Проектирование и реализация систем управления базами данных

Эдвард Сьоре

Проектирование и реализация систем управления базами данных

Эдвард Сьоре

Проектирование и реализация систем управления базами данных

Database Design and Implementation

Edward Sciore

Проектирование и реализация систем управления базами данных

Эвард Сьоре

Москва, 2021

УДК 004.655
ББК 32.973.26-018.2
С96

Эдвард Сьоре
С96 Проектирование и реализация систем управления базами данных / пер. с анг. А. Н. Киселева; научн. ред. Е. В. Рогов. – М.: ДМК Пресс, 2021. – 466 с.: ил.

ISBN 978-5-97060-488-5

В книге рассматриваются системы баз данных с точки зрения разработчика ПО. Автор подробно разбирает исходный код полностью функциональной, но при этом очень простой для изучения системы баз данных SimpleDB и предлагает читателям, изменяя отдельные ее компоненты, разобраться в том, к чему это приведет. Это отличный способ погрузиться в тему и изучить, как работают базы данных, на уровне исходного кода.

В начале книги приводится краткий обзор систем баз данных; рассказывается о том, как написать приложение базы данных на Java. Далее подробно описываются отдельные компоненты типичной системы баз данных, начиная с самого низкого уровня абстракции (управление дисками и диспетчер файлов) и заканчивая самым верхним (интерфейс клиента JDBC). Заключительные главы посвящены эффективной обработке запросов. В конце каждой главы приводятся практические упражнения и список дополнительных ресурсов.

Издание предназначено для студентов вузов, изучающих курс информатики, а также всех, кто хочет научиться создавать системы баз данных. Предполагается, что читатель знаком с основами программирования на Java.

УДК 004.655
ББК 32.973.26-018.2

ISBN (анг.) 978-3-030-33835-0
ISBN (рус.) 978-5-97060-488-5

Оглавление

Предисловие от издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте **www.dmkpress.com**, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу **dmkpress@gmail.com**; при этом напишите название книги в теме письма.

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

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в основном тексте или программном коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Если вы найдете какие-либо ошибки в тексте, пожалуйста, сообщите о них главному редактору по адресу **dmkpress@gmail.com**, и мы исправим это в следующих тиражах.

Нарушение авторских прав

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

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

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

Вступление

Системы баз данных широко распространены в корпоративном мире как видимый инструмент – сотрудники часто напрямую взаимодействуют с такими системами, чтобы отправить данные или создать отчеты. Но не менее часто они используются как невидимые компоненты программных систем. Например, представьте веб-сайт электронной коммерции, использующий базу данных на стороне сервера для хранения информации о клиентах, товарах и продажах. Или вообразите систему навигации GPS, использующую встроенную базу данных для управления картами дорог. В обоих этих примерах система баз данных скрыта от пользователя; с ней взаимодействует только код приложения.

С точки зрения разработчика программного обеспечения, обучение непосредственному использованию базы данных выглядит скучным занятием, потому что современные системы баз данных имеют интеллектуальные пользовательские интерфейсы, упрощающие создание запросов и отчетов. С другой стороны, включение поддержки базы данных в программное приложение выглядит более захватывающим, поскольку открывает множество новых и неисследованных возможностей.

Но что означает «включение поддержки базы данных»? Система баз данных обеспечивает множество новых возможностей, таких как долговременное хранение данных, поддержка транзакций и обработка запросов. Какие из этих возможностей необходимы, и как их интегрировать в программное обеспечение? Предположим, например, что программиста просят изменить существующее приложение и добавить возможность сохранения состояния, повысить надежность или эффективность доступа к файлам. Программист оказывается перед выбором между несколькими архитектурными вариантами. Он может:

- приобрести полноценную систему баз данных общего назначения, а затем изменить приложение и организовать в нем подключение к базе данных в качестве клиента;
- взять узкоспециализированную систему, реализующую только нужные функции, и встроить ее код непосредственно в приложение;
- написать необходимые функции самостоятельно.

Чтобы сделать правильный выбор, программист должен понимать последствия работы каждого из этих вариантов. Он должен знать не только то, что делают системы баз данных, но также как они это делают и почему.

В этой книге мы рассмотрим системы баз данных с точки зрения разработчика программного обеспечения. Это позволит нам понять, *почему* системы баз данных являются такими, какие они есть. Конечно, важно уметь писать запросы, но не менее важно знать, как они обрабатываются. Мы должны не просто уметь использовать JDBC, но и знать и понимать, почему API содержит именно

такие классы и методы. Мы должны понять, насколько сложно написать дисковый кеш или подсистему журналирования. И вообще, *что такое* драйвер базы данных.

Организация книги

Первые две главы содержат краткий обзор систем баз данных и принципов их использования. Глава 1 обсуждает назначение и особенности системы баз данных и знакомит с системами Derby и SimpleDB. Глава 2 рассказывает, как написать приложение базы данных на Java. В ней будут представлены основы JDBC – фундаментального API для программ на Java, взаимодействующих с базами данных.

В главах 3–11 рассматривается внутреннее устройство типичного механизма базы данных. Каждая из этих глав охватывает отдельный компонент, начиная с самого низкого уровня абстракции (диспетчер дисков и файлов) и заканчивая интерфейсом самого верхнего уровня (интерфейс клиента JDBC). При обсуждении каждого компонента объясняются вероятные проблемы и рассматриваются возможные проектные решения. Благодаря такому подходу вы сможете увидеть, какие услуги предоставляет каждый компонент и как он взаимодействует друг с другом. На протяжении этой части книги вы будете наблюдать постепенное развитие простой, но вполне функциональной системы.

Остальные четыре главы посвящены эффективной обработке запросов. В них исследуются сложные приемы и алгоритмы, предназначенные для замены простых решений, описанных в предыдущей части. Среди всего прочего здесь рассматриваются: индексация, сортировка, интеллектуальная буферизация и оптимизация запросов.

Предварительные требования

Эта книга предназначена для студентов вузов старших курсов, изучающих курс информатики. Предполагается, что читатель знаком с основами программирования на Java, например он умеет использовать классы из `java.util`, в частности коллекции и ассоциативные массивы. Более сложные понятия из мира Java (такие как RMI и JDBC) будут полностью объяснены в тексте.

Сведения, представленные в данной книге, обычно изучаются в углубленном курсе по системам баз данных. Однако в моей преподавательской практике их с успехом усваивали студенты, не имеющие опыта работы с базами данных. Поэтому можно сказать, что для понимания идей, представленных в этой книге, не требуется иметь какие-либо знания о базах данных, кроме поверхностного знакомства с SQL. Впрочем, студенты, незнакомые с SQL, также смогут усвоить необходимые им знания.

Программное обеспечение SimpleDB

По моему опыту, студенты быстро схватывают концептуальные идеи (такие как управление параллелизмом и буферами или алгоритмы оптимизации запросов), но им сложнее понять, как эти идеи взаимодействуют. В идеале сту-

дент должен написать всю систему баз данных в своей курсовой работе, так же, как он написал бы целый компилятор при изучении курса по компиляторам. Однако системы баз данных намного сложнее компиляторов, поэтому такой подход выглядит непрактичным. Поэтому я решил сам написать простую, но полнофункциональную систему баз данных под названием *SimpleDB* и дать возможность студентам применить свои знания для изучения кода SimpleDB и его изменения.

По своим возможностям и структуре SimpleDB «выглядит» как коммерческая система баз данных. Функционально это многопользовательский сервер баз данных с поддержкой транзакций, который выполняет операторы SQL и взаимодействует с клиентами через интерфейс JDBC. Конструктивно она содержит те же основные компоненты, что и коммерческая система с похожим API. Каждому компоненту SimpleDB посвящена отдельная глава в книге, где обсуждается код компонента и проектные решения, стоящие за ним.

SimpleDB является прекрасным наглядным пособием благодаря небольшому объему кода, который легко читается и модифицируется. В ней отсутствует необязательная функциональность, реализована только малая часть языка SQL и используются лишь самые простые (и часто очень неоптимальные) алгоритмы. Как результат перед студентами открывается широкое поле возможностей расширения системы дополнительными функциями и применения более эффективных алгоритмов; многие из этих расширений предлагаются в конце каждой главы как упражнения.

Исходный код SimpleDB можно получить по адресу http://cs.bc.edu/~sciore/simpledb. Подробная информация об установке и использовании SimpleDB представлена на этой же веб-странице и в главе 1. Я с благодарностью приму любые предложения по улучшению кода, а также сообщения о любых ошибках. Пишите мне по адресу sciore@bc.edu.

Список дополнительных ресурсов в конце каждой главы

Главная задача этой книги – дать ответы на два основных вопроса: какие функции предоставляют системы баз данных и какие алгоритмы и проектные решения лучше всего подходят для их реализации? Книгами, посвященными различным аспектам этих вопросов, можно заполнить не одну книжную полку. Поскольку ни одна книга не может охватить все и вся, я решил представить только те алгоритмы и методы, которые наиболее четко иллюстрируют соответствующие проблемы. Моя главная цель – познакомить вас с основными принципами, даже если придется опустить (или сократить) обсуждение более жизнеспособной версии того или иного метода. Поэтому в конце каждой главы имеется раздел «Для дополнительного чтения». В этих разделах обсуждаются интересные идеи и направления исследований, не упомянутые в тексте, и даются ссылки на соответствующие веб-страницы, исследовательские статьи, справочные руководства и книги.

Упражнения в конце каждой главы

В конце каждой главы дается несколько упражнений. Некоторые, имеющие целью закрепить полученные знания, можно решить карандашом на бумаге. В других предлагаются интересные модификации в SimpleDB, и многие из них могут служить отличными программными проектами. Для большинства упражнений я написал свои решения. Если вы преподаете курс по этому учебнику и хотите получить копию руководства с решениями, напишите мне по адресу sciore@bc.edu.

Об авторе

Эдвард Сьоре (**Edward Sciore**) – недавно вышедший на пенсию доцент кафедры информатики в Бостонском колледже. Автор многочисленных статей о системах баз данных, охватывающих теорию и практику их разработки. Больше всего ему нравится преподавание курсов баз данных увлеченным студентам. Этот опыт преподавания, накопленный за 35-летний период, привел к написанию данной книги.

Глава 1

Системы баз данных

Системы баз данных играют важную роль в компьютерной индустрии. Некоторые из них (такие как Oracle) чрезвычайно сложны и, как правило, работают на больших высокопроизводительных компьютерах. Другие (такие как SQLite) имеют небольшой размер и предназначены для хранения данных отдельных приложений. Несмотря на широкий спектр применения, все системы баз данных имеют схожие черты. В этой главе рассматриваются проблемы, которые должна решать система баз данных, и возможности, которыми она должна обладать. Здесь также будут представлены системы баз данных Derby и SimpleDB, обсуждающиеся далее в этой книге.

1.1. Зачем нужны системы баз данных?

База данных – это коллекция данных, хранящаяся на компьютере. Обычно информация в базе данных организована в *записи*, например в записи с информацией о сотрудниках, медицинские записи, записи о продажах и т. д. В табл. 1.1 представлена база данных, хранящая информацию о студентах университета и изученных ими курсах. Эта база данных будет использоваться как учебный пример на протяжении всей книги. В базе данных в табл. 1 хранятся записи пяти типов:

- для каждого студента, учившегося в университете, имеется запись STUDENT; каждая запись содержит идентификационный номер студента, имя, год выпуска и идентификационный номер основной кафедры;
- для каждой кафедры в университете имеется запись DEPT; каждая запись содержит идентификационный номер кафедры и название;
- для каждого обучающего курса, предлагаемого университетом, имеется запись COURSE; каждая запись содержит идентификационный номер курса, название и идентификационный номер кафедры, которая его предлагает;
- для каждого раздела курса, который когда-либо читался, имеется запись SECTION; каждая запись содержит идентификационный номер раздела, год, когда был предложен этот раздел, идентификационный номер курса и имя преподавателя, читающего этот раздел;
- для каждого курса, который прослушал студент, имеется запись ENROLL; каждая запись содержит идентификационные номера зачисления, студента и раздела пройденного курса, а также оценку, полученную студентом за курс.

Таблица 1.1. Примеры записей в университетской базе данных

STUDENT	SId	SName	GradYear	MajorId
	1	joe	2021	10
	2	amy	2020	20
	3	max	2022	10
	4	sue	2022	20
	5	bob	2020	30
	6	kim	2020	20
	7	art	2021	30
	8	pat	2019	20
	9	lee	2021	10

DEPT	DId	DName
	10	compsci
	20	math
	30	drama

COURSE	CId	Title	DeptId
	12	db	systems
	22	compilers	10
	32	calculus	20
	42	algebra	20
	52	acting	30
	62	elocution	30

ENROLL	EId	StudentId	SectionId	Grade
	14	1	13	A
	24	1	43	C
	34	2	43	B+
	44	4	33	B
	54	4	53	A
	64	6	53	A

SECTION	SectId	CourseId	Prof	YearOffered
	13	12	turing	2018
	23	12	turing	2016
	33	32	newton	2017
	43	32	einstein	2018
	53	62	brando	2017

Таблица 1.1 – это всего лишь концептуальная схема. Она ни в коей мере не отражает порядок хранения записей или особенности доступа к ним. Существует множество программных продуктов, называемых *системами баз данных*, которые предоставляют обширный набор средств управления записями.

Что имеется в виду под «управлением» записями? Какие возможности должны иметься в системе баз данных, а какие могут отсутствовать? Вот пять основных требований:

- *базы данных должны обеспечивать долговременное хранение*; в противном случае записи исчезнут после выключения компьютера;
- *базы данных могут быть общими*; многие базы данных, такие как наша университетская база данных, предназначены для одновременного использования несколькими пользователями;
- *базы данных должны гарантировать* безошибочное хранение информации; если пользователи не смогут доверять содержимому базы данных, она станет бесполезной;
- *базы данных могут быть очень большими*; база данных в табл. 1.1 содержит всего 29 записей, что смехотворно мало – базы данных с миллионами (и даже миллиардами) записей совсем не редкость;
- *базы данных должны быть удобными*; если пользователи не смогут с легкостью получать нужные данные, их производительность снизится, и они будут требовать использовать другой продукт.

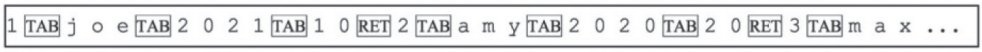


Рис. 1.1. Реализация записи STUDENT в текстовом файле

В следующих подразделах рассматриваются следствия этих требований. Каждое требование вынуждает включать в систему баз данных все больше и больше функций, из-за чего она оказывается намного сложнее, чем можно было бы ожидать.

1.1.1. Хранилище записей

Типичный способ обеспечить сохранность данных – хранить записи в файлах. А самый простой подход к хранению записей в файлах – сохранение их в текстовых файлах, по одному файлу для каждого типа записей; каждую запись можно представить в виде текстовой строки, в которой значения отделены друг от друга символами табуляции. На рис. 1.1 показано начало текстового файла с записями типа STUDENT.

Преимущество этого подхода в том, что пользователь может просматривать и изменять файлы с помощью простого текстового редактора. К сожалению, этот подход слишком неэффективен по двум причинам.

Первая причина в том, что для изменения данных в больших текстовых файлах требуется слишком много времени. Представьте, например, что кто-то решил удалить из файла STUDENT запись с информацией о студенте Joe. У системы баз данных не будет иного выбора, кроме как переписать файл, начав с записи о студенте Amy. Короткий файл будет переписан быстро, но на перезапись файла объемом 1 Гбайт легко может уйти несколько минут, а это недопустимо долго. Система баз данных должна использовать более оптимальные способы хранения записей, чтобы при изменении данных достаточно было перезаписать локальные фрагменты.

Вторая причина – для чтения больших текстовых файлов требуется слишком много времени. Представьте поиск в файле STUDENT всех студентов, выпущенных в 2019 году. Единственный способ – просканировать файл от начала до конца. Последовательное сканирование может быть очень неэффективным. Возможно, вам знакомы некоторые структуры данных, такие как деревья и хеш-таблицы, которые обеспечивают быстрый поиск. Система баз данных должна использовать аналогичные структуры для реализации своих файлов. Например, система баз данных может организовать записи в файле, используя структуру, ускоряющую один конкретный тип поиска (например, по имени студента, году выпуска или специальности), или создать несколько вспомогательных файлов, каждый из которых ускоряет свой тип поиска. Такие вспомогательные файлы называются *индексами* и являются предметом обсуждения главы 12.

1.1.2. Многопользовательский доступ

Когда базой данных пользуется несколько человек, есть вероятность, что они одновременно обратятся к одним и тем же ее файлам. Параллелизм – хорошая штука, потому что избавляет пользователей от необходимости ждать, пока другие закончат работу. Но параллелизм может привести к появлению ошибок в базе

данных. Например, представьте базу данных в системе бронирования билетов на авиарейсы. Предположим, что два клиента пытаются забронировать билеты на рейс, в котором осталось 40 мест. Если оба пользователя одновременно прочитают одну и ту же запись, они оба увидят 40 доступных мест. Затем они оба изменят эту запись так, чтобы в ней осталось 39 свободных мест. В результате такого резервирования двух мест в базе данных будет зарегистрировано только одно место.

Решением этой проблемы является ограничение параллелизма. Система баз данных должна позволить первому пользователю прочитать запись о рейсе и увидеть 40 доступных мест, но заблокировать второго пользователя до того момента, пока первый не закончит работу. Когда второй пользователь получит возможность продолжить работу, он увидит 39 доступных мест и уменьшит это количество до 38, как и должно быть. То есть система баз данных должна обнаруживать ситуации, когда один пользователь собирается выполнить действие, конфликтующее с действиями другого пользователя, и тогда (и только тогда) блокировать работу этого пользователя, пока первый не завершит операцию.

Пользователям также может понадобиться отменить свои изменения. Например, представьте, что пользователь выполнил поиск в базе данных бронирования билетов и нашел дату, на которую есть свободные места на рейс в Мадрид. Затем он нашел свободный номер в гостинице на ту же дату. Но пока пользователь бронировал место на рейс, все номера в гостиницах на эту дату были забронированы другими людьми. В этом случае пользователю может потребоваться отменить бронирование рейса и попробовать выбрать другую дату.

Изменение, которое еще можно отменить, не должно быть видно другим пользователям базы данных. В противном случае другой пользователь, увидев изменение, может подумать, что данные «настоящие», и принять решение на их основе. Поэтому система баз данных должна давать пользователям возможность указывать, когда их изменения можно сохранить, или, как говорят, *зафиксировать* (подтвердить). Как только пользователь зафиксирует изменения, они становятся видимыми всем остальным и уже не могут быть отменены. Этот вопрос рассматривается в главе 5.

1.1.3. Обработка аварийных ситуаций

Представьте, что вы запустили программу, которая повышает зарплату всем профессорам, и вдруг система баз данных аварийно завершается. После перезапуска системы вы понимаете, что кому-то из профессоров уже назначена новая зарплата, а кому-то еще нет. Как быть в такой ситуации? Нельзя просто повторно запустить программу, потому что кто-то из профессоров получит двойную прибавку. То есть вам нужно, чтобы система баз данных правильно восстановилась после сбоя, отменив все изменения, вносившиеся в момент сбоя. Для этого используется интересный и нетривиальный механизм, который рассматривается в главе 5.

1.1.4. Управление памятью

Базы данных должны хранить информацию в устройствах долговременной памяти, таких как жесткие диски или твердотельные накопители на основе флеш-памяти. Твердотельные накопители действуют примерно в 100 раз быстрее жестких дис-

ков, но и стоят значительно дороже. Типичное время доступа к диску составляет примерно 6 мс, а к флеш-памяти – 60 мкс. Однако оба этих устройства на несколько порядков медленнее оперативной памяти (или ОЗУ), время доступа которой составляет около 60 нс. То есть операции с ОЗУ выполняются примерно в 1000 раз быстрее, чем с флеш-памятью, и в 100 000 раз быстрее, чем с жестким диском.

Чтобы почувствовать разницу в производительности и увидеть, какие проблемы она порождает, рассмотрим следующую аналогию. Предположим, вы очень хотите шоколадное печенье. Есть три способа получить его: взять у себя на кухне, сходить в соседний продуктовый магазин или заказать доставку по почте. В этой аналогии кухня соответствует оперативной памяти, соседний магазин – флеш-накопителю, заказ по почте – диску. Допустим, чтобы взять печенье на кухне, требуется всего 5 секунд. Поход в магазин займет 5000 секунд (это больше часа): вам понадобится добраться до магазина, выстоять длинную очередь, купить печенье и вернуться домой. А для получения печенья по почте потребуется 500 000 секунд – больше 5 дней: вы должны будете заказать печенье через интернет, а компания отправит его вам обычной почтой. С этой точки зрения флеш-память и диски выглядят очень медленными.

Но это еще не все! Механизмы поддержки параллелизма и обеспечения надежности замедляют работу еще больше. Если данные, которые вам нужны, уже использует кто-то еще, вам придется подождать, пока эти данные будут освобождены. В нашей аналогии это можно представить так: вы пришли в магазин и обнаружили, что печенье распродано. В этом случае вам придется подождать, пока завезут новую партию.

Другими словами, система баз данных должна соответствовать противоречивым требованиям: управлять бóльшим объемом данных, чем умещается в оперативную память, используя медленные устройства, обслуживать одновременно множество людей, конкурирующих за доступ к данным, и обеспечить возможность полного восстановления этих данных, сохраняя при этом разумное время отклика.

Большая часть решения этой головоломки заключается в использовании кеширования. Всякий раз, когда требуется обработать запись, система баз данных загружает ее в оперативную память и старается хранить ее там как можно дольше. При таком подходе часть базы данных, используемая в данный момент, будет находиться в оперативной памяти. Все операции чтения и записи будут выполняться с оперативной памятью. Эта стратегия позволяет вместо медленной долговременной памяти использовать быструю оперативную память, но она имеет существенный недостаток – версия базы данных на устройствах хранения может устареть. В системе баз данных должны быть реализованы методы надежной синхронизации версии базы данных на устройствах хранения с версией в ОЗУ, даже в случае сбоя (когда содержимое ОЗУ уничтожается). Различные стратегии кеширования мы рассмотрим в главе 4.

1.1.5. Удобство

База данных может оказаться бесполезной, если пользователи не смогут с легкостью извлекать нужные данные. Например, представьте, что пользователь решил узнать имена всех студентов, окончивших учебу в 2019 году. В отсутствие системы баз данных пользователь будет вынужден написать програм-

му для сканирования файла со списком студентов. В листинге 1.1 показан код на Java такой программы, в котором предполагается, что файл хранит записи в текстовом виде. Обратите внимание, что большая часть кода связана с декодированием файла, чтением каждой записи и преобразованием ее в массив значений. Код, сопоставляющий фактические значения с искомыми (выделен жирным шрифтом) и извлекающий имена студентов, скрыт внутри малоинтересного кода, манипулирующего файлом.

Листинг 1.1. Поиск имен студентов, закончивших обучение в 2019 году

```
public static List<String> getStudents2019() {
    List<String> result = new ArrayList<>();
    FileReader rdr = new FileReader("students.txt");
    BufferedReader br = new BufferedReader(rdr);
    String line = br.readLine();
    while (line != null) {
        String[] vals = line.split("\t");
        String gradyear = vals[2];
        if (gradyear.equals("2019"))
            result.add(vals[1]);
        line = br.readLine();
    }
    return result;
}
```

По этой причине большинство систем баз данных поддерживают язык запросов, чтобы пользователи могли легко определить искомые данные. Стандартным языком запросов для реляционных баз данных является SQL. Код в листинге 1.1 можно выразить одним оператором SQL:

```
select SName from STUDENT where GradYear = 2019
```

Этот оператор намного короче и нагляднее, чем программа на Java, в первую очередь потому, что определяет, какие значения требуется извлечь из файла, а не как они должны извлекаться.

1.2. Система баз данных Derby

Изучение идей, лежащих в основе баз данных, протекает намного эффективнее, если есть возможность наблюдать за работой системы баз данных в интерактивном режиме. В настоящее время существует множество доступных систем баз данных, но я предлагаю использовать *Derby*, потому что она написана на Java, распространяется бесплатно, проста в установке и в использовании. Последнюю версию Derby можно загрузить на вкладке downloads, на странице db.apache.org/derby. Загруженный файл дистрибутива нужно распаковать в папку для установки, после чего в ней вы сможете обнаружить несколько каталогов. Например, каталог `docs` содержит справочную документацию, каталог `demo` – образец базы данных и т. д. Эта система обладает гораздо более широким кругом возможностей, чем можно описать здесь, поэтому те, кому это интересно, могут прочитать различные руководства в каталоге `docs`.

Derby имеет множество функций, которые не затрагиваются в этой книге. На самом деле вам нужно добавить в путь поиска классов `classpath` четыре фай-

ла из каталога lib: derby.jar, derbynet.jar, derbyclient.jar и derbytools.jar. Сделать это можно множеством способов, в зависимости от платформы Java и операционной системы. Я покажу, как это сделать, на примере платформы разработки Eclipse. Если вы незнакомы с Eclipse, то можете скачать ее код и документацию с сайта eclipse.org. Использующие другие платформы разработки смогут адаптировать мои указания для Eclipse под особенности своего окружения.

Прежде всего создайте в Eclipse проект для сборки Derby. Затем настройте путь сборки: в диалоге Properties (Свойства) выберите слева пункт Java Build Path (Путь сборки Java); щелкните на вкладке Libraries (Библиотеки), затем на кнопке Add External JARS (Добавить внешние JAR) и в открывшемся диалоге выберите четыре JAR-файла, перечисленных выше. Вот и все!

Дистрибутив Derby содержит приложение ij, помогающее создавать и использовать базы данных Derby. Так как Derby целиком написана на Java, название ij фактически является именем класса Java, находящегося в пакете org.apache.derby.tools. Запустить ij можно, выполнив этот класс. Чтобы выполнить класс из Eclipse, откройте диалог Run Configurations (Запуск конфигурации) в меню Run (Выполнить). Добавьте новую конфигурацию в свой проект Derby; дайте ей имя «Derby ij». В поле, определяющее главный класс, введите «org.apache.derby.tools.ij». После запуска этой конфигурации ij отобразит окно консоли с приглашением к вводу.

В данной консоли можно вводить последовательности команд. Команда – это строка, заканчивающаяся точкой с запятой. Команды можно вводить в несколько строк; клиент ij не выполнит команду, пока не встретит строку, заканчивающуюся точкой с запятой. Любой оператор SQL считается допустимой командой. Кроме того, ij поддерживает команды подключения и отключения от базы данных и завершения сеанса.

Команда connect должна иметь аргумент, определяющий базу данных, и подключается к ней, а команда disconnect отключается от нее. В одном сеансе можно подключаться к базе данных и отключаться от нее сколько угодно раз. Команда exit завершает сеанс. В листинге 1.2 показан пример сеанса ij. Сеанс состоит из двух частей. В первой части пользователь подключается к новой базе данных, создает таблицу, добавляет в нее запись и отключается. Во второй части пользователь повторно подключается к этой же базе данных, извлекает добавленные значения и отключается.

Листинг 1.2. Пример сеанса ij

```
ij> connect 'jdbc:derby:ijtest;create=true';
ij> create table T(A int, B varchar(9));
0 rows inserted/updated/deleted
ij> insert into T(A,B) values(3, 'record3');
1 row inserted/updated/deleted
ij> disconnect;
ij> connect 'jdbc:derby:ijtest';
ij> select * from T;
A          |B
----------------------
3          |record3
1 row selected
ij> disconnect;
ij> exit;
```

Аргумент команды `connect` называется *строкой подключения*. Строка подключения состоит из трех компонентов, разделенных двоеточиями. Первые два компонента – «jdbc» и «derby» – сообщают, что команда должна подключиться к базе данных Derby с использованием протокола JDBC (обсуждается в главе 2). Третий компонент идентифицирует базу данных. В данном случае «ijtest» – это имя базы данных; ее файлы будут находиться в папке с именем «ijtest», расположенной в каталоге, откуда было запущено приложение `ij`. Например, если запустить программу из Eclipse, папка базы данных окажется в каталоге проекта. Подстрока «create = true» сообщает системе Derby, что та должна создать новую базу данных; если опустить эту подстроку (как во второй команде `connect`), тогда Derby будет пытаться открыть существующую базу данных.

1.3. Механизмы баз данных

Приложение баз данных, такое как `ij`, состоит из двух независимых частей: пользовательского интерфейса и кода для доступа к базе данных. Этот код называется *механизмом (или движком) базы данных*. Отделение пользовательского интерфейса от движка базы данных – типичный архитектурный прием, упрощающий разработку приложений. Хорошо известным примером такого разделения может служить система баз данных Microsoft Access. Она имеет графический пользовательский интерфейс, позволяющий пользователю взаимодействовать с базой данных, щелкая мышью и вводя значения, а движок выполняет операции с хранилищем данных. Когда пользовательский интерфейс определяет, что ему нужна информация из базы данных, он создает запрос и передает его движку. После этого движок выполняет запрос и передает полученные значения обратно в пользовательский интерфейс.

Такое разделение также добавляет гибкости системе: разработчик приложения может использовать один и тот же пользовательский интерфейс с разными движками баз данных или создавать разные пользовательские интерфейсы для одного и того же движка. Microsoft Access как раз является примером каждого случая. Форма, созданная в пользовательском интерфейсе Access, может подключаться к движку Access или к любому другому механизму баз данных. Ячейки в электронной таблице Excel могут содержать формулы, генерирующие запросы к движку Access.

Пользовательский интерфейс обращается к базе данных, подключаясь к нужному движку и вызывая его методы. Например, обратите внимание, что программа `ij` на самом деле является простым пользовательским интерфейсом. Ее команда `connect` устанавливает соединение с указанным движком базы данных, а каждая команда посылает оператор SQL движку, получает результаты и отображает их.

Механизмы баз данных обычно поддерживают несколько стандартных API. Когда Java-программа подключается к движку, она выбирает API, который называется JDBC. Глава 2 подробно обсуждает JDBC и показывает, как написать `ij`-подобное приложение с использованием JDBC.

Пользовательский интерфейс может подключаться к *встроенному* механизму базы данных или *серверному*. Когда используется встроенный механизм базы данных, движок действует в том же процессе, что и код пользовательско-

го интерфейса. Это дает пользовательскому интерфейсу эксклюзивный доступ к движку. Встроенный движок должен использоваться, только когда база данных «принадлежит» данному приложению и хранится на том же компьютере, где выполняется приложение. В иных случаях приложения должны использовать серверные движки.

Когда используется соединение с сервером, код движка базы данных выполняется внутри специальной серверной программы. Эта серверная программа работает постоянно, ожидая запросов на соединение от клиентов, и не обязательно должна находиться на том же компьютере, что и ее клиенты. Когда клиент установит соединение с сервером, он посылает ему JDBC-запросы и получает ответы.

К серверу могут подключиться сразу несколько клиентов. Пока сервер обрабатывает запрос одного клиента, другие клиенты могут посылать свои запросы. В серверной программе имеется *планировщик*, который ставит запросы в очередь ожидания обслуживания и определяет, когда они будут обработаны. Никто из клиентов не знает о существовании других клиентов, и каждый из них полагает (если не учитывать задержки, вызванные работой планировщика), что сервер имеет дело исключительно с ним.

Сеанс `ij` в листинге 1.2 подключается ко встроенному движку. Он создал базу данных «ijtest» на том же компьютере, где выполняется, не обращаясь ни к какому серверу. Чтобы выполнить аналогичные действия на сервере, необходимо следующее: запустить движок Derby как сервер и изменить команду `connect` так, чтобы она ссылалась на сервер.

Код серверного движка Derby находится в Java-классе `NetworkServerControl`, в пакете `org.apache.derby.drda`. Чтобы запустить сервер из Eclipse, откройте диалог Run Configurations (Запуск конфигурации) в меню Run (Выполнить). Добавьте в проект Derby новую конфигурацию с именем «Derby Server». В поле с именем основного класса введите «org.apache.derby.drda.NetworkServerControl». На вкладке Arguments (Аргументы) введите аргумент «start -h localhost». После запуска конфигурации должно появиться окно консоли, сообщающее, что сервер Derby запущен.

Что означает аргумент «start -h localhost»? Первое слово – это команда «start», сообщающая классу, что тот должен запустить сервер. Остановить сервер можно, выполнив тот же класс с аргументом «shutdown» (или просто завершив процесс в окне консоли). Строка «-h localhost» указывает, что сервер должен принимать запросы только от клиентов, действующих на одном с ним компьютере. Если заменить «localhost» доменным именем или IP-адресом, сервер будет принимать запросы только от компьютера с этим именем или IP-адресом. Если указать IP-адрес «0.0.0.0», сервер будет принимать запросы от любых компьютеров[1].

Строка подключения к серверу должна определять сеть или IP-адрес сервера. Например, взгляните на следующие команды `connect`:

---

[1] Имейте в виду, что, разрешая подключаться к серверу любым клиентам, вы открываете базу данных не только для законопослушных пользователей, но и для злоумышленников. Обычно подобные серверы размещают за брандмауэром или включают механизм аутентификации Derby или и то, и другое.

```
ij> connect 'jdbc:derby:ijtest'
ij> connect 'jdbc:derby://localhost/ijtest'
ij> connect 'jdbc:derby://cs.bc.edu/ijtest'
```

Первая команда подключается ко встроенному движку базы данных «ijtest». Вторая устанавливает соединение с серверным движком базы данных «ijtest», действующим на компьютере «localhost», то есть на локальной машине. Третья команда устанавливает соединение с серверным движком базы данных «ijtest», действующим на удаленном компьютере «cs.bc.edu».

Обратите внимание, что строка подключения однозначно определяет выбор встроенного или серверного движка. Например, вернемся к листингу 1.2. Мы можем изменить сеанс и использовать подключение к серверу, просто изменив строку подключения в команде `connect`. Другие команды в сеансе не требуют изменений.

1.4. Система баз данных SimpleDB

Derby – это сложная, полноценная система баз данных, поэтому ее исходный код трудно понять или изменить. В противовес Derby я написал систему баз данных SimpleDB. Ее реализация уместилась в небольшой объем кода, который легко читается и легко модифицируется. В ней отсутствует вся необязательная функциональность, реализовано ограниченное подмножество языка SQL и используются только самые простые (и часто неоптимальные) алгоритмы. Ее цель – помочь вам получить четкое представление о каждом компоненте механизма базы данных и о взаимодействиях между ними.

Последнюю версию SimpleDB можно загрузить с веб-сайта cs.bc.edu/~sciore/simpledb. Загруженный файл архива следует распаковать в папку `SimpleDB_3.x`, после этого в ней появятся каталоги `simpledb`, `simpleclient` и `derbyclient`. В папке `simpledb` находится код механизма базы данных. В отличие от Derby, этот код не упакован в архив JAR, поэтому все файлы находятся непосредственно в папке.

Чтобы установить движок SimpleDB, добавьте папку `simpledb` в путь поиска классов classpath. Для этого в Eclipse создайте новый проект с именем «SimpleDB Engine», затем скопируйте содержимое каталога `simpledb` из папки `SimpleDB_3.x` в каталог `src` проекта и обновите проект в Eclipse, выбрав пункт Refresh (Обновить) в меню File (Файл).

В папке `derbyclient` находятся примеры программ, использующих движок Derby. Скопируйте содержимое этой папки (не саму папку) в папку `src` проекта Derby и обновите его. Эти программы мы будем рассматривать в главе 2.

В папке `simpleclient` находятся примеры программ, использующих движок SimpleDB. Для экспериментов с ними создайте новый проект с именем «SimpleDB Clients». Чтобы эти программы смогли найти код движка SimpleDB, добавьте проект «Engine SimpleDB» в путь сборки проекта «SimpleDB Clients». Затем скопируйте содержимое папки `simpleclient` в каталог `src` проекта «SimpleDB Clients».

SimpleDB поддерживает оба типа движков – встроенный и серверный. В папке `simpleclient` можно найти программу `SimpleIJ`, которая является упрощенной версией программы `ij` из Derby. Одно из отличий от `ij` состоит в том, что `SimpleIJ` позволяет подключиться к движку только один раз, в начале сеанса. После запуска программа предложит вам ввести строку подключения. Синтаксис строки подключения аналогичен синтаксису в `ij`, например:

```
jdbc:simpledb:testij
jdbc:simpledb://localhost
jdbc:simpledb://cs.bc.edu
```

Первая строка подключения описывает подключение к базе данных «testij» через встроенный движок. Так же как при использовании Derby, база данных должна находиться в каталоге программы – в данном случае клиента из проекта «SimpleDB Clients». В отличие от Derby, SimpleDB в любом случае создаст базу данных, если она отсутствует, поэтому нет необходимости добавлять флаг «create = true».

Вторая и третья строки подключения ссылаются на удаленную базу данных и используют для подключения серверный движок SimpleDB, действующий на локальной машине или на сервере `cs.bc.edu`. В отличие от Derby, эти строки подключения не описывают базу данных. Причина в том, что движок SimpleDB может обслуживать только одну базу данных, которая была определена в момент его запуска.

После выполнения каждой команды программа `SimpleIJ` снова выводит приглашение к вводу, предлагая ввести следующую команду. В отличие от Derby, вся команда целиком должна находиться в одной строке и завершаться точкой с запятой. После ввода команды программа выполнит ее. Если команда является запросом, в ответ выводится таблица с результатами. Если команда произвела какие-то изменения в базе данных, тогда выводится количество затронутых записей. Если команда содержит ошибку, будет выведено сообщение об ошибке. Движок SimpleDB поддерживает очень ограниченное подмножество языка SQL и генерирует исключение, если ему не удалось распознать команду. Эти ограничения описаны в следующем разделе.

Движок SimpleDB может действовать в режиме сервера. Основным классом в этом случае является `StartServer` из пакета `simpledb.server`. Чтобы запустить сервер из Eclipse, откройте диалог Run Configurations (Запуск конфигурации) в меню Run (Выполнить). Добавьте новую конфигурацию в проект «SimpleDB Engine» с названием «SimpleDB Server». В поле, определяющее главный класс, введите «simpledb.server.StartServer». На вкладке Arguments (Аргументы) введите имя базы данных. По умолчанию, в отсутствие аргумента, сервер использует базу данных «studentdb». После запуска конфигурации должно появиться окно консоли, сообщающее, что сервер SimpleDB запущен.

Сервер SimpleDB принимает соединения от любых клиентов, по аналогии с сервером Derby, запущенным с ключом «-h 0.0.0.0». Остановить сервер можно, только принудительно прервав его работу в консоли.

1.5. Версия SQL, поддерживаемая в SimpleDB

Движок Derby поддерживает почти весь стандартный набор операторов SQL. В отличие от Derby, SimpleDB реализует лишь небольшую часть стандарта и накладывает некоторые дополнительные ограничения. В этом разделе я лишь кратко перечислю эти ограничения. Более подробно они будут описаны в других главах книги, а в отдельных упражнениях, что приводятся в конце каждой главы, вам будет предложено реализовать некоторые недостающие функции.

Запросы в SimpleDB могут включать только предложения *select-from-where*, где `select` содержит список имен полей (без ключевого слова `AS`), а предложение `from` – список имен таблиц (без переменных области значений).

Выражения в необязательном предложении `where` можно объединять только с помощью логического оператора `and`. Выражения могут проверять лишь равенство полей константам. Движок SimpleDB не поддерживает другие стандартные операторы сравнения, логические и арифметические операторы и встроенные функции, а также скобки. Как следствие не поддерживаются вложенные запросы, агрегирование и вычисляемые значения.

Поскольку переменные области значений и псевдонимы полей не поддерживаются, все имена полей в запросе должны быть уникальными. А так как не поддерживаются предложения `group by` и `order by`, в запросах нельзя организовать группировку и сортировку. Вот еще некоторые ограничения:

- сокращенная форма «*» определения списка полей в предложении `select` не поддерживается;
- не поддерживаются неопределенные (NULL) значения;
- не поддерживаются явные и внешние соединения в предложении `from`;
- не поддерживается ключевое слово `union`;
- инструкция `insert` принимает только явные значения, то есть вставляемое значение нельзя определить с использованием вложенного запроса;
- инструкция `update` принимает лишь один оператор присваивания в предложении `set`.

1.6. Итоги

- *База данных* – это коллекция данных, хранящаяся на компьютере. Данные в базе обычно организованы в *записи*. *Система баз данных* – это программное обеспечение, управляющее записями в базе данных.
- Система баз данных должна быть способна обрабатывать большие базы данных, хранящиеся в медленной долговременной памяти, а также предоставлять высокоуровневый интерфейс для доступа к данным и *гарантировать* безошибочное хранение информации, разрешая конфликты между изменениями, вносимыми пользователями, и восстанавливая после сбоев системы. Системы баз данных отвечают этим требованиям благодаря наличию следующих функций:
  - хранение записей в файлах с использованием форматов, обеспечивающих более высокую эффективность доступа, чем позволяют стандартные средства файловой системы;
  - сложные алгоритмы индексирования для поддержки быстрого доступа;
  - возможность одновременно обслуживать нескольких пользователей с блокированием операций при необходимости;
  - поддержка фиксации и отката изменений;
  - кеширование записей из базы данных в оперативной памяти и управление синхронизацией между постоянной и оперативной версиями базы данных, с возможностью восстановления данных до состояния, предшествовавшего сбою;

- компилятор/интерпретатор языка для преобразования пользовательских запросов в выполняемый код;
- поддержка стратегий оптимизации запросов для преобразования неэффективных запросов в более эффективные.

- *Механизм (движок) базы данных* – это компонент системы баз данных, обеспечивающий доступ к данным и их хранение. Приложение базы принимает ввод пользователя и выводит результаты, для получения которых вызывает движок базы данных.
- Движок базы данных может быть встроенным в приложение или действовать как сервер. Программа со встроенным движком имеет эксклюзивный доступ к базе данных. Программа, соединяющаяся с сервером, использует движок вместе с другими программами, выполняющимися параллельно.
- *Derby* и *SimpleDB* – это две системы баз данных, написанные на Java. Derby реализует весь стандарт SQL, а SimpleDB – только ограниченное подмножество. SimpleDB можно использовать как учебный пример благодаря простому и понятному коду. В остальной части книги, начиная с главы 3, мы последовательно исследуем этот код.

