в других ситуациях. Весьма вероятно, что в иных контекстах понадобятся другие приемы. Любой миф превращается в точное знание, если известны условия его применимости, обоснование и способ проверки.

В отличие от [14], данная книга ориентирована на подготовленного читателя. Она не является руководством по использованию баз данных. В основном мы обсуждаем вопросы настройки, хотя по необходимости затрагиваем и некоторые другие темы, в частности, проектирование схем баз данных.

Некоторые из необходимых предварительных сведений в сжатом виде изложены в *главе 2*. Материал этой главы уточняет определения основных понятий и терминологию, используемую в данной книге, но ни в коем случае не может заменить изучение серьезного руководства по системам управления базами данных. Читателю, не знакомому с этими темами, мы рекомендуем прочитать, как минимум, любые две книги из [17, 18, 21]. Каждая из этих книг содержит достаточно серьезное изложение материала, однако взгляды авторов на многие вопросы различаются, и для того, чтобы не оказаться в плену религиозного фанатизма, целесообразно ознакомиться и с альтернативными точками зрения.

Материал *главы 3* предполагает некоторое знакомство с процессом создания программных систем. Полезно, но не обязательно знакомство с какой-либо конкретной методологией и соответствующими инструментальными средствами и компонентами.

Для понимания материала *главы 4* необходима готовность читать тексты, записанные на (вообще говоря, незнакомом) языке программирования, и желательно практическое знание какого-либо из распространенных языков программирования.

Для детального понимания материала *глав 6* и *7* необходимо знание языка запросов SQL. Кроме уже упомянутых книг по системам управления базами данных для изучения SQL можно использовать [19].

Многие черты современных систем управления базами данных являются воплощением идей, обсуждавшихся исследователями в 80-х годах прошлого века и в концентрированном виде представленных в работе [12], которую мы уже упоминали, и [1]. В первую очередь это относится к объектным расширениям модели данных.

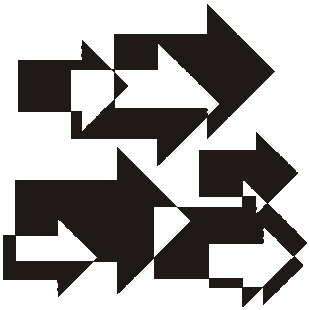
В реальности, однако, эти расширения используются относительно редко, поэтому мы также ограничиваемся в основном обсуждением реляционных средств языка SQL. Мы не затрагиваем также возможности СУБД, связанные с обработкой структур XML, потому что их использование пока не стало общепринятой практикой.

Примеры, включенные в данную книгу, подготовлены на основе задач, возникавших при настройке реальных прикладных систем, в которых применялись различные версии нескольких коммерческих СУБД, включая как наиболее высокопроизводительные, так и небольшие системы.

Во многих случаях в процессе настройки приходится учитывать особенности языка запросов и расширения, специфические для используемой системы управления базами данных. Несмотря на существование стандарта SQL, практически все производители предоставляют свои собственные диалекты. Даже в тех случаях, когда реализуется одна и та же версия стандарта, небольшие различия (например, в именах встроенных функций) позволяют легко определить, для какой именно системы написан запрос.

Авторы не могли, конечно, сделать изложение полностью независимым от специфики расширений. Тем не менее, читатель не найдет в данной книге рекомендаций по настройке каких-либо конкретных систем.

Принципы настройки мало зависят как от особенностей языка, так и от особенностей реализации конкретных систем управления базами данных, потому что для выполнения запросов во всех системах используются одни и те же алгоритмы, сложность которых не может зависеть от производителей СУБД. Вооружившись принципами и методами, изложенными в данной книге, читатель легко сможет найти необходимые инструменты и средства в документации по конкретной системе, с которой ему приходится работать.



Глава 2

Предварительные сведения

2.1. Основные понятия

В данном разделе кратко определяются основные понятия, связанные с применением баз данных, описываются главные функции баз данных и уточняется терминология, используемая в этой книге.

2.1.1. Базы данных и приложения

Прикладной системой будем называть совокупность всех программных компонентов и средств, совместно использующих некоторые данные или обеспечивающих их использование, а также выполняющих какие-либо полезные для потребителя функции. Система может считаться действительно прикладной, если ее пользователи, как правило, не являются профессиональными программистами (или проектировщиками программных систем).

В состав прикладной системы входит все, что нужно для того, чтобы она работала: операционная система (или системы), обязательно система управления базами данных (СУБД), возможно, Web-сервер и другие виды серверов, специализированные пакеты (например, средства анализа данных, генераторы отчетов), и, конечно, программный код, написанный непосредственно для данной прикладной системы. Мы не включаем в состав прикладной системы средства проектирования, разработки и тестирования, хотя без всего этого она работать не сможет.

Системой управления базами данных (СУБД) называется программный продукт, предназначенный для централизованного хранения данных. Функции систем управления базами данных обсуждаются более детально в *разд. 2.1.2*, сейчас важно отметить, что в состав прикладной системы всегда

включается готовая СУБД, т. е. СУБД никогда не входит в состав программного кода, написанного специально для конкретной прикладной системы.

Будем называть базой данных совокупность всех данных, доступ к которым в прикладной системе осуществляется через программные средства, предоставляемые системой управления базами данных. Возможно, в прикладной системе используются и файлы данных, не находящиеся под контролем СУБД, такие данные не будут больше упоминаться в данной книге.

Таким образом, база данных — это совокупность всех постоянно хранимых данных, используемых приложением, независимо от того, каким образом и на каком уровне абстракции они описаны: файлы операционной системы, в которых хранятся базы данных, хранимые данные (таблицы, кластеры, индексы и т. п.), логические структуры данных, в терминах которых происходит обработка (таблицы и представления), или концептуальная модель данных. В старых книгах писали, что база данных описывает всю вселенную для прикладной системы. В некоторых СУБД термин "база данных" используется в более узком смысле, обозначая крупную структурную единицу хранения данных.

Будем называть экземпляром сервера базы данных (или просто сервером базы данных) совокупность компонентов СУБД, находящихся в состоянии выполнения и способных принимать запросы на обработку данных. В некоторых СУБД понятие сервера базы данных обозначается термином instance.

Приложением базы данных, в отличие от прикладной системы, называется любой программный компонент, использующий услуги сервера базы данных, т. е. передающий запросы на сервер базы данных и получающий результаты их выполнения. Чаще всего в роли приложения выступает программный код, написанный специально для прикладной системы, однако с этой точки зрения приложениями являются, например, генераторы отчетов или другие специализированные пакеты для обработки данных. Иногда взаимодействие происходит опосредованно через промежуточные компоненты прикладной системы. Для сервера баз данных это не имеет значения: в роли приложения в этих случаях выступают промежуточные компоненты (например, сервер приложений). В данной книге мы будем называть приложения баз данных просто приложениями.

Фактически с сервером базы данных взаимодействует экземпляр приложения, находящийся в состоянии выполнения. Иногда нам нужно будет различать разные выполняемые экземпляры одного и того же кода приложения (например, запущенные для обслуживания нескольких пользователей) и экземпляры разных приложений (например, интерактивные средства и генератор отчетов).

Любое взаимодействие приложения с сервером базы данных происходит в рамках сеанса (session). С точки зрения сервера базы данных, приложение

начинает работу тогда, когда создает сеанс, регистрируя себя на сервере базы данных. После этого сервер может принимать и выполнять запросы на обработку данных от этого приложения. Работа приложения заканчивается, опять же с точки зрения сервера базы данных, когда приложение закрывает сеанс (или он прекращается по другим причинам). Обычно сервер баз данных может поддерживать несколько сеансов с одним экземпляром приложения, но при этом запросы, передаваемые в разных сеансах, с точки зрения сервера базы данных, приходят от разных приложений.

Заметим, что сеанс работы с сервером базы данных совсем не обязательно связан с сеансом работы пользователя прикладной системы. Эти понятия взаимосвязаны только для некоторых типов архитектур прикладной системы, которые мы коротко обсудим ниже в этой главе. В некоторых случаях сеанс работы пользователя может включать несколько сеансов сервера базы данных, или, наоборот, приложение в рамках одного сеанса может обслуживать несколько пользовательских сеансов последовательно или чередуя запросы от разных пользователей.

Вся работа, выполняемая приложением, для сервера базы данных представляет собой последовательность запросов. На самом деле запросы группируются в транзакции, но определения, связанные с понятием транзакции, мы рассмотрим отдельно в *главе 8*.

Запросы к серверу базы данных записываются на языке запросов СУБД (обычно это диалект SQL). Поскольку само приложение пишется на другом языке программирования, возникает необходимость совмещения текстов на разных языках. Существуют различные способы такого совмещения, которые можно классифицировать как встроенный, статический и динамический SQL.

Синтаксические различия могут быть очень значительными, однако по существу различие состоит в том, как соотносятся время компиляции приложения и время компиляции запроса, а также в том, каким образом запросы параметризуются. Хотя в современных интерфейсах СУБД способ передачи запросов выглядит обычно динамическим, однако сервер базы данных обрабатывает динамические запросы, преобразуя их в статические, поэтому различие между динамическим и статическим SQL не очень важно для настройки. (Однако важно, изменяется ли текст запроса при повторном выполнении.)

Для выполнения каждого запроса создается объект, который называется курсором. Понимание роли курсоров исключительно важно для всего процесса настройки прикладной системы. Детально курсоры обсуждаются в *главе 4*, сейчас мы только подчеркнем, что курсор создается для любого запроса к серверу базы данных, независимо от того, какой тип интерфейса (встроенный, статический или динамический) используется. Важность понятия курсора для настройки определяется тем, что как создание, так и вы-

полнение курсора является относительно дорогостоящей операцией и эффективность работы приложения может зависеть от количества создаваемых и используемых в нем курсоров.

2.1.2. Функции СУБД

Правильное понимание роли и функций СУБД очень важно для создания высокоэффективных систем. Многие проблемы производительности возникают вследствие упрощенного понимания этой роли.

Перечислим основные функции систем управления базами данных, как они были определены в [22] для СУБД, позже названных системами первого поколения:

- независимость данных и приложений;
- защита целостности данных;
- поддержка согласованности данных;
- высокоуровневый язык запросов;
- восстановление после отказов;
- защита от несанкционированного доступа;
- разграничение доступа.

В современных системах значение этих функций существенно изменилось. Некоторые из функций полностью или частично перешли к другим типам компонентов прикладных систем, важность и наличие других зависит от архитектуры прикладной системы и от типа СУБД.

Обеспечение независимости данных и приложений рассматривалось как важнейший элемент поддержки эволюции систем. Независимость данных и программ означает, что допускается изменение структуры данных для поддержки новых задач или в целях настройки, при этом старые приложения остаются работоспособными. Средством достижения независимости является наличие языка описания данных и словаря данных, который представляет структуру данных, хранимых под управлением СУБД. Словари данных и языки описания данных в тои или иной форме включены во все современные системы.

Кроме структуры данных, система обеспечивает выполнение некоторых соотношений, которым должны удовлетворять хранимые данные. Эти соотношения называются ограничениями целостности (integrity). Функция защиты целостности состоит в том, что СУБД не допускает нарушения этих ограничений ни при каких обстоятельствах. Операции, выполнение которых привело бы к нарушению ограничений целостности, отвергаются. Наиболее широко известными типами ограничений целостности являются ограниче-

ния ссылочной целостности, уникальности значений, запрет на неопределенные значения и условия на значения атрибутов.

Понятие согласованности (consistency) в русскоязычной литературе часто тоже обозначают термином "'целостность", что приводит к некоторой путанице, поэтому остановимся на значении этого понятия подробнее. Некоторые из целостных состояний базы данных называются согласованными, СУБД допускает выполнение операций, приводящих базу данных в несогласованное состояние, однако приложение обязано выполнить действия, возвращающие базу данных в другое согласованное состояние.

Более точно набор операций, выполняемых приложением и переводящих базу данных из одного согласованного состояния в, вообще говоря, другое согласованное состояние, называется транзакцией. Функция поддержки согласованности СУБД означает, что после завершения работы приложения база данных будет в согласованном состоянии, т. е. все транзакции приложения будут либо выполнены полностью, либо не оставят никаких следов. Это требование называется атомарностью транзакций.

Поддержка согласованности существенно усложняется, если сервер базы данных допускает конкурентное выполнение нескольких транзакций (вообще говоря, из разных сеансов), потому что в этом случае необходимо исключить взаимные помехи, которые могут возникать при конкурентном выполнении. Сколько-нибудь детальное обсуждение понятий и механизмов, связанных с управлением транзакциями, далеко выходит за рамки данной книги. Некоторые дополнительные сведения приведены в *главе 8*. Детальное изложение теории и методов, применяемых для поддержки согласованности, можно найти в книгах [16, 2, 6].

Необходимо пояснить термин "конкурентное выполнение", использованный в предыдущем абзаце. Мы говорим о конкурентном выполнении, когда нам важно, что операции, выполняемые в разных транзакциях, могут чередоваться или выполняться в каком-то смысле одновременно, но механизм, с помощью которого эта одновременность достигается, не имеет значения. Выполнение может быть конкурентным, если операции выполняются параллельно на многопроцессорном сервере, но параллельность вовсе не требуется. Точно также операции могут выполняться псевдопараллельно в режиме мультипрограммирования (в разных процессах или нитях операционной системы), но это также не требуется, конкурентное выполнение в принципе возможно, даже если сервер базы данных использует всего одну нить в одном процессе операционной системы.

Наличие высокоуровневого языка запросов является, пожалуй, наиболее важной отличительной особенностью СУБД. Значительная часть данной книги посвящена методам выполнения и настройки запросов.

Функция восстановления после отказов часто рассматривается как часть функции поддержки согласованности и также остается одной из важных функций СУБД.

Наконец, функции защиты данных от несанкционированного доступа и разграничение доступа чаще всего не используются или используются в ограниченном объеме, даже если они реализованы в СУБД. Способ реализации этих функций в прикладной системе не оказывает непосредственного влияния на производительность, и поэтому мы не будем уделять этим вопросам много внимания.

2.2. Метрики производительности

Целью настройки любой прикладной системы является улучшение ее работы по каким-либо параметрам. Исходя из этого, первая задача, которую требуется решить, приступая к настройке, — определить, как будет измеряться качество работы системы, и каким образом предполагается его улучшить в процессе настройки.

Таким образом, нас интересует производительность системы и способы ее оценки. Наиболее часто используемые для оценки производительности характеристики системы — *время ответа* и *пропускная способность*. Временем ответа называется время ожидания и время выполнения запроса, а пропускной способностью — количество запросов, обрабатываемых системой за определенный промежуток времени.

Пока мы не делаем различия между этими характеристиками для системы в целом и для базы данных. Сейчас для нас важны особенности и разновидности этих характеристик, а не система, к которой они относятся. Однако это различие становится важным, когда определяются конкретные цели настройки, поскольку время ответа, которое видит пользователь, может быть значительно большим, чем время ответа системы управления базами данных.

Говоря о времени ответа системы в целом, можно рассматривать как среднее время ответа (или время ожидания) пользователя, так и максимальное время ответа. В зависимости от особенностей конкретной системы в качестве цели настройки может выступать уменьшение как среднего, так и максимального времени ответа. Однако хорошо известно, что ограничения на предельное (максимальное) время ответа крайне трудно обеспечить в СУБД. Поэтому такие критерии используются весьма редко, обычно только в системах реального времени, в которых предсказуемость времени ответа является обязательным требованием. Практически во всех остальных классах систем можно ограничивать время выполнения для "почти всех" случаев выполнения запроса. В таком ослабленном виде ограничения на максимально допустимое время ответа становятся реализуемыми.

Аналогичные варианты определения времени ответа системы применимы, конечно, не только к прикладной системе в целом, но и к серверу базы данных.

Следует отметить, что говорить о среднем или максимальном времени ожидания имеет смысл только для запросов одного класса. В промышленных системах разные запросы и разные пользователи могут иметь разные приоритеты в обслуживании, поэтому настройка системы может быть направлена в первую очередь на минимизацию времени обработки конкретных типов запросов.

На первый взгляд может показаться, что увеличение пропускной способности системы приводит к улучшению времени ответа и наоборот, поскольку чем больше запросов может обработать система за определенный промежуток времени, тем меньше времени будет выполняться каждый отдельный запрос. Однако реально такой прямой зависимости не существует, поскольку обычно несколько запросов выполняются одновременно, и, соответственно, пропускная способность системы может быть разной при том же времени ответа.

Иными словами, высокая пропускная способность достигается при высокой степени загрузки системы (ресурсы систем не простаивают), а малое время ответа — при малой загруженности системы (необходимые ресурсы всегда доступны).

Разумной комбинацией требований является, по-видимому, достижение максимально возможной пропускной способности при условии выполнения ограничения на время ответа.

Такие характеристики системы, как надежность и доступность, не являются характеристиками производительности, однако они должны учитываться при настройке, потому что меры, улучшающие эти характеристики, могут приводить к некоторому ухудшению характеристик производительности.

Наряду с абсолютными критериями производительности во многих случаях важны относительные, как правило, объединяемые термином масштабируемость. Масштабируемость — это функция, описывающая зависимость некоторой характеристики производительности от размеров системы (количества оборудования, объема хранимых данных и количества поступающих запросов).

Например, масштабируемость по пропускной способности показывает, как изменяется пропускная способность системы при одновременном увеличении количества оборудования и объема хранимых данных. Очевидно, рост пропускной способности не может превышать увеличение количества оборудования, поэтому идеальная масштабируемость по пропускной способности описывается линейной функцией (с коэффициентом наклона, равным 1).

Часто рассматривается масштабируемость по числу пользователей, однако эта характеристика почти эквивалентна масштабируемости по пропускной способности, но, в отличие от пропускной способности, не поддается количественным измерениям в тестовой обстановке.

Для масштабируемости по времени ответа, наоборот, идеальная масштабируемость описывается постоянной функцией (время ответа не возрастает, несмотря на увеличение объема данных и нагрузки на систему).

Конечно, идеальная масштабируемость никогда не может быть достигнута на практике, так как некоторая часть ресурсов используется на организацию взаимодействия между компонентами системы. Есть и другая причина, по которой идеальная масштабируемость может оказаться недостижимой: запросы разных пользователей и приложений могут использовать и обновлять общие данные.

Не следует переоценивать значение масштабируемости. Далеко не всегда масштабируемость системы необходима, и практически всегда, когда она действительно нужна, она необходима в некоторых ограниченных пределах. Рост многих систем ограничен естественными факторами, такими как число сотрудников организации, или количеством реальных объектов, данные о которых хранятся в системе. Конечно, эти ограничения могут существенно измениться, если изменяется функциональность системы.

Более подробно характеристики масштабируемости обсуждаются в *главе 9*.

Перечисленные выше характеристики применяются для определения целей и оценки результатов настройки системы. В самом процессе настройки важны и другие характеристики производительности, на значения которых можно влиять только на фазе первоначального проектирования, когда планируется покупка оборудования, но которые важны как для оценки реалистичности требований, так и для выбора стратегии настройки. Речь, конечно, идет о характеристиках производительности оборудования.

Традиционно для систем обработки больших объемов данных наиболее важной характеристикой оборудования является быстродействие внешних устройств хранения данных, т. е. дисков. Среди многочисленных характеристик дисковых устройств (минимальное, среднее, максимальное время подвода головок, скорость вращения и др.) наиболее важными, по-видимому, являются среднее и максимальное время доступа в случайном порядке и скорость передачи данных при последовательном доступе к соседним участкам.

Как это ни удивительно, но для многих современных систем производительность процессора (или процессоров) также оказывается важной — многие операции в базах данных интенсивно используют вычислительную мощность процессора. То же самое можно сказать и о некоторых типичных функциях, реализуемых в приложениях.

Как для процессоров, так и для дисков существенное влияние на производительность оказывает наличие и размеры буферной (кэш) памяти. Ни в том, ни в другом случае проектировщики и разработчики не могут влиять на использование этой памяти непосредственно, однако косвенное управление возможно, так как возможности использования буферной памяти устройств зависят от размещения данных.

Наконец, для систем, в работе которых участвует больше одного компьютера, важна пропускная способность вычислительной сети. Обычно, особенно в разговорной речи, эта характеристика называется скоростью, что не совсем точно отражает ее смысл. Конечно, большие сообщения передаются по широкополосной сети значительно быстрее, чем по сети с низкой пропускной способностью, но это определяется именно пропускной способностью. А время передачи небольших сообщений определяется, прежде всего, количеством промежуточных узлов сети (маршрутизаторов и т. п.), а также загруженностью этих узлов.

2.3. Базы данных в архитектурах прикладных систем

В данном разделе обсуждаются различные классы архитектур приложений, использующих базы данных, и влияние особенностей каждого класса архитектур на производительность системы.

Изложение ни в какой степени не является исчерпывающим, мы предполагаем, что читатель знаком с этими вопросами из других источников, и рассматриваем только аспекты, которые важны в контексте данной книги.

Границы между различными классами архитектур, рассматриваемых в этом разделе, являются размытыми. Некоторые системы, которые следует отнести к одному из классов, могут обладать свойствами, более характерными для других классов.

Прежде чем обсуждать особенности отдельных архитектур, необходимо подчеркнуть, что эти особенности нельзя рассматривать как достоинства или недостатки. Каждая из обсуждаемых характеристик может оказывать как положительное, так и отрицательное влияние не только на качество (производительность) получаемой системы, но и на стоимость ее разработки, тиражирования и сопровождения. Не существует архитектуры или метода построения системы, которые превосходили бы другие абсолютно во всех случаях.

Кроме этого, обычно разработчик не может произвольно выбирать ни архитектуру, ни используемые компоненты — то и другое может быть навязано (или выбрано) на основе иных соображений, не имеющих прямого отношения к техническим характеристикам программных средств.

2.3.1. Встроенные СУБД

Авторы относят к этому классу системы, в которых компоненты, реализующие функции системы управления базами данных, выполняются в рамках того же процесса или той же задачи операционной системы, что и приложение.

Можно сказать, что компоненты СУБД встраиваются в код, реализующий функции приложения. Мы намеренно используем неточный термин "встраивается", потому что детали способа объединения компонентов в исполняемый объект для нас несущественны. Компоненты, реализующие функции СУБД, могут быть выполнены в виде библиотеки подпрограмм, статически или динамически подключаемой к коду приложения. В некоторых системах код приложения может интерпретироваться оболочкой, предоставляющей доступ к СУБД (в этом случае, скорее, приложение встраивается в СУБД, чем наоборот).

Системы этого класса получили очень широкое распространение, начиная с конца 80-х гг. прошлого века в связи с появлением и массовым применением персональных компьютеров, достаточно производительных для обработки небольших баз данных.

Для нас важны следующие особенности этого типа архитектур.

- Функциональность приложения и СУБД реализуются в рамках одного процесса операционной системы и поэтому взаимодействие между СУБД и приложением оказывается весьма эффективным (отсутствуют затраты на организацию взаимодействия процессов и задержки, связанные с передачей сообщений по сети).
- Как правило, базы данных невелики по объему, поэтому обычно в таких СУБД используются очень простые механизмы выполнения запросов, зачастую отсутствует оптимизатор запросов. Нередко в таких системах реализуется только некоторое подмножество языка запросов SQL.
- Обычно подобные СУБД работают в однопользовательском режиме, т. е. во время работы приложения другие программы не могут получать доступ к тем же данным. Иногда совместное использование базы данных допускается, но СУБД при этом не обеспечивает поддержку согласованности.
- Как правило, база данных используется только одним приложением.
- Производительность приложения обычно быстро падает с ростом объема данных, в особенности при выполнении сложных запросов.

Можно сказать, что из функций СУБД, перечисленных в предыдущем разделе, для встроенных СУБД наибольшее значение имеют язык запроса, средства описания данных и поддержка целостности. Согласованность

обычно реализуется в ограниченном виде, а остальные функции, как правило, не требуются совсем.

В системах, для которых применение этого класса СУБД целесообразно, единственно важным критерием производительности является время ответа, а пропускная способность и масштабируемость не имеют большого значения.

Еще одна особенность таких систем — наличие дружественного интерфейса и простота использования. Иногда пользователь, не занимающийся программированием профессионально, отчаявшийся получить хоть что-нибудь полезное от программистов, успешно осваивает такую систему сам.

В силу перечисленных особенностей такие системы практически никогда не требуют какой-либо настройки, и поэтому не будут рассматриваться в этой книге сколько-нибудь детально.

Мы упоминаем эти системы потому, что для огромного большинства специалистов, занимающихся созданием приложений, системы такого класса стали первыми, с которыми они столкнулись в своей работе, и опыт, приобретенный при использовании этих систем, существенно повлиял на формирование их навыков.

Одна из целей данной книги — показать, почему техника работы с базами данных, хорошо зарекомендовавшая себя на небольших системах, чаще всего оказывается непригодной при разработке приложений, использующих средние и большие базы данных.

Поскольку системы этого класса не обладают мощными исполнителями запросов, и, с другой стороны, накладные расходы, связанные с передачей запросов из приложения в ядро СУБД, невелики, как правило, наиболее целесообразно использование небольших (по сложности и количеству обрабатываемых данных) запросов. С другой стороны, использование сложных запросов в системах этого класса может приводить к неконтролируемому росту времени ответа.

Даже при выполнении вычислений, на которые объективно требуется много времени, разбиение задачи на небольшие запросы дает возможность информировать пользователя о ходе вычислений или выводить частичные результаты.

Небольшие объемы данных дают возможность использовать логические структуры хранения, обеспечивающие высокую степень гибкости и простоту разработки приложения, но которые были бы крайне неэффективны для больших баз данных.

В последнее время широкое распространение получает другой класс встроенных СУБД, предназначенный для работы в составе распределенных систем. Как правило, эти системы работают на мобильных устройствах и обеспечивают ограниченные возможности обработки данных в периоды, когда

по тем или иным причинам устройство используется в автономном режиме, т. е. без соединения с основной базой данных, размещенной на сервере.

Объемы данных, хранимых в таких системах, обычно совсем невелики и поэтому базы данных сами по себе практически никогда не требуют настройки в традиционном понимании. Узким местом в таких системах является, как правило, взаимодействие и обмен данными с серверными системами.

2.3.2. Серверы баз данных

В данном разделе рассматривается архитектура систем, известная с середины 80-х годов XX века под названием "клиент-сервер". Поскольку термины "клиент" и "сервер" используются в различных контекстах и в довольно разном смысле, необходимо уточнить, что для нас эти понятия характеризуют способ взаимодействия программных компонентов, а не оборудования, и определяют роли этих программных компонентов во взаимодействии, а не сами эти компоненты. В некоторых случаях один и тот же компонент может одновременно выполнять функции как клиента, так и сервера (например, в распределенных СУБД или в многослойных архитектурах, обсуждаемых в *разд. 2.3.3*).

Фактически все большие системы управления базами данных реализовывались как системы типа "клиент-сервер" с момента возникновения концепции СУБД в конце 60-х годов, значительно раньше, чем эти термины стали общепринятыми. Это вызвано тем, что по существу одна из основных функций СУБД — централизация управления данными, используемыми различными приложениями.

Для того чтобы обеспечить централизацию, необходимо выделить компонент (ядро СУБД), который бы постоянно находился в состоянии выполнения, принимал бы запросы приложений (клиентов), выполнял эти запросы и возвращал результаты выполнения, т. е. был бы сервером базы данных в смысле, определенном ранее в этой главе.

Системами класса "клиент-сервер" принято называть такие, в которых программный код приложения взаимодействует с СУБД непосредственно, но выполняется в рамках другого процесса операционной системы и обычно на другом компьютере. Именно для систем класса "клиент-сервер" особенно важно, чтобы СУБД поддерживала все основные функции, перечисленные в предыдущем разделе, в наибольшем объеме.

Во всех современных системах управления базами данных взаимодействие клиента с сервером всегда осуществляется с использованием некоторого сетевого протокола общего назначения (чаще всего TCP/IP, но как раз это для нас несущественно), над которым надстраивается протокол сервера СУБД.

Этот протокол непосредственно используется клиентом СУБД для передачи запросов и получения результатов их выполнения. Сетевые протоколы СУБД могут быть специфическими для конкретной СУБД (например, Oracle SQL*Net) или поддерживаться несколькими (или многими) СУБД различных производителей. Иногда такие обобщенные протоколы надстраиваются над специализированными протоколами конкретных СУБД, при этом возможности использования особенностей конкретной СУБД ограничиваются. Широко используемыми протоколами такого типа являются ODBC (Open DataBase Connectivity, открытый интерфейс доступа к базам данных), и JDBC (Java DataBase Connectivity, средство организации доступа Java-приложений к базам данных) и, вероятно, не существует сколько-нибудь широко распространенных альтернатив.

Любой из упомянутых в предыдущем абзаце протоколов обеспечивает взаимодействие приложения и сервера базы данных на уровне языка запросов. Иногда между приложением и протоколом взаимодействия с сервером базы данных используются дополнительные программные компоненты, изолирующие программиста приложений от языка запросов. Неправильное использование таких компонентов может очень существенно ухудшить производительность всей прикладной системы. Более детально мы рассмотрим эту проблему в *главе 4*.

Для наших целей важны не сами протоколы, а тот факт, что механизм передачи запросов (и результатов) реализуется довольно сложной "башней" протоколов, поэтому время, необходимое для этого взаимодействия, обычно оказывается довольно большим.

Мы уже отмечали, что взаимодействие каждого клиента с сервером баз данных осуществляется в рамках сеанса (подключения, сессии) с базой данных. Информация о сеансе сохраняется как в клиенте, так и на сервере, иными словами, протоколы взаимодействия с СУБД являются протоколами с состоянием. Во многих СУБД процесс установления сеанса занимает относительно много времени, поэтому часто программа-клиент использует один и тот же сеанс работы с базой данных на протяжении всего (интерактивного) сеанса работы с конечным пользователем.

Наиболее существенно для нас то, что даже при использовании сетей с очень высокой пропускной способностью время, необходимое для получения ответа на сообщение (round trip), может быть значительным, даже если обработка этого сообщения не занимает сколько-нибудь значительное время. С другой стороны, передача относительно больших сообщений на широкополосных сетях может выполняться очень быстро.

Важно отметить, что во всех современных СУБД сетевые протоколы используются даже в том случае, если клиент и сервер выполняются на одном и том же компьютере.

Обработка любого запроса к СУБД включает в себя несколько операций, требующих синхронизации клиента и сервера, т. е. несколько ожиданий ответа, поэтому выполнение любого запроса требует некоторого минимального времени, независимо от сложности запроса. С другой стороны, при передаче результатов выполнения запроса пересылка данных может осуществляться в какой-то мере асинхронно. Поэтому дополнительное время, вызванное задержками сетевых протоколов, весьма существенно для коротких запросов, быстро выполняемых на сервере и возвращающих небольшое количество данных. Однако с ростом сложности запроса или объема возвращаемых данных относительная доля сетевых задержек в общем времени ответа очень быстро уменьшается, в особенности в сетях с высокой пропускной способностью.

В *главе 4* обсуждается, каким образом эти особенности архитектуры можно учесть при разработке приложений.

2.3.3. Многоуровневые архитектуры

Системы типа "клиент-сервер" оказываются неэффективными или слишком ограничительными для некоторых распространенных классов приложений. Наиболее серьезные проблемы возникают в тех случаях, когда количество (конечных) пользователей, работающих с системой, очень велико (десятки тысяч и более). В этом случае поддержка сеансов работы с системой становится не менее важной, чем поддержка целостности и согласованности данных.

Так, глобальная система резервирования авиабилетов не может быть остановлена (или перезапущена) не только из-за потенциальной потери части транзакций, но и потому, что восстановление всех активных сеансов с системой резервирования заняло бы несколько часов и привело бы к катастрофическим убыткам [8]. Конечно, системы такой сложности и с такими требованиями к надежности уникальны, однако сходные проблемы возникают и в меньших по масштабу системах.

Системы, ориентированные на обслуживание очень большого количества пользователей, обычно используют мониторы транзакций или (в современной терминологии) серверы приложений. Основное отличие мониторов транзакций от серверов приложений состоит в том, что первые, как правило, используют специализированные сетевые протоколы, а последние — стандартные протоколы WWW. Детальное обсуждение относительных преимуществ и недостатков протоколов такого типа выходит за рамки этой книги.

Для нас существенно то, что сервер приложений, выступая в роли сервера по отношению к клиентской программе, принимает запросы на выполнение определенных приложений и организует их выполнение, в том числе их

взаимодействие с базой данных, выступая по отношению к серверу базы данных в роли клиента.

При этом сервер приложений может брать на себя часть функций СУБД (частично дублируя их). В частности, сервер приложений берет на себя функции защиты от несанкционированного доступа, выполняя идентификацию и проверку полномочий пользователей. Взаимодействие с базой данных осуществляется от имени одного пользователя базы данных для всех пользователей прикладной системы. При этом функция разграничения доступа может перекладываться на приложение.

Для сокращения времени на установление сеансов (которые в этом контексте обычно называются соединениями) с базой данных используется пул соединений (может поддерживаться как СУБД, так и сервером приложений). При использовании пула соединений сервер приложений не создает новое соединение для каждого выполнения приложения, а выделяет одно из свободных, а по окончании работы приложения возвращает это соединение в пул. В данной книге термин "соединение" будет использоваться в другом смысле (как русское название реляционной операции `JOIN`).

Преимущества использования серверов приложений определяются следующими факторами:

- при взаимодействии с клиентом (сервера приложений) может использоваться протокол без состояния;
- при использовании пула соединений, вообще говоря, экономится время на установление соединений;
- взаимодействие сервера приложений с клиентом может быть организовано на основе протокола без состояний;
- как правило, сервер приложений находится в одной локальной сети с сервером базы данных, что обеспечивает более эффективное взаимодействие приложения (выполняемого на сервере приложений, а не на клиентской машине) с базой данных.

Рассмотрим особенности этой архитектуры, важные с точки зрения настройки производительности системы.

Прежде всего, выполнение запросов на сервере баз данных не зависит от того, используется сервер приложений или нет, поэтому основные принципы настройки одинаковы для систем "клиент-сервер" и систем, включающих сервер приложений.

Имеется, однако, существенная разница в критериях оптимизации запросов. Многие СУБД могут оптимизировать запросы по-разному в зависимости от того, требуется быстрое получение первых строк ответа или результата выполнения запроса полностью. В системах "клиент-сервер" получение первых строк создает у пользователя иллюзию более быстрого ответа системы и

поэтому может быть целесообразно. В системах же с сервером приложений для формирования ответа пользователю необходимо, как правило, получить результат выполнения запроса полностью.

В связи с этим возникает проблема результатов большого объема. Если большой по объему результат выполнения запроса возвращать полностью, то непомерно возрастает нагрузка на сервер приложений и увеличивается время передачи результата клиенту, а если результат фрагментируется приложением, то неоправданно увеличивается нагрузка на сервер базы данных и в меньшей степени на сервер приложений.

Предположим, ответ на запрос пользователя содержит несколько сотен тысяч строк. Такой объем данных может быть передан из СУБД в приложение за единицы секунд, однако на формирование страницы HTML такого размера и, в особенности, на передачу такой страницы клиенту потребуется намного больше времени.

Если же вывод будет сегментироваться, потребуется многократное выполнение запроса сервером базы данных (если сервер приложений используется как сервер без состояний).

Наличие сервера приложений само по себе мало влияет на низкую эффективность слишком мелких запросов к базе данных. В некоторых случаях время, затрачиваемое на взаимодействие с базой данных, может в десятки раз превышать время выполнения самого приложения.

В зависимости от особенностей приложения эта проблема может решаться двумя различными способами:

- сохранением результатов выполнения запросов к базе данных в памяти сервера приложений (кэшированием);
- перераспределением функциональности приложения между компонентами системы.

Кэширование результатов на сервере приложений основано на предположении о том, что количество различных запросов, выполняемых пользователями, во много раз меньше количества пользователей, поэтому вероятность того, что результаты выполнения запроса вскоре понадобятся снова, достаточно велика.

В некоторых системах в кэш сервера приложений копируется вся база данных, что фактически означает отказ от использования процессора запросов базы данных.

Перераспределение функциональности приложения состоит в том, что часть кода приложения выполняется непосредственно на сервере баз данных (в форме хранимых процедур и триггеров). Такое решение дает возможность в полной мере использовать мощные возможности СУБД, в том числе объектные расширения языка запросов. Подобная архитектура, однако, может

оказаться более сложной для реализации, так как требует отказа от прямолинейной схемы: "данные в базе данных, бизнес-логика на сервере приложений", и, следовательно, предъявляет повышенные требования к квалификации коллектива разработчиков.

Для построения действительно высокопроизводительных и высокоэффективных систем необходимо правильное сочетание всех упомянутых (и, возможно, многих других) механизмов, но данная книга посвящена почти исключительно вопросам организации эффективного взаимодействия с базой данных.

2.4. Алгоритмы выполнения и оптимизации запросов

В данном разделе обсуждается обработка (т. е. компиляция, оптимизация и выполнение) запросов на сервере базы даных. Скорее всего, читателям никогда не придется программировать такие алгоритмы, однако их понимание и в особенности понимание их сложности и стоимости абсолютно необходимо для того, чтобы заниматься настройкой баз данных. Более детальное обсуждение вопросов, рассматриваемых в этой главе, можно найти в обзорах [15, 7, 5].

2.4.1. Обработчик запросов

Независимо от архитектуры системы управления базами данных и архитектуры прикладной системы запрос на обработку данных, записанный на языке манипулирования данными (в этой книге мы рассматриваем только SQL), попадает в обработчик запросов.

Обработка запросов включает перечисленные далее этапы.

- **Компиляция.** Как и для всех других языков программирования, на этапе компиляции выполняется лексический и синтаксический анализ и генерируется некоторое внутреннее представление запроса. Обычно это представление является записью алгебраического выражения, эквивалентного исходному запросу.

  Мы не будем рассматривать процесс компиляции, поскольку эти вопросы достаточно хорошо известны и не имеют прямого отношения к основной теме этой книги. Отметим только, что при компиляции запроса используется информация из схемы базы данных.

- **Оптимизация.** На фазе оптимизации строится оптимальный план выполнения запроса на основании промежуточного кода, полученного компилятором, информации о схеме базы данных, статистических ха-

рактеристик хранимых данных, модели стоимости и критериев оптимизации.

- **Интерпретация.** На этой фазе происходит фактическое выполнение операций, содержащихся в плане выполнения запроса, выработанном оптимизатором.

Необходимо объяснить смысл, в котором в этом контексте используется термин "оптимизация". Задача построения оптимального плана запроса формулируется как задача дискретного математического программирования и поэтому можно рассматривать оптимальный план выполнения запроса в точном математическом смысле. Проблема состоит в том, что функция стоимости в этой задаче всегда является приближенной, и, следовательно, точное решение задачи оптимизации не всегда будет давать лучший по фактическому времени выполнения или другим критериям план выполнения запроса.

В реальности точное решение задачи оптимизации, в особенности для сложных запросов, не всегда имеет смысл, так как многие параметры, от которых зависит качество плана, могут быть определены только в результате выполнения запроса (например, количество строк промежуточных и окончательного результатов выполнения).

Процесс обработки запросов в высокопроизводительных системах может быть более сложным. Например, некоторые системы могут менять план выполнения запроса с учетом результатов частичного выполнения этого плана.

План выполнения запроса представляет собой иерархическое дерево. Вершины этого дерева описывают алгебраические операции. Дуги, выходящие из каждой вершины, описывают операнды соответствующей операции. Корень дерева соответствует операции, которая вырабатывает конечный результат всего запроса.

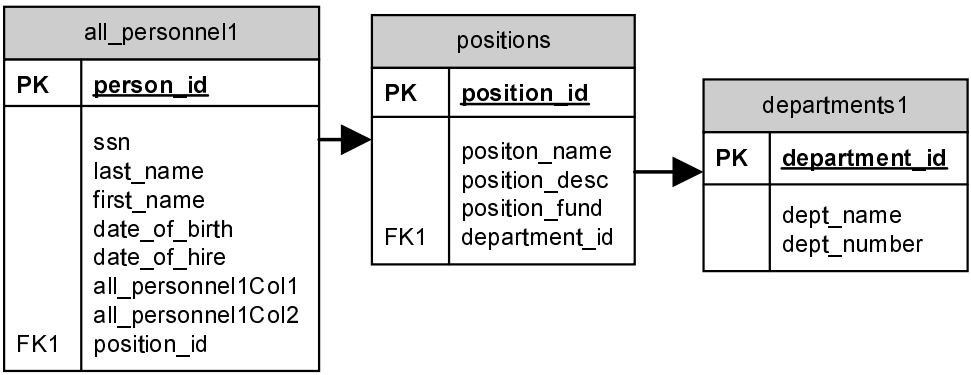


Рис. 2.1. Фрагмент схема базы данных

Например, для выполнения в базе данных персонала, фрагмент которой показан на рис. 2.1, запроса:

```
select first_name, last_name, position_name
from all_personnel p
join positions s on p.position_id=s.position_id
where department_id in (
   select department_id
   from departments
   where department_name like '33%')
```

в СУБД Microsoft SQL Server может быть сгенерирован следующий план:

```
  |--Nested Loops(Left Semi Join,
                  OUTER REFERENCES:([s].[department_id]))
       |--Nested Loops(Inner Join,
                  OUTER REFERENCES:([p].[position_id]))
       |    |--Clustered Index Scan(OBJECT:
                    ([all_personnel].[pk_person] AS [p]))
       |    |--Clustered Index Seek(OBJECT:
                      ([positions].[pk_pos] AS [s]),
                      SEEK:([s].[position_id]=[p].
                      [position_id]) ORDERED FORWARD)
       |--Clustered Index Seek(OBJECT:
                 ([departments].[pk_depts]), SEEK:
                 ([departments].[department_id]
                 =[s].[department_id]), WHERE:
                 (like([departments].
                 [department_name], '33%', NULL))
```

Приведенная здесь запись плана отличается от реальной лишними переводами строки.

При выполнении того же запроса в СУБД Oracle план может оказаться следующим:

```
xecution Plan
----------------------------------------------------
  0      SELECT STATEMENT Optimizer=CHOOSE
  1    0   NESTED LOOPS
  2    1     NESTED LOOPS
  3    2       TABLE ACCESS (FULL) OF 'ALL_PERSONNEL'
  4    2       TABLE ACCESS (BY INDEX ROWID) OF 'POSITIONS'
  5    4         INDEX (UNIQUE SCAN) OF 'PK_POS' (UNIQUE)
```

```
6    1     TABLE ACCESS (BY INDEX ROWID) OF 'DEPARTMENTS'
7    6       INDEX (UNIQUE SCAN) OF 'PK_DEPTS' (UNIQUE)
```

Существуют клиентские программы, которые могут представлять планы выполнения запросов в графической форме, но в этой книге подобное представление не будет использоваться.

Далее в данном разделе мы опишем встречающиеся в планах выполнения запроса операции. Для того чтобы изложение в этой книге не зависело от конкретной СУБД, мы будем использовать более абстрактный способ записи планов.

План является интерпретируемым представлением исходного запроса. Интерпретация плана начинается с листьев дерева, и по мере получения промежуточных результатов становится возможным выполнение операций, расположенных ближе к корню, до тех пор, пока интерпретация не закончится в корневой вершине.

Практически во всех современных системах управления базами данных при выполнении запросов алгебраические операции реализованы как операции над потоками данных. Иными словами, каждая операция, кроме операций непосредственной выборки хранимых данных, читает каждый операнд из отдельного потока данных и записывает результат как выходной поток. Этот механизм похож на механизм pipe в операционных системах, однако мы не делаем никаких предположений о том, как фактически организовано взаимодействие операций. То есть не имеет значения, используются для этого отдельные процессы, нити или выполнение организовано другим способом. В некоторых СУБД могут применяться разные механизмы в зависимости от конфигурации системы и размещения данных.

Для нас важно то, что данные, передаваемые между операциями одного запроса, не хранятся, поэтому не требуется время на их запись и считывание.

Разумеется, некоторые операции, например, сортировка, не могут начать вырабатывать результат, пока не закончено чтение входных операндов, и в этом случае промежуточное хранение данных в оперативной памяти или на диске неизбежно, однако при этом хранение организуется внутри операций, а не между операциями.

Какие же операции необходимы для того, чтобы выполнять запросы, представленные на языке SQL?

Поскольку SQL первоначально был создан как язык запросов для реляционных систем, для выполнения запросов используются операции реляционной алгебры. Из теории известно, что любой запрос можно записать с помощью алгебраического выражения, содержащего теоретико-множественные операции (объединение, пересечение, разность и декартово произведение) и реляционные операции селекции. По соображениям эффективности реализации к этому списку обычно добавляют и другие операции, в первую очередь различные варианты операции соединения.

Конечно, на самом деле SQL не является реляционным языком запросов, так как допускает появление дубликатов как в хранимых таблицах, так и в результатах выполнения. Кроме этого, современные версии SQL, в том числе стандарт, начиная с версии SQL:1999, содержат объектно-реляционные расширения.

В этой книге мы не будем обсуждать различия между реляционными языками и SQL, а также объектные расширения реляционной модели, потому что эти различия касаются семантики выполняемых операций и преобразования запроса на высокоуровневом языке (т. е. SQL) в план запроса, но мало влияют на эффективность алгоритмов выполнения низкоуровневых операций. Для реализации новых возможностей языка запросов существенной модификации алгоритмов выполнения операций не потребовалось, за исключением, пожалуй, новых типов индексов.

Мы будем рассматривать алгебру, содержащую двуместные и одноместные операции. (Некоторые операции могут быть эффективно реализованы как многоместные, и некоторые системы используют такие алгоритмы, но мы не будем рассматривать эти расширения.)

Список двуместных операций включает:

- теоретико-множественные операции (объединение, пересечение и разность);
- произведение;
- эквисоединение (внутреннее и внешнее), полусоединение и антисоединение.

Одноместные операции мы разделим на две группы. К перовй группе относятся:

- сортировка;
- устранение дубликатов;
- группировка.

Особенность операций этой группы в том, что для их выполнения могут использоваться варианты алгоритмов двуместных операций.

Вторая группа одноместных операций включает единственную операцию выборки данных. Очевидно, эта и только эта операция может находиться в листьях дерева, описывающего план выполнения запроса.

Заметим, что среди перечисленных операций нет эквивалента для операций селекции и проекции. Это объясняется тем, что выполнение селекции и выборку нужных колонок всегда можно совместить с предыдущей операцией в плане выполнения запроса. После этого, если необходимо, проекция получается с помощью операции устранения дубликатов. В некоторых случаях мы будем указывать операцию фильтрации для того, чтобы показать, в каком именно месте плана выполняется селекция или выборка подмноже-

ства атрибутов для проекции. Единственный особый случай — операция выборки хранимых данных, для которой наличие в плане выполнения запроса операции селекции может существенно повлиять на выбор алгоритма выборки данных.

Каждую из операций можно реализовать с помощью различных алгоритмов, и для большинства операций невозможно указать какой-нибудь один алгоритм, который бы во всех случаях превосходил все остальные. Поэтому все высокопроизводительные системы управления базами данных реализуют несколько алгоритмов для каждой операции. Фактически план выполнения запроса содержит не только операции, но и алгоритмы, выбранные оптимизатором для их выполнения. Иными словами, разные алгоритмы выполнения одной операции рассматриваются как различные, но эквивалентные операции.

2.4.2. Основные алгоритмы выполнения операций

В данном разделе мы неформально опишем основные алгоритмы, используемые для выполнения перечисленных выше алгебраических операций. Нас будут интересовать не детали реализации вариантов этих алгоритмов, а примерные оценки их стоимости. Можно было бы привести и более точные оценки сложности данных методов, однако эти оценки зависят от размеров операндов, которые не могут быть точно определены до того, как выполнены предыдущие операции.

Оценки хорошего качества невозможно получить без учета неравномерности распределения данных. Для получения таких оценок применяется техника построения гистограмм. Без этого не обходится ни одна высокопроизводительная СУБД, но мы не будем рассматривать эти вопросы в данной книге.

Приведенные ниже расчеты пригодны только для очень грубой оценки времени выполнения запроса. Например, независимо от качества оптимизатора любой запрос должен прочитать все данные, которые необходимы для формирования результата, поэтому время его выполнения не может быть меньше, чем время выборки этих данных.

В случае если фактическое время выполнения запроса не очень существенно отличается от этой оценки, то настраивать такой запрос не имеет смысла — скорее всего, существенные улучшения невозможны. Конечно, при неудачном выборе плана количество фактически прочитанных данных может оказаться существенно больше количества необходимых.

В качестве основной меры стоимости нас будет интересовать количество операций обмена с дисками, которое необходимо для выполнения этой операции. Несмотря на то, что фактически операции получают данные из входных потоков и записывают их в выходные потоки, при оценке стоимости операций полезно предполагать, что данные должны быть прочитаны с

диска и записаны на диск. Дело в том, что в случае, если требуется более чем однократный просмотр входных данных, они должны быть записаны на диск для повторной обработки (конечно, если только они не могут поместиться в выделенной области оперативной памяти).

Алгоритм вложенных циклов

Алгоритм вложенных циклов можно реализовать для большинства рассматриваемых нами двуместных и одноместных операций.

Идея алгоритма вложенных циклов предельно проста: строки из первого операнда читаются внешним оператором цикла, а во внутреннем цикле читаются строки второго операнда и выполняется сопоставление строк первого и второго операндов, а также, если необходимо, запись строки результата.

Для операции декартова произведения сопоставление состоит просто в том, что из двух исходных строк строится строка результата, а для операции соединения выполняется проверка условия соединения.

Очевидно, что стоимость алгоритма вложенных циклов должна быть пропорциональна произведению размеров входных операндов (плюс стоимость их чтения и записи результатов).

Практическая реализация этого алгоритма для операндов больших размеров использует циклы не по строкам операндов, а по блокам диска, в которых они хранятся. Это изменение практически не влияет на вычислительную сложность алгоритма, но существенно сокращает количество операций чтения диска.

Приведем оценку количества операций обмена с диском для этой модификации алгоритма.

Пусть операция выполняется над операндами R и S, для хранения которых занято B_R и B_S блоков соответственно. Пусть в распоряжении алгоритма имеется M единиц памяти, в каждой из которых может быть размещен один блок.

Предположим, что $B_R < B_S$. Во внешнем цикле можно считывать фрагменты операнда R, содержащие по $M-1$ блоков каждый. Оставшаяся единица памяти будет использоваться для блоков второго операнда. В этом случае, очевидно, сложность всей операции составит

$$(B_R/(M-1)\times B_S).$$

Ниже мы рассмотрим специальные случаи этого алгоритма, характеризующиеся более низкой стоимостью (как по сложности, так и по количеству обменов с диском).

Алгоритм на основе хеширования

Для выполнения многих реляционных операций требуется не вычисление предиката произвольного вида, заданного в запросе, а проверка значений некоторых атрибутов на равенство значений. Например, это справедливо для наиболее распространенных видов операции соединения (эквисоединение и естественное соединение) и для операции устранения дубликатов.

Для реализации таких операций можно применять более эффективные алгоритмы, чем алгоритм вложенных циклов. В этом разделе описывается алгоритм на основе функции хеширования, а в следующем — на основе алгоритма сортировки слиянием.