1.7. Для дополнительного чтения

Системы баз данных претерпели кардинальные изменения за эти годы. Подробный перечень этих изменений можно найти в главе 6 National Research Council (1999) и Haigh (2006). Желающие также могут прочитать статью в Википедии: en.wikipedia.org/wiki/Database_management_system#History[1].

Парадигма клиент-сервер с успехом используется во многих областях, а не только в базах данных. Общий обзор можно найти в Orfali et al. (1999). Описание функций и параметров конфигурации сервера Derby можно найти по адресу: db.apache.org/derby/manuals/index.html.

Haigh, T. (2006). «A veritable bucket of facts». Origins of the data base management system. ACM SIGMOD Record, 35 (2), 33–49.

National Research Council Committee on Innovations in Computing and Communications. (1999). «Funding a revolution». National Academy Press. Доступен по адресу: www.nap.edu/read/6323/chapter/8#159.

Orfali, R., Harkey, D., & Edwards, J. (1999). «Client/server survival guide (3rd ed.)». Wiley.

1.8. Упражнения

Теория

1.1. Представьте, что в вашей организации возникла потребность управлять относительно небольшим набором записей (например, 100 или около того) и обеспечить общий доступ к ним.

а) Имеет ли смысл использовать для этого коммерческую систему баз данных?

[1] ru.wikipedia.org/wiki/База_данных#История. – *Прим. перев.*

b) Какие возможности системы баз данных могут остаться невостребованными?
c) Разумно ли использовать электронную таблицу для хранения этих записей? Какие проблемы могут при этом возникнуть?

1.2. Представьте, вам потребовалось организовать хранение большого объема личных данных в базе. Какие возможности системы баз данных вам не понадобятся?

1.3. Представьте, что у вас есть некоторые данные, для хранения которых вы не используете систему баз данных (например, список покупок, адресная книга, сведения о вкладах в банке и т. д.).
a) Насколько большим должен стать объем таких данных, чтобы вы наконец решили сохранить их в базе данных?
b) Какие изменения в вашем подходе к использованию этих данных подтолкнули бы вас к перемещению их в систему баз данных?

1.4. Если вы знакомы с особенностями систем управления версиями (например, Git или Subversion), сравните их возможности с возможностями систем баз данных.
a) Есть ли в системе управления версиями понятие записи?
b) Как операции извлечения/отправки версий соответствуют управлению параллельным доступом в базе данных?
c) Как выполняется фиксация изменений? Как производится отмена незафиксированных изменений?
d) Многие системы управления версиями сохраняют изменения в виде *файлов различий*, имеющих небольшой размер и описывающих, как преобразовать предыдущую версию файла с кодом в новую. Когда пользователь запрашивает текущую версию файла, система извлекает исходную версию и применяет к ней все файлы различий. Насколько хорошо эта стратегия удовлетворяет потребностям систем баз данных?

Практика

1.5. Выясните, используется ли в вашем учебном заведении или организации система баз данных. Если да:
a) кто из сотрудников напрямую использует систему баз данных в своей работе? (К их числу не относятся сотрудники, запускающие программы, которые используют базу данных без их ведома.) Для чего они ее используют?
b) когда пользователю требуется сделать с данными что-то, чего прежде он не делал, кто пишет запрос? Он сам или кто-то?

1.6. Установите и запустите серверы Derby и SimpleDB.
a) Запустите программы `ij` и `SimpleIJ` на компьютере, играющем роль сервера.
b) Если у вас есть второй компьютер, попробуйте запустить на нем демонстрационные клиентские программы и подключиться к серверу с этого компьютера.

Глава 2

JDBC

Приложения взаимодействуют с движком базы данных, вызывая его методы. Для этой цели Java-приложения используют программный интерфейс *JDBC* (от Java DataBase Connectivity – Java-интерфейс подключения к базам данных). Библиотека JDBC состоит из пяти пакетов, большинство из которых предоставляют расширенные функции, полезные только в крупных коммерческих приложениях. В этой главе мы рассмотрим только базовые возможности JDBC, реализованные в пакете `java.sql`. Эти базовые возможности можно разделить на две части: *ядро JDBC*, содержащее классы и методы, необходимые для элементарного использования, и *расширенные инструменты JDBC*, к которым относятся функции, обеспечивающие дополнительное удобство и гибкость.

2.1. Ядро JDBC

Базовая функциональность JDBC сосредоточена в пяти интерфейсах: `Driver`, `Connection`, `Statement`, `ResultSet` и `ResultSetMetadata`. Однако далеко не все методы этих интерфейсов играют важную роль. Наиболее значимые методы перечислены в листинге 2.1.

Примеры программ в этом разделе демонстрируют применение данных методов. Первая из этих программ – `CreateTestDB` – иллюстрирует подключение к движку Derby и отключение от него. Ее код приводится в листинге 2.2, при этом код, использующий JDBC, выделен жирным. В следующих подразделах мы подробно рассмотрим этот код.

Листинг 2.1. Программные интерфейсы ядра JDBC

Driver

```
public Connection connect(String url, Properties prop)
                                    throws SQLException;
```

Connection

```
public Statement createStatement() throws SQLException;
public void      close()           throws SQLException;
```

Statement

```
public ResultSet executeQuery(String qry)  throws SQLException;
public int       executeUpdate(String cmd) throws SQLException;
public void      close()                   throws SQLException;
```

ResultSet

```
public boolean   next()                throws SQLException;
public int       getInt()              throws SQLException;
public String    getString()           throws SQLException;
public void      close()               throws SQLException;
public ResultSetMetaData getMetaData() throws SQLException;
```

ResultSetMetaData

```
public int        getColumnCount()          throws SQLException;
public String     getColumnName(int column) throws SQLException;
public int        getColumnType(int column) throws SQLException;
public int getColumnDisplaySize(int column) throws SQLException;
```

Листинг 2.2. Клиент CreateTestDB, использующий JDBC

```
import java.sql.Driver;
import java.sql.Connection;
import org.apache.derby.jdbc.ClientDriver;

public class CreateTestDB {
    public static void main(String[] args) {
        String url = "jdbc:derby://localhost/testdb;create=true";
        Driver d = new ClientDriver();
        try {
            Connection conn = d.connect(url, null);
            System.out.println("Database Created");
            conn.close();
        }
        catch(SQLException e) {
            e.printStackTrace();
        }
    }
}
```

2.1.1. Подключение к движку базы данных

Все движки баз данных имеют свои (иногда патентованные) механизмы подключения клиентов. Однако большинство клиентов стремятся оставаться максимально независимыми от особенностей реализации сервера – не изучать в мельчайших деталях порядок подключения к движку, а просто знать, к какому классу обратиться. Такие классы называются *драйверами*.

Классы драйверов JDBC реализуют интерфейс `Driver`. Derby и SimpleDB имеют по два класса драйверов: один устанавливает соединение с серверным движком, а другой – со встроенным. Для соединения с серверным движком Derby используется класс `ClientDriver`, а со встроенным – класс `EmbeddedDriver`. Оба класса находятся в пакете `org.apache.derby.jdbc`. Для соединения с серверным движком SimpleDB используется класс `NetworkDriver` (из пакета `simpledb.jdbc.network`), а со встроенным – класс `EmbeddedDriver` (из пакета `simpledb.jdbc.embedded`).

Клиент подключается к движку базы данных, вызывая метод `connect` объекта `Driver`. Например, следующие три строки из листинга 2.2 устанавливают соединение с серверным движком базы данных Derby:

```
String url = "jdbc:derby://localhost/testdb;create=true";
Driver d = new ClientDriver();
Connection conn = d.connect(url, null);
```

Метод connect принимает два аргумента. Первый аргумент – это URL, идентифицирующий драйвер, сервер (при подключении к серверу) и базу данных. Этот URL называют *строкой подключения*. Он имеет тот же синтаксис, что и строка подключения к серверному движку в программе ij (или SimpleIJ), как рассказывалось в главе 1. Строка подключения в листинге 2.2 состоит из четырех частей:

- подстрока «jdbc:derby:» описывает протокол и движок, используемые клиентом. В данном случае клиент сообщает, что собирается подключиться к движку Derby и использовать протокол JDBC;
- подстрока «//localhost» описывает компьютер, на котором выполняется сервер. Вместо *localhost* можете указать любое доменное имя или IP-адрес;
- подстрока «/testdb» описывает путь к базе данных на сервере. Для сервера Derby путь начинается с текущего каталога, в котором он был запущен. Конец пути (здесь «testdb») – это имя каталога, где хранятся все файлы с данными для этой базы;
- остальная часть строки соединения определяет параметры для передачи движку. Здесь подстрока «;create = true» указывает, что движок должен создать новую базу данных. Движок Derby поддерживает несколько разных параметров. Например, если движок настроен на аутентификацию пользователя, ему также необходимо передать параметры username и password. Вот как могла бы выглядеть строка подключения для пользователя «einstein»:

```
"jdbc:derby://localhost/testdb;create=true;user=einstein;password=emc2"
```

Второй аргумент метода connect – объект типа Properties. Этот объект предоставляет другой способ передать дополнительные параметры в движок. В листинге 2.2 в этом аргументе передается null, потому что все необходимые параметры уже определены в строке подключения. В альтернативной реализации параметры можно было бы поместить во второй аргумент, например:

```
String url = "jdbc:derby://localhost/testdb";
Properties prop = new Properties();
prop.put("create", "true");
prop.put("username", "einstein");
prop.put("password", "emc2");
Driver d = new ClientDriver();
Connection conn = d.connect(url, prop);
```

Каждый движок базы данных определяет свой синтаксис строки подключения. Строка подключения к серверу для SimpleDB отличается от Derby – в ней достаточно указать только протокол и имя компьютера. (Бессмысленно указывать в строке подключения имя базы данных, потому что оно определяется в настройках запуска сервера SimpleDB. Точно так же бессмысленно включать в строку какие-либо параметры, поскольку сервер SimpleDB не поддерживает дополнительных параметров.) Для примера ниже приводятся три строки кода, устанавливающие соединение с сервером SimpleDB:

```
String url = "jdbc:simpledb://localhost";
Driver d = new NetworkDriver();
Connection conn = d.connect(url, null);
```

Имя класса драйвера и синтаксис строки соединения зависят от производителя, но остальная часть JDBC-программы – нет. Например, взгляните на переменные `d` и `conn` в листинге 2.2. Они имеют JDBC-типы `Driver` и `Connection`, которые являются интерфейсами. Глядя на этот код, можно сказать, что в переменную `d` записывается ссылка на объект `ClientDriver`. Однако переменная `conn` получает ссылку на объект `Connection`, возвращаемую методом `connect`, поэтому нет никакой возможности узнать его фактический класс. Эта ситуация характерна для всех программ JDBC. Кроме имени класса драйвера и синтаксиса его строки подключения, программа JDBC знает только интерфейсы JDBC, не зависящие от производителя, и использует лишь их. Соответственно, в простейшем случае клиент JDBC должен импортировать два пакета:

- встроенный пакет `java.sql`, чтобы получить доступ к определениям универсального интерфейса JDBC;
- пакет производителя, содержащий класс драйвера.

2.1.2. Отключение от движка базы данных

Пока клиент остается подключенным к движку базы данных, для его обслуживания могут резервироваться некоторые ресурсы. Например, клиент может запросить сервер заблокировать доступ к некоторым частям базы данных для других клиентов. Даже само соединение с движком является ресурсом. Компания может иметь лицензию на использование коммерческой системы баз данных, ограничивающую количество одновременных подключений, а это значит, что удержание соединения открытым может лишить другого клиента возможности подключиться к базе данных. Поскольку соединения могут быть ценным ресурсом, клиенты должны отключаться от движка, закончив работу с базой данных. Отключение от движка производится вызовом метода `close` объекта `Connection`. Этот вызов можно увидеть в листинге 2.2.

2.1.3. Исключения SQL

В процессе взаимодействий клиента с движком базы данных иногда могут возникать исключения. Например:

- клиент передал движку неправильно сформированный оператор или запрос, обращающийся к несуществующей таблице или сравнивающий два несовместимых значения;
- движок прервал выполнение оператора или запроса из-за конфликта с действиями другого клиента, обслуживаемого параллельно;
- возникла ошибка в коде движка;
- отсутствует доступ к движку (серверному), возможно из-за неправильно указанного имени хоста или его недоступности.

Разные движки по-разному обрабатывают эти ситуации. Например, при возникновении проблем с сетью SimpleDB генерирует исключение `RemoteException`, при возникновении проблем с оператором SQL – исключение `BadSyntaxException`, в случае взаимоблокировки – `BufferAbortException` или `LockAbortException`, а при возникновении проблем на сервере – обобщенное исключение `RuntimeException`.

Чтобы обработка исключений не зависела от производителя, JDBC предоставляет собственный класс исключений `SQLException`. Когда движок базы данных сталкивается с внутренним исключением, он заворачивает его в исключение `SQLException` и передает клиентской программе.

Внутреннее исключение идентифицируется строкой сообщения, связанной с исключением `SQLException`. Каждый движок базы данных может передавать свои уникальные сообщения. Например, в Derby имеется почти 900 сообщений об ошибках, тогда как в SimpleDB все возможные проблемы объединены в шесть сообщений: «проблема сети», «недопустимый оператор SQL», «ошибка сервера», «операция не поддерживается» и две формы сообщения «транзакция прервана».

Большинство методов JDBC (и все методы в листинге 2.1) генерируют исключение `SQLException`. Это исключение относится к категории *проверяемых*, то есть клиенты должны явно обрабатывать их: перехватывать или передавать дальше. В листинге 2.2 можно видеть, что два метода JDBC вызываются внутри блока `try`; если любой из них сгенерирует исключение, код выведет трассировку стека и завершится.

Обратите внимание, что код в листинге 2.2 имеет проблему – он не закрывает соединение при появлении исключения. Это пример *утечки ресурсов* – движок не в состоянии легко и быстро освободить ресурсы, зарезервированные за соединением, после завершения клиента. Одно из возможных решений – закрыть соединение в блоке `catch`. Проблема в том, что метод `close` должен вызываться внутри блока `try`, то есть фактически блок `catch` должен выглядеть так:

```
catch(SQLException e) {
    e.printStackTrace();
    try {
        conn.close();
    }
    catch (SQLException ex) {}
}
```

Это решение выглядит некрасиво. Кроме того, возникает вопрос: что должен делать клиент, если метод `close` сгенерирует исключение? В коде выше такая вероятность просто игнорируется, но это не совсем правильно.

Более удачное решение – позволить Java автоматически закрыть соединение, используя для этого синтаксис *try-с-ресурсами*. Для этого объект `Connection` нужно создать в круглых скобках после ключевого слова `try`. Когда блок `try` завершится (в ходе нормального выполнения или из-за исключения), Java неявно вызовет метод `close` объекта. Вот как выглядит улучшенный вариант блока `try` из листинга 2.2:

```
try (Connection conn = d.connect(url, null)) {
    System.out.println("Database Created");
}
catch (SQLException e) {
    e.printStackTrace();
}
```

Этот код правильно обрабатывает все исключения, оставаясь таким же простым, как код в листинге 2.2.

2.1.4. Выполнение операторов SQL

Соединение можно рассматривать как «сеанс» работы с движком базы данных, во время которого движок выполняет операторы SQL для клиента. Далее описывается, как JDBC поддерживает эту идею.

Объект `Connection` имеет метод `createStatement`, который возвращает объект `Statement`. Объект `Statement` поддерживает два способа выполнения операторов SQL: методы `executeQuery` и `executeUpdate`. Он также имеет метод `close`, освобождающий ресурсы, занятые объектом.

В листинге 2.3 представлена клиентская программа, которая вызывает `executeUpdate` для изменения значения `MajorId` в записи, соответствующей студентке Amy, в таблице STUDENT. Метод принимает строковый аргумент – оператор `update` – и возвращает количество изменившихся записей.

Объект `Statement`, как и `Connection`, нужно закрывать. Проще всего организовать автоматическое закрытие обоих объектов в блоке `try`.

Обратите внимание, как определяется оператор SQL в листинге 2.3. Поскольку оператор передается в виде строки Java, он заключен в двойные кавычки. С другой стороны, для обозначения строк в SQL используются одинарные кавычки. Это различие облегчает нам жизнь, поскольку избавляет от беспокойства о кавычках, имеющих два разных значения: строки в SQL заключаются в одинарные кавычки, а строки Java – в двойные.

Листинг 2.3. JDBC-код для клиента ChangeMajor

```
public class ChangeMajor {
    public static void main(String[] args) {
        String url = "jdbc:derby://localhost/studentdb";
        String cmd = "update STUDENT set MajorId=30 where SName='amy'";

        Driver d = new ClientDriver();
        try ( Connection conn = d.connect(url, null);
              Statement stmt = conn.createStatement()) {
            int howmany = stmt.executeUpdate(cmd);
            System.out.println(howmany + " records changed.");
        }
        catch(SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Код клиента `ChangeMajor` предполагает, что база данных с именем «studentdb» уже существует. В дистрибутиве SimpleDB имеется класс `CreateStudentDB`, который создает базу данных и заполняет ее данными, представленным в табл. 1.1. Это первая программа, которую следует запустить, если вы собираетесь экспериментировать с университетской базой данных. Реализация этого класса показана в листинге 2.4. Он создает пять таблиц и добавляет в них записи, выполняя операторы SQL. Для краткости показан только код, заполняющий таблицу STUDENT.

Листинг 2.4. JDBC-код в реализации клиента CreateStudentDB

```
public class CreateStudentDB {
    public static void main(String[] args) {
        String url = "jdbc:derby://localhost/studentdb;create=true";
        Driver d = new ClientDriver();
        try (Connection conn = d.connect(url, null);
             Statement stmt = conn.createStatement()) {

            String s = "create table STUDENT(SId int,
                SName varchar(10), MajorId int, GradYear int)";
            stmt.executeUpdate(s);
            System.out.println("Table STUDENT created.");

            s = "insert into STUDENT(SId, SName,
                                     MajorId, GradYear) values ";
            String[] studvals = {"(1, 'joe', 10, 2021)",
                "(2, 'amy', 20, 2020)",
                "(3, 'max', 10, 2022)",
                "(4, 'sue', 20, 2022)",
                "(5, 'bob', 30, 2020)",
                "(6, 'kim', 20, 2020)",
                "(7, 'art', 30, 2021)",
                "(8, 'pat', 20, 2019)",
                "(9, 'lee', 10, 2021)"};
            for (int i=0; i<studvals.length; i++)
                stmt.executeUpdate(s + studvals[i]);
            System.out.println("STUDENT records inserted.");

            ...

        }
        catch(SQLException e) {
            e.printStackTrace();
        }
    }
}
```

2.1.5. Наборы результатов

Метод `executeQuery` выполняет запрос SQL. Он принимает строку с SQL-запросом и возвращает объект типа `ResultSet`. Объект `ResultSet` представляет набор результатов, возвращаемых в ответ на запрос. Клиент может выполнять поиск в наборе результатов или как-то иначе использовать его.

Для иллюстрации применения наборов результатов рассмотрим класс `StudentMajor`, показанный в листинге 2.5. Вызов `executeQuery` в нем возвращает набор результатов, содержащий имя студента и идентификационный номер основной кафедры. Последующий цикл `while` выводит каждую запись в наборе результатов.

Получив набор результатов, клиент может выполнить их перебор, вызывая метод `next`. Этот метод перемещается к следующей записи, возвращая значение `true`, если перемещение прошло успешно, и `false`, если достигнут конец набора. Обычно для перебора записей и их обработки клиенты используют циклы.

Новый объект `ResultSet` всегда устанавливает указатель *перед* первой записью, поэтому обязательно нужно вызвать метод `next`, чтобы получить первую за-

пись. Из-за этого требования типичный цикл перебора записей выглядит следующим образом:

```
String qry = "select ...";
ResultSet rs = stmt.executeQuery(qry);
while (rs.next()) {
    ... // обработать запись
}
```

Пример такого цикла можно увидеть в листинге 2.5. Во время выполнения n-й итерации переменная rs будет указывать на n-ю запись в наборе результатов. Цикл закончится, когда будет достигнут конец набора.

Обрабатывая записи, клиент использует методы getInt и getString, чтобы получить значения полей. Оба метода принимают аргумент с именем поля и возвращают его значение. Код в листинге 2.5 извлекает и выводит значения полей SName и DName в каждой записи.

Листинг 2.5. JDBC-код в реализации клиента StudentMajor

```
public class StudentMajor {
    public static void main(String[] args) {
        String url = "jdbc:derby://localhost/studentdb";
        String qry = "select SName, DName from DEPT, STUDENT "
            + "where MajorId = DId";

        Driver d = new ClientDriver();
        try ( Connection conn = d.connect(url, null);
              Statement stmt = conn.createStatement();
              ResultSet rs = stmt.executeQuery(qry)) {
            System.out.println("Name\tMajor");
            while (rs.next()) {
                String sname = rs.getString("SName");
                String dname = rs.getString("DName");
                System.out.println(sname + "\t" + dname);
            }
        }
        catch(SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Наборы результатов занимают ценные ресурсы в движке. Метод close освобождает эти ресурсы и делает их доступными для других клиентов. Поэтому клиент должен стремиться быть «добропорядочным гражданином» и как можно быстрее закрыть набор результатов. Для этого можно явно вызвать close, например в конце цикла while. Другой вариант – воспользоваться механизмом автоматического закрытия в Java, как показано в листинге 2.5.

2.1.6. Использование метаданных запросов

Схема набора результатов определяется именами, типами и отображаемыми размерами полей. Эта информация доступна через интерфейс ResultSetMetaData.

Когда клиент выполняет запрос, он обычно знает схему получающейся в результате таблицы. Например, код клиента StudentMajor «знает», что его набор результатов содержит два строковых поля – SName и DName.

А теперь представьте, что клиентская программа позволяет пользователям посылать произвольные запросы. Программа может вызвать метод getMetaData набора результатов, который возвращает объект типа ResultSetMetaData, и уже с его помощью определить схему таблицы с результатами. Например, код в листинге 2.6 использует ResultSetMetaData для вывода схемы набора результатов, переданного в аргументе.

Листинг 2.6. Использование ResultSetMetaData для вывода схемы набора результатов

```
void printSchema(ResultSet rs) throws SQLException {
    ResultSetMetaData md = rs.getMetaData();
    for(int i=1; i <= md.getColumnCount(); i++) {
        String name   = md.getColumnName(i);
        int size      = md.getColumnDisplaySize(i);
        int typecode  = md.getColumnType(i);
        String type;
        if (typecode == Types.INTEGER)
            type = "int";
        else if (typecode == Types.VARCHAR)
            type = "string";
        else
            type = "other";
        System.out.println(name + "\t" + type + "\t" + size);
    }
}
```

Этот код иллюстрирует типичное использование объекта ResultSetMetaData. Сначала вызывается метод getColumnCount, чтобы получить количество полей в наборе; затем вызываются методы getColumnName, getColumnType и getColumnDisplaySize, чтобы определить имя, тип и размер каждого поля. Обратите внимание, что нумерация столбцов начинается с 1, а не с 0.

Метод getColumnType возвращает целое число – код типа поля. Коды определены как константы в JDBC-классе Types. Этот класс определяет коды для 30 разных типов, что позволяет получить представление о том, насколько обширен язык SQL. Фактические значения этих кодов не важны, потому что программа JDBC всегда должна ссылаться на коды по именам, а не по значениям.

Хорошим примером клиента, использующего метаданные, может служить интерпретатор команд. Программа SimpleIJ, упоминавшаяся в главе 1, как раз является таким клиентом; ее код показан в листинге 2.7. Поскольку это наш первый пример нетривиального клиента JDBC, внимательно изучите его код.

Сначала метод main читает строку подключения, указанную пользователем, и с ее помощью определяет подходящий драйвер. Для этого выполняется поиск последовательности символов «//» в строке подключения. Если символы присутствуют, значит, строка определяет соединение с сервером, иначе – соединение со встроенным движком. Затем метод устанавливает соединение, передавая строку подключения в метод connect соответствующего драйвера.

В каждой итерации цикла while метод main обрабатывает одну текстовую строку. Если строка содержит оператор SQL, вызывается метод doQuery или doUpdate, в зависимости от оператора. Пользователь может прервать цикл, введя команду «exit», после чего программа завершится.

Листинг 2.7. JDBC-код в реализации клиента SimpleIJ

```
public class SimpleIJ {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Connect> ");
        String s = sc.nextLine();
        Driver d = (s.contains("//")) ? new NetworkDriver()
                                      : new EmbeddedDriver();

        try (Connection conn = d.connect(s, null);
             Statement stmt = conn.createStatement()) {
            System.out.print("\nSQL> ");
            while (sc.hasNextLine()) {
                // Обрабатывает введенные строки по одной
                String cmd = sc.nextLine().trim();
                if (cmd.startsWith("exit"))
                    break;
                else if (cmd.startsWith("select"))
                    doQuery(stmt, cmd);
                else
                    doUpdate(stmt, cmd);
                System.out.print("\nSQL> ");
            }
        }
        catch (SQLException e) {
            e.printStackTrace();
        }
        sc.close();
    }

    private static void doQuery(Statement stmt, String cmd) {
        try (ResultSet rs = stmt.executeQuery(cmd)) {
            ResultSetMetaData md = rs.getMetaData();
            int numcols = md.getColumnCount();
            int totalwidth = 0;

            // вывести заголовок
            for(int i=1; i<=numcols; i++) {
                String fldname = md.getColumnName(i);
                int width = md.getColumnDisplaySize(i);
                totalwidth += width;
                String fmt = "%" + width + "s";
                System.out.format(fmt, fldname);
            }
            System.out.println();
            for(int i=0; i<totalwidth; i++)
                System.out.print("-");
            System.out.println();
```

```
            // вывести записи
            while(rs.next()) {
                for (int i=1; i<=numcols; i++) {
                    String fldname = md.getColumnName(i);
                    int fldtype = md.getColumnType(i);
                    String fmt = "%" + md.getColumnDisplaySize(i);
                    if (fldtype == Types.INTEGER) {
                        int ival = rs.getInt(fldname);
                        System.out.format(fmt + "d", ival);
                    }
                    else {
                        String sval = rs.getString(fldname);
                        System.out.format(fmt + "s", sval);
                    }
                }
                System.out.println();
            }
        }
        catch (SQLException e) {
            System.out.println("SQL Exception: " + e.getMessage());
        }
    }

    private static void doUpdate(Statement stmt, String cmd) {
        try {
            int howmany = stmt.executeUpdate(cmd);
            System.out.println(howmany + " records processed");
        }
        catch (SQLException e) {
            System.out.println("SQL Exception: " + e.getMessage());
        }
    }
}
```

Метод `doQuery` выполняет запрос и получает набор результатов и его метаданные. Большая часть его кода занята правильным форматированием значений. Вызов `getColumnDisplaySize` возвращает ширину вывода для каждого поля; на основе этих чисел конструируются строки формата, обеспечивающие правильное отображение полей. Сложность кода лишний раз подчеркивает правоту принципа «дьявол кроется в деталях». То есть концептуально сложные задачи легко реализуются благодаря методам объектов `ResultSet` и `ResultSetMetaData`, тогда как для решения элементарных задач форматированного вывода требуются немалые усилия и значительный объем кода.

Столкнувшись с исключением, методы `doQuery` и `doUpdate` выводят сообщение об ошибке и возвращают управление. Такая стратегия обработки ошибок позволяет главному циклу продолжать принимать операторы, пока пользователь не введет команду «exit».

2.2. Дополнительные инструменты JDBC

Ядро JDBC не вызывает сложностей в использовании, но предлагает довольно ограниченный набор способов взаимодействия с движком базы данных. В этом разделе мы рассмотрим некоторые дополнительные инструменты JDBC, предлагающие более широкие возможности по управлению доступом к базе данных.

2.2.1. Сокрытие драйвера

Чтобы подключиться к движку базы данных с использованием ядра JDBC, клиент должен получить экземпляр объекта `Driver` и вызвать его метод `connect`. Проблема этой стратегии в том, что она вынуждает добавлять в клиентскую программу код, зависящий от производителя драйвера. Для хранения информации о драйверах в клиентских программах JDBC предлагает два независимых класса: `DriverManager` и `DataSource`. Рассмотрим их по очереди.

DriverManager

Класс `DriverManager` содержит коллекцию драйверов. Он имеет статические методы для добавления драйверов в коллекцию и поиска драйвера, подходящего для данной строки подключения. Два из этих методов показаны в листинге 2.8.

Идея состоит в том, чтобы клиент зарегистрировал драйверы всех баз данных, которые он может использовать, многократно вызывая метод `registerDriver`. Когда придет время подключиться к базе данных, ему останется только вызвать метод `getConnection` и передать ему строку подключения. Диспетчер драйверов (`DriverManager`) попробует передать эту строку каждому драйверу в коллекции, пока один из них не вернет установленное соединение.

Например, рассмотрим код в листинге 2.9. Первые две строки регистрируют драйверы для подключения к серверным движкам Derby и SimpleDB. Последние две строки устанавливают соединение с сервером Derby. Клиенту не нужно явно указывать драйвер в вызове `getConnection` – только строку подключения, а диспетчер драйверов сам определит, какой из зарегистрированных драйверов использовать.

Листинг 2.8. Два метода в классе DriverManager

```
static public void registerDriver(Driver driver)
                                  throws SQLException;
static public Connection getConnection(String url, Properties p)
                                  throws SQLException;
```

Листинг 2.9. Подключение к серверу Derby с помощью DriverManager

```
DriverManager.registerDriver(new ClientDriver());
DriverManager.registerDriver(new NetworkDriver());
String url   = "jdbc:derby://localhost/studentdb";
Connection c = DriverManager.getConnection(url);
```

Пример использования `DriverManager` в листинге 2.9 не особенно соответствует нашим желаниям, потому что фактически информация о драйверах не была скрыта – она явно присутствует в вызовах `registerDriver`. JDBC решает эту проблему, позволяя указывать драйверы в файле системных свойств Java. Например, драйверы Derby и SimpleDB можно зарегистрировать, добавив в файл следующую строку:

```
jdbc.drivers = org.apache.derby.jdbc.ClientDriver:simpledb.remote.NetworkDriver
```

Размещение информации о драйверах в файле свойств позволяет элегантно удалить информацию о драйверах из клиентского кода. Изменяя этот файл, можно менять драйверы, используемые всеми клиентами JDBC, без перекомпиляции кода.

DataSource

Диспетчер драйверов позволяет скрывать драйверы от клиентов JDBC, но он не может скрыть строку подключения. В частности, строка подключения в примере выше содержит текст «jdbc:derby», явно указывающий тип драйвера. Не так давно в JDBC появился интерфейс `DataSource` в пакете `javax.sql`. В настоящее время его использование считается предпочтительной стратегией управления драйверами.

Объект `DataSource` инкапсулирует не только драйвер, но и строку подключения, что позволяет клиенту подключаться к движку базы данных, не заботясь о деталях подключения. Чтобы создать источник данных (`DataSource`) для подключения к Derby, необходимо использовать предоставляемые Derby классы `ClientDataSource` (для соединения с серверным движком) и `EmbeddedDataSource` (для соединения со встроенным движком). Оба класса реализуют интерфейс `DataSource`. Вот как в этом случае мог бы выглядеть код клиента:

```
ClientDataSource ds = new ClientDataSource();
ds.setServerName("localhost");
ds.setDatabaseName("studentdb");
Connection conn = ds.getConnection();
```

Все производители баз данных предоставляют свои собственные классы, реализующие `DataSource`. Поскольку эти классы определяются самими производителями, они могут инкапсулировать детали их драйверов, такие как имя и синтаксис строки подключения. При их использовании программа должна только указать необходимые значения.

Использование источника данных избавляет клиента от необходимости знать имя драйвера или синтаксис строки подключения. Однако этот класс все еще зависит от производителя, поэтому клиентский код не полностью независим от него. Эту проблему можно решить несколькими способами.

Одно из решений – сохранить объект `DataSource` в файл. Администратор базы данных может создать объект и, используя механизм сериализации Java, записать его в файл. Клиент в этом случае может прочитать файл и десериализовать его содержимое обратно в объект `DataSource`. Это решение напоминает использование файла свойств. Объект `DataSource`, хранящийся в файле, может использоваться любыми клиентами JDBC. А администратор базы данных может вносить изменения в источник данных, просто меняя содержимое файла.

Второе решение – использовать сервер имен (например, сервер JNDI) вместо файла. В этом случае администратор баз данных помещает объект `DataSource` на сервер имен, а клиенты запрашивают его с сервера. Учитывая, что серверы имен широко используются во многих вычислительных окружениях, это решение легко реализовать, однако обсуждение деталей выходит за рамки этой книги.

2.2.2. Явная обработка транзакций

Каждый клиент JDBC выполняет операции с базой данных в виде последовательности *транзакций*. Концептуально транзакция – это «единица работы», когда все операции с базой данных в рамках этой транзакции рассматриваются как единое целое. Например, если в течение транзакции произойдет сбой одной операции изменения, движок автоматически отменит все изменения, выполненные в этой транзакции.

После успешного завершения текущей единицы работы транзакция должна *фиксироваться*, или *подтверждаться*. Получив подтверждение транзакции, движок базы данных сохраняет все изменения и освобождает все ресурсы (например, блокировки), занятые этой транзакцией. Закончив фиксацию, движок может запустить новую транзакцию.

Если транзакция не может быть зафиксирована, она *откатывается*. Выполняя откат, движок базы данных отменяет все изменения, внесенные этой транзакцией, снимает блокировки и переходит в режим готовности запустить новую транзакцию. После фиксации или отката транзакция считается *завершенной*.

В базовом ядре JDBC транзакции запускаются неявно. Движок базы данных сам определяет границы каждой транзакции, решая, когда их следует зафиксировать или откатить. Такие транзакции называют *автоматически фиксируемыми*.

При использовании транзакций с автоматической фиксацией движок выполняет каждый оператор SQL в своей собственной транзакции. Он фиксирует транзакцию, если оператор завершился успешно, и откатывает ее в противном случае. Транзакция с оператором изменения завершается одновременно с завершением метода `executeUpdate`, а транзакция с запросом – когда клиент закроет набор результатов.

Транзакция накапливает блокировки, которые не снимаются, пока не произойдет фиксация или откат транзакции. Поскольку блокировки могут заставить ждать другие транзакции, для параллельной работы предпочтительнее использовать более короткие транзакции. Из этого следует, что клиенты, использующие транзакции с автоматической фиксацией, должны стараться как можно быстрее закрывать наборы с результатами.

Автоматическая фиксация – наиболее разумный режим по умолчанию для клиентов JDBC. Часто это оправдано, потому что каждый оператор SQL выполняется в отдельной транзакции и такие транзакции выполняются очень быстро. Однако иногда требуется, чтобы в одной транзакции выполнялось несколько операторов SQL.

Примером, когда автоматическая фиксация нежелательна, может служить ситуация, когда клиенту требуется выполнить два оператора одновременно. Например, рассмотрим код в листинге 2.10. Он сначала выполняет запрос, который извлекает все курсы. Затем просматривает набор результатов и для каждого курса спрашивает у пользователя – следует ли удалить его. Если пользователь ответил утвердительно, он выполняет SQL-оператор удаления.

Листинг 2.10. Код, действующий неправильно в режиме автоматической фиксации транзакций

```
DataSource ds = ...
Connection conn = ds.getConnection();
Statement stmt1 = conn.createStatement();
Statement stmt2 = conn.createStatement();
ResultSet rs = stmt1.executeQuery("select * from COURSE");
while (rs.next()) {
    String title = rs.getString("Title");
    boolean goodCourse = getUserDecision(title);
    if (!goodCourse) {
        int id = rs.getInt("CId");
        stmt2.executeUpdate("delete from COURSE where CId =" + id);
    }
}
rs.close();
```

Листинг 2.11. Еще пример кода, который может действовать неправильно в режиме автоматической фиксации транзакций

```
DataSource ds = ...
Connection conn = ds.getConnection();
Statement stmt = conn.createStatement();
String cmd1 = "update SECTION set Prof= 'brando' where SectId = 43";
String cmd2 = "update SECTION set Prof= 'einstein' where SectId = 53";
stmt.executeUpdate(cmd1);
// представьте, что в этот момент движок базы данных потерпел аварию
stmt.executeUpdate(cmd2);
```

Проблема в том, что этот клиент будет пытаться выполнить оператор удаления, пока набор результатов еще открыт. Поскольку соединение может иметь только одну активную транзакцию, текущую транзакцию запроса нужно зафиксировать, и только после этого появится возможность создать новую транзакцию для удаления. А если транзакцию зафиксировать после первого утвердительного ответа пользователя, программа не сможет продолжить обход оставшейся части набора результатов. Код либо сгенерирует исключение, либо будет действовать непредсказуемо[1].

Автоматическая фиксация также нежелательна при внесении сразу нескольких изменений в базу данных. Рассмотрим пример кода в листинге 2.11. Его цель – поменять имена преподавателей разделов 43 и 53. Однако база данных станет некорректной, если движок потерпит аварию после первого вызова `executeUpdate`, но перед вторым. В данном случае важно, чтобы оба оператора SQL выполнились в рамках одной транзакции и соответствующие им изменения были зафиксированы или отменены вместе.

Листинг 2.12. Метод Connection для явной обработки транзакций

```
public void setAutoCommit(boolean ac) throws SQLException;
public void commit()                  throws SQLException;
public void rollback()                throws SQLException;
```

Режим автоматической фиксации также может доставлять определенные неудобства. Представьте, что ваша программа добавляет новые записи, например загружая данные из текстового файла. Если во время работы программы произойдет сбой движка, часть записей будет добавлена в базу, а часть – нет. Чтобы потом определить, в какой момент произошла ошибка, и добавить остальные записи, могут потребоваться немалые усилия. Этого можно избежать, если выполнить все команды вставки в рамках одной транзакции. При

---

[1] Фактическое поведение кода зависит от свойства `holdability` набора результатов, значение по умолчанию которого определяется конкретным движком базы данных. Если параметру `holdability` присвоено значение `CLOSE_CURSORS_AT_COMMIT`, набор с результатами станет недействительным, и будет сгенерировано исключение. Если присвоено значение `HOLD_CURSORS_OVER_COMMIT`, набор результатов останется открытым, но установленные им блокировки будут сняты. Поведение набора результатов в этом случае непредсказуемо и напоминает действие режима изоляции read uncommitted (когда доступны для чтения неподтвержденные изменения), который будет обсуждаться в разделе 2.2.3.

таком подходе все они будут отменены в случае сбоя, и после возобновления работы движка вам останется просто запустить программу еще раз.

Интерфейс `Connection` имеет три метода, которые позволяют клиенту выбрать режим явной обработки транзакций. Они перечислены в листинге 2.12. Клиент может отключить автоматическую фиксацию вызовом `setAutoCommit(false)`, завершить текущую транзакцию и запустить новую вызовом `commit` или `rollback` по необходимости.

Когда клиент отключает автоматическую фиксацию, он берет на себя ответственность за откат операторов SQL, потерпевших неудачу. В частности, если во время транзакции возникнет исключение, клиент должен откатить ее в своем коде обработки исключений.

Для примера вернемся к неправильному фрагменту кода в листинге 2.10. Исправленная версия показана в листинге 2.13. Код вызывает `setAutoCommit` сразу после создания соединения и `commit` после выполнения операторов SQL. Блок `catch` содержит вызов метода `rollback`. Этот вызов должен находиться внутри своего блока `try`, так как он тоже может сгенерировать исключение.

На первый взгляд исключение во время выполнения `rollback` может повредить базу данных, подобно коду в листинге 2.11. К счастью, алгоритмы отката в базах данных учитывают такую возможность, но оставим обсуждение этих подробностей до главы 5. Таким образом, код в листинге 2.13 может на вполне законных основаниях игнорировать ошибку, возникшую во время отката, зная, что движок базы данных все исправит.

Листинг 2.13. Версия кода из листинга 2.10, явно обрабатывающая транзакции

```
DataSource ds = ...
try (Connection conn = ds.getConnection()) {
    conn.setAutoCommit(false);
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery("select * from COURSE");
    while (rs.next()) {
        String title = rs.getString("Title");
        boolean goodCourse = getUserDecision(title);
        if (!goodCourse) {
            int id = rs.getInt("CId");
            stmt.executeUpdate("delete from COURSE where CId =" + id);
        }
    }
    rs.close();
    stmt.close();
    conn.commit();
}
catch (SQLException e) {
    e.printStackTrace();
    try {
        if (conn != null)
            conn.rollback();
    }
    catch (SQLException e2) {}
}
```

2.2.3. Уровни изоляции транзакций

Сервер базы данных обычно обслуживает сразу несколько клиентов, каждый из которых выполняет свою собственную транзакцию. Параллельное выполнение транзакций позволяет серверу увеличить пропускную способность и уменьшить время отклика. То есть параллелизм – хорошая штука. Однако неуправляемый параллелизм может вызвать проблемы, потому что одни транзакции могут мешать другим, неожиданно изменяя данные, используемые этими другими транзакциями. Вот три примера, которые демонстрируют возможные проблемы.

Пример 1: чтение неподтвержденных данных

Вернемся к коду в листинге 2.11, который изменяет имена преподавателей для двух разделов курса, и предположим, что он выполняется в рамках одной транзакции (то есть с ручным управлением фиксацией). Пусть это будет транзакция T1. Предположим также, что университет решил премировать своих профессоров, исходя из количества разделов, которые они преподают. Пусть код, подсчитывающий количество разделов, преподаваемых каждым профессором, выполняется в транзакции T2. Кроме того, предположим, что эти две транзакции выполняются одновременно – T2 начинается и выполняется до конца, сразу после первого оператора `update` в транзакции T1. В результате профессор Брандо получит премию больше, чем заслуживает, за один дополнительный раздел, а профессор Эйнштейн – меньше, из-за неучтенного раздела.