Обычно функции хеширования применяются при построении структур данных для очень быстрого поиска, например, в таблицах, используемых компиляторами, или в базах данных для некоторых видов индексов или таблиц. Функция хеширования получает в качестве аргумента значение ключа (например, атрибута строки) и вырабатывает в качестве результата условный адрес, по которому эта строка должна быть размещена. При этом требуется, чтобы пространство условных адресов было невелико по сравнению с количеством различных возможных значений аргумента (но достаточно для размещения всех фактически имеющихся значений). Кроме этого, требуется, чтобы фактически существующие значения аргумента достаточно равномерно заполняли пространство условных адресов.

Мы не будем в данной книге рассматривать способы построения функций хеширования, достаточно свойств этих функций, перечисленных в предыдущем абзаце.

Алгоритм выполнения реляционных операций на основе функции хеширования состоит из двух фаз.

- **Распределение.** Просматривается первый операнд (таблица), к каждой строке применяется функция хеширования и строка записывается в соответствующий условному адресу блок временной области хранения, которая может быть размещена в оперативной памяти или на диске.
- **Сравнение.** Просматривается второй операнд, для каждой строки вычисляется значение функции хеширования. Затем эта строка сравнивается со всеми строками первого операнда, для которых получено такое же значение функции хеширования (т. е. размещенными в одном блоке). В зависимости от результатов сравнения генерируется (или не генерируется) строка результата операции.

Корректность этого алгоритма следует из того, что различные значения функции хеширования не могут быть получены из одинаковых значений ключей, поэтому сопоставление строк второго операнда со строками первого, распределенными в другие блоки, не может быть успешным.

Трудоемкость этого алгоритма вычислить достаточно легко: оба операнда просматриваются один раз, поэтому, если промежуточный результат помещается по размеру в оперативной памяти, количество обменов с диском будет линейно зависеть от размеров исходных таблиц.

Непосредственная реализация этого алгоритма для таблиц, ни одна из которых не может быть размещена в оперативной памяти, может оказаться неэффективной, потому что могут потребоваться частые обмены с диском в произвольном порядке для загрузки блоков временного хранилища. Известны более сложные варианты этого алгоритма (например, каскадное хеширование), которые работают эффективно с таблицами любого размера. Мы не будем сколько-нибудь детально рассматривать такие алгоритмы в этой книге.

Нам далее понадобится только тот факт, что время выполнения алгоритма на основе функции хеширования (почти) линейно зависит от размеров входных таблиц.

Алгоритм на основе слияния упорядоченных списков

Алгоритм выполнения реляционных операций на основе сортировки и слияния, как и алгоритм на основе функции хеширования, применим в тех случаях, когда для вычисления результата необходимо сравнивать значения атрибутов только на равенство.

Как и в алгоритме хеширования, эффективность алгоритма достигается за счет того, что строки с одинаковыми значениями атрибутов, участвующих в сравнении, собираются вместе, однако для этого применяется не функция хеширования, а упорядочивание.

Алгоритм выполнения реляционных операций на основе сортировки состоит из двух фаз.

- **Сортировка.** На первой фазе обе входных таблицы сортируются по составному ключу, содержащему все атрибуты, по которым должно производиться сравнение. Результаты сортировки записываются во временную область памяти (чаще всего на диске); естественно, каждый операнд сортируется и записывается отдельно.
- **Слияние.** На фазе слияния выполняется однократное считывание отсортированных операндов и сопоставление их строк для генерации строк результата.

  Фактически для каждого значения ключа сортировки выполняется алгоритм вложенных циклов.

Трудоемкость этого алгоритма, очевидно, складывается из трудоемкости операций сортировки входных операндов и затрат на выполнение фазы слияния. Первая составляющая, как известно, пропорциональна $N \log N$ для

входного аргумента размером *N*, затраты на фазу слияния линейно зависят от суммы размеров входных таблиц и результата операции.

Заметим, что выполнение вложенных циклов на фазе слияния не является источником неэффективности, потому что каждое повторение внутреннего цикла генерирует строку результата. Вложенные циклы применяются только к совпадающим значениям атрибутов сравнения.

Трудоемкость этого алгоритма может оказаться значительно меньшей, если один или оба операнда уже правильно упорядочены к моменту начала операции.

Важной особенностью этого алгоритма является то, что результат операции всегда получается упорядоченным по атрибутам сравнения.

Некоторые оптимизаторы умеют использовать этот факт при построении планов выполнения запросов, содержащих несколько операций соединения по одному и тому же набору атрибутов. В этом случае применение алгоритма на основе сортировки дает значительно лучшие результаты, чем сочетание различных алгоритмов благодаря тому, что входные аргументы последующих операций автоматически оказываются упорядоченными.

2.4.3. Выборка хранимых данных и индексы

В данном разделе обсуждаются операции выборки данных из хранимых структур, таких как таблицы в реляционных базах данных. В качестве примеров других хранимых структур можно упомянуть материализованные представления и моментальные копии. Нас, однако, пока не интересуют различия между этими структурами.

Операция выборки данных извлекает хранимые данные и преобразует их в потоки, передаваемые как операнды других операций запроса. Как и для прочих операций, одновременно с выборкой данных может выполняться селекция (если, конечно, условие селекции задано в запросе). В реляционных языках запросов эти критерии сводятся к проверке простых условий на значения атрибутов (сравнение значения атрибута с константой).

Существуют два принципиально разных алгоритма выполнения этой операции: полный просмотр и прямой доступ (по некоторому указателю). В зависимости от условий селекции и структуры схемы хранения (наличия индексов и других вспомогательных структур), для выборки используется некоторая их комбинация. Наиболее простыми и наиболее распространенными комбинациями являются полный просмотр хранимых таблиц и поиск по точному значению атрибута в индексе с последующим доступом к хранимой таблице по указателю. Более сложные варианты могут включать просмотр участка (диапазона значений) или обходиться без чтения хранимых таблиц, извлекая все данные только из индексов.

Индексом называется избыточная структура хранения, предназначенная для ускорения выборки данных по некоторому классу условий на значения атрибутов. Известно большое количество разнообразных индексных структур, поддерживающих различные типы данных и различные классы запросов, однако детальное рассмотрение индексных структур выходит за рамки этой книги. Нас в основном будет интересовать наиболее распространенный тип индексов: одномерные упорядоченные индексы, как правило, реализуемые в современных СУБД как B-деревья.

Более того, для рассматриваемых в этой книге аспектов важна не сама структура B-дерева, а предоставляемая этой структурой функциональность. Единицей хранения в индексе является запись, состоящая из ключа и списка ссылок на строки таблицы, по которой построен этот индекс. В качестве ключа используются значения одного или нескольких атрибутов таблицы. Таким образом, каждая индексная запись содержит значение атрибута (или нескольких атрибутов) и список строк таблицы, содержащих это значение.

Поисковая функциональность индексов может быть реализована следующими операциями:

1. Поиск по значению ключа возвращает позицию в индексе, наименьшее значение ключа, не меньшее, чем аргумент операции, и список ссылок на строки, содержащие это значение ключа.
2. Операция получения следующей индексной записи получает в качестве аргумента позицию в индексе и возвращает следующую позицию и содержимое следующей индексной записи.

Эти две операции позволяют осуществить точечный поиск (по заданному значению ключа) и поиск по заданному диапазону значений ключа. Вычислительная сложность поиска по значению ключа пропорциональна высоте (т. е. количеству уровней) дерева, в частности, количество обращений к диску, необходимых для выполнения этой операции, не превышает количество уровней дерева. Для больших списков ссылок могут потребоваться дополнительные обращения к диску (в том случае, если список не помещается полностью в один блок). Операция получения следующей индексной записи не требует спуска по дереву, и ее сложность поэтому совпадает со сложностью получения списка ссылок (т. е. обычно не больше одного обращения к диску).

Теперь мы можем более точно описать алгоритм выборки данных по индексу. Для того чтобы этот алгоритм был применимым, необходимо, чтобы исходный запрос содержал условие на значение атрибута, по которому построен индекс, или условие на диапазон значений этого атрибута.

1. Работа алгоритма выборки начинается с поиска по заданному значению атрибута в индексе (или по нижней границе заданного диапазона).
2. Если возвращенная индексная запись удовлетворяет условию запроса (т. е. возвращенное значение атрибута совпадает с заданным в точечном

запросе или не превышает верхнюю границу диапазона), то выполняется выборка строк таблицы по каждой из ссылок, содержащихся в индексной записи. Иначе (если возвращенная индексная запись не удовлетворяет условиям запроса) работа алгоритма завершается.

3. Для запросов по диапазону выбирается следующая индексная запись и осуществляется переход к предыдущему шагу.

Поскольку строки таблицы, содержащие одинаковые значения атрибута, как правило, рассеяны по всей таблице, для получения каждой строки таблицы потребуется, скорее всего, отдельное обращение к диску. Следовательно, сложность алгоритма выборки по индексу оценивается сверху количеством извлекаемых строк таблицы плюс постоянная составляющая, необходимая для спуска по дереву.

Алгоритм полного просмотра таблицы не нуждается в детальном описании. Его сложность определяется размерами таблицы и оценивается количеством блоков, которые занимает таблица.

Как правило, размеры строк таблицы значительно меньше, чем размеры блоков, поэтому число строк обычно в десятки раз больше числа блоков (это может быть иначе для очень больших строк или для кластеризованных таблиц, но мы пока не рассматриваем эти случаи).

Из этого следует, что алгоритм выборки по индексу выигрывает в том случае, если количество извлекаемых строк мало по сравнению с размерами таблицы, и проигрывает, если количество извлекаемых строк велико.

Можно улучшить алгоритм выборки по индексу таким образом, чтобы каждый блок таблицы считывался не более одного раза, однако и в этом случае считывание каждого блока будет выполняться как отдельная дисковая операция, т. е. будет использоваться доступ к диску в случайном порядке, в отличие от последовательного считывания при полном просмотре.

Как правило, последовательное считывание выполняется во много раз быстрее, чем доступ в произвольном порядке (возможно опережающее чтение и считывание нескольких блоков за один оборот диска, требуется меньше перемещений головок дисков). Может оказаться, что считывание 10% блоков в произвольном порядке требует больше времени, чем полный просмотр.

Перечисленные свойства алгоритмов выборки данных необходимо учитывать как при проектировании структуры хранения, так и при оценке планов выполнения запросов. Как правило, построение индексов для таблиц небольшого размера (содержащих десятки или сотни строк и помещающихся в единицы блоков) нецелесообразно.

Важная особенность алгоритма выборки по индексу состоит в том, что ее результат получается упорядоченным по значению индексного ключа. Эта

особенность может быть полезна в том случае, если результат выборки является входным операндом операции соединения по этому ключу.

Индексы могут быть полезны и для повышения эффективности алгоритма вложенных циклов. В случае если таблица, просматриваемая во внутреннем цикле, имеет индекс по атрибуту соединения, вместо полного просмотра этой таблицы на каждой итерации можно применить алгоритм выборки по индексу. В результате внутренний цикл существенно сокращается, поскольку в нем выбираются только строки, необходимые для формирования результата операции соединения.

Если значения атрибута уникальны (например, атрибут соединения является первичным ключом), внутренний цикл вообще вырождается до выборки одной строки таблицы. В частности, это может иметь место для очень частого случая соединения таблиц, связанных отношением ссылочной целостности (первичный-внешний ключ).

До сих пор мы предполагали, что при хранении данных в таблицах не поддерживается никакое логическое упорядочение. Существуют структуры данных, сочетающие свойства индексов и таблиц. В этих структурах при модификации данных поддерживается заранее заданное упорядочение строк по какому-либо атрибуту или комбинации атрибутов. Вообще говоря, этот набор атрибутов не совпадает с первичным ключом и даже не обязательно является уникальным.

Обычно в СУБД реализуются отдельные варианты операций, использующие преимущества таких структур для определенных классов запросов. На уровне операций эти преимущества определяются тем, что:

- количество обращений к диску при выборке по диапазону значений атрибута упорядочения определяется числом блоков, как при последовательной выборке, а не числом выбранных строк;
- если атрибут упорядочения является ключом операции соединения, то в алгоритме соединения на основе сортировки-слияния исключается фаза сортировки.

2.4.4. Представление планов

Мы уже приводили два примера представления плана в текстовой записи, которая используется в реальных промышленных СУБД. В данной книге мы будем использовать более абстрактный способ записи планов выполнения запросов, не совпадающий ни с одним из используемых в реальных системах.

Каждая операция будет представляться в скобочной записи. В тех случаях, когда это важно, перед операцией будет указываться, какой алгоритм выбирается для ее выполнения. Например, `HASH JOIN` обозначает операцию

соединения, реализуемую алгоритмом хеширования, NESTED LOOPS GROUP — операцию группировки, выполняемую с помощью алгоритма вложенных циклов, SORT UNIQUE — удаление дубликатов методом сортировки, и т. д.

Операция доступа к таблице через индекс будет обозначаться как TABLE ACCESS, а операция полного просмотра таблицы — TABLE SCAN. Аналогично операция просмотра индекса, за которой не следует доступ к таблице, обозначается INDEX ACCESS или, соответственно, INDEX SCAN.

Для улучшения читаемости каждую операцию будем размещать на отдельной строке и использовать отступы для указания уровня вложенности. Например, план может выглядеть следующим образом:

```
HASH JOIN (
   TABLE SCAN(departments),
   TABLE SCAN(employees)
   )
```

Этот план, очевидно, выполняет запрос на соединение двух хранимых таблиц.

Кроме уже упомянутых, нам понадобятся и другие операции, из которых наиболее важной является операция FILTER. Эта одноместная операция выполняет селекцию и проекцию в тех случаях, когда они не совмещены с доступом к хранимым данным. Фактически, конечно, эта операция тоже совмещается с предшествующей операцией (например, соединения), однако для наших целей важно показать, что селекция выполняется именно после операции соединения (или другой операции), а не предшествует ей.

2.4.5. Оптимизатор

Как правило, запрос, сформулированный на высокоуровневом языке запросов (в частности, на языке SQL), может быть преобразован в алгебраическое выражение различными способами, и для большинства алгебраических операций могут быть использваны различные алгоритмы (некоторые из них кратко описаны выше). Поэтому практически любой сколько-нибудь сложный запрос можно выполнить очень многими способами, т. е. для запроса можно построить несколько различных планов его выполнения. Все эти планы вырабатывают один и тот же результат, однако количество выполняемых операций и, следовательно, время, необходимое для вычисления ответа на запрос, может отличаться на несколько порядков.

Легко построить примеры запросов, для которых время выполнения различных планов варьируется от долей секунды до сотен лет и более. Такой большой разброс возможного времени выполнения показывает, насколько важна оптимизация в системах управления базами данных.

Неформально функция оптимизатора запросов состоит в том, чтобы выбрать из множества (пространства) эквивалентных планов выполнения запросов по возможности наилучший.

Как правило, наилучшим планом является тот, на выполнение которого необходимо меньшее время, однако более точные критерии могут зависеть от требований приложения. Наиболее известные варианты — время получения первой строки (или строк) ответа и время получения полного ответа. Сейчас различия между критериями оптимальности такого типа не очень существенны. Более существенно то, что оптимизатор не может непосредственно использовать подобные критерии, поскольку время выполнения становится известным только после выполнения запроса.

Поэтому реально оптимизатор выбирает или строит план на основе некоторой явно или неявно определенной функции стоимости. В зависимости от того, как именно определена функция стоимости, различаются два класса оптимизаторов (или два режима работы одного оптимизатора).

- Оптимизатор, работающий на основе правил, выполняет преобразование плана, используя тождества, связывающие алгебраические выражения, в тех случаях, когда преобразование заведомо (или почти всегда) приводит к улучшению качества плана. (Например, если селекция коммутирует с операцией соединения, то лучше селекцию выполнить раньше.)

  В этом случае оценка качества плана производится неявно: предполагается, что преобразование улучшает значение этой функции, однако фактически функция не вычисляется.

- При оптимизации на основе стоимости обязательно используется явная функция оценки стоимости выполнения запроса, которая вычисляется на основе информации о сложности алгоритма и статистических характеристик фактически хранимых данных.

  В этом случае оптимизатор имеет возможность применять не только безусловно улучшающие преобразования, но и такие, которые дают улучшение функции стоимости данного конкретного запроса, но не обязательно улучшают ее в любом случае.

Очевидно, что оптимизатор, использующий явную функцию стоимости, потенциально может получить лучший план, чем оптимизатор, преобразующий запросы только на основе безусловных правил.

Задача оптимизации запросов не может быть решена точно по нескольким причинам. Во-первых, сама функция стоимости не может быть точной, поскольку ее вычисление основано на приближенных оценках. Во-вторых, пространство возможных планов выполнения запросов очень велико, поэтому точные алгоритмы оптимизации оказываются применимыми только для относительно небольших запросов.

Поэтому любой высококачественный оптимизатор использует следующие элементы:

- правила, безусловно улучшающие план и сокращающие пространство рассматриваемых планов (например, к этому типу относится правило по возможности раннего выполнения селекций);
- эвристические правила улучшения запроса (например, следует исключать вычисление декартовых произведений, которые не нужны в окончательном результате);
- точные алгоритмы оптимизации (чаще всего — метод динамического программирования);
- различные варианты метода случайного поиска.

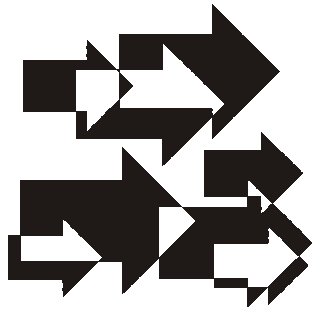
Поскольку время работы оптимизатора ограничено, а получение точного решения не имеет большого смысла, обычно оптимизаторы прекращают поиск, когда получен план с достаточно хорошей оценкой, хотя, возможно, и не оптимальный. Во многих случаях выбор плана может зависеть от случайных факторов, поэтому, вообще говоря, для одного и того же запроса могут получаться разные планы.

Это создает впечатление "непредсказуемости" поведения оптимизатора. Некоторые администраторы баз данных так сильно боятся этой непредсказуемости, что предпочитают отказаться от применения оптимизации по стоимости вообще, одновременно отказываясь от возможности получить высокую производительность системы. Во всех случаях, которые встречались авторам, переход от оптимизации на основе правил к оптимизации по стоимости приводил к существенному улучшению производительности. Иногда оказывается, что применение оптимизации на основе правил унаследовано из устаревших версий приложения, которые были рассчитаны на менее совершенные оптимизаторы запросов.

Высококачественные оптимизаторы обычно предоставляют администратору возможности управления процессом оптимизации как на уровне глобального управления стратегией оптимизации (например, указание предпочтений при выборе алгоритмов), так и на уровне отдельных запросов (подсказки для ограничения пространства просматриваемых планов).

Необходимо подчеркнуть, что при правильной настройке сервера оптимизаторы ведущих промышленных систем управления базами данных почти всегда строят очень хорошие планы выполнения запросов. Довольно часто вполне удовлетворительные результаты получаются автоматически, без какого-либо применения дополнительных средств ручной настройки сервера.

Тем не менее, иногда бывает необходима настройка отдельных запросов, когда по каким-либо причинам стратегия оптимизатора не приводит к построению эффективного плана. В *главах 6* и *7* детально обсуждаются приемы такой настройки.



Глава 3

Когда настраивать БД?

В данной главе мы уточняем, что такое настройка базы данных и какова ее роль в процессе разработки и эксплуатации системы. Основная цель этой главы — показать, что настройку необходимо предусматривать, начиная с самых ранних фаз проекта. Мы продемонстрируем, каким образом решения, принимаемые на различных этапах процесса разработки и эксплуатации системы, влияют на ее характеристики и ограничивают возможности последующей настройки.

Можно сравнить настройку системы с ее отладкой. Программный код, полученный в результате разработки проекта, может выполнять требуемые функции неправильно, и цель отладки состоит в том, чтобы добиться (по возможности) правильного выполнения функциональных требований к системе. Точно также цель настройки — приведение системы в соответствие с требованиями к производительности.

Одна из основных функций современных методологий, языков и инструментов разработки программ — предотвращение ошибок. Правильное использование этих средств существенно сокращает затраты времени и ресурсов на отладку и в то же время улучшает качество продукта. Точно также правильный учет требований к производительности на всех стадиях проектирования и разработки может значительно упростить настройку и в то же время сделать ее более результативной.

3.1. Функциональность и настройка системы

Любое хорошее руководство по разработке баз данных, а также документация по основным СУБД подчеркивает важность правильной настройки СУБД и программного приложения. Указывается, что для обеспечения эффективной работы системы необходимо выполнять определенные действия

по настройке, начиная с самых ранних фаз проектирования системы. Настройка в период эксплуатации, как правило, не может быть в такой же степени результативной.

Однако на практике довольно часто менеджер (не очень опытный) формулирует задачу специалисту по настройке примерно таким образом: "Система, которую мы сдали заказчику несколько месяцев назад, работает очень медленно, заказчики жалуются на недопустимо большие задержки при работе с базой данных. Пожалуйста, посмотрите, какие надо построить индексы". В этой формулировке содержится сразу несколько грубых ошибок:

- во-первых, скорее всего, необходимость настройки не была осознана при проектировании системы, и никаких действий по настройке во время разработки не выполнялось;
- во-вторых, проблема не была сформулирована конкретно;
- в-третьих, процесс настройки далеко не ограничивается созданием индексов. Инструменты, которыми пользуется специалист по настройке, многочисленны и разнообразны, и построение индексов далеко не всегда может дать необходимый результат.

Как объяснить такое несоответствие между рекомендациями учебников и документации по СУБД с одной стороны, и реальной практикой с другой?

Конечно, дело вовсе не в том, что квалификация менеджеров или разработчиков недостаточна. Можно утверждать, что довольно часто программисты недостаточно хорошо знакомы с тем, как следует использовать базы данных, но это обстоятельство не является определяющим.

Каждая программная система создается в рамках некоторого проекта, имеющего вполне конкретные сроки и ресурсы (более или менее точно определенные). Как руководители проекта, так и разработчики ограничены в своих действиях бюджетом времени и ресурсов проекта, а также другими условиями, и поэтому вынуждены принимать различные компромиссные решения.

Любой проект начинается с определения его целей. По отношению к проекту разработки программной системы мы можем условно разделить эти цели на две категории. Первую категорию составляют функциональные требования, т. е. описание того, что система должна делать. Ко второй категории мы относим требования к производительности системы. Чем сложнее требования (не важно, функциональные или к производительности), тем больше ресурсов необходимо для их реализации. Иначе говоря, создание высокопроизводительных систем требует значительно больших ресурсов, чем систем средней или малой производительности.

Хорошо известно, что обсуждать производительность системы, которая не выполняет необходимые функции, бессмысленно. Поэтому чаще всего основные усилия разработчиков в первую очередь направлены на реализацию

функциональности системы. Для проверки функциональности удобно использовать наборы данных небольшого размера. Выявить проблемы, связанные с производительностью, на таких наборах данных невозможно. Если участники проекта попадают под "пресс времени", то возникает опасность стихийного компромисса, сводящегося к полному игнорированию или частичному ослаблению требований к производительности.

Как правило, в любом проекте используются стандартные компоненты, блоки, типовые решения и т. п., которые существенно сокращают время разработки, но не всегда обеспечивают наивысшую производительность для проектируемой системы. Такие решения могут определяться специализацией или стандартами компании, в этом случае компромиссы принимаются еще до начала работы над проектом.

Мощность современной вычислительной техники такова, что во многих случаях даже не оптимально построенная система способна обеспечивать удовлетворительные характеристики производительности. В таких случаях настройка действительно совсем не нужна.

Тем не менее, представляется важным учитывать требования к производительности, начиная с самых ранних фаз проекта. Далее в этой главе мы обсуждаем, какие решения могут приниматься на различных фазах проекта и каким образом эти решения могут влиять на производительность системы и возможности ее настройки.

Существуют различные методологии проектирования и разработки программных систем. Иногда некоторые из обсуждаемых в этой главе фаз развития проекта могут не выделяться как отдельные этапы или, возможно, проектирование выполняется в другом порядке. Возможно, некоторые фазы сливаются вместе, все это не имеет для наших целей большого значения. Обсуждаемые здесь решения необходимо принимать в большинстве проектов приложений, использующих базы данных.

3.2. Определение и анализ требований к системе

Некоторые действия, связанные с процессом настройки, выполняются еще на фазе определения требований к системе. Эти действия состоят в том, что в состав требований включаются требования к характеристикам производительности. На последующих фазах проекта эти требования становятся основой для формулирования целей настройки.

Эти требования могут быть сформулированы с различной степенью точности и детализации, например:

- время ответа системы не должно превышать 5 с для 95% запросов при одновременной работе 200 пользователей;
- система должна обрабатывать не менее 50 000 запросов в сутки;

- сводные отчеты должны вырабатываться за разумное время;
- система должна быть высокоэффективной.

Конечно, неточно сформулированные требования значительно менее полезны, однако далеко не всегда на начальном этапе проекта могут быть сформулированы реалистические и достижимые требования с большой точностью.

Не менее важным для последующей настройки является и оценка размеров и других количественных характеристик данных, которые будут обрабатываться в системе. Очень важно максимально полно определить способы предполагаемого использования данных, а именно: какие запросы предполагается выполнять, с какой частотой, какие данные будут с наибольшей вероятностью использоваться в комбинации, каким образом будет происходить ввод данных, предполагаемая скорость роста объемов данных, обновляемость и т. п. Все это невозможно определить абсолютно точно, но чем реалистичнее будут сделаны предварительные оценки, тем эффективнее будет последующая настройка.

Такие характеристики, как примерное количество объектов каждого типа, количество и суммарный размер атрибутов, количество различных значений атрибутов, частота обновления и другие, могут быть очень важны для правильного проектирования структуры базы данных, проектирования запросов и создания возможностей для настройки системы во время эксплуатации.

Оценки ожидаемой скорости роста объемов данных и нагрузки на систему очень важны для прогнозирования того, насколько важна масштабируемость системы.

Конечно, степень детализации этих требований зависит от размеров и сложности проекта, однако даже решение о том, что никакая настройка разрабатываемой системы никогда не потребуется, и все требования по производительности будут удовлетворены автоматически, должно приниматься явным образом, с учетом степени риска.

3.3. Высокоуровневое проектирование

3.3.1. Выбор типа архитектуры

Практически для всех систем (кроме уникальных по размерам или сложности) задача выбора архитектуры очень проста, поскольку фактически надо выбирать один из двух вариантов: использование встроенной СУБД или сочетание многоуровневой архитектуры с архитектурой типа "клиент-сервер". Тем не менее, этот выбор, очевидно, очень существенно определяет не только возможную функциональность системы, но и в значительно большей мере — характеристики производительности и возможности настройки.

Фактически в большинстве случаев выбор навязывается не особенностями требований приложения, а возможностями и опытом коллектива разработчиков. Это не обязательно означает, что коллектив "подгоняет" задачу под свои возможности, скорее разработчики берутся за реализацию таких задач, которые они лучше умеют решать.

Приложения, построенные на основе встроенных СУБД, обладают рядом привлекательных свойств, в первую очередь — относительно низкой стоимостью. Тем не менее, выбирая такую архитектуру, разработчик должен понимать, что система вряд ли будет обладать какими-либо свойствами масштабируемости.

Альтернативой, как уже отмечено, является комбинация многослойной архитектуры с архитектурой "клиент-сервер". Несмотря на то, что обычно эти две архитектуры противопоставляются, практически в любой прикладной системе с развитой функциональностью приходится применять компоненты, работающие с базой данных непосредственно, минуя сервер приложений. Это вызвано тем, что многослойная архитектура, в особенности в случае использования протоколов без состояний, накладывает довольно серьезные ограничения на класс запросов, который может выполняться эффективно. В частности, это относится к функциям, выполнение которых требует значительного времени (десятки минут или часы), например, получение отчетов на основании данных большого объема.

Выбор конкретной СУБД имеет, с технической точки зрения, существенно меньшее значение. Конечно, СУБД различных производителей очень значительно отличаются по набору предоставляемых средств, организации структур хранения данных, производительности и по многим другим характеристикам, однако принципы и основные методы настройки остаются примерно одинаковыми для всех больших промышленных СУБД (разумеется, для реализации этих принципов и методов надо использовать разные средства, зависящие от конкретной СУБД).

Поскольку многие коллективы разработчиков специализируются на использовании СУБД какого-либо одного производителя, можно сказать, что выбор СУБД в большей мере определяется организационными и коммерческими причинами, чем техническими характеристиками систем.

3.3.2. Распределение функций в системе

После того как выбран тип архитектуры, который будет использован для создания приложения, необходимо определить, каким образом выполнение функции системы будет распределено между уровнями и компонентами.

Сложность этого этапа в том, что принятые решения практически невозможно изменить на более поздних фазах выполнения проекта. Поскольку никаких объективных оценок производительности на этой фазе еще нет,

решения должны приниматься исключительно на основе таких субъективных факторов, как предпочтения и опыт команды разработчиков, а также доминирующих типовых решений (продвигаемых на рынок производителями базовых программных продуктов). Как и на других ранних фазах, решение может быть навязано внешними факторами, и тогда у разработчиков просто нет выбора.

Иногда рекламная информация о продуктах принимается за объективную.

Наиболее широко распространенное типовое распределение функций может быть кратко охарактеризовано формулировкой: "Интерфейс на клиенте, логика на сервере приложений, таблицы в базе данных". Основное достоинство этой стратегии — хорошо разработанные возможности взаимодействия с объектными методологиями разработки программного обеспечения. Однако бездумное и слишком прямолинейное применение может приводить к нежелательным эффектам. Прежде всего, логическая структура базы данных полностью подчиняется особенностям кода приложения, часто игнорируются такие возможности базы данных, как проверка ограничений целостности, что приводит к снижению качества приложения в целом.

Довольно часто приложения, построенные таким способом, используют возможности СУБД крайне неэффективно. Мы подробнее обсуждаем эту проблему в *главе 4*. Можно сказать, что во многих случаях системы обеспечивают удовлетворительные характеристики производительности только вследствие избыточной мощности оборудования.

Альтернативой является архитектура, в которой значительная часть функций приложения реализована на сервере базы данных с использованием объектных возможностей современных СУБД (хранимых процедур, функций, пакетов, объектов, типов и т. п.). Достоинства таких архитектур — высокая степень логической целостности данных, идентичность представления данных для всех приложений и возможность высокоэффективной реализации. Основной недостаток — требуется более высокая квалификация команды разработчиков и необходимость более тщательной разработки структуры базы данных с учетом потребностей всех приложений, а не только интерактивных. (Как мы отмечали, обычно любая достаточно большая система включает компоненты, работающие по схеме "клиент-сервер", даже если основные функции реализованы в многоуровневой архитектуре.)

В действительности охарактеризованные выше альтернативные решения можно рассматривать как крайние решения. Фактическая реализация почти всегда занимает некоторое промежуточное положение между этими крайностями. Например, возможно использование высокоэффективных средств обработки запросов, предоставляемых СУБД, даже если хранимые процедуры и другие объектные возможности СУБД не используются. И, конечно, неадекватный стиль программирования может сделать крайне неэффективными не только

компоненты, выполняемые на сервере приложений, но и хранимые объекты в базе данных.

Иногда довольно простые на вид требования к системе приводят к существенному усложнению архитектуры. Например, требование обеспечить возможность автономной работы клиентов почти неизбежно приводит к необходимости реализовать распределенную неоднородную базу данных, сочетающую многоуровневую архитектуру на сервере и встроенные СУБД (или их заменители) на клиентах.

3.4. Детальное проектирование и разработка

Результатом детального проектирования является модель системы, формально описывающая ее структуру и выполняемые функции. Существуют различные методы, языки и методологии для построения таких моделей, обсуждение этих инструментов выходит за рамки этой книги.

Для нас, тем не менее, важно то, что разные участники процесса проектирования (или разные роли участников) могут использовать разные языки и средства для представления этой модели.

Например, разработчик программного кода приложения может использовать язык моделирования UML (Unified Modeling Language, унифицированный язык моделирования), а результатом детального проектирования может в этом случае быть диаграмма классов. В то же время для разработчика схемы базы данных более естественно строить модель в терминах диаграмм "сущность-связь".

Каждая из этих моделей отражает некоторые важные свойства системы, которые чаще всего невозможно учесть в рамках другой модели. Даже в тех случаях, когда одна из моделей генерируется на основе другой (например, диаграмма "сущность-связь" генерируется на основе диаграммы классов), обычно необходима доработка полученной модели.

Согласование моделей разного типа является одной из самых сложных задач на фазе детального проектирования. Участники проектирования должны быть готовы к поиску и принятию компромиссных решений.

Рассмотрим более детально различные задачи, решаемые на фазе проектирования.

3.4.1. Проектирование логической структуры базы данных

Общеизвестно, что логическая модель данных должна определять представление в базе данных сущностей, взаимосвязей между ними, ограничений целостности и, может быть, еще каких-то свойств данных таким образом,

чтобы обеспечить их корректность и соответствие функциональным требованиям к системе.

Наша задача — показать, что ошибки в логической структуре базы данных могут приводить не только к некорректностям в поведении системы и усложнению кода приложений, но и вызывать очень существенное ухудшение характеристик производительности.

В особенности сильное влияние на производительность могут иметь:

- выбор между естественными и суррогатными первичными ключами для сущностей;
- выбор составных или суррогатных ключей для связей;
- выбор между статическими и динамическими атрибутами;
- ограничения целостности по ссылкам (foreign key) и уникальности;
- нормализация хранимых таблиц.

Более подробно эти вопросы обсуждаются в *главе 5*.

3.4.2. Разработка приложения и проектирование запросов

Результатом разработки является программный код, реализующий требуемые функции создаваемой системы. По своему влиянию на характеристики производительности эта фаза является наиболее важной. Неправильное программирование может уничтожить любые удачные решения, принятые на предыдущих фазах проекта.

Для того чтобы построить эффективно работающее приложение, необходимо:

- правильно распределить работу между кодом приложения, написанным на языке программирования, и запросами, передаваемыми в базу данных на языке запросов СУБД (обычно SQL);
- подготовить запросы таким образом, чтобы они могли эффективно выполняться сервером базы данных.

Соотношение между языками программирования и языками запросов обсуждается в *главе 4*, а методы эффективного программирования запросов — в *главах 6* и *7*.

3.4.3. Проектирование структур хранения базы данных

Один из основных принципов технологии баз данных — независимость логической структуры данных и структуры хранения. Это позволяет изменять структуры хранения, не затрагивая ни код приложения, ни семантику выполнения запросов.

По этой причине проектирование структуры хранения является относительно изолированной задачей. Необходимость учитывать структуры хранения при проектировании запросов возникает в относительно редких случаях, например, при явном использовании материализованных представлений.

В процессе проектирования структуры хранения определяются:

- размещение данных на дисках (т. е. привязка объектов логической структуры, таких как таблицы, к табличным пространствам, файлам и т. п.);
- структуры хранения отдельных таблиц (неупорядоченные, кластеризованные и т. п.);
- атрибуты или группы атрибутов, для которых должны быть построены индексы;
- материализованные представления;
- другие структуры хранения, возможно, зависящие от СУБД.

Хотя проектирование физической структуры, безусловно, важно для достижения высокой производительности, мы не рассматриваем его детально. Вместо этого мы обсуждаем структуры хранения при освещении других тем, главным образом — проектирования запросов.

3.4.4. Тестирование

Цель тестирования — выявление несоответствий результата разработки тем требованиям, которые были установлены в начале проекта.

Если на фазе определения требований были указаны не только функциональные возможности системы, но и характеристики производительности, то, очевидно, при тестировании необходимо проверять и эти требования.

Наиболее важная особенность тестирования производительности (в отличие от функциональности) состоит в том, что для такого тестирования необходимы наборы данных реалистического размера и с близкими статистическими характеристиками. Создание таких тестовых наборов данных часто становится достаточно непростой и дорогостоящей задачей.

Сложность подготовки тестовых баз данных вызвана тем, что предсказать поведение пользователей будущей системы, а, следовательно, и статистические характеристики реальной смеси запросов оказывается практически невозможно.

По этой причине методы типа "стресс-тестов", возможно, полезные для выявления слабых мест в системе, чаще всего не помогают правильно настроить систему для работы в реальных условиях.

3.5. Эксплуатация системы

Как правило, любая успешно используемая информационная система (начиная с некоторого порога сложности и размеров базы данных) нуждается в настройке в период эксплуатации.

Мы уже говорили о том, что вопреки распространенному мнению, согласно которому настройка может быть необходима только в период эксплуатации системы, она должна начинаться с определения требований и продолжаться на всех этапах проектирования и разработки. Однако в данном разделе мы будем использовать слово "настройка" в ограниченном смысле, т. е. только на фазе эксплуатации готовой системы.

Необходимость в дополнительной настройке в процессе эксплуатации системы может быть вызвана, например, следующими факторами:

- изменение или уточнение требований к системе;
- изменение объема данных;
- изменение частоты сбора статистики;
- изменяющаяся частота выполнение запросов;
- неточности исходного планирования.

Часто необходимость настройки может возникать вследствие недостаточного внимания к требованиям по производительности в период разработки. Однако важно подчеркнуть, что меры, перечисленные в предыдущих разделах этой главы, не могут полностью исключить необходимость настройки в период эксплуатации.

Можно сказать, что тщательный учет требований к характеристикам производительности создает предпосылки для того, чтобы последующая настройка была результативной, и наоборот, никакая настройка не может компенсировать грубые ошибки, допущенные при разработке системы.

Можно рассматривать процесс настройки как отдельный проект (или подпроект), поэтому процесс настройки всегда должен начинаться с формулировки и анализа требований. Чем точнее сформулированы требования, тем лучше будут результаты настройки. Приведем несколько примеров требований.

- Необходимо, чтобы форумы были быстрее.
- Нас не устраивает время ответа 45 с, оно должно быть не больше 20 с.
- Поиск проектов по населенному пункту выполняется за 5—8 с, это приемлемое время ответа, но поиск по региону занимает 1.5—2 мин, это неприемлемо.
- Пользователи жалуются на неоправданные задержки при работе с системой, иногда до нескольких минут на операциях, которые обычно выполняются за 2—3 с.

Очевидно, требование первого типа нуждается в уточнении.

После того как требования определены, обычно необходим анализ того, что именно является причиной неудовлетворительной производительности. В современных системах со сложной многоуровневой архитектурой, в которых одновременно могут выполняться запросы десятков и более пользователей, найти узкие места обычно бывает достаточно сложно. Довольно часто причиной низкой производительности оказывается не какой-либо отдельный компонент системы, а их взаимодействие. Мы подробно рассмотрим примеры таких ситуаций в следующих главах, но уже сейчас можно отметить, что наиболее частой причиной неудовлетворительной производительности становится чрезмерное количество очень простых запросов к СУБД.

Поскольку система обычно состоит из разнородных компонентов, работающих в неоднородной среде, характеристики производительности, полученные с помощью средств профилирования и трассировки отдельных компонентов, могут давать результаты, уводящие от правильного решения. Например, тот факт, что программа на сервере приложений 99% времени ожидает результатов выполнения запросов в СУБД, не обязательно означает, что необходима настройка именно СУБД. С другой стороны, относительно высокая загрузка сервера приложений и низкая 2—3% загрузка процессора на сервере базы данных не обязательно означает, что база данных не требует настройки!

Для того чтобы правильно проанализировать причины низкой производительности, кроме реального и процессорного времени, необходимо оценивать загруженность систем ввода-вывода, загруженность сети, размеры очередей и т. п.

В некоторых случаях неудовлетворительное поведение системы не удается воспроизвести на тестовой копии и приходится выполнять измерения непосредственно на работающей системе. В подобной ситуации самым важным требованием к процессу измерений и анализа становится их неразрушающий характер: никакие действия во время анализа не должны мешать нормальной работе пользователей системы.

Тем не менее, основная сложность анализа состоит не в том, чтобы собрать необходимые данные, а в том, чтобы на основании этих данных выявить места, нуждающиеся в настройке, и доказать (на основе собранных данных), что узкие места выявлены правильно.

Другая задача, решаемая во время анализа системы, — определение того, какие значения характеристик производительности являются достижимыми при заданной конфигурации системы и нагрузке на нее. Иными словами, нужно убедиться в том, что требования к настройке реализуемы хотя бы теоретически.

В этой книге нас в основном интересуют ситуации, в которых узким местом является именно база данных, т. е. в результате анализа удалось выявить запросы или группы запросов, которые выполняются недопустимо долго.

После того как такие запросы выявлены, необходимо снова уточнить требования по настройке. Дело в том, что время ответа, которое видит пользователь системы, может быть значительно больше, чем время ответа СУБД. Необходимо также проанализировать технические причины неудовлетворительной производительности (узкие места): обмен данными с диском, процессорное время, слишком частые состояния ожидания, пропускная способность сети и т. п.

Можно выделить три класса методов воздействия на систему, позволяющих улучшить ее производительность.

- Настройка параметров СУБД, таких как размер буферного пула, число конкурентно выполняемых транзакций, количество процессов, параметры оптимизатора запросов и т. п.

  Изменения этого класса оказывают влияние на все без исключения запросы, выполняемые в СУБД, при этом улучшение характеристик для одного класса запросов может привести к ухудшению характеристик других классов.

- Изменения схемы хранения (создание или удаление индексов, материализованных представлений, кластеризации таблиц и т. п.).

  Влияние таких изменений более локально по сравнению с изменениями параметров системы: характеристики запросов, которые не используют модифицированные структуры данных, скорее всего, не изменятся.

- Модификация SQL-кода отдельных запросов, в том числе использование подсказок оптимизатору и эквивалентные преобразования кода запроса.

  Такие изменения всегда являются локальными, поскольку затрагивают только отдельные запросы.

При выполнении настройки необходимо соблюдать некоторые технологические требования, обеспечивающие воспроизводимость полученных результатов. Все изменения должны протоколироваться вместе с количественными характеристиками результата этих изменений. Необходимо проверять, что улучшения в одних частях системы не привели к ухудшению производительности других.

Наконец, настройка должна быть прекращена, когда либо установленные требования достигнуты, либо установлено, что на имеющейся конфигурации оборудования получить требуемые характеристики невозможно. (Например, получаемый объем результата не может быть передан за требуемое время при имеющейся пропускной способности сети.)

3.6. Примеры

В данном разделе мы кратко опишем примеры, которые будут использоваться в следующих главах, и попутно покажем, каким образом решения, принятые на фазе проектирования базы данных, могут повлиять на производительность системы.

3.6.1. Анализ требований и структура БД

Пример, рассматриваемый в этом разделе, очень прост и поэтому будет далее использоваться для демонстрации различных приемов настройки и преобразования запросов.

Компания HATKOMC продает услуги по размещению в Интернете информации о продавцах товаров (или услуг). База данных хранит информацию о заказчиках (т. е. продавцах) и используется для (бесплатного) подбора сведений по запросам анонимных клиентов с учетом вида требуемого товара, географического расположения продавцов и других условий, которые могут задаваться клиентами.

Мы не будем описывать базу данных или тем более систему целиком. Нам понадобятся только отдельные фрагменты базы данных для того, чтобы показать, каким образом проектные решения, принимаемые на различных фазах развития проекта, влияют на характеристики производительности.

Пока нас будет интересовать судьба только одной таблицы в этом проекте.

Кроме прочего, клиентам может предоставляться информация о номерах телефонов компаний-заказчиков и их филиалов. В то время, когда компания начала проектировать свою информационную систему, число этих телефонов не превышало нескольких десятков. В большинстве случаев предполагалось хранить только номер телефона, и лишь для нескольких филиалов дополнительно номер факса и/или пэйджера. В процессе анализа предполагаемых типов запросов к базе данных было установлено, что подавляющее большинство из них будет касаться полной информации относительно какого-то определенного заказчика, т. е. параметром запроса будет имя (или идентификатор) заказчика, а по запросу будет выдаваться вся имеющаяся информация об этом заказчике.

Далее мы покажем, как эти выводы отразились на последующих этапах проектирования, разработки и поддержки системы.

По результатам анализа требований была создана следующая таблица:

```
create table all_phones (
   company_name varchar2(100),
   phone_type varchar2(15),
   phone_number varchar2(14),
  extention varchar2(10));
```

Эта таблица дает возможность вполне эффективно выполнять, например, следующие типы запросов, которые, очевидно, обеспечивают наиболее частые потребности системы:

```
select phone_number
from all_phones
where company_name ='ABC';
select phone_number from all_phones
where phone_type='FAX' and company_name='ABC';
```

Однако в дальнейшем, в процессе развития системы к ней добавляется еще один часто встречающийся тип запросов, предназначенный для получения отчета в табличном формате:

```
select company_name, phone_number, fax_number, pager ...
```

Т. е. в одном запросе требуется получить номера телефона, факса и пейджера в разных колонках. Для знатоков SQL программирование такого запроса не составляет проблемы, но его выполнение при выбранной структуре данных, скорее всего, будет значительно менее эффективным.

В последующих главах мы рассмотрим способы ускорения подобных запросов, но здесь следует отметить, что проблемы с эффективностью будут возникать в результате недостаточно хорошо продуманного проектирования схемы базы данных, что, в свою очередь, явилось результатом недостаточно тщательно проведенного анализа требований.

Далеко не всегда и далеко не любые изменения схемы базы данных возможны, когда система находится в постоянной (24x7) эксплуатации. Однако в нашем случае история имеет счастливый конец.

Консультанту по настройке (который не участвовал в разработке системы и был привлечен только тогда, когда проблемы с производительностью системы стало невозможно решить силами сотрудников компании, разрабатывавших и сопровождавших систему) удалось найти решение, приемлемое для службы сопровождения. Оказалось возможным поменять структуру таблицы, так что она стала выглядеть следующим образом:

```
create table all_phones (
   company_name varchar2(100),
   phone_number varchar2(14),
   fax_number varchar2(14),
   pager_number varchar2(14),
   extention varchar2(10));
```

Сложность задачи, конечно, была не в том, чтобы придумать такую структуру таблицы (после того как причины низкой производительности были выявлены), а в том, чтобы разработать процесс перевода работающей системы на новую структуру без остановки системы.

Для того чтобы осуществить такое преобразование, потребовалось написать процедуру конверсии данных из старой таблицы в новую, при этом новая версия приложения должна была быть запущена в производство одновременно с выполнением конверсии данных. Таким образом, хотя преобразование было возможно, оно потребовало гораздо больших затрат, чем было бы нужно для простого изменения программы.

3.6.2. Гибкость структуры и производительность

Во втором примере мы рассмотрим ситуацию, когда компания принимает решение использовать готовый прикладной пакет, и на этапе проектирования необходимо определить, каким именно образом готовые решения могут быть применены в конкретных условиях данного предприятия.

В качестве готового пакета мы будем рассматривать Oracle Human Resouce Application [10, 9]. Этот пакет ориентирован в первую очередь на достижение максимальной гибкости системы, чтобы обеспечить возможности ее использования на предприятиях самых разнообразных профилей и разных размеров.

На рис. 3.1 приведена упрощенная схема базы данных, используемой в этом пакете. Мы будем воспроизводить фрагменты этой схемы, когда они будут нужны в примерах, и, если это необходимо, будем более детально описывать назначение отдельных таблиц и их взаимосвязи.

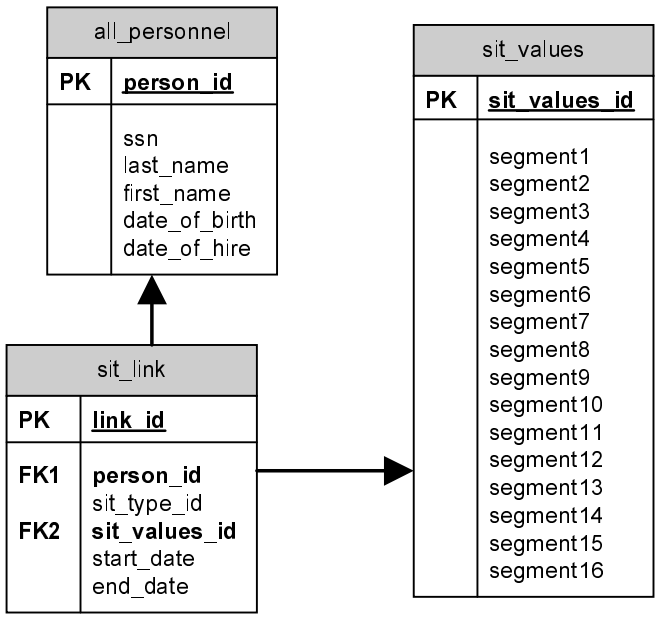


Рис. 3.1. Упрощенная схема базы данных персонала

Здесь мы обсудим только один из аспектов, связанных с применением этого пакета: соотношение между гибкостью реализации и производительностью.

Одним из существенных элементов, обеспечивающих гибкость рассматриваемого пакета, является наличие так называемых специальных информационных типов — СИТ. Эти структуры предназначены для хранения дополнительных данных о сотрудниках в тех случаях, когда эти данные не предусмотрены предопределенной в пакете структурой данных и поэтому их формат заранее не определен.

Соответствующий фрагмент схемы приведен на рис. 3.2.

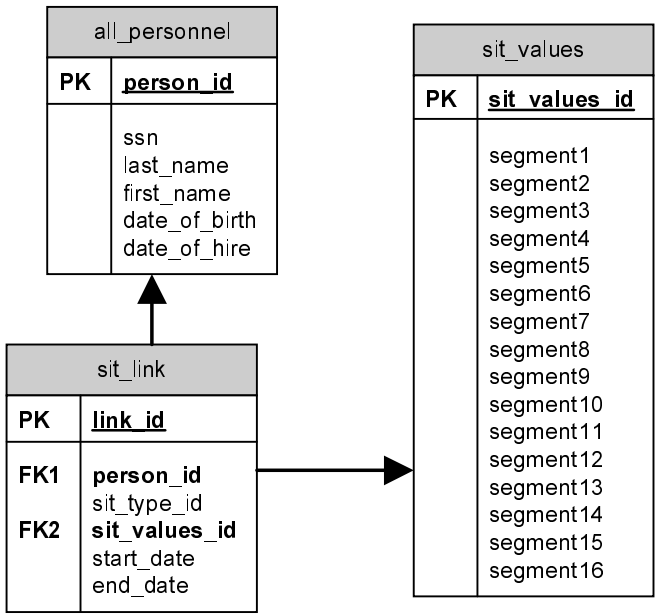


Рис. 3.2. Специальные типы данных

При использовании пакета разработчики конкретной системы могут определять различные типы СИТ, а для каждого типа указать, как именно будут использоваться сегменты данных для этого типа. При этом одни и те же сегменты в разных типах СИТ могут использоваться в разном смысле, в том числе хранить различные типы данных (разумеется, конвертированные в строковый формат).

Такая структура, действительно, обеспечивает максимальную гибкость в представлении информации и при тщательно продуманном использовании не должна создавать больших проблем, связанных с производительностью.

Однако, принимая решение об использовании этой структуры, нужно учесть результаты анализа требований. В частности, необходимо обратить

внимание на то, что, хотя на уровне базы данных не устанавливается никаких ограничений на тип информации, однако структура таблиц неявно ориентирована на то, что для каждого типа СИТ существует некоторое ограниченное число возможных комбинаций сегментов, которые могут повторяться для относительно больших групп сотрудников.

В процессе проектирования, если к нему подходить формально, может оказаться очень привлекательным использовать СИТ для любой информации, для которой не предусмотрены явно выделенные структуры в стандартных таблицах приложения.

В рассматриваемом нами примере было принято решение использовать СИТ для хранения следующих типов информации.

- Образование: название вуза, год окончания, специальность; тип записи — 1.
- Прохождение службы в вооруженных силах: дата начала, дата окончания, вид войск; тип записи — 2.
- Владение иностранными языками: первый язык, уровень владения, второй язык, уровень; тип записи — 3.