Почему так получилось? Каждая транзакция в отдельности правильная, но вместе они приводят к неправильному начислению премии. Проблема в том, что T2 исходит из ошибочного предположения о *согласованности* записей, то есть она предполагает, что весь комплекс записей в целом правильно отражает действительное положение вещей. Однако данные, записанные незафиксированной (неподтвержденной) транзакцией, в какой-то момент могут оказаться в несогласованном (противоречивом) состоянии. В T1 несогласованность возникла в тот момент, когда было выполнено первое из двух изменений. Когда T2 в этот момент прочитала неподтвержденные измененные записи, несогласованность данных привела к неправильному результату вычислений.

Пример 2: неожиданное изменение существующей записи

Для этого примера предположим, что таблица STUDENT содержит поле `MealPlanBal` с суммой денег у студента на покупку еды в столовой.

Рассмотрим две транзакции в листинге 2.14. Транзакция T1 была запущена, когда студент Joe заплатил за обед 10 долларов. Транзакция выполняет запрос, чтобы узнать текущий баланс, проверяет, достаточно ли денег на счете, и уменьшает сумму на счете на стоимость обеда. Транзакция T2 была запущена, когда родители Joe выслали ему чек на 1000 долларов, и его сумму потребовалось добавить к сумме счета с деньгами для питания. Эта транзакция просто выполняет SQL-оператор `update`, увеличивая баланс студента Joe.

Листинг 2.14. Две конкурирующие транзакции могут «потерять» изменения, сделанные одной из них: (a) транзакция T1 уменьшает сумму, выделенную на питание, (b) транзакция T2 увеличивает сумму, выделенную на питание

(a)

```
DataSource ds = ...
Connection conn = ds.getConnection();
conn.setAutoCommit(false);
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("select MealPlanBal from STUDENT "
                                 + "where SId = 1");
rs.next();
int balance = rs.getInt("MealPlanBal");
rs.close();

int newbalance = balance - 10;
if (newbalance < 0)
    throw new NoFoodAllowedException("You cannot afford this meal");

stmt.executeUpdate("update STUDENT "
                + "set MealPlanBal = " + newbalance
                + " where SId = 1");
conn.commit();
```

(b)

```
DataSource ds = ...
Connection conn = ds.getConnection();
conn.setAutoCommit(false);
Statement stmt = conn.createStatement();
stmt.executeUpdate("update STUDENT "
                + "set MealPlanBal = MealPlanBal + 1000 "
                + "where SId = 1");
conn.commit();
```

Теперь представьте, что эти две транзакции выполняются одновременно, когда на счете у Joe имеется 50 долларов. В частности, предположим, что T2 запускается и полностью выполняется сразу после того, как T1 вызовет `rs.close`. Транзакция T2 завершается первой, увеличив баланс до 1050 долларов. Однако T1 не знает об этом изменении и все еще считает, что на счете находится 50 долларов. Она уменьшает эту сумму на 10 долларов, записывает на счет результат, равный 40 долларам, и завершается. В результате сумма перевода в 1000 долларов не зачисляется на счет, то есть изменение, произведенное транзакцией T2, теряется.

Проблема обусловлена ошибочным предположением транзакции T1 о том, что сумма на счете со средствами для питания не изменится между моментом, когда она прочитала значение, и моментом, когда она изменила его. Формально это предположение называется *повторяемым чтением* (repeatable read), поскольку транзакция предполагает, что повторное чтение элемента из базы данных всегда будет возвращать то же самое значение.

Пример 3: неожиданное изменение числа записей

Предположим, что в прошлом году университетские столовые получили прибыль в размере 100 000 долларов. В руководстве университета посчитали, что неправильно заставлять студентов переплачивать, и решили разделить прибыль поровну между ними. То есть если в настоящее время в университете учится 1000 студентов, то каждому из них на счет со средствами для питания следует перечислить 100 долларов. Код, реализующий эту операцию, показан в листинге 2.15.

Листинг 2.15. Транзакция, которая может потратить на компенсации больше денег, чем планировалось

```
DataSource ds = ...
Connection conn = ds.getConnection();
conn.setAutoCommit(false);
Statement stmt = conn.createStatement();
String qry = "select count(SId) as HowMany from STUDENT "
           + "where GradYear >= extract(year, current_date)"
ResultSet rs = stmt.executeQuery(qry);
rs.next();
int count = rs.getInt("HowMany");
rs.close();

int rebate = 100000 / count;
String cmd = "update STUDENT "
           + "set MealPlanBalance = MealPlanBalance + " + rebate
           + " where GradYear >= extract(year, current_date)";
stmt.executeUpdate(cmd);
conn.commit();
```

Проблема этой транзакции заключается в ошибочном предположении, что число текущих студентов не изменится между вычислением суммы начислений и обновлением записей в таблице STUDENT. Допустим, что между закрытием набора результатов и выполнением оператора `update` в таблицу STUDENT было добавлено несколько новых записей. Эти новые записи окажутся неучтенными при вычислении скидки, и университет потратит больше 100 000 долларов. Такие новые записи называют *фантомными записями* (phantom records), потому что они неожиданно появились после начала транзакции.

Эти примеры иллюстрируют проблемы, которые могут возникнуть при одновременном выполнении двух транзакций. Единственный способ гарантировать отсутствие проблем в произвольной транзакции – выполнить ее в полной изоляции от других транзакций. Эта форма изоляции называется *упорядоченным выполнением* (serializability) и подробно обсуждается в главе 5.

К сожалению, упорядоченные транзакции могут выполняться очень медленно, потому что они требуют, чтобы движок базы данных ограничил параллельное выполнение. Поэтому JDBC определяет четыре уровня изоляции, позволяющих клиентам указывать степень изолированности их транзакций:

- *read uncommitted* (чтение неподтвержденных данных) предполагает полное отсутствие любой изоляции; транзакция с таким уровнем изоляции может страдать от всех трех проблем, описанных в примерах выше;

- *read committed* (чтение только подтвержденных данных) запрещает транзакциям доступ к неподтвержденным (незафиксированным) значениям; на этом уровне изоляции все еще возможны проблемы с неповторяемым чтением и фантомами;
- *repeatable read* (повторяемое чтение) дополняет уровень изоляции read committed, гарантируя повторяемость чтения; на этом уровне возможны лишь проблемы с фантомами;
- *serializable* (упорядоченное выполнение) гарантирует полное отсутствие любых проблем.

Указать желаемый уровень изоляции клиент JDBC сможет в вызове метода `setTransactionIsolation` класса `Connection`. Например, следующий код устанавливает уровень изоляции с упорядоченным выполнением:

```
DataSource ds = ...
Connection conn = ds.getConnection();
conn.setAutoCommit(false);
conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
```

Эти четыре уровня изоляции определяют компромиссы между скоростью выполнения и потенциальными проблемами. То есть увеличение скорости выполнения транзакций увеличивает риск столкнуться с проблемами. Этот риск можно уменьшить путем тщательного анализа клиента.

Например, вы можете решить, что фантомное и неповторяемое чтение не является проблемой. Это вполне оправдано, например, если транзакция только вставляет новые или удаляет определенные существующие записи (как в операторе «delete from STUDENT where SId = 1»). В подобном случае уровня изоляции read committed (чтение только подтвержденных данных) будет вполне достаточно, и его применение не приведет к ненужным потерям в скорости.

Еще пример: вы можете решить, что вам не угрожают никакие потенциальные проблемы. Предположим, что транзакция вычисляет среднюю оценку за каждый год. Вы решаете, что даже если во время выполнения транзакции какие-то оценки изменятся, это вряд ли существенно повлияет на итоговую статистику. В таком случае вполне разумно выбрать уровень изоляции read committed (чтение только подтвержденных данных) или даже read uncommitted (чтение неподтвержденных данных).

По умолчанию во многих серверах баз данных (включая Derby, Oracle и Sybase) выбирается уровень изоляции read committed (чтение только подтвержденных данных). Он прекрасно подходит для простых запросов, которые пишутся неискушенными пользователями и выполняются в режиме автоматической фиксации. Однако для критических задач очень важно со всем тщанием подходить к выбору подходящего уровня изоляции. Программист, отключающий режим автоматической фиксации, должен выбрать правильный уровень изоляции для каждой транзакции.

2.2.4. Параметризованные операторы

Многие клиентские программы JDBC имеют *параметры* в том смысле, что принимают аргументы от пользователя и выполняют операторы SQL, используя эти аргументы. Примером такого клиента может служить демонстрационная программа `FindMajors`, представленная в листинге 2.16.

Листинг 2.16. JDBC-код для клиента FindMajors

```
public class FindMajors {
    public static void main(String[] args) {
        System.out.print("Enter a department name: ");
        Scanner sc = new Scanner(System.in);
        String major = sc.next();
        sc.close();
        String qry = "select sname, gradyear from student, dept "
            + "where did = majorid and dname = '" + major + "'";

        ClientDataSource ds = new ClientDataSource();
        ds.setServerName("localhost");
        ds.setDatabaseName("studentdb");
        try ( Connection conn = ds.getConnection();
              Statement stmt = conn.createStatement();
              ResultSet rs = stmt.executeQuery(qry)) {

            System.out.println("Here are the " + major + " majors");
            System.out.println("Name\tGradYear");
            while (rs.next()) {
                String sname = rs.getString("sname");
                int gradyear = rs.getInt("gradyear");
                System.out.println(sname + "\t" + gradyear);
            }
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

Сначала этот клиент предлагает пользователю ввести название кафедры. Затем включает это название в запрос SQL и выполняет его. Например, допустим, что пользователь ввел значение «math». Тогда сгенерированный SQL-запрос будет выглядеть так:

```
select SName, GradYear from STUDENT, DEPT
where DId = MajorId and DName = 'math'
```

Обратите внимание, что, генерируя запрос, код явно заключает название кафедры в одинарные кавычки. Однако вместо динамического создания оператора SQL, как показано выше, клиент может использовать *параметризованный* оператор SQL. Параметризованный оператор – это оператор SQL, в котором символы «?» обозначают отсутствующие значения параметров. Оператор может иметь несколько параметров, каждый из которых обозначается символом «?». Каждый параметр имеет *индекс*, соответствующий его позиции. Например, следующий параметризованный оператор удалит все записи, соответствующие студентам с указанными годом выпуска и основной специализацией. Значение поля `GradYear` здесь сопоставляется с параметром с индексом 1, а значение поля `MajorId` – с параметром с индексом 2.

```
delete from STUDENT where GradYear = ? and MajorId = ?
```

Параметризованные операторы в JDBC представляет класс PreparedStatement. Подготовка параметризованных операторов выполняется в три этапа:

1) на основе заданного параметризованного оператора SQL создается объект PreparedStatement;
2) ему передаются значения параметров;
3) затем подготовленный оператор выполняется.

Для примера в листинге 2.17 показана исправленная версия клиента FindMajors, использующая параметризованный оператор. Изменения выделены жирным шрифтом. Последние три инструкции в коде, выделенные жирным, соответствуют пунктам в предыдущем списке. Сначала клиент создает объект PreparedStatement, вызывая метод prepareStatement и передавая аргумент с параметризованным оператором SQL. Затем вызывается метод setString, чтобы присвоить значение первому (и единственному) параметру. Наконец, вызывается метод executeQuery, выполняющий оператор.

Листинг 2.17. Измененная версия клиента FindMajors, использующая параметризованный оператор

```
public class PreparedFindMajors {
    public static void main(String[] args) {
        System.out.print("Enter a department name: ");
        Scanner sc = new Scanner(System.in);
        String major = sc.next();
        sc.close();
        String qry = "select sname, gradyear from student, dept "
                   + "where did = majorid and dname = ?";
        ClientDataSource ds = new ClientDataSource();
        ds.setServerName("localhost");
        ds.setDatabaseName("studentdb");
        try ( Connection conn = ds.getConnection();
            PreparedStatement pstmt = conn.prepareStatement(qry)) {;
            pstmt.setString(1, major);
            ResultSet rs = pstmt.executeQuery();
            System.out.println("Here are the " + major + " majors");
            System.out.println("Name\tGradYear");
            while (rs.next()) {
                String sname = rs.getString("sname");
                int gradyear = rs.getInt("gradyear");
                System.out.println(sname + "\t" + gradyear);
            }
            rs.close();
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

В листинге 2.18 перечислены наиболее часто используемые методы класса PreparedStatement. Методы executeQuery и executeUpdate действуют аналогично соответствующим методам класса Statement; разница лишь в том, что они не принимают никаких аргументов. Методы setInt и setString присваивают значения параметрам. Вызов setString в листинге 2.17 присваивает назва-

ние отдела параметру с индексом 1. Обратите внимание, что метод `setString` автоматически заключает значение в одинарные кавычки, поэтому клиенту делать это не нужно.

Листинг 2.18. Часть API класса PreparedStatement

```
public ResultSet executeQuery()              throws SQLException;
public int executeUpdate()                   throws SQLException;
public void setInt(int index, int val)       throws SQLException;
public void setString(int index, String val) throws SQLException;
```

Большинству удобнее использовать параметризованные операторы, нежели явно создавать их. Параметризованные операторы также более эффективны при использовании в цикле, как показано в листинге 2.19. Причина в том, что движок базы данных может скомпилировать параметризованный оператор, не зная значений параметров. Он компилирует оператор один раз, а затем выполняет его внутри цикла без повторной компиляции.

Листинг 2.19. Использование параметризованного оператора в цикле

```
// Подготовка запроса
String qry = "select SName, GradYear from STUDENT, DEPT "
           + "where DId = MajorId and DName = ?";
PreparedStatement pstmt = conn.prepareStatement(qry);

// Многократная подстановка параметров и выполнение запроса
String major = getUserInput();
while (major != null) {
    pstmt.setString(1, major);
    ResultSet rs = pstmt.executeQuery();
    displayResultSet(rs);
    major = getUserInput();
}
```

2.2.5. Прокручиваемые и обновляемые наборы результатов

Наборы результатов, поддерживаемые ядром JDBC, допускают перемещение от записи к записи только в прямом направлении и не могут обновляться. Однако в JDBC имеются дополнительные инструменты, позволяющие получать *прокручиваемые* и *обновляемые* наборы результатов. Клиенты могут читать записи из таких наборов, выбирая произвольную позицию, обновлять текущую запись и вставлять новые записи. Эти дополнительные методы перечислены в листинге 2.20.

Листинг 2.20. Часть API класса ResultSet

Методы для работы с прокручиваемыми наборами результатов

```
public void    beforeFirst()        throws SQLException;
public void    afterLast()          throws SQLException;
public boolean previous()           throws SQLException;
public boolean next()               throws SQLException;
public boolean absolute(int pos)    throws SQLException;
public boolean relative(int offset) throws SQLException;
```

Методы для работы с обновляемыми наборами результатов

```
public void updateInt(String fldname, int val) throws SQLException;
public void updateString(String fldname, String val)
                                                  throws SQLException;
public void updateRow()        throws SQLException;
public void deleteRow()        throws SQLException;
public void moveToInsertRow()  throws SQLException;
public void moveToCurrentRow() throws SQLException;
```

Метод `beforeFirst` устанавливает позицию чтения перед первой записью в наборе результатов, а метод `afterLast` – после последней записи. Метод `absolute` устанавливает позицию чтения на указанную запись и возвращает `false`, если такой записи нет. Метод `relative` устанавливает позицию чтения через указанное число записей относительно текущей. В частности, вызов `relative(1)` идентичен вызову `next()`, а вызов `relative(-1)` – вызову `previous()`.

Методы `updateInt` и `updateString` изменяют указанное поле в текущей записи на стороне клиента. Но изменения не отправляются в базу данных, пока не будет вызван метод `updateRow`. Необходимость вызова `updateRow` доставляет некоторые неудобства, но это позволяет JDBC обновить несколько полей в записи одним обращением к движку.

При добавлении новых записей используется идея *записи для вставки*. Добавляемая запись не существует в таблице с набором результатов (вы не сможете перейти к ней). Цель записи для вставки – служить площадкой для новых записей. Клиент вызывает `moveToInsertRow`, чтобы перейти в позицию записи для вставки, затем использует методы `updateXXX`, чтобы присвоить полям необходимые значения, потом вызывает `updateRow`, чтобы вставить запись в базу данных, и, наконец, `moveToCurrentRow`, чтобы вернуться к записи в наборе результатов, которая была текущей до вставки.

По умолчанию наборы результатов допускают перемещение от записи к записи только в прямом направлении и не могут обновляться. Если клиенту нужен более мощный набор записей, он должен сообщить об этом в вызове метода `createStatement` класса `Connection`. В дополнение к методу `createStatement` без аргументов в ядре JDBC существует также метод с двумя аргументами, с помощью которых клиент может определить необходимость поддержки прокрутки и обновления. Например, взгляните на следующий оператор:

```
Statement stmt =
conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                     ResultSet.CONCUR_UPDATABLE);
```

Все наборы результатов, возвращаемые этим оператором, будут поддерживать прокрутку и обновление. Константа `TYPE_FORWARD_ONLY` сообщает, что набор результатов не должен поддерживать прокрутку, а `CONCUR_READ_ONLY` – что набор результатов не должен поддерживать возможность обновления. Эти константы можно комбинировать для получения набора результатов с желаемыми характеристиками прокручиваемости и обновляемости.

Для примера вернемся к коду в листинге 2.10, который позволял пользователю выполнить обход записей в таблице COURSE и удалить ненужные. В листинге 2.21 показана измененная версия этого кода, использующая прокручиваемый и обновляемый набор результатов. Обратите внимание, что удаленная строка остается текущей до вызова `next()`.

Листинг 2.21. Измененная версия кода из листинга 2.10

```
DataSource ds = ...
Connection conn = ds.getConnection();
conn.setAutocommit(false);

Statement stmt = conn.createStatement(ResultSet.TYPE_FORWARD_ONLY,
                                      ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("select * from COURSE");
while (rs.next()) {
    String title = rs.getString("Title");
    boolean goodCourse = getUserDecision(title);
    if (!goodCourse)
        rs.deleteRow();
}
rs.close();
stmt.close();
conn.commit();
```

Прокручиваемый набор результатов имеет ограниченный круг применения, потому что в большинстве случаев клиент точно знает, что делать с полученными записями, и нет необходимости проверять их дважды. Прокручиваемый набор результатов обычно требуется, только если пользователям необходимо взаимодействовать с результатами запроса. Например, представьте клиента, который отображает результаты запроса в виде Swing-объекта `JTable`. Объект `JTable` будет отображать полосу прокрутки, если все строки не уместятся по высоте экрана, и с ее помощью пользователь сможет перемещаться по записям взад и вперед. В этой ситуации необходимо, чтобы клиент получил прокручиваемый набор результатов для объекта `JTable`, позволяющий извлекать предыдущие записи при прокрутке назад.

2.2.6. Дополнительные типы данных

Кроме целочисленных и строковых значений, JDBC поддерживает также множество других типов данных. Например, рассмотрим интерфейс `ResultSet`. Помимо методов `getInt` и `getString` он имеет также методы `getFloat`, `getDouble`, `getShort`, `getTime`, `getDate` и ряд других. Каждый из этих методов будет извлекать значение из указанного поля текущей записи и преобразовывать его (если возможно) в указанный тип. Разумеется, для извлечения числовых значений предпочтительнее использовать числовые методы JDBC (такие как `getInt`, `getFloat` и другие). Но, как бы то ни было, JDBC будет пытаться преобразовать любое значение SQL в тип, определяемый методом. В частности, любое значение SQL всегда можно преобразовать в строку.

2.3. Вычисления в Java и SQL

Всякий раз, работая над созданием клиента JDBC, программист должен принять важное решение: какую часть вычислений будет выполнять движок базы данных, а какую – код на Java. В этом разделе мы займемся исследованием данного вопроса.

Вернемся к демонстрационному клиенту `StudentMajor` в листинге 2.5. В этой программе все вычисления, необходимые для соединения таблиц STUDENT и DEPT, выполняет движок базы данных, а сам клиент отвечает только за получение результатов и их вывод на экран.

Однако клиента можно реализовать так, что все вычисления будут выполняться в коде на Java, как показано в листинге 2.22. В этом примере движок отвечает только за извлечение данных из таблиц STUDENT и DEPT; всю остальную работу выполняет клиент, вычисляя соединение и отображая результаты.

Какая из этих версий лучше? Очевидно, что первая выглядит более элегантной. Она не только короче, но еще и проще читается. А если взглянуть с точки зрения эффективности? Как правило, чем меньше обязанностей возлагается на клиента, тем он эффективнее. Тому есть две основные причины:

- при таком подходе часто приходится передавать меньше данных между движком и клиентом, что особенно важно, если они действуют на разных машинах;
- движок имеет полную информацию о том, как реализована каждая таблица, и о возможных способах вычисления сложных запросов (например, соединений). Маловероятно, что клиент сможет вычислить запрос так же эффективно, как движок.

Например, код в листинге 2.22 вычисляет соединение с использованием двух вложенных циклов. Внешний цикл перебирает записи с учащимися из таблицы STUDENT, а внутренний ищет запись в таблице DEPT, соответствующую основной специальности для каждого учащегося. Это простой и понятный алгоритм соединения таблиц, но не особенно эффективный. В главах 13 и 14 мы обсудим несколько более эффективных решений.

Листинги 2.5 и 2.22 иллюстрируют две крайности – действительно хороший и действительно плохой код JDBC, поэтому выбор между ними очевиден. Но иногда сделать правильный выбор намного сложнее. Например, обратимся к демонстрационному клиенту `PreparedFindMajors` в листинге 2.17, который возвращает список студентов, выбравших заданную основную специализацию. Этот код посылает движку SQL-запрос, соединяющий таблицы STUDENT и DEPT. Операция соединения таблиц, как известно, может занять много времени. После некоторых серьезных размышлений мы выясняем, что можем получить нужные данные без использования соединения. Идея заключается в использовании двух запросов, каждый из которых просматривает только одну таблицу. Первый просматривает таблицу DEPT, ищет запись с заданным названием кафедры и возвращает ее значение `DId`. Затем второй запрос отыскивает записи в таблице STUDENT, содержащие это значение в поле `MajorID`. Код с реализацией данной идеи показан в листинге 2.23.

Этот алгоритм прост, элегантен и эффективен. Он требует от движка выполнить лишь последовательное сканирование двух таблиц, что намного быстрее, чем соединение. Мы можем гордиться таким решением, правда?

Листинг 2.22. Альтернативная (и худшая) реализация клиента StudentMajor

```
public class BadStudentMajor {
    public static void main(String[] args) {
        ClientDataSource ds = new ClientDataSource();
        ds.setServerName("localhost");
        ds.setDatabaseName("studentdb");
        Connection conn = null;
        try {
            conn = ds.getConnection();
            conn.setAutoCommit(false);
            try (Statement stmt1 = conn.createStatement();
                 Statement stmt2 = conn.createStatement(
                     ResultSet.TYPE_SCROLL_INSENSITIVE,
                     ResultSet.CONCUR_READ_ONLY);
                 ResultSet rs1 = stmt1.executeQuery(
                                      "select * from STUDENT");
                 ResultSet rs2 = stmt2.executeQuery(
                                      "select * from DEPT") ) {

                System.out.println("Name\tMajor");
                while (rs1.next()) {
                    // получить следующего студента
                    String sname = rs1.getString("SName");
                    String dname = null;
                    rs2.beforeFirst();
                    while (rs2.next())
                        // найти название основной специализации этого студента
                        if (rs2.getInt("DId") == rs1.getInt("MajorId")) {
                            dname = rs2.getString("DName");
                            break;
                        }
                    System.out.println(sname + "\t" + dname);
                }
            }
            conn.commit();
            conn.close();
        }
        catch(SQLException e) {
            e.printStackTrace();
            try {
                if (conn != null) {
                    conn.rollback();
                    conn.close();
                }
            }
            catch (SQLException e2) {}
        }
    }
}
```

Листинг 2.23. Более эффективная реализация клиента FindMajors

```
public class CleverFindMajors {
    public static void main(String[] args) {
        String major = args[0];
        String qry1 = "select DId from DEPT where DName = ?";
        String qry2 = "select * from STUDENT where MajorId = ?";

        ClientDataSource ds = new ClientDataSource();
        ds.setServerName("localhost");
        ds.setDatabaseName("studentdb");
        try (Connection conn = ds.getConnection()) {
            PreparedStatement stmt1 = conn.prepareStatement(qry1);
            stmt1.setString(1, major);
            ResultSet rs1 = stmt1.executeQuery();
            rs1.next();
            int deptid = rs.getInt("Did"); // получить код кафедры
            rs1.close();
            stmt1.close();

            PreparedStatement stmt2 = conn.prepareStatement(qry2);
            stmt2.setInt(1, deptid);
            ResultSet rs2 = stmt2.executeQuery();
            System.out.println("Here are the " + major + " majors");
            System.out.println("Name\tGradYear");
            while (rs2.next()) {
                String sname = rs2.getString("sname");
                int gradyear = rs2.getInt("gradyear");
                System.out.println(sname + "\t" + gradyear);
            }
            rs2.close();
            stmt2.close();
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

К сожалению, наши усилия были напрасны. Новый алгоритм на самом деле не новый. Это просто хорошо продуманная реализация соединения, в частности это *многобуферное произведение* (multibuffer product), о котором рассказывается в главе 14, с материализованной внутренней таблицей. В хорошем движке базы данных обязательно будет реализован этот алгоритм (вместе с другими), и он будет использоваться, если окажется наиболее эффективным. Таким образом, движок базы данных свел на нет все наши интеллектуальные усилия. Мораль та же, что и в случае с клиентом `StudentMajor`: доверить движку выполнить всю работу – это, как правило, самая эффективная стратегия (а также самая простая в реализации).

Одна из ошибок, которую делают начинающие программисты JDBC, состоит в том, что они пытаются сделать слишком много в клиенте. Программист может думать, что реализация запроса на Java действует более эффективно. Или он может не знать, как выразить запрос на SQL, и ему удобнее написать код на Java. В каждом из этих случаев решение реализовать запрос на Java почти

всегда ошибочно. Программист должен верить, что движок базы данных лучше справится со своей работой[1].

2.4. Итоги

- Методы JDBC управляют передачей данных между клиентом на Java и движком базы данных.
- Ядро JDBC включает пять интерфейсов: `Driver`, `Connection`, `Statement`, `ResultSet` и `ResultSetMetaData`.
- Объект `Driver` скрывает за своей оболочкой низкоуровневые детали соединения с движком. Чтобы подключиться к движку, клиент должен получить экземпляр соответствующего класса драйвера. Класс драйвера и его строка подключения – это единственное, что связывает код JDBC-программы с производителем базы данных. Все остальное реализуют независимые интерфейсы JDBC.
- Наборы результатов и соединения занимают ресурсы, которые могут понадобиться другим клиентам. Клиент JDBC всегда должен закрывать их как можно скорее.
- Каждый метод JDBC может сгенерировать исключение `SQLException`. Клиент обязан проверять эти исключения.
- Методы класса `ResultSetMetaData` позволяют получить информацию о схеме набора результатов, то есть именах, типах и размерах содержащихся в нем полей. Эта информация может пригодиться в клиенте, принимающем запросы непосредственно от пользователя, как в интерпретаторе SQL.
- Клиент, использующий только ядро JDBC, напрямую обращается к классу драйвера. Расширенный JDBC, предоставляющий класс `DriverManager` и интерфейс `DataSource`, позволяет упростить процесс подключения и сделать клиента более независимым от производителей баз данных.
- Класс `DriverManager` хранит коллекцию драйверов. Клиент регистрирует в этом классе необходимые ему драйверы явно или (что предпочтительнее) через файл системных свойств. Когда клиенту требуется подключиться к базе данных, он передает строку подключения диспетчеру драйверов, а тот устанавливает соединение от имени клиента.
- Объект `DataSource` обеспечивает еще большую независимость от производителя, инкапсулируя не только драйвер, но и строку подключения. Благодаря ему клиент может подключиться к движку базы данных, не зная ни одной детали подключения. Администратор базы данных может создавать различные объекты `DataSource` и размещать их на сервере для использования клиентами.
- Клиент, использующий ядро JDBC, не управляет транзакциями. Все запросы клиента движок базы данных выполняет в режиме *автоматиче-*

[1] По крайней мере, вы должны начать с веры, что движок будет работать эффективно. Если обнаружится, что ваше приложение работает медленно, потому что движок выполняет соединение таблиц неэффективно, можете попробовать реорганизовать программу, как показано в листинге 2.23. Но в любом случае старайтесь не спешить с далеко идущими выводами.

ской фиксации, то есть каждый оператор SQL выполняется в отдельной транзакции.

- Все операции с базой данных, выполняемые в транзакции, рассматриваются как единое целое. В случае успешного завершения текущей единицы работы транзакция *фиксируется*. Если транзакция не может быть зафиксирована, она *откатывается*. Выполняя откат, движок базы данных отменяет все изменения, сделанные транзакцией.
- Автоматическая фиксация – удобный режим по умолчанию для простых клиентов JDBC. Если клиент выполняет критические задачи, программист должен тщательно проанализировать необходимость использования транзакций. Отключение режима автоматической фиксации производится вызовом `setAutoCommit(false)`. Этот вызов заставляет движок запустить новую транзакцию. Затем, чтобы завершить текущую транзакцию и начать новую, клиент должен вызвать `commit()` или `rollback()`. Отключая автоматическую фиксацию, клиент должен предусматривать обработку ошибок, возникающих при выполнении операторов SQL, и откатывать соответствующие транзакции.
- Клиент также может использовать метод `setTransactionIsolation`, чтобы установить определенный уровень изоляции. В JDBC поддерживается четыре уровня изоляции:
  - *read uncommitted* (чтение неподтвержденных данных) предполагает полное отсутствие любой изоляции; транзакция с таким уровнем изоляции может страдать от проблем, связанных с чтением неподтвержденных данных, неповторямым чтением и чтением фантомных записей;
  - *read committed* (чтение только подтвержденных данных) запрещает транзакциям доступ к неподтвержденным (незафиксированным) значениям; на этом уровне изоляции все еще возможны проблемы с неповторяемым чтением и фантомами;
  - *repeatable read* (повторяемое чтение) дополняет уровень изоляции read committed, гарантируя повторяемость чтения; на этом уровне возможны только проблемы с фантомами;
  - *serializable* (упорядоченное выполнение) гарантирует полное отсутствие любых проблем.
- Уровень изоляции serializable (упорядоченное выполнение) выглядит особенно заманчивым, но при его использовании транзакции выполняются медленно. Программист должен проанализировать риск ошибок, обусловленных параллельным обслуживанием нескольких клиентов, и выбрать наименее ограничивающий уровень изоляции, обеспечивающий только допустимые риски.
- *Параметризованные операторы* определяются как обычные операторы SQL, но включают заполнители для параметров. Используя такие операторы, клиент может позднее задавать значения параметров, а затем выполнять их. Параметризованные операторы являются удобной заменой динамически генерируемых операторов SQL. Более того, параметризованный оператор можно скомпилировать до назначения параметров, благодаря чему такой оператор будет выполняться в несколько раз (например, в цикле) эффективнее.

- Дополнительные инструменты в JDBC позволяют получать *прокручиваемые* и *обновляемые* наборы результатов. По умолчанию набор результатов поддерживает обход только в прямом направлении и не может обновляться. Если клиенту нужен более мощный набор результатов, он должен указать это в вызове метода `createStatement` класса `Connection`.
- Правило, которому желательно следовать при разработке клиента JDBC, – позволить движку выполнить как можно больше работы. Движки баз данных чрезвычайно сложны и, как правило, способны выбрать наиболее эффективный способ получения нужных данных. Для клиента почти всегда предпочтительнее определить оператор SQL, извлекающий только нужные данные, и передать его движку. Проще говоря, программист должен доверить движку выполнить свою работу.

2.5. Для дополнительного чтения

Хорошее подробное описание JDBC можно найти в книге Fisher et al. (2003). Часть этой книги доступна в виде онлайн-учебника по адресу: docs.oracle.com/javase/tutorial/jdbc. Кроме того, все производители баз данных предоставляют документацию, описывающую порядок использования драйверов, а также другие специфические особенности. Если вы намереваетесь написать клиента для конкретного движка, обязательно ознакомьтесь с документацией.

Fisher, M., Ellis, J., & Bruce, J. (2003). «JDBC API tutorial and reference (3rd ed.)». Addison Wesley.

2.6. Упражнения

Теория

2.1. В документации Derby рекомендуется отключать автоматическую фиксацию при выполнении подряд нескольких операторов вставки записей. Объясните, почему была дана такая рекомендация.

Практика

2.2. Напишите несколько SQL-запросов для университетской базы данных. Для каждого запроса напишите программу, использующую Derby, которая выполняет этот запрос и выводит полученные результаты.

2.3. Программа `SimpleIJ` требует, чтобы каждый оператор SQL вводился в одной строке. Измените ее, добавив возможность ввода многострочных операторов, заканчивающихся точкой с запятой, как это сделано в программе `ij` в Derby.

2.4. Напишите класс `NetworkDataSource` для SimpleDB, действующий подобно классу `ClientDataSource` в Derby. Добавьте этот класс в пакет `simpledb.jdbc.network`. От вас не требуется реализовать все методы интерфейса `javax.sql.DataSource` (и его суперклассов); достаточно реализовать единственный метод `getConnection()` без аргументов. Какие методы, зависящие от производителя баз данных, должен иметь `NetworkDataSource`?

2.5. Часто полезно иметь возможность создавать текстовый файл, содержащий операторы SQL. Эти операторы затем можно выполнять в пакетном режиме с помощью программы JDBC. Напишите такую программу, которая читает операторы из заданного текстового файла и выполняет их. Считайте, что каждая строка файла является отдельной командой.

2.6. Изучите, как можно использовать набор результатов для заполнения Java-объекта `JTable`. (Подсказка: для этого нужно расширить класс `AbstractTableModel`.) Затем, взяв за основу демонстрационного клиента `FindMajors`, напишите программу с графическим интерфейсом, которая отображает результаты в `JTable`.

2.7. Напишите JDBC-код, выполняющий следующие действия:

a) импортирует данные из текстового файла в существующую таблицу. Каждая запись в текстовом файле должна находиться в отдельной строке, а поля должны отделяться друг от друга символами табуляции. В первой строке файла должны перечисляться имена полей. Клиент должен принимать имя файла и имя таблицы на входе и вставлять записи в таблицу;

b) экспортирует данные из таблицы в текстовый файл. Клиент должен принимать имя файла и имя таблицы на входе и записывать содержимое каждой записи в файл. В первой строке файла должны перечисляться имена полей.

2.8. Эта глава не рассматривала возможность появления неопределенных (NULL) значений в наборе результатов. Для проверки неопределенных значений используется метод `wasNull` класса `ResultSet`. Представьте, что вы используете `getInt` или `getString` для получения значения поля. Если вызвать `wasNull` сразу после извлечения значения, он вернет `true`, если полученное значение было неопределенным. Например, следующий цикл выводит годы окончания обучения, предполагая, что некоторые из них могут быть неопределенными:

```
while(rs.next()) {
  int gradyr = rs.getInt("gradyear");
  if (rs.wasNull())
    System.out.println("null");
  else
    System.out.println(gradyr);
}
```

a) Перепишите клиента `StudentMajor`, предположив, что имена студентов могут быть неопределенными.

b) Измените клиента `SimpleIJ`, чтобы он подключался к Derby (вместо SimpleDB). Затем перепишите его, предположив, что любые поля могут быть неопределенными.

Глава 3

Управление дисками и файлами

Движки баз данных сохраняют свои данные на устройствах долговременного хранения, таких как диски и твердотельные накопители. В этой главе рассматриваются свойства этих устройств и методы (такие как объединение в RAID) увеличения их надежности и скорости доступа к ним. Здесь также рассматриваются два интерфейса, которые операционная система предлагает для взаимодействий с такими устройствами, – интерфейс на уровне блоков и интерфейс на уровне файлов – и описывается комбинированный подход к использованию этих интерфейсов, наиболее подходящий для систем управления базами данных. Наконец, здесь подробно рассматриваются диспетчер файлов в SimpleDB, его API и реализация.

3.1. Долговременное хранилище данных

Содержимое базы данных должно *сохраняться в долговременном хранилище*, чтобы данные не были потеряны, если система баз данных или компьютер выйдет из строя. В этом разделе рассматриваются две особенно полезные аппаратные технологии: *дисковые* и *твердотельные накопители*. Твердотельные накопители еще не так широко распространены, как дисковые, однако их важность будет возрастать по мере развития технологий. Начнем с дисковых накопителей.

3.1.1. Дисковые накопители

Дисковый накопитель содержит одну или несколько вращающихся *пластин*. На пластине имеются концентрические дорожки, и каждая дорожка состоит из последовательности байтов. Чтение (и запись) байтов с пластины производятся с помощью подвижного рычага, на конце которого закреплена головка чтения/записи. Рычаг поворачивается так, чтобы головка оказалась над желаемой дорожкой, после чего головка считывает (или записывает) байты с дорожки на пластине, вращающейся под ней. На рис. 3.1 показан вид сверху дискового накопителя с одной пластиной. Конечно, на этом рисунке не соблюден масштаб, потому что на типичной пластине находится много тысяч дорожек.

Современные дисковые накопители обычно имеют несколько пластин. Для экономии пространства пары пластин чаще всего соединяются друг с другом и образуют нечто вроде двухсторонней пластины; но концептуально каждая сторона считается отдельной пластиной. Для каждой пластины имеется своя отдельная головка чтения/записи. Головки не могут двигаться независимо друг от друга, потому что все они закреплены на общем *механизме позиционирования* (актуаторе), который перемещает их все одновременно. Кроме того, в каждый момент активной может быть только одна головка чтения/записи, потому что имеется только одна шина передачи данных в компьютер. На рис. 3.2 показан вид сбоку дискового накопителя с несколькими пластинами.

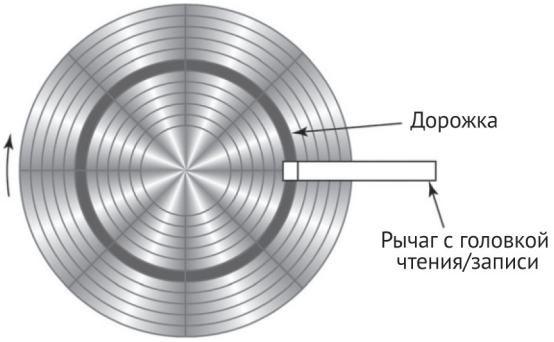


Рис. 3.1. Вид сверху дискового накопителя с одной пластиной

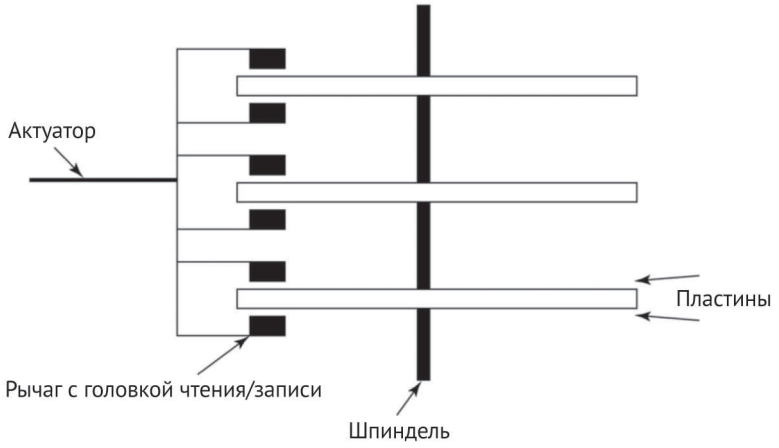


Рис. 3.2. Вид сбоку дискового накопителя с несколькими пластинами

В общем случае дисковый накопитель можно охарактеризовать четырьмя величинами: емкостью, скоростью вращения, скоростью передачи данных и временем поиска.

Емкость накопителя – это количество байтов, которое можно в нем сохранить. Емкость зависит от количества пластин, количества дорожек на пластине и количества байтов на дорожке. Учитывая, что обычно пластины имеют более

или менее стандартные размеры, производители увеличивают емкость главным образом за счет повышения плотности, то есть увеличения количества дорожек на пластине и байтов на дорожке. В настоящее время уже не редкость пластины, вмещающие более 40 Гбайт данных.

Скорость вращения пластин обычно измеряется в оборотах в минуту. Типичные современные накопители имеют скорость вращения от 5400 до 15 000 об/мин.

Скорость передачи данных – это скорость, с какой байты проходят через головку диска в/из памяти. Например, за время полного оборота пластины можно передать все байты на дорожке. То есть скорость передачи определяется скоростью вращения и количеством байтов на дорожке. Типичной можно назвать скорость 100 МБайт/с.

Время поиска дорожки (позиционирования) – это время, необходимое приводу для перемещения головок из текущего местоположения к требуемой дорожке. Это время зависит от того, сколько дорожек нужно пересечь, и может колебаться от 0 мс (если дорожка назначения совпадает с начальной дорожкой) до 15–20 мс (если дорожка назначения и начальная дорожка находятся на разных концах диска). Среднее время позиционирования обычно дает разумную оценку скорости актуатора. Среднее время позиционирования на современных дисках составляет около 5 мс.

Рассмотрим пример. Допустим, что у нас есть привод со средним временем позиционирования 5 мс и с четырьмя пластинами, вращающимися со скоростью 10 000 об/мин. Каждая пластина содержит 10 000 дорожек, каждая дорожка содержит 500 000 байт. Вот некоторые расчетные значения[1]:

Емкость привода:

```
500 000 байт на дорожке × 10 000 дорожек на пластине × 4 пластины в накопителе
= 20 000 000 000 байт, или примерно 20 Гбайт
```

Скорость передачи данных:

```
500 000 байт на оборот × 10 000 оборотов в 60 секунд
= 83 333 333 байт/с, или примерно 83 Мбайт/с
```

3.1.2. Доступ к дисковому накопителю

Под доступом к диску понимается выполнение запроса на чтение нескольких байтов из привода в память или на запись нескольких байтов из памяти на диск. Байты должны занимать непрерывный участок дорожки на некоторой пластине. Накопитель осуществляет доступ к диску в три этапа:

- перемещает головку на заданную дорожку за время, которое называется *временем поиска (позиционирования)*;
- ждет, пока пластина повернется так, что первый желаемый байт окажется под головкой; это называется *задержкой вращения*;
- пока пластина продолжает вращаться, головка читает (или записывает) байты, проносящиеся под ней, пока не будет прочитан последний желаемый байт; это время называется *временем передачи*.

---

[1] Технически 1 Кбайт = 1024 байт, 1 МБайт = 1 048 576 байт и 1 Гбайт = 1 073 741 824 байт. Для удобства я округлю их до 1000, 1 000 000 и 1 000 000 000 байт соответственно.

Время, необходимое для выполнения полного цикла доступа к приводу, определяется как сумма времени позиционирования, задержки вращения и времени передачи. Каждое из этих значений времени определяется скоростью работы механической части накопителя. Механические части двигаются значительно медленнее, чем электрические импульсы, поэтому накопители намного медленнее, чем ОЗУ. Время позиционирования и задержка вращения вносят наибольший вклад в общее время доступа. Это – накладные расходы, сопутствующие каждой операции с накопителем.

Вычислять точное время позиционирования и задержки вращения не имеет смысла, потому что для этого необходимо знать предшествующее состояние накопителя. Поэтому на практике используются усредненные значения. Вы уже познакомились со средним временем позиционирования. Среднюю задержку вращения легко можно рассчитать. Задержка вращения может изменяться в диапазоне от 0 (если первый байт оказался под головкой) до времени, необходимого, чтобы выполнить полный оборот пластины (если первый байт только что миновал головку). Средняя задержка равна половине времени полного оборота пластины, то есть зависит от скорости вращения.

Общее время передачи тоже легко рассчитать, исходя из скорости передачи. В частности, если скорость передачи составляет `r` байт/с и требуется передать `b` байт, тогда общее время переноса составит `b/r` секунд.

Например, рассмотрим накопитель со скоростью вращения 10 000 об/мин, средним временем позиционирования 5 мс и скоростью передачи 83 МБайт/с. Вот результаты вычислений параметров накопителя:

Средняя задержка вращения:

```
60 секунд в минуте × 1 минута на 10 000 оборотов × 1/2 оборота
= 0.003 секунды, или 3 мс
```

Время передачи одного байта:

```
1 байт × 1 секунду / 83 000 000 байт
= 0.000000012 секунды, или 0.000012 мс
```

Время передачи 1000 байт:

```
1 000 байт × 1 секунду / 83 000 000 байт
= 0.000012 секунды, или 0.012 мс
```

Расчетное время доступа к одному байту:

```
5 мс (позиционирование) + 3 мс (задержка вращения) + 0.000012 мс (передача)
= 8.000012 мс
```

Расчетное время доступа к 1000 байт:

```
5 мс (позиционирование) + 3 мс (задержка вращения) + 0.012 мс (передача)
= 8.012 мс
```

Обратите внимание, что расчетное время доступа к 1000 байт и к 1 байту почти одинаковое. Иначе говоря, нет смысла читать или записывать на диск лишь несколько байтов. На самом деле вы не сможете сделать этого, даже если очень захотите. Современные дисковые накопители сконструированы так, что каждая дорожка делится на *секторы* фиксированной длины; операция чтения (или записи) выполняется над всем сектором сразу. Размер сектора может быть

определен изготовителем привода или выбран при форматировании. Типичный размер сектора составляет 512 байт.

3.1.3. Уменьшение времени доступа к диску

Поскольку дисковые накопители работают очень медленно, было разработано несколько методов, помогающих сократить время доступа. В этом разделе рассматриваются три таких метода: кеширование, цилиндры и чередование дисков.

Кеширование

Дисковый кеш – это память, встроенная в дисковый накопитель, объем которой обычно достаточно велик для хранения содержимого нескольких тысяч секторов. Всякий раз, когда привод читает сектор с диска, он сохраняет его содержимое в своем кеше; если кеш заполнен, новое содержимое затирает старое. Когда компьютер запрашивает чтение сектора, привод проверяет свой кеш. Если сектор находится в кеше, он немедленно возвращается компьютеру без фактического доступа к диску.

Представим, что в течение короткого интервала времени приложение запросило один и тот же сектор несколько раз. Первый запрос перенесет сектор в кеш, а для удовлетворения последующих запросов данные будут извлекаться из кеша, что даст экономию на доступе к диску. Однако эта особенность накопителей не особенно полезна для движка базы данных, потому что он использует свои кеши (как будет показано в главе 4). Если сектор запрашивается несколько раз, движок найдет соответствующие данные в своем кеше и даже не потрудится обратиться к диску.

Истинная ценность дискового кеша заключена в его способности производить *предварительную выборку* секторов. Привод может прочитать не только запрошенный сектор, но всю дорожку, и поместить ее содержимое в кеш, в надежде, что чуть позже компьютер потребует прочитать другие секторы этой же дорожки. Дело в том, что чтение всей дорожки занимает лишь чуть больше времени, чем чтение одного сектора. В частности, при таком подходе отсутствует задержка вращения, потому что привод может начать читать дорожку с любого сектора, находящегося под головкой, и продолжать чтение, пока не будет сделан полный оборот. Давайте сравним время доступа:

```
Время чтения сектора = время позиционирования + 1/2 времени одного оборота
                       + время прохождения сектора

Время чтения дорожки = время позиционирования + время одного оборота
```

То есть разница между временем чтения одного сектора и временем чтения дорожки целиком составляет менее половины времени одного оборота диска. Если чуть позже движок базы данных запросит другой сектор с той же дорожки, прием чтения всей дорожки в кеш сэкономит время.

Цилиндры

Система базы данных может уменьшить время, расходуемое на доступ к диску, сохраняя связанные между собой данные в соседних секторах. Например, иде-

альный способ хранения файла – записать его содержимое на одну дорожку. Очевидно, что эта стратегия лучше всего подходит, если накопитель кеширует дорожки целиком, потому что потом весь файл можно будет прочитать за одно обращение к диску. Но данная стратегия хороша даже без кеширования, потому что удаляет из формулы время позиционирования – когда потребуется прочитать другой сектор, головка уже будет находиться над требуемой дорожкой[1].

Предположим, что файл занимает несколько дорожек. В таком случае хорошо было бы хранить его содержимое на соседних дорожках, чтобы время позиционирования головки было минимальным. Однако еще лучше хранить его содержимое на одной и той же дорожке на нескольких пластинах. Поскольку головки чтения/записи для всех дисков перемещаются вместе, все дорожки с одинаковым порядковым номером будут доступны без дополнительных затрат времени на позиционирование.

Набор дорожек с одинаковым порядковым номером называется *цилиндром*, потому что если посмотреть на эти дорожки сверху, они будут описывать внешнюю поверхность цилиндра. С практической точки зрения цилиндр можно рассматривать как одну очень большую дорожку, потому что для доступа ко всем его секторам не требуется тратить время на позиционирование головки.

Чередование дисков

Еще один способ уменьшить время доступа – использовать несколько накопителей. Два небольших накопителя работают быстрее, чем один большой, потому что содержат два независимых актуатора и могут одновременно отвечать на два запроса к разным секторам. Например, два диска объемом 20 Гбайт, работающих непрерывно, будут действовать примерно в два раза быстрее, чем один диск объемом 40 Гбайт. Это ускорение хорошо масштабируется: в общем случае N дисков будут работать примерно в N раз быстрее, чем один накопитель. (Конечно, несколько дисков небольшого объема обойдутся дороже, чем один диск большой емкости, поэтому прирост эффективности имеет свою цену.)

Однако эффективность нескольких дисков меньшего объема будет потеряна, если использовать их недостаточно интенсивно. Предположим, например, что один диск содержит часто используемые файлы, а другие – редко используемые архивные файлы. В этом случае всю работу будет выполнять первый диск, а остальные – простаивать большую часть времени. Подобная конфигурация будет иметь примерно такую же эффективность, как конфигурация с одним диском.

Таким образом, проблема сводится к балансировке рабочей нагрузки между несколькими дисками. Администратор базы данных может попытаться проанализировать интенсивность использования файлов и наиболее оптимально распределить их между дисками, но это очень непрактично, потому что трудно сделать, и с течением времени придется постоянно переоценивать и перераспределять файлы. К счастью, есть гораздо лучший подход, известный как *чередование дисков*.

---

[1] Файл, содержимое которого разбросано по разным дорожкам диска, называется *фрагментированным*. Многие операционные системы предлагают утилиту дефрагментации, которая сокращает время доступа, перемещая файлы так, чтобы их сектора размещались на диске как можно ближе друг к другу.

Стратегия чередования дисков основана на использовании контроллера, скрывающего маленькие диски от операционной системы и создающего иллюзию одного большого диска. Этот контроллер отображает обращения к секторам виртуального диска в обращения к секторам реальных дисков, действуя следующим образом. Предположим, есть N маленьких дисков, каждый из которых имеет k секторов. Соответственно, виртуальный диск будет иметь $N \times k$ секторов; эти сектора поочередно связываются с секторами реальных дисков. Диск 0 будет содержать виртуальные сектора с номерами 0, N, $2N$, ...; диск 1 будет содержать виртуальные сектора 1, N+1, $2N$+1, ..., и т. д. Термин *чередование дисков* можно объяснить так: если представить, что каждый маленький диск окрашен в свой цвет, тогда дорожки виртуального диска будут выглядеть как цепочки секторов, окрашенных в чередующиеся цвета[1] (см. рис. 3.3).

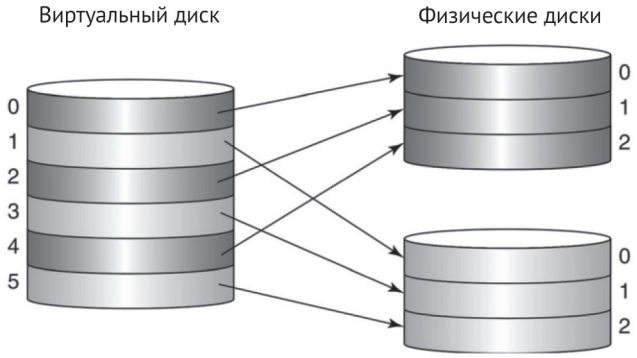


Рис. 3.3. Чередование дисков

Чередование дисков – эффективный прием, потому что равномерно распределяет базу данных между небольшими дисками. Если поступит запрос на доступ к произвольному сектору, он будет отправлен одному из маленьких дисков с равной вероятностью. А если поступит несколько запросов на доступ к смежным секторам, они будут отправлены разным дискам. Благодаря этому диски будут загружены более или менее равномерно.

3.1.4. Увеличение надежности дисков зеркалированием

Пользователи базы данных надеются, что их данные останутся в безопасности на диске и не будут потеряны или повреждены. К сожалению, дисковые накопители иногда выходят из строя. Магнитный материал на пластине может вырождаться, что приводит к появлению сбойных секторов. Проникновение пыли в корпус привода или ударные воздействия на него могут привести к тому, что головка чтения/записи будет царапать пластину, разрушая сектора и разрушаясь сама.

---

[1] Большинство контроллеров позволяет пользователям определять шаг чередования, не ограничивая их размером сектора. Например, хорошим выбором может стать размер дорожки, особенно если приводы осуществляют кеширование дорожек, а не секторов. Оптимальный размер шага чередования зависит от многих факторов и часто определяется методом проб и ошибок.

Наиболее очевидный способ защиты от сбоя дисков – хранение копии содержимого. Например, можно создавать ночные резервные копии диска; когда диск выйдет из строя, вы просто купите новый диск и скопируете на него резервную копию. Проблема, однако, в том, что при этом потеряются все изменения, имевшие место между моментом резервного копирования диска и моментом его выхода из строя. Единственный способ обойти эту проблему – копировать каждое изменение на резервный диск в тот момент, когда оно происходит. Другими словами, нужно хранить две идентичные версии диска; эти версии называют *зеркалами*.

Так же как в случае чередования, для управления двумя зеркальными дисками необходим контроллер. Когда система базы данных выполняет чтение с диска, контроллер может обратиться к указанному сектору на любом диске. Когда выполняется запись, контроллер записывает одни и те же данные на оба диска. Теоретически эти две операции записи могут выполняться параллельно, без дополнительных затрат времени. На практике, однако, важно, чтобы запись на зеркала происходила последовательно, дабы защититься от сбоя системы. Проблема в том, что если в процессе записи на диск произойдет сбой системы, содержимое сектора потеряется. То есть если запись производить на оба зеркала одновременно, можно потерять обе копии сектора, а если запись производить последовательно, то по крайней мере одно из зеркал останется неповрежденным.

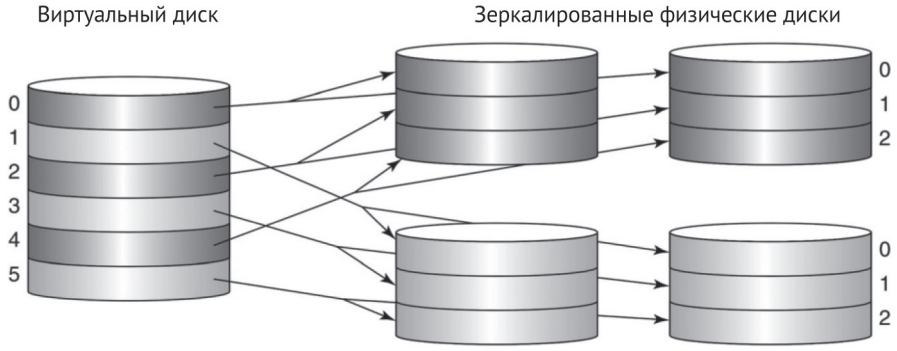


Рис. 3.4. Чередование дисков с зеркалированием

Предположим, что один из дисков в зеркальной паре вышел из строя. Администратор базы данных сможет восстановить систему, выполнив следующую процедуру:

1. Остановить систему.
2. Заменить вышедший из строя диск новым.
3. Скопировать данные с исправного диска на новый.
4. Запустить систему.

К сожалению, эта процедура не является абсолютно надежной. Все еще существует вероятность потери данных, если исправный диск выйдет из строя в процессе копирования на новый диск. Вероятность выхода из строя обоих дисков в течение пары часов невелика (для современных дисков она равна примерно 1/60 000), но для очень важных баз данных даже такой не-

большой риск может оказаться неприемлемым. Риск можно уменьшить, используя три зеркальных диска вместо двух. В этом случае данные будут потеряны, только если все три диска выйдут из строя в течение той же пары часов; такая вероятность, хоть и ненулевая, настолько низка, что ею можно просто пренебречь.

Зеркалирование можно совместить с чередованием дисков. Зеркалирование чередующихся дисков – распространенная стратегия. Например, можно организовать хранение 40 Гбайт данных на четырех приводах емкостью по 20 Гбайт: два привода будут чередоваться, а два других – служить зеркалами для чередующихся приводов. Такая конфигурация обеспечивает высокую скорость и надежность (см. рис. 3.4).

3.1.5. Увеличение надежности дисков путем контроля четности

Недостаток зеркалирования заключается в необходимости хранения полной копии данных, для чего требуется вдвое больше дисков. Эта проблема особенно заметна при использовании чередования дисков – если вы решите хранить 300 Гбайт данных на 15 дисках объемом 20 Гбайт, вам придется купить еще 15 дисков для зеркалирования. В конфигурациях с большими базами данных часто создается огромный виртуальный диск, чередующий множество маленьких дисков, поэтому перспектива покупки еще такого же количества дисков для зеркалирования не выглядит привлекательной. Лучше было бы иметь возможность восстановления данных с неисправного диска без использования большого количества зеркальных дисков.

На самом деле есть способ использовать один диск для резервного копирования любого количества других дисков. Достигается это путем сохранения на резервном диске информации о *четности*. Четность определяется для набора S бит следующим образом:

- четность для множества из S бит равна 1, если это множество содержит нечетное число битов со значением 1;
- четность для множества из S бит равна 0, если это множество содержит четное число битов со значением 1.

Иначе говоря, если добавить бит четности к множеству S бит, результат всегда будет содержать четное число единиц.

Четность обладает следующим интересным и важным свойством: значение любого бита можно определить по значениям других битов, если известна четность. Например, пусть $S = \{1, 0, 1\}$. Четность для множества S равна 0, потому что оно содержит четное число единиц. Допустим, вы потеряли значение первого бита. Поскольку четность равна 0, множество {?, 0, 1} должно иметь четное число единиц; отсюда следует, что потерянный бит должен иметь значение 1. Аналогичные вычисления можно сделать для любого другого бита (включая бит четности).

Это свойство четности можно распространить на диски. Предположим, у вас есть $N + 1$ дисков одинакового размера. Вы выбираете один из дисков на роль диска четности, а на остальных N дисках организуете хранение данных с чередованием. Каждый бит на диске четности вычисляется с использованием

соответствующих битов на всех других дисках. Если какой-то диск выйдет из строя (включая диск четности), содержимое этого диска можно восстановить, просканировав, бит за битом, содержимое других дисков (см. рис. 3.5).

Диски управляются контроллером. Запросы на чтение и запись обрабатываются в основном так же, как при чередовании, – контроллер определяет, какой диск содержит требуемый сектор, и выполняет операцию чтения/записи. Разница лишь в том, что операции записи должны также обновлять соответствующий сектор диска четности. Контроллер может вычислять новое значение четности, определяя, какие биты в данном секторе изменились, руководствуясь простым правилом: если бит изменился, то соответствующий бит четности тоже должен измениться. Соответственно, чтобы выполнить запись сектора, контроллеру потребуется четырежды обратиться к диску: прочитать сектор с данными и соответствующий сектор контроля четности (чтобы вычислить новые биты контроля четности) и записать новое содержимое обоих секторов.

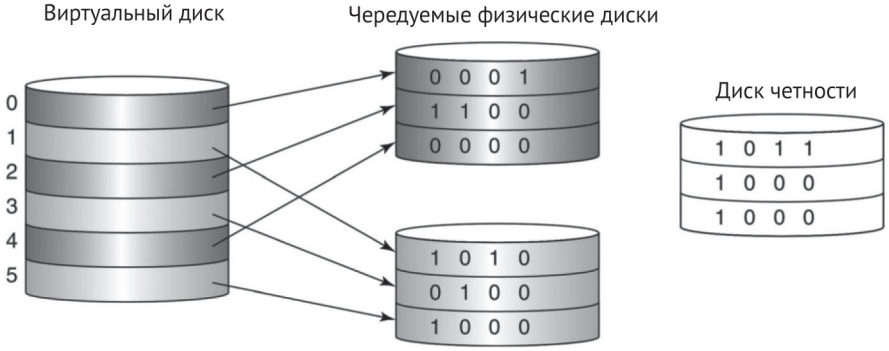


Рис. 3.5. Чередование дисков с контролем четности

Такое использование информации о четности может показаться волшебством, в том смысле, что, добавив всего один диск, можно надежно создавать резервные копии любого количества других дисков. Однако это волшебство имеет два недостатка.

Первый недостаток использования контроля четности заключается в значительных затратах времени на операцию записи, потому что для этого требуется выполнить чтение/запись с двух дисков. Как показывает опыт, использование контроля четности снижает эффективность чередования примерно на 20 %. Второй недостаток: с контролем четности база данных более уязвима в случае выхода из строя сразу нескольких дисков. Представьте ситуацию, когда один диск вышел из строя, – для восстановления информации необходимы все остальные диски, и отказ любого из них станет катастрофой. Если база данных хранится на большом количестве маленьких дисков (скажем, 100), тогда вероятность выхода из строя второго диска более чем реальна. Сравните эту ситуацию с зеркалированием, когда для восстановления поврежденного диска достаточно, чтобы его зеркало не вышло из строя, что гораздо менее вероятно.

3.1.6. RAID

В предыдущих разделах мы рассмотрели три способа использования нескольких дисков: *чередование* – для ускорения доступа к диску, *зеркалирование* и *четность* – для защиты от сбоев. Эти стратегии основаны на использовании контроллера, скрывающего существование нескольких дисков от операционной системы и создающего иллюзию единого виртуального диска. Контроллер отображает каждую виртуальную операцию чтения/записи в одну или несколько операций с физическими дисками. Контроллер может быть реализован программно или аппаратно, но аппаратные контроллеры более распространены.

Все эти стратегии входят в набор стратегий, известный под названием *RAID*, от англ. *Redundant Array of Inexpensive Disks* (массив недорогих дисков с избыточностью). Всего имеется семь уровней RAID:

- *RAID-0* – чередование без какой-либо защиты от выхода дисков из строя. Если один из дисков в таком массиве выйдет из строя, есть риск потерять всю базу данных;
- *RAID-1* – зеркалирование;
- *RAID-2* использует чередование битов вместо чередования секторов и механизм избыточности на основе кодов Хемминга с исправлением ошибок вместо четности. Эта стратегия сложна в реализации и имеет низкую производительность. В настоящее время практически не используется;
- *RAID-3* и *RAID-4* используют чередование и четность. Отличаются только тем, что RAID-3 использует чередование байтов, а RAID-4 – чередование секторов. Чередование секторов, как правило, дает большую эффективность, потому что соответствует единице доступа к диску;
- *RAID-5* – то, что и RAID-4, но вместо сохранения информации о четности на отдельном диске распределяет ее по дискам с данными. То есть для случая *N* дисков каждый *N*-й сектор каждого диска содержит информацию о четности. Эта стратегия более эффективна, чем RAID-4, благодаря отсутствию единственного диска четности, который может стать узким местом (см. упражнение 3.5);
- *RAID-6* – то же, что и RAID-5, но хранит два вида информации о четности. Как следствие эта стратегия способна обрабатывать одновременный сбой двух дисков, но для хранения дополнительной информации о четности необходим отдельный диск.

Наибольшей популярностью пользуются RAID-1 и RAID-5. Выбор между ними на самом деле заключается в выборе между зеркалированием и контролем четности. Зеркалирование, как правило, предпочтительнее для баз данных, потому что, во-первых, обеспечивает более высокую скорость и надежность и, во-вторых, потому что стоимость дисков постоянно снижается.

3.1.7. Твердотельные накопители

Дисковые накопители широко используются в современных системах баз данных, но они имеют непреодолимый недостаток – в них используются механические компоненты: вращающиеся пластины и актуатор. Этот недостаток

делает дисковые накопители медленными, по сравнению с электронной памятью, и чувствительными к ударам и вибрации.

Флеш-память – более новая технология, которая со временем может заменить дисковые накопители. В ней используются полупроводниковые компоненты, как в ОЗУ, но, в отличие от ОЗУ, флеш-память не требует бесперебойного питания. Поскольку в ней отсутствуют механические компоненты, флеш-память позволяет читать/записывать данные гораздо быстрее, чем дисковые накопители.

Современные твердотельные накопители имеют время поиска около 50 мс, то есть примерно в 100 раз меньше, чем у дисковых накопителей. Скорость передачи данных твердотельных накопителей зависит от интерфейса шины, к которой она подключена. Твердотельные накопители, подключаемые к быстрым внутренним шинам, в этом отношении можно сравнить с дисковыми накопителями; однако внешние USB-накопители работают медленнее, чем дисковые.

Флеш-память подвержена изнашиванию. Каждую ячейку можно перезаписать ограниченное число раз; попытка записи в ячейку, исчерпавшую свой ресурс, приведет к отказу твердотельного накопителя. В настоящее время этот предел исчисляется миллионами циклов перезаписи, чего достаточно для большинства приложений баз данных. В высококачественных накопителях используются методы «выравнивания износа», когда при перезаписи одного и того же байта автоматически выбираются разные ячейки, чтобы обеспечить их равномерный износ. Этот метод позволяет накопителю сохранять работоспособность, пока все ячейки не достигнут предела перезаписи.

Твердотельные накопители поддерживают абстракцию дисковых секторов, благодаря чему с точки зрения операционной системы выглядят как дисковые накопители. К твердотельным накопителям можно применять технологии RAID, хотя чередование в данном случае менее важно, потому что время доступа к твердотельным накопителям очень мало.

Основным препятствием к внедрению твердотельных накопителей является их стоимость. В настоящее время единица объема флеш-памяти стоит примерно в 100 раз больше, чем единица объема дискового накопителя. Однако стоимость твердотельных накопителей и дисковых технологий будет продолжать снижаться, и в конечном итоге твердотельные накопители станут достаточно дешевыми, чтобы их можно было рассматривать как выгодную альтернативу дискам. После этого дисковые накопители могут переместиться в нишу архивных хранилищ и хранилищ очень больших баз данных.

Флеш-память также можно использовать как промежуточное хранилище, в дополнение к дисковым накопителям. Если база данных полностью помещается во флеш-память, дисковый накопитель не будет использоваться. Но по мере увеличения базы данных редко используемые сектора будут перемещаться на диск.

С точки зрения движка базы данных, твердотельный накопитель обладает теми же свойствами, что и диск: он способен долго хранить данные, работает довольно медленно и обеспечивает доступ к данным по секторам. (Просто он не такой медленный, как диск.) Поэтому в оставшейся части этой книги я буду придерживаться текущей терминологии и называть постоянную память «диском».

3.2. Интерфейс блочного доступа к диску

Диски могут иметь разные аппаратные характеристики, например сектора на разных дисках могут иметь разные размеры и адресоваться по-разному. Обработку всех этих деталей берет на себя операционная система, предоставляя приложениям простой интерфейс доступа к дискам.

Центральным для этого интерфейса является понятие *блока*. Блок похож на сектор, за исключением того, что его размер определяется операционной системой. Все блоки имеют одинаковый фиксированный размер для всех дисков. Операционная система поддерживает все необходимые преобразования между блоками и секторами. Каждому блоку на диске она присваивает свой, уникальный *номер блока*, по которому определяет фактические сектора.

Содержимое блока нельзя получить непосредственно с диска. Для этого операционная система должна прочитать все сектора, составляющие блок, в *страницу* памяти, после чего приложение сможет получить доступ к нему. Чтобы изменить содержимое блока, клиент должен прочитать его в страницу памяти, изменить его в этой странице, а затем записать страницу обратно в блок на диске.

Операционная система обычно предлагает несколько методов доступа к дисковым блокам:

- `readblock(n,p)` – читает байты из блока `n` на диске в страницу памяти `p`;
- `writeblock(n,p)` – записывает байты из страницы в памяти `p` в блок `n` на диске;
- `allocate(k,n)` – находит непрерывный фрагмент из `k` неиспользуемых блоков на диске, отмечает их как используемые и возвращает номер первого блока. Фрагмент с новыми блоками выбирается так, чтобы они находились максимально близко к блоку `n`;
- `deallocate(k,n)` – отмечает непрерывную последовательность из `k` блоков, начиная с блока `n`, как неиспользуемые.

Операционная система запоминает, какие блоки на диске используются, а какие нет. Для этого применяются две основные стратегии: карта диска и список свободных блоков.

Карта диска – это последовательность битов, каждый из которых соответствует одному блоку на диске. Значение 1 в бите означает, что блок свободен, а 0 – занят. Карта диска хранится на диске, обычно в первых нескольких блоках. Операционная система может освободить блок с номером n, просто записав 1 в n-й бит в карте диска. Она может выделить k смежных блоков, отыскав в карте диска последовательность из k бит со значением 1, и затем записать в эти биты значение 0.

Список свободных блоков – это цепочка *фрагментов*, каждый из которых представляет непрерывную последовательность незанятых блоков. В первом блоке каждого фрагмента хранятся два значения: длина фрагмента и номер блока следующего фрагмента в цепочке[1]. Первый блок диска хранит указатель

[1] Незанятые блоки могут использоваться операционной системой для любых своих целей. В данном случае она использует первые 8 байт блока для хранения двух целых чисел.

на первый фрагмент в цепочке. Когда операционная система получает запрос на выделение *k* смежных блоков, она ищет в списке фрагмент достаточного размера и затем исключает из списка весь фрагмент или только его часть размером *k*. Когда операционная система получает запрос на освобождение последовательности блоков, она просто вставляет ее в список свободных блоков.

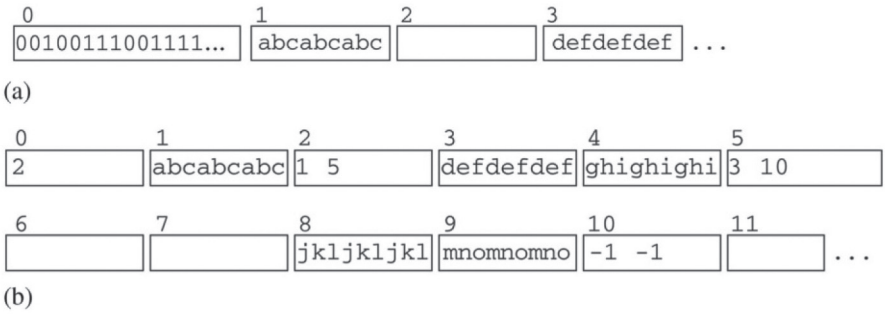


Рис. 3.6. Две стратегии хранения информации о свободных блоках: (a) карта диска; (b) список свободных блоков

На рис. 3.6 показано применение этих двух методов для работы с диском, на котором уже заняты блоки с номерами 0, 1, 3, 4, 8 и 9. В верхней части (a) показана карта диска, хранящаяся в блоке 0; значение 0 в бите указывает, что соответствующий ему блок занят. В нижней части (b) изображен соответствующий список свободных блоков. Значение 2 в блоке 0 указывает, что первый свободный фрагмент начинается с блока 2. Значения 1 и 5 в блоке 2 указывают, что соответствующий фрагмент содержит 1 блок, а следующий фрагмент начинается в блоке 5. Аналогично, блок 5 сообщает, что начинающийся в нем фрагмент включает 3 блока, а следующий свободный фрагмент находится в блоке 10. Значение –1 в блоке 10 указывает, что это последний фрагмент, и он содержит все оставшиеся блоки.

Стратегия на основе списка свободных блоков требует меньше дополнительного пространства; достаточно просто сохранить целое число в блоке 0, определяющее первый свободный блок в списке. Стратегия на основе карты требует выделить на диске место для хранения карты. Схема на рис. 3.6a предполагает, что карта может уместиться в один блок, однако вообще для карты может потребоваться выделить несколько блоков (см. упражнение 3.7). Преимущество карты дисков в том, что она позволяет операционной системе иметь более полную картину о местонахождении незанятых блоков на диске. Например, стратегия на основе карты диска часто предпочтительнее, когда операционная система должна поддерживать возможность обслуживать параллельные запросы на выделение блоков.

3.3. Интерфейс файлов для доступа к диску

Операционная система предлагает еще один интерфейс доступа к диску, более высокого уровня, который называют *файловой системой*. С точки зрения клиента файл – это именованная последовательность байтов. На этом уровне от-

сутствует понятие блока, и клиент имеет возможность читать (или записывать) любое количество байтов, начиная с любой позиции в файле.

В языке Java имеется класс `RandomAccessFile`, предоставляющий типичный API для доступа к файловой системе. Каждый объект `RandomAccessFile` хранит текущую позицию в файле, с которой начнется следующая операция чтения или записи. Текущую позицию можно явно изменить вызовом метода `seek`. Вызов метода `readInt` (или `writeInt`) также переместит текущую позицию на размер целого числа.

Примером может служить фрагмент кода в листинге 3.1, который увеличивает целое число, хранящееся в байтах 7992–7995 в файле «junk». Вызов `readInt` читает целое число, начиная с байта 7992, и перемещает текущую позицию за него, в байт 7996. Последующий вызов `seek` возвращает текущую позицию обратно в байт 7992, после чего метод `writeInt` записывает новое целое число в это же место.

Обратите внимание, что вызовы `readInt` и `writeInt` выглядят так, будто они обращаются к диску напрямую, – они скрывают тот факт, что обращения к дисковым блокам производятся через страницы памяти. Обычно операционная система резервирует несколько страниц памяти для собственного использования; эти страницы называются *буферами ввода/вывода*. Когда файл открывается, операционная система создает для него свой буфер ввода/вывода, без ведома клиента.

Интерфейс файлов позволяет рассматривать файлы как последовательности блоков. Например, если блоки имеют длину 4096 байт (то есть 4 Кбайт), тогда байт 7992 будет находиться в блоке 1 файла (то есть во втором блоке). Ссылки на блоки, такие как «блок 1 файла», называют ссылками на *логические* блоки, потому что они сообщают номер блока в файле, а не фактический номер блока на диске.

Для конкретного местоположения в файле метод `seek` определяет фактический блок диска, содержащий это местоположение. В частности, `seek` выполняет два преобразования:

- преобразует заданную позицию в номер логического блока;
- преобразует номер логического блока в номер физического блока.

Первое преобразование выполняется просто: номер логического блока – это заданная позиция, деленная на размер блока. Например, для блоков с размером 4 Кбайт байт в позиции 7992 находится в блоке 1, потому что 7992/4096 = 1 (целочисленное деление).

Второе преобразование сложнее и зависит от особенностей реализации файловой системы. В оставшейся части этого раздела мы рассмотрим три стратегии реализации файлов: *непрерывное размещение*, *размещение на основе экстентов* и *индексированное размещение*. Каждая из этих стратегий предполагает хранение информации о размещении файлов на диске в *каталоге файловой системы*. Метод `seek` обращается к блокам в этом каталоге, когда преобразует номера логических блоков в номера физических блоков. Эти обращения к диску можно рассматривать как скрытые «накладные расходы», налагаемые файловой системой. Операционные системы пытаются минимизировать эти издержки, но не могут устранить их полностью.

Листинг 3.7. Использование интерфейса файлов для доступа к диску

```
RandomAccessFile f = new RandomAccessFile("junk", "rws");
f.seek(7992);
int n = f.readInt();
f.seek(7992);
f.writeInt(n+1);
f.close();
```

Непрерывное размещение

Непрерывное размещение – самая простая стратегия. В этом случае каждый файл хранится в непрерывной последовательности блоков. При использовании стратегии непрерывного размещения каталог файловой системы хранит длину каждого файла и местоположение его первого блока. Преобразование логических номеров блоков в физические выполняется просто: если файл начинается в дисковом блоке `b`, тогда *N*-й блок файла находится в дисковом блоке `b + N`. В табл. 3.1 показан каталог файловой системы, содержащей два файла: файл «junk» с длиной 48 блоков, который начинается с блока 32, и файл «temp» с длиной 16 блоков, который начинается с блока 80.

Непрерывное размещение имеет две проблемы. Первая заключается в невозможности расширения файла, если сразу за ним следует другой файл. Файл «junk» в табл. 3.1 является примером такого файла. Из-за этого клиенты должны создавать свои файлы с максимальным количеством блоков, которое им может понадобиться, что приводит к напрасному расходованию дискового пространства, когда файл заполнен не до конца. Эта проблема известна как *внутренняя фрагментация*. Вторая проблема заключается в появлении на диске, по мере его заполнения, множества мелких незанятых фрагментов и отсутствии больших фрагментов. Из-за этого может не получиться создать большой файл, даже если на диске много свободного места. Эта проблема известна как *внешняя фрагментация*. Другими словами:

- внутренняя фрагментация – это напрасное расходование дискового пространства внутри файлов;
- внешняя фрагментация – это напрасное расходование дискового пространства между файлами.

Размещение на основе экстентов

Стратегия размещения на основе экстентов – это частный случай стратегии непрерывного размещения, уменьшающий внутреннюю и внешнюю фрагментацию. В этом случае операционная система хранит файл в виде последовательности экстентов фиксированной длины, где каждый экстент является непрерывной последовательностью блоков. Файл расширяется на величину одного экстента за раз. Каталог файловой системы в этой стратегии хранит для каждого файла список первых блоков всех его экстентов.

Например, предположим, что операционная система хранит файлы в экстентах по 8 блоков в каждом. В табл. 3.2 показан каталог файловой системы с двумя файлами: «junk» и «temp». Эти файлы имеют тот же размер, что и в предыдущем примере, но теперь разбиты на экстенты. Файл «junk» имеет шесть экстентов, а файл «temp» – два экстента.

Таблица 3.1. Каталог файловой системы при использовании стратегии непрерывного размещения

Имя	Первый блок	Длина
junk	32	48
temp	80	16

Таблица 3.2. Каталог файловой системы при использовании стратегии на основе экстентов

Имя	Экстенты
junk	32, 480, 696, 72, 528, 336
temp	64, 8

Чтобы найти номер блока на диске, содержащий *N*-й блок файла, метод seek ищет в каталоге файловой системы список экстентов для этого файла; затем в списке экстентов находит экстент, содержащий блок *N*, из которого потом вычисляет местоположение блока. Например, рассмотрим каталог файлов в табл. 3.2. Местоположение блока 21 файла «junk» можно вычислить так:

1) блок 21 находится в экстенте 2 файла, потому что 21/8 = 2 (целочисленное деление);
2) экстент 2 начинается в логическом блоке 2 × 8 = 16 файла;
3) то есть блок 21 находится в блоке 21 − 16 = 5 этого экстента;
4) как определено в списке экстентов файла, экстент 2 начинается в физическом блоке 696;
5) таким образом, логический блок 21 находится в физическом блоке 696 + 5 = 701.

Стратегия размещения на основе экстентов уменьшает внутреннюю фрагментацию, потому что в этом случае для каждого файла впустую расходуется не более одного экстента. Внешняя фрагментация вообще отсутствует, поскольку все экстенты имеют одинаковый размер.

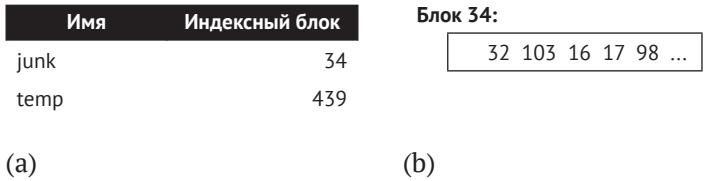
Индексированное размещение

Стратегия индексированного размещения использует другой подход – она не пытается размещать файлы в последовательных блоках. Вместо этого каждый блок файла (или, если хотите, экстент длиной в один блок) выделяется индивидуально. Операционная система реализует эту стратегию, выделяя специальный *индексный блок* для каждого файла, в котором перечисляются дисковые блоки, занятые этим файлом. То есть индексный блок `ib` можно представить как массив целых чисел, где значение `ib[N]` определяет номер дискового блока, содержащего логический блок `N` файла. Благодаря такому подходу местоположение любого логического блока вычисляется тривиально – номер соответствующего физического блока просто извлекается из индексного блока.

На рис. 3.7a показан каталог файловой системы с двумя файлами: «junk» и «temp». Роль индексного блока для файла «junk» играет блок 34. На рис. 3.7b перечислены первые несколько целых чисел в этом блоке. Судя по этому рисунку, блок 1 файла «junk» находится в дисковом блоке 103.

Преимущество данного подхода заключается в том, что блоки выделяются по одному за раз, поэтому фрагментация отсутствует. Основная проблема за-

ключается в ограничении максимального размера файлов, потому что в индексном блоке может храниться ограниченное число номеров физических блоков. Файловая система UNIX решает эту проблему, поддерживая несколько уровней индексации, что позволяет увеличить максимальный размер файла до огромных значений (см. упражнения 3.12 и 3.13).



Имя	Индексный блок
junk	34
temp	439

(a)

Блок 34:

32 103 16 17 98 ...

(b)

Рис. 3.7. Каталог файловой системы при использовании стратегии индексированного размещения: (a) таблица каталога; (b) содержимое индексного блока 34

3.4. Система баз данных и операционная система

Операционная система предлагает два способа доступа к диску: на уровне блоков и на уровне файлов. Какой уровень предпочтительнее для разработчиков движка базы данных?

Преимущество доступа на уровне блоков состоит в том, что он дает полный контроль над использованием дисковых блоков. Например, часто используемые блоки могут храниться в середине диска, где время позиционирования меньше. Аналогично блоки, к которым нередко обращаются вместе, могут храниться рядом друг с другом. Еще одно преимущество состоит в том, что движок базы данных не подвержен ограничениям, которые операционная система накладывает на файлы, что позволяет ему хранить таблицы, превышающие по размеру ограничения операционных систем или охватывающие несколько дисков.

С другой стороны, использование интерфейса блоков имеет существенный недостаток – сложность реализации стратегии, потому что в этом случае требуется отформатировать и смонтировать диск как *неструктурированный*, то есть без файловой системы, а для этого необходимо, чтобы администратор базы данных обладал обширными знаниями о закономерностях доступа к блокам для точной настройки системы.

Другой крайностью является использование только файловой системы. Например, каждая таблица может храниться в отдельном файле, и движок будет обращаться к записям, используя операции с файлами. Реализовать эту стратегию намного проще, к тому же она позволяет скрыть фактический доступ к диску от системы баз данных. Однако такая ситуация недопустима по двум причинам. Во-первых, система баз данных должна знать, где находятся границы блоков, чтобы наиболее эффективно сохранять и извлекать данные. Во-вторых, система баз данных должна управлять своими собственными страницами, потому что механизм управления буферами ввода/вывода в операционной системе не подходит для запросов к базе данных. Мы еще вернемся к этим проблемам в следующих главах.

Компромисс заключается в хранении всех данных в одном или нескольких файлах операционной системы и работе с ними, как если бы они были неструктурированными дисками. То есть система баз данных обращается к своему «диску», используя логические файловые блоки. Операционная система отвечает за отображение каждого логического блока в соответствующий физический блок с помощью метода `seek`. Поскольку при проверке каталога файловой системы метод `seek` может вызывать обращения к диску, система баз данных не будет полностью контролировать диск. Однако эти дополнительные издержки обычно незначительны, по сравнению с большим количеством обращений к блокам в системе баз данных. Итак, система баз данных может использовать высокоуровневый интерфейс операционной системы, сохраняя за собой значительный контроль над доступом к диску.

Эта компромиссная стратегия используется во многих системах баз данных. Microsoft Access хранит все в одном файле `.mdb`, тогда как Oracle, Derby и SimpleDB используют несколько файлов.

3.5. Диспетчер файлов в SimpleDB

Компонент движка базы данных, взаимодействующий с операционной системой, называется *диспетчером файлов*. В этом разделе мы рассмотрим диспетчер файлов в SimpleDB. В разделе 3.5.1 вы увидите, как клиенты используют диспетчер файлов, а в разделе 3.5.2 мы исследуем его реализацию.

3.5.1. Использование диспетчера файлов

База данных SimpleDB хранится в нескольких файлах. Для каждой таблицы и для каждого индекса создается свой файл. Также для базы данных в целом создается файл журнала и несколько файлов каталогов. Диспетчер файлов в SimpleDB реализует блочный доступ к этим файлам и находится в пакете `simpledb.file`. В этом пакете определено три класса: `BlockId`, `Page` и `FileMgr`. Их API показан в листинге 3.2.

Листинг 3.2. API диспетчера файлов в SimpleDB

BlockId