Отметим некоторые проблемы, которые возникнут в случае принятия этого решения.

- Усложнение проверки целостности данных. Один и тот же сегмент может хранить разные фактические типы данных для разных типов СИТ, однако с точки зрения базы данных все сегменты имеют тип строки символов, следовательно, контроль типов данных должен будет осуществляться на уровне приложения.
- Могут возникать ошибки преобразования данных при обработке запросов. Допустим, требуется выбрать всех служащих, которые окончили вуз после 1995 года. Такой запрос будет выглядеть приблизительно таким образом:

```
select person_id
from person_sit_link p, sit_segment_values v
where v.sit_values_id=p.sit_values_id
   and sit_type_id=1
   and to_date(segment2)>'31-dec-1995';
```

Если условие на `segment2` будет проверяться первым, то произойдет ошибка при попытке конвертировать в дату `segment2`, если тип записи — 3. Если не принять специальных мер при кодировании запроса, то порядок проверки условий будет определяться оптимизатором запросов и может изменяться в процессе эксплуатации системы. Следовательно, такие ошибки далеко не всегда могут быть выявлены даже при очень тщательной отладке.

- Производительность будет зависеть от типа наиболее часто выполняющихся запросов.

  Если в подавляющем большинстве случаев запрос будет выглядеть примерно таким образом:

```
select
   sit_type_id,
   segment1,
   segment2,
   segment3,
   segment4,
   segment5
from person_sit_link p,
   sit_segment_values v
where p.sit_values_id=v.sit_values_id
   and person_id=111;
```

  то описанная структура данных будет работать достаточно эффективно.

  Следует, однако, ожидать, что гораздо чаще потребуются результаты выполнения запросов следующего вида:

```
select
   last_name, first_name,
   college_name,
   graduation_date,
   branch,
   date_of_discharge,
   language1,
   level1
from <...>
```

  Обработка таких запросов потребует многократного соединения нескольких копий обеих таблиц. В *главе 7* обсуждаются способы более эффективного написания подобных запросов, но все же наиболее желательно вовсе не допускать появления таких проблем.

Важная особенность этого примера состоит в том, что, поскольку используется готовый пакет, изменить схему базы данных не представляется возможным.

Тем не менее, решения, принимаемые при проектировании, имеют для кода приложения (использующего готовый пакет) не меньшее значение, чем решения, принимаемые при обычном проектировании схемы базы данных. Можно сказать, что логическая структура базы данных в этом

примере надстраивается над фиксированной структурой таблиц, предопределенной используемым пакетом.

В последующих главах мы будем неоднократно возвращаться к примерам, рассмотренным в этом разделе.

3.6.3. Избыточность хранения

В данном разделе мы вводим еще один пример базы данных, который будет использоваться для иллюстрации приемов настройки. На рис. 3.3 показана упрощенная схема базы данных автоматизированного учета и оплаты командировочных расходов.

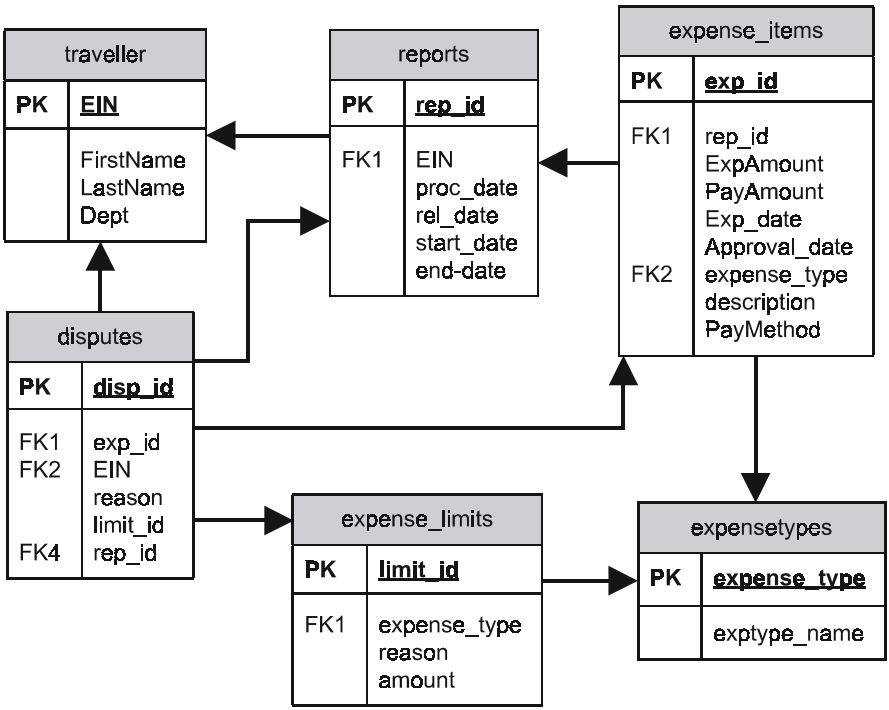


Рис. 3.3. База данных для учета командировочных расходов

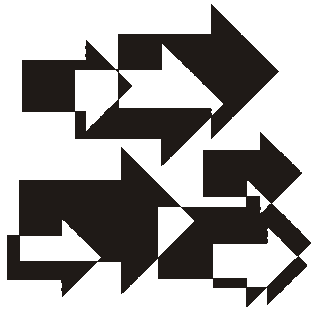
Командированные (путешественники) заполняют отчет о командировке в электронной форме и предъявляют его в систему. Каждый отчет состоит из заголовка, который хранится в таблице `reports`, и нескольких расходов, каждый из которых описывается строкой в таблице `expense_items`. Система автоматически проверяет, удовлетворяют ли произведенные расходы ограничениям, принятым в компании.

Расходы различаются по типу (проезд, проживание, питание и т. п.), в зависимости от типа расхода могут быть установлены различные ограничения, записываемые в таблицу `expense_limits`.

Если нормы расходования средств не превышены, отчет считается принятым автоматически и система выполняет банковский перевод компенсации на счет командированного немедленно. Если же какие-либо единицы расходов или отчет в целом превышают установленные нормы, то отчет считается спорным и для каждого нарушения норм делается запись (строка) в таблице `disputes`.

Администратор (не администратор системы, ответственный за функционирование оборудования и программного обеспечения, а администратор, ответственный за расходование средств) просматривает спорные отчеты и имеет возможность оплатить спорные единицы расходов или отчет в целом полностью, частично или вообще не оплачивать, если считает произведенные расходы нецелесообразными.

Особенность этой схемы состоит в том, что в ней имеется избыточность: в таблице `disputes` имеются атрибуты `expense_type`, `rep_id` и `EIN`, значения которых могут быть получены из других таблиц, используя атрибут `exp_id`. Это, безусловно, является нарушением правил нормализации, которые мы обсудим в *главе 5*, а в *главе 6* мы покажем, как эта избыточность используется для улучшения времени ответа.



Глава 4

Императивные и декларативные языки

Время выполнения программ, работающих с базой данных, а также диапазон возможностей настройки операций, выполняемых программой при обращении к базе данных, очень существенно зависят от того, каким образом написан код прикладной программы.

В этой главе обсуждаются способы структурирования кода прикладной программы (т. е. с точки зрения СУБД — клиента базы данных) для того, чтобы взаимодействие этой программы с базой данных было эффективным, и при этом сохранялись более широкие возможности для настройки (в том случае, если настройка окажется необходимой на фазе эксплуатации приложения).

Основная идея этой главы состоит в том, что при взаимодействии с базой данных следует, как правило, использовать по возможности "крупноблочные" запросы и как можно точнее описывать в запросах, какие именно данные требуются для конкретного выполнения программы, т. е. максимально использовать возможности СУБД по оптимизации и выполнению запросов.

4.1. Различия в подходах к описанию алгоритмов

Сформулированная выше идея кажется очевидной, однако ее использование на практике оказывается довольно сложным. Общепризнано, что языки запросов СУБД (практически это означает SQL) являются очень мощными и предоставляют высокоуровневый интерфейс для доступа к данным, однако использование этих языков запросов оказывается трудным для

программистов, привыкших работать на традиционных языках программирования (таких, как C++, C#, Java). Сами программисты предпочитают говорить, что язык SQL неудобен. Так или иначе, организация правильного взаимодействия кода, написанного на языке программирования, с запросами, написанными на SQL, вызывает некоторые осложнения.

Конечно, такие проблемы не возникают у высококвалифицированных программистов, в особенности у тех, кто имеет опыт сопровождения реальных прикладных систем. Однако профессия программиста уже давно стала массовой, а профессионализм высшего уровня никогда и ни в какой профессии массовым быть не может. Стандарты разработки, применяемые при реализации прикладных систем, должны быть ориентированы на разработчиков среднего уровня квалификации.

Предпосылки для возникновения этих проблем заложены в современных объектных методологиях разработки приложений, плохо учитывающих потенциал систем управления базами данных, в частности, возможности мощных языков запросов. Можно сказать, что проблемы возникают тогда, когда эти методологии применяются слишком прямолинейно.

Выделим наиболее важные в контексте данной книги причины появления таких проблем:

- языки запросов более декларативны, чем языки программирования;
- природа объектов, которыми манипулируют языки запросов, отличается от природы объектов в языках программирования;
- несоответствие объектной модели приложения и модели базы данных.

Приведенные утверждения нуждаются в пояснениях.

Языки запросов являются декларативными в том смысле, что запрос (например, оператор `SELECT... FROM...WHERE`) описывает условия, которым должен удовлетворять результат его выполнения, но оставляет большую свободу для выбора конкретного способа вычисления результата.

С другой стороны, код, написанный на языке программирования, задает последовательность операций, которые необходимо выполнить для получения результата, а не условия, которым должен удовлетворять результат. Иначе говоря, текст на языке программирования представляет собой последовательность команд. Поэтому мы будем называть эти языки императивными[1].

Это противопоставление нельзя рассматривать как абсолютное. Оптимизирующие компиляторы языков программирования могут выполнять довольно сложные преобразования программы, в каком-то смысле рассматривая ее

[1] В литературе по языкам программирования иногда различают структурные (императивные) и объектные языки, однако для нас это различие не очень существенно.

как декларативную. С другой стороны, в SQL есть конструкции (например, явные алгебраические операции `JOIN`, `UNION` и др.), которые могут рассматриваться как императивные (хотя оптимизаторы запросов имеют возможность преобразовывать и такие выражения). Можно сказать, что язык запросов декларативен примерно в такой же мере, как арифметические выражения языка программирования, не содержащие побочных эффектов.

Известны и декларативные языки программирования (например, Prolog). Как и языки запросов СУБД, они дают возможность устанавливать условия, которым должен удовлетворять результат вычислений, а алгоритм вычисления задается неявно. Эти языки не получили широкого распространения (сравнимого с популярностью объектных языков программирования), возможно, потому, что для человека императивный стиль мышления, как правило, оказывается более простым, т. е. действия описывать проще, чем правила.

Ниже мы покажем на примерах, каким образом чрезмерное использование императивного стиля приводит к появлению низкокачественных программ, как по их читаемости, так и по эффективности.

Вторая из названных выше причин — несоответствие объектов, которые обрабатываются языками запросов, и объектов языков программирования — широко обсуждалась в исследовательских публикациях по теории баз данных, начиная с середины 80-х годов, и получила метафорическое название "несоответствие импеданса". Объекты, которыми оперируют в языках запросов (множества, таблицы), являются "слишком крупными" для того, чтобы их было удобно обрабатывать в языках программирования.

Конечно, в современных объектных языках программирования можно определить (или использовать библиотечные) соответствующие типы (или классы) и ввести операции надо множествами, списками и т. п., аналогичные реляционным операциям. Именно это делается в объектно-ориентированных базах данных, например, в спецификации ODMG [3]. Тем не менее, даже в этом случае остается необходимость отображения типов данных, по крайней мере, скалярных.

Однако для того, чтобы такая схема могла работать достаточно хорошо, необходимо, чтобы код приложения был очень тесно связан с СУБД и в каком-то смысле в нее интегрирован (в частности, предполагается, что методы классов выполняются под управлением СУБД). Даже в этом случае высокая эффективность взаимодействия приложения с базой данных не достигается автоматически. Для того чтобы приложение работало действительно эффективно, необходимо использовать язык запросов OQL или объектные возможности SQL (предусмотренные в стандарте SQL:1999). Как OQL, так и SQL являются значительно более декларативными, чем языки программирования, следовательно, проблема несоответствия остается.

Наиболее распространенные архитектуры приложений такую интеграцию не допускают. Поэтому введение, скажем, аналогов реляционных операций для объектов, определенных в программе, привело бы к дублированию функциональности базы данных в приложении. Поскольку реализация полноценного процессора запросов, включающего оптимизатор, едва ли возможна в рамках проекта реализации приложения, операции над множествами в большинстве случаев выполнялись бы крайне неэффективно. На практике такие операции часто реализуются неявно (т. е. вычисление, эквивалентное реляционной операции, выполняется функциями (методами) разных классов объектов, возможно, написанными разными программистами), что также приводит к неэффективному выполнению.

Игнорирование особенностей взаимодействия с базами данных чаще всего приводит к тому, что в приложении используются очень короткие запросы (обрабатывающие только одну строку базы данных). В результате оказывается, что приложение работает слишком медленно, причем значительная часть времени (90—95% и более) уходит на работу с базой данных (точнее, на ожидание ответа сервера базы данных). Такое соотношение времени, в особенности, полученное в результате измерений, часто приводит к очевидным, но обычно неправильным выводам о том, что база данных является узким местом в системе. Причина не в том, что СУБД вообще работают медленно, а в том, что время, затрачиваемое на подготовку любого запроса к выполнению (пересылка из клиента в ядро СУБД, компиляция, оптимизация) занимает относительно много времени независимо от количества обрабатываемых данных. Кроме того, использование слишком коротких запросов фактически блокирует работу оптимизатора.

Несоответствие моделей данных приложения и базы данных приводит к необходимости описывать и постоянно использовать отображение этих моделей, что также является источником неэффективности.

Еще один аспект взаимодействия между языком программирования и языком запросов базы данных связан не с производительностью системы или скоростью программы, а с ее надежностью и устойчивостью к ошибкам.

Хорошо известно, что система, в которой компоненты созданы исходя из предположения, что другие элементы всегда работают правильно, не может быть надежной. В действительно надежной системе ни один из компонентов не доверяет другим. Применительно к взаимодействию с базами данных это означает, что запросы должны быть написаны таким образом, чтобы нарушение каких-либо ограничений, существенных для логики приложения, приводило к индикации ошибки. Например, если требуется, чтобы некоторый запрос возвращал ровно одну строку результата, необходимо, чтобы нарушение этого требования приводило к возбуждению исключительной ситуации.

Использование декларативной природы SQL позволяет сделать подобные проверки неявными.

4.2. Процедуры или запросы?

В этом разделе показано, почему лучше использовать сложные запросы, чем сложный процедурный код с большим количеством небольших запросов.

4.2.1. Курсоры

Для выполнения любого запроса к серверу базы данных приложение создает и выполняет курсор. Можно сказать, что курсор является распределенным объектом, используемым для передачи запроса из приложения на сервер базы данных и обратной передачи результатов выполнения запроса.

Рассмотрим пример использования курсора (листинг 4.1), записанный на языке Transact-SQL (применяемом в СУБД Microsoft SQL Server).

Листинг 4.1

```
--- локальная переменная ----
declare @pers_name  varchar(50)
--- Описание курсора ----
declare pers cursor for
   select name from t_person
---- Подготовка курсора к выполнению ---
open pers
--- Получение первой строки результата ---
fetch next from pers into @pers_name
--- обработка результата в цикле ---
WHILE @@FETCH_STATUS = 0
begin
   --- Обработка одной строки ---
   print @pers_name
   --- Получение очередной строки результата ---
   fetch next from pers into @pers_name
   end
--- Окончание работы с курсором ---
close pers
--- уничтожение курсора
deallocate pers
```

Язык Transact-SQL использован здесь потому, что в нем необходимо явно записывать все операции, связанные с выполнением запроса. Аналогичный код на языке Java выглядит следующим образом (листинг 4.2).

Листинг 4.2

```
    Statement stmt = conn.createStatement ();
 ResultSet rset = stmt.executeQuery (
    "SELECT name FROM t_person");
    while (rset.next ())
       System.out.println (rset.getString (1));
    rset.close();
    stmt.close();
```

Читатель легко установит соответствие между этими двумя текстами. Текстуально лишний вызов операции `fetch next` в первом фрагменте нужен только потому, что синтаксис Transact-SQL не разрешает использование этого оператора при вычислении условия окончания цикла.

В других языках многие из этих операций выполняются или могут выполняться неявно. Подчеркнем, однако, что курсор создается в любом случае, потому что другого способа передать запрос на сервер баз данных просто не существует. Отдельный курсор создается для каждого оператора SQL, но, конечно, не для каждой операции или функции, используемой для описания взаимодействия с сервером базы данных.

Так, в приведенном выше фрагменте выполняется всего один SQL-запрос

```
select name from t_person
```

Для него и создается курсор. Создание курсора, заданное операцией `declare`, в других языках может выполняться операцией `prepare` (подготовить). Операция `open` обычно называется `execute`.

Для запросов на обновление (`insert`, `delete`, `update`) курсор также необходим. Отличие состоит в том, что эти запросы не вырабатывают данных, поэтому обработка результата в цикле не требуется.

Исключением является запрос вида:

```
update t_person set ...
where current of pers
```

Этот запрос, также как и операция получения строки результата `fetch`, ссылается на уже существующий и открытый курсор, поэтому для него отдельный курсор не требуется.

Курсоры необходимы также для операций управления транзакциями (например, `commit`) и для вызовов хранимых процедур.

Создание и выполнение курсора включает несколько синхронных взаимодействий между приложением и сервером базы данных, поэтому обычно занимает относительно большое время. В противоположность этому, такие операции, как `fetch` и `update current of cursor`, не требуют синхронизации между приложением и сервером базы данных.

4.2.2. Решения, приводящие к неэффективности

Рассмотрим несколько типичных ситуаций, в которых могут возникать названные выше проблемы.

В некоторых компаниях приняты внутренние технологические стандарты, исключающие возможность эффективной работы с базами данных.

Во многих руководствах по упрощенным методикам объектного проектирования и программирования настоятельно рекомендуется ввести дисциплину взаимодействий между различными классами на основе разделения всех классов, используемых в приложении, на следующие группы:

- интерфейс конечного пользователя;
- бизнес-логика;
- взаимосвязи;
- объекты данных.

Разрешены взаимодействия только между объектами соседних уровней, и только последний уровень может непосредственно взаимодействовать с базой данных.

В результате оказывается, что возможности СУБД не используются, вместо этого эквивалентные операции реализуются менее эффективно в коде приложения. Более того, подобные неэффективности очень трудно обнаружить, потому что реализация алгоритма, эквивалентного реляционной операции, может быть рассредоточена по разным объектам, возможно, в распределенной системе выполняемым на разных машинах. Например, таким образом могут быть разнесены вложенные циклы.

Подобные решения могут давать приемлемые результаты на небольших объемах данных, однако производительность очень быстро падает с ростом объемов данных.

Система, разработанная с использованием таких стандартов, легко проходит приемо-сдаточные испытания, но оказывается малопригодной для эксплуатации. Обычное решение, предлагаемое для подобных случаев производителями ЭВМ, состоит в том, чтобы использовать более производительные процессоры или, еще лучше, устанавливать (дорогостоящие) кластеры. К сожалению, такие действия не всегда решают проблему производительности, потому что время выполнения оптимального алгоритма может отличаться от времени выполнения неоптимальных алгоритмов на несколько порядков. Например, в зависимости от используемого алгоритма операция соединения трех таблиц среднего размера, содержащих несколько миллионов строк, может занимать от нескольких минут до нескольких лет.

Дисциплина в проектировании классов, безусловно, необходима, однако лучше устанавливать такую дисциплину, которая не противоречит требованиям к проектируемой системе, в том числе и требованиям к производительности.

Существует простой и поэтому очень часто открываемый заново метод преодоления неудобств, связанных с явным использованием языка запросов: создание библиотек для работы с базами данных.

Наиболее простые и в то же время очень гибкие по своим возможностям библиотеки основаны на использовании функций (процедур, методов и т. п.) следующего вида:

```
GetDatabaseValue (Tablename, ColumnName, RowId, Result)
```

где первые три параметра задают соответственно имя таблицы, имя колонки и первичный ключ строки таблицы, а последний параметр используется для записи возвращаемого значения атрибута (или нового значения для операций обновления).

По причинам, названным выше, приложения, в которых взаимодействие с базой данных ограничивается использованием подобных функций, не могут быть эффективными.

В последнее время появились средства для работы с базой данных, в которых предпринимаются попытки организовать взаимодействие с базой данных более эффективным образом. Однако даже при использовании таких программных продуктов понимание механизмов взаимодействия может быть полезно для разработчика приложения.

4.2.3. Пример процедурной реализации

В следующем примере более детально показаны эффекты, возникающие в результате неявной замены реляционных операций на вычисления, выполняемые кодом приложения. В данном примере использован язык PL/SQL, поэтому процедуры выполняются непосредственно на сервере, а не на машине клиента. В результате исключается замедление, связанное с неэффективностью сетевых обменов при выполнении небольших запросов.

Предположим, требуется вывести список работников, которые в настоящее время работают в заданном отделе, указать для каждого из них должность, зарплату и количество пропущенных дней в текущем календарном году. Принадлежность сотрудника к отделу определяется, в соответствии с приведенной на рис. 4.1 схемой, его назначениями.

Необходимый для решения этой задачи фрагмент схемы базы данных показан на рис. 4.1. Эта схема представляет собой упрощенный вариант структуры данных для пакета Oracle Human Resouce Application.

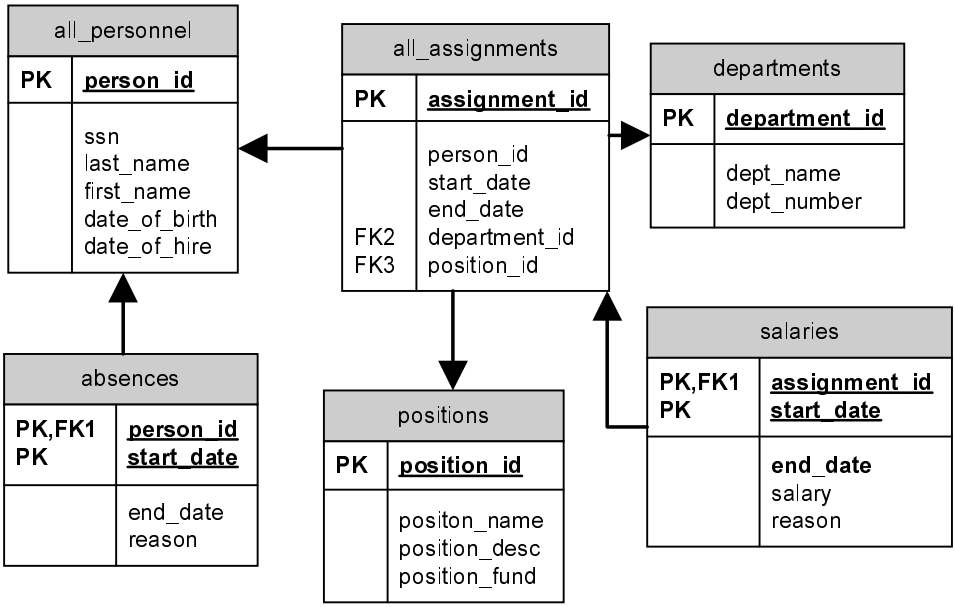


Рис. 4.1. Упрощенная схема базы данных сотрудников

Прежде всего, необходима функция, вычисляющая первичный ключ отдела по его названию (листинг 4.3).

Листинг 4.3

```
function get_dept_id_by_name(
   p_dept_name varchar2
   ) return number
is
   v_dept number
begin
      select department_id into v_dept
      from all_departments where dept_name=p_dept_name);
      return v_dept;
      end;
```

Очевидно, нам понадобится функция (или метод), проверяющий, принадлежит ли сотрудник заданному отделу (листинг 4.4).

Листинг 4.4

```
function belongs_to_dept(
   p_emp_id number,
   p_dept_id number
   ) return number
is
   v_ass_cnt number;
begin
   select count(*) into v_ass_cnt
   from all_assignments
   where person_id = p_emp_id
      and department_id= p_dept_id
      and sysdate between start_date and end date;
   if v_ass_cnt > 0 then return 1;
   else return 0;
      end if;
   end;
```

Скорее всего, при выполнении запросов, включенных в состав этой функции, будут использоваться индексы, поэтому она должна работать достаточно эффективно. Еще одна, столь же простая функция, вычисляет количество пропущенных дней (листинг 4.5).

Листинг 4.5

```
function get_absences (p_emp_id number) return number
is
   v_abs number := 0;
begin
   for a_rec in (
      select * from absences
      where person_id=p_emp_id
         and end_date > START_OF_YEAR
   loop
      v_abs := v_abs + a_rec.start_date - a_rec.start_date+1;
      end loop;
   return v_abs;
   end;
```

Поскольку для определения позиции и зарплаты необходимо знать текущее назначение, удобно использовать функцию, которая находит текущее назначение сотрудника (листинг 4.6).

Листинг 4.6

```
function get_assignment (p_emp_id number) return number
is
   v_ass number;
begin
   select assignment_id into v_ass
   from all_assignments
   where person_id=p_emp_id
      and sysdate between start_date and end_date;
   return v_ass;
   end;
```

Функция, приведенная в листинге 4.7, находит название текущей позиции.

Листинг 4.7

```
function get_pos_name (p_ass_id number) return varchar2
is
   v_pos varchar2(50);
begin
   select position_name into v_pos
   from positions
   where position_id = (
      select position_id from all_assignments
      where assignment_id=v_ass_id);
   return v_pos;
   end;
```

Наконец, для извлечения величины текущей зарплаты будем использовать функцию, приведенную в листинге 4.8.

Листинг 4.8

```
function get_salary (p_ass_id number) return number
is
   v_sal number;
begin
```

```
   select salary into v_sal
   from salaries
   where assignment_id = p_ass_id
      and sysdate between start_date and end_date;
   return v_ass;
   end;
```

С помощью этих функций можно записать главную процедуру, приведенную в листинге 4.9, выбирающую нужных сотрудников и вызывающую процедуру их обработки, которую мы не будем записывать.

Листинг 4.9

```
procedure produce emp_list (p_dept_name varchar2)
is
begin
   for emp in (select
         first_name,
         last_name,
         person_id
      from all_personnel
      where belongs_to_dept(person_id,
         get_dept_id_by_name(p_dept_name)=1
   loop
      Process_emp(
         emp.first_name,
         emp.last_name,
         get_absences(emp.person_id),
         get_position(get_assignment(emp.person_id)),
         get_salary(get_assignment(emp.person_id)),
         );
      end loop;
   end;
```

Каждая из этих функций записана достаточно компактно и, взятая отдельно, выполняется вполне эффективно. Операторы SQL очень просты и почти все могут выполняться с использованием индексов.

Однако время выполнения этой процедуры оказывается абсолютно неприемлемым по следующим причинам:

- при каждом вызове любой функции создается и выполняется отдельный курсор;

- оптимизатор СУБД не может учесть зависимости между различными операторами SQL;
- выполняется многократная выборка из таблицы `all_assignments`;
- в процедуре `emp_list` выполняется полный просмотр таблицы `all_personnel`, потому что значения функции `belongs_to_dept` не могут быть определены на основе индексов.

Последнюю из причин легко устранить. Можно несколько испортить код и включить запрос, выполняемый внутри функции `belongs_to_dept`, непосредственно в процедуру `emp_list` оператора SQL. Однако избавиться от остальных проблем, сохраняя структуру программного кода, невозможно.

В реальном прототипе этой процедуры программист, по-видимому, заботился о том, чтобы получить высокую производительность, и поэтому вместо многочисленных функций организовал все вычисления в явно выписанных вложенных циклах по следующему плану: сначала выбрать всех сотрудников, которые работают в заданном отделе, потом для каждого из них выяснить, какова его должность и зарплата, а затем во вложенном цикле выбрать все дни, когда он отсутствовал на работе, и просуммировать их. Чтобы не загромождать текст большим количеством неправильного кода, приведем только структурные операторы этой процедуры (листинг 4.10).

Листинг 4.10

```
for emp_rec in (
   select first_name, last_name, person_id
      from all_personnel
      where person_id in (
         select person_id
         from all_assignments
         where department_id in (
            select department_id
            from departments
            where department_name=p_dept_name))
      )
loop
   for ass_rec in (select assignment_id, position_id
      from all_assignments
      where person_id = emp_rec.person_id)
   loop
      for pos_rec in (select position_name
```

```
        from positions
        where position_id=ass_rec.position_id)
    loop
        --- выбрать текущую должность --
        end loop;
    for sal_rec in (select salary, start_date, end_date
        from salaries
        where assignment_id=ass_rec.assignment_id)
    loop
        --- выбрать текущую зарплату --
        end loop;
    end loop;
    for abs_rec in (select start_date, end_date
        from absences where person_id=emp_rec.person_id)
loop
    -- суммировать количество дней
    end loop;
---вывести строку результата
end loop;
```

Несмотря на то, что условия выборки в запросах оказываются более сложными, что создает некоторый простор для оптимизатора, результат получается неудовлетворительным, так как лишние обращения к таблице `all_assignments` все равно происходят и количество выполняемых курсоров примерно такое же, как при использовании функций. С точки зрения настройки, большой разницы между этими двумя способами кодирования нет.

Еще раз заметим, что в данном случае, поскольку вся программа написана на языке PL/SQL, все обращения к базе данных происходят в рамках одного процесса в ядре СУБД. Кроме этого, отсутствует необходимость в преобразовании форматов данных, поэтому этот код выполняется несколько более эффективно, чем программа, делающая аналогичные вызовы, например, из языка Java.

4.2.4. Замена вложенных циклов соединением

Покажем, каким образом можно написать приведенную выше программу с использованием более крупных (и более декларативных) запросов. В действительности для решения рассматриваемой задачи вполне достаточно одного курсора, в котором выполняется запрос, приведенный в листинге 4.11.

Листинг 4.11

```
select
   first_name,
   last_name,
   position_name,
   salary,
   (select
         sum(end_date-start_date+1)
      from absences
      where person_id=p.person_id
         and end_date > START_OF_YEAR) absence_days
from all_personnel p
   join all_assignments a
      on p.person_id=a.person_id
   join departments d
      on d.department_id = a.department_id
   join positions pos
      on pos.position_id=a.position_id
   join salaries s
      on s.assignment_id = a.assignment_id
where d.department_name = p_dept_name
   and sysdate between a.start_date and a.end_date
   and sysdate between s.start_date and s.end_date
```

В этой записи запроса использована явная операция `join`, чтобы улучшить его читаемость. Количество дней, когда сотрудник отсутствовал, подсчитывается агрегирующей функцией, которая заменяет оператор цикла. Тем самым оптимизатор получает свободу в выборе порядка просмотра таблицы `absences` (а при использовании оператора цикла последовательность обработки предписывается этим циклом). Суммирование выполняется во вложенном подзапросе, однако оптимизатор, скорее всего, преобразует запрос так, чтобы исключить вложенные подзапросы, поэтому обработка таблицы `absences` (и остальных таблиц) будет выполняться всего один раз, независимо от числа сотрудников.

После этих преобразований запрос будет выглядеть примерно так, как это показано в листинге 4.12.

Листинг 4.12

```
select
   first_name,
```

```
   last_name,
   position_name,
   salary,
   sum(end_date-start_date+1)  absence_days
from all_personnel p,
   absences abs,
   all_assignments a,
   departments d,
   positions pos,
   salaries s
where d.department_name = p_dept_name
   and sysdate between a.start_date and a.end_date
   and sysdate between s.start_date and s.end_date
   and end_date > START_OF_YEAR
   and abs.person_id=p.person_id
   and p.person_id=a.person_id
   and d.department_id = a.department_id
   and pos.position_id=a.position_id
   and s.assignment_id = a.assignment_id
group by
   first_name,
   last_name,
   position_name,
   salary
```

Мы привели эту запись для того, чтобы еще раз показать, насколько велика свобода оптимизатора при построении плана выполнения данного запроса. Какая из форм записи используется не имеет для оптимизатора никакого значения.

Оформление этого запроса в виде процедуры на PL/SQL не составляет никакой проблемы, и мы не будем приводить этот код.

4.2.5. Сравнение

Для данного примера были проведены эксперименты на больших объемах реальных данных. В экспериментах использовалась процедурная реализация (в варианте с вложенными циклами) и реализация с одним запросом. Далее эти реализации будут обозначаться `ProcNL` и `DecSQL` соответственно. Фактически использовались не тексты, приведенные в предыдущих разделах, а более сложные реальные процедуры.

При выполнении обеих реализаций на одной базе данных можно заметить следующую закономерность: если количество сотрудников данного отдела невелико, то время выполнения обоих вариантов примерно одинаково, иногда `ProcNL` выполняется несколько быстрее, чем `DecSQL`. Но по мере увеличения количества обрабатываемых записей время выполнения `ProcNL` увеличивается пропорционально их количеству, а время выполнения `DecSQL` увеличивается незначительно.

В табл. 4.1 представлено соотношение времени выполнения процедур `ProcNL` и `DecSQL` в зависимости от количества сотрудников в обрабатываемом отделе.

Таблица 4.1. *Зависимость времени выполнения от количества обрабатываемых строк*

Количество сотрудников	**2047**	**9576**	**28 942**	**41 184**	**89 813**
`ProcNL` сек.	17	32	80	110	211
`DecSQL` сек.	21	25	30	39	47

Выигрыш `ProcNL` на малых объемах результата объясняется, тем, что запрос в `DecSQL` был настроен для выполнения именно больших запросов, поскольку это было необходимо для реального использования. Никакая дополнительная настройка для данного эксперимента не производилась.

4.2.6. Обработка ошибок

Приведенный в *разд. 4.2.4* код написан, исходя из предположения, что база данных находится в корректном состоянии, т. е. в данных нет ошибок. Предположение об отсутствии ошибок в данном случае допустимо, потому что в схеме базы данных установлены необходимые ограничения целостности, гарантирующие корректность ссылок.

Ограничения целостности, однако, могут обеспечить отсутствие ошибок далеко не всегда. Кроме этого, причиной некорректного поведения могут быть и ошибки в коде запросов. Конечно, ошибки могут быть и в других частях кода приложения, но в данном разделе мы обсуждаем только те из них, которые проявляются так же, как некорректности в данных.

При наличии ошибок поведение приложения будет существенно отличаться в зависимости от того, какой из рассматриваемых подходов используется.

При процедурном кодировании одним из наиболее часто используемых стереотипов является чтение данных из курсора в цикле:

```
for next_record in (select ... from ... where)
loop
```

```
-- обработка очередной строки --
end loop;
```

В большинстве языков программирования синтаксис определения курсора может значительно отличаться, но по существу используется тот же самый стереотип.

Точно такой же стереотип используется и в тех случаях, когда требуется получить всего одну строку данных. Безопасные конструкции вида

```
select attr into local_variable from ... where...
```

имеются далеко не во всех интерфейсах передачи запросов. В данном случае безопасность состоит в том, что выполнение такого оператора приводит к возбуждению исключительной ситуации как при отсутствии данных, удовлетворяющих условию, так и при наличии нескольких строк данных, удовлетворяющих условию.

Любая из этих двух исключительных ситуаций может возникнуть вследствие ошибки в условиях запроса (на практике чаще всего это забытые условия), что, конечно, не может быть предотвращено ограничениями целостности.

Поскольку в большинстве языков программирования конструкция `select-into` недопустима, разработчики вынуждены применять обычный стереотип выборки данных из курсора, а фактически они часто используют тот же стереотип, даже если безопасная конструкция доступна. Достаточно часто разработчики программ в таких случаях не проверяют ни того факта, что хотя бы одна запись была возвращена, ни того, что, возможно, было возвращено несколько записей.

Такая программа будет правильно работать при условии, что содержимое базы данных корректно, т. е. в предположении, что все остальные части системы работают без ошибок. Это предположение, конечно, не может считаться реалистическим.

Ошибки, связанные с чтением курсоров, особенно трудно локализовать, когда такое чтение происходит внутри цикла, потому что в этом случае защита от использования неинициализированных переменных, имеющаяся практически во всех языках программирования, не помогает.

Рассмотрим пример. Предположим, что система учета командировочных расходов должна принимать на входе отчеты, подготовленные в другой системе и передаваемые по сети. Поскольку полного доверия к внешней системе нет, данные сначала записываются в таблицу `raw_input` и затем обрабатываются с целью проверки и замены внешнего представления типов расходов их внутренними кодами. Для реализации этой функциональности может быть использован фрагмент кода, приведенный в листинге 4.13.

Листинг 4.13

```
begin
   for input_rec in (select * from raw_input)
   loop
      for code_rec in (
         select expense_type from expense_types
         where expense_name=input_rec.expensetype)
      loop
         exp_ctype := code_rec.expense_type;
         end loop;
      --- Проверка корректности входной строки --
      if input_is_OK then
         insert into expense_items
            values(... exp_type,...);
         end if;
      end loop;
   end;
```

Очевидно, что если входные данные содержат некорректное значение типа расходов, то будет использовано значение кода, полученное для предыдущей записи, а исключительная ситуация будет возбуждена только в случае, если некорректное значение имеется в первой обрабатываемой строке входной таблицы.

Несложно видоизменить этот пример так, чтобы внутренний цикл мог возвращать несколько различных строк. Например, список возможных типов расходов может быть зависимым от даты, для правильного назначения кода необходимо учитывать в запросе дату расхода, а это условие может быть забыто.

Конечно, обе описанные проблемы легко решить, аккуратно обрабатывая все возможные ошибки, но это требует дополнительных затрат времени (а разработчики находятся под прессом времени почти всегда) и загромождает текст программы, делая его трудночитаемым.

Можно сказать, что появление ошибок такого рода возможно только у низкоквалифицированных программистов, однако для нас важнее тот факт, что эти ошибки спровоцированы использованием элегантной конструкции языка программирования (оператора `for…loop…end loop`).

Важное преимущество сложных запросов, содержащих операции соединения вместо вложенных циклов в приложении, состоит в том, что, с одной стороны, поведение приложения становится более управляемым, и, с другой стороны, средства управления становятся более декларативными.

В рассматриваемом примере можно отказаться от использования вложенного цикла, используя более декларативную форму записи (листинг 4.14).

Листинг 4.14

```
begin
   for input_rec in (
      select ri.*,
      (select expense_type from expense_types
         where expense_name=input_rec.expensetype) expense_type
      from raw_input ri)
   loop
      --- Проверка корректности входной строки --
      if input_is_OK then
         insert into expense_items
            values(... input_rec.expense_type,...);
         end if;
      end loop;
   end;
```

В приведенном варианте процедуры появление множественных значений вызовет возбуждение исключительной ситуации при выполнении запроса, а строки с неверными значениями типа расхода просто не будут обработаны. Это решение может оказаться вполне приемлемым, если в процессе обработки корректные строки удаляются из входной таблицы, потому что в этом случае все строки, которые в ней останутся после завершения процедуры, содержат ошибки.

Легко видоизменить этот оператор таким образом, чтобы реакция на ошибки была другой. Например, при замене вложенного подзапроса (определяющего операцию соединения) на явное внешнее соединение, можно получить неопределенные значения для строк, содержащих неправильные значения кода расходов.

```
select
   ri.*,
   et.expense_type
from raw_input ri
left join expense_types et
```

Несложно предусмотреть и необходимое поведение результата запроса в случае появления множественных строк вместо одной.

Заметим, что для решения исходной задачи можно вообще отказаться от процедурного кода, используя оператор вставки в следующей форме:

```
insert into expense_items
select
   ...
from ... where...
```

В этом случае, однако, кодирование условий проверки корректности (обозначенное комментарием в приведенном выше процедурном коде) может оказаться чрезмерно декларативным и поэтому недостаточно легко читаемым.

4.3. Когда полезен процедурный стиль?

Упомянутые недостатки императивного стиля программирования работы с базой данных не означают, что такой стиль не должен использоваться ни при каких обстоятельствах. В данном разделе мы рассмотрим несколько ситуаций, в которых программирование запроса к базе данных в императивном стиле позволяет существенно улучшить производительность.

Чаще всего такие ситуации относятся к одной из следующих категорий.

- Обработка данных существенно упрощается при использовании их упорядоченности (например, значение вычисляемой колонки результата зависит от атрибутов строки, непосредственно предшествующей текущей).
- Оптимизатор запросов не может построить план, в котором вложенные подзапросы выполняются однократно. В этих случаях предварительное выполнение вложенных подзапросов может существенно упростить основной запрос и ускорить его выполнение.

4.3.1. Темпоральные запросы

Неформально темпоральные базы данных можно определить как такие базы данных, в которых вместе с каждым значением хранится информация о времени, когда это значение было актуальным. Более точно и подробно темпоральные свойства баз данных рассматриваются в книге [4]. Темпоральные свойства могут быть добавлены в любой модели данных. В наших примерах мы будем считать, что время ассоциируется со строками таблицы. Далее положим, что каждая строка имеет атрибут `ID`, выполняющий роль первичного ключа, и атрибут `DataValue`, значения которого могут изменяться при фиксированном `ID`.

Для того чтобы хранить информацию о времени, введем атрибут `StartTS`, который будет хранить метку времени (`timestamp`) того момента, когда значение было помещено в базу данных. Например, данные в табл. 4.2 показывают со-

стояние атрибута DataValue (возможно, зарплату) для двух объектов (предположим, сотрудников компании) в некоторые периоды времени.

Таблица 4.2. *Пример темпоральных данных*

TemporalExample	ID	DataValue	StartTS
	Антон	310	1.10.2003
	Полина	325	1.02.2004
	Полина	340	1.06.2004
	Полина	350	1.03.2005
	Ольга	320	1.04.2003
	Ольга	335	1.12.2004

Мы предполагаем, что любое значение действительно до тех пор, пока оно не будет заменено другим значением, поэтому данные в этой таблице полностью описывают состояние всех объектов в любой момент времени. Однако для того, чтобы выполнять запросы более эффективно, целесообразно ввести еще один атрибут EndTS, значения которого будут хранить метку времени конца интервала. Окончательное описание нашей таблицы может выглядеть следующим образом (листинг 4.15).

Листинг 4.15

```
create table TemporalExample(
    ID varchar(20),\\
   DataValue varchar(30),
   StartTS date,
   EndTS date
   )
```

Мы можем легко вычислить значения атрибута EndTS, выполняя следующий оператор[1]:

```
update TemporalExample te1 set EndTS =
   (select min(StartTS) from TemporalExample te2
   where te2.ID= te1.ID and te2.StartTS > te1.StartTS)
```

Скорее всего, план выполнения этого оператора будет включать операции соединения и агрегирования. При этом, поскольку в условии соединения

[1] Точный синтаксис этого оператора может зависеть от СУБД.

присутствует неравенство, скорее всего, операция соединения будет выполняться с использованием алгоритма вложенных циклов.

Трудоемкость этого вычисления можно существенно сократить, если для каждого значения `ID` обрабатывать данные в порядке убывания `StartTS`, поскольку в этом случае значение `EndTS` для текущей строки просто совпадает со значением `StartTS` предыдущей строки. Поскольку реляционные языки запросов не оперируют такими отношениями, как "следующий" или "предыдущий", для реализации этой идеи необходимо использовать явный оператор цикла (листинг 4.16).

Листинг 4.16

```
declare
   PrevID varchar(20) := NULL,
   PrevTS date := NULL;
begin
   for cur in (select * from TemporalExample
         order by ID, StartTS desc)
   loop
      if cu.ID <> PrevID then
         PrevTS := NULL;
         end if;
      update TemporalExample set EndTs=PrevTS
      where ID=cur.ID and StartTS=cur.StartTS;
      PrevTS := cur.StartTS;
      PrevID := cur.ID;
      end loop;
   end;
```

При выполнении этого блока, очевидно, выполняется операция сортировки, имеющая примерно такую же сложность, как операции агрегирования, и затем однократный просмотр результата сортировки. Заметим, однако, что для обновления каждой строки необходимо открывать отдельный курсор. Мы можем ожидать, что при больших размерах таблицы оператор цикла будет значительно более эффективным, чем приведенный выше оператор обновления, однако при небольших размерах таблицы или при небольшом количестве обновляемых строк более эффективным может оказаться декларативный оператор обновления.

Структура данных, которую мы здесь рассматриваем, допускает некоторую избыточность. В случае если строки, описывающие состояния одного объекта в смежные интервалы времени, содержат одинаковые значения всех

атрибутов, кроме меток времени, то такие строки можно заменить на одну без потери информации. Например, строки, содержащиеся в табл. 4.3, можно заменить одной строкой, в которой `StartTS` и `EndTS` имеют значения 01/05/2003 и 01/09/2004 соответственно.

Таблица 4.3. *Избыточные строки в темпоральной таблице*

TemporalExample	ID	DataValue	StartTS	EndTS
	Сергей	400	1/05/2003	1/06/2003
	Сергей	400325	1/06/2003	1/08/2003
	Сергей	400340	1/08/2003	1/09/2004

Операция, устраняющая избыточность такого вида, называется операцией темпоральной склейки. Разумеется, можно предотвратить избыточность хранимых данных в момент внесения изменений, однако избыточные строки могут появляться при выполнении реляционных операций над темпоральными данными, поэтому операция склейки необходима.

Мы можем записать операцию склейки декларативно. Результат склейки описывается как строка, полученная из двух строк исходной таблицы, таких, что:

1. Не существует других значений `ID` и `DataValue` в интервале времени от `StartTS` первой строки до `EndTS` второй.
2. Значение `StartTs` первой строки минимально среди строк, удовлетворяющих условию 1, т. е. не существует строк, удовлетворяющих условию 1, с меньшим значением `StartTS`.
3. Значение `EndTS` втрой строки максимально среди строк, удовлетворяющих условию 1.

На языке SQL эти условия можно записать следующим образом (листинг 4.17).

Листинг 4.17

```
select te1.ID, te1.DataValue, te1.StartTS, te2.EndTS
from TemporalExample te1, TemporalExample te2
where te1.ID = te2.ID and te1.DataValue = te2.DataValue
   and not exists(
      select * from TemporalExample
      where (ID <> te1.ID or DataValue=te1.DataValue)
         and StartTS > te1.StartTS and EndTS < te2.EndTS)
```

```
and not exists (
   select * from TemporalExample
      where ID=te1.ID and DataValue = te1.DataValue
      and StartTS < fe1.StartTS
      and not exists(
         select * from TemporalExample
         where (ID <> te1.ID or DataValue=te1.DataValue)
            and StartTS > te1.StartTS and EndTS < te2.EndTS))
and not exists (
   select * from TemporalExample
      where ID=te1.ID and DataValue = te1.DataValue
      and EndTS > fe2.EndTS
      and not exists(
         select * from TemporalExample
         where (ID <> te1.ID or DataValue=te1.DataValue)
            and StartTS > te1.StartTS and EndTS < te2.EndTS))
```

Поскольку предикат `not exists` соответствует операции (анти)соединения, выполнение приведенного выше запроса включает вычисление шести операций соединения.

Если, как и при вычислении `EndTS`, использовать отношения предшествования в списке строк, упорядоченном по значениям меток времени, то результат склейки может быть получен за один просмотр этого списка, т. е. сложность этого вычисления (почти) эквивалентна сложности сортировки.

4.3.2. Выделение подзапросов

В некоторых случаях можно очень существенно ускорить выполнение запроса, предварительно выполнив некоторые содержащиеся в нем вычисления в отдельном запросе. Если эквивалентность получаемых результатов основана на соотношениях реляционной алгебры, то такие преобразования могут выполняться оптимизатором запросов и поэтому нет необходимости о них особо заботиться. Нас интересуют случаи, когда эквивалентность имеет место, но не может быть получена в рамках реляционной алгебры.

Один из таких случаев представляет собой реляционная операция деления. Эта операция очень редко используется на практике, потому что не включена в явном виде в язык SQL. Мы не будем обсуждать, насколько полезна эта операция, но покажем, каким образом она может быть вычислена, на примере.

Рассмотрим таблицу, содержащую результаты экзаменов (скажем, в студенческую экзаменационную сессию). Вероятно, эта таблица должна содержать

колонки с оценками, датой сдачи экзамена и еще какие-нибудь атрибуты, но для нас сейчас важны только две колонки: идентификатор студента (например, номер зачетной книжки) и код сданного экзамена. Описание таблицы может выглядеть следующим образом:

```
reate table StudExam (
   StudID varchar(10),
   ExamID varchar(10)
   )
```

Мы предполагаем, что каждый экзамен сдан хотя бы одним студентом. (Если это не так, можно заменить данную таблицу внешним соединением с таблицей, содержащей список экзаменов.)

Предположим, что требуется получить список студентов, которые сдали все экзамены. Такой список является результатом операции деления, но, поскольку в языке SQL операции деления нет, такие запросы обычно записываются в другой форме (листинг 4.18).

Листинг 4.18

```
Select distinct StudID from StudExam SE1
where not exists (Select * from StudExam SE2
   where not exists(select * from StudExam SE3
      where SE1.StudID=SE3.Stud.ID
         and SE2.ExamID=SE3.ExamID))
```

Этот запрос эквивалентен операции деления в рамках реляционной алгебры (докажите!). При выполнении данного запроса вычисляются два антисоединения и затем операция устранения дубликатов.

Можно получить тот же результат более эффективным методом, если переформулировать запрос следующим образом: получить список студентов, которые сдали столько экзаменов, сколько их имеется в таблице. Для того чтобы определить количество экзаменов, сданных студентом, нужно использовать операцию группировки, которая выводит нас за пределы реляционной алгебры (листинг 4.19).

Листинг 4.19

```
select StudID from StudExam
group by StudID
having count(ExamID)=(
   select count(distinct EXamID)
   from StudExam)
```

К сожалению, некоторые оптимизаторы не могут правильно преобразовать такой запрос, поэтому оказывается полезным выполнить вложенный подзапрос отдельно, т. е. сначала вычислить количество экзаменов и записать его в переменную программы, а затем использовать это значение в основном запросе для сравнения со счетчиком экзаменов, сданных студентом.

4.3.3. Удаление дубликатов

Материал этого примера возник в результате серии неточностей и ошибок, допущенных менеджерами, проектировщиками и разработчиками в различных проектах.

Мы снова обращаемся к проекту создания прикладной системы с использованием готового пакета Oracle Applications, упомянутого ранее в *разд. 3.6*. Напомним, что в этом пакете особенности конкретного приложения можно учесть, вводя специальные информационные типы (СИТ). Все значения всех СИТ хранятся в таблице `sit_segment_values`, а персональные записи связываются со значениями СИТ с помощью таблицы `person_sit_link`, которая фактически связана с таблицей `sit_segment_values` отношением "внешний-первичный ключ", однако почему-то проектировщики пакета не определили такое ограничение в базе данных, оно проверяется в коде пакета.

Авторы пакета предполагали, что абсолютно все операции над данными будут производиться только через процедуры пакета, и предоставили для этого все необходимые операции манипулирования данными, т. е. инкапсулируют базу данных в объектную оболочку.

По причинам, рассмотренным ранее в данной главе, эти процедуры не могут работать эффективно. Исключений нет и для больших производителей программных продуктов. Тем не менее, время реакции оказывается вполне приемлемым при относительно небольших объемах данных, т. е. если СИТ используются действительно для регистрации исключительных свойств объектов.

В проекте приложения, который мы рассматриваем, количество записей в таблице связей составило несколько миллионов, что нельзя считать небольшим объемом. В результате процесс загрузки данных из устаревшей системы, базирующейся на процедурах пакета, оказался недопустимо медленным.

Этот факт был осознан руководством слишком поздно, поэтому разработчики, находившиеся "под прессом времени", при прямой загрузке данных в БД (минуя процедуры пакета) записывали каждое значение СИТ столько раз, сколько оно использовалось, т. е. связь между таблицами стала взаимнооднозначной.

При проверке результатов выяснилось, что разработчики пакета подготовили еще одну ловушку. Значения СИТ должны быть уникальными, иначе процедуры пакета работать не могут. Это типичное ограничение целостности `unique`, однако оно также не определено в базе данных, а проверяется в коде пакета.

Теперь, наконец, можно сформулировать задачу удаления дубликатов.

В таблице `sit_segment_values` имеются строки с совпадающими значениями всех атрибутов, кроме уникального идентификатора `sit_value_id`, значения которого генерируются при добавлении строк. Здесь и далее мы называем такие строки логическими дубликатами. Необходимо устранить логические дубликаты, т. е. оставить в таблице только одну из строк-дубликатов и соответствующим образом изменить ссылки в таблице `person_sit_link`.

Для решения этой задачи в качестве оставляемой строки можно выбрать ту, в которой значение `sit_value_id` минимально в этой группе дубликатов. Тогда задача может быть решена с помощью следующих двух запросов на языке SQL (листинг 4.20).

Листинг 4.20

```
update person_sit_link
set sit_values_id = (
   select min(sit_values_id) from sit_segment_values
   where (sit_type_id, segment1, segment2,...,segment30) in (
      select sit_type_id, segment1, segment2,...,segment30
      from sit_segment_values
      where sit_values_id = person_sit_link.sit_values_id));
delete from sit_segment_values
where sit_values_id not in (
   select sit_values_id from person_sit_link);
```

Осложнение, конечно, заключается в том, что обе таблицы содержат миллионы строк. Требуется выполнять их соединение, причем в первом операторе таблица значений участвует дважды. Поскольку дубликатов довольно много, соединение трех таких таблиц будет содержать триллионы записей и его вычисление потребовало бы слишком много времени.

Фактически разработчики попытались реализовать примерно такой же алгоритм процедурно, используя вместо минимального первое встретившееся в таблице `person_sit_link` значение указателя. Это, конечно, не могло привести к сокращению времени обработки (хотя и позволяло сегментировать

процесс на отдельные части, что хорошо по технологическим причинам). По оценкам, этот процесс занял бы несколько суток.

Правильное решение должно быть основано на использовании упорядочения по значениям всех атрибутов таблицы `sit_segment_values`.

Поскольку в результате упорядочения все логические дубликаты оказываются в соседних позициях, для их удаления достаточно одного просмотра. Конечно, этот факт используется и в обычной операции группировки, реализованной в процессоре запросов, но нам необходимо одновременно изменять ссылки на удаляемые записи, поэтому требуется реализовать операцию удаления процедурно (листинг 4.21).

Листинг 4.21

```
declare
   cursor cur is
      select
         sit_type_id, segment1, segment2, segment3,
         <...>,
         segment30,
         sit_values_id
      from sit_segment_values
      order by 1,2,3,...,30,31;
   first_dup_rec cur%rowtype;
   rec_count integer:=0;
begin
for rec in cur
loop
   if first_dup_rec.sit_type_id is not null
      AND compare_sit_values(first_dup_rec,rec)
   then
      update person_sit_link
         set sit_values_id=first_dup_rec.sit_values_id
         where sit_values_id=rec.sit_values_id;
      delete sit_segments_values
      where sit_values_id=rec.sit_values_id;
   else
      first_dup_rec:=rec;
      end if;
   rec_count:=rec_count+1;
```

```
   if rec_count >=10000 then
      rec_count:=0;
      commit;
      end if;
   end loop;
commit;
end;
```

В этом фрагменте кода использована функция `compare_sit_values`, возвращающая `true`, если ее аргументы являются логическими дубликатами.

Отметим, что этот алгоритм обладает хорошей повторяемостью и отказоустойчивостью. Единственное условие, необходимое для корректности рестарта, состоит в том, что операции `update` и `delete`, выполняемые на одной итерации цикла, выполнялись в одной транзакции.

В случае если для какого-либо значения СИТ логических дубликатов нет, то никакие операции обновления не выполняются. Это, разумеется, верно и в том случае, если дубликаты были удалены до рестарта. Поэтому значение счетчика, по которому выполняется фиксация, может быть любым.

Выполнение этого алгоритма заняло менее часа, рестарты не понадобились.

4.3.4. Массовая обработка данных

Необходимость процедурной работы с базой данных в предыдущих примерах данного раздела была связана с использованием естественного порядка данных, т. е. отношений "предыдущий" и "следующий" на обрабатываемых объектах (строках) данных. Эти отношения не могут быть выражены средствами реляционной модели данных, поэтому процедурная обработка в этих случаях неизбежна.

Другой причиной, по которой процедурная обработка может оказаться желательной, является большой объем выполняемых вычислений или обновлений. Несмотря на то, что и в этом случае, скорее всего, выполнение всех необходимых действий в рамках одного запроса будет вычислительно более эффективно, оно может оказаться технологически неприемлемым по другим причинам, приведенным ниже.

- Если вычисления занимают продолжительное время, возможно, целесообразно разделить это вычисление на несколько этапов и сохранять промежуточные результаты, чтобы в случае необходимости процесс вычислений мог быть возобновлен с определенной точки, а не с самого начала.
- Любой SQL-запрос выполняется в рамках одной транзакции. Недорогие и средние СУБД используют простейшие механизмы управления замками, сильно ограничивающими конкурентный доступ, поэтому длитель-

ное выполнение отдельных запросов (и, следовательно, транзакций) может приводить к существенным задержкам в работе других пользователей системы.

Необходимо подчеркнуть, что такие действия, как снижение уровня изоляции в СУБД или отмена использования замков, приводят к снижению надежности работы системы и не могут рассматриваться как метод повышения производительности.

При реализации процесса вычислений как серии относительно небольших запросов необходимо заботиться о том, чтобы не подменять функциональность СУБД в коде приложения. Наибольшая опасность, как мы уже упоминали ранее, связана с использованием вложенных циклов.

4.3.5. Необязательные параметры

В некоторых приложениях необходимо выполнять запросы, в которых используется большое количество необязательных параметров, определяющих условия селекции. Обычно каждый из параметров задает условие, так или иначе ограничивающее значения одного из атрибутов таблицы, из которой извлекаются данные, или другой связанной с ней таблицы. Отсутствие какого-либо из параметров интерпретируется при этом как отсутствие ограничений на соответствующий атрибут, а условия на атрибуты, для которых ограничивающие параметры заданы, связываются в единый предикат селекции логической операцией конъюнкции (т. е. `AND`).

Например, в системе может быть реализован поиск сотрудника по началу фамилии, по имени, по диапазону даты поступления на работу, по должности в настоящее время или по любому сочетанию этих условий.

Программирование таких запросов на статическом SQL вызывает некоторые осложнения, потому что условия, для которых не заданы значения параметра, не должны влиять на результаты выполнения запроса. Обычный в подобных случаях прием состоит в том, что условие на атрибут `attr`, зависящее от параметра `p_attr_value` (допустим, это проверка на равенство), записывается в следующем виде:

```
(p_attr_value is NULL or attr=p_attr_value)
```

Выражения такого вида для каждого из необязательных параметров связываются операцией `AND`, что дает правильный результат селекции.

Этот метод обладает очень существенным недостатком: во многих случаях появление логической операции `OR` блокирует использование индексов, даже если они имеются для атрибута `attr` и могут быть использованы с целью выборки по основному условию. Поэтому подобный метод можно использовать только для небольших таблиц или в тех случаях, когда имеется другой, обязательный критерий селекции, отбор по которому поддерживается

индексами, при этом необязательные параметры используются для уточнения результатов отбора по основному критерию.

Если же все критерии являются необязательными или если обязательные критерии не ограничивают множество строк до небольшого количества, необходимо использовать другие решения.

Наиболее подходящим в этом случае является, по-видимому, формирование запроса и выполнение его как динамического. В итоге достигается разумный баланс между построением запроса на процедурном языке и его выполнением как декларативного.

Конечно, этот способ тоже не лишен недостатков. Количество различных запросов, которые могут быть сгенерированы таким образом, экспоненциально зависит от числа необязательных параметров, поэтому не представляется возможным настроить их все. Следовательно, необходимо либо выявить часто используемые комбинации параметров и выполнять настройку в предположении, что они заданы, либо надеяться на то, что настройка вообще не потребуется.

4.4. Правила гранулярности запросов

Примеры, приведенные в этой главе, показывают важность правильного распределения функций между языком программирования и языком запросов. Как слишком мелкие, так и слишком сложные запросы могут оказаться неэффективными.

В этом разделе мы сформулируем правила, которые помогают должным образом выбрать это соотношение.

В первую очередь эти правила применимы для приложений, в которых требуется малое время ответа системы (чаще всего это интерактивные приложения). Не все из этих правил применимы для приложений, выполнение которых занимает значительное время.

Правило 1. *Количество операторов SQL, передаваемых в СУБД в процессе обработки одного запроса пользователя, не должно зависеть от количества строк таблиц, извлекаемых из базы данных и необходимых для выполнения этого запроса.*

Это правило рекомендует не выполнять SQL-запросы внутри циклов, обрабатывающих курсор. Если в процессе обработки строк одной таблицы необходимы данные из других, можно сделать одно из двух:

- открыть один курсор, в котором выбрать из базы данных все необходимые данные, используя операцию соединения;
- открыть один курсор для выборки из главной таблицы и еще один курсор для выборки данных из подчиненных таблиц, необходимых для

обработки всех строк главной таблицы (также используя операцию соединения) и упорядочить результат по ключу главной таблицы.

Напомним, что операция получения строки данных из курсора не рассматривается как отдельный запрос к СУБД (и не является таковым). Важно именно количество SQL-операций, т. е. курсоров, а не строк, обработанных в этих курсорах. (Технически причина состоит в том, что результаты выполнения запроса передаются в буферную память приложения асинхронно.)

Это правило иногда бывает трудно проверить, так как внутренние курсоры могут открываться в методах других объектов, вызываемых из основного цикла. Для диагностики подобных ситуаций необходимо использовать трассировочные средства СУБД.

В случае, когда обработка требует очень большого времени, это правило может быть ослаблено: *Количество операторов SQL не должно быть пропорционально количеству извлекаемых строк.*

Так, если выполняется обработка строк главной таблицы, и для каждой ее строки обрабатываются несколько детальных строк, то, возможно, допустимо выполнять отдельный оператор SQL для каждой строки главной таблицы, но недопустимо — для каждой строки таблицы деталей.

Правило 2. *Идентичные операторы* `SELECT` *с идентичными параметрами не должны выполняться многократно.*

Нарушения этого правила возникают в том случае, если приложение не сохраняет в своих локальных переменных справочные значения, полученные из базы данных (например, тексты, размещаемые на генерируемой форме, или другие параметры генерируемой формы).

Опасность нарушения этого правила увеличивается, если программа хорошо структурирована и состоит из большого количества небольших по размеру процедур (функций, методов и т. д.).

Правило 3. *Если некоторая строка таблицы передается в приложение, то в том же запросе должны быть получены все поля этой строки, необходимые для завершения выполнения запроса пользователя.*

Это правило рекомендует никогда не использовать широко распространенный стереотип, в соответствии с которым сначала выбирается первичный ключ необходимой строки таблицы (обычно — суррогатный ключ), а затем отдельным оператором SQL выбираются другие поля из этой строки.

Конечно, маловероятно, что следующие два оператора SQL будут размещены рядом:

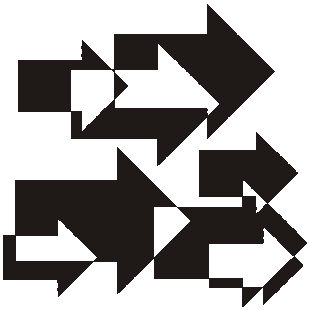
```
select pers_id into LocalVariable from PERSON where name='Леонид'
select * from PERSON where pers_id=LoacalVariable
```

Однако вполне может оказаться, что они размещены в разных процедурах, написанных разными участниками проекта, и значение `LocalVariable` передается в качестве параметра.

Правило 4. *Все условия на выбираемые строки таблицы следует включать в оператор SQL.*

Идея "прочитать все данные в память и потом быстро найти то, что нужно для выполнения запроса клиента" привлекательна тем, что позволяет ограничиться программированием на одном языке, т. е. фактически исключает язык запросов. Эта идея может неплохо работать для данных небольшого объема, таких как справочники государств или областей. Может оказаться, что однократное считывание всей таблицы занимает меньше времени, чем выборка двух строк отдельными операторами `SELECT`.

В общем случае, однако, механизмы поиска, реализованные в СУБД, окажутся более эффективными, чем код, написанный в приложении.



Глава 5

Нормализация или высокая производительность?

В данной главе обсуждается процесс проектирования базы данных в контексте настройки системы.

5.1. Модель приложения и схема базы данных

Массовое применение объектных методологий проектирования и разработки программного обеспечения привело к тому, что важность проектирования структуры базы данных часто недооценивается, этап ее проектирования исключается из процесса разработки или используются упрощенные методы проектирования базы данных.

Обычно схема базы данных генерируется из объектной модели приложения. Полученная таким образом схема зачастую не учитывает специфические особенности баз данных, которые трудно описать в объектной модели и, как правило, нуждается в существенной доработке.

Многие элементы и конструкции объектной модели (такие как наследование и агрегирование) могут быть представлены в базе данных различными способами. Неправильный выбор этих представлений способен отрицательно повлиять как на сложность разработки приложения, так и на характеристики готовой системы.

Довольно часто схема базы данных строится на основе шаблонов или стереотипов, привычных для разработчиков (или определяемых стандартами компании). В качестве примеров таких шаблонов можно привести способы определения суррогатных ключей, способы отображения иерархии классов

объектов, выбор типов данных и т. п. Само по себе использование шаблонов и стереотипов вполне допустимо и желательно, поскольку до некоторой степени унифицирует структуру баз данных, разрабатываемых в компании, а также сокращает потребность в ресурсах, необходимых для реализации системы.

Проблема состоит в том, что довольно часто отсутствие понимания причин, по которым появился тот или иной прием, и условий его применимости подменяется религиозным фанатизмом и верой в его универсальность. Мы обсудим некоторые из таких шаблонов и стереотипов и покажем, каким образом их выбор может повлиять на производительность и другие характеристики создаваемой системы.

Будем различать два уровня описания структуры базы данных:

- логическое описание структуры данных;
- описание структуры хранения.

В классической теории баз данных выделяется еще и концептуальный уровень описания данных, однако для наших целей можно считать, что роль концептуального описания выполняет объектная модель приложения.

При строгом разделении логического уровня и уровня хранения противопоставление, вынесенное в заголовок главы, полностью теряет смысл, потому что нормализация относится к уровню логической структуры, а эффективность определяется структурой хранения.

На практике, однако, по готовой схеме базы данных бывает довольно сложно определить, что в ней относится к логическому уровню, а что — к уровню хранения. Происходит это не (только) потому, что разработчики недостаточно тщательно проектируют базу данных, а потому, что язык описания данных SQL не дает возможности для такого разделения. Это и создает условия для смешивания уровней и приводит к появлению термина *денормализация*, на самом деле обозначающего материализацию некоторых видов избыточных данных, т. е. таких, которые можно вычислить на основе других хранимых данных. Мы обсудим далее, при каких условиях этот прием может приводить как к повышению, так и к снижению производительности.

5.2. Нормализация

Рассмотрим пример, приводимый, вероятно, абсолютно во всех учебниках по системам управления базами данных. Пусть имеется таблица, определенная следующим образом (листинг 5.1).

Листинг 5.1

```
create table t_empd (
   emp_id,
   emp_name,
```

```
   dept_id,
   dtptmgr_id,
   dept_name)
```

Синтаксис SQL требует, чтобы для каждой колонки был определен, по крайней мере, ее тип, однако для того, чтобы подчеркнуть, что сейчас мы рассматриваем логическую структуру данных, просто предположим, что значения всех атрибутов лежат в соответствующих доменах, а конкретные типы можно определить позже.

Определенная таким образом таблица обладает целым рядом неприятных свойств:

- данные хранятся избыточно (название отдела повторяется для каждого сотрудника отдела);
- невозможно сохранить информацию об отделе, в котором (возможно, временно) нет сотрудников;
- при обновлении информации об одном отделе необходимо изменять несколько строк таблицы.

Все эти недостатки можно устранить, заменяя исходную таблицу двумя ее проекциями (листинг 5.2).

Листинг 5.2

```
create table t_emp (
   emp_id,
   emp_name,
   dept_id)
create table t_dept (
   dept_id,
   dtptmgr_id,
   dept_name)
```

Вторая таблица `t_dept` описывает некоторые классы объектов реального мира, а именно — отделы. Таким образом, новая схема лучше отражает свойства реального мира, чем исходная, в которой этот класс объектов не был представлен.

При этом исходная таблица может быть восстановлена с помощью операции соединения, поэтому потери информации не происходит (листинг 5.3).

Листинг 5.3

```
Select
   emp_id,
   emp_name,
```

```
   dept_id,
   dtptmgr_id,
   dept_name)
from t_emp e join t_dept d on e.dept_id=d.dept_id
```

Эта операция соединения и является причиной того, что нормализация вообще упоминается в данной книге. Хотя потери информации не происходит, при использовании нормализованных таблиц, вообще говоря, требуется выполнять больше операций. Очевидно, поэтому можно ожидать, что выборка данных из нормализованной схемы будет занимать больше времени. Несколько менее очевидна возможность того, что потребуется меньше времени, поскольку таблицы получились меньшего размера не только по количеству атрибутов, но и по количеству строк.

5.2.1. Функциональные зависимости

Более формально теория нормализации основана на анализе функциональных зависимостей (кроме функциональных, рассматриваются и другие типы зависимостей, но это для нас сейчас несущественно).

Говорят, что атрибут X функционально зависит от совокупности атрибутов ABC, $(ABC->X)$, если для любой возможной комбинации значений ABC существует только одно возможное значение X.

Совокупность атрибутов, от которой функционально зависят все атрибуты отношения, называется ключом.

Нежелательные свойства возникают в тех случаях, когда в отношении имеются функциональные зависимости между атрибутами определенного вида. Говорят, что схема отношения находится в нормальной форме, если нежелательные зависимости в этой схеме отсутствуют. Всего известно 5 нормальных форм, отличающихся тем, какие типы нежелательных зависимостей в них запрещены.

Неформально нормализация сводится к декомпозиции отношений на их проекции с возможностью последующего восстановления исходных таблиц. Более формально существуют алгоритмы синтеза нормализованных схем на основе множества атрибутов и функциональных зависимостей.

В примере, приведенном в начале этого раздела, имеются зависимости атрибутов `dept_name` и `deptmgr_id` от атрибута `dept_id`, который сам зависит от атрибута `emp_id` (являющегося ключом этого отношения). В результате декомпозиции атрибуты, зависящие от неключевого атрибута, перенесены в другое отношение, и таким образом, нежелательная зависимость устранена из исходного отношения.

5.2.2. Функциональные зависимости и семантика прикладной системы

Необходимо уточнить, какие функциональные зависимости нужно учитывать. Прежде всего, все рассматриваемые зависимости должны отражать законы или правила, выполняющиеся в реальном мире. В примере функциональная зависимость отражает тот факт, что в каждом отделе имеются ровно одно название и ровно один руководитель.

Рассмотрим немного более сложный пример. В схеме базы данных персонала имеется фрагмент, описывающий взаимосвязи между поручениями, отделами, должностями и местоположением. Этот фрагмент приведен на рис. 5.1.

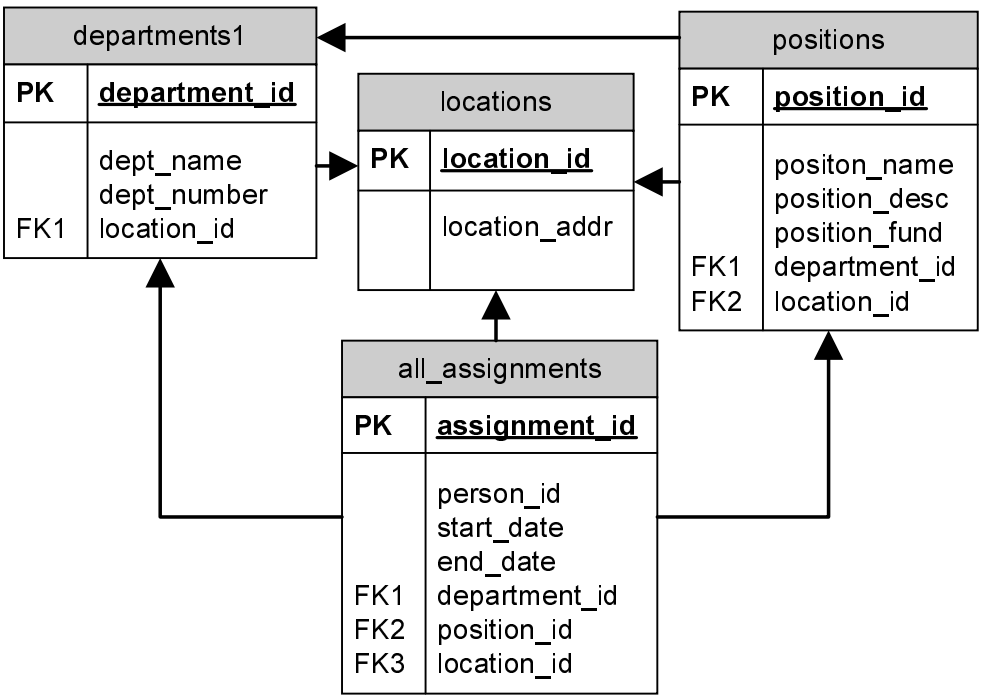


Рис. 5.1. Фрагмент схемы: отделы, назначения и местоположение

Можно заметить, что в этой схеме имеется некоторая избыточность, вызванная наличием функциональных зависимостей. Так, значение атрибута `department_id`, имеющегося в таблице `assignments`, может быть вычислено, исходя из значения атрибута `position_id` и атрибута `department_id` в таблице `positions`. Зависимости такого типа называются транзитивными. Для устранения такой зависимости нужно просто удалить атрибут `department_id`

из таблицы assignments. Точно также избыточным является атрибут location_id в той же таблице assignments. Более того, его значения можно вычислить даже двумя способами: через position_id и через departme_id, при этом результат, вообще говоря, может получиться разный!

Возможно, однако, что эта избыточность является кажущейся и в реальности таких зависимостей не существует, а избыточные атрибуты имеют различающийся смысл. Например, если отдел занимает несколько помещений, то, скорее всего, местоположение отдела показывает, где размещается руководство отдела, а местоположение должности — где находится сотрудник, занимающий эту должность. Поручение, выданное сотруднику, может относиться не к тому отделу, в котором он работает. Поэтому решение вопроса о том, имеется функциональная зависимость или нет, и, соответственно, имеется ли избыточность, зависит от того, как специфицированы и как используются атрибуты.

При обсуждении схемы базы данных персонала мы употребляем такие выражения, как "возможно" и "может быть" не случайно. Дело в том, что в данном случае схема базы данных проектировалась разработчиками пакета, а не разработчиками конкретной прикладной системы, которая этот пакет использует. Разработчики пакета заботились о том, чтобы в рамках этой схемы можно было реализовать базы данных для различных прикладных систем. Можно надеяться, что это было сделано сознательно для того, чтобы обеспечить гибкость, а не вследствие ошибки проектирования.

При этом разработчики прикладной системы, использующие этот пакет, должны тщательно изучить схему базы данных и определить, какие из возможных зависимостей имеют место в их прикладной системе, и в соответствии с этим принять решение о том, какие именно атрибуты будут использоваться и в каком смысле. В реальном случае, к сожалению, у проектировщиков прикладной системы на это не нашлось времени.

Другой пример избыточности вследствие транзитивных зависимостей мы уже упоминали при описании примера базы данных учета командировочных расходов в *разд. 3.6.3*. Напомним, что в этом примере избыточные атрибуты введены в таблицу disputes, содержащую описание спорных расходов. В этом случае избыточность, безусловно, имеет место, и была введена вполне сознательно для того, чтобы ускорить выполнение запросов определенного типа. Мы обсудим другие подобные случаи далее в этой главе.

5.2.3. Вычисляемые атрибуты

В некоторых случаях функциональные зависимости могут быть описаны достаточно простым алгоритмом или формулой. Например, пусть некоторая таблица хранит информацию о геометрических точках, причем точки идентифицируются своими декартовыми координатами. Можно рассматривать в

качестве атрибутов точки значения ее полярных координат. Безусловно, такие атрибуты могут быть определены в логической модели данных. Совсем другой вопрос — нужно ли эти зависимые атрибуты хранить. Ответ на этот вопрос совсем неоднозначен. Нет сомнений в том, что их вычисление "на лету" при выполнении запроса более эффективно, чем выборка хранимых значений. Однако в том случае, если эти атрибуты используются при поиске, возможно, их значения должны быть включены в индекс и, следовательно, материализованы.

Наибольший интерес представляют функциональные зависимости, которые не могут быть выражены с помощью вычислительного алгоритма и поэтому должны быть представлены в табличной форме, как, например, зависимость названия отдела от его номера. Если в таблице имеются нежелательные функциональные зависимости такого типа, то декомпозиция на логическом уровне, безусловно, необходима.

Использование принципов нормализации на практике ограничивается тем, что выявление функциональных зависимостей не всегда является простой задачей. Если схема базы данных генерируется на основе объектной модели приложения, то все функциональные (и другие) зависимости должны быть учтены уже на уровне объектной модели. Если в схеме базы данных удается обнаружить нежелательные (например, транзитивные) зависимости, то, скорее всего, это означает, что какой-то класс объектов реального мира не был представлен в объектной модели приложения (по ошибке или намеренно, с целью упрощения модели).

Учитывая это, можно сказать, что при использовании объектных методологий или высокоуровневых моделей данных (типа "сущность-связь") логическая структура данных получается нормализованной автоматически, если, конечно, в модели правильно выделены классы объектов. При этом функциональные зависимости представляются ограничениями ссылочной целостности (первичный-внешний ключ).

5.3. Хранение взаимосвязанных таблиц

Уточним задачу, которую необходимо решать при проектировании структуры хранения для взаимосвязанных таблиц (например, полученных в результате нормализации на логическом уровне проектирования).

Пусть имеются две таблицы, связанные по значению ключа одной из них (не важно, определена ли эта связь как ограничение "первичный-внешний ключ" в схеме), такие, что их соединение по этому ключу часто используется в запросах. Примерами таких таблиц являются таблицы `t_emp` и `t_dept` из предыдущего раздела.

Нас интересует вопрос, при каких условиях хранение этих таблиц более эффективно, чем хранение их соединения (в примере — таблицы t_empd) и как организовать хранение, чтобы преимущества раздельного хранения таблиц реализовались.

5.3.1. Оценка эффективности метода хранения

Очевидно, единственный выигрыш, который может быть получен при хранении результата соединения, состоит в том, что выборка даных из таблицы t_empd может выполняться более эффективно, чем выполнение выборки из таблиц t_dept и t_emp с последующим соединением. С другой стороны, в тех случаях, когда в запросе нужны данные только из одной из таблиц t_dept или t_emp, следует ожидать, что использование большой таблицы будет менее эффективно.

Для того чтобы сравнить различные варианты организации хранения, мы будем оценивать сложность полного просмотра композиции для этих вариантов хранения. В качестве метрики выберем количество обращений к диску.

Предположим, что рассматриваемые таблицы связаны отношением $1:m$, т. е. каждой строке одной таблицы (t_dept) соответствует несколько строк второй (t_emp). Значения строк первой таблицы будем обозначать

$$D_1,\ D_2,\ D_3,\ ...,$$

а строки второй таблицы, соответствующие строкам D_i, будем обозначать

$$E_{i1},\ E_{i2},\ E_{i3}, E_{i4},\ ...$$

Кроме этого, нам понадобятся:

b — размер блока в базе данных;

l_D и l_E — (средняя) длина строки в таблицах t_dept и t_emp соответственно;

N_D — количество строк в таблице t_dept.

Будем считать, что в среднем каждой строке t_dept соответствует m строк таблицы t_emp, поэтому количество строк в таблице t_emp составляет mN_D.

При хранении таблицы-композиции t_empd данные будут располагаться следующим образом:

$$D_1E_{11}, D_1E_{12}, D_1E_{13}, ..., D_2E_{21}, D_2E_{22}, ...$$

Количество блоков диска, занятых этой таблицей, составит $mN_D(l_D + l_E)/b$.

Для хранения таблиц `t_dept` и `t_emp` понадобится соответственно $N_D l_D / b$ и $mN_D l_E / b$ блоков. Разница составляет

$$(m-1)N_D l_D / b.$$

Для того чтобы раздельное хранение таблиц было более эффективным, чем хранение производной таблицы `t_empd`, нужно, чтобы эта разница была больше, чем сложность выполнения операции соединения.

Поскольку результаты операций обычно передаются в качестве входных данных для следующих операций без промежуточной записи на диск, при оценке сложности операции соединения не нужно учитывать операции, необходимые для записи результата (так же, как эти операции не учитывались при оценке выборки из хранимой таблицы `t_empd`).

Сложность операции соединения зависит, конечно, от используемого алгоритма.

Метод вложенных циклов в общем случае использует многократный просмотр входных данных и поэтому в общем случае не может быть эффективнее однократной выборки данных. То же самое справедливо и для алгоритма соединения на основе сортировки-слияния.

Алгоритм соединения на основе хеширования, наоборот, использует однократный просмотр входных данных и поэтому, скорее всего, будет более эффективным, чем выборка из хранимой таблицы `t_empd` (по количеству обращений к диску).

Можно существенно ускорить выполнение операций соединения, если использовать кластеризацию.

Если СУБД предоставляет возможности для управления кластеризацией хранимых данных, то можно кластеризовать (т. е. упорядочить) обе таблицы `t_dept` и `t_emp` по значению атрибута `dept_id` (который является первичным ключом в таблице `t_dept`, но не в таблице `t_emp`).

В этом случае таблицы оказываются упорядоченными так, как нужно для выполнения соединения методом слияния (исключается сортировка), поэтому сложность операции соединения будет просто совпадать со сложностью выборки данных из двух таблиц, т. е. хранение таблицы `t_empd` будет менее эффективным.

Некоторые СУБД позволяют организовать совместное хранение таблиц в одном кластере (естественно, таблицы при этом упорядочиваются по одному и тому же атрибуту). В этом случае данные могут быть размещены примерно в следующем порядке:

$$D_1\ E_{11}\ E_{12}\ E_{13}\ D_2\ E_{21}\ E_{22}\ D_3\ E_{31}\ E_{32}\ E_{33}\ E_{34}\ldots$$

При таком хранении соединение выполняется еще эффективнее, чем при использовании отдельных кластеров для таблиц `t_dept` и `t_emp`, однако для

того, чтобы это показать, нужны более точные оценки, чем приведенные выше. С другой стороны, легко показать, что выборка только из таблицы t_dept будет в этом случае значительно менее эффективной.

Конечно, далеко не во всех случаях результат соединения требуется полностью и поэтому, вообще говоря, не требуется полный просмотр хранимых таблиц. Если при выборке данных могут использоваться индексы, потребуются другие оценки. Мы не будем приводить эти оценки, однако заметим, что при выборке небольшого числа строк (а именно в таких случаях могут быть полезны индексы) разница в сложности выполнения операций будет очень мала, независимо от выбранного способа хранения данных. Поэтому нет никакого смысла отказываться от нормализации: выигрыш в эффективности получить не удается, но появляются нежелательные свойства схемы базы данных.

Оценки числа обращений к диску в случае, когда количество выбираемых строк очень мало по сравнению с общим числом строк в таблицах, основываются на том факте, что ожидаемое количество обращений к диску равно количеству выбираемых строк. Иными словами, кластеризация не имеет в этом случае большого значения.

Допустим, в результате выполнения запроса из таблицы t_empd выбирается (по индексам) s строк. Тогда ожидаемое количество записей таблицы t_dept, необходимых для выполнения этого запроса, составит s/m, и еще s потребуется для выборки из t_emp. Поэтому выборка из таблицы t_empd будет только в $1+1/m$ раз быстрее. Такое ускорение абсолютно не имеет значения, потому что оно меньше, чем погрешности оценок и разброс при различных значениях параметров запроса.

5.3.2. Объекты большого размера

В некоторых случаях декомпозиция оказывается очень полезной, независимо от того, какие функциональные зависимости известны и в какой мере нормализована схема базы данных.

Рассмотрим таблицу, содержащую относительно небольшое число строк очень большой длины, например, таблицу, хранящую тексты документов или изображения и т. п. (листинг 5.4).

Листинг 5.4

```
create table t_document (
   doc_id,
   author,
   version,
```

```
   title,
   last_updated,
   created,
   doc_body LOB,
   approved_by,
   approval_date)
```

Для нас по-прежнему не имеют большого значения типы данных, кроме атрибута `doc_body`, о котором известно, что его значения велики по размеру.

Поиск в такой таблице чаще всего будет выполняться по значениям небольших атрибутов. (Для поиска по содержанию документа нужны инструменты информационного поиска, но мы не рассматриваем такие средства в данной книге. Можно обойти обсуждение проблемы поиска по содержанию, если заменить документ, например, видеоклипом.)

Вследствие того, что число строк в таблице невелико, довольно часто СУБД будет использовать полный просмотр, а поскольку таблица велика по размеру, то ее обработка будет занимать много времени.

Можно заменить исходную таблицу на две ее проекции (листинг 5.5).

Листинг 5.5

```
create table t_document_header (
   doc_id,
   author,
   version,
   title,
   last_updated,
   created,
   approved_by,
   approval_date)
create table t_document_body (
   doc_id,
   doc_body LOB)
```

После такой замены время выполнения запросов, включающих эту таблицу, сокращается.

В реальном случае, который дал материал для этого примера, аналогичная таблица использовалась в довольно громоздких запросах, поэтому локализация причин низкой производительности оказалась не очень простой, а время ответа после изменения схемы улучшилось в десятки раз.

Конечно, в каждом конкретном случае результат может быть очень разным в зависимости от используемой СУБД и выбранного способа хранения таблицы. Некоторые системы допускают хранение больших объектов во внешних файлах, другие — выделяют их в отдельную область памяти автоматически. В том и другом случае, естественно, декомпозиция таблицы может оказать только отрицательное влияние на производительность.

5.3.3. Необходимость избыточности хранения

Можно было бы закончить этот раздел теоретически обоснованным выводом о том, что отказ от нормализации не дает существенного выигрыша в производительности и что этот выигрыш никак не может компенсировать ухудшение свойств схемы, возникающее вследствие избыточности.

Если вторая часть этого вывода не вызывает никаких сомнений — избыточность и другие нежелательные свойства ненормализованных в достаточной мере схем абсолютно недопустимы на логическом уровне описания данных, — то первая часть просто неверна.

Приведенные оценки учитывают только количество обращений к диску. Эта метрика обычно, но далеко не всегда, является наиболее важной. Во многих случаях нельзя игнорировать и время процессора, затрачиваемое на вычисление соединений.

Далеко не всегда оптимизатор СУБД может найти оптимальный алгоритм для выполнения операции соединения, в особенности в сложных запросах (содержащих большое количество алгебраических операций).

Таблица может участвовать в соединениях с несколькими таблицами по разным ключам, что исключает возможность ее кластеризации для всех запросов одновременно.

Поэтому при построении структуры хранения приходится вводить некоторую избыточность, т. е. создавать дополнительные структуры для хранения данных, которые могут быть вычислены на основе других хранимых данных, т. е. *материализовать* вычисляемые данные.

Подчеркнем, что речь идет о контролируемой избыточности только на уровне хранения. Это значит, что все вопросы, связанные с обновлением этих данных, следует по возможности решать на уровне базы данных или централизованно в достаточно хорошо выделенной и изолированной части приложения.

5.4. Материализация

Все современные большие СУБД предоставляют возможности для создания материализованных представлений (materialized views). Мы обсудим далее, в каких случаях целесообразно их использовать и как выбирать их структуру,

но, прежде всего, подчеркнем, что понятие материализации далеко не исчерпывается только материализованными представлениями.

Материализацией можно назвать создание любых избыточных структур данных, предназначенных для ускорения некоторого класса запросов. Избыточность здесь означает, что данные, хранящиеся в этих дополнительных структурах, могут быть вычислены на основе базовых хранимых данных.

Смысл материализации заключается в том, что эти производные (вычисляемые) данные могут использоваться в запросах, при этом при выполнении запроса исключается время, необходимое для их вычисления. С другой стороны, материализация всегда влечет за собой дополнительную работу при обновлении, поскольку все изменения базовых данных должны быть так или иначе отражены в материализованных производных данных.

В соответствии с приведенным выше определением, одним из видов материализации является создание индексов, хотя обычно индексы не рассматриваются в контексте материализации. Индексы, безусловно, избыточны и предназначены для ускорения обработки запросов. Более того, в некоторых случаях оптимизатор запросов может использовать индекс без обращения к таблице, для которой этот индекс построен. Можно поэтому сказать, что различие между индексами и материализованными представлениями состоит в том, что индексы нельзя явно указывать в операторах запросов языка SQL, а материализованные представления можно.

5.4.1. Избыточные атрибуты

Материализацию можно использовать как технический прием, независимо от того, предоставляет СУБД такие возможности (кроме индексов) или нет. Один из наиболее частых случаев материализации, которая обычно не предоставляется СУБД явно, — включение в таблицу избыточных атрибутов.

По составу атрибутов полученные в результате такой материализации таблицы могут не отличаться от тех, которые получаются в результате денормализации, однако процесс материализации не является обратным к нормализации, он происходит с другими целями и на другом уровне проектирования.

Рассмотрим пример.

В базе данных хранится информация о проектах, связанных с населенными пунктами, расположенными в отдельных регионах. Информация о проекте, его исполнителе, источниках финансирования и т. п. размещена во многих таблицах, однако сейчас нас интересует только связь проекта с населенным пунктом и регионом (в реальной БД имеется два уровня регионов, но здесь это несущественно). Интересующая нас часть схемы данных показан на рис. 5.2.

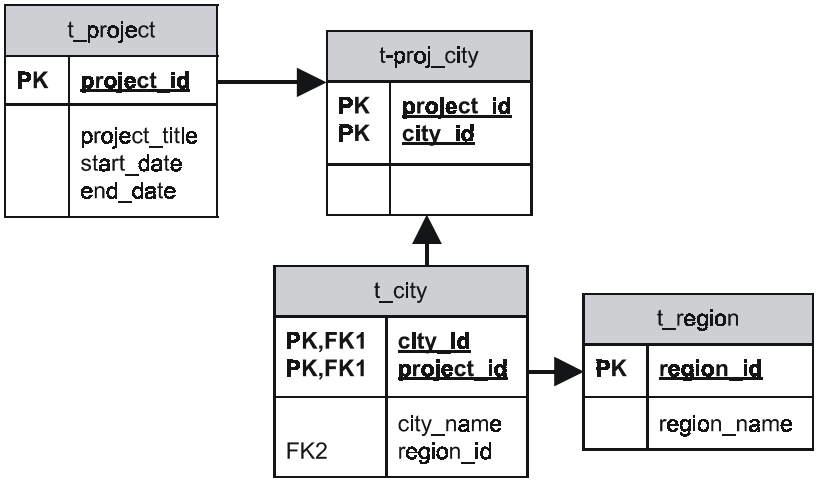


Рис. 5.2. Фрагмент схемы базы данных проектов

Нас интересуют запросы на поиск проектов, связанных с населенным пунктом и регионом, которые выглядят примерно следующим образом (листинг 5.6).

Листинг 5.6

```
select  ... from t_project join ... (еще 6 таблиц)
where project_id in (
   select project_id from t_proj_city where city_id= 125)
select  ... from t_project join ... (еще 6 таблиц)
where project_id in (
   select project_id from t_proj_city where city_id in (
      select city_id from t_city where region_id=87))
```

Для того чтобы не загромождать тексты запросов ненужными деталями, мы исключили все операции соединения со справочниками, которые в этих запросах одинаковы и поэтому не влияют на сравнение характеристик производительности.

Разница во времени выполнения (первый запрос выполняется за 2—5 с в зависимости от числа найденных проектов, а второй требует от 90 с до 120 с) не соответствует разнице в их сложности: во втором запросе добавлена только одна операция соединения с небольшой таблицей к уже имеющимся 7 операциям соединения (считая и полусоединения для вычисления предиката `IN`).

Возможно, оптимизаторы больших СУБД справились бы с этой проблемой, однако в данном проекте использовалась СУБД, которая, скорее, является недорогой, чем высокопроизводительной.

После введения избыточного атрибута `region_id` в таблицу `t_proj_city` и построения индекса

```
create index x_projregion on t_proj_city(region_id, project_id)
```

схема базы данных приобрела вид, показанный на рис. 5.3.

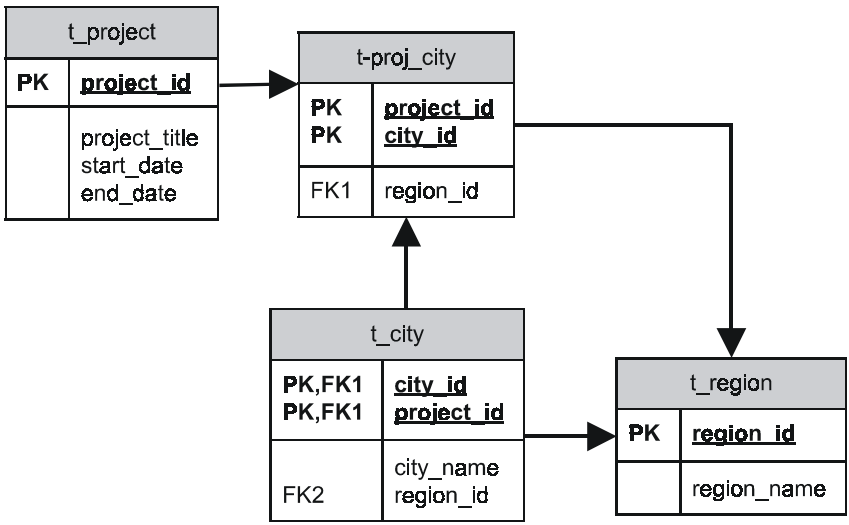


Рис. 5.3. Фрагмент схемы базы данных проектов после введения избыточности

В модифицированной схеме стало возможно упростить второй запрос, исключив одну операцию соединения:

```
select  ... from t_project join ... (еще 6 таблиц)
where project_id in (
   select project_id from t_proj_city where region_id=87))
```

В результате время выполнения запроса по региону практически сравнялось с временем выполнения запроса по населенному пункту.

Причина этого, однако, не в том, что была исключена операция соединения, а в том, что фактически операция полусоединения стала выполняться только по индексу без обращения к таблицам вообще. (Для запроса по населенному пункту это происходило с самого начала.)

Для того чтобы правильно заполнять избыточную колонку, были использованы триггеры, вычисляющие значение `region_id` в таблице `t_proj_city`. Заметим, что расширенная таблица `t_proj_city` представляет собой материализованное

соединение исходного вида этой таблицы с таблицей населенных пунктов, при этом, поскольку состояние исходной таблицы всегда совпадает с состоянием первых двух колонок расширенной таблицы, нет необходимости хранить исходную таблицу, однако триггеры необходимы для обеих соединяемых таблиц.

Таким образом, в этом случае избыточность была введена для того, чтобы можно было построить необходимый индекс.

Еще один пример использования избыточных атрибутов уже упомянут выше в *разд. 5.2.2.*

5.4.2. Когда использовать материализацию?

При решении вопроса о том, следует ли материализовать какие-либо данные и в какой форме, необходимо учитывать следующие факторы:

- сложность выполнения запросов над базовыми (неизбыточными) данными и с использованием материализации;
- относительную частоту выполнения запросов, которые могут выиграть от материализации;
- частоту и методы обновления материализованных данных.

Обсудим эти факторы подробнее.

Сложность выполнения запроса нужно оценивать в сопоставлении с требованиями к производительности системы. То есть сложными запросами в данном контексте являются те, которые объективно не могут быть выполнены за требуемое время. Грубые оценки, приведенные выше, показывают, что, скорее всего, материализация потребуется для улучшения характеристик таких запросов, в которых выполняется большое количество операций над большими объемами данных, т. е. запросов, связанных с получением детальных или суммарных отчетов.

Ситуации, в которых материализация может быть полезна для коротких запросов, крайне редки. Если время выполнения запроса, содержащего операцию соединения, составляет, скажем, десятки миллисекунд, то выборка результата из материализованного соединения, скорее всего, потребует примерно столько же времени. Исключения возможны при очень большом потоке запросов.

Необходимость материализации чаще всего возникает в тех случаях, когда нужно выполнять соединения таблиц большого объема, при этом по каким-либо причинам система не может использовать эффективные алгоритмы для операции соединения (например, необходимые для этого ресурсы памяти недоступны) или выполнение даже оптимальных планов занимает слишком много времени.

Поскольку даже простая выборка данных большого объема из материализованных представлений может занимать много времени, наибольший выигрыш от материализации достигается в тех случаях, когда результат соединения агрегируется (т. е. материализуется не результат соединения, а полученные из него суммарные данные).

Любые изменения базовых данных должны распространяться и на материализации. В системах управления базами данных, поддерживающих материализованные представления, используются два метода их обновления (не все системы поддерживают оба варианта):

- отложенное обновление по расписанию;
- немедленное обновление материализованных данных.

Оба метода можно применять и в том случае, когда СУБД не поддерживает необходимый вид материализации. Например, немедленное обновление можно реализовать с помощью триггеров (как в приведенном выше примере), а отложенное — с помощью программы обновления, запускаемой по расписанию.

Для того чтобы немедленное обновление материализованных данных было возможно, необходимо, чтобы по любой строке базовых данных можно было определить, какие именно строки материализованного представления зависят от этой строки базовых данных, причем количество этих строк должно быть невелико. Только в этом случае немедленное обновление может быть выполнено за приемлемое время. Поэтому системы, поддерживающие немедленное обновление материализованных представлений, накладывают довольно сильные ограничения на вид запросов, по которым можно создавать материализованные представления.

При обновлении материализованных данных по расписанию возможно как полное перевычисление всего материализованного представления, так и локальное обновление тех строк, которые зависят от измененных данных базовых таблиц. Для реализации локальных обновлений по расписанию необходимо, конечно, вести журналы обновлений базовых данных.

Очевидным преимуществом немедленного обновления является актуальность материализованных данных, а преимуществами отложенного — возможность материализации запросов произвольного вида и, как правило, большая эффективность процесса обновления.

Влияние частоты использования достаточно очевидно: чем чаще материализованное представление используется в запросах, тем больше эффект от его создания.

Например, если для обновления материализованного представления используется метод полного перевычисления по расписанию, а результаты нужны для генерации файла, содержащего сведения об операциях с кредитными картами клиентов, передаваемого в банковские сети один раз в сутки, то

создание материализованного представления не имеет никакого смысла. Однако если те же данные нужны для получения часто используемых отчетов, для которых не важна актуальность (в пределах периода обновления), то создание материализованного представления вполне оправданно.

Если, однако, немедленные обновления не создают чрезмерной нагрузки на систему, то использование материализованного представления может быть полезно, даже если оно используется всего один раз в сутки.

Если материализация целесообразна, необходимо определить, какие именно данные следует материализовать. Материализованное представление должно содержать всю необходимую информацию из тех таблиц, на основании которых оно строится, чтобы в запросах не нужно было выполнять повторные соединения с теми же таблицами. При этом не следует включать в материализованное представление абсолютно все таблицы, с которыми можно выполнить соединение, — это приводит к чрезмерному росту размеров материализованного представления, и, следовательно, к увеличению времени его обработки.

Для создания материализованного представления нужно выбрать подзапрос, который содержит наиболее трудоемкую часть вычислений, необходимых для выполнения группы запросов (например, соединения с последующим агрегированием).

Мы рассмотрим примеры запросов, которые, скорее всего, целесообразно материализовать, в *главе 7*, посвященной настройке больших запросов.

5.5. Идентификация

Задача идентификации объектов очень важна для любой информационной системы (неважно, компьютерной или нет). Никакая система не может работать, если в ней нельзя определить, какие из объектов являются разными и, наоборот, находить "тот же самый" объект.

Ошибки идентификации стали основой бесчисленных комедийных ситуаций (например, в кинофильмах с близнецами).

5.5.1. Способы идентификации

Проблема идентификации чрезвычайно трудна и ни один из многочисленных методов не позволяет решить ее полностью. Можно выделить следующие основные классы методов идентификации.

- *Идентификация по свойствам* использует значения некоторых или всех атрибутов идентифицируемого объекта.

  Такая идентификация применяется в классической реляционной модели данных (не всегда точно представленной в реальных СУБД): ключом,

т. е. идентификатором, является любая совокупность атрибутов, от которой функционально зависят все остальные атрибуты отношения.

Другой пример из этого класса — идентификация личности по приметам [20]: "Ростом он мал, грудь широкая, одна рука короче другой, глаза голубые, волоса рыжие, на щеке бородавка, на лбу другая".

Читатель легко найдет более современные методы, применяемые с аналогичными целями.

Идентификация по признакам не может быть абсолютно надежной и однозначной (именно на ошибках идентификации строятся ситуации с близнецами). Возможно, более серьезным дефектом является невозможность отождествления объекта как "того же самого" при изменении значений его атрибутов, выбранных для идентификации.

- *Позиционная идентификация* основана на информации о положении объекта в каком-либо пространстве или по отношению к другим объектам.

  К этому классу можно отнести географические координаты, адреса (в том числе в компьютерной памяти или на диске), а также относительные указания (например, второй "перекресток после компьютерного магазина").

  Менее очевидный пример использования позиционной идентификации — подсчет количества одинаковых покупок при их сканировании в кассе магазина. В этом случае различается положение на входном или на выходном транспортере.

- *Суррогатная идентификация* основана на присвоении объекту искусственных атрибутов, не связанных ни с его свойствами, ни с положением по отношению к другим объектам. Такой идентификатор генерируется при первом появлении объекта в системе и в дальнейшем никогда не изменяется и не используется для идентификации других объектов.

  Идентификация на основе суррогатных атрибутов гарантирует, что по предъявленному идентификатору объект будет опознан как "тот же самый", независимо от изменений его атрибутов или положения.

  К этому классу относятся идентификаторы в объектных системах, номера, выдаваемые налоговыми, пенсионными и другими регистрирующими службами.

  Понятие суррогатного идентификатора имеет смысл только в контексте системы, которая его создала. Если значение суррогатного идентификатора выводится за пределы системы, то оно становится внешним атрибутом объекта и вопрос о том, можно ли его использовать для идентификации в другой системе, требует отдельного рассмотрения.

5.5.2. Какая идентификация лучше?

Поскольку практика использования объектных методологий при проектировании систем общепринята, практически всегда для идентификации объектов в базе данных используются суррогаты. Многие СУБД предоставляют средства для генерации суррогатов, а также могут использовать внутренние суррогаты для идентификации строк в таблицах, независимо от того, определены суррогаты в схеме или нет.