```
public BlockId(String filename, int blknum);
public String filename();
public int    number();
```

Page

```
public Page(int blocksize);
public Page(byte[] b);
public int    getInt(int offset);
public byte[] getBytes(int offset);
public String getString(int offset);
public void   setInt(int offset, int val);
public void   setBytes(int offset, byte[] val);
public void   setString(int offset, String val);
public int    maxLength(int strlen);
```

FileMgr

```
public FileMgr(String dbDirectory, int blocksize);
public void    read(BlockId blk, Page p);
public void    write(BlockId blk, Page p);
public BlockId append(String filename);
public boolean isNew();
public int     length(String filename);
public int     blockSize();
```

Объект `BlockId` определяет конкретный физический блок по имени файла и номеру логического блока. Например, инструкция

```
BlockId blk = new BlockId("student.tbl", 23)
```

создаст ссылку на блок 23 файла `student.tbl`. Методы `filename` и `number` возвращают имя файла и номер блока сответственно.

Объект `Page` хранит содержимое дискового блока. Его первый конструктор создает страницу, в которую помещается содержимое из буфера ввода/вывода операционной системы; этот конструктор используется диспетчером буферов. Второй конструктор создает страницу, в которую помещается содержимое из массива Java; этот конструктор используется главным образом диспетчером журнала. Различные методы `get` и `set` позволяют клиентам сохранять или извлекать значения из указанных местоположений в странице. Страница может хранить три типа значений: целые числа, строки и «двоичные объекты» (то есть произвольные массивы байтов). При желании можно добавить дополнительные методы для других типов данных (см. упражнение 3.17). Клиент может сохранить значение по любому смещению от начала страницы, но несет всю ответственность за знание, где и какие значения были сохранены. Попытка получить значение по неправильному смещению будет иметь непредсказуемые результаты.

Класс `FileMgr` осуществляет фактическое взаимодействие с файловой системой. Его конструктор принимает два аргумента: строку с именем базы данных и целое число, обозначающее размер каждого блока. Имя базы данных используется в качестве имени папки, где хранятся файлы базы данных; эта папка должна находиться в текущем каталоге движка. Если такой папки не существует, движок создаст папку для новой базы данных. В этом случае метод `isNew` возвращает `true`, иначе – `false`. Этот метод необходим для правильной инициализации новой базы данных.

Метод `read` читает содержимое указанного блока в указанную страницу. Метод `write` выполняет обратную операцию: записывает содержимое страницы в указанный блок. Метод `length` возвращает количество блоков в указанном файле.

Движок имеет один объект диспетчера файлов `FileMgr`, который создается во время запуска системы. Его создает класс `SimpleDB` (в пакете `simpledb.server`), а метод `fileMgr` этого класса возвращает созданный объект.

Класс `FileTest` в листинге 3.3 иллюстрирует использование этих методов. Этот код делится на три части. В первой части инициализируется объект `SimpleDB`. Его конструктору передаются три аргумента: имя базы данных «studentdb», размер блока 400 байт и размер пула, равный 8 буферам. Размер блока в 400 байт используется в SimpleDB по умолчанию. Этот размер специально выбран таким маленьким, чтобы упростить создание демонстрационных баз

данных, содержащих много блоков. В коммерческой системе баз данных это значение желательно установить равным размеру блока в операционной системе; типичный размер блока составляет 4 Кбайт. Пул буферов будет обсуждаться в главе 4.

Во второй части в листинге 3.3 выполняется запись строки «abcdefghijklm» в блок 2 файла «testfile» со смещением 88. Затем вызывается метод `maxLength`, чтобы определить максимальную длину этой строки и вычислить смещение свободного места в блоке за ней. Затем в это место записывается целое число 345.

В третьей части производится чтение этого блока в другую страницу и извлечение из нее двух значений.

Листинг 3.3. Тестирование диспетчера файлов в SimpleDB

```
public class FileTest {
    public static void main(String[] args) throws IOException {
        SimpleDB db = new SimpleDB("filetest", 400, 8);
        FileMgr fm = db.fileMgr();

        BlockId blk = new BlockId("testfile", 2);
        Page p1 = new Page(fm.blockSize());
        int pos1 = 88;
        p1.setString(pos1, "abcdefghijklm");
        int size = Page.maxLength("abcdefghijklm".length());
        int pos2 = pos1 + size;
        p1.setInt(pos2, 345);
        fm.write(blk, p1);

        Page p2 = new Page(fm.blockSize());
        fm.read(blk, p2);
        System.out.println("offset " + pos2 +
                           " contains " + p2.getInt(pos2));
        System.out.println("offset " + pos1 +
                           " contains " + p2.getString(pos1));
    }
}
```

3.5.2. Реализация диспетчера файлов

В этом подразделе исследуются реализации трех классов диспетчера файлов.

Класс BlockId

Код класса `BlockId` показан в листинге 3.4. Кроме простых методов `fileName` и `number`, класс также реализует методы `equals`, `hashCode` и `toString`.

Листинг 3.4. Код класса BlockId в SimpleDB

```
public class BlockId {
    private String filename;
    private int blknum;

    public BlockId(String filename, int blknum) {
        this.filename = filename;
        this.blknum   = blknum;
    }
```

```
    public String fileName() {
        return filename;
    }
    public int number() {
        return blknum;
    }

    public boolean equals(Object obj) {
        BlockId blk = (BlockId) obj;
        return filename.equals(blk.filename) && blknum == blk.blknum;
    }

    public String toString() {
        return "[file " + filename + ", block " + blknum + "]";
    }

    public int hashCode() {
        return toString().hashCode();
    }
}
```

Класс Page

Код с реализацией класса `Page` показан в листинге 3.5. Каждый объект `Page`, представляющий страницу памяти, использует Java-объект `ByteBuffer`, представляющий массив байтов с методами для чтения и записи данных с произвольным смещением от начала массива. Данные могут быть элементарными значениями (например, целыми числами) или массивами байтов меньшего размера. Например, метод `setInt` класса `Page` записывает целое число в страницу, вызывая метод `putInt` класса `ByteBuffer`. Метод `setBytes` класса `Page` записывает массив байтов в виде двух значений: количества байтов в заданном массиве, а затем самого массива. Он вызывает метод `putInt` класса `ByteBuffer` для записи целого числа с размером массива и метод `put` для записи самого массива.

Класс `ByteBuffer` не имеет методов для чтения и записи строк, поэтому `Page` записывает и извлекает строковые значения в виде массивов байтов. Класс `String` в Java имеет метод `getBytes`, который преобразует строку в массив байтов; он также имеет конструктор, преобразующий массив байтов обратно в строку. Соответственно, метод `setString` класса `Page` вызывает `getBytes`, чтобы преобразовать строку в байты, а затем записывает эти байты в виде массива. Аналогично метод `getString` класса `Page` извлекает массив байтов из буфера, а затем преобразует их в строку.

Порядок преобразования строк в байты и обратно определяется *кодировкой символов*. Существует несколько стандартных кодировок, таких как ASCII или UTF-16. Класс `Charset` в Java содержит объекты, реализующие большинство из этих кодировок. Конструктор класса `String` и его метод `getBytes` принимают аргумент `Charset`. В листинге 3.5 можно видеть, что `Page` использует кодировку ASCII, но вы можете изменить константу `CHARSET` и использовать свою кодировку.

Кодировка определяет, сколько байтов будет использоваться для представления каждого символа. В ASCII каждый символ представлен одним

байтом, тогда как в UTF-16 используется от 2 до 4 байтов на символ. То есть движок базы данных может не знать точно, сколько байтов займет заданная строка. Метод `maxLength` класса `Page` вычисляет максимальный размер массива байтов для строки, в который уместится указанное количество символов. Это достигается умножением количества символов на максимальное число байтов на символ и добавлением 4 байтов для целого числа, которое хранит размер массива.

Листинг 3.5. Код класса Page в SimpleDB

```
public class Page {
    private ByteBuffer bb;
    public static final Charset CHARSET = StandardCharsets.US_ASCII;

    // Конструктор для создания страниц с данными
    public Page(int blocksize) {
        bb = ByteBuffer.allocateDirect(blocksize);
    }

    // Конструктор для создания страниц с журнальными записями
    public Page(byte[] b) {
        bb = ByteBuffer.wrap(b);
    }

    public int getInt(int offset) {
        return bb.getInt(offset);
    }

    public void setInt(int offset, int n) {
        bb.putInt(offset, n);
    }

    public byte[] getBytes(int offset) {
        bb.position(offset);
        int length = bb.getInt();
        byte[] b = new byte[length];
        bb.get(b);
        return b;
    }

    public void setBytes(int offset, byte[] b) {
        bb.position(offset);
        bb.putInt(b.length);
        bb.put(b);
    }

    public String getString(int offset) {
        byte[] b = getBytes(offset);
        return new String(b, CHARSET);
    }

    public void setString(int offset, String s) {
        byte[] b = s.getBytes(CHARSET);
        setBytes(offset, b);
    }
```

```
    public static int maxLength(int strlen) {
        float bytesPerChar = CHARSET.newEncoder().maxBytesPerChar();
        return Integer.BYTES + (strlen * (int)bytesPerChar);
    }

    // приватный метод пакета, необходим для FileMgr
    ByteBuffer contents() {
        bb.position(0);
        return bb;
    }
}
```

Массив байтов, лежащий в основе объекта `ByteBuffer`, может быть получен либо из массива Java, либо из буферов ввода/вывода операционной системы. Класс `Page` имеет два конструктора, каждый из которых создает массив байтов своего вида. Поскольку буферы ввода/вывода являются ценным ресурсом, использование первого конструктора тщательно контролируется диспетчером буферов и будет обсуждаться в следующей главе. Другие компоненты движка базы данных (такие как диспетчер журналов) используют другой конструктор.

Класс FileMgr

Код класса `FileMgr` показан в листинге 3.6. Его главная цель – дать методы для чтения и записи страниц в дисковые блоки. Его метод `read` ищет соответствующую позицию в указанном файле и читает содержимое блока в буфер байтов указанной страницы. Метод `write` действует аналогично. Метод `append` ищет конец файла и записывает в него пустой массив байтов, что заставляет операционную систему расширить файл. Обратите внимание, что диспетчер файлов всегда читает или записывает байты целыми блоками и всегда начинает с границы блока. При этом он гарантирует, что при каждом обращении к методам `read`, `write` и `append` будет выполняться только одно обращение к диску.

Каждый объект `RandomAccessFile` в ассоциативном массиве `openFiles` соответствует открытому файлу. Обратите внимание, что файлы открываются в режиме «rws», где пара символов «rw» (read/write) указывает, что файл открывается для чтения и записи, а символ «s» (sync) – что операционная система не должна задерживать операцию дискового ввода/вывода для оптимизации производительности: каждая операция записи должна немедленно записать данные на диск. Такой подход гарантирует, что движок базы данных будет точно знать, когда происходят записи на диск, что особенно важно для реализации алгоритмов восстановления данных, описываемых в главе 5.

Методы `read`, `write` и `append` выполняются *синхронно*, то есть в каждый конкретный момент только один поток может выполнять эти методы. Синхронность необходима для поддержания согласованности, когда методы совместно используют обновляемые объекты, такие как `RandomAccessFile`. Например, если не синхронизировать чтение, может произойти следующая ситуация: два клиента JDBC, выполняющихся в своих собственных потоках, пытаются прочитать разные блоки из одного файла. Поток A начинает первым. Он приступает к операции чтения, но прерывается сразу после вызова `f.seek`, то есть он установил текущую позицию в файле, но еще ничего не прочитал из

него. Поток B начинает вторым и выполняет операцию чтения от начала до конца. Когда поток A возобновит выполнение, текущая позиция в файле изменится, но поток не заметит этого; в результате он прочитает данные не из того блока.

В SimpleDB есть только один объект `FileMgr`, который создается конструктором `SimpleDB` из пакета `simpledb.server`. Конструктор `FileMgr` определяет, существует ли указанная папка базы данных, и создает ее при необходимости. Конструктор также удаляет любые временные файлы, которые могли быть созданы операторами материализации (см. главу 13).

Листинг 3.6. Код класса FileMgr для SimpleDB

```
public class FileMgr {
    private File dbDirectory;
    private int blocksize;
    private boolean isNew;
    private Map<String,RandomAccessFile> openFiles = new HashMap<>();

    public FileMgr(File dbDirectory, int blocksize) {
        this.dbDirectory = dbDirectory;
        this.blocksize = blocksize;
        isNew = !dbDirectory.exists();

        // создать каталог, если создается новая база данных
        if (isNew)
            dbDirectory.mkdirs();

        // удалить оставшиеся временные таблицы
        for (String filename : dbDirectory.list())
            if (filename.startsWith("temp"))
                new File(dbDirectory, filename).delete();
    }

    public synchronized void read(BlockId blk, Page p) {
        try {
            RandomAccessFile f = getFile(blk.fileName());
            f.seek(blk.number() * blocksize);
            f.getChannel().read(p.contents());
        }
        catch (IOException e) {
            throw new RuntimeException("cannot read block " + blk);
        }
    }

    public synchronized void write(BlockId blk, Page p) {
        try {
            RandomAccessFile f = getFile(blk.fileName());
            f.seek(blk.number() * blocksize);
            f.getChannel().write(p.contents());
        }
        catch (IOException e) {
            throw new RuntimeException("cannot write block" + blk);
        }
    }
```

```
    public synchronized BlockId append(String filename) {
        int newblknum = size(filename);
        BlockId blk = new BlockId(filename, newblknum);
        byte[] b = new byte[blocksize];
        try {
            RandomAccessFile f = getFile(blk.fileName());
            f.seek(blk.number() * blocksize);
            f.write(b);
        }
        catch (IOException e) {
            throw new RuntimeException("cannot append block" + blk);
        }
        return blk;
    }

    public int length(String filename) {
        try
            RandomAccessFile f = getFile(filename);
            return (int)(f.length() / blocksize);
        }
        catch (IOException e) {
            throw new RuntimeException("cannot access " + filename);
        }
    }

    public boolean isNew() {
        return isNew;
    }

    public int blockSize() {
        return blocksize;
    }

    private RandomAccessFile getFile(String filename)
                throws IOException {
        RandomAccessFile f = openFiles.get(filename);
        if (f == null) {
            File dbTable = new File(dbDirectory, filename);
            f = new RandomAccessFile(dbTable, "rws");
            openFiles.put(filename, f);
        }
        return f;
    }
}
```

3.6. Итоги

- *Дисковый накопитель* содержит одну или несколько вращающихся *пластин*. На пластине имеются концентрические *дорожки*, и каждая дорожка разбита на *сектора*. Размер сектора определяется производителем диска, но обычно выбирается размер, равный 512 байт.
- Для каждой пластины имеется своя головка чтения/записи. Головки не имеют возможности перемещаться независимо – все они закреплены на одном *актуаторе*, который перемещает их одновременно на одну и ту же дорожку на каждой пластине.

- Накопитель осуществляет доступ к диску в три этапа:
  - перемещает головку на заданную дорожку за время, которое называется *временем позиционирования*;
  - ждет, пока пластина повернется так, что первый желаемый байт окажется под головкой; это называется *задержкой вращения*;
  - пока пластина продолжает вращаться, головка читает (или записывает) байты, проносящиеся под головкой; это время называется *временем передачи*.
- Дисковые накопители работают медленно, потому что имеют механическую часть. Время доступа к диску можно сократить, используя такие методы, как *кеширование*, организация хранения в *цилиндрах* и *чередование дисков*. Кеширование основано на предварительной выборке секторов за счет чтения дорожки целиком. Цилиндр состоит из дорожек с одинаковым номером на всех пластинах. Для доступа к блокам в одном и том же цилиндре не требуется тратить время на позиционирование головок. Чередование дисков заключается в распределении содержимого виртуального диска между несколькими маленькими физическими дисками. Ускорение достигается за счет того, что физические диски могут работать одновременно.
- Для увеличения надежности дисков можно использовать методы *RAID*. Вот основные уровни RAID:
  - *RAID-0* – чередование без защиты от выхода дисков из строя. Если один из дисков в таком массиве выйдет из строя, есть риск потерять всю базу данных;
  - *RAID-1* – зеркалирование. Для каждого диска в массиве имеется его зеркальная копия, которую можно использовать, если диск выйдет из строя;
  - *RAID-4* – используется чередование и один дополнительный диск для контроля четности. Если диск выйдет из строя, его можно будет восстановить, используя информацию с других дисков и с диска контроля четности.
- Технология RAID требует применения *контроллера*, скрывающего существование нескольких дисков от операционной системы и создающего иллюзию единого виртуального диска. Контроллер отображает каждую виртуальную операцию чтения/записи в одну или несколько операций с физическими дисками.
- *Флеш-память* бросает вызов дисковым технологиям. Флеш-память способна долго хранить данные и действует быстрее, чем диски, потому что состоит только из электронных компонентов. Однако флеш-память все еще значительно медленнее ОЗУ, поэтому операционные системы действуют с твердотельными накопителями так же, как с дисковыми.
- Операционная система скрывает физические детали организации дисков и твердотельных накопителей от своих клиентов, предоставляя блочный интерфейс доступа к ним. *Блок* похож на сектор, но его размер определяется операционной системой. Клиент обращается к содержимому устройства по номеру блока. Операционная система запоминает, какие блоки на диске свободны, используя либо *карту диска*, либо список *свободных блоков*.

- *Страница* – это область памяти размером с блок. Клиент изменяет блок, читая его содержимое в страницу, изменяя страницу, а затем записывая страницу обратно в блок.
- Операционная система также предоставляет интерфейс файлов. Клиент видит файл как именованную последовательность байтов.
- Операционная система может реализовывать файлы, используя три стратегии: *непрерывное размещение*, *размещение на основе экстентов* и *индексированное размещение*. При непрерывном размещении каждый файл хранится в непрерывной последовательности блоков. При размещении на основе экстентов каждый файл хранится в виде последовательности экстентов, где каждый экстент является непрерывной последовательностью блоков. При индексированном размещении каждый блок выделяется для файла отдельно. Для каждого файла также имеется специальный индексный блок, хранящий номера дисковых блоков, выделенных для этого файла.
- Система баз данных может использовать блочный или файловый интерфейс для работы с диском. Хорошим компромиссом является хранить данные в файлах, но обращаться к файлам на уровне блоков.

3.7. Для дополнительного чтения

В статье Chen et al. (1994) приводится подробный обзор различных стратегий RAID и их характеристик производительности. Замечательное обсуждение файловых систем в UNIX можно найти в книге von Hagen (2002), а описание NTFS для Windows – в книге Nagar (1997). Также краткие обзоры различных реализаций файловых систем можно найти во многих учебниках по операционным системам, таких как Silberschatz et al. (2004).

Флеш-память обладает одним нежелательным свойством: перезапись существующего значения выполняется значительно медленнее, чем запись совершенно нового значения. По этой причине было проведено множество исследований в сфере создания файловых систем на основе флеш-памяти, которые не перезаписывают значения. Такие файловые системы хранят обновления в журнале, который рассматривается в главе 4. Описание этих исследований можно найти в статьях Wu и Kuo (2006) и Lee and Moon (2007).

Chen, P., Lee, E., Gibson, G., & Patterson, D. (1994) «RAID: High-performance, reliable secondary storage». ACM Computing Surveys, 26 (2), 145–185.

Lee, S., & Moon, B. (2007) «Design of flash-based DBMS: An in-page logging approach». Материалы конференции ACM-SIGMOD Conference, стр. 55–66.

Nagar, R. (1997) «Windows NT file system internals». O'Reilly.

Silberschatz, A., Gagne, G., & Galvin, P. (2004) «Operating system concepts». Addison Wesley.

von Hagen, W. (2002) «Linux filesystems». Sams Publishing.

Wu, C., & Kuo, T. (2006) «The design of efficient initialization and crash recovery for log-based file systems over flash memory». ACM Transactions on Storage, 2 (4), 449–467.

3.8. Упражнения

Теория

3.1. Пусть есть дисковый накопитель с одной пластиной, содержащей 50 000 дорожек и вращающейся со скоростью 7200 об/мин. Каждая дорожка содержит 500 секторов, а каждый сектор – 512 байт.

a) Определите емкость пластины.
b) Определите среднюю задержку вращения.
c) Определите максимальную скорость передачи.

3.2. Пусть есть диск емкостью 80 Гбайт, вращающийся со скоростью 7200 об/мин и имеющий скорость передачи 100 Мбайт/с. Предположим, что каждая дорожка содержит одинаковое количество байтов.

a) Сколько байтов содержит каждая дорожка? Сколько дорожек имеется на диске?
b) Определите скорость передачи, если скорость вращения диска увеличить до 10 000 об/мин.

3.3. Пусть есть 10 дисков емкостью по 20 Гбайт. Каждый имеет 500 секторов на дорожку. Предположим, что вы решили создать виртуальный диск объемом 200 ГБайт с чередованием, но с чередованием целых дорожек, а не отдельных секторов.

a) Предположим, что контроллер получает запрос на доступ к виртуальному сектору M. Напишите формулу, вычисляющую соответствующий фактический номер диска и сектора.
b) Объясните, когда чередование дорожек может быть эффективнее чередования секторов.
c) Объясните, когда чередование дорожек может оказаться менее эффективно, чем чередование секторов.

3.4. Все процедуры восстановления после сбоя, описанные в этой главе, требуют выключения системы на время замены неисправного диска. Во многих системах такие простои недопустимы, но при этом в них также желательно избегать риска потери данных.

a) Обдумайте простую стратегию зеркалирования. Опишите алгоритм восстановления неисправного зеркала без простоя. Увеличивает ли ваш алгоритм риск отказа второго диска? Что нужно сделать, чтобы уменьшить этот риск?
b) Измените стратегию контроля четности, чтобы ее можно было использовать для устранения простоев. Как в этом случае уменьшить риск отказа второго диска?

3.5. Одним из следствий применения стратегии контроля четности в RAID-4 является тот факт, что каждая операция записи инициирует обращение к диску четности. Одним из предлагаемых улучшений является исключение диска четности и «чередование» данных с информацией о четности на всех дисках. Например, сектора 0, N, 2N, … диска 0 будут содержать информацию о четности, так же как сектора 1, N+1, 2N+1, … диска 1, и т. д. Это улучшение называется RAID-5.

a) Предположим, что диск вышел из строя. Объясните, как его можно восстановить.
b) Покажите, что даже с этим улучшением для чтения и записи на диск все равно требуется такое же количество обращений к диску, как и в RAID-4.
c) Объясните, почему это улучшение тем не менее увеличивает эффективность доступа к диску.

3.6. Рассмотрите рис. 3.5. Предположим, что один из чередующихся дисков вышел из строя. Объясните, как восстановить его содержимое, используя диск четности.

3.7. Пусть имеется база данных объемом 1 Гбайт, которая хранится в файле с размером блока 4 Кбайт.
a) Сколько блоков содержит файл?
b) Предположим, что система баз данных использует карту диска для управления свободными блоками. Сколько дополнительных блоков необходимо для хранения карты диска?

3.8. Рассмотрите рис. 3.6. Нарисуйте карту диска и список свободных блоков после выполнения следующих операций:

```
allocate(1,4); allocate(4,10); allocate(5,12);
```

3.9. На рис. 3.8 показана система RAID-4, в которой один из дисков вышел из строя. Используйте диск четности, чтобы восстановить утерянные значения.

3.10. Стратегия на основе списка свободных блоков может привести к тому, что в списке окажется два смежных блока.
a) Объясните, как изменить технику управления списком, чтобы можно было объединять смежные фрагменты.
b) Объясните, почему объединение свободных фрагментов является хорошей идеей при непрерывном размещении файлов.
c) Объясните, почему объединение не имеет смысла при размещении файлов на основе экстентов или индексированном размещении.

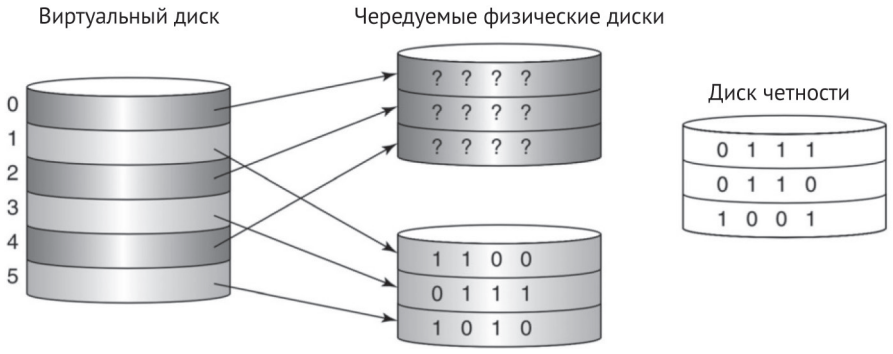


Рис. 3.8. Неисправный физический диск в массиве RAID-4

3.11. Предположим, что операционная система размещает файлы на основе экстентов с размером 12 и список экстентов для файла имеет вид: [240, 132, 60, 252, 12, 24].
   a) Определите размер файла.
   b) Вычислите номера физических блоков для логических блоков 2, 12, 23, 34 и 55.

3.12. Представьте реализацию индексированного размещения файлов. Допустим, что размер блока равен 4 Кбайт. Каков максимально возможный размер файла?

3.13. Элемент каталога, соответствующий файлу, в UNIX ссылается на блок, который называется *inode* (индексный узел). В одной из реализаций inode начало блока содержит заголовок с различной информацией, а последние 60 байт содержат 15 целых чисел. Первые 12 – это физическое местоположение первых 12 блоков данных в файле. Следующие два числа – это местоположение двух индексных блоков, и последнее целое число – это местоположение *индексного блока второго уровня*. Индексный блок содержит только номера блоков данных в файле; индексный блок второго уровня – только номера индексных блоков (которые, в свою очередь, ссылаются на блоки данных).
   a) Если предположить, что блок имеет размер 4 Кбайт, на какое количество блоков данных может сослаться индексный блок?
   b) Если не учитывать индексный блок второго уровня, какой максимальный размер могут иметь файлы в UNIX?
   c) На какое количество блоков данных можно сослаться, используя индексный блок второго уровня?
   d) Какой максимальный размер могут иметь файлы в UNIX?
   e) Сколько обращений к блокам потребуется для чтения последнего блока данных из файла объемом 1 Гбайт?
   f) Представьте алгоритм для функции `seek` в UNIX.

3.14. Название фильма и песни «On a clear day you can see forever» («В ясный день увидишь вечность») иногда искажают до «On a clear disk you can seek forever» («На чистом диске будешь искать вечность»). Прокомментируйте меткость этого каламбура.

Практика

3.15. Системы баз данных часто содержат диагностические процедуры.
   a) Измените класс `FileMgr` так, чтобы он позволял получать полезную статистику, например количество прочитанных/записанных блоков. Добавьте новые методы в класс, возвращающие эту статистику.
   b) Измените методы `commit` и `rollback` в классе `RemoteConnectionImpl` (в пакете `simpledb.jdbc.network`), чтобы они выводили эту статистику. Сделайте то же самое в классе `EmbeddedConnection` (в пакете `simpledb.jdbc.embedded`). В результате движок должен выводить статистику для каждого выполняемого оператора SQL.

3.16. Методы `setInt`, `setBytes` и `setString` класса `Page` не проверяют, умещается ли новое значение в страницу.
   a) Добавьте в код эти проверки. Что делать, если проверка не пройдена?
   b) Объясните, почему предпочтительнее отказаться от проверок.

3.17. Класс `Page` имеет методы для получения/записи целых чисел, двоичных объектов (массивов байтов) и строк. Добавьте поддержку других типов, таких как короткие целые числа, логические значения и даты.

3.18. Класс `Page` поддерживает строки, преобразуя их в массивы байтов. Другой способ реализовать такую поддержку – записывать каждый символ отдельно, добавляя в конец символ-разделитель. Роль такого символа-разделителя в Java может играть «\0». Измените класс соответственно.

Глава 4

Управление памятью

В этой главе рассматриваются два компонента движка базы данных: *диспетчер журнала* и *диспетчер буферов*. Каждый из них отвечает за определенные файлы: диспетчер журнала отвечает за файл журнала, а диспетчер буферов – за файлы данных.

Оба компонента сталкиваются с проблемой эффективного управления чтением дисковых блоков в оперативную память и записью из оперативной памяти. Обычно объем базы данных намного больше объема оперативной памяти, поэтому этим компонентам может потребоваться перемещать блоки в память и из нее. В этой главе рассматриваются их потребности и используемые ими алгоритмы управления памятью. Диспетчер журнала поддерживает только последовательный доступ к файлу журнала и имеет простой, оптимальный алгоритм управления памятью. Диспетчер буферов, напротив, должен поддерживать произвольный доступ к пользовательским файлам, что является гораздо более сложной задачей.

4.1. Два принципа управления памятью баз данных

Как вы помните, прочитать значение с диска движок базы данных может, только прочитав блок с этим значением в страницу памяти, а записать значение на диск – только записав измененную страницу обратно в соответствующий блок. При перемещении данных между диском и памятью движки баз данных следуют двум важным принципам: *минимизировать число обращений к диску* и *не полагаться на виртуальную память*.

Принцип 1: минимизация числа обращений к диску

Рассмотрим приложение, которое читает данные с диска, просматривает их, выполняет вычисления, вносит изменения и записывает данные обратно. Как оценить, сколько времени это займет? Как вы помните, оперативная память работает в 1000 раз быстрее флеш-памяти и в 100 000 раз быстрее диска. Это означает, что в большинстве ситуаций время, необходимое для чтения/записи дискового блока, окажется не меньше времени обработки блока в ОЗУ. Поэтому основное, что может сделать движок базы данных, – это минимизировать число обращений к диску.

Один из способов минимизировать число обращений к диску – предотвратить многократное обращение к одним и тем же дисковым блокам. Это распространенная проблема, и она имеет стандартное решение, известное как *кеши-*

рование. Например, микропроцессор имеет локальный аппаратный кеш ранее выполнявшихся инструкций; если следующая инструкция находится в кеше, процессору не придется загружать ее из ОЗУ. Другой пример: браузер хранит кеш ранее посещавшихся веб-страниц; если пользователь запросит страницу, находящуюся в кеше (скажем, нажав кнопку «Назад» в браузере), браузер сможет извлечь ее из кеша, не тратя времени на загрузку из сети.

Для кеширования дисковых блоков движок базы данных использует страницы памяти. Запоминая блоки в страницах памяти, движок может быстро удовлетворить запрос клиента, используя существующую страницу, и тем самым избежать чтения с диска. Аналогично запись страниц на диск движок производит только в случае явной необходимости, надеясь одной операцией записи на диск сохранить сразу несколько изменений в странице.

Необходимость минимизировать число обращений к диску настолько важна, что пронизывает всю реализацию движка базы данных. Например, для извлечения данных подбираются специальные алгоритмы, способные сэкономить на обращениях к диску. И когда для выполнения конкретного SQL-запроса имеется несколько стратегий поиска, планировщик выберет стратегию, которая, по его мнению, потребует наименьшего числа обращений к диску.

Принцип 2: не полагаться на виртуальную память

Современные операционные системы поддерживают *виртуальную память*. Операционная система создает у каждого процесса иллюзию, что он имеет очень большой объем памяти, в которой можно хранить код и данные. Процесс произвольно размещает объекты в своем пространстве виртуальной памяти, а операционная система отображает каждую виртуальную страницу в страницу физической памяти.

Объем виртуальной памяти, поддерживаемой операционной системой, обычно намного больше объема физической памяти компьютера. Поскольку не все виртуальные страницы могут уместиться в физической памяти, операционная система вынуждена сохранять некоторые из них на диске. Когда процесс обращается к виртуальной странице, отсутствующей в физической памяти, выполняется операция *подкачки*. Операционная система выбирает физическую страницу, записывает ее содержимое на диск (если оно было изменено) и читает в нее содержимое затребованной виртуальной страницы.

Самый простой способ организовать управление дисковыми блоками в движке базы данных – разместить каждый блок на своей виртуальной странице. Например, движок может создать массив страниц для каждого файла, выделив по одной странице для каждого блока. Такие массивы получались бы огромными, хотя и помещались бы в виртуальную память. Когда системе баз данных потребуется обратиться к этим страницам, механизм виртуальной памяти будет при необходимости вытеснять их на диск и загружать с диска. Это простая и легко реализуемая стратегия. К сожалению, у нее есть серьезный недостаток: управление записью страниц на диск осуществляется операционной системой, а не движком базы данных. Из-за этого недостатка возникает две проблемы.

Первая проблема заключается в том, что стратегия подкачки страниц в операционной системе может ухудшить способность движка базы данных восстанавливаться после сбоя системы. Причина, как вы увидите в главе 5, в том,

что измененная страница имеет несколько связанных с ней записей в журнале, и эти записи *должны* быть перенесены на диск до записи самой страницы. (Иначе записи в журнале нельзя будет использовать для восстановления базы данных после сбоя из-за их отсутствия.) Поскольку операционная система ничего не знает о журнале, она может вытеснить измененную страницу с данными раньше, чем страницу с журнальными записями, и тем самым создать препятствие для механизма восстановления[1].

Вторая проблема заключается в том, что операционная система не знает, какие страницы используются в данный момент, а какие больше не нужны движку базы данных. Операционная система может попытаться угадать и, например, вытеснить страницу, которая не использовалась дольше всех. Но если догадка окажется неверной, она вытеснит страницу, которая может потребоваться буквально через доли секунды, и тогда будет выполнено два ненужных обращения к диску. Движок базы данных, напротив, лучше знает, какие страницы нужны, и может делать гораздо более обоснованные предположения.

Поэтому движок базы данных должен сам управлять своими страницами. С этой целью он выделяет небольшое количество страниц, которые точно уместятся в физической памяти; эти страницы известны как *пул буферов* базы данных. Движок запоминает, какие страницы можно вытеснить, и когда возникает необходимость прочитать дисковый блок, движок (а не операционная система) выбирает доступную страницу из пула, сбрасывает ее содержимое (и журнальные записи) на диск, если необходимо, и только затем читает требуемый блок.

4.2. Управление журналом

Всякий раз, когда пользователь изменяет что-то в базе данных, движок должен запомнить это изменение на случай, если его придется отменить. Сведения, описывающие изменение, заносятся в *журнальные записи*, а журнальные записи сохраняются в *файле журнала*. Новые журнальные записи всегда добавляются в конец журнала.

Диспетчер журнала – это компонент движка базы данных, отвечающий за сохранение журнальных записей в файл журнала. Он не анализирует содержимое записей – это прерогатива диспетчера восстановления, который описывается в главе 5. Диспетчер журнала рассматривает журнал просто как постоянно увеличивающуюся последовательность записей.

В этом разделе мы посмотрим, как диспетчер журнала управляет памятью, добавляя журнальные записи в файл журнала. Ниже приводится простой (и неэффективный алгоритм) добавления новых записей в конец журнала.

1. Создать страницу в памяти.
2. Прочитать в эту страницу последний блок из файла журнала.

3a. Если в странице достаточно места, добавить новую запись после последней и записать блок на диск.

3b. Если места недостаточно, создать пустую страницу, поместить в нее новую запись и добавить как новый блок в конце файла журнала.

---

[1] На самом деле некоторые операционные системы решают эту проблему, но они не относятся к числу распространенных.

Этот алгоритм требует выполнить по одной операции чтения с диска и записи на диск для каждой записи, добавляемой в журнал. Он прост, но очень неэффективен. На рис. 4.1 показано, как алгоритм выполняет этап 3а. Файл журнала содержит три блока, в которых поместилось восемь записей: от r1 до r8. Размеры записей могут отличаться, поэтому в блок 0 уместилось четыре записи, а в блок 1 – только три. Блок 2 еще не заполнен и содержит только одну запись. Содержимое блока 2 находится в странице памяти. В страницу r8 только что была добавлена новая запись (r9).

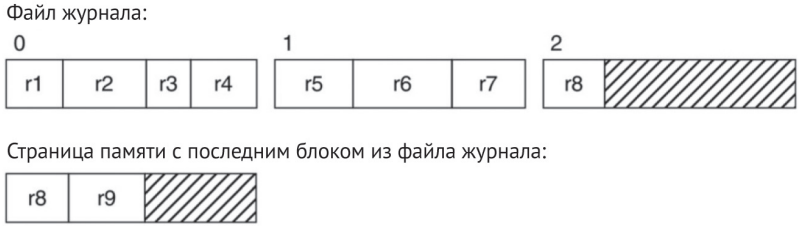


Рис. 4.1. Добавление новой записи (r9) в журнал

Предположим теперь, что диспетчер журнала завершает алгоритм, записывая страницу обратно в блок 2 файла. Когда диспетчер журнала получит запрос на добавление еще одной записи в файл, он выполнит шаги 1 и 2 алгоритма и прочитает блок 2 в страницу памяти. Но обратите внимание, что эта операция чтения с диска совершенно не нужна, потому что страница уже содержит блок 2! То есть шаги 1 и 2 алгоритма не нужны. Диспетчеру журнала достаточно выделить одну постоянную страницу для хранения содержимого последнего блока журнала. Благодаря этому все операции чтения с диска исключаются.

Количество операций записи на диск тоже можно уменьшить. В алгоритме выше диспетчер журнала записывает свою страницу на диск всякий раз, когда в нее добавляется новая запись. Глядя на рис. 4.1, можно заметить, что нет никакой необходимости сразу сохранять запись r9 на диск. Можно продолжать добавлять новые записи, пока в странице остается свободное место, а когда она заполнится, диспетчер журнала может записать страницу на диск, очистить ее и начать использовать заново. Этот новый алгоритм требует выполнять только одну операцию записи на диск для каждого блока журнала, что, безусловно, более оптимально.

Однако у данного алгоритма есть одно упущение: страницу журнала может потребоваться записать на диск до того, как она будет заполнена, из-за обстоятельств, не зависящих от диспетчера журнала. Дело в том, что диспетчер буферов не может записать на диск измененную страницу с данными, пока не будут сохранены соответствующие журнальные записи. Если одна из этих записей находится в странице журнала, но еще не была сохранена на диске, тогда диспетчер журнала должен будет записать свою страницу на диск, независимо от того, заполнена она или нет. Этот вопрос мы рассмотрим в главе 5.

Ниже приводится усовершенствованный алгоритм управления журналом. Он предусматривает два случая записи страницы памяти на диск: когда журнальная запись обязана быть сохранена на диск и когда страница заполнена.

Соответственно, страница памяти может записываться в один и тот же блок файла журнала несколько раз. Но так как эти операции записи абсолютно необходимы и их нельзя избежать, можно сделать вывод, что алгоритм является оптимальным.

1) Выделить одну постоянную страницу в памяти для хранения содержимого последнего блока файла журнала. Назовем эту страницу P.
2) Получив запрос на добавление новой записи в журнал:
   a) если в P недостаточно места, то записать P на диск и очистить ее содержимое;
   b) добавить новую запись в P.
3) Когда система баз данных потребует сохранить конкретную запись на диск:
   a) проверить, находится ли эта запись в P;
   b) если да, записать P на диск.

4.3. Диспетчер журнала в SimpleDB

В этом разделе рассматривается диспетчер журнала системы баз данных SimpleDB. Раздел 4.3.1 иллюстрирует использование диспетчера, а в разделе 4.3.2 рассматривается его реализация.

4.3.1. API диспетчера журнала

Реализация диспетчера журнала SimpleDB находится в пакете `simpledb.log`. В этом пакете определяется класс `LogMgr`, API которого показан в листинге 4.1. Движок базы данных создает единственный объект `LogMgr` во время запуска системы и передает конструктору ссылку на диспетчер файлов и имя файла журнала.

Метод `append` добавляет запись в журнал и возвращает целое число. С точки зрения диспетчера журнала, запись – это массив байтов произвольного размера; он сохраняет массив в файл журнала, но не знает, что означает его содержимое. Единственное ограничение – массив должен умещаться внутри страницы. Возвращаемое значение метода `append` является идентификатором новой записи, который называется *порядковым номером журнала* (Log Sequence Number, LSN).

Вызов метода `append` не гарантирует, что запись немедленно будет сохранена на диске; диспетчер журнала сам выбирает, когда сохранить записи на диск, как описано в алгоритме управления журналом выше. Клиент может форсировать сохранение конкретной записи, вызвав метод `flush` и передав ему аргумент с номером LSN записи. Этот метод гарантирует немедленное сохранение этой и всех предыдущих записей на диск.

Для чтения записей из журнала клиент может использовать метод `iterator`; этот метод возвращает Java-итератор, выполняющий обход записей в журнале. Каждый вызов метода `next` итератора будет возвращать массив байтов со следующей записью из журнала. Записи возвращаются в обратном порядке, начиная с самой последней. Такой порядок возврата записей обусловлен особенностями диспетчера восстановления.

Листинг 4.1. API диспетчера журнала в SimpleDB

LogMgr

```
public LogMgr(FileMgr fm, String logfile);
public int  append(byte[] rec);
public void flush(int lsn);
public Iterator<byte[]> iterator();
```

Класс `LogTest` в листинге 4.2 демонстрирует порядок использования API диспетчера журнала. Он добавляет в журнал 70 записей, каждая из которых состоит из строки и целого числа. Целое число – это номер записи N, а строка состоит из слова «record» и номера N: «recordN». После добавления первых 35 записей код выводит их на экран, а затем, после добавления всех 70 записей, снова выводит их.

Запустив этот код, вы обнаружите, что первый вызов `printLogRecords` напечатает только 20 записей. Причина в том, что в первый блок журнала уместилось 20 записей, и он был сброшен на диск при создании 21-й записи. Остальные 15 записей остались в странице памяти и не были сброшены[1]. Второй вызов `createRecords` создаст записи с 36 по 70. Вызов `flush` требует от диспетчера журнала сохранить запись с номером 65 на диск. А так как записи 66–70 находятся в той же странице памяти, что и запись 65, они также записываются на диск. Как результат второй вызов `printLogRecords` напечатает все 70 записей в обратном порядке.

Обратите внимание, как метод `createLogRecord` выделяет массив байтов для журнальной записи. Он создает объект-обертку `Page` для массива, чтобы получить возможность использовать методы `setInt` и `setString` страницы и с их помощью поместить строку и целое число в запись, и возвращает готовый массив байтов. Аналогично, метод `printLogRecords` создает объект-обертку `Page` для записи из журнала, чтобы с его помощью извлечь строку и целое число из записи.

4.3.2. Реализация диспетчера журнала

Код `LogMgr` показан в листинге 4.3. Его конструктор принимает аргумент со строкой и интерпретирует ее как имя файла журнала. Если файл журнала пуст, конструктор добавляет в него новый пустой блок. Также конструктор выделяет в памяти одну страницу (`logpage`) и инициализирует ее содержимым последнего блока в файле журнала.

Как вы помните, записи в журнале идентифицируются порядковыми номерами журнала (LSN). Метод `append` назначает номера LSN последовательно, начиная с 1, запоминая последний назначенный номер в переменной `latestLSN`. Диспетчер журнала хранит следующий доступный номер LSN и номер LSN последней записи в журнале, сохраненной на диск. Метод `flush` сравнивает последний сохраненный номер LSN с полученным в аргументе. Если номер LSN в аргументе меньше, значит, заданная запись уже была сохранена на диск; в противном случае он записывает страницу `logpage` на диск, а последний назначенный номер LSN становится последним, сохраненным на диске.

---

[1] В приведенной реализации диспетчера журнала (листинг 4.3) итератор вызывает метод flush, поэтому будут выведены все 35 записей. – *Прим. ред.*

Метод append вычисляет размер записи, чтобы определить, поместится ли она в текущую страницу. Если нет, то записывает текущую страницу на диск и вызывает appendNewBlock, чтобы очистить страницу и добавить ее в конец файла журнала. Эта стратегия немного отличается от алгоритма управления журналом, описанного выше: диспетчер журнала расширяет файл журнала, добавляя пустую страницу вместо заполненной. Эту стратегию проще реализовать, потому что она позволяет методу flush предположить, что блок уже находится на диске.

Листинг 4.2. Тестирование диспетчера журнала

```
public class LogTest {
    private static LogMgr lm;

    public static void main(String[] args) {
        SimpleDB db = new SimpleDB("logtest", 400, 8);
        lm = db.logMgr();
        createRecords(1, 35);
        printLogRecords("The log file now has these records:");
        createRecords(36, 70);
        lm.flush(65);
        printLogRecords("The log file now has these records:");
    }

    private static void printLogRecords(String msg) {
        System.out.println(msg);
        Iterator<byte[]> iter = lm.iterator();
        while (iter.hasNext()) {
            byte[] rec = iter.next();
            Page p = new Page(rec);
            String s = p.getString(0);
            int npos = Page.maxLength(s.length());
            int val = p.getInt(npos);
            System.out.println("[" + s + ", " + val + "]");
        }
        System.out.println();
    }

    private static void createRecords(int start, int end) {
        System.out.print("Creating records: ");
        for (int i=start; i<=end; i++) {
            byte[] rec = createLogRecord("record"+i, i+100);
            int lsn = lm.append(rec);
            System.out.print(lsn + " ");
        }
        System.out.println();
    }

    private static byte[] createLogRecord(String s, int n) {
        int npos = Page.maxLength(s.length());
        byte[] b = new byte[npos + Integer.BYTES];
        Page p = new Page(b);
        p.setString(0, s);
        p.setInt(npos, n);
        return b;
    }
}
```

Листинг 4.3. Класс LogMgr в SimpleDB

```
public class LogMgr {
    private FileMgr fm;
    private String logfile;
    private Page logpage;
    private BlockId currentblk;
    private int latestLSN = 0;
    private int lastSavedLSN = 0;

    public LogMgr(FileMgr fm, String logfile) {
        this.fm = fm;
        this.logfile = logfile;
        byte[] b = new byte[fm.blockSize()];
        logpage = new Page(b);
        int logsize = fm.length(logfile);
        if (logsize == 0)
            currentblk = appendNewBlock();
        else {
            currentblk = new BlockId(logfile, logsize-1);
            fm.read(currentblk, logpage);
        }
    }

    public void flush(int lsn) {
        if (lsn >= lastSavedLSN)
            flush();
    }

    public Iterator<byte[]> iterator() {
        flush();
        return new LogIterator(fm, currentblk);
    }

    public synchronized int append(byte[] logrec) {
        int boundary = logpage.getInt(0); // текущая граница
        int recsize = logrec.length;
        int bytesneeded = recsize + Integer.BYTES;
        if (boundary - bytesneeded < Integer.BYTES) { // Не умещается?
            flush();                  // тогда перейти к следующему блоку.
            currentblk = appendNewBlock();
            boundary = logpage.getInt(0);
        }
        int recpos = boundary - bytesneeded;
        logpage.setBytes(recpos, logrec);
        logpage.setInt(0, recpos); // новая граница
        latestLSN += 1;
        return latestLSN;
    }

    private BlockId appendNewBlock() {
        BlockId blk = fm.append(logfile);
        logpage.setInt(0, fm.blockSize());
        fm.write(blk, logpage);
        return blk;
    }

    private void flush() {
        fm.write(currentblk, logpage);
        lastSavedLSN = latestLSN;
    }
}
```

Обратите внимание, что метод append размещает записи в странице справа налево. Переменная boundary хранит смещение последней добавленной записи. Такая стратегия позволяет итератору журнала читать записи в обратном порядке, выполняя обход слева направо. Граничное значение (из переменной boundary) записывается в первые четыре байта страницы, чтобы итератор знал, где начинаются записи.

Метод iterator сбрасывает журнал на диск (чтобы гарантировать, что весь журнал находится на диске), а затем возвращает объект LogIterator. Класс LogIterator, реализующий итератор, является приватным на уровне пакета; его код показан в листинге 4.4. Объект LogIterator создает страницу для хранения содержимого блока журнала. Конструктор позиционирует итератор на первую запись в последнем блоке журнала (то есть, как вы помните, на последнюю запись в журнале). Метод next выполняет переход к следующей записи в странице; если перед этим итератор ссылался на последнюю запись в странице, он загружает в страницу предыдущий блок и возвращает первую запись. Метод hasNext возвращает false, если в странице больше нет записей и нет предыдущих блоков.

Листинг 4.4. Код класса LogIterator в SimpleDB

```
class LogIterator implements Iterator<byte[]> {
    private FileMgr fm;
    private BlockId blk;
    private Page p;
    private int currentpos;
    private int boundary;

    public LogIterator(FileMgr fm, BlockId blk) {
        this.fm = fm;
        this.blk = blk;
        byte[] b = new byte[fm.blockSize()];
        p = new Page(b);
        moveToBlock(blk);
    }

    public boolean hasNext() {
        return currentpos<fm.blockSize() || blk.number()>0;
    }

    public byte[] next() {
        if (currentpos == fm.blockSize()) {
            blk = new BlockId(blk.fileName(), blk.number()-1);
            moveToBlock(blk);
        }
        byte[] rec = p.getBytes(currentpos);
        currentpos += Integer.BYTES + rec.length;
        return rec;
    }
    private void moveToBlock(BlockId blk) {
        fm.read(blk, p);
        boundary = p.getInt(0);
        currentpos = boundary;
    }
}
```

4.4. Управление пользовательскими данными

Алгоритм управления журнальными записями прост и понятен. Благодаря этому диспетчер журнала может оптимально использовать память; в частности, для выполнения своей работы ему достаточно единственной страницы. Точно так же одной страницы достаточно каждому объекту `LogIterator`.

С другой стороны, в отношении данных приложения JDBC действуют совершенно непредсказуемо. Нет никакой возможности заранее узнать, какой блок приложение запросит следующим и будет ли оно когда-либо снова обращаться к запрошенному блоку. И даже когда приложение закончит работу со своими блоками, нельзя узнать, не понадобятся ли эти блоки другому приложению в ближайшем будущем. В этом разделе рассказывается, как движок базы данных может эффективно управлять памятью в такой ситуации.

4.4.1. Диспетчер буферов

Диспетчер буферов – это компонент движка базы данных, отвечающий за управление страницами с пользовательскими данными. Диспетчер буферов выделяет фиксированный набор страниц, называемый *пулом буферов*. Как упоминалось в начале этой главы, пул буферов должен умещаться в физической памяти компьютера, а его страницы должны заполняться содержимым из буферов ввода/вывода, которыми управляет операционная система.

Чтобы получить доступ к блоку, клиент взаимодействует с диспетчером буферов в соответствии со следующим протоколом:

1) клиент просит диспетчера буферов закрепить страницу из пула с требуемым блоком;
2) клиент обращается к содержимому страницы столько раз, сколько потребуется;
3) закончив работу со страницей, клиент сообщает диспетчеру буферов, что тот может открепить ее.

Говорят, что страница *закреплена*, если какой-то клиент закрепил ее к настоящему моменту; в противном случае страница считается *незакрепленной*. Диспетчер буферов обязан гарантировать доступность страницы для своих клиентов, пока она закреплена. Но как только страница открепляется, диспетчеру буферов разрешается записать в нее другой блок.

Когда клиент просит закрепить страницу с блоком, возможны четыре ситуации:

- содержимое блока уже загружено в какую-то из страниц в буфере, и:
  - страница закреплена (1);
  - страница откреплена (2);
- в данный момент содержимого блока нет ни в одной из страниц, и:
  - в пуле буферов есть по крайней мере одна незакрепленная страница (3);
  - все страницы закреплены (4).

Ситуация (1) возникает, когда в данный момент содержимое блока уже используется кем-то из клиентов. Поскольку страница может быть закреплена несколькими клиентами, диспетчер буферов просто увеличивает счетчик клиентов, закрепивших страницу, и возвращает страницу. Клиенты, закрепившие

страницу, могут параллельно читать и изменять ее содержимое. Диспетчер буферов не заботится о возможных конфликтах; эта задача возлагается на диспетчера транзакций, описываемого в главе 5.

Ситуация (2) возникает, когда клиенты, использовавшие страницу, закончили работу с ней, но в нее еще не был загружен другой блок. Поскольку содержимое блока все еще находится на странице, диспетчер буферов может повторно закрепить ее и вернуть клиенту.

Ситуация (3) требует, чтобы диспетчер буферов прочитал блок с диска на страницу. Для этого диспетчер буферов должен, во-первых, выбрать незакрепленную страницу (потому что закрепленные страницы используются клиентами). Во-вторых, если выбранная страница была изменена, диспетчер буферов должен записать ее содержимое на диск; это действие называется *выталкиванием*, или *сбросом*, страницы на диск. И только после этого можно прочитать запрошенный блок в выбранную страницу и закрепить ее.

Ситуация (4) возникает при интенсивном использовании буферов, например в алгоритмах обработки запросов, описываемых в главе 14. В этом случае диспетчер буферов оказывается не в состоянии удовлетворить запрос клиента. Лучшее, что он может сделать, – добавить клиента в список ожидания, пока не появится незакрепленная страница.

4.4.2. Буферы

Каждая страница в пуле буферов имеет информацию о своем состоянии, например закреплена ли она, и если да, то какой блок хранит в данный момент. *Буфер* – это объект, хранящий такую информацию. Каждая страница в пуле буферов связана с определенным буфером. Каждый буфер наблюдает за изменениями в своей странице и отвечает за запись измененной страницы на диск. Так же как в случае с журналом, буфер может минимизировать число обращений к диску, задерживая запись своей страницы. Например, если страница изменяется клиентом несколько раз, то эффективнее записать ее один раз, когда все изменения будут сделаны. Одна из разумных стратегий реализации такого поведения – откладывание записи страницы на диск, пока та не будет откреплена.

На самом деле буфер может откладывать запись страницы на еще более долгий срок. Представьте, что измененная страница была откреплена, но не записана на диск. Если клиент вновь закрепит страницу с тем же блоком (как во второй ситуации из описанных выше), клиент увидит измененное содержимое в том виде, в каком он его оставил. В результате эффект тот же, как если бы страница была записана на диск, а затем прочитана, но при этом не выполняется никаких обращений к диску. В некотором смысле страница в буфере действует как версия дискового блока, хранящаяся в памяти. Любому клиенту, желающему получить блок, диспетчер просто передаст страницу из буфера, которую клиент сможет исследовать или изменить без любых обращений к диску.

Фактически немедленная запись измененной страницы на диск необходима в двух случаях: если ее содержимое требуется заменить содержимым другого дискового блока (как в ситуации (3) из описанных выше) или если диспетчеру восстановления (будет обсуждаться в главе 5) понадобилось записать страницу для защиты от возможного сбоя системы.

4.4.3. Стратегии замещения буферов

Изначально пул буферов пуст. По мере поступления запросов на закрепление страниц диспетчер буферов заполняет пул, выделяя страницы и загружая в них запрошенные блоки. Как только все страницы будут распределены, диспетчер буферов начнет вытеснять их, замещая другими. Для вытеснения диспетчер буферов может выбрать любую незакрепленную страницу.

Если по запросу клиента диспетчеру буферов понадобится вытеснить страницу и все страницы окажутся закреплены, то клиенту придется подождать. Чтобы такое случалось как можно реже, все клиенты обязаны «соблюдать нормы поведения в обществе» и открeплять страницы, как только они станут не нужны.

При наличии нескольких открепленных страниц диспетчер буферов должен решить, какую из них вытеснить. Этот выбор может оказать существенное влияние на эффективность системы баз данных. Например, наихудшим вариантом будет замена страницы, которая еще понадобится в ближайшем будущем, потому что диспетчеру буферов придется вновь загрузить блок с диска и заменить другую страницу. Более разумным выглядит вариант выбора страницы, которая не будет использоваться дольше остальных.

Поскольку диспетчер буферов не может предсказать, какие страницы понадобятся в следующий раз, он вынужден угадывать. Диспетчер буферов оказывается практически в той же ситуации, что и операционная система, когда та сбрасывает страницы виртуальной памяти на диск. Но есть одно большое отличие: диспетчер буферов *знает*, используется ли страница в данный момент или нет, потому что используемые страницы всегда закреплены. Ограничение, препятствующее вытеснению закрепленных страниц, играет только на руку. Клиенты, закрепляя страницы, удерживают диспетчера буферов от неправильных предположений. На выбор остаются только ненужные в данный момент страницы, что гораздо менее критично.

Но даже ограниченный набором незакрепленных страниц, диспетчер буферов должен решить, какая из них не понадобится в течение самого долгого времени. Например, типичная база данных имеет несколько страниц (например, файлы каталогов, о которых рассказывается в главе 7), которые постоянно используются на протяжении всего времени ее работы. Диспетчер буферов должен избегать вытеснения таких страниц, так как они почти наверняка будут повторно закреплены в ближайшем будущем. Есть несколько стратегий замещения, которые пытаются угадать наиболее подходящую страницу. В этом разделе рассматриваются четыре из них: *наивная*, *FIFO*, *LRU* и *круговая*.

На рис. 4.2 показан пример, который поможет нам сравнить поведение этих алгоритмов замещения. В верхней части (a) показана последовательность операций, которые закрепляют и открепляют пять блоков, а в нижней части (b) показано конечное состояние пула буферов, если исходить из предположения, что он содержит четыре буфера. Единственное вытеснение страницы произошло при выполнении пятой операции закрепления (закрепившей блок 50). Однако, поскольку в тот момент был открeплен только один буфер, у диспетчера буферов не было выбора. То есть пул буферов будет выглядеть как на рис. 4.2b, независимо от используемой стратегии вытеснения страницы.

```
pin(10); pin(20); pin(30); pin(40); unpin(20);
pin(50); unpin(40); unpin(10); unpin(30); unpin(50);
```

(a)

Буфер:	0	1	2	3
номер блока	10	50	30	40
время чтения с диска	1	6	3	4
время открепления	8	10	9	7

(b)

Рис. 4.2. Влияние операций pin/unpin на пул из четырех буферов: (a) последовательность из десяти операций pin/unpin; (b) конечное состояние пула буферов

Каждый буфер на рис. 4.2b имеет три атрибута: номер блока, время чтения с диска и время открепления. Времена соответствуют позициям операций на рис. 4.2a.

Буферы, изображенные на рис. 4.2b, не закреплены. Предположим теперь, что диспетчер буферов получает еще два запроса на закрепление:

```
pin(60); pin(70);
```

Ему нужно заместить два буфера. Все буферы доступны для замещения; так какие из них выбрать? Каждый из следующих алгоритмов даст свой ответ на этот вопрос.

Наивная стратегия

Самая простая стратегия заключается в последовательном обходе пула буферов и замещении первого найденного незакрепленного буфера. Для ситуации, изображенной на рис. 4.9, блок 60 будет загружен в буфер 0, а блок 70 – в буфер 1.

Эта стратегия проста в реализации, и это единственное, за что ее можно было бы рекомендовать. Например, вернемся к пулу буферов на рис. 4.2 и предположим, что клиент многократно закрепляет и открепляет блоки 60 и 70:

```
pin(60); unpin(60); pin(70); unpin(70); pin(60); unpin(60); ...
```

Всякий раз наивная стратегия замещения будет выбирать буфер 0, а значит, блоки будут читаться с диска при каждом закреплении. В результате пул буферов будет использоваться неравномерно. Если бы стратегия замещения выбирала разные буферы для блоков 60 и 70, тогда они читались бы с диска по одному разу, что дало бы существенный прирост эффективности.

Стратегия FIFO

Наивная стратегия выбирает буфер, учитывая только свое удобство. Стратегия FIFO[1] старается действовать более разумно, выбирая буфер, который был замещен раньше всех, то есть страницу, которая находится в пуле буферов дольше всех. Эта стратегия обычно эффективнее, чем наивная, потому что более старые страницы с меньшей вероятностью понадобятся в ближайшем будущем,

[1] First In First Out – первым пришел, первым ушел. – *Прим. перев.*

чем недавно загруженные. На рис. 4.2 самыми старыми являются страницы с наименьшими значениями «время чтения». Поэтому блок 60 будет загружен в буфер 0, а блок 70 – в буфер 2.

FIFO – разумная стратегия, но она не всегда делает правильный выбор. Например, базы данных обычно имеют часто используемые страницы, такие как страницы каталога (см. главу 7). Поскольку эти страницы используются почти каждым клиентом, есть смысл по возможности не вытеснять их. Однако в какой-то момент эти страницы неизбежно станут самыми старыми в пуле, и стратегия FIFO выберет их для замещения.

Стратегию FIFO можно реализовать двумя способами. Первый – хранить в каждом буфере время последней замены страницы, как показано на рис. 4.2b. В этом случае алгоритм замещения будет сканировать пул буферов и выбирать незакрепленную страницу с самым ранним временем вытеснения. Второй способ, более эффективный для диспетчера буферов, – хранить список указателей на буферы, упорядоченный по времени замещения. В этом случае алгоритму замещения будет достаточно найти в списке первую незакрепленную страницу, а указатель на нее переместить в конец списка.

Стратегия LRU

Стратегия FIFO принимает решение о замещении, опираясь на время *добавления* страницы в пул. Похожая на нее стратегия принимает решение, опираясь на время последнего *обращения* к странице, исходя из предположения, что давно не использовавшаяся страница едва ли понадобится в ближайшем будущем. Эта стратегия называется LRU, что означает *least recently used* (наиболее давно не использовавшийся). В примере на рис. 4.2 времени последнего использования буфера соответствует атрибут «время открепления». Соответственно, для блока 60 эта стратегия выберет буфер 3, а для блока 70 – буфер 0.

Стратегия LRU считается эффективной стратегией общего назначения и уменьшает вероятность вытеснения часто используемых страниц. Оба варианта реализации FIFO можно преобразовать в стратегию LRU. Единственное изменение, которое необходимо для этого сделать, – обновлять отметку времени (в первом варианте) или список (во втором) при каждом откреплении страницы, а не при ее вытеснении.

Круговая (clock) стратегия

Эта стратегия является интересной комбинацией двух предыдущих стратегий и имеет простую и понятную реализацию. Так же как в наивной стратегии, алгоритм круговой замены сканирует пул буферов, выбирая первую найденную незакрепленную страницу. Разница в том, что сканирование всегда начинается со страницы, следующей за последней вытесненной. Если представить пул буферов как кольцо, тогда алгоритм замены пробегает пул, подобно стрелке часов, останавливается, найдя страницу для вытеснения, и запускается вновь, когда требуется выполнить следующее замещение.

Пример на рис. 4.2b не указывает положение стрелки часов. Но последнему замещению подвергся буфер 1, а значит, стрелка будет установлена на следующий за ним буфер. Соответственно, блок 60 будет загружен в буфер 2, а блок 70 – в буфер 3.

Круговая стратегия пытается использовать буферы как можно более равномерно. Если страница закреплена, стратегия пропустит ее и не будет рассматривать, пока не проверит все другие буферы. Эта особенность придает стратегии вид LRU. Идея заключается в том, что если страница часто используется, высока вероятность, что она будет закреплена, когда придет ее очередь проверки возможности замещения. В этом случае алгоритм пропустит ее и даст ей «еще один шанс».

4.5. Диспетчер буферов в SimpleDB

В этом разделе рассматривается диспетчер буферов системы баз данных SimpleDB. В разделе 4.5.1 описывается API диспетчера и даются примеры его использования. А в разделе 4.5.2 демонстрируется реализация его классов на Java.

4.5.1. API диспетчера буферов

Реализация диспетчера буферов для SimpleDB находится в пакете `simpledb.buffer`. Этот пакет содержит два класса: `BufferMgr` и `Buffer`. Их API показан в листинге 4.5.

Каждый экземпляр системы баз данных имеет один объект `BufferMgr`, который создается во время запуска. Конструктор класса принимает три аргумента: размер пула буферов и ссылки на диспетчеров файлов и журнала.

Объект `BufferMgr` имеет методы для закрепления и открепления страниц. Метод `pin` возвращает объект `Buffer` с закрепленной страницей, содержащей указанный блок, а метод `unpin` открепляет страницу. Метод `available` возвращает количество доступных незакрепленных страниц в пуле буферов. А метод `flushAll` гарантирует запись на диск всех страниц, измененных указанной транзакцией.

Получив объект `Buffer`, клиент может вызвать его метод `contents`, чтобы извлечь связанную с ним страницу. Если клиент изменит страницу, он также обязан создать соответствующие записи в журнале и вызвать метод `setModified` буфера.

Листинг 4.5. API диспетчера буферов в SimpleDB

BufferMgr

```
public BufferMgr(FileMgr fm, LogMgr lm, int numbuffs);
public Buffer pin(BlockId blk);
public void   unpin(Buffer buff);
public int    available();
public void   flushAll(int txnum);
```

Buffer

```
public Buffer(FileMgr fm, LogMgr lm);
public Page    contents();
public BlockId block();
public boolean isPinned();
public void    setModified(int txnum, int lsn);
public int     modifyingTx();
```

Метод `setModified` принимает два аргумента: целое число – идентификатор транзакции, выполнившей изменения, и номер LSN сгенерированной записи в журнале.

Код в листинге 4.6 тестирует класс `Buffer`. Он выводит сообщение «The new value is 1» («Новое значение равно 1») при первом выполнении и увеличивает выводимое значение при каждом последующем выполнении. Код действует следующим образом: создает объект `SimpleDB` с тремя буферами; связывает страницу с блоком 1 и закрепляет ее; увеличивает целое число в смещении 80 и вызывает `setModified`, чтобы сообщить, что страница изменена. Методу `setModified` передаются номер транзакции и номер LSN сгенерированной записи в файле журнала. Подробнее эти два значения будут обсуждаться в главе 5, а пока отмечу, что в этих аргументах просто передаются подходящие величины.

Диспетчер буферов скрывает от клиентов фактические обращения к диску. Клиент не знает, сколько происходит обращений к диску от его имени и когда они происходят. Чтение с диска может происходить только в вызове метода `pin`, в частности когда затребованный блок в данный момент отсутствует в буфере. Запись на диск может происходить только во время вызова `pin` или `flushAll`. Вызов `pin` выполняет запись на диск, если вытесняемая страница была изменена, а вызов `flushAll` запишет на диск все страницы, измененные указанной транзакцией.

Например, код в листинге 4.6 дважды изменяет блок 1. Но ни одно из этих изменений не записывается на диск явно. Как сообщают комментарии в коде, первое изменение записывается на диск, а второе – нет. Рассмотрим первое изменение. Поскольку пул содержит всего три буфера, диспетчеру буферов потребуется вытеснить страницу с блоком 1 (и, следовательно, записать ее на диск), чтобы закрепить страницы для блоков 2, 3 и 4. С другой стороны, после второго изменения страница с содержимым блока 1 не нуждается в вытеснении, поэтому она не записывается на диск и изменения, произведенные в ней, теряются. Вопрос об утерянных изменениях будет обсуждаться в главе 5.

Листинг 4.6. Тестирование класса Buffer

```
public class BufferTest {
    public static void main(String[] args) {
        SimpleDB db = new SimpleDB("buffertest", 400, 3);
        BufferMgr bm = db.bufferMgr();

        Buffer buff1 = bm.pin(new BlockId("testfile", 1));
        Page p = buff1.contents();
        int n = p.getInt(80);
        p.setInt(80, n+1);        // Это изменение
        buff1.setModified(1, 0);  // будет записано на диск.
        System.out.println("The new value is " + (n+1));
        bm.unpin(buff1);
        // Один из следующих вызовов pin сбросит содержимое buff1 на диск:
        Buffer buff2 = bm.pin(new BlockId("testfile", 2));
        Buffer buff3 = bm.pin(new BlockId("testfile", 3));
        Buffer buff4 = bm.pin(new BlockId("testfile", 4));

        bm.unpin(buff2);
        buff2 = bm.pin(new BlockId("testfile", 1));
        Page p2 = buff2.contents();
        p2.setInt(80, 9999);        // Это изменение
        buff2.setModified(1, 0);    // не будет записано на диск.
        bm.unpin(buff2);
    }
}
```

Предположим, что движок базы данных обслуживает множество клиентов, каждый из которых использует несколько буферов. Может оказаться, что все страницы в буферах будут закреплены. В этом случае диспетчер буферов не сможет немедленно удовлетворить запрос `pin` и должен поместить его в список ожидания. Когда появится незакрепленный буфер, диспетчер буферов извлечет запрос из списка ожидания и удовлетворит его. Проще говоря, клиент не будет знать о конфликте; он может лишь заметить, что движок замедлился.

Возможна одна ситуация, когда конкуренция за обладание буферами может вызвать серьезную проблему. Представьте сценарий, когда двум клиентам, A и B, потребовалось закрепить по два буфера, но в текущий момент доступны только два буфера. Допустим, что клиент A закрепил первый буфер и вступил в гонку за обладание вторым буфером. Если клиент A получит его раньше клиента B, тогда диспетчер добавит запрос клиента B в список ожидания. В какой-то момент клиент A завершит работу и открепит буферы, после чего клиент B сможет их закрепить. Это удачное разрешение ситуации. Теперь предположим, что клиенту B удалось закрепить второй буфер раньше клиента A. Тогда оба клиента, A и B, окажутся в списке ожидания. Если это единственные клиенты в системе, закрепленные ими буферы никогда не будут откреплены, и оба они застрянут в списке ожидания навсегда. Это плохой сценарий. В этом случае говорят, что клиенты A и B *зашли в тупик*.

В настоящей системе баз данных с тысячами буферов и сотнями клиентов такая тупиковая ситуация крайне маловероятна. Тем не менее диспетчер буферов должен быть готов к встрече с ней. В SimpleDB, например, реализовано такое решение: движок следит, как долго клиент ждет освобождения буфера. Если ожидание длится слишком долго (скажем, 10 с), тогда диспетчер буфера делает предположение, что клиент находится в тупике, и возбуждает исключение `BufferAbortException`. Ответственность за его обработку лежит на клиенте, который может откатить транзакцию или перезапустить ее.

Код в листинге 4.7 тестирует действия диспетчера буферов в этом случае. Он снова создает объект `SimpleDB` с тремя буферами, а затем связывает страницы в буферах с блоками 0, 1 и 2 в файле «testfile» и закрепляет их. После этого он открепляет блок 1, повторно закрепляет блок 0 и снова закрепляет блок 1. Последние три действия не вызовут обращения к диску и не оставят доступных буферов. При попытке закрепить блок 3 диспетчер поместит клиента (поток выполнения) в список ожидания. Однако, поскольку все буферы закреплены этим потоком, ни один из них не будет открепен, и через 10 с диспетчер буферов сгенерирует исключение. Программа перехватит исключение и продолжит выполнение. Она открепит блок 2, после чего попытка закрепить блок 3 увенчается успехом, потому что появился незакрепленный буфер.

Листинг 4.7. Тестирование диспетчера буферов

```
public class BufferMgrTest {
    public static void main(String[] args) throws Exception {
        SimpleDB db = new SimpleDB("buffermgrtest", 400, 3);
        BufferMgr bm = db.bufferMgr();

        Buffer[] buff = new Buffer[6];
        buff[0] = bm.pin(new BlockId("testfile", 0));
        buff[1] = bm.pin(new BlockId("testfile", 1));
        buff[2] = bm.pin(new BlockId("testfile", 2));
        bm.unpin(buff[1]); buff[1] = null;
        buff[3] = bm.pin(new BlockId("testfile", 0));
        buff[4] = bm.pin(new BlockId("testfile", 1));
        System.out.println("Available buffers: " + bm.available());
        try {
            System.out.println("Attempting to pin block 3...");
            buff[5] = bm.pin(new BlockId("testfile", 3));
        }
        catch(BufferAbortException e) {
            System.out.println("Exception: No available buffers\n");
        }
        bm.unpin(buff[2]); buff[2] = null;
        buff[5] = bm.pin(new BlockId("testfile", 3)); // теперь все получится
        System.out.println("Final Buffer Allocation:");
        for (int i=0; i<buff.length; i++) {
            Buffer b = buff[i];
            if (b != null)
                System.out.println("buff["+i+"] pinned to block "
                                   + b.block());
        }
    }
}
```

4.5.2. Реализация диспетчера буферов

В листинге 4.8 приводится код с реализацией класса `Buffer`. Объект `Buffer` хранит следующие сведения о своей странице:

- *ссылку на блок, связанный со страницей*; если в страницу не загружен никакой блок, это значение равно `null`;
- *сколько раз страница была закреплена*; этот счетчик увеличивается с каждым вызовом `pin` и уменьшается с каждым вызовом `unpin`;
- *целое число – признак изменения страницы*; значение –1 указывает, что страница не была изменена; в любом другом случае целое число определяет идентификатор транзакции, которая внесла изменение;
- *информация о записи в журнале*; если страница была изменена, в буфере хранится LSN самой последней записи в журнале. Значения LSN никогда не бывают отрицательными. Вызвав `setModified` с отрицательным номером LSN, клиент сообщает, что для этого изменения не была создана запись в журнале.

Листинг 4.8. Реализация класса Buffer в SimpleDB

```
public class Buffer {
    private FileMgr fm;
    private LogMgr lm;
    private Page contents;
    private BlockId blk = null;
    private int pins = 0;
    private int txnum = -1;
    private int lsn = -1;

    public Buffer(FileMgr fm, LogMgr lm) {
        this.fm = fm;
        this.lm = lm;
        contents = new Page(fm.blockSize());
    }

    public Page contents() {
        return contents;
    }

    public BlockId block() {
        return blk;
    }

    public void setModified(int txnum, int lsn) {
        this.txnum = txnum;
        if (lsn>=0) this.lsn = lsn;
    }

    public boolean isPinned() {
        return pins > 0;
    }

    public int modifyingTx() {
        return txnum;
    }

    void assignToBlock(BlockId b) {
        flush();
        blk = b;
        fm.read(blk, contents);
        pins = 0;
    }

    void flush() {
        if (txnum >= 0) {
            lm.flush(lsn);
            fm.write(blk, contents);
            txnum = -1;
        }
    }

    void pin() {
        pins++;
    }

    void unpin() {
        pins--;
    }
}
```

Метод flush гарантирует запись измененного содержимого страницы в буфере в соответствующий блок на диске. Если страница не была изменена, метод ничего не делает. Если она изменена, метод flush сначала вызовет метод LogMgr.flush, чтобы гарантировать сохранение на диске соответствующей журнальной записи, а затем запишет страницу на диск.

Метод assignToBlock связывает буфер с дисковым блоком. При этом буфер сначала сбрасывается на диск, благодаря чему сохраняются все имевшие место изменения, а затем в страницу буфера читается содержимое указанного дискового блока.

В листинге 4.9 приводится код с реализацией класса BufferMgr. Метод pin связывает буфер с указанным блоком. Для этого вызывается приватный метод tryToPin, состоящий из двух частей. В первой части tryToPin вызывается метод findExistingBuffer, чтобы найти буфер, который уже связан с указанным блоком. Если такой буфер имеется, tryToPin возвращает его. Иначе выполняется вторая часть, которая вызывает метод ChooseUnpinnedBuffer, реализующий наивный алгоритм выбора незакрепленного буфера. Для полученного буфера вызывается метод assignToBlock, который записывает существующую страницу на диск (при необходимости) и читает новый блок с диска. Если найти незакрепленный буфер не удалось, ChooseUnpinnedBuffer возвращает null.

Если tryToPin вернет null, метод pin вызовет Java-метод wait. В Java каждый объект имеет список ожидания. Метод wait объекта прерывает выполнение вызывающего потока и помещает его в этот список. Согласно коду в листинге 4.9, поток будет оставаться в этом списке, пока не выполнится одно из двух условий:

- другой поток вызовет notifyAll (что происходит при вызове метода unpin);
- истечет интервал времени, равный MAX_TIME миллисекунд, что означает слишком долгое ожидание.

Когда ожидающий поток возобновит выполнение, он продолжает цикл, пытаясь получить буфер (как и все другие ожидающие потоки). Поток будет снова и снова помещаться в список ожидания, пока он не получит буфер или не превысит свой лимит времени.

Метод unpin открепляет указанный буфер и проверяет, остался ли буфер закрепленным. Если нет, то вызывает notifyAll, чтобы уведомить о появлении свободного буфера все ожидающие клиентские потоки. Эти потоки будут состязаться за обладание буфером; выиграет первый из них, который возобновит работу. Когда другие потоки получат шанс выполниться, они могут обнаружить, что все буферы снова заняты, и вернутся в список ожидания.

Листинг 4.9. Реализация класса BufferMgr в SimpleDB

```
public class BufferMgr {
    private Buffer[] bufferpool;
    private int numAvailable;
    private static final long MAX_TIME = 10000; // 10 секунд
    public BufferMgr(FileMgr fm, LogMgr lm, int numbuffs) {
        bufferpool = new Buffer[numbuffs];
        numAvailable = numbuffs;
        for (int i=0; i<numbuffs; i++)
            bufferpool[i] = new Buffer(fm, lm);
    }
    public synchronized int available() {
        return numAvailable;
    }
```

```
    public synchronized void flushAll(int txnum) {
        for (Buffer buff : bufferpool)
            if (buff.modifyingTx() == txnum)
                buff.flush();
    }

    public synchronized void unpin(Buffer buff) {
        buff.unpin();
        if (!buff.isPinned()) {
            numAvailable++;
            notifyAll();
        }
    }

    public synchronized Buffer pin(BlockId blk) {
        try {
            long timestamp = System.currentTimeMillis();
            Buffer buff = tryToPin(blk);
            while (buff == null && !waitingTooLong(timestamp)) {
                wait(MAX_TIME);
                buff = tryToPin(blk);
            }
            if (buff == null)
                throw new BufferAbortException();
            return buff;
        }
        catch(InterruptedException e) {
            throw new BufferAbortException();
        }
    }

    private boolean waitingTooLong(long starttime) {
        return System.currentTimeMillis() - starttime > MAX_TIME;
    }

    private Buffer tryToPin(BlockId blk) {
        Buffer buff = findExistingBuffer(blk);
        if (buff == null) {
            buff = chooseUnpinnedBuffer();
            if (buff == null)
                return null;
            buff.assignToBlock(blk);
        }
        if (!buff.isPinned())
            numAvailable--;
        buff.pin();
        return buff;
    }

    private Buffer findExistingBuffer(BlockId blk) {
        for (Buffer buff : bufferpool) {
            BlockId b = buff.block();
            if (b != null && b.equals(blk))
                return buff;
        }
        return null;
    }

    private Buffer chooseUnpinnedBuffer() {
        for (Buffer buff : bufferpool)
            if (!buff.isPinned())
                return buff;
        return null;
    }
}
```

4.6. Итоги

- Движок базы данных должен стремиться минимизировать число обращений к диску. Поэтому он тщательно управляет страницами в памяти, в которых хранит дисковые блоки. Управляют этими страницами *диспетчер журнала* и *диспетчер буферов*.
- Диспетчер журнала отвечает за сохранение записей в файл журнала. Поскольку записи всегда добавляются в конец файла журнала и никогда не изменяются, диспетчер журнала имеет возможность действовать очень эффективно. Ему достаточно выделить в памяти одну страницу и использовать простой алгоритм для сохранения этой страницы на диск как можно реже.
- Диспетчер буферов выделяет в памяти несколько страниц, называемых *пулом буферов*, для обработки пользовательских данных. Диспетчер буферов *закрепляет* и *открепляет* страницы по запросам клиентов, при необходимости читая соответствующие дисковые блоки. Клиент получает доступ к странице в буфере после ее закрепления, и открепляет страницу, завершив работу с ней.
- Измененный буфер записывается на диск в двух случаях: когда страница вытесняется содержимым другого блока и когда это необходимо диспетчеру восстановления.
- Когда клиент требует связать страницу с дисковым блоком, диспетчер буферов выбирает подходящий буфер. Если страница с этим блоком уже находится в буфере, он возвращается клиенту; в противном случае диспетчер буферов замещает содержимое существующего буфера.
- Алгоритм, выбирающий буфер для вытеснения, использует *стратегию замещения буфера*. Вот четыре интересные стратегии замещения:
  - *наивная*: выбирается первый найденный незакрепленный буфер;
  - *FIFO*: выбирается незакрепленный буфер, замещение которого произошло раньше других;
  - *LRU*: выбирается незакрепленный буфер, который был откреплен раньше других;
  - *круговая*: буферы сканируются последовательно, начиная от того, что был замещен последним, и выбирается первый найденный незакрепленный буфер.

4.7. Для дополнительного чтения

В статье Effelsberg et al. (1984) вы найдете подробное и исчерпывающее описание подхода к управлению буферами, который развивает многие идеи, представленные в этой главе. В главе 13 Gray and Reuter (1993) вы найдете углубленное обсуждение механизма управления буферами, иллюстрированное реализацией типичного диспетчера буферов на языке C.

В Oracle по умолчанию используется стратегия LRU. Однако при сканировании больших таблиц используется стратегия FIFO. Обосновано это тем, что при сканировании таблицы, как правило, блок становится ненужным после открепления, из-за чего применение стратегии LRU приводит к со-

хранению ненужных блоков. Подробности можно найти в главе 14 Ashdown et al. (2019).

Несколько исследователей занимались изучением вопроса эффективной реализации диспетчера буферов и пришли к следующей идее: отслеживать в диспетчере буферов запросы на закрепление блоков в каждой транзакции. Если обнаружится определенный шаблон (например, транзакция снова и снова читает одни и те же N блоков из файла), диспетчер может попытаться избежать вытеснения соответствующих страниц, даже если они не закреплены. Эта идея подробно описывается в статье Ng et al. (1991), где также предоставлены некоторые результаты моделирования.

Ashdown, L., et al. (2019). «Oracle database concepts». Document E96138-01, Oracle Corporation. Книга доступна по адресу: https://docs.oracle.com/en/database/oracle/oracle-database/19/cncpt/database-concepts.pdf.

Effelsberg, W., & Haerder, T. (1984). «Principles of database buffer management». *ACM Transactions on Database Systems*, 9 (4), 560–595.

Gray, J., & Reuter, A. (1993). «Transaction processing: concepts and techniques». Morgan Kaufman.

Ng, R., Faloutsos, C., & Sellis, T. (1991). «Flexible buffer allocation based on marginal gains». *Proceedings of the ACM SIGMOD Conference*, p. 387–396.

4.8. Упражнения

Теория

4.1. Метод `LogMgr.iterator` вызывает `flush`. Нужен ли этот вызов? Объясните, почему.

4.2. Объясните, почему метод `BufferMgr.pin` объявлен синхронным. Какая проблема может возникнуть, если опустить это объявление?

4.3. Можно ли связать несколько буферов с одним и тем же блоком? Объясните, почему.

4.4. Стратегии замещения буферов, представленные в этой главе, не различают измененные и неизмененные страницы при поиске доступного буфера. Возможно, будет эффективнее, если диспетчер буферов будет пытаться вытеснять неизмененные страницы.

a) Укажите хотя бы одну причину, почему это предложение может уменьшить количество обращений к диску, выполняемых диспетчером буферов.

b) Укажите хотя бы одну причину, почему это предложение может увеличить количество обращений к диску, выполняемых диспетчером буферов.

c) Как вы думаете, имеет ли смысл реализовать эту стратегию? Почему?

4.5. Еще одна возможная стратегия замещения буферов: выбор *измененного наиболее давно* (Least Recently Modified, LRM). Согласно этой стратегии диспетчер буферов должен выбрать измененный буфер с наименьшим номером LSN. Объясните, почему такая стратегия может оказаться эффективной.

4.6. Предположим, что страница в буфере изменялась несколько раз без записи на диск. Буфер хранит только номер LSN самого последнего изменения и передает его диспетчеру журнала, когда страница наконец записывается на диск. Объясните, почему буферу не требуется передавать другие номера LSN диспетчеру журнала.

4.7. В сценарий, изображенный на рис. 4.2a, добавьте две дополнительные операции: `pin(60)`; `pin(70)`. Для каждой из четырех стратегий замещения, приведенных в тексте, нарисуйте состояние буферов, исходя из предположения, что пул содержит пять буферов.

4.8. Начав с состояния буферов, как показано на рис. 4.2b, опишите сценарий, когда:

a) стратегия FIFO обеспечит наименьшее число обращений к диску;
b) стратегия LRU обеспечит наименьшее число обращений к диску;
c) круговая стратегия обеспечит наименьшее число обращений к диску.

4.9. Предположим, что два разных клиента хотят закрепить один и тот же блок, но были помещены в список ожидания из-за отсутствия доступных буферов. Покажите, что реализация класса `BufferMgr` в SimpleDB позволяет обоим клиентам использовать один и тот же буфер, как только он освободится.

4.10. Рассмотрим пословицу «Virtual is its own reward» (Виртуальность – сама себе награда)[1]. Прокомментируйте остроумность этого каламбура и обсудите его применимость к диспетчеру буферов.

Практика

4.11. Диспетчер журнала в SimpleDB выделяет в памяти свою страницу и записывает ее на диск явно. Однако точно так же можно было бы связать буфер с последним блоком журнала и передать управление этим буфером диспетчеру буферов.

a) Подумайте, как можно реализовать такой вариант. Какие проблемы могут возникнуть? Насколько хороша эта идея?
b) Добавьте реализацию данного варианта в SimpleDB.

4.12. Каждый объект `LogIterator` выделяет в памяти страницу для хранения блоков журнала, к которым он обращается.

a) Объясните, почему использование буфера вместо страницы было бы намного эффективнее.
b) Измените код так, чтобы он использовал буфер вместо страницы. Как такой буфер должен открепляться?

4.13. В этом упражнении вы должны проверить, сможет ли программа JDBC злонамеренно закрепить все буферы в пуле.

a) Напишите программу JDBC, закрепляющую все буферы в пуле. Что случится, когда все буферы будут закреплены?

---

[1] Здесь автор проводит параллель с пословицей «Virtue is its own reward» (Добродетель – сама себе награда). – *Прим. перев.*

b) Система баз данных Derby управляет буферами иначе, чем SimpleDB. Когда JDBC-клиент запрашивает буфер, Derby закрепляет буфер, передает клиенту копию буфера и открепляет свой буфер. Объясните, почему ваша программа из пункта «a» не сможет нанести вред другим клиентам Derby.

c) Derby предотвращает проблемы, свойственные SimpleDB, всегда передавая клиентам копии страниц. Объясните последствия такого подхода. Можно ли сказать, что подход, реализованный в SimpleDB, лучше?

d) Еще один способ предотвратить монополизацию всех буферов злонамеренным клиентом – разрешить каждой транзакции закреплять не более определенного процента (скажем, 10 %) буферов. Реализуйте и протестируйте эту идею в диспетчере буферов SimpleDB.

4.14. Измените класс `BufferMgr` и реализуйте все другие стратегии замещения, описанные в этой главе.

4.15. В упражнении 4.4 предложена стратегия замещения, которая старается выбирать неизмененные страницы. Реализуйте эту стратегию.

4.16. В упражнении 4.5 предложена стратегия замещения, которая выбирает измененную страницу с наименьшим номером LSN. Реализуйте эту стратегию.

4.17. Диспетчер буферов в SimpleDB последовательно просматривает пул, чтобы найти подходящий буфер. Этот поиск будет занимать много времени, когда в пуле будут храниться тысячи буферов. Измените код и используйте структуры данных (такие как списки и хеш-таблицы), способные уменьшить время поиска.

4.18. В упражнении 3.15 было предложено написать код, поддерживающий статистику использования диска. Расширьте этот код, чтобы он также предоставлял информацию об использовании буферов.

Глава 5

Управление транзакциями

Диспетчер буферов позволяет нескольким клиентам одновременно обращаться к одному и тому же буферу, произвольно читая и изменяя значения в нем. Отсутствие контроля может привести к хаосу: каждый раз, просматривая страницу, клиент может видеть разные (и даже несогласованные) значения, что делает невозможным получение точного представления базы данных. Или два клиента могут невольно затирать изменения друг друга, тем самым повреждая базу данных.

Для предотвращения подобных неблагоприятных сценариев в движке базы данных имеются *диспетчер конкуренции* и *диспетчер восстановления*, задача которых заключается в поддержании порядка и обеспечении целостности базы данных. Все операции с базой данных в клиентских программах выполняются как последовательность *транзакций*. Диспетчер конкуренции управляет выполнением этих транзакций так, чтобы обеспечить согласованность. Диспетчер восстановления читает и сохраняет записи в журнал, благодаря чему имеется возможность при необходимости отменять изменения, внесенные незафиксированными (неподтвержденными) транзакциями. В этой главе рассматриваются функциональные возможности этих диспетчеров и приемы, используемые для их реализации.

5.1. Транзакции

Рассмотрим базу данных в системе бронирования авиабилетов, в которой имеются две таблицы со следующими полями:

```
SEATS(FlightId, NumAvailable, Price)
CUST(CustId, BalanceDue)
```

В листинге 5.1 показан JDBC-код, реализующий покупку билета на указанный рейс для указанного клиента. Этот код не содержит ошибок, но если на сервере произойдет сбой или с базой данных одновременно будет работать несколько клиентов, могут возникнуть различные проблемы. Следующие три сценария иллюстрируют эти проблемы.

Листинг 5.1. JDBC-код, бронирующий билет на авиарейс

```
public void reserveSeat(Connection conn, int custId,
                        int flightId) throws SQLException {
    Statement stmt = conn.createStatement();
    String s;
    // Шаг 1: получить число свободных мест и стоимость
    s = "select NumAvailable, Price from SEATS " +
        "where FlightId = " + flightId;
    ResultSet rs = stmt.executeQuery(s);
    if (!rs.next()) {
        System.out.println("Flight doesn't exist");
        return;
    }
    int numAvailable = rs.getInt("NumAvailable");
    int price = rs.getInt("Price");
    rs.close();

    if (numAvailable == 0) {
        System.out.println("Flight is full");
        return;
    }

    // Шаг 2: уменьшить число свободных мест
    int newNumAvailable = numAvailable - 1;
    s = "update SEATS set NumAvailable = " + newNumAvailable +
        " where FlightId = " + flightId;
    stmt.executeUpdate(s);

    // Шаг 3: изменить баланс клиента
    s = "select BalanceDue from CUST where CustID = " + custId;
    rs = stmt.executeQuery(s);
    int newBalance = rs.getInt("BalanceDue") + price;
    rs.close();

    s = "update CUST set BalanceDue = " + newBalance +
        " where CustId = " + custId;
    stmt.executeUpdate(s);
}
```

В первом сценарии предполагается, что оба клиента, A и B, одновременно выполняют JDBC-код в следующей последовательности:

- клиент A целиком выполняет шаг 1 и затем приостанавливается;
- клиент B выполняет все шаги полностью;
- клиент A выполняет остальные шаги.

В этом случае оба потока получат одно и то же значение `numAvailable`. В результате будет продано два билета, но количество доступных мест уменьшится только один раз.

Во втором сценарии предполагается, что на сервере происходит сбой сразу после того, как клиент выполнит второй шаг. В этом случае место будет забронировано, но клиент не заплатит за него.

В третьем сценарии предполагается, что клиент выполнил все шаги полностью, но диспетчер буферов отложил запись измененных страниц на диск. Если

в этот момент на сервере произойдет сбой (пусть даже через несколько дней), то невозможно будет узнать, какие страницы были (если были) в конечном итоге записаны на диск. Если первое изменение было записано, а второе – нет, тогда клиент получит бесплатный билет; если было записано только второе изменение, тогда клиент заплатит, не получив билета. А если ни одна страница не была записана, тогда будут потеряны все действия клиента.

Приведенные выше сценарии показывают, как можно потерять или повредить данные, когда клиентские программы работают без контроля. Движки баз данных решают эту проблему, заставляя клиентские программы выполнять *транзакции*. Транзакция – это группа операций, которая действует подобно одной операции. Под словами «подобно одной операции» в данном случае понимаются следующие ACID-свойства: *атомарность* (atomicity), *согласованность* (consistency), *изоляция* (isolation) и *долговечность* (durability):

- атомарность означает, что транзакция выполняется по принципу «все или ничего». То есть либо все ее операции завершаются успешно (транзакция *фиксируется*), либо все терпят неудачу (транзакция *откатывается*);
- согласованность означает, что по завершении любой транзакции база данных остается в согласованном состоянии. То есть каждая транзакция представляет законченную единицу работы, которая может выполняться независимо от других транзакций;
- изоляция означает, что транзакция ведет себя так, будто она является единственным потоком выполнения, использующим движок. Если одновременно выполнить несколько транзакций, их результат получится таким же, как если бы они выполнялись последовательно в некотором порядке;
- долговечность означает, что изменения, внесенные зафиксированной транзакцией, гарантированно будут сохранены.

Все сценарии, приведенные выше, являются результатом нарушения тех или иных ACID-свойств. В первом сценарии нарушено свойство изоляции, потому что оба клиента получают одно и то же значение `numAvailable`, тогда как при последовательном выполнении второй клиент получит значение, записанное первым клиентом. Во втором сценарии нарушено свойство атомарности, а в третьем – долговечности.

Свойства атомарности и долговечности описывают правильное поведение операций фиксации и отката. Операция фиксации транзакции должна сохранить измененные данные, а операция отката (которая может быть выполнена клиентом явно или в результате сбоя системы) должна полностью отменить изменения. Эти функции находятся в ведении *диспетчера восстановления* и обсуждаются в разделе 5.3.

Свойства согласованности и изоляции описывают правильное поведение клиентов, действующих одновременно. Движок базы данных должен защищать клиентов от конфликтов друг с другом. Типичная стратегия состоит в том, чтобы определить, когда может произойти конфликт, и заставить одного из клиентов подождать, пока конфликт не исчезнет. Эти функции находятся в ведении *диспетчера конкуренции* и обсуждаются в разделе 5.4.

5.2. Использование транзакций в SimpleDB

Прежде чем углубляться в детали работы диспетчеров конкуренции и восстановления, важно понять правила использования транзакций клиентами. В SimpleDB каждой транзакции соответствует свой объект `Transaction`; его API показан в листинге 5.2.

Листинг 5.2. API транзакций в SimpleDB

Transaction

```
public Transaction(FileMgr fm, LogMgr lm, BufferMgr bm);
public void commit();
public void rollback();
public void recover();

public void pin(BlockId blk);
public void unpin(BlockId blk);
public int  getInt(BlockId blk, int offset);
public String getString(BlockId blk, int offset);
public void setInt(BlockId blk, int offset, int val, boolean okToLog);
public void setString(BlockId blk, int offset, String val, boolean okToLog);
public int availableBuffs();

public int   size(String filename);
public Block append(String filename);
public int   blockSize();
```

Методы класса `Transaction` делятся на три категории. В первую входят методы, так или иначе влияющие на продолжительность жизни транзакции. Конструктор начинает новую транзакцию, методы `commit` и `rollback` завершают ее, а метод `recovery` откатывает все незафиксированные транзакции. Методы `commit` и `rollback` автоматически открепляют страницы, закрепленные транзакцией.

Во вторую категорию входят методы доступа к буферам. Транзакция скрывает существование буферов от своего клиента. Когда клиент вызывает метод `pin` блока, транзакция сохраняет буфер внутри, но не возвращает его клиенту. Когда клиент вызывает такой метод, как `getInt`, он передает ссылку `BlockId`. Транзакция находит соответствующий буфер, вызывает метод `getInt` страницы в буфере и возвращает результат клиенту.

Так как буфер недоступен клиенту непосредственно, транзакция может вызывать все необходимые методы диспетчеров конкуренции и восстановления. Например, реализация `setInt` получает все необходимые блокировки (для управления конкурентным выполнением), записывает текущее значение из буфера в журнал (на случай, если потребуется выполнить восстановление), после чего производит изменение в буфере. В четвертом аргументе методам `setInt` и `setString` передается логический флаг, указывающий на необходимость журналирования изменения. Обычно этот флаг имеет значение `true`, но иногда (например, при форматировании нового блока или отмене транзакции), когда журналирование не требуется, в этом флаге передается `false`.

Листинг 5.3. Тестирование класса Transaction в SimpleDB

```
public class TxTest {
    public static void main(String[] args) throws Exception {
        SimpleDB db = new SimpleDB("txtest", 400, 8);
        FileMgr fm = db.fileMgr();
        LogMgr lm = db.logMgr();
        BufferMgr bm = db.bufferMgr();
        Transaction tx1 = new Transaction(fm, lm, bm);
        BlockId blk = new BlockId("testfile", 1);
        tx1.pin(blk);
        // Не журналировать начальные значения в блоке.
        tx1.setInt(blk, 80, 1, false);
        tx1.setString(blk, 40, "one", false);
        tx1.commit();

        Transaction tx2 = new Transaction(fm, lm, bm);
        tx2.pin(blk);
        int ival = tx2.getInt(blk, 80);
        String sval = tx2.getString(blk, 40);
        System.out.println("initial value at location 80 = " + ival);
        System.out.println("initial value at location 40 = " + sval);
        int newival = ival + 1;
        int newsval = sval + "!";
        tx2.setInt(blk, 80, newival, true);
        tx2.setString(blk, 40, newsval, true);
        tx2.commit();

        Transaction tx3 = new Transaction(fm, lm, bm);
        tx3.pin(blk);
        System.out.println("new value at location 80 = "
                        + tx3.getInt(blk, 80));
        System.out.println("new value at location 40 = "
                        + tx3.getString(blk, 40));
        tx3.setInt(blk, 80, 9999, true);
        System.out.println("pre-rollback value at location 80 = "
                        + tx3.getInt(blk, 80));
        tx3.rollback();

        Transaction tx4 = new Transaction(fm, lm, bm);
        tx4.pin(blk);
        System.out.println("post-rollback at location 80 = "
                        + tx4.getInt(blk, 80));
        tx4.commit();
    }
}
```

В третью категорию входят три метода, связанных с диспетчером файлов. Метод `size` читает маркер конца файла, а `append` изменяет его; чтобы избежать потенциальных конфликтов, эти методы вызывают методы диспетчера конкуренции. Метод `blockSize` существует для удобства клиентов, которым он может понадобиться.

Код в листинге 5.3 иллюстрирует простые примеры использования методов класса `Transaction`. Он выполняет четыре транзакции, которые решают те же задачи, что и класс `BufferTest` в листинге 4.6. Все четыре транзакции обращаются к блоку 1 в файле «testfile». Транзакция `tx1` инициализирует значения в смещениях 80 и 40; эти обновления не регистрируются в журнале. Транзакция `tx2` читает

эти значения, выводит их и увеличивает. Транзакция tx3 читает и выводит увеличенные значения, затем записывает целое число 9999 и откатывается. Транзакция tx4 читает целое число, чтобы убедиться, что откат был выполнен правильно.

Сравните этот код с кодом из главы 4 и обратите внимание, какую работу класс Transaction делает за вас: он управляет буферами, генерирует журнальные записи для каждого обновления и сохраняет их в файле журнала, а также может откатить ваши изменения, если вы того потребуете. Но особенно важно, как этот класс работает за кулисами, гарантируя соблюдение свойств ACID. Например, представьте, что вы случайно прерываете программу во время ее выполнения. После повторного запуска движка базы данных изменения, выполненные зафиксированными транзакциями, сохранятся на диске (долговечность), а изменения, выполненные незафиксированными транзакциями, будут отменены (атомарность).

Кроме того, класс Transaction также гарантирует, что программа будет удовлетворять свойству изоляции. Рассмотрим код, использующий транзакцию tx2. Переменные newival и newsval (см. код в листинге 5.3, выделенный жирным) инициализируются, как показано ниже:

```
int newival = ival + 1;
String newsval = sval + "!";
```

Этот код предполагает, что значения со смещениями 80 и 40 в блоке не изменились. Однако в отсутствие контроля за одновременными операциями данное предположение может оказаться неверным. Эта проблема была описана в сценарии «неповторяемого чтения» в разделе 2.2.3. Предположим, что поток, выполняющий tx2, приостанавливается сразу после инициализации ival и sval, и другая программа изменяет значения со смещениями 80 и 40. Тогда значения в переменных ival и sval окажутся устаревшими, и транзакция tx2 должна снова вызвать getInt и getString, чтобы получить их правильные значения. Класс Transaction решает эту проблему, гарантируя невозможность такой ситуации, поэтому этот код гарантированно выполнится правильно.

5.3. Управление восстановлением

Диспетчер восстановления – это компонент движка базы данных, который читает и обрабатывает записи в журнале. Он выполняет три функции: сохраняет записи в журнале, выполняет откат транзакций и восстанавливает базу данных после сбоя системы. Все эти функции подробно рассматриваются в этом разделе.

5.3.1. Записи в журнале

Чтобы иметь возможность откатить транзакцию, диспетчер восстановления регистрирует действия транзакции. В частности, он сохраняет *журнальную запись* каждый раз, когда выполняется действие, требующее журналирования. Существует четыре основных типа журнальных записей: *начальные записи* (START), *записи фиксации* (COMMIT), *записи отката* (ROLLBACK) и *записи обновления*. Я буду следовать за особенностями SimpleDB и делить записи обновления на два вида: записи обновления целых чисел (SETINT) и записи обновления строк (SETSTRING).

Журнальные записи генерируются в ходе выполнения следующих журналируемых действий:

- начальная запись создается в момент начала транзакции;
- запись фиксации или отката создается в момент завершения транзакции;
- запись обновления создается, когда транзакция изменяет значение.

Еще одно потенциально журналируемое действие – добавление блока в конец файла. Если после этого происходит откат транзакции, новый блок, добавленный методом `append`, может быть удален из файла. Для простоты я буду игнорировать данную возможность, но в упражнении 5.48 вам будет предложено решить эту задачу.

Для примера рассмотрим код в листинге 5.3 и предположим, что `tx1` имеет идентификационный номер 1 и т. д. В листинге 5.4 показаны журнальные записи, сгенерированные этим кодом.

Листинг 5.4. Журнальные записи, сгенерированные кодом из листинга 5.3

```
<START, 1>
<COMMIT, 1>
<START, 2>
<SETINT, 2, testfile, 1, 80, 1, 2>
<SETSTRING, 2, testfile, 1, 40, one, one!>
<COMMIT, 2>
<START, 3>
<SETINT, 3, testfile, 1, 80, 2, 9999>
<ROLLBACK, 3>
<START, 4>
<COMMIT, 4>
```

Каждая запись содержит описание ее типа (`START`, `SETINT`, `SETSTRING`, `COMMIT` или `ROLLBACK`) и идентификационный номер соответствующей транзакции. Записи обновления содержат пять дополнительных значений: имя файла и номер измененного блока, смещение, где произошло изменение, а также старое и новое значения в этом смещении.

Часто несколько транзакций будут одновременно выводить записи в журнал, поэтому записи, принадлежащие разным транзакциям, будут перемежаться.

5.3.2. Откат

Одно из назначений журнала – помочь диспетчеру восстановления *откатить* указанную транзакцию T. Диспетчер восстановления откатывает транзакцию, отменяя ее изменения. Поскольку эти изменения перечислены в записях обновления, диспетчер может просто просканировать журнал, найти каждую запись обновления и восстановить исходное содержимое каждого измененного значения. Соответствующий алгоритм представлен ниже.

1. Назначить текущей самую последнюю запись в журнале.
2. Выполнять следующие шаги, пока текущей не станет начальная запись транзакции T:
   a) если текущая запись является записью обновления транзакции T, то записать старое значение в указанное местоположение;
   b) перейти к предыдущей записи в журнале.
3. Добавить запись отката в конец журнала.

Есть две причины, почему этот алгоритм читает журнал не в прямом, а в обратном направлении. Первая причина заключается в том, что в начале файла журнала будут содержаться записи о давно выполненных транзакциях, тогда как искомые записи, скорее всего, находятся ближе к концу журнала, и поэтому эффективнее начинать читать журнал с конца. Вторая, более важная причина – обеспечение правильности работы алгоритма. Представьте, что значение в указанном местоположении было изменено несколько раз. В результате в журнале появится несколько записей для этого местоположения, каждая из которых описывает свое значение. Значение для восстановления должно браться из самой ранней из этих записей. Именно это и случится, если записи в журнале обрабатывать в обратном порядке.

5.3.3. Восстановление

Другое назначение журнала состоит в *восстановлении* базы данных. Восстановление выполняется каждый раз, когда запускается движок базы данных. Цель – восстановить базу данных до надлежащего состояния. Под словами «надлежащее состояние» подразумевается:

- откат всех незавершенных транзакций;
- запись на диск всех изменений, произведенных зафиксированными транзакциями.

Когда движок базы данных запускается после штатного завершения работы, база данных уже должна находиться в надлежащем состоянии, потому что процедура штатного завершения работы состоит в том, чтобы дождаться завершения существующих транзакций и затем сбросить на диск все буферы. Однако если движок завершился неожиданно в результате сбоя, какие-то транзакции могут оказаться незавершенными. Поскольку движок не может их завершить, соответствующие им изменения должны быть отменены. Также могут иметься зафиксированные транзакции, изменения которых еще не были сброшены на диск; эти изменения необходимо записать.

Диспетчер восстановления предполагает, что транзакция завершилась, если файл журнала содержит соответствующую запись фиксации или отката. То есть если транзакция была зафиксирована до сбоя системы, но соответствующая запись фиксации не попала в файл журнала, тогда диспетчер восстановления обработает такую транзакцию, как если бы она не завершилась. Эта ситуация может показаться несправедливой, но диспетчер восстановления не может предложить ничего лучшего. Ему доступно только то, что содержится в файле журнала, потому что все остальные результаты действий транзакции были утрачены из-за сбоя системы.

На самом деле откат зафиксированной транзакции не только несправедлив, но и нарушает ACID-свойство *долговечности*. Поэтому диспетчер восстановления должен гарантировать невозможность такого сценария. С этой целью журнальные записи фиксации сбрасываются на диск до завершения операции фиксации. Как вы наверняка помните, при сбросе определенной журнальной записи на диск также сбрасываются все предыдущие записи. Поэтому когда диспетчер восстановления обнаруживает запись фиксации в журнале, он знает, что все записи обновления, соответствующие этой транзакции, также имеются в журнале.

Каждая запись обновления в журнале содержит старое и новое значения. Старое значение используется для отмены изменения, а новое – для его повторного выполнения. Алгоритм восстановления представлен ниже.

// Этап отмены

1) Для каждой записи в журнале (при чтении в обратном порядке, начиная с конца):
   a) если текущая запись является записью фиксации, то добавить эту транзакцию в список зафиксированных транзакций:
   b) если текущая запись является записью отката, то добавить эту транзакцию в список отмененных транзакций;
   c) если текущая запись является записью обновления и соответствует транзакции, которая не была зафиксирована или отменена, то восстановить старое значение в соответствующем местоположении.

// Этап повторного выполнения

2) Для каждой записи в журнале (при чтении в прямом порядке, с начала):
   a) если текущая запись является записью обновления и соответствует зафиксированной транзакции, тогда восстановить новое значение в соответствующем местоположении.

Этап 1 отменяет незавершенные транзакции. Как при выполнении алгоритма отката, для правильной отмены транзакций журнал необходимо читать в обратном направлении, от конца. При чтении журнала в обратном порядке запись фиксации транзакции всегда будет обнаруживаться раньше записи обновления; поэтому, встретив запись обновления, алгоритм будет знать, как следует поступить с ней – отменить или нет.

На этапе 1 важно прочитать весь журнал. Например, самая первая транзакция могла внести изменения в базу данных, прежде чем войти в бесконечный цикл. Эта запись обновления не будет найдена, если не прочитать журнал целиком.

Этап 2 повторно выполняет все зафиксированные транзакции. Поскольку диспетчер восстановления не знает, какие буферы были сброшены, а какие нет, он повторяет все изменения, сделанные всеми зафиксированными транзакциями.

На этапе 2 диспетчер восстановления читает журнал в прямом направлении, с самого начала. Он знает, какие изменения следует записать, потому что уже имеет список зафиксированных транзакций, полученный на этапе 1. Обратите внимание, что на этапе 2 журнал *должен* читаться в прямом направлении. Если несколько зафиксированных транзакций изменят одно и то же значение, то окончательным будет самое последнее изменение.

Алгоритм восстановления не учитывает текущее состояние базы данных. Он сохраняет старые или новые значения в базе данных, несмотря на текущие значения в соответствующих смещениях, потому что журнал *точно* сообщает, каким должно быть содержимое базы данных. Эта особенность имеет два следствия:

- восстановление идемпотентно;
- в процессе восстановления может быть выполнено обращений к диску больше, чем необходимо.

Под *идемпотентостью* я подразумеваю тот факт, что многократное выполнение алгоритма восстановления приведет к тому же результату, что и однократное. То есть вы получите тот же результат, даже если повторно запустите алгоритм восстановления после частичного его выполнения. Это свойство важно для правильной работы алгоритма. Например, представьте, что система баз данных потерпела сбой, когда находилась в середине алгоритма восстановления. После повторного запуска система баз данных снова запустит алгоритм восстановления. Если бы алгоритм не был идемпотентным, повторный запуск повредил бы базу данных.

Поскольку этот алгоритм не анализирует текущее содержимое базы данных, он может производить избыточные операции изменения. Например, допустим, что изменения, сделанные зафиксированной транзакцией, были сброшены на диск; тогда повторная запись этих изменений на этапе 2 установит значения, которые уже были сохранены. Мы могли бы пересмотреть алгоритм, чтобы избавиться от ненужных операций записи; именно это будет предложено сделать в упражнении 5.44.

5.3.4. Восстановление только с отменой и только с повторным выполнением

Алгоритм восстановления, представленный в предыдущем разделе, производит отмены и повторные выполнения. Движок базы данных мог бы использовать упрощенный алгоритм и производить только отмены или только повторные выполнения, то есть либо этап 1, либо этап 2, но не оба.

5.3.4.1. Восстановление только с отменой

Диспетчер восстановления может пропустить этап 2, если будет уверен, что все зафиксированные изменения записаны на диск. Это возможно, если принудительно сохранять буферы на диске *перед* сохранением записей фиксации в журнале. Этот подход описан в следующем алгоритме. Диспетчер восстановления должен выполнить шаги этого алгоритма в указанном порядке.

1. Сбросить на диск буферы, измененные транзакцией.
2. Сохранить запись фиксации в журнал.
3. Сбросить на диск страницу журнала с записью фиксации.

Какой алгоритм восстановления лучше: только с отменой или с отменой и повторным выполнением? Восстановление только с отменой выполняется быстрее, потому что для этого требуется выполнить лишь один проход по файлу журнала. Кроме того, размер журнала получается немного меньше, потому что отпадает необходимость сохранять новые значения в записях обновления. С другой стороны, операция *фиксации* значительно медленнее, потому что в ходе ее выполнения приходится сбрасывать на диск измененные буферы. Если исходить из предположения, что сбои в системе происходят нечасто, тогда алгоритм с отменой и повторным выполнением выглядит предпочтительнее, поскольку в этом случае не только фиксация транзакций выполняется быстрее, но и требуется меньшее число обращений к диску благодаря отложенной записи буферов.

5.3.4.2. Восстановление только с повторным выполнением

Этап 1 тоже можно опустить, если незафиксированные буферы никогда не будут записываться на диск. Диспетчер восстановления может обеспечить это, сохраняя буферы транзакции закрепленными, пока та не будет зафиксирована. Закрепленный буфер не будет выбираться для замещения, а значит, его содержимое не будет сбрасываться на диск. Кроме того, для отката транзакции достаточно будет просто «стереть» измененные буферы. Необходимые изменения в алгоритме отката при использовании подхода к восстановлению только с повторным выполнением представлены ниже.

Для каждого буфера, измененного транзакцией:

a) отметить буфер как незанятый (в SimpleDB достаточно установить номер блока равным −1);
b) отметить буфер как неизмененный;
c) открепить буфер.

Восстановление только с повторным выполнением тоже будет выполняться быстрее, чем восстановление с отменой и повторным выполнением, потому что есть возможность игнорировать незафиксированные транзакции. Однако при таком подходе требуется, чтобы каждая транзакция удерживала закрепленными свои буферы со всеми измененными блоками, что увеличивает конкуренцию за буферы в системе. В большой базе данных это может серьезно повлиять на производительность всех транзакций, что делает восстановление только с повторным выполнением опасным выбором.

А теперь подумайте: можно ли объединить предпосылки подходов к восстановлению только с отменой и только с повторным выполнением, чтобы получить алгоритм, не требующий ни этапа 1, ни этапа 2 (см. упражнение 5.19)?

5.3.5. Журналирование с опережением

Этап 1 алгоритма восстановления в разделе 5.3.3 требует дальнейшего изучения. Напомню, что на этом этапе последовательно просматриваются записи в журнале и выполняется отмена каждой записи обновления, принадлежащей незавершенной транзакции. Обосновывая правильность этого этапа, я исходил из предположения, что *для всех изменений в незавершенных транзакциях имеются соответствующие записи в файле журнала*. В противном случае база данных будет повреждена, потому что не будет никакого способа отменить такие обновления.

Поскольку система может аварийно завершиться в любой момент, единственный способ удовлетворить это предположение – заставить диспетчера журнала сбрасывать каждую запись на диск сразу после ее создания. Но, как было показано в разделе 4.2, эта стратегия крайне неэффективна. Должен быть лучший способ.

Проанализируем, что может пойти не так. Предположим, что незавершенная транзакция изменила содержимое страницы и создала в журнале соответствующую запись обновления. В случае сбоя сервера возможны четыре варианта:

a) страница с измененными данными и журнальная запись будут сброшены на диск;
b) на диск будет сброшена только страница с данными;

c) на диск будет сброшена только журнальная запись;
d) ни страница, ни журнальная запись не будут сброшены на диск.

Рассмотрим эти варианты по очереди. В варианте «a» алгоритм восстановления благополучно найдет запись в журнале и отменит изменение в блоке данных на диске. В варианте «b» алгоритм восстановления не найдет записи в журнале и поэтому не отменит изменение в блоке данных. Это серьезная проблема. В варианте «c» алгоритм восстановления найдет запись в журнале и отменит несуществующее изменение в блоке. Поскольку блок фактически не был изменен, время на его восстановление будет потрачено впустую, но это не породит никаких ошибок. В варианте «d» алгоритм восстановления не найдет записи в журнале, но так как изменения в блоке данных не были сохранены, то отменять все равно нечего.

То есть проблема возникает только в варианте «b». Движок базы данных предотвращает этот вариант, сбрасывая запись обновления в файл журнала на диске до сохранения соответствующей измененной страницы из буфера. Эта стратегия называется *журналированием с опережением*. Обратите внимание, что журнал может описывать изменения в базе данных, которых на самом деле не было (см. вариант «c» выше), но если изменения действительно были сохранены в базе данных, для них всегда будут иметься соответствующие записи в журнале на диске.

Стандартный способ реализации журналирования с опережением – для каждого буфера хранить номер LSN его последнего изменения. Прежде чем вытолкнуть страницу на диск, буфер сообщит диспетчеру журнала, что тот должен сохранить запись с номером LSN. Благодаря этому журнальная запись, соответствующая изменению, всегда будет сохраняться на диске до сохранения самого блока с данными.

5.3.6. Блокирующие контрольные точки

Журнал хранит историю всех изменений в базе данных. Со временем файл журнала может стать огромным, и его размер может даже превысить суммарный размер файлов с данными. Попытка прочитать весь журнал во время восстановления, отменяя или повторяя каждое изменение в базе данных, может оказаться неосуществимой за разумное время. Для преодоления этой проблемы были разработаны стратегии восстановления, позволяющие читать только часть журнала. Основная идея заключается в том, что алгоритм восстановления может прекратить поиск в журнале, узнав, что:

- все более ранние записи в журнале были созданы завершившимися транзакциями;
- буферы, измененные этими транзакциями, были записаны на диск.

Первый пункт в этом списке относится к этапу отмены в алгоритме восстановления. Выполнение этого условия говорит об отсутствии других незафиксированных транзакций, которые нужно откатить. Второй пункт относится к этапу повторного выполнения изменений; выполнение условия говорит о том, что все транзакции, зафиксированные ранее, повторять не требуется. Обратите внимание, что если диспетчер восстановления реализует восстановление только с отменой, тогда второе условие будет выполняться всегда.

В любой момент диспетчер восстановления может выполнить блокирующую контрольную точку (quiescent checkpoint), как описывается в следующем алгоритме. Шаг 2 этого алгоритма гарантирует удовлетворение первого условия, а шаг 3 – второго условия в списке выше.

1. Прекратить прием новых транзакций.
2. Дождаться завершения всех активных транзакций.
3. Сбросить на диск все измененные буферы.
4. Добавить в журнал запись блокирующей контрольной точки и сбросить эту запись на диск.
5. Возобновить прием новых транзакций.

Запись блокирующей контрольной точки играет роль маркера. Встретив такую запись в процессе выполнения этапа 1 с обратным перемещением по журналу, алгоритм восстановления может смело игнорировать все более ранние записи и приступить к выполнению этапа 2, начав прямое перемещение по журналу с этой точки. Другими словами, алгоритму восстановления не требуется просматривать записи, предшествующие записи блокирующей контрольной точки.

Лучшее время для создания записи блокирующей контрольной точки – в момент запуска системы, после завершения операции восстановления и до начала приема новых транзакций. Поскольку алгоритм восстановления только что завершил обработку журнала, запись о контрольной точке поможет ему больше никогда не проверять предыдущие записи в журнале.

Для примера рассмотрим журнал в листинге 5.5. Этот пример иллюстрирует следующее: во-первых, новые транзакции не могут быть запущены после начала процесса контрольной точки; во-вторых, запись контрольной точки сбрасывается на диск сразу после завершения последней транзакции и сохранения буферов; и в-третьих, новые транзакции могут начаться, как только запись контрольной точки будет сохранена на диске.

Листинг 5.5. Журнал с блокирующей контрольной точкой

```
<START, 0>
<SETINT, 0, junk, 33, 8, 542, 543>
<START, 1>
<START, 2>
<COMMIT, 1>
<SETSTRING, 2, junk, 44, 20, hello, ciao>
    // Здесь началась процедура блокирующей контрольной точки
<SETSTRING, 0, junk, 33, 12, joe, joseph>
<COMMIT, 0>
    // Здесь пыталась начаться транзакция 3, но была отложена
<SETINT, 2, junk, 66, 8, 0, 116>
<COMMIT, 2>
<CHECKPOINT>
<START, 3>
<SETINT, 3, junk, 33, 8, 543, 120>
```

5.3.7. Неблокирующие контрольные точки

Блокирующая контрольная точка проста в реализации и понятна. Однако она требует, чтобы база данных была недоступна, пока диспетчер восстановления

ожидает завершения имеющихся транзакций. Во многих случаях это серьезный недостаток – мало кому понравится, если его база данных периодически будет прекращать откликаться на запросы. Поэтому был разработан алгоритм установки контрольных точек, не требующий состояния покоя:

1. Пусть T1, ..., Tk – выполняющиеся в данный момент транзакции.
2. Прекратить прием новых транзакций.
3. Сбросить на диск все измененные буферы.
4. Вывести в журнал запись <NQCKPT T1, ..., Tk>.
5. Возобновить прием новых транзакций.

Этот алгоритм создает запись о контрольной точке другого типа – *неблокирующей контрольной точке* (nonquiescent checkpoint)[1]. Запись неблокирующей контрольной точки содержит список транзакций, выполняемых в данный момент.

Алгоритм восстановления доработан следующим образом. На этапе 1 он читает журнал в обратном направлении, как и раньше, и запоминает завершенные транзакции. Встретив запись неблокирующей контрольной точки <NQCKPT T1, ..., Tk>, он определяет, какие из этих транзакций не были завершены, а затем продолжает читать журнал в обратном направлении, пока не встретит начальную запись самой ранней из этих транзакций. Все другие записи в журнале, предшествующие этой, можно игнорировать.

Для примера вернемся к журналу в листинге 5.5. При использовании неблокирующей контрольной точки журнал будет выглядеть, как показано в листинге 5.6. Обратите внимание, что запись <NQCKPT...> появляется в журнале в том месте, где начался процесс контрольной точки в листинге 5.5, и сообщает, что транзакции 0 и 2 все еще выполняются в этот момент. Этот журнал отличается от журнала в листинге 5.5 тем, что в нем отсутствует запись о фиксации транзакции 2.

Листинг 5.6. Журнал с неблокирующей контрольной точкой

```
<START, 0>
<SETINT, 0, junk, 33, 8, 542, 543>
<START, 1>
<START, 2>
<COMMIT, 1>
<SETSTRING, 2, junk, 44, 20, hello, ciao>
<SETSTRING, 0, junk, 33, 12, joe, joseph>
<COMMIT, 0>
<START, 3>
<NQCKPT, 0, 2>
<SETINT, 2, junk, 66, 8, 0, 116>
<SETINT, 3, junk, 33, 8, 543, 120>
```

Если алгоритм восстановления получит такой журнал во время запуска системы, он будет действовать на этапе 1 следующим образом:

---

[1] Если вы удивлены, почему алгоритм по-прежнему не принимает новые транзакции, загляните в упражнения 5.12 и 5.13. – *Прим. ред.*

- встретив запись <SETINT, 3, ...>, он проверит присутствие транзакции 3 в списке зафиксированных транзакций. Поскольку в данный момент этот список пуст, алгоритм выполнит отмену, записав в блок 33 файла «junk» целое число 543 со смещением 8;
- запись <SETINT, 2, ...> будет обработана аналогично: в блок 66 файла «junk» будет записано целое число 0 со смещением 8;
- встретив запись <COMMIT, 0>, алгоритм добавит 0 в список зафиксированных транзакций;
- запись <SETSTRING, 0, ...> будет проигнорирована, потому что 0 присутствует в списке зафиксированных транзакций;
- встретив запись <NQCKPT 0, 2>, алгоритм будет знать, что может игнорировать все записи в журнале, предшествующие начальной записи транзакции 2, потому что транзакция 0 зафиксирована;
- встретив запись <START, 2>, алгоритм перейдет к этапу 2 и начнет перемещение по журналу в прямом направлении;
- запись <SETSTRING, 0, ...> заставит алгоритм повторить изменение, потому что 0 присутствует в списке зафиксированных транзакций. В результате в блок 33 файла «junk» будет записана строка «joseph» со смещением 12.

5.3.8. Гранулярность элементов данных

Алгоритмы восстановления, описанные в этом разделе, в качестве единицы регистрации используют значения данных. То есть для каждого изменяемого значения создается своя запись в журнале, причем каждая запись содержит предыдущую и новую версии значения. Эта единица регистрации называется *элементом данных восстановления*. Размер элемента данных называется *гранулярностью*.

Вместо отдельных значений в качестве элементов данных диспетчер восстановления может использовать блоки или даже файлы. Например, предположим, что роль элемента данных играет блок. В этом случае запись обновления в журнале будет создаваться при каждом изменении блока и содержать предыдущую и новую версии блока.

При таком подходе в журнал потребуется сохранять меньше записей, если использовать алгоритм восстановления только с отменой. Предположим, что транзакция закрепляет блок, изменяет несколько значений в нем, а затем открепляет его. Вы можете сохранить исходное содержимое блока в одной записи вместо создания отдельных записей для каждого измененного значения. При этом, конечно, записи обновления в журнале будут очень велики; они будут хранить содержимое всего блока независимо от количества действительно измененных значений. Таким образом, журналировать блоки целиком имеет смысл, только если большинство транзакций выполняют множество изменений в одном блоке.

Теперь посмотрим, что получится, если в качестве элементов данных использовать файлы. Транзакция будет генерировать одну запись обновления для каждого файла, который она изменила. Каждая запись в журнале будет хранить все исходное содержимое каждого файла. Чтобы откатить такую транзакцию, достаточно просто заменить существующие файлы их исходными версиями. Этот подход почти наверняка менее практичен, чем подход с ис-

пользованием значений или блоков в качестве элементов данных, потому что каждая транзакция должна будет создать копию всего файла, независимо от того, сколько значений изменилось.

Хотя элементы данных, вмещающие файлы целиком, нецелесообразны для систем баз данных, они часто используются приложениями, не связанными с базами данных. Представьте, например, что ваш компьютер потерпел сбой, когда вы редактировали файл. После перезагрузки системы некоторые текстовые процессоры могут показать вам две версии файла: последнюю сохраненную версию и версию, существовавшую на момент сбоя. Это возможно, потому что такие текстовые процессоры записывают изменения не в исходный файл непосредственно, а в его копию, и при сохранении копируют измененный файл в исходный. Эта стратегия может служить грубым представлением версии журналирования на основе файлов.

5.3.9. Диспетчер восстановления в SimpleDB

Диспетчер восстановления в SimpleDB реализован в виде класса `RecoveryMgr` в пакете `simpledb.tx.recovery`. API класса `RecoveryMgr` показан в листинге 5.7.

Для каждой транзакции создается свой объект `RecoveryMgr`, методы которого сохраняют в журнал записи, соответствующие этой транзакции. Например, конструктор сохраняет начальную запись; методы `commit` и `rollback` выводят записи фиксации и отката соответственно; а методы `setInt` и `setString` извлекают старое значение из указанного буфера и сохраняют запись обновления в журнал. Методы `rollback` и `recover` выполняют алгоритмы отката и восстановления.

Объект `RecoveryMgr` реализует алгоритм восстановления только с отменой и в качестве элементов данных использует отдельные значения. Его код можно разделить на две части: создание записей для журнала и реализацию алгоритмов отката и восстановления.

5.3.9.1. Записи в журнале

Как упоминалось в разделе 4.2, диспетчер журнала интерпретирует записи в журнале как массивы байтов. Каждому виду записи соответствует свой класс, отвечающий за включение значений в массив. Первым значением в каждом массиве является целое число, обозначающее *оператор* журнальной записи; оператор может быть одной из констант: `CHECKPOINT`, `START`, `COMMIT`, `ROLLBACK`, `SETINT` или `SETSTRING`. Остальные значения зависят от оператора – запись блокирующей контрольной точки не имеет значений, запись обновления имеет пять дополнительных значений, а другие записи – по одному значению.

Листинг 5.7. API диспетчера восстановления в SimpleDB

RecoveryMgr