В большинстве случаев использование суррогатов вполне оправданно, потому что упрощает установление и поддержание связей между объектами внутри системы. Наша задача поэтому — показать, какие проблемы может вызывать бездумное применение этого стереотипа, и его влияние на производительность.

Заметим, что хотя использование суррогатов имеет смысл для идентификации сущностей, как правило, суррогаты бессмысленны для таблиц, представляющих связи. Например, в таблице `t_proj_city`, рассмотренной выше, первичным ключом является комбинация атрибутов `project_id` и `city_id`, Введение дополнительного суррогатного ключа в такой таблице привело бы только к ненужному усложнению структуры данных. Кроме этого, возникла бы необходимость отдельно проверять уникальность сочетания `project_id` и `city_id`. Можно сказать, что для таблиц связей комбинации значений суррогатных ключей из других таблиц являются естественными идентификаторами.

Часто бывает необходимо хранить несколько связей между двумя экземплярами сущностей. Например, с заказом на перевозку крупногабаритных грузов могут быть связаны несколько (физических или юридических) лиц: агент по организации перевозки, плательщик, грузоотправитель, грузополучатель и т. д., при этом некоторые из этих ролей могут выполняться одним лицом. В таких случаях в состав ключа связи, кроме идентификаторов связываемых сущностей, необходимо включить атрибут роли или типа связи, но и в этих случаях суррогатные идентификаторы совершенно излишни.

К сожалению, некоторые пакеты предполагают наличие суррогатов в каждой таблице, в таких случаях у проектировщика базы данных нет возможности выбрать наиболее эффективную структуру.

В работе [12] рекомендуется никогда не использовать суррогаты, если имеются естественные уникальные атрибуты. Попробуем аргументировать эту рекомендацию.

Прежде всего, обратим внимание на то, что суррогаты решают проблему идентификации внутри системы, но никак не помогают сопоставить объект реального мира (или объект другой системы) с записью об этом объекте. Можно гарантировать, что наша система обработает тот самый объект, который был выбран пользователем из списка, представленного ему на экране, но

внутренние суррогаты никоим образом не помогают проверить, что физическое лицо, обратившееся к системе, — то же самое, которое обращалось к этой же системе неделю назад.

В примере с удалением дубликатов, который обсуждается в *разд. 4.3.3*, устраняются последствия игнорирования этого факта: дубликаты в этом примере были неразличимы по внешним признакам, однако имели, естественно, разные суррогаты.

Для решения задачи внешней идентификации необходимо либо использовать идентификацию по признакам (в том числе полагаться на суррогаты других систем, таких как удостоверение личности), или выдавать значение суррогата для последующих ссылок. Второй метод используется, например, при бронировании авиабилетов (пассажиру выдается номер бронирования). Заметим, что выдача суррогата не исключает идентификации по признакам, например, авиакомпании обычно идентифицируют пассажира по имени и дате вылета, а номер бронирования используют для дополнительного контроля.

Важное решение, влияющее на характеристики производительности, — выбор типа данных, используемых для значений суррогата. Для применения внутри системы обычно бывает удобнее использовать числовые идентификаторы. Они, вообще говоря, занимают меньше места в памяти, быстрее обрабатываются процессором, проще генерируются. Генераторы суррогатов, встроенные в СУБД, обычно вырабатывают числовые значения (например, `sequence` в Oracle или автоинкрементные типы данных). Однако если суррогат может использоваться вне системы (выдается пользователю, например, как номер бронирования или номер заказа), то его значение должно быть, во-первых, строкой, во-вторых, содержать какую-либо контрольную информацию.

Иногда в подобных случаях используется два суррогата: внутренний числовой и внешний символьный, что, конечно, абсолютно избыточно и, как правило, приводит к ухудшению характеристик производительности. Например, описание таблицы

```
create table customer_order (
   order_id integer PRIMARY KEY,
   confirmation_nmuber varchar(20),
   ...
   )
create unique index conf_num
   on customer_order(confirmation_number)
```

не дает никаких преимуществ по сравнению с описанием

```
create table customer_order (
```

```
confirmation_nmuber varchar(20) PRIMARY KEY,
...
)
```

но, возможно, приведет к необходимости лишних операций соединения при выборке из других таблиц, связанных с таблицей заказов отношением "внешний-первичный ключ".

5.5.3. Неэффективность, вызванная использованием суррогатов

Основное свойство суррогатов, определяющее их полезность, — неизменяемость. Иногда это свойство выполняется и для естественных атрибутов объекта, и тогда использование суррогатов, не давая никаких преимуществ, может привести к серьезному ухудшению производительности, как показано в следующем примере.

В базе данных хранится информация о потреблении энергии некими субъектами. Каждый час для каждого потребителя снимаются показания счетчиков и в базе данных фиксируется объем потребленной энергии в течение прошедшего часа. Конечно, интервал времени, к которому относится запись (т. е. один час), вполне может быть идентифицирован одной меткой времени (например, началом часа), однако этот интервал имеет некоторые дополнительные характеристики, от которых зависит применяемый тариф, а также способ обработки данных при получении отчетов, поэтому для описания свойств интервалов времени используется отдельная таблица. Интересующий нас фрагмент схемы представлен на рис. 5.4.

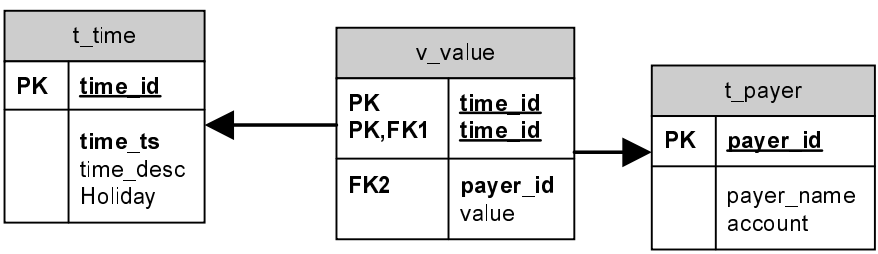


Рис. 5.4. Фрагмент схемы учета энергопотребления с сурогатным ключом

Ошибка проектировщиков схемы проявилась в том, что в качестве первичного ключа этой таблицы был использован суррогатный ключ. На самом деле сама по себе метка времени в данном случае обладает необходимыми свойствами, т. е. ее значение гарантированно уникально и для любого момента времени, естественно, никогда не может измениться. Таким образом, в этом

случае естественный признак объекта обладает и необходимыми свойствами суррогата, поэтому использование суррогата бессмысленно.

Для большинства отчетов требуется выборка объемов потребления за некоторый интервал времени. Поэтому отрицательный эффект состоит в том, что для указания интервала времени (скажем, месяц или год), за который надо получить отчет, неизбежно потребуется операция соединения.

Если в качестве ссылки (внешнего ключа в таблице потребления `t_value`) использовать непосредственно значения меток времени, то выбирать интервал можно по значениям этого ключа, не выполняя для этого операцию соединения. Более того, становится возможной кластеризация таблицы по этому ключу (что ограничивает просмотр таблицы относительно небольшим количеством смежных блоков). Измененный таким образом фрагмент схемы показан на рис. 5.5.

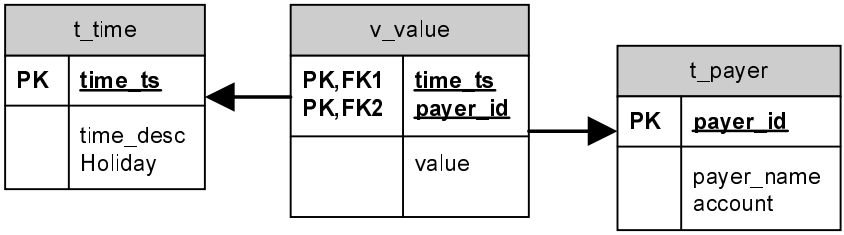


Рис. 5.5. Фрагмент схемы учета энергопотребления с меткой времени

Заметим, что в общем случае значения метки времени не являются уникальными. Разрешение часов в СУБД недостаточно для того, чтобы назначать различные метки при конкурентном выполнении многих операций. Однако в рассматриваемом случае эти метки уникальны, потому что показания счетчиков снимаются один раз в час.

5.6. Агрегаты или слабые сущности?

Слабыми сущностями в модели "сущность-связь" называются такие, которые не могут существовать, если не существует экземпляра сильной сущности, с которым слабая сущность связана.

В терминах SQL описание таблицы, предназначенной для хранения слабых сущностей, может иметь примерно следующий вид (листинг 5.7).

Листинг 5.7

```
create table t_weak_entity (
   weak_entity_id,
```

```
   strong_entity_id,
   ...
primary key(weak_entity_id),
foreign key(strong_entity_id)
   references(t_strong_entity(strong_entity_id))
```

Нас сейчас интересует случай, когда каждому экземпляру сильной сущности может соответствовать ограниченное количество слабых, причем каждая из них имеет дополнительное свойство, которое должно быть представлено только один раз.

Так, в примере с телефонами в базе данных компании НАТКОМС, описанном в *разд. 3.6*, в первоначальном проекте базы данных телефонные номера были представлены как слабые сущности такого типа: каждый номер мог быть номером телефона, факсимильного аппарата или пейджера, причем допускается только один номер каждого типа на компанию.

Альтернатива, которая фактически была использована в этом примере для ускорения обработки, состоит в том, что вместо фиксированного количества слабых сущностей определяются дополнительные атрибуты в сильной.

При описании примера в структуре таблицы присутствовали только телефонные номера, однако строки этой таблицы находятся во взаимно-однозначном соответствии со строками таблицы компаний, и поэтому эти таблицы могут быть просто объединены (если нет особых причин для декомпозиции).

При рассмотрении примера мы показывали преимущества размещения атрибутов слабых сущностей в основной таблице (т. е. все телефонные номера одной компании при этом размещались в одной строке), поэтому теперь мы обратим внимание на преимущества выделения слабых сущностей в отдельную таблицу.

Прежде всего, слабые сущности не обязательно атомарны (т. е. не обязательно имеют только один атрибут). Мы рассматривали телефонный номер как строку, но его можно рассматривать и как структурный тип данных, состоящий из кода страны, префикса (кода региона), номера в регионе и, возможно, добавочного номера (листинг 5.8).

Листинг 5.8

```
   ...
   Phone_country_code varchar(3),
   prefix varchar(5),
   local_number varchar(7),
   extension varchar(4)
   ...
```

(Далеко не все реальные системы, например, доступные в Интернете, способны правильно сохранить любой телефонный номер, но это — совсем другая тема.)

Если по каким-либо причинам необходимо хранить несколько типов телефонных номеров в структурированном виде, описание таблицы становится труднообозримым. Конечно, телефонные номера хранятся в структурированном виде весьма редко, но структурирование адресов бывает необходимо очень часто.

Намного более серьезным недостатком структуры с фиксированным числом колонок является ее недостаточная гибкость. Если по каким-либо причинам требуется ввести дополнительные телефонные номера или адреса, понадобится не только изменение схемы базы данных, но и модификация кода приложения.

Если в проекте с самого начала было принято решение о хранении подобных атрибутов в колонках основной таблицы, то при последующих изменениях, если необходимость в новых атрибутах такого же вида возникает постепенно, и атрибуты добавляются по одному, то, скорее всего, они тоже будут добавляться как колонки в основную таблицу. Сейчас мы опишем пример, в котором этот процесс привел к появлению серьезных проблем с производительностью системы.

В таблице, описывающей заходы судов в порт, кроме других атрибутов, в первоначальной структуре была предусмотрена колонка для хранения даты и времени прибытия судна в порт. В процессе эксплуатации и развития системы появилась потребность хранить также дату извещения о прибытии, затем — дату и время фактического прибытия (в отличие от планируемого) и т. д.

В итоге количество различных колонок с метками времени в этой таблице достигло двенадцати, причем некоторые из них могут отсутствовать и возможно любое сочетание присутствующих меток. Одним из самых частых видов условий выбора из этой таблицы оказались условия на интервал времени по любой из этих временных меток (т. е. в этих запросах требуется выбрать все строки, в которых хотя бы одна из временных меток попадает в заданный интервал).

При таких требованиях представляется вполне целесообразным выделить все метки времени в отдельную таблицу (листинг 5.9).

Листинг 5.9

```
create table t_Pv_date (
   pvdate_id,
   port_visit_id,
```

```
   pvdt_type,
   pv_date_time,
primary key(pvdate_id),
foreign key(port_visit_id)
   references t_port_visit(port_visit_id))
```

Такое изменение схемы могло бы очень существенно упростить написание запросов и ускорить работу системы благодаря использованию индекса.

5.7. Динамические атрибуты

Динамическими атрибутами здесь называются такие атрибуты объектов, которые могут быть добавлены во время выполнения. Это означает, что такие атрибуты не были явно определены в схеме базы данных и разные экземпляры объектов могут иметь разные наборы динамических атрибутов.

Безусловно, динамические атрибуты представляют собой очень мощный инструмент повышения гибкости системы, но эта дополнительная гибкость, как и в других случаях, может оказаться несовместимой с требованиями к характеристикам производительности.

Использование слабых сущностей таким способом, как описано в *разд. 5.6*, по существу, представляет собой способ введения динамических атрибутов, ограниченных, однако, тем, что все их значения должны иметь один и тот же тип данных. Такое ограничение, по-видимому, является разумным компромиссом, так как с одной стороны, обеспечивает некоторую гибкость, и, с другой стороны, позволяет использовать такие механизмы базы данных, как контроль и преобразование типов, ограничения целостности и индексы.

Другим примером компромиссного способа введения динамических свойств хранимых в базе данных объектов могут служить специальные информационные типы (СИТ), рассмотренные в *разд. 3.6*. Введение новых специальных типов неявно предполагает внесение изменений в приложение, однако с точки зрения базы данных и самого пакета, эти типы безусловно являются динамическими. Мы уже обсуждали, каким образом использование СИТ может повлиять на производительность системы.

Хорошо известен и другой, более общий способ представления динамических атрибутов. В наиболее простом варианте описание таблицы для их хранения может выглядеть так, как показано в листинге 5.10.

Листинг 5.10

```
create table Totally_dynamic (
   object_id,
```

```
attribute_name,
attribute_value)
```

В такую таблицу можно записать абсолютно произвольную структуру данных. Эта техника многократно открывалась, наверное, тысячами программистов, начиная, вероятно, с 60-х годов прошлого века.

До тех пор, пока подобные структуры применяются в оперативной памяти, особых проблем не возникает, но использование подобных структур в базах данных возможно только при малых объемах данных. Причины этого вызваны тем, что при такой организации данных нельзя использовать многие механизмы СУБД, а именно:

- обеспечить какую-либо кластеризацию данных;
- индексы не могут учитывать типы данных;
- требуется большое количество операций соединения;
- при большом количестве объектов одного типа очень велики расходы памяти на хранение имен атрибутов.

Рассмотрим пример. При разработке программного обеспечения сервера, рассылающего по заявкам некоторые виды сообщений владельцам мобильных телефонов, должна протоколироваться работа системы в журнале. Для каждой заявки в протоколе регистрируются некоторые события, такие как поступление заявки и различные фазы ее обработки (проверена платежеспособность клиента, сформировано и записано в очередь сообщение, сообщение отправлено, получено подтверждение о доставке и т. п.).

Ведение журнала необходимо как для подготовки платежных документов, так и для анализа с целью выявления закономерностей в использовании системы.

В период, когда система разрабатывалась, реальных потребителей еще не было, поэтому было трудно определить реальные требования к системе. Разработчики обошли эти трудности, сделав систему максимально гибкой, т. е. использовали для журнала структуру данных, очень близкую к полностью динамической. Каждое из перечисленных событий регистрировалось как отдельная запись в журнале, при этом размер имен атрибутов был довольно большим (10—15 байтов), а значение, как правило, занимало 1—2 бита.

Предусмотренная разработчиками возможность добавления новых типов сообщений фактически никогда не была использована в реальных установках системы.

При первой же установке системы для реального применения выяснилось, что за первые несколько дней работы системы размеры журнала превысили суммарный объем всей остальной базы данных (хранившей информацию, на основе которой генерировались рассылаемые сообщения) и затем продолжали

увеличиваться с возрастающей скоростью по мере увеличения нагрузки на систему.

Через два месяца, когда возникли проблемы производительности при обработке журнала, по требованию заказчиков была введена процедура сжатия журнала, которая раз в сутки переписывала накопленные записи в таблицу статически определенной структуры. Эта таблица занимала примерно в 9 раз меньше места и, благодаря использованию индексов, никаких проблем с производительностью не вызывала.

В данном примере проблемы возникли почти исключительно вследствие избыточного хранения имен атрибутов (в журнале они назывались событиями). Другие потенциально опасные факторы не были существенны, потому что фактически хранимая структура данных очень проста.

С другой стороны, при относительно небольших объемах данных использование динамических структур оказывается вполне целесообразным и может приводить к повышению производительности.

В качестве примера удачного применения динамических атрибутов рассмотрим структуры данных для системы проведения опросов через Интернет.

Для каждого обследования разрабатывается список вопросов, на которые должны ответить респонденты. Этот список должен быть представлен в виде формы, в которой респонденты могут, в зависимости от типа поля, выбирать одну или несколько из предложенных альтернатив или вводить текст ответа.

Для описания формы используется таблица, представленная в листинге 5.11.

Листинг 5.11

```
create table question (
   question_id,
   form_id,
   queestion_type,
   question_title,
   ...)
```

Колонка `form_id` идентифицирует форму, т. е. обследование, к которому относится вопрос, а колонка `question_title` содержит текстовое описание поля, используемое при генерации вывода на экран (скорее всего HTML).

Для хранения ответов респондентов в этом случае можно использовать таблицу следующей структуры, показанную в листинге 5.12.

Листинг 5.12

```
create table answer (
   response_id,
   form_id,
   question_id,
   answer_num_value,
   answer_text_value
   ...
   )
```

В этой таблице колонке `response_id` идентифицирует совокупность ответов, содержащуюся в одной заполненной форме, `form_id` и `question_id` описывают соответственно форму и вопрос в этой форме, и, поскольку возможны ответы разных типов, предусмотрены две колонки для хранения ответа, при этом для каждого вопроса заполняется только одна из них.

Отметим, что в этой таблице не используется суррогатный ключ, потому что в нем нет необходимости: строки в ней однозначно идентифицируются тройкой колонок (`response_id`, `form_id`, `question_id`). При этом допускается, что вопросы из разных форм могут иметь совпадающие идентификаторы.

В отличие от полностью динамической структуры, отсутствует избыточность, потому что описания динамических атрибутов хранятся в отдельной таблице (таблице вопросов), а не вместе с данными (т. е. не в таблице ответов).

В этом примере необходимость использования динамического описания структуры форм вполне очевидна с самого начала, поскольку программный код не должен зависеть от конкретного обследования. Использование подобной техники описаний может быть целесообразно и в тех случаях, когда необходимо обрабатывать всего одну форму, и при этом маловероятно, что состав полей этой формы может измениться в будущем. (Например, формы, структура которых утверждается на уровне правительств, обычно очень стабильны.)

В одном из проектов удалось предотвратить создание таблицы, содержащей около 300 колонок (и использовать вместо нее динамическое описание полей формы), хотя эта таблица уже успела появиться в спецификациях. В этом случае динамическое описание позволило очень существенно упростить код приложения.

Вероятно, приведенные аргументы в пользу динамического описания форм излишни, так как такие описания являются почти общепринятой практикой.

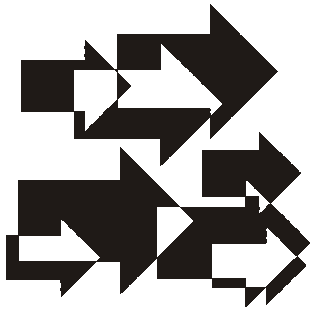
Несмотря на все преимущества динамического описания форм, выбор между динамической структурой и статическим определением должен учитывать и характер использования этих данных, т. е. от того, какие запросы будут выполняться, с какой частотой и каковы требования к производительности.

При обработке результатов опросов практически никогда не требуется выбирать ответы одного респондента на все вопросы. Наиболее типичные запросы в этом случае выбирают ответы на один или небольшое число вопросов всеми респондентами (или группами респондентов) с целью получения статистических характеристик набора ответов (распределений, корреляций и т. д.). Для таких запросов размещение ответов респондента в одной строке таблицы не дало бы практически никаких преимуществ.

Несколько иначе обрабатывались данные в проекте, в котором динамическое описание было применено для единственной формы с большим числом полей. Единственный вид запроса всегда выбирал все значения, введенные одним респонденотом, однако эти данные использовались только для генерации бумажной копии формы, так что динамическое описание упростило программный код и в этом случае без существенных потерь в производительности.

Однако для форм небольшого размера, таких как контактная информация (адреса и телефоны), применение таблиц фиксированной структуры, скорее всего, приведет к заметному выигрышу в производительности в случае наиболее частых запросов (что и произошло в примере компании НАТКОМС *(см. разд. 3.6)*). Компоненты почтового адреса, скорее всего, будут обрабатываться вместе.

Таким образом, применение динамических атрибутов или типов данных (таких как СИТ) всегда требует тщательного анализа возможных последствий. Одни только количественные характеристики для этого недостаточны, важно учитывать, каким образом эти динамические структуры будут использоваться. Самое важное правило состоит в том, что применение динамических атрибутов ни в коем случае не должно подменять анализ реальных потребностей прикладной системы. Распространенный, к сожалению, подход "мы не знаем, что понадобится, поэтому обеспечиваем максимальную гибкость" вряд ли может привести к успеху сколько-нибудь серьезного проекта.



Глава 6

Короткие запросы: использование индексов

Методы выполнения запросов очень существенно зависят от того, какое количество данных обрабатывается при их выполнении. В традиционных областях применения баз данных (например, в банковских системах) наиболее важными являются запросы небольшого размера, выполняемые (в современных системах) за десятки миллисекунд или быстрее. Именно возможность выполнения таких запросов оказалась основным преимуществом систем управления базами данных по сравнению с системами предыдущего поколения, использовавшими последовательную обработку файлов (чаще всего — на магнитных лентах) и поэтому выполнявшими накопленные запросы в пакетном режиме.

Системы, в основном ориентированные на обработку коротких запросов, принято называть системами оперативной обработки транзакций (OLTP, on-line transaction processing). При этом важно то, что разные запросы выполняются абсолютно независимо друг от друга в разных транзакциях и для разных пользователей приложения. Обычно в таких системах сложность обработки данных в приложении невелика, и поэтому основная нагрузка ложится именно на СУБД.

Другой класс запросов составляют отчеты, для получения которых требуется обработка относительно больших объемов данных. Поскольку подобные запросы часто возникают в системах поддержки принятия решений (представляющих результаты анализа базы данных), этот класс запросов принято называть системами аналитической обработки в реальном времени (OLAP, on-line analytical processing). Реальные технические возможности для решения задач такого типа появились только в конце 80-х годов прошлого века, и даже в современных системах их далеко не всегда можно выполнить действительно оперативно.

Обычно современные большие СУБД имеют средства для эффективного выполнения обоих классов запросов, но конфигурация базы данных и системы может зависеть от того, какой класс запросов является наиболее важным для конкретной прикладной системы.

Методы настройки запросов также очень существенно зависят от характера запроса, поэтому мы рассматриваем короткие и длинные запросы отдельно.

6.1. Какие запросы являются короткими?

Мы называем короткими запросами такие, для выполнения которых достаточно обработать только небольшую часть данных из таблиц большого размера, и для которых размер результата также невелик. (Здесь можно считать, что все упомянутые размеры измеряются в строках таблиц.)

Важно уточнить, что речь идет именно о том количестве строк, которое необходимо для получения ответа, т. е. результат выполнения запроса зависит от этих строк. Возможно, неудачный план выполнения такого запроса будет включать полный просмотр большой таблицы — запрос от этого не становится длинным, хотя он, безусловно, нуждается в настройке.

Короткие запросы в традиционных системах оперативной обработки транзакций, как правило, не требуют настройки по времени ответа на уровне отдельных операторов SQL. Это связано с тем, что подобные запросы обычно используют не более нескольких десятков строк из небольшого числа таблиц. Большинство современных СУБД могут успешно решить задачу оптимизации таких запросов при условии, что система в целом правильно настроена для выполнения запросов этого типа.

Вместе с тем, для этих систем часто характерен очень интенсивный поток запросов, поэтому, несмотря на то, что время ответа для каждого отдельного запроса оказывается приемлемым, может потребоваться настройка сервера базы данных в целом для того, чтобы удовлетворить требования к пропускной способности.

Далее в этой главе мы будем рассматривать несколько более сложные запросы, в которых могут обрабатываться сотни или единицы тысяч строк из относительно большого числа таблиц, т. е. включающих, возможно, десятки операций соединения. Тем не менее, в нашем понимании такие запросы тоже являются короткими. Время ответа, измеряемое десятками миллисекунд, в этих случаях практически никогда не достижимо, однако время ответа до нескольких секунд оказывается вполне приемлемым для пользователя (в особенности если ответ содержит сотни или тысячи строк).

Даже в системах с потенциально очень большим числом пользователей (таких как интернет-приложения) выполнение подобных запросов необходимо для реализации функций, используемых ограниченным числом пользователей

системы (администраторами, менеджерами и т. п.), поэтому проблемы с пропускной способностью, как правило, не возникают, однако для таких запросов настройка операторов SQL нужна значительно чаще, чем для совсем простых классических OLTP-запросов.

Более того, суммарная доля подобных запросов в общем балансе ресурсов, используемых сервером базы данных, может быть незначительной, поэтому автоматические средства диагностики производительности вряд ли выделят такие запросы, как существенный ограничивающий фактор. Их эффективное выполнение (по времени ответа) имеет скорее психологическое значение для оценки качества системы.

Какими способами можно добиться высокой эффективности выполнения коротких запросов?

Многие руководства по системам управления базами данных рекомендуют, например, следующее:

- не использовать операцию указания `Distinct`;
- по возможности исключить использование операции `ORDER BY`;
- сократить количество операций соединения за счет введения избыточности;
- располагать колонки в таблицах в порядке убывания частоты использования и возрастания их длин;
- использовать подготовленные (prepared), а не динамические операторы SQL;
- использовать составные индексы.

Многие из рекомендаций такого типа выработаны на основе опыта использования ранних реляционных СУБД, работавших на очень медленных (с современной точки зрения) процессорах с весьма ограниченными объемами памяти. При этом в ранних системах не всегда были реализованы высокоэффективные алгоритмы реляционных операций, а также высококачественные алгоритмы оптимизации.

Так, исключение `DISTINCT` и `ORDER BY`, очевидно, связано с исключением операции сортировки, которая рассматривалась как достаточно сложная.

Для современных вычислительных систем сортировка нескольких сотен или даже тысяч строк обычно не составляет проблемы, потому что такие объемы данных легко помещаются в оперативной памяти.

Влияние избыточности хранения на производительность мы уже обсуждали в *главе 5*. Как правило, операции соединения в коротких запросах, так же как и операции сортировки, не вызывают никаких проблем, конечно, при условии, что почти все операции селекции выполняются раньше, чем операции соединения, и количество строк, участвующих в соединении, действительно невелико.

Использование подготовленных операторов, безусловно, может быть наиболее полезно именно для коротких запросов, которые используются многократно, так как при этом экономится время, необходимое для компиляции и оптимизации запроса. В ранних системах подготовка операторов SQL занимала значительное время и поэтому выполнялась статически, затем откомпилированные запросы включались в исполняемый код приложения.

В современных системах статическая (т. е. выполняемая во время компиляции приложения) компиляция запросов обычно не применяется. Запросы, оформленные как подготовленные, компилируются во время выполнения приложения. С другой стороны, высокопроизводительные СУБД преобразуют динамические запросы в формат подготовленных (т. е. заменяют, например, константы, указанные в запросе, на параметры) и обрабатывают их так же, как подготовленные. Поэтому выигрыш от предварительной подготовки операторов менее существенен.

Необходимо также учитывать, что во время предварительной компиляции не всегда бывают известны параметры запроса, поэтому полученный план может оказаться не самым оптимальным. Конечно, для очень простых запросов, таких как выборка одной строки таблицы по значению первичного ключа, план выбирается единственным способом и никакой дополнительной оптимизации не требуется. Именно такие запросы используются многократно, поэтому их подготовка могла бы иметь смысл, однако обычно значительно лучшие результаты можно получить, если выбирать все необходимые строки одним запросом, а не отдельным запросом для каждой строки.

Поэтому перечисленные выше рекомендации (за исключением последней) могут быть полезны только в тех случаях, когда система работает без большого запаса мощности, что случается крайне редко. В большинстве систем реальная нагрузка не может исчерпать ресурсы процессора (и памяти) даже в пиковые периоды.

Основная задача, которую следует решать при настройке коротких запросов, — это исключение из обработки тех данных, которые не нужны для формирования ответа на запрос. Прежде всего, необходимо исключить все операции полного просмотра больших таблиц. Далее, если запрос содержит несколько условий селекции, нужно позаботиться о том, чтобы условия, исключающие большее количество строк, обрабатывались первыми.

Заметим, что условия селекции в коротких запросах присутствуют обязательно, потому что по определению количество строк, выбираемых из таблиц большого размера, должно быть ограничено, но вполне возможно, что условия наложены на другие таблицы, а размер выборки из больших таблиц ограничивается операциями соединения.

6.2. Короткие запросы и индексы

Как правило, эффективная обработка коротких запросов возможна только с использованием индексов. Можно сказать, что наиболее важным фактором, влияющим на эффективность выполнения коротких запросов, является правильность использования индексов.

Для выполнения коротких запросов чаще всего достаточно использовать наиболее традиционные индексы, построенные на основе B-деревьев или близких по своим функциональным свойствам структур данных. Каждый такой индекс строится для одного атрибута или группы атрибутов и обеспечивает точечный поиск по значениям или просмотр интервала значений.

Можно выделить следующие способы использования индексов:

- отбор строк по условию на значения атрибута с последующей выборкой данных из таблицы (как правило, при этом используется вторичный индекс);
- доступ к строкам таблицы при выполнении операции соединения (как правило, используется индекс по первичному или внешнему ключу);
- выборка всех необходимых данных из индекса без обращения к таблице.

Каждый индекс, построенный для некоторого атрибута, может быть полезен только для выполнения тех запросов, в которых этот атрибут использован в условиях селекции или в условиях соединения, поэтому для того, чтобы эффективно выполнять различные типы запросов, необходимо построить индексы по разным атрибутам.

Задача выбора атрибутов для построения индексов является, пожалуй, наиболее известной из множества задач, решаемых при проектировании физической структуры хранения.

Точное решение этой задачи как задачи оптимизации вряд ли имеет практический смысл вследствие неизбежно существующих неопределенностей (невозможно точно предсказать, с какой частотой будут выполняться одни или другие запросы в будущем). Поэтому мы, следуя обычной практике учебников и руководств, приведем типичные рекомендации по выбору индексов.

- В каждом коротком запросе хотя бы одно условие на значения атрибутов из больших таблиц должно поддерживаться индексом.
- Не имеет смысла строить индексы для атрибутов, имеющих мало различных значений, однако такие атрибуты могут участвовать в составных индексах.
- Не имеет смысла создание индексов для таблиц малого размера, занимающих всего несколько блоков дисковой памяти.

- Создание нового индекса может повлиять не только на план выполнения того запроса, для поддержки которого этот индекс создается, но и других запросов, при этом далеко не всегда новый план будет более эффективным.
- Индексы по часто обновляемым атрибутам могут существенно увеличить нагрузку на систему при выполнении операций обновления.
- Составные индексы (по нескольким атрибутам) могут использоваться не только для доступа к таблицам, но и для выполнения запросов без обращения к таблицам (если все необходимые атрибуты включены в индекс).

В некоторых случаях индексы автоматически строятся системой для поддержки ограничений целостности (например, индексы по первичным ключам или атрибутам с уникальными значениями).

Построение индекса является относительно сложной и дорогостоящей операцией, поэтому структура индексов оказывается статической. Можно доказать, что динамическое построение индексов почти никогда не имеет смысла.

Поскольку индексы являются частью схемы базы данных, может оказаться, что построение необходимых индексов затруднено или невозможно по организационным причинам. Так может случиться, если приложение строится не непосредственно над СУБД, а с использованием прикладного пакета, который навязывает структуру схемы базы данных, при этом изменения схемы запрещены условиями лицензии этого пакета.

Поскольку индексы являются избыточной структурой данных, они неизбежно вызывают замедление операций обновления базы данных. В большинстве случаев операции обновления выполняются значительно реже, чем операции выборки даных, поэтому чаще всего это замедление не оказывает существенного влияния на характеристики производительности системы.

Однако в некоторых случаях при массовом обновлении данных влияние индексов может быть очень значительным. Хорошо известный прием, используемый в этих случаях, состоит в том, что индекс уничтожается перед выполнением массового обновления и строится вновь после него.

В наибольшей степени это замедление влияет на операции обновления значений атрибутов, по которым построены индексы, потому что в этом случае при обновлении одной строки выполняются операции удаления из индекса и вставки в индекс для каждого изменяемого атрибута, которые могут привести к перестройке индексного дерева.

Рассмотрим пример. Для того чтобы даты в демонстрационной базе данных системы учета командировочных расходов, описанной в *разд. 3.6.3*, оставались близкими к текущей, периодически ко всем датам в базе данных при-

бавлялась некоторая константа (скажем, 90 дней) с помощью оператора SQL вида:

```
update t_expense_items
set exp_date=exp_date+100,
    approval_date=approval_date+90
```

При наличии индексов по обновляемым атрибутам данный оператор выполнялся около 40 минут, а без индексов — менее 5 минут, вместе с построением индексов после операции. Большое время обновления вызвано, конечно, тем, что вся система, включая сервер базы данных, работала на портативных компьютерах торговых представителей компании (это все-таки демонстрационная версия!), не обладавших большими ресурсами памяти и процессора.

Конечно, такой значительный эффект наблюдается далеко не всегда. Например, во многих случаях удаление индексов не оказывает существенного влияния на время выполнения массовой загрузки данных, вопреки рекомендациям, которые можно найти в руководствах по конкретным системам. Как правило, различие во времени на 20—30% не является существенным для операций такого типа.

Конечно, ни массовое обновление, ни загрузка данных не являются короткими операциями.

Кроме общепринятых одномерных упорядоченных индексов (т. е. B-деревьев) по атрибутам или группам атрибутов, СУБД могут предоставлять другие типы индексов, например, пространственные индексы, функциональные индексы и индексы на основе битовых шкал. Пространственные индексы полезны для индексирования данных специального типа (геометрических или географических), в остальном их использование не отличается от использования обычных индексов.

Функциональные индексы отличаются способом построения: вместо значений атрибутов используются выражения, которые могут включать определенные пользователем функции. Такие индексы могут быть полезны для выполнения запросов по атрибутам, содержащим значения сложного типа (например, определенного пользователем типа данных), или в тех случаях, когда функциональное выражение (по которому построен индекс) часто встречается в условиях запросов. Область применения и методы настройки этих индексов такие же, как для обычных индексов, поскольку значения функциональных выражений записываются в дерево тем же образом, как и значения атрибутов при построении традиционных индексов.

Индексы, основанные на битовых шкалах, в отличие от традиционных, наиболее эффективны для атрибутов, имеющих небольшое количество различных значений. По этой причине они мало полезны при выполнении коротких запросов и могут использоваться только в комбинации с традиционными

индексами. Основная область применения индексов этого типа — сложные (длинные) запросы, выполняемые в системах аналитической обработки данных.

6.3. Настройка критериев селекции

Как правило, при наличии необходимых индексов оптимизатор запросов строит план, достаточно близкий к оптимальному, поэтому ручная настройка коротких запросов в большинстве случаев не требуется. В данном разделе мы рассмотрим несколько случаев, которые можно считать исключениями из общего правила: по разным причинам изменение исходного текста запроса приводило в этих случаях к существенному улучшению времени ответа.

6.3.1. Использование индексов

Далеко не каждое условие на значения атрибута, допускаемое синтаксисом языка SQL, может быть проверено с помощью индексов. Конечно, любая система управления базами данных использует индексы для селекции по условиям, в которых значение атрибута сравнивается с константой на равенство, неравенство (больше или меньше) или производится поиск по диапазону (`between`), но в более сложных случаях поведение разных СУБД может различаться.

В некоторых случаях условия, для которых оптимизатор не может использовать индекс, могут быть переписаны таким образом, что использование индекса становится возможным.

В других случаях, наоборот, необходимо предотвратить использование индекса при выполнении одного из запросов, но сохранить сам индекс для применения в других запросах.

В этой главе мы не рассматриваем аппарат подсказок, предоставляемый многими СУБД, который дает возможность влиять на поведение оптимизаторов, потому что и синтаксис, и семантика подсказок полностью зависят от конкретной СУБД.

Почти во всех случаях любая трансформация колонки таблицы блокирует использование соответствующего индекса. Так, для условия

```
where coalesce(a.department_id, 1)= 100
```

индекс по `department_id`, скорее всего, не будет использоваться.

Довольно часто бывает необходимо сравнивать текстовые колонки, игнорируя различие между строчными и прописными буквами. Встроенная функция `upper` (или ее эквиваленты) конечно, превращает колонку в выражение, и поэтому, вообще говоря, индекс использоваться не будет. Однако некоторые

СУБД выполняют сравнение без учета регистра и без использования этой функции, при этом индекс будет использоваться, если, конечно, он существует.

Операция сравнения по шаблону (`like`) относительно часто появляется в запросах, формируемых на основе экранных форм, потому что позволяет избавить пользователей от необходимости полностью набирать длинные текстовые значения. Как правило, эта операция не поддерживается индексами, даже если начало шаблона не содержит символов-множителей. Так, выборка по условию

```
where department_name like '033%'
```

скорее всего, не будет использовать индекс, в то время как выборка по любому из условий:

```
where department_name >='033' and department_name<'034'
```

или

```
where department_name between '033' and '034'
```

может быть выполнена сканированием индекса по интервалу.

6.3.2. Избыточные условия выборки

Замена сравнения с шаблоном на ограничение интервала значений, конечно, не всегда корректна, потому что эти условия могут не быть эквивалентными. Можно, однако, указать условие поиска по интервалу в дополнение к условию сопоставления с шаблоном, т. е. ввести избыточное условие для того, чтобы значительная часть строк, не удовлетворяющих условиям запроса, была отсечена с помощью индекса.

Такая "предварительная выборка" бывает полезна для того, чтобы:

- ❒ добиться использования определенных индексов;
- ❒ уменьшить размер аргументов операции соединения.

Рассмотрим более сложный случай использования избыточного критерия на примере базы данных учета командировочных расходов:

```
select <...>
 from reports r,
           expense_items i
where r.rep_id=i.rep_id
  and (r.proc_date='1-jul-2003' and i.expamount<0
              or  r.proc_date='15-jun-2005' and i.expamount=0 );
```

В этом случае сложный критерий поиска невозможно применить, пока не произведена операция соединения, при этом таблица `expense_items` может содержать в десятки раз больше строк, чем таблица `reports`, а индекс по

атрибуту proc_date не работает, что приводит к полному просмотру большой таблицы.

План выполнения этого запроса будет выглядеть приблизительно таким образом:

```
FILTER (proc_date, expamount)(
      NESTED LOOP JOIN (
         TABLE ACCESS(reports, INDEX rep_id)
         TABLE ACCESS(expense_items, INDEX rep_id)
         )
    )
```

Можно добавить избыточное условие, которое включает все строки, удовлетворяющие исходным условиям, и может быть проверено с помощью индексов.

Модифицируем поисковый критерий следующим образом:

```
select <…>
from reports r,
           expense_items i
where r.rep_id=i.rep_id
and (r.proc_date in ('1-jul-2003', '15-jun-2003') )
and (r.proc_date='1-jul-2003' and i.expamount<0
              or  r.proc_date='15-jun-2003' and i.expamount=0 );
```

Такое изменение даст возможность сначала ограничить выборку из таблицы reports только отчетами за первое и пятнадцатое июля, а потом выполнить соединение только этих строк с нужными строками expense_items и после этого окончательно завершить селекцию по основному критерию.

План выполнения данного запроса можно представить следующим образом:

```
FILTER (proc_date, expamount)(
      NESTED LOOP JOIN (
         FILTER (proc_date)
              TABLE ACCESS (reports, INDEX proc_date)
         TABLE ACCESS(expense_items, INDEX rep_id)
         )
    )
```

6.3.3. Составные индексы

Составным индексом называется индекс, построенный по нескольким атрибутам, значения которых упорядочиваются в индексе в лексикографическом порядке.

Составные индексы предоставляют следующие возможности.

- Поддерживать несколько условий селекции одним индексом.

  Полезность этого ограничивается тем, что атрибуты в составном индексе неравноправны. Так, индекс по атрибутам (`X,Y,Z`) может эффективно поддерживать запросы по совокупностям атрибутов (`X,Y,Z`), (`X,Y`) и, конечно, `X`, но не по (`Y,Z`) или `Z`.

- Увеличить количество различных значений в индексе.

  Обычные индексы по колонкам, имеющим мало различных значений (например, пол, статус и т. п.), практически бесполезны. Однако количество сочетаний этих значений со значениями других колонок составного индекса будет достаточно большим. Благодаря этому, индексы могут быть использованы и для поиска по атрибутам имеющим мало различных значений.

- Неявно материализовать часто используемые проекции хранимых таблиц возможно с другим упорядочением.

  В тех случаях, когда все колонки, включенные в оператор выборки, являются частью одного составного индекса, их значения могут быть извлечены из базы данных без обращения к базовой таблице.

Например, при наличии индекса по трем колонкам

```
create index comp_index on
   expense_items (RepID, Expense_type, Expense_Date);
```

оператор

```
select
   Expense_Type, Expense_date
from Expense_Items
where RepID=1234;
```

можно выполнить без обращения к хранимой таблице, используя только индекс.

Упорядоченность составных индексов позволяет очень эффективно выполнять запросы, содержащие агрегирование, зависящее только от колонок, включенных в индекс, а также операции соединения.

6.3.4. Выбор индексов при выполнении запросов

Если в запросе имеются несколько условий селекции, для каждого из которых существует индекс, то, скорее всего, оптимизатор выберет тот индекс, по которому будет отобрано наименьшее количество строк. При выполнении запроса используется этот индекс, а остальные условия проверяются

при выборке записей из таблицы (не все СУБД и не всегда используют этот алгоритм, но мы рассматриваем сейчас именно поведение этого алгоритма).

При выборе индекса учитывается как количество различных значений атрибута (и, возможно, гистограмма, описывающая распределение этих значений), так и тип условия, указанного в запросе.

В некоторых случаях, например, если значения констант, с которыми выполняется сравнение в условиях селекции, неизвестны во время оптимизации, план выбирается на основе усредненных значений и выбор индекса может оказаться далеким от оптимального.

Рассмотрим пример:

```
select *  from
      all_assignments a
where  a.department_id =110
and a.last_update_date >current_date-1;
```

Мы предполагаем, что для таблицы `all_assignments` построены индексы по колонкам `department_id` и `last_update_date`. Тогда оптимизатор, скорее всего, использует индекс по `department_id`, потому что условие, в котором значение атрибута проверяется на равенство, в общем случае удовлетворяется меньшим количеством строк, чем неравенство. Фактически в этом запросе выбираются только строки, измененные в последний день, поэтому на самом деле второе условие предпочтительно, однако значение, с которым выполняется сравнение, неизвестно во время оптимизации, поэтому статистика, даже если она доступна, оказывается бесполезной.

В подобных случаях целесообразно изменить запрос таким образом, чтобы индекс по `department_id` не использовался. Это можно сделать, например, с помощью приемов, рассмотренных выше.

Необходимость в таком управлении порядком применения условий селекции может возникать и в тех случаях, когда условия относятся к разным таблицам. Мы вернемся к этому вопросу в следующем разделе.

6.4. Операции соединения в коротких запросах

Операция соединения является наиболее трудоемкой среди часто встречающихся операций реляционной алгебры, поэтому исключение полных просмотров больших таблиц при выполнении соединений еще важнее, чем их исключение при выполнении операции селекции. В этом разделе рассматриваются особенности алгоритма вложенных циклов и ситуации, в которых может потребоваться настройка.

6.4.1. Алгоритм вложенных циклов и индексы

Выполнение операции соединения таблиц малого размера не составляет никакой проблемы для современных процессоров. Даже наиболее простой алгоритм вложенных циклов будет, скорее всего, выполняться за приемлемое время. Поэтому наибольший интерес представляют операции соединения таблиц, содержащих большое количество строк.

Для нас будет важно различать операнды операций соединения, полученные на предыдущих этапах плана выполнения запроса (промежуточные результаты), и хранимые таблицы.

Можно утверждать, что для коротких запросов все промежуточные результаты должны быть небольшими по числу строк, потому что если такой план построить невозможно, то запрос не может быть отнесен к классу коротких.

Поэтому аргументы операции соединения большого размера являются хранимыми таблицами, и, следовательно, для них могут быть построены индексы.

Можно также утверждать, что в плане выполнения короткого запроса не может быть операций соединения двух больших таблиц, потому что если такие операции невозможно исключить, то запрос не относится к классу коротких (так как все строки больших таблиц оказываются необходимыми для вычисления результата).

Для того чтобы операция соединения в коротких запросах могла выполняться эффективно, необходимо, чтобы существовал индекс по ключу соединения (возможно, составному) в каждом операнде большого размера.

Если необходимый индекс существует, то алгоритм вложенных циклов может просматривать операнд с малым числом строк (не имеет значения, хранимую таблицу или промежуточный результат) и для каждого значения ключа соединения выполнять операцию селекции из большой таблицы с использованием индекса.

Этот вариант алгоритма эффективен только в том случае, если количество строк, извлекаемых из большой таблицы, невелико (иначе полный просмотр был бы более эффективным), т. е. этот вариант алгоритма применим только для коротких запросов.

Так же, как и для операций селекции, условие, по которому выполняется соединение, должно удовлетворять определенным ограничениям для того, чтобы мог использоваться индекс. Например, если в условии соединения вычисляется некоторая функция от колонки таблицы, скорее всего, индекс не будет использован.

6.4.2. Порядок выполнения соединений

При выполнении коротких запросов результат операции соединения может получиться небольшим по одной из следующих причин:

- ограничения соединяемых таблиц (уменьшение размера аргументов операции соединения);
- полусоединение (один аргумент существенно уменьшает размер результата).

Если запрос содержит несколько операций соединения, оптимизатор пытается оценить размер операндов и определяет порядок выполнения операций соединения таким образом, чтобы промежуточные результаты имели малый размер. В большинстве случаев первым должно выполняться соединение с операндом, содержащим наименьшее число строк.

Если оптимизатор генерирует план выполнения запроса, в котором промежуточные результаты получаются большого размера, порядок операций соединения необходимо изменить.

Рассмотрим пример, аналогичный примеру с выбором критериев селекции из *разд. 6.3.4* (листинг 6.1).

Листинг 6.1

```
select
   a.assignment_id, s.start_date, salary
from
      all_assignments a,
       departments d,
       salaries s
where  a.department_id =d.department_id
           and department_name='33'
           and s.assignment_id=a.assignment_id
           and s.last_update_date >current_date-1;
```

Поскольку условие на `department_name` представляет собой условие на равенство по уникальному атрибуту и, следовательно, будет выбрана только одна строка из таблицы `departments`, скорее всего, оптимизатор построит план следующего вида (листинг 6.2).

Листинг 6.2

```
FILTER (last_update_date,
   NESTED LOOP JOIN (
      NESTED LOOP JOIN ( --- index on all_assignments (department_id)
```

```
         TABLE ACCESS (departments, INDEX department_name)
         TABLE ACCESS(all_asignments, INDEX department_id)
         )
      TABLE ACCESS(salaries, INDEX assignement_id)
      )
   )
```

При этом индекс по `last_update_date`, скорее всего, использоваться не будет, потому что значение выражения в правой части условия оптимизатору неизвестно и, следовательно, ожидаемое количество строк, выбираемых по этому критерию, равно половине количества строк в таблице.

Однако реально по одной строке таблицы `departments` выбирается очень большое количество строк из таблицы `all_assignments`, поэтому план вида

```
FILTER (department_name,
   NESTED LOOP JOIN (
      NESTED LOOP JOIN ( --- index on salaries (assignment_id)
         TABLE ACCESS (salaries, INDEX last_update_date)
         TaBLE ACCESS(all_asignments, INDEX assignment_id)
         )
      TABLE ACCESS (departments, INDEX deparement_id)
      )
   )
```

в котором изменен порядок операций соединения, будет значительно более эффективным.

6.5. Преобразования запросов

В некоторых случаях оптимизаторы не могут построить приемлемый план выполнения запроса по его исходной записи, однако вполне справляются с этим после эквивалентного преобразования запросов. Это явление, безусловно, зависит от конкретной СУБД, поскольку разные оптимизаторы работают по-разному для одних и тех же запросов. Поэтому применимость и полезность способов, упоминаемых в этом разделе, необходимо проверять в каждом конкретном случае отдельно.

6.5.1. Сложные условия, содержащие операцию *OR*

Некоторые оптимизаторы включают операцию полного просмотра таблицы в тех случаях, когда в условии селекции записано сложное выражение, содержащее операцию `OR`, даже если каждое из условий, входящих в операнды этой логической операции, можно обработать с использованием индексов.

В таких случаях может быть полезно переписать запрос с использованием явной теоретико-множественной операции UNION вместо логической операции OR.

Например, в базе данных учета командировочных расходов *(см. разд. 3.6.3)* при выполнении запроса

```
select <…> from reports r,
           expense_items i
   where r.rep_id=I.rep_id
         and (r.proc_date<'1-jul-2003' and i.expamount<0
              or  r.rel_date<'15-jun-2003' and i.expamount=0 );
```

выполнялся полный просмотр:

```
FILTER (
      NESTED LOOP JOIN (
         TABLE ACCESS(reports, INDEX rep_id)
         TABLE ACCESS(expense_items, INDEX rep_id)
         )
    )
```

Если преобразовать этот запрос следующим образом:

```
select <…> from reports r,
           expense_items i
    where r.rep_id=I.rep_id
          and r.proc_date<'1-jul-2003' and i.expamount<0
union all
select <…>from reports r,
               expense_items I
   where r.rep_id=I.rep_id
         and r.rel_date<'15-jun-2003' and i.expamount=0;
```

то каждый из операндов операции объединения вычисляется по индексам.

План выполнения последнего запроса может быть представлен следующим образом:

```
UNION ALL (
        NESTED LOOP JOIN ( --- index on expense_items (rep_id)
             TABLE ACCESS INDEX(reports, INDEX proc_date)
             TABLE ACCESS INDEX(expense_items, INDEX expamount)
         )
      NESTED LOOP JOIN (
             TABLE ACCESS INDEX(reports, INDEX rel_date)
```

```
            TABLE ACCESS INDEX(expense_items, INDEX expamount)
        )
    )
```

Примеры преобразования других запросов, содержащих теоретико-множественные операции, рассмотрены в *главе 7*.

6.5.2. Преобразование условий выборки

Если в запросе используются несколько таблиц, некоторые из условий могут быть наложены на атрибуты разных таблиц, при этом запросы остаются эквивалентными по получаемому результату, однако время ответа может отличаться очень существенно.

Наиболее часто такое переписывание запроса оказывается полезным в тех случаях, когда дополнительные ограничения накладываются на значения атрибута, по которому производится соединение. Например, запрос

```
select *
from all_personnel p
   join all_assignments a
   on p.person_id=a.person_id
where a.person_id = 12345
```

эквивалентен запросу

```
select *
from all_personnel p
   join all_assignments a
   on p.person_id=a.person_id
where p.person_id = 12345
```

Отличие этих запросов только в том, что условие на значение `person_id` проверяется по другой таблице, в которой этот атрибут является первичным ключом. Кроме этого, вторая таблица значительно меньше по размеру. По названным причинам, скорее всего, второй запрос будет выполняться значительно быстрее.

Аналогичная ситуация возникает, когда требуется выполнить соединение более чем двух таблиц по одному и тому же набору атрибутов. Например, при соединении трех таблиц по одному атрибуту всегда достаточно указать только два из трех возможных условий равенства, при этом время выполнения запроса может зависеть от того, какие именно условия указаны, а также важно учитывать, какие условия селекции заданы для каждой из трех таблиц.

Иногда подобные преобразования запросов становятся возможными вследствие избыточности схемы базы данных. Проиллюстрируем это на примере, описанном в *разд. 3.6.3*. На рис. 6.1 показан фрагмент схемы базы данных учета командировочных расходов, описанной в *разд. 3.6.3*.

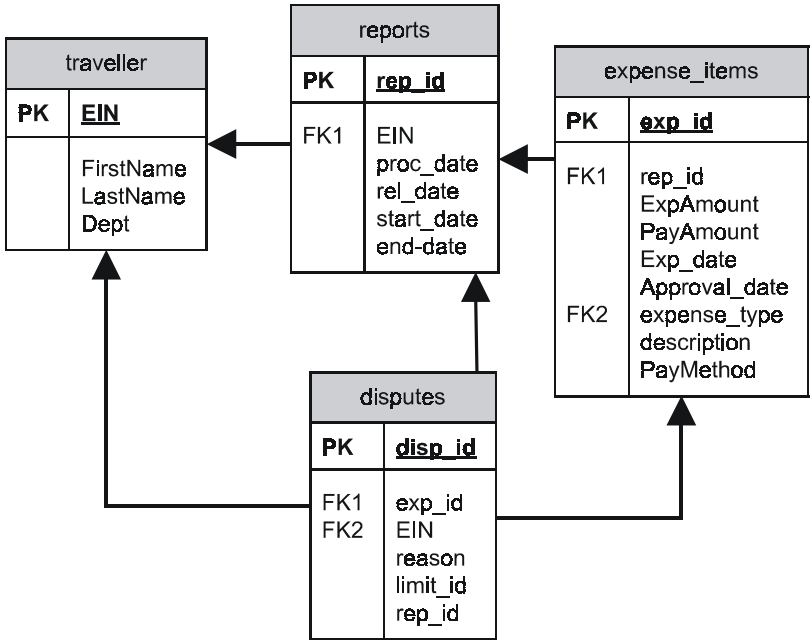


Рис. 6.1. Учет командировочных расходов

Напомним, что таблица `disputes` содержит информацию о спорных расходах, которые не могут быть оплачены автоматически и поэтому нуждаются в подтверждении их целесообразности администратором или менеджером.

Для того чтобы вывести сведения о спорных расходах на терминал администратора, выполнялся следующий запрос:

```
select
      d.ExpType,
      e.ExpDate,
      e.expamount,
      d.Maxi,
      e.PayAmount,
      d.Reason,
      e.RepNum,
      d.Disp_ID,
      e.exp_ID,
      d.Rep_ID,
      e.receiptid
from
   disputes d
```

```
join Expense_Items e
   on d.exp_id = e.exp_id
where
   e.Rep_ID = параметр
```

Как заголовки отчетов, так и записи о единицах расходов никогда не удалялись из системы, поэтому размеры этих таблиц постоянно увеличивались. Это стало причиной того, что время ответа для этого запроса постепенно росло. Когда время ответа достигло единиц секунд, оказалось, что оно неприемлемо, потому что, хотя эти запросы составляют небольшую долю общей нагрузки на систему, они являются наиболее часто выполняемыми администраторами.

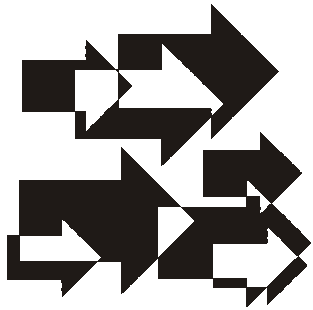
Записи о спорных единицах удалялись из системы после утверждения отчета, поэтому размеры таблицы `disputes` оставались ограниченными. Кроме того, как правило, путешественники не нарушали правила, следовательно, общее количество спорных единиц редко превышало несколько десятков, т. е. таблица `disputes` всегда оставалась очень маленькой.

После переписывания запроса в виде

```
select
      d.ExpType,
      d.ExpDate,
      d.expamount,
      d.Maxi,
      e.PayAmount,
      d.Reason,
      d.RepNum,
      d.Disp_ID,
      d.exp_ID,
      d.Rep_ID,
      e.receiptid
from
   disputes d
join Expense_Items e
   on d.exp_id = e.exp_id
where d.Rep_ID = параметр
```

время ответа сократилось СУБД до 20—30 миллисекунд, что было вполне достаточно для получения удовлетворительного времени ответа приложения.

Отметим, что таблица `disputes` настолько мала, что создание на ней каких-либо индексов с целью увеличения производительности нецелесообразно.



Глава 7

Искусство полного просмотра

Перейдем к рассмотрению приемов, которые могут быть полезны при настройке больших запросов. Напомним, что мы относим запрос к категории длинных (или больших) если для получения результата необходимо обработать все или значительную часть строк из одной или нескольких таблиц большого размера.

Конечно, мы не можем точно определить, какие именно запросы и таблицы являются большими, а какие — не являются. Однако обычно в базе данных можно выделить относительно небольшую группу таблиц, размеры которых значительно превышают размеры всех остальных.

Абсолютные размеры при этом не имеют решающего значения. Даже не очень высокопроизводительные СУБД в состоянии выполнить довольно сложные запросы, включающие более десятка таблиц, из которых три-четыре имеют большой размер, и выработать результат, содержащий десятки тысяч строк, за единицы секунд. Такое время ответа является приемлемым и для некоторых видов коротких запросов.

Возвращаясь к *главе 3*, напомним, что многие проблемы больших запросов потенциально могут быть разрешены на более ранних стадиях разработки системы. Однако обычно и разработчики, и руководители проектов недооценивают значение настройки до тех пор, пока проблемы не возникают реально.

7.1. Когда полный просмотр необходим

Можно выделить две основные причины, по которым полный просмотр базовых (хранимых) таблиц оказывается неизбежным или целесообразным:

- низкая селективность имеющихся индексов по отношению к запросу;
- необходимость использования большей части данных для получения окончательного результата запроса.

Низкая селективность индексов (т. е. ситуация, когда процент строк, отобранных с помощью индекса, от общего числа строк в таблице, достаточно велик) может возникнуть из-за отсутствия индексов, нужных для данного запроса.

В таких случаях создание дополнительных индексов, скорее всего, превратило бы рассматриваемый запрос в короткий. Но далеко не всегда создание дополнительных индексов возможно и целесообразно, например, по следующим причинам:

- создание индекса приводит к ухудшению производительности на других запросах или при обновлении;
- изменения схемы запрещены по условиям лицензии на пакет, через который производится работа с базой данных.

Если же для получения результата нужны (почти) все строки хранимых таблиц, то необходимость и неизбежность полного просмотра сомнений не вызывает, хотя в некоторых случаях можно ограничиться полным просмотром индексов. Напомним, что доступ к таблицам через индексы имеет смысл только в тех случаях, когда количество выбираемых строк мало, поэтому "почти все" в этом контексте может означать "более 5—10%".

Обычно большие запросы характеризуются тем, что время их выполнения на сервере базы данных становится доминирующим фактором, определяющим время ответа прикладной системы. Поэтому наиболее важным критерием производительности для длинных запросов является время ответа. Пропускная способность системы имеет меньшее значение, потому что обычно в системе одновременно выполняется небольшое количество длинных запросов.

7.2. Особенности настройки больших запросов

Рассмотрим приемы и действия, которые можно использовать при настройке больших запросов.

Прежде всего, необходимо убедиться в том, что полный просмотр действительно неизбежен и перевести запрос в категорию коротких невозможно.

Далее следует удостовериться, что поставленные цели настройки достижимы и оценить снизу время, необходимое для выполнения запроса. (Например, время выполнения запроса не может быть меньше времени считывания всех необходимых данных.) Даже такие грубые оценки дают возможность предсказать, какие результаты могут быть достигнуты.

На практике полный просмотр опасен не сам по себе, а потому, что его результат используется в операциях, сложность которых нелинейно зависит от объема исходных данных.

Иными словами, допуская однократный просмотр данных, следует по возможности исключить его многократное повторение, в том числе многократный просмотр промежуточных результатов.

В частности, необходимо проверить, что алгоритмы для операций соединения выбраны (оптимизатором) правильно. Для больших запросов, как правило (но, конечно, не всегда!), алгоритм соединения на основе хеширования превосходит остальные.

Если в результате операций соединения происходит отсечение части данных, то оказывается очень важной последовательность, в которой выполняются операции соединения, и оптимизатор не всегда в состоянии правильно ее построить.

Операции группировки, как правило, уменьшают объем данных, поэтому целесообразно выполнять их как можно раньше. Не все преобразования такого типа (для запросов, включающих группировку) могут быть выполнены автоматически, поэтому может потребоваться переработка текста запроса.

В некоторых случаях замена операций полусоединения на теоретико-множественные дает оптимизатору возможность использовать более эффективные алгоритмы.

Далее эти приемы иллюстрируются на несложных (по возможности) примерах.

Результативность любого из этих преобразований запросов зависит как от свойств оптимизатора, так и от характера данных. По-видимому, любой из примеров в разных случаях может работать в любом направлении (т. е. как улучшать, так и ухудшать производительность), хотя в нашей практике обычно результаты совпадали с ожидаемыми.

Все примеры, приведенные в данной главе, подготовлены на основе реальных запросов, которые приходилось настраивать авторам. Однако во всех случаях реальные запросы были существенно упрощены для того, чтобы выделить главную идею, которая привела к успеху в каждом конкретном случае. При упрощении исключались менее значимые для иллюстрации обсуждаемых методов таблицы (и соответствующие операции соединения), сокращались списки атрибутов, исключались условия, не влияющие на структуру получаемого плана. Как поведение оптимизатора, так и время выполнения упрощенных запросов могут значительно отличаться от тех, которые получались на реальных планах, поэтому мы не приводим никакие количественные характеристики, связанные с этими запросами, кроме оценок размеров исходных данных, когда это необходимо для объяснения применяемых методов.

Во многих случаях, рассматриваемых в этой главе, поведение оптимизатора существенно зависит от конкретной СУБД и от настроек сервера базы данных, управляющих работой оптимизатора. Как правило, для управления поведением оптимизатора приходится использовать средства, зависящие от СУБД, например, подсказки оптимизатору или изменение настроек сервера. Мы, однако, будем обсуждать, какие именно планы выполнения запросов желательно получить, а не способы заставить оптимизатор выработать именно такие планы.

7.2.1. Выбор алгоритмов соединения

Мы уже отмечали, что операция соединения является наиболее трудоемкой из часто встречающихся на практике. Это верно для всех типов запросов, как коротких, так и длинных, однако для длинных выбор алгоритма оказывается особенно важным.

Как ни странно, поведение оптимизатора может определяться не только ожидаемыми значениями функции стоимости, но и модой на те или иные алгоритмы. Так, несколько лет назад оптимизаторы использовали алгоритм `hash-join` чрезмерно осторожно, затем наступил период его повсеместного использования, сменившийся в последнее время возвратом внимания к алгоритму вложенных циклов.

Напомним, что алгоритм соединения на основе хеширования наиболее эффективен для операндов большого объема, однако требует при этом значительное количество ресурсов оперативной памяти и дискового пространства для хранения промежуточных результатов. В то же время этот алгоритм обычно уступает другим в случае операндов относительно небольшого объема.

Оптимизатор может неправильно выбрать алгоритм по одной из следующих причин.

- Размеры выделенной (или доступной) для выполнения запроса оперативной памяти недостаточны для выполнения некоторых алгоритмов.
- Параметры сервера указывают на предпочтительное использование определенного алгоритма (возможно, неоптимального для данного запроса).

  В известном нам случае это произошло в результате использования файла параметров инициализации, заимствованного из предыдущей версии системы.
- Время, выделяемое системой на работу оптимизатора, недостаточно для выбора оптимального плана.

При рассмотрении примеров мы будем указывать, какие именно алгоритмы операций соединения оказывались наилучшими.

7.2.2. Большие исходные таблицы, но небольшой результат

Для больших запросов, так же, как и для коротких, одной из наиболее важных задач является выбор порядка выполнения соединений. Однако причины для этого несколько отличаются. Если в коротких запросах порядок выполнения соединений диктует использование определенных индексов, то в длинных запросах важно добиться того, чтобы наиболее сильное соединение или полусоединение выполнялось первым.

В данном разделе мы рассмотрим запросы, для выполнения которых требуется просмотр больших таблиц, но в результате должно быть выбрано небольшое число строк.

Запросы такого типа очень близки к коротким, в частности, рассмотренным в *разд. 6.4.2*. Отличие состоит в том, что в рассматриваемом здесь случае не накладывается условие на выборку из больших таблиц, поэтому их полный просмотр оказывается необходимым.

Фактически в этом случае одна (или несколько) операций соединения выполняют ограничивающую роль (которую в коротких запросах всегда выполняет операция селекции).

Для настройки выполнения таких запросов очень важно определить, в каком порядке должно выполняться соединение. Цель состоит в том, чтобы как можно раньше исключить записи, которые заведомо не будут нужны, и, следовательно, сократить объем данных, участвующих в последующих операциях соединения.

Во многих случаях оптимизатор может правильно определить нужный порядок соединения, но иногда требуется явное вмешательство разработчика.

Рассмотрим пример.

Требуется выбрать информацию о сотрудниках отдела 33 и всех его подотделов. Иерархия отделов описывается (кроме других способов) атрибутом названия отдела, начало которого совпадает с названием более крупного отдела, в который он входит. Например, отделы 331, 3354 и 33125 являются подразделениями отдела 33 (в реальных условиях, скорее всего, разные уровни структуры называются по-разному, но в этой базе данных подразделения любого уровня описывались одинаково). Таким образом, для того, чтобы выбрать все подотделы отдела 33, можно использовать условие на префикс номера отдела, выраженный предикатом `LIKE`. В реальном прототипе этого запроса вместо `LIKE '33%'` был применен предикат

```
BETWEEN '33' AND '34'
```

потому что СУБД не использовала индекс по атрибуту номера отдела для вычисления предиката `LIKE`. Интересующий нас запрос имеет следующий вид (листинг 7.1).

Листинг 7.1

```
select  last_name, first_name, salary
   from all_personnel p,
           all_assignments a,
           salaries s,
           departments d
where p.person_id=a.person_id
           and a.assignment_id=s.assignment_id
           and d.department_id=a.department_id
           and d.department_name like '33_'
           and salary >2000;
```

Примерные размеры данных следующие: в таблице `departments` 2700 строк, `all_personnel` — 280 000 строк, в таблице `all_assignments` — 800 000 строк. В этом случае ограничивающей таблицей является `department`, поэтому операции соединения должны начинаться с нее.

Этот запрос не может быть отнесен к категории коротких, потому что в нем отсутствуют явные ограничения на большую таблицу. Заметим, что выбор оптимального алгоритма для операции соединения может зависеть, тем не менее, от наличия индексов, абсолютных размеров таблиц и селективности ограничения на таблицу отделов. Скорее всего, если количество отделов невелико и, следовательно, количество сотрудников также относительно невелико, то оптимальным окажется алгоритм вложенных циклов с использованием индексов (листинг 7.2).

Листинг 7.2

```
NEDSTED LOOP JOIN (
  TABLE ACCESS(all_personnel,INDEX (person_id)
  FILTER (salary)(
     NESTED LOOP JOIN (
        NESTED LOOP JOIN (
           TABLE ACCESS (departments, INDEX department_name)
           TaBLE ACCESS(all_asignments, INDEX department_id)
           )
        TABLE ACCESS (salaries, InDEX assignment_id)
        )
     )
  )
)
```

В случае плохой селективности или отсутствия таких индексов — алгоритм соединения на основе хеширования (листинг 7.3).

Листинг 7.3

```
HASH JOIN (
   FILTER(salary)(
     HASH JOIN (
       HASH JOIN (
         FILTER(department_name)
            TABLE SCAN(departments)
         TABLE SCAN (all_assignments)
        )
      TABLE SCAN(salaries)
      )
    )
   TABLE SCAN(all_personell)
   )
```

Для нас, однако, в этом примере важен порядок выполнения операций соединения, а не выбор алгоритма.

Другие важные примеры запросов подобного типа можно получить в тех случаях, когда ограничивающий список категорий, к которым должны принадлежать объекты, представленные в большой таблице, передается в качестве параметра (например, как список категорий, отмеченных пользователем в предложенном перечне).

7.2.3. Операции над множествами

Для больших запросов использование теоретико-множественных операций может помочь оптимизатору выбрать более эффективные алгоритмы. В практике авторов такая необходимость чаще всего возникала в тех случаях, когда оптимизатор использовал алгоритм вложенных циклов для выполнения подзапросов в предложении `WHERE`, т. е. (как правило) операций полусоединения и антисоединения.

Преобразование, которое в этих случаях необходимо выполнить, является вполне эквивалентным и нет никаких теоретически обоснованных причин, мешающих оптимизатору сделать такое же преобразование. Более того, некоторые оптимизаторы с этой задачей успешно справляются. Следовательно, мы можем констатировать, что значение приемов, рассматриваемых в этом разделе, зависит от конкретной системы и от параметров ее настройки.

Преобразования такого типа, конечно, не зависят от того, является запрос длинным или коротким. Заметим, однако, что оптимальный выбор алгоритмов для выполнения этих операций в случае коротких и длинных запросов будет, вообще говоря, разным. В некоторых случаях подобное преобразование фактически превращает длинный запрос в короткий.

Переписывание запроса в другой форме может привести к тому, что оптимизатор выбирает другие алгоритмы для выполнения отдельных операций.

В частности, в некоторых системах, которые явно предоставляют теоретико-множественную операцию `MINUS` в своем диалекте SQL, часто оказывается полезной замена условия `NOT IN (SELECT ...)`, т. е. операции антисоединения, на теоретико-множественную операцию `MINUS`. Например, для запроса

```
select SSN from all_personnel
where ...
   and SSN not in (select SSN from staging_table)
```

оптимизатор построил следующий план:

```
 FILTER (
   TABLE SCAN (all_personnel)
   TABLE SCAN (staging_table)
)
```

В этом плане выполняется полный просмотр больших таблиц. Оказалось целесообразным переписать запрос в следующем виде:

```
select SSN from all_personnel
where ...
minus
select SSN from staging_table
```

Для этого варианта запроса план выглядел следующим образом:

```
MINUS (
   SORT UNIQUE
      INDEX FULL SCAN (idx_all_personell_ssn  --- index on ssn
   SORT UNIQUE
      TABLE SCAN (staging_table)
   )
```

В рассмотренном примере переписывание запроса привело к тому, что второй операнд операции `MINUS` не вычисляется полностью, а вместо этого используется индекс для того, чтобы исключить строки из первого операнда. Нет никаких причин, которые не позволяли бы применить тот же алгоритм для выполнения запроса в исходной форме, но оптимизатор не смог в этом случае построить правильный план.

7.2.4. Избыточные критерии селекции

При выполнении длинных запросов, так же, как и при выполнении коротких, может оказаться полезным введение избыточных поисковых критериев. Следует, однако, обратить внимание, что в больших запросах избыточный критерий применяют для достижения других целей. Если в коротких запросах главной целью введения избыточных условий является включение в план операций выборки по индексам, то для длинных запросов это целесообразно делать с целью сокращения объема промежуточных результатов полного просмотра.

Условие является избыточным, если результат выполнения запроса не зависит от того, имеется это условие в тексте запроса или нет. Иначе говоря, условие избыточно, если данные, удовлетворяющие другим условиям, имеющимся в запросе, автоматически удовлетворяют и этому условию, т. е. избыточное условие является следствием других условий, имеющихся в запросе.

Избыточные условия полезно вводить в тех случаях, когда неизбыточные условия не могут быть проверены на ранних этапах выполнения запроса. Как правило, подобная ситуация возникает, когда условие, применяемое к некоторой таблице, можно перенести, используя предикат соединения, на другую таблицу.

Мы проиллюстрируем применение избыточных критериев ниже в сочетании с другими приемами преобразования запросов.

7.3. Агрегирование

Операции агрегирования или группировки встречаются в больших запросах очень часто, потому что с одной стороны, результат большого запроса зависит от большого количества данных. С другой, в конечном итоге пользователя интересует результат такого размера, который он в состоянии просмотреть. Иными словами, реальный большой запрос, вырабатывающий результаты для пользователя (а не для передачи в другую базу данных или программу), скорее всего, содержит фразу `GROUP BY` и функции агрегирования, такие как `count`, `sum` и `avg`.

Трудности выполнения таких запросов вызваны тем, что операции агрегирования выходят за рамки реляционной алгебры и поэтому преобразования таких запросов не поддерживаются достаточно развитой теорией. Практическим следствием этого факта является то, что оптимизаторы далеко не всегда могут выполнить эквивалентные преобразования запросов, содержащих группировки.

Далее мы проиллюстрируем приемы преобразования запросов, содержащих операции агрегирования, на примерах.

7.3.1. Раннее агрегирование

Обычно запрос, содержащий агрегирование, записывается в наиболее простой форме без использования группировки во вложенных подзапросах. Оптимизатор в этом случае выполняет операцию группировки последней. В результате все промежуточные операции выполняются над тем количеством данных, которое извлечено из хранимых таблиц. Для больших запросов это, как правило, означает, что обрабатываются большие объемы данных.

В то же время операция группировки, вообще говоря, сокращает количество строк, поэтому ее выполнение на более ранней фазе может привести к существенному сокращению общего времени выполнения запроса.

Конечно, такие преобразования не всегда возможны. Для того чтобы можно было выполнить группировку раньше, чем операцию соединения, необходимо, чтобы группируемые значения не зависели от таблицы, с которой выполняется соединение. Иногда это ограничение можно обойти, выполняя агрегирование в несколько этапов.

Рассмотрим пример. В Oracle Applications для обеспечения наибольшей гибкости при подсчете балансов предусмотрена следующая схема: балансы не хранятся в базе данных, а вычисляются каждый раз при выполнении запроса, основываясь на списке составляющих каждого конкретного баланса и значениях составляющих (`run results`), хранящихся в базе данных. Упрощенная схема данных представлена на рис. 7.1.

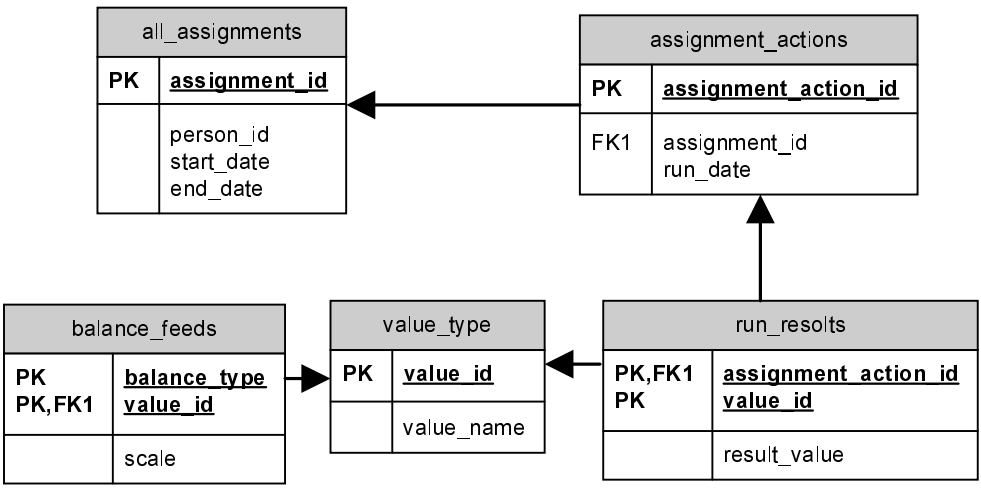


Рис. 7.1. Фрагмент схемы для вычисления балансов

Заметим, что таблица `value_type` реализует связь типа "многие-ко-многим". В рассматриваемых далее запросах она не используется.

Вычисления, связанные с балансами, вызывали проблемы производительности, потому что таблица `run_results` огромна (содержит в десятки раз больше строк, чем все остальные). В то же время как количество балансов, так и типов значений (идентифицируемых с помощью `value_id`) невелико (несколько десятков).

Во всех случаях использования таблицы `run_results` выполняется группировка, поэтому объем результата всегда оказывается значительно меньшим, однако в плане, получаемом для этих запросов, операция группировки выполнялась последней, так что промежуточные операции соединения оперировали с таблицами, число строк в которых оценивается произведением числа строк в `run_results` и количества типов значений.

Например, следующий запрос (листинг 7.4) возвращает список сотрудников и начисленных выплат в определенный день платежа.

Листинг 7.4

```
select
   last_name,
   first_name,
   sum( scale* result_value) gross
from
          pay_all_personnel p
    join  all_assignments a  on a.person_id=p.person_id
    join assignment_actions ac
       on a.assignment_id=ac.assighment_id
    join run_results rr
       on ac.assignment_action_id=rr.assignment_action_id
    join balance_feeds b on b.value_id=rr.value_id
where
    ac.run_date='1-jun-2005'
   and b.balance_type='Gross'
group by
   last_name,
   first_name
```

План выполнения этого запроса представлен в листинге 7.5.

Листинг 7.5

```
GROUP BY(
   JOIN(
```

```
      TABLE SCAN(all_personnel)
      JOIN(
         TABLE SCAN(all_assignments)
         JOIN(
            FILTER(
               TABLE SCAN(assignment_actions))
            JOIN(
               TABLE SCAN(run_results)
               FILTER(
                  TABLE SCAN(balance_feeds)
                  )
               )
            )
         )
      )
   )
)
```

Мы намеренно не выписали конкретный алгоритм для операций соединения, потому что ни выбор алгоритма, ни выбор порядка, в котором выполняется операция соединения, не может исключить вычисление соединения таблиц `balance_feeds` и `run_results`, которое может содержать сотни миллионов строк и должно участвовать в операциях соединения с другими таблицами.

Суммируемые значения зависят только от атрибутов таблиц `balance_feeds` и `run_results`, поэтому было бы хорошо выполнить агрегирование сразу после соединения этих таблиц, однако ни одна из этих таблиц не содержит атрибуты `last_name` и `first_name`, по которым производится агрегирование, поэтому промежуточное агрегирование нужно выполнять по другому атрибуту (листинг 7.6).

Листинг 7.6

```
select
   last_name,
   first_name,
   sum(balance) gross
from
          pay_all_personnel p
    join  all_assignments a  on a.person_id=p.person_id
    join assignment_actions ac
```

```
      on a.assignment_id=ac.assighment_id
   join (
      select
         sum( scale* result_value) balance,
         assignment_action_id
      from
         run_results rr
      join balance_feeds bf on bf.value_id=rr.value_id
     where   b.balance_type='Gross'
      group by assignment_action) b
      on ac.assignment_action_id = b.assignment_action_id
where ac.run_date='1-jun-2005'
group by
   last_name,
   first_name
```

План выполнения этого запроса содержит дополнительную операцию группировки, которая очень существенно сокращает объем промежуточных результатов, участвующих в последующих операциях соединения (листинг 7.7).

Листинг. 7.7

```
GROUP BY(
   JOIN(
      TABLE SCAN(all_personnel)
      JOIN(
         TABLE SCAN(all_assignments)
         JOIN(
             FILTER(
            TABLE SCAN(assignment_actions))
            GROUP BY(
                JOIN(
                   TABLE SCAN(run_results)
                   FILTER(
                      TABLE SCAN(balance_feeds)
                      )
                    )
                )
             )
```

```
          )
       )
    )
 )
```

Время выполнения такого запроса оказывается приемлемым.

В реальности для упрощения написания программ, выбирающих значения балансов, внутренний подзапрос был выделен в отдельное представление (листинг 7.8), которое мы будем использовать в других примерах.

Листинг 7.8

```
create view pay_run_balances as
select
   sum( scale* result_value) balance_value,
   b.balance_type,
   asc.run_date,
   asc.assignment_id
from
          balance_feeds_f b,
   join   run_results rr,
      on   rr.value_id = b.value_id
   join assignment_actions asc
      on rr.assignment_action = asc.assignment_action
group by
     b.balance_type,
     asc.assignment_id,
     run_date
```

Это представление отличается от приведенного выше подзапроса тем, что в него включена дополнительная таблица `assignment_actions`, что позволяет несколько упростить текст запросов, в которых это представление используется.

Исходный запрос может быть записан так, как показано в листинге 7.9.

Листинг 7.9

```
Select
   last_name,
   first_name,
   sum(balance) gross
```

```
from pay_run_balances b,
   all_assignments a,
   all_personnel p
where b.assignment_id=a.assignment_id
   and a.person_id=p.person_id
   and run_date='1-jun-2005'
   and balance_type='Gross'
group by
   last_name,
   first_name
```

Использование подзапросов, однако, может создавать некоторые осложнения. В данном случае `assignment_id` является внешним ключом к таблице `assignment_actions`, но оптимизатор не может распознать его как внешний ключ, поскольку в подзапросе использован оператор `group by`. Поэтому в этом случае использование алгоритма на основе хеширования дает наилучший результат.

Каскадное агрегирование

При выполнении группировок могут вычисляться довольно сложные выражения, содержащие агрегирующие функции и значения атрибутов из разных таблиц. Например, при вычислении балансов значение атрибута `result_value` перед суммированием масштабируется (т. е. умножается на значение `scale`).

Если все или некоторые из таблиц, атрибуты которых используются в агрегирующих выражениях, велики по количеству строк, может быть полезным выполнить предварительную группировку, т. е. применить прием, продемонстрированный в предыдущем разделе, несколько раз.

Например, в запросе, вычисляющем балансы, можно выполнить предварительную группировку таблиц `balance_feeds` и `run_results`, при этом определение представления изменится следующим образом (листинг 7.10).

Листинг 7.10

```
create view pay_run_balances as
select
   sum(balance_value) balance_value,
   rb.balance_type,
   asc.run_date,
   asc.assignment_id
from
```

```
   (select
      sum( scale* result_value) balance_value,
      b.balance_type,
      rr.assignment_action_id
   from
          balance_feeds_f b,
      join   run_results rr,
         on   rr.value_id = b.value_id
      group by b.balance_type,
         rr.assignment_action_id) rb
   join assignment_actions asc
      on rb.assignment_action = asc.assignment_action
group by
     b.balance_type,
     asc.assignment_id,
     run_date
```

Конечно, такие каскадные группировки имеют смысл только в том случае, если в результате группировки происходит значительное сокращение количества строк. В рассматриваемом примере это не так, поэтому в реальности каскадная группировка не потребовалась.

Более точно целесообразность применения каскадной группировки можно оценить, исходя из того, что сложность операции группировки близка к сложности операции сортировки. Поэтому, если в результате предварительной группировки сложность последующих соединений уменьшается больше, чем стоимость сортировки, то каскадная группировка имеет смысл. Отметим, что с ростом размеров таблиц сложность соединения может расти быстрее, чем сложность сортировки.

Важно отметить, что далеко не всегда каскадная группировка выполняется также просто, как в этом примере, потому что не все агрегатные функции обладают свойствами коммутативности и ассоциативности, что необходимо для того, чтобы каскадные вычисления давали правильный ответ.

Конечно, функции `sum` и `count` обладают необходимыми свойствми, однако `avg` и другие более сложные функции этими свойствами не обладают. Поэтому, если каскадная группировка оказывается полезной для улучшения характеристик производительности, вместо функции `avg` во внутреннем подзапросе необходимо вычислять другие функции, на основе которых можно рассчитать требуемое значение. (Например, `avg(x)=sum(x)/count(x)` при условии, что множество агрегируемых значений непусто и не происходит потери точности.) Некоторые СУБД предоставляют специальные варианты агрегатных функций для каскадных группировок в своих диалектах SQL.

7.3.2. Как исключить многократные просмотры

В практике авторов довольно часто встречались задачи, подобные задачам с телефонами или со специальными типами, описанными в *главе 5*. Общим в подобных задачах является то, что требуется выбрать в одном запросе несколько значений, хранящихся в разных строках одной таблицы, и вывести эти значения в одной строке результата. Иными словами, нужно выбрать значения динамических атрибутов как значения статических.

Очевидное решение использует операцию соединения и поэтому приводит к многократному просмотру одной и той же большой таблицы. Мы рассмотрим несколько примеров таких задач, обращая внимание, во-первых, на то, как писать подобные запросы, и, во-вторых, какая стратегия настройки будет оптимальной.

Простой пример

Сначала рассмотрим пример с телефонами. Напомним, что в одной таблице хранятся телефоны компаний-клиентов разных типов. Запрос, который, как оказалось, нужно выполнять довольно часто, должен возвращать список компаний-клиентов, а также для каждой из компаний — номер телефона главного офиса, номер факса и дежурный пейджер. Запрос, который был первоначально написан разработчиком приложения, выглядел следующим образом (листинг 7.11).

Листинг 7.11

```
select
   p.company_name,
   p.phone_number   PHONE,
   f. phone_number  FAX,
   g.phone_number   PAGER
from
   (select * from all_phones where phone_type='PHONE') p,
   (select * from all_phones where phone_type='FAX') f,
   (select * from all_phones where  phone_type='PAGER') g
where
    p.company_name=f.company_name
    and  p.company_name=g.company_name
```

Эквивалентная запись этого запроса, не использующая вложенных подзапросов, может иметь вид, представленный в листинге 7.12.

Листинг 7.12

```
select
   p.company_name,
   p.phone_number   PHONE,
   f. phone_number  FAX,
   g.phone_number   PAGER
from
         all_phones p
   join  all_phones  f
      on  p.company_name=f.company_name
   join  all_phones  g
      on  p.company_name=g.company_name
where
       p.phone_type='PHONE'
   and f.phone_type='FAX'
   and g.phone_type='PAGER'
```

В этой записи, возможно, содержимое результата становится менее наглядным, но такая запись показывает, что будет на самом деле происходить при его выполнении. Подчеркнем, что оба запроса абсолютно эквивалентны и оптимизатор фактически преобразует первую запись во вторую. План выполнения этого запроса будет иметь вид, показанный в листинге 7.13.

Листинг 7.13

```
JOIN(
   FILTER(
      TABLE ACCESS(all_phones)
      )
   JOIN(
      FILTER(
         TABLE ACCESS(all_phones)
         )
      FILTER(
         TABLE ACCESS(all_phones)
         )
      )
   )
```

В этой записи мы не конкретизируем используемые алгоритмы, поскольку независимо от того, выбирал ли оптимизатор в качестве алгоритма для операции соединения NESTED LOOP или HASH JOIN, эффективность данного запроса была крайне низкой, так как три копии большой таблицы не могли быть размещены в оперативной памяти, и поэтому промежуточные результаты располагались во временной области на диске.

Заметим, что этот запрос (так же, как и исходный, написанный разработчиками) выбирает только те компании, для которых известны номера всех трех типов телефонных линий. Если требуется выбрать все компании независимо от наличия у них телефонных линий того или иного типа, запрос необходимо переписать так, как это показано в листинге 7.14.

Листинг 7.14

```
select
   c.company_name,
   p.phone_number   PHONE,
   f. phone_number  FAX,
   g.phone_number   PAGER
from
         all_companies c
   left join  all_phones p
      on  c.company_name=p.company_name
   left join  all_phones f
      on  c.company_name=f.company_name
   left join  all_phones  g
      on  c.company_name=g.company_name
where
       p.phone_type='PHONE'
   and f.phone_type='FAX'
   and g.phone_type='PAGER'
```

Эта запись показывает способ преобразования динамических атрибутов в статические, применимые в общем случае. Отсутствующие значения атрибутов будут представлены неопределенными значениями NULL. Практически этот способ вряд ли можно считать подходящим, потому что на каждый атрибут требуется отдельная операция соединения.

В этом конкретном случае, когда количество различных динамических атрибутов (т. е. типов телефонных линий) невелико, наилучшим решением данной проблемы будет изменение структуры данных, обсуждавшееся в предыдущих главах, т. е. замена динамических атрибутов на статические.

Сейчас нас, однако, интересует, каким образом выполнять подобные запросы в ситуациях, когда использование статических атрибутов по каким-либо причинам нецелесообразно. В этом случае можно применить прием, основанный на использовании операции группировки.

В нашем примере запрос, использующий операцию группировки и вырабатывающий требуемый результат, может иметь вид, представленный в листинге 7.15.

Листинг 7.15

```
select
   company_name,
   max(case phone_type
      when 'PHONE' then phone_number
      else NULL end) PHONE,
   max(case phone_type
      when 'FAX' then phone_number
      else NULL end)  FAX,
   max(case phone_type
      when 'PAGER' then phone_number
      else ULL  end) PAGER
from  all_phones
group by company_name
```

В данном случае таблица просматривается только один раз, и план выполнения этого запроса выглядит совсем просто:

```
GROUP BY(
   TABLE SCAN(all_phones)
   )
```

Обратим внимание на то, что этот запрос не эквивалентен ни одному из исходных. В нем выбираются только компании, имеющие хотя бы одну телефонную линию, в отличие от первого варианта, в котором выбираются только компании, имеющие все три типа телефонных линий, а также в отличие от второго, в котором выбираются все компании, даже не имеющие телефонных линий указанных типов.

Мы можем, однако, легко получить результат, эквивалентный результату первого запроса, добавляя для каждого типа телефонной линии условие вида:

```
having max(case phone_type
      when 'PHONE' then phone_number
      else ' ' end) is not NULL
```

Для того чтобы включить в выборку все компании, даже не имеющие телефонных линий нужного типа, достаточно выполнить операцию внешнего соединения с таблицей `all_companies`. Понадобится только одна операция соединения, однако при этом важно позаботиться о порядке, в котором будут выполняться операции группировки и соединения. Во многих случаях план

```
GROUP BY(
   JOIN(
      TABLE SCAN(all_companies)
      TABLE SCAN(all_phones)
      )
   )
```

может оказаться хуже, чем план

```
JOIN(
   TABLE SCAN(all_companies)
   GROUP BY(
      TABLE SCAN(all_phones)
      )
   )
```

В нашей практике встречались случаи, когда оптимизатор оказывался не в состоянии преобразовать первый план во второй.

Специальные информационные типы

Немного более сложное преобразование необходимо выполнить в следующем примере.

Информация, хранящаяся в специальных типах данных, необходима для получения очень большого количества отчетов, вырабатываемых с помощью других пакетов, ориентированных на обычные таблицы и поэтому непригодных для обработки специальных информационных типов (СИТ) непосредственно.

Определим представление, в котором в одну строку собираются данные из всех строк, связанные с некоторой строкой из таблицы `all_personnel`. Для того чтобы избежать многократных (по числу различных типов записей) операций соединения, используется операция группировки. Это представление позволяет использовать специальные информационные типы как обычные колонки в данном представлении.

Отличие запроса, представленного в листинге 7.16, от рассмотренного в предыдущем разделе, состоит в том, что одни и те же атрибуты из таблицы `sit_values` (фрагменты) используются в представлении несколько раз в зависимости от типа СИТ.

Листинг 7.16

```
select
   person_id,
   max(case sit_type_id
      when 1 then coalesce(pac.segment1,' ')
      else ' ' end) license_num,
   max(case sit_type_id
      when 1 then coalesce(pac.segment2,' ')
      else ' ' end) license_exp_date,
   max(case sit_type_id
      when 1 then coalesce(pac.segment4,' ')
      else ' ' end) license_class,
   max(case sit_type_id
      when 2 then coalesce(pac.segment1,' ')
      else ' ' end) lang_1,
   max(case sit_type_id
      when 2 then coalesce(pac.segment2,' ')
      else ' ' end) speak_1,
   max(case sit_type_id
      when 2 then coalesce(pac.segment3,' ')
      else ' ' end) read_1,
   max(case sit_type_id
      when 2 then coalesce(pac.segment4,' ')
      else ' ' end) write_1,
   max(case sit_type_id
      when 2 then coalesce(pac.segment5,' ')
      else ' ' end) lang_2,
   max(case sit_type_id
      when 2 then coalesce(pac.segment6,' ')
      else ' ' end) speak_2,
   max(case sit_type_id
      when 2 then coalesce(pac.segment7,' ')
      else ' ' end) read_2,
   max(case sit_type_id
      when 2 then coalesce(pac.segment8,' ')
      else ' ' end) write_2,
    ...
```

```
   max(case sit_type_id
      when 5 then coalesce(pac.segment1,' ')
      else ' ' end) emer_name,
   ...
from
   sit_links sl
   join sit_values sv
   on sl.values_id=sv.values_id
group by person_id;
```

Для большого количества разных СИТ и сегментов (в реальности — 75 сегментов из 14 разных СИТ) даже такое представление выполняется очень долго.

В реальной системе все СИТ имеют историю, т. е. для каждого сотрудника может существовать несколько записей с одинаковым типом СИТ, относящихся к разным промежуткам времени. Большинство отчетов должно соответствовать актуальному состоянию, что делает запрос еще более сложным.

Это представление является хорошим кандидатом для материализации, потому что оно включает сложные агрегатные функции, вычислительно сложное и используется многократно для различных отчетов. Раз в сутки происходит полное вычисление материализованного представления. Такой частоты достаточно для обеспечения актуальности отчетов. Заметим, что в материализованное представление включены только данные, требующие ресурсоемких вычислений, и не включены те, которые могут быть получены в результате простого соединения, например, ФИО, год рождения сотрудника и т. п.

7.3.3. Динамические таблицы

Рассмотрим еще один пример выбора значений динамических атрибутов с использованием полного просмотра и группировок.

Данный пример использует упрощенную структуру данных из Oracle Applications. Это приложение предоставляет пользователям возможность создавать собственные виртуальные таблицы, доступ к которым на уровне хранимых процедур не отличается от доступа к предопределенным таблицам этого пакета. Поэтому с точки зрения пользователя, которому полагается пользоваться исключительно процедурным интерфейсом пакета, эти структуры неотличимы от таблиц.

На уровне базы данных пользовательские таблицы хранятся в схеме, показанной на рис. 7.2.

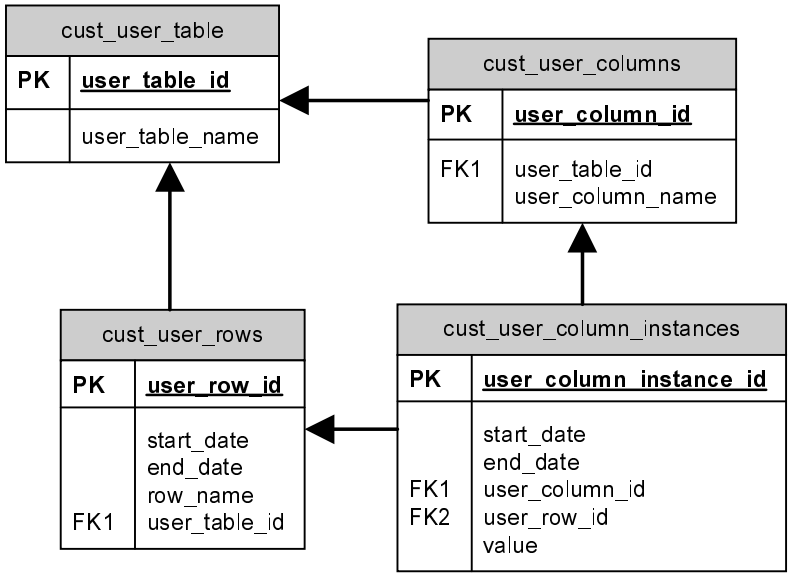


Рис. 7.2. Фрагмент схемы хранения таблиц

Причина, по которой процедурный интерфейс оказался неприменимым, достаточно очевидна: недостаточная производительность. Конечно, процедурный интерфейс не использует многократные операции соединения, однако для выборки каждого значения применяется отдельный курсор. Мы обсуждали неизбежные последствия подобных технических решений в *главе 4*. Процедурный интерфейс хорошо работает на небольших объемах, но в данном проекте на ранней фазе было принято решение использовать этот механизм для хранения больших объемов данных. К моменту, когда проблемы производительности стало невозможно игнорировать, изменять это решение было уже поздно.

Приемлемые характеристики производительности удалось получить, многократно применяя прием, показанный в *разд. 7.3.2* на простом примере: использованием группировки условных выражений.

Прежде всего, необходимо выбрать из этой структуры информацию, которая для обычных таблиц размещается в словаре данных (т. е. извлечь описание колонок динамической таблицы). Для того чтобы вычислить идентификаторы колонок пользовательской таблицы `my_table`, необходимо выполнить запрос, представленный в листинге 7.17, в котором имена колонок выбираются в одну строку результата.

Листинг 7.17

```
select t.user_table_id,
 max(case user_column_name
```

```
    when 'COLUMN_1' then column_id
    else null end) col_1_id,
 max(case user_column_name
    when 'COLUMN_2' then column_id
    else null end) col_2_id,
 max(case user_column_name
    when 'COLUMN_3' then column_id
    else null end) col_3_id
from cust_user_tables t,
   cust_user_columns c
where user_table_name='MY_TABLE_NAME'
   and t.user_table_id=c.user_table_id
```

План выполнения данного запроса, скорее всего, может быть таким:

```
    NESTED LOOP GROUP BY (
         FILTER (table_name,
            TABLE ACCESS INDEX (cust_user_tables) )
         TABLE SCAN (cust_user_columns)
         )
```

Далее для того, чтобы выбрать значения атрибутов, выполняется следующий запрос, показанный в листинге 7.18.

Листинг 7.18

```
select name,
      max( case vl.column_id
         when col_1_id then vl.value
         else null end)  column_1,
      max( case vl.column_id
         when col_2_id then vl.value
         else null end) column_2,
      max( case vl.column_id
         when col_3_id then vl.value
         else null end)  column_3
from cust_user_rows_f r,
        cust_user_column_instances_f vl,
(-- пол---- Приведенный выше оператор выборки колонок таблицы ---) cl
where r.user_table_id=cl.user_table_id
   and r.user_row_id=vl.user_row_id
   and sysdate between r. start_date and r.end_date
```

И, наконец, для полного вывода данных, хранящихся в данной пользовательской таблице, выполняется каскадная группировка, предложенная в листинге 7.18.

Листинг 7.18

```
select name,
      max( case vl. column_id
         when col_1_id then vl.value
         else null end)  column_1,
      max( case vl. column_id
         when col_2_id then vl.value
         else null end) column_2,
      max( case vl. column_id
         when col_3_id then vl.value
         else null end)  column_3
from
   cust_user_rows_f r,
   cust_user_column_instances_f vl,
   (select
       t.user_table_id,
      max( case user_column_name
         when 'COLUMN_1' then column_id
         else null end) col_1_id,
      max( case user_column_name
         when 'COLUMN_2' then column_id
         else null end) col_2_id,
      max( case user_column_name
         when 'COLUMN_3' then column_id
         else null end) col_3_id
   from
      cust_user_tables t,
      cust_user_columns c
   where
           user_table_name='MY_TABLE_NAME'
       and t.user_table_id=c.user_table_id
    group by t.user_table_id
    ) cl
where
```

```
      r.user_table_id=cl.user_table_id
   and r.user_row_id=vl.user_row_id
   and sysdate between r. start_date and r.end_date
```

План запроса будет выглядеть следующим образом:

```
HASH JOIN (
   HASH JOIN (
     NESTED LOOP (
         FILTER (table_name,
            TABLE ACCESS INDEX (cust_user_tables) )
         TABLE SCAN (cust_user_columns)
         )
   TABLE SCAN  (cust_user_column_instances_f)
 TABLE SCAN(cust_user_rows_f)
```

Записанный выше код не является в такой же мере динамическим, как исходное представление таблиц, поскольку максимальное количество извлекаемых атрибутов определяется текстом запроса. Кроме того, мы извлекаем только состояние данных на момент выполнения запроса и никак не реализуем потенциальную возможность использования этих таблиц как темпоральных. Конечно, извлечение данных по состоянию на любой другой момент времени в прошлом никаких осложнений не вызовет, однако в случае, если требуется сравнение значений, относящихся к разным моментам времени, наши запросы, скорее всего, придется существенно изменить.

7.3.4. Вычисление нескольких балансов

В *разд. 7.3.1* был рассмотрен запрос, вычисляющий значения балансов. При каждом выполнении этот запрос вычисляет значения балансов одного типа. Если требуется выбрать несколько разных балансов для каждого работника (например — `gross`, `net` и `federal tax`), то запрос придется выполнять отдельно для каждого типа баланса.

Можно исключить многократную обработку больших таблиц, заменяя критерий выборки типа баланса (указывая список вместо одного значения). При этом, однако, значения разных типов баланса будут размещаться в разных строках результата, а требуется, чтобы они были в одной строке для каждого сотрудника. Решение, очевидно, основано на использовании группировки условных выражений, точно так же, как мы это делали в примерах с телефонными линиями и динамическими таблицами.

Запрос, выбирающий значение трех балансов для каждого сотрудника, можно записать так, как это показано в листинге 7.19.

Листинг 7.19

```
Select
   last_name,
   first_name,
   sum( case balance_type
          when 'Gross'  then balance_value
          else 0 end ) gross,
   sum( case balance_type
          when 'Net'    then balance_value
          else 0 end ) net,
   sum( case balance_type
          when 'FedTax' then balance_value
          else 0 end ) fed
from all_personnel p,
        all_assignments a,
       pay_run_balances b
where p.person_id=a.person_id
          and a.assignment_id=b.assignment_id
         and run_date='31-jul-2005'
group by
   last_name,
   first_name;
```

Для выполнения этого запроса будет построен следующий план:

```
GROUP BY(
   HASH JOIN(
      TABLE SCAN(all_personnel)
      HASH JOIN(
         TABLE SCAN(all_assignments)
         HASH JOIN(
             FILTER(
            TABLE SCAN(assignment_actions))
            GROUP BY(
```

```
                HASH JOIN(
                   TABLE SCAN(run_results)
                   TABLE SCAN(balance_feeds)
                   )
                 )
               )
             )
           )
         )
      )
   )
```

Единственное отличие этого плана от плана запроса, вычисляющего один тип баланса, в том, что отсутствует фильтрация таблицы `balance_feeds`. Сама по себе эта таблица невелика, однако в результате фильтрации этой таблицы существенно сокращается количество строк в промежуточном результате, получаемом после ее соединения с таблицей `run_results`.

Поэтому целесообразно включить в запрос дополнительное условие, ограничивающее типы балансов, как показано в листинге 7.20.

Листинг 7.20

```
Select
   last_name,
   first_name,
   sum( case balance_type
          when 'Gross'  then balance_value
          else 0 end ) gross,
   sum( case balance_type
          when 'Net'    then balance_value
          else 0 end ) net,
   sum( case balance_type
          when 'FedTax' then balance_value
          else 0 end ) fed
from all_personnel p,
        all_assignments a,
       pay_run_balances b
where p.person_id=a.person_id
```

```
          and a.assignment_id=b.assignment_id
         and run_date='31-jul-2005'
         and balance_type in ('Gross', 'Net', 'FedTax')
group by
   last_name,
   first_name;
```

Следует подчеркнуть, что это условие является избыточным, так как оно никак не изменяет результат выполнения запроса. Единственная цель, с которой данный критерий включен в запрос, — сокращение размеров промежуточных результатов. Окончательный план выполнения запроса приобретает следующий вид:

```
GROUP BY(
   HASH JOIN(
      TABLE SCAN(all_personnel)
      HASH JOIN(
         TABLE SCAN(all_assignments)
         HASH JOIN(
             FILTER(
            TABLE SCAN(assignment_actions))
            GROUP BY(
                HASH JOIN(
                   TABLE SCAN(run_results)
                   FILTER(
                      TABLE SCAN(balance_feeds)
                      )
                   )
                )
             )
          )
      )
   )
```

Заметим, что такое ограничение имеет смысл только в том случае, если количество типов балансов, значения которых нужно вычислить, невелико.

Время выполнения всего запроса мало зависит от количества вычисляемых балансов, потому что самая большая таблица в любом случае обрабатывается полностью и только один раз, остальные большие таблицы также обрабатываются однократно.

Необходимо отметить, что алгоритмы, выбранные в этом плане (т. е. алгоритмы на основе хеширования), в данном случае выполняют работу за приемлемое время, однако используют очень большие объемы оперативной и внешней памяти.

7.3.5. Время выполнения или память?

Большой объем промежуточных результатов при использовании группировки условных выражений объясняется тем, что фактически при такой группировке сначала строится разреженная матрица с большим количеством пустых значений, которые исключаются только после агрегирования. При выполнении очень больших запросов потребности в памяти могут стать ограничивающим фактором.

Можно попробовать разделить процесс вычислений на несколько фрагментов, в каждом из которых обрабатывается только часть исходных данных. Сложность такого разбиения состоит в том, что атрибуты, по которым производится окончательная группировка, не входят в таблицы, с которых начинается вычисление, поэтому выделить какие-либо части результата и получить его отдельно не удается. То есть, конечно, ничто не мешает ограничить количество сотрудников, для которых вычисляются балансы, однако это никак не поможет сократить объем обрабатываемых исходных данных.

Возможное решение состоит в том, чтобы получить частичные суммы для всех строк окончательного результата и затем выполнить дополнительное агрегирование всех полученных строк. Для того чтобы гарантировать повторное использование памяти для промежуточных данных, каждое вычисление частичных сумм будем выполнять в отдельном курсоре и записывать полученные частичные суммы в таблицу `partial_balances`, атрибуты которой в точности соответствуют результатам нашего запроса.

Разбиение на порции будем производить по значениям `assignment_id`. Допустим, что максимальное значение этого атрибута не превышает миллион, и будем разбивать на части, содержащие не более 20 000 значений (листинг 7.21).

Листинг 7.21

```
begin
   for i in 1..50 loop
      insert into partial_balances
      Select
```

```
        last_name,
        first_name,
        sum( case balance_type
               when 'Gross'  then balance_value
               else 0 end ) gross,
        sum( case balance_type
               when 'Net'    then balance_value
               else 0 end ) net,
        sum( case balance_type
               when 'FedTax' then balance_value
               else 0 end ) fed
     from all_personnel p,
             all_assignments a,
            pay_run_balances b
     where p.person_id=a.person_id
              and a.assignment_id=b.assignment_id
             and run_date='31-jul-2005'
             and balance_type in ('Gross', 'Net', 'FedTax')
             and a.assignment_id between (i-1)*20000
                    and i*2000-1
             and b.assignment_id between (i-1)*20000
                    and i*2000-1
     group by
        last_name,
        first_name;
     end loop;
  end;
```

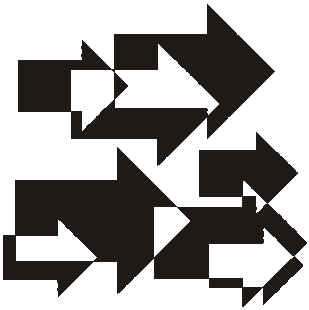
Для того чтобы несколько сократить время выполнения операции соединения, мы включили избыточное условие на значения `assignment_id` из таблицы `all_assignments`. После того как все частичные суммы вычислены, окончательный результат получается их агрегированием, как показано в листинге 7.22.

Листинг 7.22

```
Select
   last_name,
```

```
   first_name,
   sum(gross) gross,
   sum(net)   net,
   sum(fed)   fed
from partial_balances
group by
   last_name,
   first_name;
```

Очевидно, чем меньше размер порции, тем меньше потребуется памяти; и чем больше количество порций, тем больше будет общее время выполнения. Следовательно, изменяя размеры и, соответственно, количество порций, можно изменять размер необходимой временной памяти, т. е. обменивать память на время выполнения или наоборот.



Глава 8

Транзакции и производительность

Как известно, производительность СУБД может ограничиваться не только возможностями оборудования, но и внутренними механизмами СУБД, обеспечивающими корректность данных и выполняемых операций, в первую очередь — средств управления транзакциями.

Важная особенность, отличающая проблемы, связанные с транзакциями, состоит в том, что обнаружить существование таких проблем и диагностировать их можно, только анализируя поведение сервера базы данных в целом, а не отдельных приложений. Вопросы настройки сервера базы данных обсуждаются в *главе 10*, но проблемы управления транзакциями мы рассмотрим отдельно, так как для их диагностики и особенно решения требуются особые методы.

Проблемы производительности, связанные с транзакциями, возникают чаще всего при высокой интенсивности потока запросов (скажем, в системе одновременно обрабатываются сотни запросов). Эти проблемы довольно трудно анализировать в силу их динамического характера. Обычно эти проблемы невозможно воспроизвести и часто нельзя моделировать на тестовых копиях сервера.

Конечно, можно пытаться создать большую нагрузку с помощью системы стресс-тестов, однако при этом будет трудно доказать, что проблемы, наблюдаемые на тестах, адекватно моделируют поведение сервера, находящегося в производственной эксплуатации.

8.1. Как обнаружить проблемы, связанные с транзакциями?

Наиболее характерными признаками того, что проблемы производительности вызваны неправильно спроектированными транзакциями или неверными настройками сервера, являются следующие:

- неудовлетворительное время ответа сервера базы данных в сочетании с низкой загруженностью сервера (как процессора, так и дисковых систем). При этом, как правило, загрузка сервера приложений также является низкой;
- большой разброс времени выполнения запросов одного класса (скажем, от долей секунды до минут), который часто воспринимается пользователями как неоправданные задержки в работе системы;
- неоправданно большое время ответа при выполнении очень простых операций обновления;
- большое отличие времени ответа на короткие запросы при малой и большой нагрузках на систему, когда ресурсы сервера далеки от исчерпания.

Все эти признаки указывают на наличие очередей в системе, вызванных не ее перегруженностью. Запросы слишком часто оказываются в состоянии ожидания, хотя вычислительные мощности для их выполнения имеются.

Очереди могут возникать и по другим причинам. Например, в конфигурации сервера базы данных может быть установлен недостаточный уровень мультипрограммирования, т. е. максимальное количество открытых сеансов недостаточно для обслуживания поступающего потока заявок. Возможно, неправильно распределяется оперативная память или какие-либо из открытых сеансов не используются. Способы обнаружения подобных ситуаций и меры по их устранению обсуждаются в *главе 10*.

8.2. Основные понятия, связанные с транзакциями

Напомним основные определения, понятия и механизмы, связанные с поддержкой согласованности в базах данных.

Функции средств управления транзакциями в современных СУБД можно условно разделить на две категории:

- поддержка согласованности данных при конкурентном выполнении нескольких транзакций;
- обеспечение устойчивости результатов зафиксированных транзакций и восстановление согласованности после отказов системы.

Обе эти категории исключительно важны и трудно отделимы одна от другой, однако их влияние на производительность системы различно. Средства обеспечения устойчивости и восстановления (т. е. ведение журналов системы и создание резервных копий), конечно, используют некоторую часть вычислительных ресурсов сервера базы данных. Чем выше надежность, обеспечиваемая этими средствами, тем больше доля ресурсов, которая ими используется. Поэтому конфигурация средств ведения журналов выбирается в зависимости от требований к надежности прикладной системы, но в любом случае работа этих средств вполне предсказуема и не может вызвать симптомы, охарактеризованные выше.

Поэтому далее в этой главе мы будем обсуждать только вопросы, связанные с конкурентным выполнением транзакций в нормальном режиме (т. е. в отсутствие отказов системы).

Будем называть *транзакцией* конечное упорядоченное множество операций (запросов) приложения. Приложение формирует транзакцию таким образом, что транзакция переводит согласованное состояние базы данных в другое согласованное, если:

- транзакция выполнена до конца;
- при выполнении транзакции не было помех со стороны других транзакций.

Транзакция может быть завершена одним из двух способов. Успешное завершение называется фиксацией (commit), неуспешное — обрывом (abort). В последнем случае СУБД обязана восстановить базу данных в состояние, в котором она была перед началом выполнения транзакции, т. е. тоже в согласованное состояние.

Из этого следует, что последовательное выполнение транзакций (без конкурентности) сохраняет согласованность базы данных, однако такой метод неприемлем по соображениям производительности.

Расписанием для некоторого набора транзакций называется упорядочение операций этих транзакций (возможно, конкурентное, т. е. операции из разных транзакций могут чередоваться). Расписание, упомянутое в предыдущем абзаце, в котором конкурентность отсутствует, называется серийным.

Для того чтобы некоторое расписание было корректным, необходимо, чтобы оно было эквивалентно серийному, или, как говорят, было *сериализуемо.* Существуют различные виды эквивалентности расписаний. Единственным видом эквивалентности расписаний, эффективно проверяемым и поэтому используемым на практике, является эквивалентность по конфликтам.

Говорят, что две операции находятся в конфликте, если выполнены следующие условия:

- они обрабатывают один и тот же элемент данных;
- они принадлежат разным транзакциям;
- по крайней мере одна из них является операцией обновления.

Расписания эквивалентны по конфликтам, если в этих расписаниях совпадают множества конфликтов.

Менее формально можно сказать: для того, чтобы конкурентное расписание было корректным, необходимо, чтобы в нем конфликтующие операции для каждого элемента данных выполнялись в том же порядке, в котором бы они выполнялись в некотором серийном расписании.

Можно сказать, что основная задача механизмов управления транзакциями — обеспечение *изоляции* транзакции от других транзакций, выполняемых конкурентно, для того, чтобы обеспечить сериализуемость их выполнения.

Среди большого количества различных методов обеспечения изоляции единственным широко известным и наиболее широко используемым является метод, основанный на использовании замков. Для каждой операции обработки данных, прежде чем эта операция может начаться, должен быть получен замок на все элементы данных, используемые или модифицируемые в этой операции. Если замки не могут быть установлены, то эта операция и, следовательно, вся транзакция переводится в состояние ожидания до тех пор, пока несовместимые замки, установленные другой транзакцией, не будут сняты. Таким образом, перевод операций в состояние ожидания гарантирует их выполнение в правильном порядке.

Конечно, одни только замки не решают проблему сериализуемости расписаний. Требуется определенная дисциплина установки и снятия замков, которая определяется протоколом управления транзакциями. Существует довольно много различных протоколов, обеспечивающих сериализуемость. Наиболее простые из них широко известны, но не могут обеспечить высокую степерь конкурентности, поэтому в высокопроизводительных системах применяются более сложные протоколы.

8.3. Причины снижения производительности

Самое важное сейчас для нас слово из предыдущего раздела — *ожидание*. Механизмы управления транзакциями могут переводить транзакцию, а, следовательно, и приложение, в состояние ожидания, таким образом ухудшать время ответа и пропускную способность системы, т. е. снижая производительность.

Степень ухудшения производительности зависит от многих факторов.

- Время работы транзакций. Чем дольше выполняется транзакция, тем выше вероятность того, что обрабатываемые ею элементы данных понадобятся для работы другой транзакции, выполняемой конкурентно.

- Объем обрабатываемых данных. Чем больше относительная доля данных, обрабатываемая транзакцией, тем выше вероятность того, что другие транзакции будут пытаться конкурентно выполнить конфликтующие операции и поэтому окажутся в состоянии ожидания.
- Гранулярность замков. Чем крупнее элементы данных, на которые устанавливаются замки, тем меньше различных элементов в базе данных и, следовательно, тем меньше транзакций может выполняться конкурентно.

Приведенные выше качественные характеристики поведения транзакций можно формализовать, вводя понятие рабочих множеств транзакции — множество элементов, которые транзакция читает и которые обрабатывает, — и оценивая количество элементов в этих множествах по отношению к общему количеству элементов в базе данных. На основе этих оценок можно определить вероятность того, что транзакция окажется в состоянии ожидания и далее оценить предельное количество транзакций, которые могут выполняться конкурентно, а также другие характеристики производительности системы управления транзакциями. Более детальный обзор этих методов можно найти в статье [13]. Мы не будем рассматривать эти методы детально.

Влияние этих факторов зависит от того, каким образом реализуется изоляция в СУБД. В простейших реализациях, как правило, используется строгий двухфазный протокол управления транзакциями, в котором все установленные для транзакции замки сохраняются до завершения (т. е. фиксации или обрыва) транзакции.

В высокопроизводительных системах используются более сложные многоуровневые многоверсионные протоколы управления замками, что позволяет получить значительно большую степень конкурентности, чем при использовании простых вариантов строгого двухфазного протокола. Однако при неудачном проектировании транзакций проблемы могут возникать даже в высокопроизводительных СУБД.

Степень влияния перечисленных факторов зависит от частоты использования тех или иных элементов данных. Не имеет значения, насколько велика база данных, если почти все транзакции используют только небольшую ее часть.

Если, допустим, каждая транзакция, в дополнение к другой работе, модифицирует один и тот же элемент данных, то в такой системе фактически все транзакции будут выполняться последовательно. В более реальных случаях могут существовать очень часто используемые части базы данных — узкие места (hot spots). Транзакции, считывающие или модифицирующие такие элементы данных, будут относительно часто попадать в очередь ожидания, даже если они используют мало данных и выполняются быстро.

Выполнение транзакций, обрабатывающих большие объемы данных, также может привести к задержкам других транзакций. И, наконец, транзакции, выполняющие большие по объему вычисления, могут вызывать задержки, даже если они обрабатывают относительно небольшое количество данных.

Для решения проблем производительности, связанных с транзакциями, можно использовать различные методы или их сочетание, а именно:

- разделение одной транзакции на несколько независимых транзакций;
- снижение уровня изоляции при выполнении транзакций;
- устранение узких мест.

Любая из этих мер потенциально может повлиять на корректность получаемых приложением результатов и поэтому их применение должно сопровождаться тщательным анализом допустимости предполагаемых изменений. Мы исходим из того, что никакие меры по увеличению производительности не должны приводить к некорректной работе прикладной системы.

8.4. Разбиение транзакций

Наиболее простой способ разбиения транзакций на части меньшего размера навязывается некоторыми СУБД и применяется некоторыми разработчиками, недооценивающими роль согласованности: фиксация транзакций выполняется после каждой операции (автоматическая фиксация, autocommit). Использование этого режима лучше, чем полный отказ от использования транзакций, потому что обеспечивает восстановимость базы данных после отказов системы, однако его, как правило, нельзя применять в потенциально многопользовательских системах.

Главная проблема состоит в том, что слишком большое количество согласованных состояний усложняет работу приложения: оно должно быть готово выполнять правильные действия, когда база данных находится в состояниях, которые не могут возникнуть при более сильных условиях согласованности.

Транзакции должны представлять собой логически осмысленные единицы работы, и произвольное разделение этих единиц на части может приводить к потерям или дублированию данных. (Например, в классической банковской транзакции снятие средств с одного счета и занесение на другой не могут разделяться).

В некоторых случаях, однако, риск потери или повторных обновлений может быть вполне допустимым, т. е. потеря обновлений может не приводить к разрушению согласованности данных. Например, операции копирования могут быть повторены без потери согласованности.

Не менее важно и то, что слишком короткие транзакции могут привести к снижению производительности сервера базы данных. Фиксация транзакции является довольно медленной операцией и обязательно включает синхронные операции записи на диск (в отличие от операций обновления, которые могут выполняться полностью в буфере, находящемся в оперативной памяти, который записывается на диск асинхронно). Транзакция, выполняющая обновление нескольких сотен строк таблиц, скорее всего, не вызовет заметных задержек в работе системы, если, конечно, наиболее часто выполняемые транзакции обновляют значительно меньшее количество данных.

8.4.1. Транзакции большой продолжительности

Рассмотрим пример, в котором иллюстрируются диагностика системы с транзакционными проблемами и разделение продолжительной транзакции на части без потери логической согласованности базы данных.

В системе, обслуживающей пользователей через интернет-интерфейс, использовалась архитектура с сервером приложений. Кроме интерактивных запросов, система обрабатывала в фоновом режиме сообщения, приходившие из других систем, в основном эти сообщения содержат новые данные для ввода в систему.

После появления жалоб пользователей на неоправданные задержки (до нескольких минут) при обновлениях базы данных сервер базы данных и приложение, обрабатывающее сообщения, были перенесены на отдельные компьютеры, однако это не помогло решить проблему. Затем к решению проблемы был привлечен эксперт по настройке баз данных.

В результате анализа работы серверов удалось установить следующее. Для обработки запросов пользователей на сервере базы данных работает несколько сотен процессов, что обеспечивает отсутствие входных очередей на обработку запросов. Загрузка сервера баз данных невелика и составляла 3—6% мощности процессора. Примерно такая же степень загруженности наблюдалась и для дисковых устройств. Загрузка сервера интерактивных приложений была также невелика, а загрузка сервера обработки сообщений была близка к 100%, потому что обработка входных сообщений оказалась вычислительно трудоемкой. Программа обработки сообщений очень мало использовала базу данных и поэтому была, по мнению заказчика, вне подозрений.

Первичный анализ показал, что проблема связана с управлением транзакциями, однако выявить какие-либо длительно выполняемые операции не удалось. Все транзакции состояли из небольших операций выборки данных или обновления и не могли приводить к таким длительным задержкам.

Анализ несколько усложнялся тем, что все приложения использовали обезличивающий интерфейс (т. е. все сеансы выглядели, с точки зрения СУБД,

одинаково), поэтому идентификация приложений была затруднена. Поскольку измерения проводились на рабочей системе, какие-либо модификации или трассировки, значительно нагружающие систему, были невозможны.

Проблему удалось локализовать в результате анализа количества транзакций, находящихся в состоянии ожидания. Оказалось, что зависимость количества ожидающих транзакций от времени имеет ярко выраженную пилообразную форму: количество ожидающих транзакций росло в течение нескольких минут, после чего падало до нуля и снова начинало расти. Период этой зависимости примерно соответствовал продолжительности обработки одного входного сообщения, что позволило построить модель поведения системы и найти решение.

Очевидно, что причиной было узкое место в базе данных. Для его обнаружения пришлось анализировать текст запросов, находящихся в состоянии ожидания.

В этой базе данных для получения суррогатных ключей применялись счетчики, хранимые в пользовательской таблице. Естественно, эти счетчики потенциально являлись узким местом, так как при любом назначении суррогатных уникальных идентификаторов строк требовалось обновление счетчиков. Это узкое место, однако, не мешало нормальной работе системы, потому что нагрузка на систему была невелика, а обновляющие транзакции — короткими.

Единственным исключением оказалась транзакция, обрабатывающая входные сообщения. Работа этой транзакции начиналась с получения уникальных идентификаторов для записи сообщения в базу данных, т. е. обновления счетчиков. Затем в течение нескольких минут эта транзакция вообще не обращалась к базе данных, выполняя обработку входного сообщения, однако счетчики были в это время недоступны для других транзакций, что и привело к возникновению проблемы.

Значения счетчиков, полученные перед обработкой сообщения, были необходимы для обработки сообщения (точнее, с целью формирования выходных строк для записи в базу данных), поэтому их значения запрашивались до его обработки, а не после.

Стандартное решение могло бы состоять в том, чтобы использовать для назначения суррогатов системные средства, предоставляемые СУБД, так как для них применяется другой протокол обеспечения корректности, не основанный на замках, удерживаемых до конца транзакции. Это решение, однако, потребовало бы переделки всех приложений.

Решение, рекомендованное экспертом, заключается в том, чтобы фиксировать транзакцию сразу после получения значений суррогатов, а запись обработанного сообщения в базу данных выполнять в другой транзакции. Это

решение связано с риском потери значения суррогата в случае отказа системы во время обработки сообщения.

Такая потеря, однако, не является нарушением согласованности, так как неиспользуемые значения суррогатов не разрушают структуру хранимых данных. Более того, неиспользуемые значения идентификаторов могут возникать и по другим причинам, например, в результате удаления строк из таблиц базы данных. Вероятность рестарта относительно невелика, поэтому исчерпания пространства суррогатов не произойдет.

Обратим внимание на то, что одна транзакция продолжительностью в несколько минут была заменена на две, первая из которых продолжалась несколько миллисекунд, а вторая — сотни миллисекунд, но не затрагивала при этом узкие места.

Подчеркнем еще раз, что рассмотренная в этом примере проблема могла быть обнаружена и решена только на уровне сервера базы данных. Проблема возникла в результате взаимодействия различных программных компонентов, написанных в независимых компаниях, поэтому ни одна из этих компаний не могла обнаружить эту проблему при тестировании. Другая важная особенность этого примера состоит в том, что удалось найти компактную характеристику, позволившую локализовать причину возникновения проблемы.

В общем случае замена одной транзакции на несколько транзакций меньшего размера (по времени выполнения и по количеству вовлеченных элементов данных) означает отказ от атомарности и от изоляции исходной транзакции. Насколько допустимо то и другое, полностью зависит от семантики приложения.

8.4.2. Массовые обновления

Выполнение массовых обновлений может занимать довольно продолжительное время. Независимо от того, насколько эффективный протокол управления транзакциями реализован в СУБД, строки таблиц, подвергаемые изменению, должны быть недоступны для других транзакций до окончания обновляющей транзакции. Поскольку большая таблица составляет значительную часть базы данных, а транзакция, выполняющая изменение этой таблицы, является относительно долгой, массовые обновления с большой вероятностью приводят к задержкам других транзакций, по крайней мере, обновляющих ту же таблицу (некоторые протоколы допускают выполнение только читающих транзакций конкурентно с выполнением обновлений).

Если невозможно выполнить большое обновление в период низкой загруженности системы (для многих систем таким периодом является ночное время), то очевидным решением является разбиение большой обновляющей транзакции на независимые транзакции меньшего размера, выполнение которых завершается за приемлемое время.

При этом по необходимости приходится считать, что частично обновленная база данных находится в согласованном состоянии, поскольку цель разбиения как раз и состоит в том, чтобы открыть доступ к обновляемым данным из других транзакций. Главная проблема, однако, заключается в другом: необходимо гарантировать корректность выполнения обновления в случае отказа системы или приложения, выполняющего серию обновляющих транзакций. Требуется гарантировать, что в случае, если выполнение приложения не было завершено, эффект от повторного запуска этого приложения будет таким же, как от его однократного выполнения.

Сложность этой задачи различна для различных операций модификации данных.

Проще всего обстоит дело с операцией удаления: поскольку повторное удаление тех же строк эквивалентно пустому оператору, никаких специальных мер по предотвращению повторного выполнения не требуется.

Операция вставки также не может быть выполнена повторно, если в таблице имеется первичный ключ или, по крайней мере, ограничение целостности, определяющее уникальный атрибут или комбинацию атрибутов. Однако в случае повторного выполнения операция отвергается. Для того чтобы предотвратить возникновение исключительной ситуации, необходимо включить в оператор вставки условия, предотвращающие повторную вставку идентичных строк.

Для операции `update` необходимо различать два случая в зависимости от того, используются ли старые значения изменяемых колонок при вычислении новых значений или нет.

Если новые значения не зависят от старых, то повторное выполнение операции не приведет к некорректному результату, однако на его выполнение будут затрачены ресурсы системы, и, поскольку процесс обновления занимает значительное время, этим не следует пренебрегать.

Если же новое значение обновляемых элементов данных может зависеть от старых, то повторное выполнение недопустимо.

Решение состоит в том, чтобы регистрировать выполненную часть работы по обновлению в базе данных, производя запись, регистрирующую обновление, в той же транзакции, что и обновляемая порция данных.

Способ регистрации должен быть, с одной стороны, не очень трудоемким, чтобы не слишком заметно увеличивать объем работы по обновлению, с другой, должен достаточно точно определять обработанную порцию данных. Далеко не полный перечень возможных вариантов решения включает следующие альтернативы.

- Отмечать факт обновления в специально выделенном атрибуте обновляемой строки. Отсутствие этой пометки может проверяться при селекции строк для обновления. Необходимые дополнительные затраты ре-

сурсов незначительны, потому что помечаются строки, которые все равно необходимо изменить.

- Обновляемый набор данных разбивается на логические группы и делается одна запись для каждой группы. Например, для обновления списка сотрудников такими группами могут быть отделы (при условии, что каждый сотрудник работает только в одном отделе).
- Разбиение записей на группы по счетчику и регистрация идентификатора последней обработанной строки (при условии, что обработка выполняется в порядке возрастания идентификаторов).

Читатель легко сможет оценить возможные последствия каждого из этих решений, построить примеры потенциальных несогласованностей, которые могут возникнуть в результате их применения, и предложить свои альтернативы, учитывая особенности выполняемых изменений и семантику прикладной системы.

8.4.3. Агрегирование больших объемов данных

Проблемы производительности могут возникать не только вследствие больших по объему обновлений. Рассмотрим запрос

```
select avg(amount) from large_table
```

Для получения единственной строки (содержащей одно число) необходимо обработать значения из всех строк таблицы (сейчас не имеет значения, используется для этого просмотр таблицы или только просмотр индекса, в обоих случаях требуется логический доступ к значениям).

В высокопроизводительных системах, использующих многоверсионные протоколы управления транзакциями, подобные запросы никаких проблем не вызывают, однако в более простых системах любая транзакция, обновляющая ту же таблицу, будет ожидать завершения транзакции, в которой выполняется рассматриваемый запрос.

Если такая проблема возникает, одним из решений может быть, как и для больших обновлений, разбиение работы большого объема на порции. При этом для каждой порции необходимо вычислять значения функций `sum` и `count`, суммировать получаемые значения для всех порций и затем вычислить искомое значение. По существу необходимо выполнить каскадную группировку, рассмотренную в *главе 7*, однако в данном случае каскадная группировка выполняется не в рамках одного запроса, а реализуется процедурно, потому что во время вычисления необходимо выполнять операцию фиксации транзакций `commit`.

При этом, конечно, результат может получиться некорректным, т. е. не соответствующим ни одному из состояний базы данных, потому что состояние таблицы может быть изменено другими транзакциями, выполняемыми между выборкой значения для разных порций.

8.5. Уровни изоляции

Стандарт SQL предусматривает возможность выбора правил конкурентного выполнения транзакций. В дополнение к сериализуемому выполнению (которое является единственным методом, для которого корректность гарантируется теорией), допускаются ослабленные варианты изоляции вплоть до полного отказа от использования замков, т. е. полного отказа от поддержки согласованности.

Подчеркнем, что любое ослабление правил изоляции потенциально может приводить к нарушению согласованности данных и появлению некоторых нежелательных эффектов (фантомов, грязного чтения и др.), т. е. ухудшает качество прикладной системы.

В действительности многие системы используют более низкий уровень изоляции, чем сериализуемость, по умолчанию, и это означает, что в таких системах полномасштабная поддержка согласованности базы данных фактически почти никогда не применяется. Чаще всего подобный компромисс вполне оправдан, однако необходимо, чтобы он принимался проектировщиками прикладной системы осознанно.

В связи с этим важно указать случаи, в которых снижение уровня изоляции фактически не приводит к потере уровня согласованности.

8.5.1. Результаты большого объема

Если в интерактивном приложении пользователю необходимо вывести результаты большого объема, например, список объектов, удовлетворяющих критериям поиска, обычно этот вывод делится на страницы. Если при этом средний слой приложения реализован как сервер без состояния (что чаще всего делается, например, для интернет-приложений), то фактически при переходе на следующую страницу весь запрос выполняется заново, естественно, в рамках другой транзакции. Пользователь при этом видит, вообще говоря, несогласованные состояния разных страниц.

Естественно, в этом случае нет никакого смысла устанавливать высокую степень изоляции и для каждого запроса в отдельности необходимо только обеспечить достаточный уровень изоляции для обновляющих транзакций и включить в приложение проверку корректности обновлений непосредственно перед выполнением обновлений (т. е. в той же транзакции).

Таким образом, этот подход предполагает наличие двух типов транзакций в системе:

- читающие транзакции — не изменяют состояние базы данных и выполняются с ослабленным уровнем изоляции;
- обновляющие транзакции — должны производить дополнительную проверку корректности и выполняются как сериализуемые.

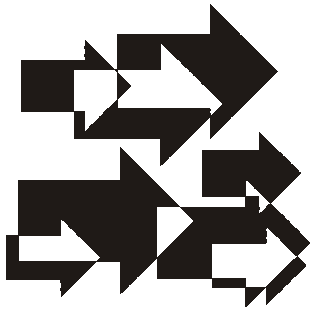
Наиболее широко используемый способ проверки корректности состоит в сравнении метки времени обновления, прочитанной в предварительной читающей транзакции, с текущим значением этой метки в момент обновления.

8.5.2. Идентификация сеансов

Для многих приложений, даже реализованных как сервер без состояния, важно понятие сеанса приложения. Например, состояние корзины анонимного покупателя в электронном магазине должно сохраняться в течение сеанса.

Одним из наиболее естественных способов хранения таких данных является запись уникального идентификатора сеанса вместе с каждой строкой данных, созданной для этого сеанса. При этом, несмотря на то, что система может обрабатывать конкурентно большое количество запросов, конкурентное выполнение запросов, связанных с одним сеансом, невозможно в силу особенностей реализации.

Фактически в этом случае данные не являются разделяемыми между пользователями системы, и, следовательно, применение системных механизмов базы данных для обеспечения изоляции не требуется вовсе.



Глава 9

Параллельные системы

В данной главе мы обсуждаем использование параллелизма для улучшения характеристик производительности системы. Наличие или отсутствие параллелизма никак не влияет на функциональные возможности системы, т. е. может влиять только на характеристики производительности.

9.1. Критерии качества для параллельных систем

Современные вычислительные системы являются глубоко параллельными. Для того чтобы добиться высокой производительности процессоров, необходимо распараллеливать разные этапы выполнения инструкций; кроме того, внешние устройства работают одновременно с процессором и друг с другом и т. д. Все эти виды параллелизма обеспечиваются аппаратурой, поэтому проектировщики и разработчики прикладных систем практически не могут влиять на их использование. Кроме того, операционные системы обычно обеспечивают псевдопараллельное выполнение различных процессов.

В данной главе мы будем рассматривать только такие виды параллелизма, которые находятся под контролем проектировщиков и разработчиков прикладной системы, т. е. на уровне базы данных, сервера приложений (если он используется) и реализации самого приложения.

Для того чтобы говорить о реальном параллелизме на уровне прикладной системы, необходимо, чтобы в конфигурации оборудования имелись необходимые устройства. Более точно будем предполагать, что конфигурация оборудования включает несколько процессоров и несколько устройств, используемых в качестве носителя базы данных (т. е. дисков).

Необходимо отметить, что разработка и отладка программного кода, использующего параллелизм или даже псевдопараллелизм, существенно сложнее, а также требует более высокой квалификации разработчиков, и, следовательно, существенно дороже, чем разработка обычного кода. Поэтому реально параллелизм может использоваться тогда, когда это обеспечивается базовым программным обеспечением (т. е. операционной системой, СУБД или сервером приложений). При этом для того, чтобы использовать параллелизм, не требуется изменение программного кода приложения, достаточно изменить конфигурацию оборудования и базовых программных средств, а также, возможно, схему базы данных.

Существует несколько различных архитектур оборудования, удовлетворяющих этим условиям: процессоры могут иметь или не иметь общую оперативную память, а также собственную оперативную память, недоступную для других процессоров (естественно, доступ хоть к какой-нибудь памяти необходим). Диски могут быть разделяемыми или управление каждым из дисков может выполняться только одним процессором.

Наиболее часто рассматриваются следующие варианты архитектур вычислительных систем для параллельной обработки:

- SM (shared memory) — многопроцессорные вычислительные установки с общей памятью;
- SD (shared disks) — вычислительные системы, в которых дисковые устройства могут непосредственно использоваться разными компьютерами (через общее устройство управления или по сети);
- SN (shared nothing) — системы без разделяемой памяти и дисков, т. е. комплексы вычислительных систем, связанные широкополосными сетями (кластеры).

Можно рассматривать и различные варианты гибридных архитектур, например, кластеры многопроцессорных систем, а также неоднородные комплексы.

Эти особенности архитектуры, безусловно, важны для реализации параллелизма, но сейчас мы обсудим цели использования параллелизма и критерии, которые не зависят от архитектуры параллельной вычислительной системы.

В любом случае целью использования параллелизма является улучшение работы системы. Нас будут интересовать не абсолютные характеристики системы, а то, насколько эти характеристики улучшаются при использовании более мощных конфигураций оборудования. Улучшение работы системы, естественно, должно достигаться за счет параллельной работы процессоров и дисков. Поэтому мы будем сравнивать систему, содержащую один процессор и один диск, с системой, содержащей n процессоров и n дисков. Кроме этого, важно учитывать и размер базы данных, поэтому иногда мы

будем предполагать, что на параллельной системе хранится база данных в n раз большего размера.

В идеале увеличение количества аппаратных ресурсов в n раз могло бы приводить к пропорциональному улучшению характеристик системы, однако в реальности такое улучшение недостижимо по следующим причинам:

- не вся работа, необходимая для формирования ответа системы, может быть выполнена параллельно — всегда имеется какая-то часть, которая должна быть выполнена последовательно (например, возврат результатов пользователю или завершающий проход алгоритма сортировки);
- может потребоваться дополнительная работа для синхронизации параллельных устройств и процессов;
- некоторые компоненты системы могут простаивать из-за неравномерного распределения нагрузки.

Вторая из указанных причин способна приводить даже к ухудшению характеристик системы.

Наиболее важными характеристиками прикладной системы являются время ответа и пропускная способность. Будем обозначать через $R(m, n, s)$ и $T(m, n, s)$ соответственно время ответа и пропускную способность системы, использующей m процессоров, n дисков и обслуживающую базу данных размера s..

Эффект от использования параллелизма, оказываемый на время ответа для отдельных запросов, можно охарактеризовать с помощью ускорения, которое определяется как

$$R(1,1,1)/R(n,n,1).$$

Иными словами, характеристика ускорения показывает, во сколько раз уменьшается время ответа при увеличении мощности вычислительных ресурсов в n раз при сохранении размеров базы данных. На рис. 9.1 показаны варианты возможного поведения этой характеристики.

Идеальным было бы линейное ускорение, однако оно недостижимо даже теоретически, поскольку при выполнении любого запроса некоторая часть работы не может быть выполнена параллельно. В реальности может оказаться, что увеличение степени параллелизма может приводить не к улучшению, а ухудшению времени ответа.

Как правило, характеристика ускорения важна для запросов, требующих большого времени для своего выполнения.

Другие характеристики качества параллельной системы связаны с понятием масштабируемости. В последние годы, в связи с появлением общепринятых стандартов организации взаимодействия программных компонентов в распределенных системах, значение масштабируемости усиленно подчеркивается в технической и коммерческой (рекламной) литературе.

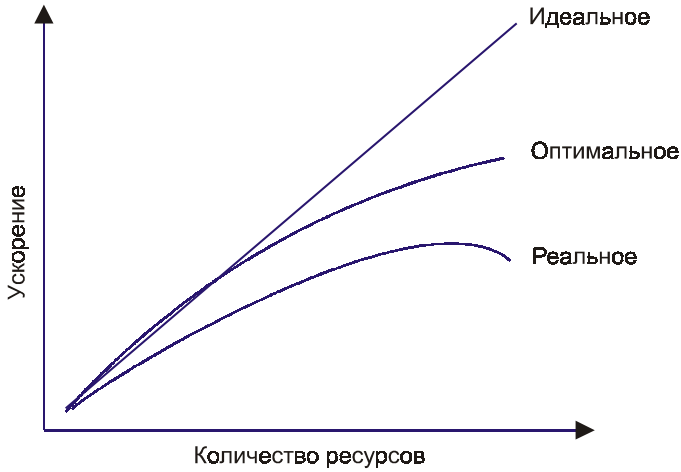


Рис. 9.1. Ускорение

Как и многие другие модные термины, понятие масштабируемости часто используется неточно или в переносном смысле. Например, масштабируемостью называют возможность расширения функциональности системы или вообще не определяют значение этого термина. Мы будем использовать этот термин в определенном смысле, который приведем далее в этом разделе.

На практике важность масштабируемости не столь велика по следующим причинам:

- необходимость масштабирования ограничена реальными потребностями объекта, обслуживаемого прикладной системой;
- при значительном увеличении нагрузки на систему или объемов данных, скорее всего, потребуется перепроектирование системы, так как простое увеличение количества параллельно работающего оборудования вряд ли решит проблемы производительности.

Перейдем к точному определению понятия масштабируемости.

Масштабируемость по пропускной способности определяется как

$$T(n, n, n)/T(1, 1, 1).$$

Иными словами, масштабируемость по пропускной способности показывает, как изменяется пропускная способность системы при одновременном увеличении вычислительной мощности этой системы и количества хранимых в ней данных. Возможное поведение этой характеристики показано на рис. 9.2.

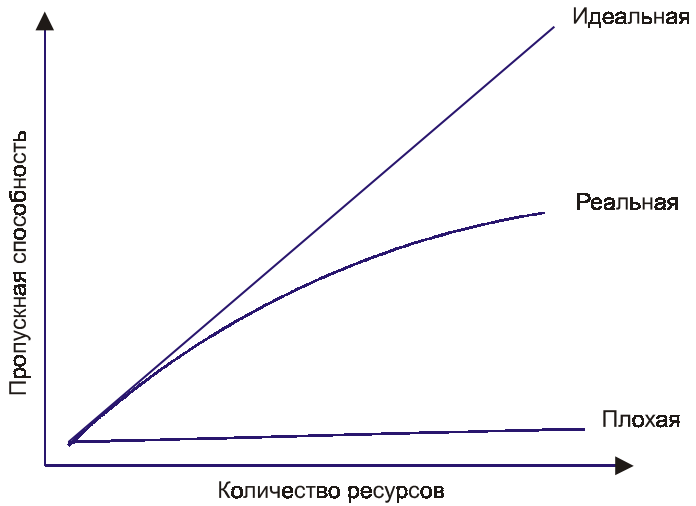


Рис. 9.2. Масштабируемость по пропускной способности

Напомним, что пропускная способность системы — это количество запросов, которые система способна выполнить за единицу времени.

Как и ускорение, в идеальном случае масштабируемость по времени должна линейно зависеть от степени параллелизма. Конечно, на практике идеальная масштабируемость недостижима. Очень плохая масштабируемость характеризуется константой: ресурсы системы увеличились в несколько раз, а количество работы, которое она может выполнить, осталось почти прежним.

Масштабируемость по времени ответа определяется как

$$R(n, n, n) / R(1, 1, 1).$$

Эта характеристика показывает, как изменяется время ответа при увеличении как объема базы данных, так и мощности вычислительной системы. Возможное поведение масштабируемости по времени ответа показано на рис. 9.3.

Для этой характеристики, наоборот, идеальное поведение описывается константой: при увеличении объема базы данных время ответа не ухудшается (естественно, благодаря соответствующему увеличению мощности вычислительной системы). При плохом использовании возможностей параллелизма время ответа может расти.

Рассмотренные здесь характеристики дают возможность только грубой оценки качества реализации системы. Очевидно, никакой параллелизм не может решить проблемы производительности на больших объемах данных, если трудоемкость алгоритмов, применяемых в прикладной системе, зависит

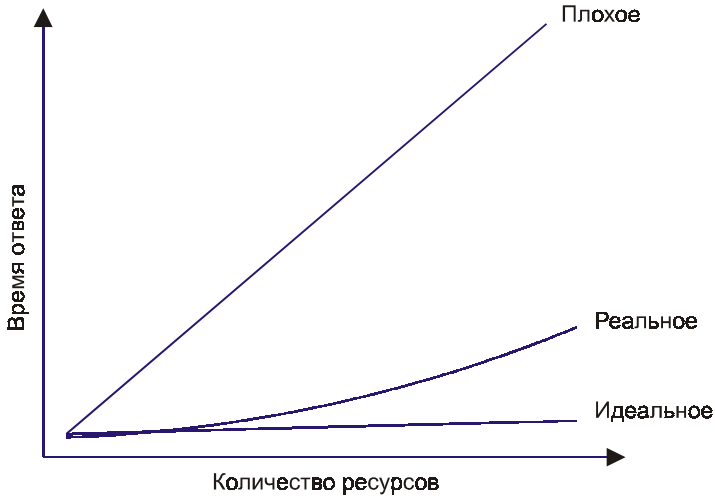


Рис. 9.3. Масштабируемость по времени ответа

от объемов хуже, чем линейно (скажем, квадратично или еще хуже). В подобных случаях необходимо использование методов, рассмотренных в предыдущих главах, или, если алгоритмы высокой вычислительной сложности необходимы по существу решаемых задач, полное перепроектирование системы, начиная с пересмотра требований к ней.

9.2. Уровни параллелизма

Параллелизм используется различными компонентами прикладной системы или системного программного обеспечения. Для наших целей можно выделить следующие варианты организации параллельного выполнения.

- Параллелизм между транзакциями.

  Например, параллельно могут выполняться несколько копий приложения для того, чтобы увеличить пропускную способность системы или ускорить выполнение каких-либо длительных вычислений.

- Параллелизм между запросами в рамках одной транзакции.

  Например, операции модификации базы данных могут выполняться параллельно с последующими операциями выборки данных, выполняемыми в той же транзакции.

- Параллелизм между операциями внутри одного запроса к СУБД.

  Например, выборка из разных таблиц для последующего соединения может выполняться параллельно (при условии, что это допускается конфигурацией вычислительной системы).

- Параллельное выполнение в пределах одной алгебраической операции.

  Например, выборка из таблицы, размещенной на нескольких устройствах, может выполняться параллельно.

Наиболее важными являются межтранзакционный параллелизм и параллелизм внутри операций (т. е. первый и последний в этом списке). Причина состоит в том, что современные высокопроизводительные СУБД, как правило, автоматически используют средства параллельной обработки в тех случаях, когда это позволяет конфигурация оборудования (тогда как низкопроизводительные системы совсем не используют параллелизм).

Разработчики и администраторы прикладной системы практически не имеют возможностей влиять на использование параллелизма между запросами внутри транзакции и между операциями внутри запроса. Если использование этих видов параллелизма возможно, то СУБД сделает это автоматически.

С другой стороны, как организация параллелизма между тарнзакциями, так и создание условий для применения параллельных версий алгоритмов алгебраических операций требуют определенных усилий при проектировании конфигурации прикладной системы и разработке схемы базы данных.

Выбор уровня, на котором реализуется параллелизм, и конфигурации системы зависит от того, как распределена вычислительная нагрузка между компонентами системы, в частности, между программным кодом, выполняемым на сервере базы данных, и кодом приложения.

Многие современные приложения выполняют вычислительно сложную обработку данных. Например, если для обмена данными между компонентами внутри системы используется язык XML, как правило, затраты процессорного времени на генерацию сообщений и их последующий синтаксический разбор составляют значительную часть общего времени работы приложения. При этом нагрузка на сервер базы данных может оказаться относительно небольшой, особенно если приложение использует исключительно короткие запросы. В таких случаях, скорее всего, целесообразно применять конфигурацию системы, состоящую из нескольких серверов приложений с общим сервером базы данных. В некоторых случаях альтернативой является перенос части функций приложения на сервер базы данных с использованием объектных расширений или средств генерации XML, предоставляемых СУБД.

Использование такой конфигурации может обеспечить масштабируемость по пропускной способности, поскольку запросы разных пользователей могут направляться на разные серверы приложений, и косвенно может обеспечить масштабируемость по времени ответа, так как наличие нескольких серверов потенциально предотвращает увеличение размера очереди запросов.

Несомненным достоинством подобной конфигурации является простота расширения. Для того чтобы включить в конфигурацию дополнительные серверы приложений, не требуется что-либо изменять ни в коде приложения, ни в схеме базы данных. Это достоинство успешно используется создателями высокопроизводительных серверов.

Возможности наращивания этой конфигурации ограничиваются тем, что при увеличении количества серверов приложений рано или поздно узким местом в системе становится сервер базы данных. Для того чтобы уменьшить влияние этого фактора, высокопроизводительные серверы приложений поддерживают кэш базы данных. В подавляющем большинстве случаев приложение использует базу данных почти исключительно для чтения данных, поэтому кэш большого объема оказывается вполне эффективным.

Примерно тот же механизм может успешно применяться и в рамках одного сервера приложений при условии, что этот сервер может использовать многопроцессорные конфигурации оборудования или хотя бы псевдопараллелизм.

Важно учитывать то, что полезность такого вида параллелизма существенно зависит от распределения функций между компонентами приложения. Например, если для генерации того же XML используются средства, предоставляемые современными версиями всех высокопроизводительных СУБД, а не код, работающий на сервере приложений, то существенная часть нагрузки способна сместиться на СУБД, и в этом случае параллелизм на уровне сервра приложений не будет эффективен. К такому же результату может привести использование хранимых процедур, реализующих часть функциональности приложения на сервере базы данных.

Необходимо также отметить, что многое из вышеизложенного применимо к конфигурациям, в которых сервер приложений не используется, и приложение работает непосредственно с базой данных.

Обсудим теперь, каким образом междутранзакционный параллелизм можно использовать для достижения ускорения.

Очевидно, нас должны интересовать задачи, для которых время ответа велико, иначе нет необходимости в ускорении. По-видимому, в такой задаче обрабатывается большое количество данных, т. е. либо выполняются длинные запросы, либо приложение выполняет большое количество относительно небольших запросов итеративно. В том и другом случае очевидное решение состоит в том, чтобы разбить весь объем данных, подлежащих обработке, на части, которые можно обрабатывать независимо, и выполнять обработку этих частей в независимых транзакциях (не имеет значения, к одному экземпляру относятся приложения или к разным).

Далее, для того, чтобы ускорение было достигнуто, необходимо организовать вычисления таким образом, чтобы степень параллелизма при одновре-

менном выполнении нескольких фрагментов работы была больше, чем при их последовательном выполнении. Это может быть достигнуто в следующих случаях.

- Значительная часть необходимой вычислительной мощности используется в коде приложения.

  По существу этот случай совпадает с рассмотренным ранее: для того, чтобы получить ускорение, необходимо запустить несколько копий приложения на параллельно работающих вычислительных системах, и ускорение будет достигаться до тех пор, пока узким местом не станет сервер базы данных.

- Ускорение достигается за счет более эффективного использования мощности сервера базы данных.

  Увеличение нагрузки на сервер базы данных может приводить к ускорению за счет лучшего совмещения операций ввода-вывода и операций центрального процессора. Конечно, этот источник ускорения работает только до тех пор, пока производительность сервера базы данных ограничена операциями ввода-вывода.

- Сервер базы данных имеет возможности параллельной обработки, но они не могут использоваться в рамках одной транзакции (для этого приложения).

  Типичный случай, когда такая ситуация возникает, связан с применением пользовательских хранимых процедур. Каждая процедура, как правило, выполняется в рамках одной нити, что практически исключает возможность ее выполнения на нескольких процессорах, даже если они имеются в вычислительной системе.

Описанный в последнем пункте эффект может быть существенным в тех случаях, когда хранимые процедуры являются вычислительно сложными и содержат большое количество небольших запросов.

Здесь важно напомнить, что в наиболее удачном случае применение n процессоров, так же, как простое увеличение частоты процессора, никак не может привести к улучшению характеристик системы более чем в n раз. Если алгоритмы обработки зависят от объемов данных хуже, чем линейно, хорошую масштабируемость получить невозможно.

В связи с этим уместно вернуться к примеру, рассмотренному в *разд. 4.3.3*, в котором применение более эффективного алгоритма привело к снижению времени выполнения в сотни раз, что было бы невозможно получить за счет параллелизма даже при идеальной масштабируемости системы. В реальной ситуации, которая стоит за этим примером, технические возможности для распараллеливания присутствовали, однако было очевидно, что они недостаточны для того, чтобы завершить работу к необходимому директивному сроку.

Для больших запросов, полностью записанных в виде одного оператора SQL, более эффективным может оказаться использование внутриоперационного параллелизма. Для того чтобы его можно было применить, необходимо соответствующим образом изменить схему базы данных.

9.3. Размещение данных

Будем далее предполагать, что сервер базы данных может использовать несколько процессоров и несколько дисковых устройств, способных работать параллельно. В терминологии конкретных СУБД это может означать, что система состоит из нескольких взаимодействующих серверов, однако для приложения они выглядят как единый сервер базы данных, принимающий и выполняющий запросы. Другими словами, параллелизм сервера базы данных полностью прозрачен для приложения, и, следовательно, код приложения не зависит от того, является ли сервер базы данных параллельным.

Алгоритмы, используемые в параллельных СУБД, конечно, существенно зависят от конфигурации аппаратуры, и в первую очередь от возможностей совместного использования памяти и дисков разными процессорами. Однако эти особенности не очень существенны для проектирования и настройки схемы базы данных.

Напомним, что алгебраические операции, выполняемые сервером базы данных, делятся на две группы:

- операции чтения и модификации хранимых данных (такие операции находятся в листьях плана выполнения запроса);
- операции над потоками данных (такие операции находятся в промежуточных вершинах и корне дерева, описывающего план выполнения запроса).

Очевидно, что при обработке хранимых данных время выполнения операций в значительной мере определяется временем работы дисков, поэтому для того, чтобы эти операции могли получать какой-либо выигрыш от параллелизма, необходимо обеспечить распараллеливание именно работы дисков.

С другой стороны, при обработке потоков диски используются только для промежуточного хранения, если не хватает оперативной памяти, поэтому эти операции могут получать выигрыш только в том случае, если для их выполнения несколько процессоров используются параллельно. Мы кратко рассмотрим алгоритмы параллельного выполнения основных операций в *разд. 9.4.*

Очевидно, что для того, чтобы дисковые устройства могли работать одновременно, необходимо разместить данные на разных устройствах. Степень

параллелизма для операций, работающих с хранимыми данными, определяется тем, как эти данные распределены между устройствами.

Такое распределение данных принято называть декластеризацией. Известно довольно много различных способов декластеризации, не все они могут быть легко реализованы в коммерческих системах. Декластеризацию можно реализовать как на уровне логических объектов базы данных (т. е. таблиц или строк таблиц), так и на уровне блоков или файлов, в которых логические структуры хранятся.

Выбор варианта декластеризации определяется тем, какие характеристики системы требуется улучшить свойствами наиболее часто выполняемых или наиболее важных запросов.

Если целью распараллеливания является увеличение пропускной способности, то данные могут быть размещены таким образом, чтобы при выполнении разных операций или разных запросов, причем не обязательно от одной и той же копии приложения, использовались данные, размещенные на разных устройствах. Иначе говоря, для увеличения пропускной способности системы, скорее всего, целесообразно использовать параллелизм между операциями.

Если же целью является ускорение, то, очевидно, параллельное выполнение различных запросов никак не поможет достижению цели. В этих случаях необходимо размещать данные таким образом, чтобы при выполнении каждого запроса было использовано как можно больше устройств (в надежде, что при этом каждое из устройств будет выполнять меньший объем работы).

Если же целью является масштабируемость по времени ответа, то наилучший способ декластеризации зависит от характера выполняемых запросов. Если при увеличении объема базы данных объем данных, обрабатываемых каждым запросом, не возрастает, то, скорее всего, время ответа может сохраняться на приемлемом уровне за счет увеличения количества параллельно выполняемых запросов. Иначе, если объем данных, обрабатываемых каждым запросом, растет, то может понадобиться такая же декластеризация, как для достижения ускорения.

Вопрос о том, какой способ декластеризации использовать, решается при проектировании структуры хранения отдельно для каждой таблицы. (Из этого, очевидно, следует, что проектирование параллельной базы данных существенно сложнее по трудозатратам, чем обычное проектирование структуры хранения.)

Как правило, декластеризацию целесообразно применять только для больших таблиц, потому что таблицы малого размера обычно целиком помещаются в кэш базы данных и поэтому доступ к ним не требует реальной работы дисковых устройств.

Мы будем далее рассматривать только декластеризацию на уровне логических объектов, а именно — строк таблиц.

Для того чтобы описать декластеризацию, или разбиение таблицы, создаются разделы этой таблицы на каждом из устройств, которое будет использоваться для ее хранения. Конечно, в реальности таблицы размещаются в файлах операционной системы, а не непосредственно на устройствах, но пока это для нас несущественно.

Для каждого раздела таблицы задается предикат, определяющий, при каком условии строка таблицы размещается именно в этом разделе. Совокупность этих предикатов должна задавать разбиение пространства всех возможных значений строк таблицы. Напомним, что разбиением множества называется совокупность его непересекающихся подмножеств, объединение которых составляет все множество. Здесь мы использовали термин "разбиение" именно в этом точном смысле.

Для реальной проверки это условие целесообразно переформулировать таким образом: дизъюнкция всех предикатов разбиения должна быть тождественно истинной, а попарные конъюнкции этих предикатов — тождественно ложными.

Важное практическое требование состоит в том, что эти предикаты должны быть легко проверяемыми. Обычно промышленные СУБД допускают только разбиения по диапазону значений некоторого атрибута или по значению функции хеширования.

Проиллюстрируем выбор метода декластеризации на примере.

Предположим, что наиболее часто выполняемые короткие запросы выбирают строки из таблицы `assignments`, относящиеся только к одному служащему.

Если эта таблица будет декластеризована по атрибуту `employee_id`, то при выполнении каждого запроса будет использоваться только одно дисковое устройство, а для разных запросов будут, вообще говоря, использованы разные устройства. Очевидно, такая декластеризация может увеличить пропускную способность системы, но не может улучшить время ответа. Поэтому можно ожидать, что масштабируемость по пропускной способности будет удовлетворительной, ускорение будет плохим, а поведение масштабируемости по времени ответа будет зависеть от степени загруженности системы.

Если же таблица `assignments` декластеризована по другому атрибуту, скажем, по времени начала работы в должности, то при выполнении каждого запроса будут задействованы, вообще говоря, несколько устройств, и можно ожидать, что произойдет ускорение выполнения запросов.

Если при выполнении длинных запросов выводится большой объем данных, то для ускорения важно, в каком порядке выполняется обработка результата

запроса. Если последовательность, в которой строки выбираются приложением, совпадает с упорядоченностью по атрибуту, по которому таблица декластеризована, то эффект от использования параллелизма будет незначителен, потому что сначала будут выбраны все записи с первого устройства, затем — со второго и т. д.

Для того чтобы предотвратить подобные эффекты, используется механизм чередования, состоящий в том, что строки, имеющие соседние по упорядоченности значения атрибута декластеризации, размещаются на разных устройствах. Например, при наличии D дисковых устройств первая строка размещается на первом, вторая — на втором, D-я на устройстве D, а $(D + 1)$-я строка — снова на устройстве 1 и так далее по кругу.

Конечно, непосредственная реализация такого метода невозможна, поскольку любое обновление данных приводило бы к необходимости перемещения значительной части данных. Практически приемлемый вариант можно получить, декластеризуя данные, например, по младшим разрядам атрибута, а не по старшим, как при обычном упорядочивании.

Аналогичные рассмотрения применимы и по отношению к индексам: в зависимости от того, какие из характеристик масштабируемости наиболее важны, можно строить различные варианты индексов: отдельно для каждого раздела декластеризованных таблиц или общие. Сами индексы также могут быть декластеризованными.

В качестве примера рассмотрим задачу вычисления балансов, описанную в *разд. 7.3.1.* Поскольку вычисление балансов занимает значительное время, желательно как-нибудь использовать возможности параллелизма для улучшения времени ответа, т. е. для ускорения. Поскольку при выполнении наиболее важного запроса производится считывание значительной части хранимых данных, целесообразно использовать декластеризацию для самых больших таблиц, задействованных в этом запросе, т. е. для таблиц `run_results` и `assignment_actions`.

Поскольку после считывания выполняется операция соединения этих таблиц, целесообразно декластеризовать их таким образом, чтобы эта операция соединения могла быть выполнена без пересылки данных этих таблиц. Поскольку ключом соединения является атрибут `assignment_action_id`, общий для этих таблиц, следует декластеризовать их именно по этому атрибуту.

Разбиение по диапазонам значений будет в этом случае эквивалентно разбиению на порции для обработки, как это сделано в *разд. 7.3.5.* Отличие, конечно, состоит в том, что на параллельных системах обработка этих порций будет выполняться параллельно в рамках одной операции соединения.

Такая декластеризация, однако, имеет существенный недостаток. Дело в том, что значения суррогатных ключей, каковыми являются `assignment_id`, скорее всего, представляют собой возрастающую последовательность целых

чисел, поэтому при росте объемов таблиц будет требоваться частая реорганизация с практически полным перемещением данных этих таблиц. Данная проблема не возникает при разбиении на порции, так как, во-первых, размеры порций и их количество можно задать непосредственно перед выполнением вычислений, когда размеры таблиц уже не могут значительно измениться, и, во-вторых, поскольку размеры порций могут быть различными, никакое выравнивание нагрузки не требуется.

Более удачным решением является декластеризация по тому же атрибуту, но с помощью функции хеширования. Даже простейшая функция хеширования, а именно вычисление остатка от деления `assignment_action_id` на число параллельных подсистем, даст вполне удовлетворительные результаты.

9.4. Параллельные версии основных алгоритмов

Напомним, что все вычислительно сложные операции реляционной алгебры (произведение, соединение, группировка и устранение дубликатов) могут быть реализованы с помощью одного или нескольких из следующих трех алгоритмов:

- алгоритм вложенных циклов;
- алгоритм слияния упорядоченных потоков данных.

Замечательный факт состоит в том, что все эти алгоритмы имеют параллельные версии, обладающие хорошими характеристиками масштабируемости.

Поскольку операции, рассматриваемые в данном разделе, никогда не находятся в листьях дерева, представляющего план выполнения запроса, то и входные операнды, и результаты представляются как потоки данных. В связи с тем, что параллельные версии алгоритмов должны использовать несколько процессоров, обычно первая фаза этих алгоритмов выполняет распределение входного потока между процессорами, участвующими в выполнении операции.

Некоторые алгоритмы вырабатывают результат также в виде нескольких потоков. Если этот результат не является окончательным, то фаза распределения для следующей операции может выполняться параллельно (независимо для каждого из потоков) или, если следующая операция выполняется на том же наборе процессоров, фаза распределения может быть исключена.

9.4.1. Параллельный алгоритм вложенных циклов

Алгоритм вложенных циклов состоит из двух фаз.

На фазе распределения строки (или блоки) операнда меньшего размера распределяются между процессорами, участвующими в выполнении операции

(примерно поровну), при этом каждая строка направляется только на один процессор.

На фазе соединения каждая строка (или блок) второго операнда направляется на все процессоры, участвующие в выполнении операции. Получив строку входного операнда, эти процессоры реализуют обычный алгоритм вложенных циклов и выводят полученные строки результата в выходные потоки.

Как и в однопроцессорном варианте, при некоторых условиях возможны упрощенные версии этого алгоритма, обладающие меньшей вычислительной сложностью.

Так, если один из операндов получается непосредственной выборкой из таблицы, декластеризованной по атрибуту соединения, и для выполнения операции выделены те процессоры, по которым декластеризован данный операнд, то фаза распределения может быть исключена, а алгоритм вложенных циклов на фазе соединения может выполняться с использованием хранимых индексов, т. е. фактически цикл будет вырожденным.

Важно отметить, что выработка результата происходит одновременно с выполнением фазы соединения, т. е. результирующий поток может сразу направляться на вход следующей операции (возможно, выполняемой параллельно с рассматриваемой).

9.4.2. Параллельный алгоритм сортировки-слияния

На фазе распределения первый операнд декластеризуется по атрибуту соединения: каждому из процессоров выделяется диапазон значений атрибута соединения и распределение строк первого операнда, затем каждый из процессоров выполняет сортировку.

Далее таким же образом выполняются распределение и сортировка второго операнда с использованием тех же диапазонов значений ключа соединения.

После этого на фазе соединения выполняется слияние упорядоченных сегментов с одновременным выводом нескольких упорядоченных потоков. Если эти потоки являются окончательным результатом, то выполняется их слияние в один поток. Конечно, один упорядоченный поток не может быть получен на нескольких процессорах, однако этот процессор может работать параллельно с процессорами, вырабатывающими для него упорядоченные входные потоки.

9.4.3. Параллельный алгоритм хеширования

По существу параллельная версия этого алгоритма отличается от однопроцессорной только тем, что блоки, в которые хешируется первый операнд, обрабатываются разными процессорами.

На фазе распределения первый операнд распределяется между процессорами с помощью функции хеширования, зависящей от атрибута соединения и вырабатывающей номер процессора, на который направляется строка входного операнда.

На фазе соединения второй операнд распределяется с помощью той же самой функции хеширования, при этом одновременно выполняется операция соединения, и вырабатываются результирующие потоки (на каждом из процессоров, участвующих в операции).

9.5. Параллелизм между операциями

Поскольку передача данных между операциями при выполнении запроса использует механизм потоков данных, несколько операций могут выполняться параллельно при условии, что для этих операций выделены разные процессоры (или наборы процессоров, если каждая из операций тоже выполняется параллельно).

Возможность такого параллелизма ограничивается в первую очередь тем, что многие операции не могут начать выработку результата до тех пор, пока не выполнены определенные фазы их алгоритмов. Мы уже упоминали о том, что никакой алгоритм операции сортировки не может начать вырабатывать результат до тех пор, пока не будет завершено получение входного операнда.

Степень параллелизма между операциями может зависеть от структуры плана выполнения запроса. Для иллюстрации этого утверждения рассмотрим запрос, в котором требуется выполнить соединение нескольких таблиц:

```
R1  join R2  join R3  join R4 join R5  join  R6
```

Поскольку операция соединения ассоциативна, различные способы расстановки скобок в этом выражении эквивалентны. Предположим, что для всех операций соединения используется алгоритм на основе хеширования, и рассмотрим следующий план выполнения этого запроса:

```
HASH JOIN(
   HASH JOIN(
      HASH JOIN(
         HASH JOIN(
            HASH JOIN(R1,R2),
            R3),
         R4),
      R5),
   R6)
```

В этом плане первый операнд каждой операции соединения, кроме первой, является результатом предыдущей операции соединения, т. е. в исходном выражении вычисление выполняется слева направо. Каждая из операций соединения осуществляется в соответствии с алгоритмом в две фазы: сначала распределяется первый операнд и затем обрабатывается второй, при этом во время выполнения второй фазы происходит вывод результата.

Фаза распределения следующей операции может выполняться одновременно с фазой соединения предыдущей. Общее время выполнения всего запроса будет, следовательно, примерно равно сумме времен, необходимых для выполнения фаз распределения, плюс последняя фаза соединения, вырабатывающая окончательный результат.

Алгоритм на основе хеширования несимметричен, т. е. первый и второй операнды обрабатываются по-разному. Поэтому другая структура плана может привести к существенно отличающимся результатам.

Рассмотрим теперь другой вариант плана выполнения того же запроса:

```
HASH JOIN(R1,
   HASH JOIN(R2,
      HASH JOIN(R3,
         HASH JOIN(R4,
            HASH JOIN(R5,
               R6)
            )
         )
      )
   )
```

При выполнении такого плана фазы распределения для всех операций используют хранимые таблицы, поэтому фазы распределения могут выполняться параллельно сразу для всех операций соединения. После того как все фазы распределения закончены, результаты фазы соединения могут немедленно передаваться на следующую операцию, т. е. все фазы соединения также могут выполняться параллельно (при условии, конечно, что для всех операций выделены разные наборы процессоров). Общее время выполнения запроса будет примерно равно максимальному времени фазы распределения плюс максимальное время фазы соединения.

Таким образом, выполнение операций справа налево дает возможность для очень существенного увеличения степени параллелизма. (Конечно, направление справа налево получается только потому, что операнд, используемый на фазе распределения, мы записываем слева.)

В реальности количество процессоров всегда ограничено, и поэтому оптимизатор должен выбирать, какое количество ресурсов выделить для каждой

операции и какая степень параллелизма между операциями оптимальна. Далеко не всегда высокая степень параллелизма между операциями обеспечивает наилучшие результаты для системы в целом.

9.6. Выравнивание нагрузки

Очевидно, что хорошее поведение характеристик масштабируемости в параллельной системе возможно только в том случае, если загрузка всех эквивалентных ресурсов примерно одинакова (т. е. все диски загружены примерно поровну и все процессоры загружены примерно одинаково). В противоположном крайнем случае, когда вся работа выполняется, скажем, одним процессором, наличие дополнительного оборудования (других процессоров) никак не помогает улучшить характеристики системы.

Поэтому качество параллельной системы в значительной мере определяется тем, насколько хорошо в ней выравнивается нагрузка. Многое в этом направлении делается сервером базы данных автоматически, но в некоторых случаях вмешательство проектировщика или администратора базы данных все же необходимо.

Очевидно, никакой автоматический механизм не может обеспечить выравнивание нагрузки при декластеризации, если метод декластеризации определяется в схеме базы данных.

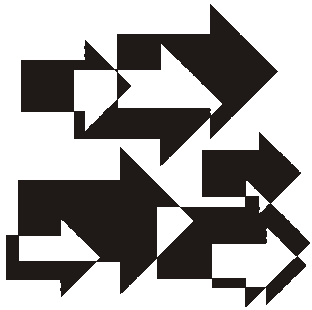
При декластеризации по диапазону значения атрибута граничные значения должны быть подобраны таким образом, чтобы распределение данных между разделами было примерно одинаковым. Этот очевидный факт, вероятно, известен всем, но в такой формулировке он неверен, потому что нужно еще учитывать распределение запросов!

Например, допустим, что выполняется декластеризация по некоторому атрибуту даты, данные распределены по датам равномерно, необходимо использовать чередование, поэтому данные размещаются, скажем, по номеру месяца вполне равномерно. Если при этом запросы наиболее часто относятся к данным последней недели, то почти всегда будет работать не более двух устройств из 12.

При выполнении операций соединения может возникать эффект *перекоса данных*. Данный эффект состоит в том, что даже при равномерном распределении входных операндов между процессорами, выполняющими операцию соединения, количество строк результата, вырабатываемых этими процессорами, будет очень существенно различаться и поэтому нагрузка на процессоры не будет равномерной.

Основная опасность эффекта перекоса данных заключается в том, что его появление трудно предсказать до тех пор, пока операция не начала выполняться.

Высокопроизводительные системы имеют средства для выравнивания нагрузки при перекосах, однако мы не будем их рассматривать.



Глава 10

Настройка сервера базы данных

В данной главе обсуждаются задачи настройки, связанные с работой сервера базы данных в целом. Как правило, действия по настройке сервера затрагивают все приложения, использующие этот сервер.

Именно эта часть работы по настройке является одной из основных функций администратора базы данных. Иногда роль администратора сводится только к этому виду настройки, но мы уже отмечали, что такое ограниченное понимание настройки является недостаточным.

Необходимость настройки сервера базы данных может возникнуть в перечисленных далее случаях.

- В приложениях имеется большое количество различных запросов, которые по каким-то причинам плохо оптимизируются, причем для их улучшения требуются примерно одинаковые действия.

  В этом случае, очевидно, целесообразно рассмотреть возможность настройки оптимизатора запросов.

- В системе выполняется очень большое количество небольших запросов. Например, эти запросы выполняются из разных копий приложения (для разных пользователей) или являются частью сложного процесса вычислений, реализуемого приложением.

- Данные неудачно размещены на носителе. Такая ситуация может возникать вследствие роста объемов хранимых данных, в особенности в тех случаях, когда тот же носитель используется и для других целей.

- Время ответа становится неудовлетворительным при увеличении нагрузки на систему из-за конфликтов обновления базы данных.

Не существует четкой границы между настройкой сервера базы данных и другими видами настройки. Во многих случаях для устранения причин, вызывающих неудовлетворительную производительность системы, требуется внести изменения на разных уровнях. Отличительная особенность ситуаций, рассматриваемых в этой главе, состоит в том, что обычно для обнаружения и диагностики причин возникновения проблем производительности требуется анализ работы всего сервера.

Обнаружить подобные проблемы при автономном запуске приложения, как правило, не удается. Некоторые виды проблем невозможно найти даже при тщательном и интенсивном тестировании.

Общепризнано, что любое вмешательство в работу системы, находящейся в режиме реальной эксплуатации, является чрезвычайно дорогостоящим. Поэтому как владельцы баз данных, так и производители систем управления базами данных стремятся свести к минимуму необходимость такого вмешательства.

Значительная часть усилий по совершенствованию систем управления базами данных, в особенности высокопроизводительных, в последние годы была направлена на разработку средств автоматической настройки серверов баз данных. В результате этих улучшений, а также в связи с ростом размеров оперативной памяти и мощности процессоров необходимость в ручной настройке серверов значительно сократилась и во многих случаях не требуется совсем.

В результате сложность управления высокопроизводительными серверами существенно снизилась. Однако здесь необходимо подчеркнуть заметное отличие дорогостоящих высокопроизводительных систем от недорогих, которые также очень просты в управлении: высокопроизводительные системы имеют большое количество различных механизмов и средства их автоматической настройки, над которыми надстроены средства автоматического конфигурирования, а недорогие обычно предлагают только один вариант конфигурации.

Как средства управления сервером базы данных, так и средства анализа производительности не регулируются какими-либо стандартами и поэтому реализуются совершенно различными способами в разных СУБД. Несмотря на это, мы будем по возможности избегать привязки к особенностям отдельных СУБД, т. е. в основном рассмотрим проблемы и методы их решения, а не инструменты или параметры, управляющие поведением компонентов СУБД.

Мы уже отметили ранее, что настройка сервера базы данных должна выполняться в комплексе с настройкой приложений. С другой стороны, настройка сервера существенно зависит от конфигурации оборудования, конфигурации и настройки операционной системы, а также от наличия или отсутствия других задач, решаемых на том же оборудовании. Тесное взаи-

модействие с администратором операционной системы не менее важно для настройки сервера базы данных, чем взаимодействие с разработчиками и администраторами приложений.

Еще одна важная особенность настройки сервера состоит в том, что довольно часто улучшение характеристик поведения системы для одного из приложений или класса запросов может приводить к ухудшению характеристик других приложений или классов запросов. По этой причине обычно при настройке сервера требуются компромиссные решения.

Перейдем теперь к обсуждению основных видов настройки сервера базы данных. Прежде чем предпринимать какие-либо действия по настройке, необходимо установить, какие причины вызывают проблемы производительности.

10.1. Диагностика

Как правило, поводом для того, чтобы начинать настройку сервера базы данных, становятся жалобы пользователей на неприемлемо большое время ответа, и часто разработчики (или администраторы сервера приложений) утверждают, что все проблемы связаны с неудовлетворительной работой сервера базы данных. Первое из этих утверждений обычно оказывается справедливым, но для выяснения того, насколько обоснованно второе, требуется серьезный анализ поведения всех компонентов прикладной системы.

Процесс диагностики начинается с выяснения основных характеристик производительности и загруженности для системы в целом и отдельных ее компонентов, в том числе динамики их поведения.

Прежде всего, необходимо установить, каков характер проблем производительности, наблюдаемых пользователем, ответив на следующие вопросы.

- Какие действия (т. е. запросы или транзакции) выполняются неоправданно долго?
- Является ли неудовлетворительное время ответа стабильным или неожиданные задержки возникают только иногда?
- Зависит ли поведение системы от времени суток или других факторов, которые могут вызывать пиковые нагрузки?

Далее необходимо проанализировать нагрузку на различные компоненты системы. Наиболее важными характеристиками, полезными для диагностики, могут быть следующие:

- соотношение времени ответа, видимого пользователем, с временами ответа на различных уровнях системы, например, на сервере приложений и на сервере базы данных;
- степень загруженности процессора на сервере приложений и на сервере базы данных;

- соотношение количества запросов пользователя и количества запросов к базе данных, выполняемых приложением;
- загруженность устройств ввода-вывода на сервере баз данных;
- загруженность сети.

Важно подчеркнуть, что не существует однозначных зависимостей между значениями и поведением этих интегральных характеристик с одной стороны и реальными причинами низкой производительности системы с другой.

Например, низкая загруженность процессора на сервере базы данных по сравнению с сервером приложений может указывать на необходимость увеличения мощности сервера приложений или на проблемы с управлением транзакциями на сервере базы данных.

Большое время ожидания ответа СУБД может указывать как на необходимость настройки оптимизатора, так и на слишком низкую гранулярность запросов, выполняемых приложением. Высокая загруженность ввода-вывода на сервере базы данных может указывать на необходимость настройки оптимизатора, на плохое размещение данных или необходимость создания дополнительных индексов.

Даже 100%-ная загруженность процессора далеко не всегда означает, что покупка более высокопроизводительного оборудования может решить проблему! Во многих случаях настройка оптимизатора может дать более радикальные улучшения.

Диагностика на уровне сервера базы данных особенно важна для систем с высокой интенсивностью коротких запросов (т. е. классических систем оперативной обработки транзакций — OLTP). Дело в том, что время выполнения коротких запросов может быть сопоставимо с разрешением предоставляемых системой средств измерения времени, поэтому получение надежных результатов при настройке на уровне приложения оказывается затрудненным. В таких системах важной характеристикой становится не время ответа, а суммарное время использования ресурсов сервера, потраченное на выполнение запросов определенного вида.

Перечисленные выше характеристики дают возможность установить, какие части системы нуждаются в дополнительной настройке, однако для более точного анализа проблемы обычно необходима дополнительная информация. Получить такую информацию в принципе несложно, любая современная система предоставляет достаточно мощные средства слежения за поведением системы. Проблема состоит в том, чтобы организовать сбор только той информации, которая может быть полезна для решения проблемы.

Важно организовать процесс анализа работы системы так, чтобы ее поведение не изменилось. Любые действия по сбору информации используют некоторую часть ресурсов системы, и если эта часть существенна, поведение системы будет искажено процессом сбора данных. Поэтому выбор дополни-

тельных характеристик и определение методики их сбора составляет значительную часть работы по анализу и решению проблем производительности на уровне сервера.

10.2. Управление оптимизатором

Настройка оптимизатора запросов может быть полезна в тех случаях, когда неудовлетворительное время ответа СУБД для большинства запросов наблюдается в сочетании с высокой загруженностью внешних устройств или процессора сервера базы данных. Цель настройки состоит в том, чтобы уменьшить нагрузку на систему и улучшить время ответа за счет построения более эффективных планов для (почти) всех запросов, выполняемых прикладной системой. Как правило, подобную настройку можно выполнять и на уровне отдельных запросов. Это дает возможность применять различные стратегии сочетания настройки сервера и приложения. Например, после начальной диагностики на уровне сервера можно выполнить настройку отдельных, наиболее важных запросов, а затем, если необходимо, выполнить аналогичные действия на уровне сервера.

Управление работой оптимизатора — классическая тема, включенная во многие руководства по системам управления базами данных.

Необходимость управления оптимизатором объясняется следующими причинами.

- Само понятие оптимального плана выполнения запроса зависит от того, каким образом определены критерии производительности. Вообще говоря, оптимальный план выполнения запроса может быть разным для разных критериев.
- Время работы оптимизатора ограничено, поэтому обычно оптимизатор не может выработать оптимальный план, поэтому качество получаемого плана зависит от стратегии перебора.
- В состав СУБД могут быть включены различные типы оптимизаторов.

Алгоритмы, используемые в оптимизаторах запросов, весьма сложны и поэтому производители СУБД обычно не обеспечивают средства для непосредственного вмешательства в их работу. Вместо этого обычно предоставляются параметры, дающие возможность косвенно влиять на работу оптимизатора.

Например, хорошо известная альтернатива между наиболее быстрым получением всего результата и наиболее быстрым получением первых строк результата на самом деле сводится к выбору предпочтительного алгоритма для операции соединения между алгоритмом на основе хеширования и алгоритмом вложенных циклов соответственно. Такой способ управления оптимизатором используется в системе IBM DB2. Другие производители дают воз-

можность аналогичного выбора, однако управление этим выбором делается совсем в других терминах.

В связи с этим заметим, что в современных архитектурах приложений получение первых строк результата обычно не имеет смысла, потому что ответ, выводимый на терминал конечного пользователя, формируется, как правило, на основе полного результата выполнения запроса, однако выбор метода выполнения все равно важен, потому что выбор алгоритма для операций соединения по-прежнему имеет значение.

Другой очень важный аспект работы оптимизатора связан с выбором индексов для выполнения запроса. В *главе 6* было показано, как управлять использованием индексов для отдельных запросов. Мы уже не раз упоминали общее правило, которое состоит в том, что индексы целесообразно использовать в том случае, если количество строк, выбираемых в запросе, невелико по сравнению с количеством строк в таблице. В таком неформальном виде, конечно, это правило не может использоваться оптимизатором, и фактически поведение оптимизатора зависит от ожидаемых значений селективности запроса. Некоторые СУБД предоставляют возможность управления граничными значениями селективности или изменять относительную стоимость использования индексов по сравнению с полным просмотром.

Ранее в данной книге уже отмечалось различие между основными критериями производительности: пропускной способностью и временем ответа. Это различие, безусловно, важно для проектирования системы в целом, однако на уровне оптимизатора запросов оно не имеет столь большого значения. Дело в том, что промышленные оптимизаторы всегда выполняют поиск оптимального плана для одного запроса, т. е. в первую очередь оптимизируют время ответа.

Групповая (одновременная) оптимизация нескольких запросов представляет собой значительно более трудную задачу, как вычислительно, так и по сложности реализации. Методы групповой оптимизации рассматриваются иногда в научной литературе, но пока они непригодны для применения в промышленных системах.

Критерии пропускной способности и времени ответа, хотя и различны, не являются вполне независимыми. Плохие по времени ответа планы увеличивают нагрузку на систему и, следовательно, снижают пропускную способность. Различие по затратам вычислительных ресурсов между плохими и хорошими планами обычно бывает очень велико (в десятки, сотни и более раз), это различие значительно больше, чем различие между планами, оптимальными по разным критериям.

На этих наблюдениях основана работа адаптивных средств автоматической настройки.

Целью такой настройки является сокращение времени выполнения для часто выполняемых запросов. Хотя в принципе СУБД может выполнить любой

запрос на языке SQL, количество различных запросов, фактически выполняемых приложением, ограничено (во всяком случае, количество часто выполняемых запросов относительно невелико). Для каждого из таких запросов система накапливает информацию о времени их выполнения.

Запросы, на выполнение которых потрачено наибольшее суммарное количество ресурсов системы, подвергаются дополнительной оптимизации. При этом на работу оптимизатора отводится значительно большее время. В результате вырабатывается план лучшего качества, и время, затрачиваемое на выполнение данного запроса, сокращается, что приводит к снижению как общей загруженности системы, т. е. к увеличению пропускной способности, так и к улучшению времени ответа для этого запроса.

Еще один важный аспект, связанный с управлением оптимизатором, касается размеров временных областей дисковой памяти, выделенной для выполнения запросов. В некоторых случаях наиболее эффективные по времени алгоритмы используют значительно большее количество (внешней) памяти, чем другие. В частности, алгоритм соединения на основе хеширования может использовать временную память, сопоставимую с размерами хранимых таблиц (или подмножеств строк таблиц), участвующих в операции соединения. В подобных случаях стандартные размеры временных областей диска могут оказаться недостаточными, в результате чего может потребоваться вмешательство администратора базы данных для диагностики и расширения области временных данных. Далеко не всегда нехватка временной области памяти означает, что выполняется непреднамеренное вычисление декартова произведения больших таблиц.

10.3. Управление памятью и процессами

Настройка использования оперативной памяти и процессов на сервере базы данных может быть полезна в тех случаях, когда неудовлетворительное время ответа на уровне сервера базы данных сочетается с низкой загрузкой процессора и высокой загрузкой систем ввода-вывода. Эти же симптомы, однако, могут указывать и на слишком низкую гранулярность запросов, выполняемых приложением. В последним случае настройка только на уровне сервера, очевидно, способна дать только ограниченные результаты.

Отметим, что изменение параметров распределения оперативной памяти может косвенно повлиять и на работу алгоритмов выполнения алгебраических операций, и даже на работу оптимизатора запросов.

Распределение оперативной памяти традиционно составляло одну из важных функций администратора базы данных, потому что производительность СУБД в значительной мере зависит от того, каким образом используется оперативная память. Это особенно важно в тех случаях, когда сервер баз данных выполняет очень большое количество небольших запросов.

Во время работы СУБД в оперативной памяти размещаются:

- данные, прочитанные с диска или ожидающие записи на диск (буферный пул, кэш базы данных);
- данные, описывающие состояние сеансов работы приложений;
- информация о выполненных обновлениях, предназначенная для записи в журнал и/или для отката изменений;
- временные данные, необходимые для выполнения запросов (например, рабочие области сортировки или хеширования);
- данные системы управления транзакциями (например, очереди замков);
- результаты выполнения запросов, еще не переданные клиенту (курсоры);
- возможно, другие данные.

Практически все современные СУБД выполняют распределение памяти автоматически, поэтому вмешательство администратора, как правило, не требуется. Более точно, если доступный объем оперативной памяти недостаточен, обычно перераспределение, даже если оно возможно, мало помогает решению проблем производительности системы. Расширение объема памяти компьютера часто дает значительно лучшие результаты.

Тем не менее, рассмотрим несколько ситуаций, в которых могут возникать проблемы, связанные с использованием оперативной памяти.

10.3.1. Несколько экземпляров БД

Наиболее важным параметром, определяющим использование оперативной памяти, является общий объем, выделяемый на нужды данного экземпляра СУБД. Все остальные параметры, очевидно, определяются в зависимости от размера доступной памяти.

При этом работа компонентов СУБД основывается на предположении, что предоставленная СУБД память является оперативной. В реальности, однако, эта память выделяется операционной системой и практически всегда является виртуальной. Как известно, количество виртуальной памяти, которое способна выделить операционная система, может значительно превышать объем реальной оперативной памяти. Страницы виртуальной памяти могут вытесняться на диск и при обращении к ним считываться в память, вытесняя другие виртуальные страницы.

СУБД, тем не менее, планирует использование памяти, исходя из предположения, что выделенная для ее работы память (почти) полностью является оперативной. Эффективность выполнения запросов, как мы отмечали, зависит от количества обменов с диском. Дополнительные обмены, вносимые механизмами управления виртуальной памятью, не могут быть учтены при

выполнении запросов, поэтому в тех случаях, когда виртуальная оперативная память существенно меньше реальной, фактическая производительность СУБД может значительно уменьшиться.

Обычно подобные проблемы не возникают, если на компьютере работает только один экземпляр СУБД и нет других крупных потребителей оперативной памяти (или если нагрузка на СУБД невелика).

Однако в случае, когда на одном компьютере должны работать несколько экземпляров одной или разных СУБД, необходимо позаботиться о том, чтобы активно используемые экземпляры могли разместить свои рабочие области в реальной памяти (т. е. суммарный объем выделенной памяти не должен существенно превышать размер реальной оперативной памяти).

10.3.2. Незакрытые курсоры

Ситуация, рассматриваемая в этом разделе, возникает вследствие ошибок в коде приложения, однако обнаруживается она только на уровне сервера базы данных. Опишем эту ситуацию детально.

Проблема возникает в системах с многоуровневой архитектурой, как правило, в том случае, если код приложения написан на языке программирования Java. Одним из важных достоинств этого языка является наличие полностью автоматических механизмов управления динамическим распределением памяти. В частности, программист избавлен от необходимости явно заботиться об уничтожении ненужных более объектов, потому что занятая ими память освобождается процедурой сборки мусора.

Более того, размеры доступной оперативной памяти обычно таковы, что выполнение небольшой программы, скорее всего, закончится раньше, чем память будет исчерпана, поэтому процедура сборки мусора не потребуется. Это вполне оправданно по отношению к объектам, создаваемым внутри адресного пространства приложения.

Весьма распространенная ошибка состоит в том, что программисты таким же образом обращаются с объектами, занимающими память за пределами адресного пространства приложения, в интересующем нас случае — в памяти сервера базы данных.

Для каждого оператора SQL, выполняемого по инициативе приложения, создается курсор (т. е. объект, описывающий взаимодействие приложения с сервером БД). После выполнения запроса курсор не может быть уничтожен, потому что приложение имеет право повторно выполнить тот же самый запрос (возможно, с другими значениями параметров) и, если этот запрос является запросом на выборку данных, приложение должно прочитать результаты и может это делать, вообще говоря, многократно.

Таким образом, сервер базы данных не может закрывать курсоры без явного запроса на уничтожение курсора со стороны приложения. Все курсоры создаются в рамках сеанса, и сервер базы данных защищается от ошибок приложения, закрывая курсоры при завершении сеанса работы с базой данных. Однако в рассматриваемом случае приложение выполняется под управлением сервера приложений, в котором обычно ведется пул сеансов. При завершении приложения сеанс возвращается в пул, поэтому СУБД не получает никаких сигналов о завершении приложения, в результате чего курсоры остаются открытыми.

В зависимости от типа СУБД ошибка проявляется по-разному. Если количество одновременно открытых курсоров ограничено (например, в Oracle), то приложение получает сообщение об ошибке, которое обычно интерпретируется как ошибка в конфигурации сервера базы данных. Естественно, увеличение предельного количества курсоров не может решить проблему, однако к тому времени, когда это обнаруживается, код приложения уже достаточно сложен и локализовать и исправить ошибку оказывается довольно трудно. Поиск места ошибки не облегчается тем, что сообщение об ошибке получает то приложение, которому не хватило курсоров, а не то, которое оставляло открытые курсоры.

В других СУБД явное ограничение на количество курсоров отсутствует, и поэтому накопление курсоров приводит к постепенной деградации производительности системы. Обнаружение ошибки в такой ситуации становится еще более сложным.

В тех случаях, когда проявления этой или аналогичных ошибок обнаруживаются во время производственной эксплуатации системы, в качестве быстрого решения применяется периодический рестарт сервера базы данных.

10.4. Управление индексами

Использование индексов при выполнении запросов уже рассматривалось в *главе 6*. На уровне сервера базы данных в целом управление индексами следует упомянуть только потому, что наличие (или отсутствие) определенных индексов влияет не только на поведение того класса запросов, для которого они создавались, но и на выполнение других запросов. Поэтому для принятия правильных решений необходим анализ использования индексов на уровне сервера.

Задача оптимального выбора индексов рассматривалась как задача дискретной оптимизации, однако сложность этой задачи значительно выше, чем, скажем, сложность оптимизатора запросов, поэтому применимые на практике методы являются эвристическими, а не точными, даже при использовании довольно сложных инструментов (необходимых для настройки индексов на очень больших серверах баз данных).

Убедиться в необходимости настройки индексов достаточно трудно, потому что поведение глобальных интегральных характеристик примерно такое же, как при неправильных настройках оптимизатора или неудачном использовании оперативной памяти.

По существу процесс настройки индексов можно представлять как расширенный процесс адаптивной оптимизации, упомянутый ранее в *разд. 10.2*. В отличие от адаптивной оптимизации, этот процесс является полуавтоматическим, потому что построение и уничтожение индексов рассматривается как модификация схемы хранения.

Новая структура индексов определяется на основе анализа запросов, на выполнение которых было потрачено наибольшее суммарное время. После построения новой индексной структуры необходимо заново выполнить оптимизацию всех запросов и получить новые данные о времени их выполнения.

Заметим, что, в отличие от адаптивной оптимизации, изменение структуры индексов может отрицательно повлиять на время выполнения других запросов, в итоге общая производительность системы может ухудшиться.

10.5. Размещение данных

Задача размещения данных на носителе всегда рассматривалась как одна из основных функций администратора базы данных. Важность размещения данных определяется тем, что от него существенно зависит не только производительность системы управления базами данных, но и качество работы оптимизатора.

Основным признаком плохого размещения данных является высокая загруженность дисковых устройств при относительно невысокой загрузке процессора сервера базы данных, однако такая комбинация характеристик может быть вызвана и другими причинами. Не менее важным признаком, указывающим на необходимость реорганизации размещения данных, является быстрое ухудшение времени ответа и пропускной способности системы, которое невозможно объяснить ростом объема хранимых данных или увеличением количества запросов, выполняемых системой.

При оценке стоимости выполнения операций обычно используются следующие предположения:

- время произвольного доступа к носителю данных примерно равно среднему времени обмена с носителем;
- время считывания данных при последовательной обработке намного (в десятки и сотни раз) ниже, чем время выборки при произвольном доступе.

Оба эти предположения не вполне точны. Время произвольного доступа складывается из времени подвода головок и времени подвода нужной части дорожки, поэтому оно зависит от исходного расположения головок перед началом операции. Этот факт, однако, не оказывает существенного влияния на поведение системы, так как независимо от размещения данных, время произвольного доступа изменяется в небольших пределах.

Предположение о времени доступа при последовательной обработке основано на другом предположении, а именно, предполагается, что данные, обрабатываемые последовательно, размещены в физически смежных областях дисковой памяти. Это предположение может оказаться неверным по нескольким причинам, приведенным далее.

- Мультипрограммирование. Операции последовательного считывания данных могут чередоваться с другими операциями обмена с тем же диском для выполнения иных запросов или для выполнения других задач в той же операционной системе.
- Внутренняя фрагментация. Обычно память для таблиц выделяется внутри табличного пространства несколькими участками.
- Внешняя фрагментация. Память для табличных пространств выделяется операционной системой также несколькими экстентами.

Влияние мультипрограммирования на уровне операционной системы обычно не очень существенно, потому что относительно низкая стоимость вычислительной техники позволяет выделять отдельные серверы для интенсивно используемых баз данных. Времена систем, хорошо работающих только по воскресеньям, ушли в прошлое.

Уровень конкурентности внутри сервера базы данных (т. е. количество одновременно выполняемых запросов) может регулироваться администратором базы данных, однако авторам не встречались случаи, в которых уровень мультипрограммирования существенно влиял бы на время последовательной обработки. Управление степенью конкурентности рассмотрено в *главе 8*.

Ухудшение производительности вследствие фрагментации, напротив, может быть очень значительным.

Как внутренняя, так и внешняя фрагментация дисков зависит от того, каким образом планируется и реализуется распределение памяти.

В связи с упомянутой выше тенденцией к полной автоматизации функций администратора базы данных все современные СУБД предоставляют средства автоматического распределения памяти для постоянного хранения данных как на внутреннем уровне (размещение хранимых объектов внутри табличных пространств), так и на внешнем (определение количества памяти, запрашиваемого для табличных пространств в файловой системе). Во многих случаях эти средства работают вполне удовлетворительно, но, как и в

других разделах книги, нас интересуют сейчас случаи, когда необходимо ручное вмешательство.

Независимо от того, в какой мере автоматизировано распределение памяти, при планировании учитываются следующие факторы:

- начальный объем данных, которые будут загружены при подготовке системы к производственному использованию;
- скорость и характер изменения объема данных;
- тип внутренней структуры данных (обычная таблица, кластеризованная, индекс и т. п.).

Фрагментация возникает в тех случаях, когда при проектировании системы какие-либо из этих характеристик неправильно оценены или адаптивные механизмы автоматической системы распределения памяти не смогли правильно предсказать поведение отдельных объектов. Например, если эксплуатация системы начинается с почти пустой базы данных, которая наполняется данными в начальный период эксплуатации, к тому времени, когда адаптивные механизмы накопят достаточно информации о характере поведения и размерах хранимых объектов, база данных может оказаться уже значительно фрагментированной.

По характеру динамического поведения можно выделить следующие виды объектов базы данных:

- малоизменяемые (почти статические) объекты, например, таблицы, содержащие большие объемы данных, импортированные из других систем;
- быстрорастущие объекты, например, накапливаемые данные, возникающие в результате функционирования приложения и подлежащие длительному хранению;
- объекты переменного размера, которые способны как расти, так и сокращаться; подобное поведение может быть характерно, например, для таблиц, содержащих большие объекты (BLOB или CLOB или другие эквивалентные типы данных).

Иногда характер поведения объектов может изменяться: например, быстрорастущие таблицы могут становиться малоизменяемыми после некоторого периода начального накопления данных.

Малоизменяемые объекты не способны вызвать появление фрагментации сами по себе. Если основная часть данных помещена в базу данных каким-либо средством массовой загрузки, скорее всего, такая таблица будет размещена в небольшом количестве экстентов большого размера, и в дальнейшем практически не будет ни влиять на механизмы распределения памяти, ни подвергаться фрагментации сама.

Наибольшую опасность представляет сочетание объектов с разными типами динамического поведения в одном табличном пространстве. Обычно СУБД

запрашивает расширение табличного пространства только в том случае, если нет возможности выделить место для расширения хранимого объекта в памяти, уже имеющейся в составе табличного пространства. Если к тому моменту, когда требуется расширить быстрорастущий объект, объекты переменного размера уже успели расшириться и затем сократиться, память, выделяемая для быстрорастущего объекта, скорее всего, окажется фрагментированной.

Для предотвращения фрагментации следует:

- распределять объекты по табличным пространствам на основе характера их поведения, с учетом скорости роста, динамики размеров и периодичности изменений размера;
- для объектов большого или потенциально большого размера задавать параметры, определяющие характер роста, явным образом, переопределяя значения, используемые системой по умолчанию.

Заметим, что в тех случаях, когда один логический объект базы данных представлен несколькими физическими, разные физические объекты могут иметь разную динамику изменения размеров. Наиболее известный пример — поведение индексов отличается от поведения таблицы, для которой эти индексы построены. На этом факте основана рекомендация размещать индексы в отдельном табличном пространстве.

Обнаружение внутренней фрагментации не составляет большой проблемы, потому что обычно информация о количестве и размерах экстентов доступна в системных таблицах базы данных.

Единственным способом устранения фрагментации является копирование данных на новое место, что для объектов (таблиц) большого размера может быть связано с организационными и технологическими осложнениями (во время переноса данные будут, скорее всего, недоступны для приложения, необходимо выделение большого количества дополнительной памяти и т. п.).

Возникновение внешней фрагментации определяется аналогичными причинами, однако встречается значительно реже, потому что табличные пространства намного менее динамичны. По-видимому, в наибольшей мере опасность внешней фрагментации появляется в том случае, если на рассматриваемой вычислительной системе работают (одновременно или периодически) несколько систем с большими потребностями в дисковой памяти, например, несколько серверов баз данных. Как мы уже отмечали, возникновение такой ситуации на сервере базы данных, находящейся в производственной эксплуатации, маловероятно.

10.6. Эволюция системы

Средства управления памятью в разных версиях даже одной и той же СУБД могут существенно различаться. Это может быть вызвано как улучшением самих этих средств, так и изменениями в других компонентах системы,

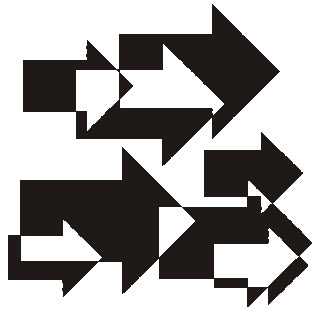
например, модификацией алгоритмов выполнения запросов или оптимизатора, а также увеличением объема оперативной памяти, который может использовать система.

Тем не менее, обычно новая версия системы может обрабатывать и учитывать параметры конфигурационных файлов, которые были необходимы в старой версии. Наличие таких параметров может полностью или частично блокировать работу средств автоматической конфигурации в новой версии.

В подобных случаях копирование старого файла параметров в новую систему является ошибкой администратора базы данных.

Правильная стратегия перехода на новую версию системы должна предусматривать повторную настройку параметров управления оперативной памятью, а не перенос значений из старого файла, даже если эти параметры были получены в результате тщательной настройки системы.

Разумеется, это относится не только к параметрам управления оперативной памятью. Например, перенос параметров управления оптимизатором также может блокировать новые возможности оптимизатора и, следовательно, ухудшить производительность системы.



Глава 11

Заключение

Технологии, связанные с применением систем баз данных, непрерывно развиваются. Серверы баз данных, в особенности высокопроизводительные, становятся все более совершенными. Одна из основных современных тенденций развития — включение средств автоматической настройки. Новые версии продуктов крупных производителей способны не только автоматически выбирать параметры сервера в целом, но также выявлять и автоматически настраивать отдельные запросы, на выполнение которых затрачивается слишком много ресурсов.

В то же время постоянно расширяется круг задач, автоматизированное решение которых оказывается технологически возможным и экономически целесообразным, тогда как сложность этих задач растет.

Одновременно изменяются как характер проблем, которые приходится решать при настройке, так и приемы, необходимые для достижения целей настройки.

В период, когда в приложениях доминировали очень короткие (классические OLTP) запросы, основные усилия при настройке были необходимы для правильного подбора параметров сервера базы данных (таких как количество процессов и размеры буферного пула). Современные системы делают это автоматически, если, конечно, чересчур старательный администратор базы данных не мешает им в этом.

С ростом сложности запросов центр тяжести переместился на настройку отдельных запросов SQL, однако улучшение качества оптимизаторов, скорее всего, приведет к почти полной автоматизации и на этом уровне, а задачи настройки будут смещаться в другие области.

Неудачное применение технологий разработки программ или неадекватное использование инструментов, средств, готовых компонентов и пакетов

программ может полностью блокировать работу оптимизатора запросов, как показано в *главе 4* данной книги.

Расширение объектных возможностей СУБД и повсеместное использование языка XML в качестве основного транспортного механизма могут привести к тому, что наиболее сложные задачи настройки будут связаны с распределением функциональности приложения между уровнями и компонентами прикладной системы.

Использование объектных возможностей СУБД позволяет перенести часть функциональности, реализуемой в современной практике на сервере приложений, непосредственно в СУБД, что может привести к существенному повышению эффективности всей прикладной системы. С другой стороны, использование методов объектов, осуществляющих доступ к хранимым данным, в сложных запросах может привести к появлению проблем производительности.

Одной из привлекательных особенностей расширений для обработки XML является возможность вывода сложных объектов в качестве ответа на один-единственный запрос к серверу базы данных. В результате выполнения запроса генерируется дерево, содержащее данные из, вообще говоря, нескольких взаимосвязанных таблиц. Это позволяет применять более крупные запросы и одновременно упростить код приложения. Подобные возможности предоставляются и объектными расширениями, однако они редко использовались на практике. Возможно, для XML положение будет другим.

Причины, по которым производительность системы оказывается неудовлетворительной, весьма разнообразны, поэтому их анализ и выявление часто являются трудной задачей. Методы и приемы, хорошо показавшие себя в одних случаях, часто бывают абсолютно бесполезными в других. Это, конечно, относится и к конкретным приемам, описанным в данной книге. В новых проектах могут потребоваться другие методы настройки. Набор инструментов также приходится обновлять довольно часто. Возможно, именно это и делает работу специалиста по настройке систем трудной и привлекательной одновременно.

Литература

1. Atkinson M. P., Bancilhon F., DeWitt D. J., Dittrich K. R., Maier D., and Zdonik S. B. The object-oriented database system manifesto. In DOOD, p. 223—240, 1989.
2. Bernstein P., Hadzilacos V., and Goodman N. Concurrency Control in the Database Systems. Addison-Wesley, Reading, Mass., 1987.
3. Catell R., Jordan D., Barry D. K., Russell C., and Berler M. The Object Data Standard: ODMG 3.0. Morgan-Kaufmann, San Francisco etc., 1999.
4. Date C., Darwin H., and Lorentzos N. A. Temporal data and relational model. Morgan-Kaufmann, San Francisco, 2003.
5. Graefe G. Query evaluation techniques for large databases. ACM Comput. Surv., 25(2):73—169, 1993.
6. Gray J. Transaction processing: concepts and techniques. Morgan Kaufmann Publishers, San-Francisco, California, 1993.
7. Mishra P. and Eich M. H. Join processing in relational databases. ACM Comput. Surv., 24(1):63—113, 1992.
8. C. Mohan. Recent trends in workflow management. In Products, Standards and Research, Proc. NATO Advanced Study Institute (ASI) on Workflow Management Systems and Interoperability, Istanbul, August 1997, number 164 in NATO ASI Series, Series F: Computer and Systems Sciences, p. 133—139, Heidelberg etc., 1998. Springer Verlag.
9. Human Resources Management Systems. Oracle Corp., **http://www.oracle.com/appsnet/products/hr/content.html**.
10. Optimizing Human Resources for E-Business. Oracle Corp., **http://www.oracle.com/appsnet/products/hr/collateral/hr-brochure.pdf**, 2000.
11. Shasha D. and Bonnet P. Database tuning: principles, experiments and troubleshooting techniques. Morgan-Kaufmann, San Francisco, CA, 2002.

12. Stonebraker M., Rowe L. A., Lindsay B. G., Gray J., Carey M. J., Brodie M. L., Bernstein P. A., and Beech D. Third-generation database system manifesto. SIGMOD Rec., 19(3):31—44, 1990.

13. Thomasian A. Concurrency control: methods, performance, and analysis. ACM Comput. Surv., 30(1):70—119, 1998.

14. Tow D. SQL tuning. O'Reily, Sebastopol, CA, 2003.

15. Vitter J. S. External memory algorithms and data structures: dealing with massive data. ACM Comput. Surv., 33(2):209—271, 2001.

16. G. Vossen and G. Weikum. Transactional Systems. Morgan-Kaufmann, San Francisco, 2001.

17. Гарсиа-Молина Г., Ульман Д., Видом Д. Системы баз данных. Полный курс: Пер. с англ. — М.: Издательский дом "Вильямс", 2003.

18. Дейт К. Введение в системы баз данных: Пер. с англ. — М.: Издательский дом "Вильямс", 8-е изд., 2005.

19. Кузнецов С. Основы баз данных. Основы информационных технологий. Интернет-университет информационных технологий. — М., 2005.

20. Пушкин А. Борис Годунов. — М.: Современник, 1974. С. 177—254.

21. Роб П., Корнел К. Системы баз данных: проектирование, реализация, управление. — СПб.: БХВ-Петербург, 2004.

22. Информационные системы общего назначения. Аналитический обзор систем управления базами данных / Под ред. Е. Ющенко: Пер. с англ. — М.: Статистика, 1975.

Предметный указатель

Д

Ж

З

И

К

Л

М

Н

О

П

Р

С

Т

У

Ф

Х

Ч

Э

Я