```
public RecoveryMgr(Transaction tx, int txnum, LogMgr lm, BufferMgr bm);
public void commit();
public void rollback();
public void recover();
public int  setInt(Buffer buff, int offset, int newval);
public int  setString(Buffer buff, int offset, String newval);
```

Все классы журнальных записей реализуют интерфейс LogRecord, который показан в листинге 5.8. Интерфейс определяет три метода, извлекающих компоненты из записи. Метод op возвращает оператор журнальной записи. Метод txNumber – идентификационный номер транзакции, создавшей данную запись. Этот метод имеет смысл для всех журнальных записей, кроме записей контрольных точек, для которых возвращается фиктивный номер. Метод undo отменяет изменение, описываемое данной записью. Только записи SETINT и SETSTRING будут иметь непустую реализацию метода undo, которая закрепит буфер с указанным блоком, запишет указанное значение в указанное смещение и открепит буфер.

Листинг 5.8. Интерфейс LogRecord для SimpleDB

```
public interface LogRecord {
    static final int CHECKPOINT = 0, START = 1, COMMIT = 2,
                     ROLLBACK = 3, SETINT = 4, SETSTRING = 5;
    int op();
    int txNumber();
    void undo(int txnum);

    static LogRecord createLogRecord(byte[] bytes) {
        Page p = new Page(bytes);
        switch (p.getInt(0)) {
        case CHECKPOINT:
            return new CheckpointRecord();
        case START:
            return new StartRecord(p);
        case COMMIT:
            return new CommitRecord(p);
        case ROLLBACK:
            return new RollbackRecord(p);
        case SETINT:
            return new SetIntRecord(p);
        case SETSTRING:
            return new SetStringRecord(p);
        default:
            return null;
        }
    }
}
```

Классы разных видов журнальных записей имеют схожую реализацию, поэтому достаточно будет рассмотреть один из классов, скажем SetStringRecord, представленный в листинге 5.9.

Листинг 5.9. Реализация класса SetStringRecord

```
public class SetStringRecord implements LogRecord {
   private int txnum, offset;
   private String val;
   private BlockId blk;
```

```
   public SetStringRecord(Page p) {
      int tpos = Integer.BYTES;
      txnum = p.getInt(tpos);
      int fpos = tpos + Integer.BYTES;
      String filename = p.getString(fpos);
      int bpos = fpos + Page.maxLength(filename.length());
      int blknum = p.getInt(bpos);
      blk = new BlockId(filename, blknum);
      int opos = bpos + Integer.BYTES;
      offset = p.getInt(opos);
      int vpos = opos + Integer.BYTES;
      val = p.getString(vpos);
   }

   public int op() {
      return SETSTRING;
   }

   public int txNumber() {
      return txnum;
   }

   public String toString() {
      return "<SETSTRING " + txnum + " " + blk + " " + offset + " "
             + val + ">";
   }

   public void undo(Transaction tx) {
      tx.pin(blk);
      tx.setString(blk, offset, val, false); // не регистрировать в журнале!
      tx.unpin(blk);
   }

   public static int writeToLog(LogMgr lm, int txnum, BlockId blk,
                                int offset, String val) {
      int tpos = Integer.BYTES;
      int fpos = tpos + Integer.BYTES;
      int bpos = fpos + Page.maxLength(blk.fileName().length());
      int opos = bpos + Integer.BYTES;
      int vpos = opos + Integer.BYTES;
      int reclen = vpos + Page.maxLength(val.length());
      byte[] rec = new byte[reclen];
      Page p = new Page(rec);
      p.setInt(0, SETSTRING);
      p.setInt(tpos, txnum);
      p.setString(fpos, blk.fileName());
      p.setInt(bpos, blk.number());
      p.setInt(opos, offset);
      p.setString(vpos, val);
      return lm.append(rec);
   }
}
```

Класс имеет два важных метода: статический метод `writeToLog`, записывающий шесть значений в массив байтов, представляющий запись `SETSTRING`, и конструктор, извлекающий шесть значений из этого массива. Рассмотрим реали-

зацию writeToLog. Сначала он вычисляет размер массива и смещение каждого значения в нем. Затем создает массив байтов такого же размера, заключает его в объект Page и использует методы setInt и setString этого объекта для записи значений в соответствующие смещения. Конструктор действует аналогично. Он определяет смещение каждого значения в странице и извлекает их.

Метод undo принимает один аргумент – идентификатор транзакции для отмены. Он закрепляет блок, использовавшийся транзакцией, записывает в блок исходное значение и открепляет блок. За сброс содержимого буфера на диск отвечает метод, который вызывает undo (rollback или recover).

5.3.9.2. Откат и восстановление

Класс RecoveryMgr реализует алгоритм восстановления только с отменой; его реализация приводится в листинге 5.10. Методы commit и rollback сбрасывают буферы транзакции перед сохранением соответствующих записей в журнале, а методы doRollback и doRecover выполняют один проход по журналу в обратном направлении.

Метод doRollback перебирает записи в журнале. Каждый раз, когда обнаруживается запись для данной транзакции, он вызывает метод undo записи, и останавливается, встретив начальную запись транзакции.

Метод doRecover реализован аналогично. Он читает записи из журнала, пока не достигнет записи блокирующей контрольной точки или конца журнала, сохраняя номера подтвержденных транзакций в списке. Отмена записей обновления незафиксированных транзакций осуществляется точно так же, как в методе rollback, с той лишь разницей, что обрабатываются все незафиксированные транзакции, а не только какая-то конкретная. Этот метод реализует немного иной алгоритм, чем представленный в разделе 5.3.3, потому что отменяет транзакции, которые уже были отменены. Это уточнение не делает алгоритм в разделе 5.3.3 неправильным, просто он оказывается менее эффективным. В упражнении 5.50 вам будет предложено усовершенствовать его.

Листинг 5.10. Реализация класса RecoveryMgr

```
public class RecoveryMgr {
    private LogMgr lm;
    private BufferMgr bm;
    private Transaction tx;
    private int txnum;

    public RecoveryMgr(Transaction tx, int txnum, LogMgr lm, BufferMgr bm) {
        this.tx = tx;
        this.txnum = txnum;
        this.lm = lm;
        this.bm = bm;
        StartRecord.writeToLog(lm, txnum);
    }

    public void commit() {
        bm.flushAll(txnum);
        int lsn = CommitRecord.writeToLog(lm, txnum);
        lm.flush(lsn);
    }
```

```
    public void rollback() {
        doRollback();
        bm.flushAll(txnum);
        int lsn = RollbackRecord.writeToLog(lm, txnum);
        lm.flush(lsn);
    }

    public void recover() {
        doRecover();
        bm.flushAll(txnum);
        int lsn = CheckpointRecord.writeToLog(lm);
        lm.flush(lsn);
    }

    public int setInt(Buffer buff, int offset, int newval) {
        int oldval = buff.contents().getInt(offset);
        BlockId blk = buff.block();
        return SetIntRecord.writeToLog(lm, txnum, blk, offset, oldval);
    }

    public int setString(Buffer buff, int offset, String newval)
    {
        String oldval = buff.contents().getString(offset);
        BlockId blk = buff.block();
        return SetStringRecord.writeToLog(lm, txnum, blk, offset, oldval);
    }

    private void doRollback() {
        Iterator<byte[]> iter = lm.iterator();
        while (iter.hasNext()) {
            byte[] bytes = iter.next();
            LogRecord rec = LogRecord.createLogRecord(bytes);
            if (rec.txNumber() == txnum) {
                if (rec.op() == START)
                    return;
                rec.undo(tx);
            }
        }
    }

    private void doRecover() {
        Collection<Integer> finishedTxs = new ArrayList<Integer>();
        Iterator<byte[]> iter = lm.iterator();
        while (iter.hasNext()) {
            byte[] bytes = iter.next();
            LogRecord rec = LogRecord.createLogRecord(bytes);
            if (rec.op() == CHECKPOINT)
                return;
            if (rec.op() == COMMIT || rec.op() == ROLLBACK)
                finishedTxs.add(rec.txNumber());
            else if (!finishedTxs.contains(rec.txNumber()))
                rec.undo(tx);
        }
    }
}
```

5.4. Диспетчер конкуренции

Диспетчер конкуренции (диспетчер транзакций) – это компонент движка базы данных, отвечающий за правильное выполнение конкурирующих транзакций. В этом разделе вы узнаете, что подразумевается под «правильным» выполнением, а также познакомитесь с некоторыми алгоритмами, обеспечивающими такую правильность.

5.4.1. Сериализуемое расписание

Последовательность вызовов методов, которые обращаются к файлам базы данных, в частности методов `get` и `set`, называют *историей* транзакции[1]. Например, как показано в листинге 5.11a, можно старательно выписать истории всех транзакций, выполняемых кодом в листинге 5.3. Другой способ выразить историю транзакции – перечислить задействованные блоки, как показано в листинге 5.11b. Например, история транзакции `tx2` сообщает, что она дважды прочитала данные из блока `blk`, а затем дважды записала данные в блок `blk`.

Листинг 5.11. Истории транзакций, выполняемых кодом в листинге 5.3: (a) история в виде перечня обращений к данным; (b) история в виде перечня обращений к блокам

```
tx1: setInt(blk, 80, 1, false);
     setString(blk, 40, "one", false);

tx2: getInt(blk, 80);
     getString(blk, 40);
     setInt(blk, 80, newival, true);
     setString(blk, 40, newsval, true);

tx3: getInt(blk, 80));
     getString(blk, 40));
     setInt(blk, 80, 9999, true);
     getInt(blk, 80));

tx4: getInt(blk, 80));
```

(a)

```
tx1: W(blk); W(blk)
tx2: R(blk); R(blk); W(blk); W(blk)
tx3: R(blk); R(blk); W(blk); R(blk)
tx4: R(blk)
```

(b)

Формально история транзакции – это последовательность *операций с базой данных*, выполненных данной транзакцией. Я намеренно использовал довольно расплывчатый термин «операция с базой данных». Часть (a) листинга 5.11 представляет операции с базой данных в виде последовательностей изменения значений, а часть (b) – в виде последовательностей чтения и записи дисковых блоков. Есть также другие способы представления операций с разной степенью гранулярности, которые обсуждаются в разделе 5.4.8. Но пока под операцией с базой данных я буду подразумевать чтение или запись дискового блока.

---

[1] Методы `size` и `append` тоже обращаются к файлу базы данных, но менее явно, чем методы `get`/`set`. Подробнее о влиянии методов `size` и `append` рассказывается в разделе 5.4.5.

Когда одновременно выполняется несколько транзакций, движок базы данных будет по очереди давать возможность поработать своим потокам выполнения, периодически приостанавливая один и запуская другой. (В SimpleDB это автоматически делается средой выполнения Java.) В результате фактическая последовательность операций, выполняемых диспетчером конкуренции, будет иметь вид непредсказуемой череды историй работающих транзакций. Эта череда называется *расписанием*.

Задача управления конкурентным выполнением – организовать правильное расписание, то есть чередование, выполняемых операций. Но что значит «правильное»? Рассмотрим простейшее расписание из возможных, в котором все транзакции выполняются последовательно (как, например, в листинге 5.11). Операции в этом расписании не будут перемежаться между собой, то есть расписание будет иметь вид простой последовательности историй всех транзакций. Этот вид расписания называется *последовательным расписанием*.

Управление конкурентным выполнением основывается на предположении, что последовательное расписание *должно быть* правильным, потому что в нем конкуренция как таковая отсутствует. Рассуждая о правильности, интересно отметить, что разные последовательные расписания одних и тех же транзакций могут давать разные результаты. Например, рассмотрим две транзакции, T1 и T2, имеющие следующие идентичные истории:

```
T1: W(b1); W(b2)
T2: W(b1); W(b2)
```

Эти транзакции имеют совершенно одинаковые истории (то есть обе сначала выполняют запись в блок b1, а затем в блок b2), но сами транзакции могут быть не идентичны. Например, T1 может записать символ «X» в начале каждого блока, а T2 – символ «Y». Если T1 выполнится до T2, в блоках сохранятся значения, записанные транзакцией T2, а если они выполнятся в обратном порядке, в блоках сохранятся значения, записанные транзакцией T1.

В этом примере транзакции T1 и T2 имеют разные мнения о том, что должны содержать блоки b1 и b2. А поскольку с точки зрения движка базы данных все транзакции равны, нельзя сказать, что один результат является более правильным, чем другой. Таким образом, мы вынуждены признать, что *любой* результат последовательного расписания является правильным. То есть может быть несколько правильных результатов.

Непоследовательное расписание называется *сериализуемым*, если дает тот же результат, что и некоторое последовательное расписание[1]. Поскольку последовательные расписания верны, сериализуемые расписания также должны быть правильными. Рассмотрим для примера следующее непоследовательное расписание транзакций, представленных выше:

```
W1(b1); W2(b1); W1(b2); W2(b2)
```

Здесь W1(b1) означает, что транзакция T1 записывает данные в блок b1, и т. д. Это расписание соответствует выполнению первой половины T1, затем

[1] Термин *сериализуемый* (serializable) также широко используется в Java – сериализуемым классом называют класс, экземпляры которого можно представить в виде потока байтов. К сожалению, такое значение этого термина не имеет абсолютно ничего общего с его значением применительно к базам данных.

первой половины T2, второй половины T1 и второй половины T2. Это расписание сериализуемо, потому что эквивалентно последовательному выполнению сначала T1, а затем T2. А теперь рассмотрим другое расписание:

```
W1(b1); W2(b1); W2(b2); W1(b2)
```

Здесь сначала выполняется первая половина T1, потом вся T2, а затем вторая половина T1. В результате выполнения данного расписания блок b1 будет хранить значения, записанные T2, а блок b2 – значения, записанные T1. Этот результат нельзя получить никаким последовательным расписанием, поэтому данное расписание считается *несериализуемым*.

Вспомните ACID-свойство *изоляции*, которое утверждает, что каждая транзакция должна выполняться так, как если бы она была единственной транзакцией в системе. Несериализуемое расписание не обладает этим свойством. Поэтому мы вынуждены признать, что несериализуемые расписания неверны. Иначе говоря, расписание верно тогда и только тогда, когда оно сериализуемо.

5.4.2. Таблица блокировок

Движок базы данных обязан обеспечить сериализуемость всех расписаний. Для решения этой задачи часто используются *блокировки*, позволяющие отложить выполнение транзакции. Как можно использовать блокировку для обеспечения сериализуемости, рассказывается в разделе 5.4.3, а здесь мы просто рассмотрим работу механизма блокировок в целом.

Каждый блок имеет блокировки двух видов – *разделяемую* блокировку (*shared lock*, или *slock*) и *монопольную* блокировку (*exclusive lock*, или *xlock*). Если транзакция удерживает монопольную блокировку для блока, никакая другая транзакция не сможет установить какую-либо блокировку для этого блока; если транзакция удерживает разделяемую блокировку для блока, тогда другие транзакции смогут устанавливать разделяемые (но не монопольные) блокировки для этого же блока. Обратите внимание, что эти ограничения применяются только к другим транзакциям. Одна и та же транзакция может удерживать как разделяемую, так и монопольную блокировку для блока.

Таблица блокировок – это компонент движка базы данных, отвечающий за предоставление блокировок транзакциям. В SimpleDB таблицу блокировок реализует класс `LockTable`. Его API показан в листинге 5.12.

Листинг 5.12. API класса LockTable в SimpleDB

LockTable

```
public void sLock(Block blk);
public void xLock(Block blk);
public void unlock(Block blk);
```

Метод `sLock` запрашивает разделяемую блокировку для указанного блока. Если для этого блока уже была получена монопольная блокировка, метод переходит в режим ожидания, пока монопольная блокировка не будет снята. Метод `xLock` запрашивает монопольную блокировку для указанного блока. Он ждет, пока все другие транзакции снимут свои блокировки с этого блока. Метод `unlock` снимает блокировку с блока.

В листинге 5.13 представлен класс `ConcurrencyTest`, демонстрирующий некоторые примеры использования блокировок.

Листинг 5.13. Тестирование механизма блокировок

```
public class ConcurrencyTest {
    private static FileMgr fm;
    private static LogMgr lm;
    private static BufferMgr bm;

    public static void main(String[] args) {
        // инициализировать движок базы данных
        SimpleDB db = new SimpleDB("concurrencytest", 400, 8);
        fm = db.fileMgr();
        lm = db.logMgr();
        bm = db.bufferMgr();
        A a = new A(); new Thread(a).start();
        B b = new B(); new Thread(b).start();
        C c = new C(); new Thread(c).start();
    }

    static class A implements Runnable {
        public void run() {
            try {
                Transaction txA = new Transaction(fm, lm, bm);
                BlockId blk1 = new BlockId("testfile", 1);
                BlockId blk2 = new BlockId("testfile", 2);
                txA.pin(blk1);
                txA.pin(blk2);
                System.out.println("Tx A: request slock 1");
                txA.getInt(blk1, 0);
                System.out.println("Tx A: receive slock 1");
                Thread.sleep(1000);
                System.out.println("Tx A: request slock 2");
                txA.getInt(blk2, 0);
                System.out.println("Tx A: receive slock 2");
                txA.commit();
            }
            catch(InterruptedException e) {};
        }
    }

    static class B implements Runnable {
        public void run() {
            try {
                Transaction txB = new Transaction(fm, lm, bm);
                BlockId blk1 = new BlockId("testfile", 1);
                BlockId blk2 = new BlockId("testfile", 2);
                txB.pin(blk1);
                txB.pin(blk2);
                System.out.println("Tx B: request xlock 2");
                txB.setInt(blk2, 0, 0, false);
                System.out.println("Tx B: receive xlock 2");
                Thread.sleep(1000);
                System.out.println("Tx B: request slock 1");
                txB.getInt(blk1, 0);
                System.out.println("Tx B: receive slock 1");
                txB.commit();
            }
            catch(InterruptedException e) {};
        }
    }
```

```
    static class C implements Runnable {
        public void run() {
            try {
                Transaction txC = new Transaction(fm, lm, bm);
                BlockId blk1 = new BlockId("testfile", 1);
                BlockId blk2 = new BlockId("testfile", 2);
                txC.pin(blk1);
                txC.pin(blk2);
                System.out.println("Tx C: request xlock 1");
                txC.setInt(blk1, 0, 0, false);
                System.out.println("Tx C: receive xlock 1");
                Thread.sleep(1000);
                System.out.println("Tx C: request slock 2");
                txC.getInt(blk2, 0);
                System.out.println("Tx C: receive slock 2");
                txC.commit();
            }
            catch(InterruptedException e) {};
        }
    }
}
```

Метод `main` запускает три параллельных потока выполнения, соответствующих объектам классов `A`, `B` и `C`. Эти транзакции не используют блокировки явно. Вместо этого метод `getInt` класса `Transaction` получает разделяемую блокировку, метод `setInt` – монопольную блокировку, а метод `commit` освобождает все блокировки, установленные транзакцией. Последовательность установки и освобождения блокировок в каждой транзакции выглядит следующим образом:

```
txA: sLock(blk1); sLock(blk2); unlock(blk1); unlock(blk2)
txB: xLock(blk2); sLock(blk1); unlock(blk1); unlock(blk2)
txC: xLock(blk1); sLock(blk2); unlock(blk1); unlock(blk2)
```

Потоки приостанавливают выполнение, вызывая метод `sleep`, чтобы заставить транзакции чередовать свои запросы на получение блокировок. В результате события развиваются следующим образом:

1. Поток A устанавливает разделяемую блокировку для блока blk1.
2. Поток B устанавливает монопольную блокировку для блока blk2.
3. Поток C пытается и не может установить монопольную блокировку для блока blk1, потому что какая-то другая транзакция уже удерживает ее. В результате поток C переходит в ожидание.
4. Поток A не может установить разделяемую блокировку для блока blk2, потому что какая-то другая транзакция удерживает монопольную блокировку для него. В результате поток A тоже переходит в ожидание.
5. Поток B может продолжить работу. Он устанавливает разделяемую блокировку для блока blk1, потому что в данный момент никакая другая транзакция не владеет монопольной блокировкой для него. (Совершенно не важно, что поток C ждет возможности получить монопольную блокировку для этого же блока.)
6. Поток B освобождает блокировку для блока blk1, но это ничего не дает ожидающим потокам.

7. Поток B освобождает блокировку для блока blk2.
8. Теперь поток A может продолжить и установить разделяемую блокировку для блока blk2.
9. Поток A освобождает блокировку для блока blk1.
10. Наконец, поток C устанавливает монопольную блокировку для блока blk1.
11. Потоки A и C могут продолжить работу в любом порядке до завершения.

5.4.3. Протокол блокирования

Теперь рассмотрим вопрос использования блокировок для обеспечения сериализуемости всех расписаний. Рассмотрим две транзакции со следующими историями:

```
T1: R(b1); W(b2)
T2: W(b1); W(b2)
```

Что заставляет последовательные расписания давать разные результаты? Транзакции T1 и T2 выполняют запись в один и тот же блок b2, а это означает, что порядок выполнения операций имеет значение – конкуренцию «выигрывает» транзакция, выполнившая запись последней. Операции {`W1(b2)`, `W2(b2)`} называются *конфликтующими*. В общем случае две операции конфликтуют, если порядок их выполнения может привести к разным результатам. Если две транзакции имеют конфликтующие операции, их последовательные расписания могут давать разные (но одинаково правильные) результаты.

Этот конфликт является примером конфликта *запись–запись*. Второй тип конфликта – *чтение–запись*. Например, конфликтуют операции {`R1(b1)`, `W2(b1)`} – если сначала выполнится `R1(b1)`, тогда T1 прочитает одну версию блока b1, а если сначала выполнится `W2(b1)`, тогда T1 прочитает другую версию блока b1. Обратите внимание, что две операции чтения не могут конфликтовать друг с другом, равно как и любые операции с разными блоками.

Причина, по которой важно позаботиться о конфликтах, заключается в том, что они влияют на сериализуемость расписания. Порядок выполнения конфликтующих операций в непоследовательном расписании определяет эквивалентное последовательное расписание. Если в примере, приведенном выше, `W2(b1)` выполнится до `R1(b1)`, тогда в любом эквивалентном последовательном расписании T2 должна выполниться до T1. В общем случае, если все операции в T1 конфликтуют с операциями в T2, тогда все операции в T1 должны быть выполнены до или после любых конфликтующих с ними операций в T2. Бесконфликтные операции могут выполняться в любом порядке[1].

Избежать конфликтов запись–запись и чтение–запись можно с помощью блокировок. В частности, предположим, что все транзакции используют блокировки в соответствии со следующим протоколом:

---

[1] На самом деле можно создать запутанные примеры, в которых определенные конфликты запись–запись могут происходить в любом порядке (см. упражнение 5.26). Однако такие примеры редко встречаются на практике, поэтому мы не будем их рассматривать.

1) перед чтением блока установить разделяемую блокировку для него;
2) перед изменением блока установить монопольную блокировку для него;
3) после фиксации или отката транзакции освободить блокировки.

Из этого протокола можно сделать два важных вывода. Во-первых, если транзакция установила разделяемую блокировку для блока, то никакая другая активная транзакция не сможет выполнить запись в блок (потому что для этого ей необходимо получить монопольную блокировку). Во-вторых, если транзакция установила монопольную блокировку для блока, никакая другая активная транзакция не сможет обратиться к блоку ни для каких целей (потому что для этого ей необходимо удерживать блокировку). Все это подразумевает, что операция, выполняемая транзакцией, никогда не будет конфликтовать с предыдущей операцией другой активной транзакции. Иначе говоря, если все транзакции неукоснительно следуют протоколу блокирования, тогда:

- полученное расписание всегда будет сериализуемым (и, следовательно, правильным);
- эквивалентное последовательное расписание определяется порядком фиксации транзакций.

Вынуждая транзакции удерживать свои блокировки до завершения, данный протокол резко ограничивает конкуренцию в системе. Было бы хорошо, если бы транзакция могла снимать свои блокировки, когда они больше не нужны, это избавило бы другие транзакции от необходимости ждать слишком долго. Однако если транзакция освободит свои блокировки до завершения, могут возникнуть две серьезные проблемы: расписание может перестать быть сериализуемым, и другие транзакции могут прочитать незафиксированные изменения. Эти две проблемы обсуждаются далее.

5.4.3.1. Проблема сериализуемости

Как только транзакция освободит блокировку для блока, она не сможет установить блокировку для другого блока, не повлияв на сериализуемость. Чтобы понять, почему, рассмотрим транзакцию T1, которая освобождает блокировку (UL = UnLock) для блока x и затем устанавливает разделяемую блокировку (SL = Shared Lock) для блока y:

```
T1: ... R(x); UL(x); SL(y); R(y); ...
```

Допустим, что выполнение T1 приостанавливается где-то в промежутке между освобождением блокировки x и установкой блокировки y. В этот период T1 чрезвычайно уязвима, потому что блоки x и y разблокированы. Предположим теперь, что в этот момент запускается другая транзакция T2, которая блокирует блоки x и y, записывает в них свои данные, фиксируется и освобождает свои блокировки. В результате мы имеем следующую ситуацию: T1 должна выполниться до запуска T2, потому что прочитала версию блока x, существовавшую до того, как T2 выполнила запись в него. С другой стороны, T1 также должна выполниться после T2, чтобы прочитать версию блока y, измененного T2. В результате получившееся расписание не сериализуемо.

Можно показать, что обратное также верно – если транзакция устанавливает все необходимые блокировки перед разблокировкой любой из них, получающе-

еся расписание гарантированно будет сериализуемым (см. упражнение 5.27). Этот вариант протокола блокирования называется *двухфазным блокированием*. Это название подчеркивает, что в соответствии с данным протоколом транзакция имеет две фазы: фазу, в которой она накапливает блокировки, и фазу, в которой она освобождает блокировки.

Теоретически двухфазное блокирование является более общим протоколом, но получить от него какие-то преимущества в движке базы данных непросто. Обычно к тому времени, когда транзакция завершает доступ к своему последнему блоку (то есть когда она наконец может освободить блокировки), она в любом случае готова к фиксации. Таким образом, общий протокол двухфазного блокирования редко бывает эффективным на практике.

5.4.3.2. Чтение незафиксированных данных

Другая проблема с преждевременным освобождением блокировок (даже при использовании протокола двухфазного блокирования) – возможность чтения незафиксированных данных. Рассмотрим следующее неполное расписание:

```
... W1(b); UL1(b); SL2(b); R2(b); ...
```

В данном случае T1 записывает данные в блок b и разблокирует его; затем транзакция T2 блокирует блок b и читает данные из него. Если T1 в конце концов будет зафиксирована, то никаких проблем не возникает. Но представьте, что был выполнен откат T1. Тогда транзакцию T2 тоже придется откатить, потому что ее действия были основаны на изменениях, которых больше не существует. А если откатить T2, тогда может потребоваться откатить другие транзакции. Это явление известно как *каскадный откат*.

Когда движок базы данных позволяет транзакциям читать незафиксированные данные, это открывает более широкие возможности для одновременного выполнения, но при этом возникает риск, что транзакция, записавшая данные, не будет зафиксирована. Конечно, откаты происходят редко, и каскадные откаты случаются еще реже. Но вопрос в том, насколько допустим для движка базы данных *любой* риск возможного отката транзакции. Большинство коммерческих систем баз данных не хотят брать на себя этот риск, поэтому всегда ждут завершения транзакции, прежде чем снимать монопольные блокировки.

5.4.4. Взаимоблокировка

Протокол блокирования гарантирует сериализуемость расписаний, но не гарантирует, что все транзакции будут зафиксированы. В частности, транзакции могут попасть в состояние *взаимоблокировки* (в тупиковую ситуацию).

В разделе 4.5.1 приводился пример тупиковой ситуации, когда каждый из двух клиентских потоков ожидал освобождения буфера другим. То же возможно с блокировками. Взаимоблокировка возникает с образованием циклической взаимозависимости транзакций, когда первая транзакция ждет освобождения блокировки, удерживаемой второй транзакцией, вторая транзакция ждет освобождения блокировки, удерживаемой третьей транзакцией, и т. д. до последней транзакции, ожидающей освобождения блокировки, удерживаемой первой транзакцией. В такой ситуации ни одна из ожидающих транзакций не может продолжить работу, и все будут ждать вечно. В качестве простого при-

мера рассмотрим следующие две истории, где транзакции выполняют запись в одни и те же блоки, но в разном порядке:

```
T1: W(b1); W(b2)
T2: W(b2); W(b1)
```

Предположим, что T1 сначала установила блокировку для блока b1, а затем вступила в гонку за блокировку для блока b2. Если T1 получит ее первой, то T2 приостановится и дождется, пока T1 зафиксируется и снимет свои блокировки, после чего продолжит работу. В этом сценарии проблема не возникает. Но если T2 успеет первой получить блокировку для блока b2, то возникает тупик – T1 будет ждать, когда T2 разблокирует блок b2, а T2 – когда T1 разблокирует блок b1. Ни одна из транзакций не сможет продолжить работу.

Диспетчер конкуренции может обнаруживать взаимоблокировки, формируя граф ожидания. Граф имеет по одному узлу для каждой транзакции. Узлы будут соединены ребром, направленным от T1 к T2, если T1 ожидает блокировки, которую удерживает T2; каждое ребро отмечено номером блока, доступа к которому ждет транзакция. Каждый раз, когда какая-то транзакция запрашивает или освобождает блокировку, диспетчер обновляет граф. Например, граф ожидания, соответствующий описанному выше сценарию взаимоблокировки, показан на рис. 5.1.

Легко показать, что ситуация взаимоблокировки возникает тогда и только тогда, когда в графе образуется цикл[1] (см. упражнение 5.28). Когда диспетчер транзакций обнаруживает возникновение взаимоблокировки, он может устранить ее, просто откатив одну из транзакций, участвующих в цикле. Одна из разумных стратегий – откатить транзакцию, запрос блокировки в которой «образовал» цикл, хотя возможны и другие варианты (см. упражнение 5.29).

Проверка на наличие взаимоблокировок усложняется, если кроме потоков выполнения, ожидающих освобождения блокировок, учитывать еще и потоки, ожидающие доступа к буферам. Например, предположим, что пул буферов содержит только два буфера, и рассмотрим следующий сценарий:

```
T1: xlock(b1); pin(b4)
T2: pin(b2); pin(b3); xlock(b1)
```

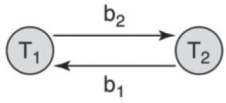


Рис. 5.1. Граф ожидания

Предположим, что транзакция T1 приостанавливается после получения блокировки для блока b1, а затем T2 закрепляет блоки b2 и b3. Попытавшись выполнить xlock(b1), T2 попадет в список ожидания блокировки, а T1 – в список ожидания буфера. Возникает тупик, хотя граф ожидания не содержит циклов.

Чтобы обнаружить такую тупиковую ситуацию, диспетчер блокировок должен не только хранить граф ожидания, но и знать, какие транзакции каких буферов ожидают. Как оказывается, добавить этот дополнительный аспект в ал-

[1] То есть, двигаясь из некоторой вершины по стрелкам, можно вернуться в нее же. Формально такой цикл в направленном графе называется контуром. – *Прим. ред.*

горитм обнаружения взаимоблокировок довольно трудно. Отважные читатели могут попробовать выполнить упражнение 5.37.

Проблема применения графа ожидания для обнаружения взаимоблокировок заключается в том, что граф довольно сложно поддерживать, а процедура поиска циклов в графе занимает много времени. Поэтому были разработаны более простые стратегии приблизительного определения взаимоблокировок. Эти стратегии являются консервативными, в том смысле, что они всегда обнаруживают взаимоблокировки, но при этом могут считать тупиковыми некоторые нетупиковые ситуации. В этом разделе рассматриваются две такие стратегии; еще одна будет представлена в упражнении 5.33.

Первая стратегия называется *wait-die* (*ожидание–отмена*). Вот алгоритм работы этой стратегии:

```
Пусть T1 запрашивает блокировку, которую удерживает T2.
    Если T1 старше T2, то:
            T1 ждет освобождения блокировки.
    Иначе:
            T1 откатывается (то есть «отменяется»).
```

Эта стратегия гарантирует невозможность возникновения взаимоблокировок, потому что граф ожидания будет содержать только ребра от старых транзакций к новым. Любую потенциальную возможность появления взаимоблокировки эта стратегия рассматривает как повод для отката. Например, предположим, что транзакция T1 старше, чем T2, и T2 запрашивает блокировку, в данный момент удерживаемую T1. Даже если этот запрос может не привести к взаимоблокировке немедленно, есть вероятность, что она возникнет позднее, когда T1 запросит блокировку, удерживаемую T2. Поэтому стратегия wait-die превентивно откатит T2.

Вторая стратегия использует ограничения по времени для обнаружения возможных взаимоблокировок. Если транзакция находится в состоянии ожидания дольше определенного времени, диспетчер транзакций решает, что она оказалась в состоянии взаимоблокировки, и откатывает ее. Вот алгоритм работы этой стратегии:

```
Пусть T1 запрашивает блокировку, которую удерживает T2.
  1. T1 ждет освобождения блокировки.
  2. Если T1 остается в списке ожидания слишком долго, то:
            T1 откатывается.
```

Независимо от используемой стратегии, диспетчер конкуренции должен ликвидировать взаимоблокировку, откатив активную транзакцию в надежде, что освобождение блокировок, установленных этой транзакцией, позволит остальным транзакциям завершиться. После отката транзакции диспетчер конкуренции генерирует исключение; в SimpleDB это исключение называется `LockAbortException`. Так же как `BufferAbortException`, о котором рассказывалось в главе 4, это исключение перехватывается клиентом JDBC, выполнявшим прерванную транзакцию, который затем решает, как его обработать. Например, клиент может просто завершиться или попытаться запустить транзакцию еще раз.

5.4.5. Конфликты на уровне файлов и фантомы

До сих пор в этой главе рассматривались конфликты, возникающие при чтении и записи блоков. Другой тип конфликтов вовлекает методы `size` и `append`, которые читают и записывают маркер конца файла. Эти два метода явно конфликтуют друг с другом: представьте, что транзакция T1 вызвала `append`, а вслед за ней транзакция T2 вызвала `size`; в этом случае T1 должна предшествовать T2 в любом последовательном порядке.

Одно из последствий этого конфликта известно как *проблема фантомов*. Предположим, что T2 многократно читает содержимое файла целиком и вызывает `size` перед каждой итерацией, чтобы определить, сколько блоков прочитать. Предположим также, что после того, как T2 прочитала файл первый раз, транзакция T1 начала добавлять в него новые блоки, заполнять их своими значениями и в конце концов зафиксировалась. Когда в следующей итерации T2 снова прочитает файл, она увидит эти дополнительные значения в нарушение ACID-свойства изоляции. Эти дополнительные значения называются *фантомами*, потому что для T2 их появление выглядит мистическим.

Может ли диспетчер конкуренции предотвратить этот конфликт? Протокол блокирования требует, чтобы T2 получала блокировку для каждого блока, который она читает, поэтому T1 не сможет записывать новые значения в эти блоки. Однако в данном случае этот подход не работает, поскольку требует, чтобы T2 получила блокировки на новые блоки *еще до того*, как T1 создаст их!

Решение состоит в том, чтобы разрешить транзакциям блокировать маркер конца файла. В частности, транзакция должна получить монопольную блокировку перед вызовом метода `append` и разделяемую блокировку перед вызовом метода `size`. В сценарии, приведенном выше, если T1 вызовет `append` первой, тогда T2 не сможет определить размер файла до завершения T1; и наоборот, если T2 уже определила размер файла, тогда T1 будет приостановлена при попытке добавить новые блоки до фиксации T2. В любом случае фантомы больше возникать не будут.

5.4.6. Многоверсионное блокирование

Большинство транзакций в приложениях баз данных только читают данные. Такие транзакции прекрасно уживаются друг с другом в движке базы данных, потому что используют разделяемые блокировки и не должны ждать друг друга. Однако они плохо ладят с транзакциями, изменяющими данные. Представьте, что одна изменяющая транзакция записывает данные в блок. Тогда все читающие транзакции, которым нужно прочитать этот блок, будут вынуждены ждать не только окончания записи в блок, но и фиксации пишущей транзакции. И наоборот, если пишущей транзакции потребуется записать какие-то данные в блок, ей придется подождать завершения всех транзакций, читающих этот же блок.

Другими словами, при конфликте пишущих и читающих транзакций очень много времени тратится на ожидание, независимо от того, какая из транзакций получит блокировку первой. Учитывая распространенность этой ситуации, исследователи разработали стратегии сокращения потерь времени на ожидание. Одна такая стратегия называется *многоверсионным блокированием*.

5.4.6.1. Принцип работы многоверсионного блокирования

Как следует из названия, многоверсионное блокирование сохраняет несколько версий каждого блока. Основная идея заключается в следующем:

- каждая версия блока отмечается временем фиксации транзакции, внесшей изменения;
- когда читающая транзакция запрашивает значение из блока, диспетчер конкуренции использует версию блока, зафиксированную последней на момент начала этой транзакции.

Иначе говоря, читающая транзакция видит моментальный снимок зафиксированных данных, как они выглядели в момент ее начала. Обратите внимание на слова «зафиксированных данных». Транзакция видит только данные, записанные другими транзакциями, которые были зафиксированы до ее начала, и не видит данных, записанных более поздними транзакциями.

Рассмотрим следующий пример использования многоверсионного блокирования. Предположим, что имеются четыре транзакции со следующей историей:

```
T1: W(b1); W(b2)
T2: W(b1); W(b2)
T3: R(b1); R(b2)
T4: W(b2)
```

– и все они выполняются согласно следующему расписанию:

```
W1(b1); W1(b2); C1; W2(b1); R3(b1); W4(b2); C4; R3(b2); C3; W2(b1); C2
```

Это расписание предполагает, что транзакции устанавливают блокировки непосредственно перед тем, как они потребуются. Операция Ci – это операция фиксации транзакции Ti. Пишущие транзакции – T1, T2 и T4 – следуют протоколу блокирования, в чем легко убедиться, исследовав расписание. Транзакция T3 только читает данные и не следует протоколу.

Диспетчер конкуренции сохраняет версию блока для каждой транзакции, выполняющей запись в него. То есть в данном сценарии будет создано две версии блока b1 и три версии блока b2, как показано на рис. 5.2.

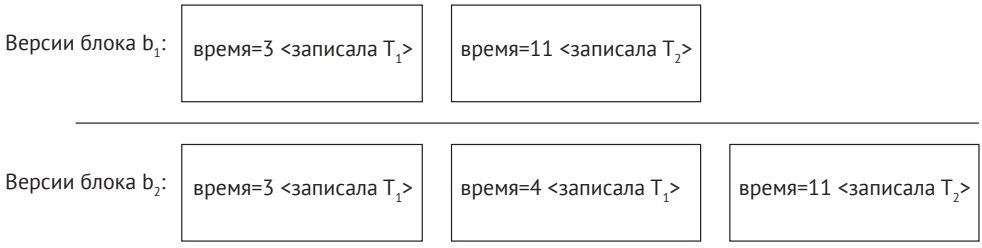


Рис. 5.2. Многоверсионный доступ

Отметка времени в каждой версии – это время фиксации транзакции, создавшей ее, а не время записи. Предположим, что каждая операция занимает одну единицу времени, поэтому T1 зафиксируется в момент времени 3, T4 – в момент времени 7, T3 – в момент времени 9 и T2 – в момент времени 11.

Теперь рассмотрим транзакцию T3, которая только читает данные. Она начинается в момент времени 5, а значит, должна увидеть значения, зафиксированные к этому моменту, а именно изменения, сделанные транзакцией T1, но не T2 или T4. То есть она увидит версии блоков b1 и b2 с отметкой времени 3. Обратите внимание, что T3 не увидит версию b2 с отметкой времени 7, даже притом что эта версия была зафиксирована к моменту, когда происходило чтение.

Самое замечательное в идее многоверсионного блокирования – читающие транзакции не должны устанавливать блокировки и, следовательно, не должны тратить время на ожидание. Диспетчер конкуренции выберет подходящую версию запрашиваемого блока, опираясь на время начала транзакции. Пишущие транзакции могут в это же время вносить изменения в тот же блок, но это не затронет читающую транзакцию, потому что она будет видеть совсем другую версию блока.

Многоверсионное блокирование оказывает благотворное влияние только на читающие транзакции. Пишущие транзакции по-прежнему должны следовать протоколу блокирования, устанавливая разделяемые и монопольные блокировки. Это объясняется тем, что каждая пишущая транзакция читает и пишет в текущую версию данных (и никогда в предыдущую), поэтому возможны конфликты. Но имейте в виду, что эти конфликты возникают только между пишущими транзакциями – участие в них читающих транзакций исключено. То есть, учитывая, что количество конфликтующих пишущих транзакций относительно невелико, на ожидание будет тратиться намного меньше времени.

5.4.6.2. Реализация многоверсионного блокирования

Теперь, узнав, как работает многоверсионное блокирование, рассмотрим, как должен действовать диспетчер конкуренции. Основная проблема заключается в сохранении версий каждого блока. Прямолинейное, но несколько сложное решение – явно сохранять каждую версию в отдельном «файле версии». Другой способ – использовать журнал для воссоздания любой желаемой версии блока, как описывается далее.

Каждой читающей транзакции присваивается отметка времени в момент ее запуска. Каждой пишущей транзакции присваивается отметка времени в момент ее фиксации. При вызове для пишущей транзакции метод `commit` выполняет дополнительные действия:

- диспетчер восстановления добавляет отметку времени транзакции в журнальную запись;
- для каждой монопольной блокировки, удерживаемой транзакцией, диспетчер конкуренции закрепляет блок, записывает в его начало отметку времени и открепляет буфер.

Предположим, что читающая транзакция с отметкой времени t запросила блок b. В этом случае диспетчер конкуренции выполнит следующие шаги, чтобы воссоздать соответствующую версию:

- скопирует текущую версию блока b в новую страницу;
- трижды прочитает журнал в обратном направлении, как описано ниже:

- *создаст список транзакций, зафиксированных после момента времени t*. Поскольку транзакции фиксируются в порядке присвоенных им отметок времени, диспетчер конкуренции может прекратить чтение журнала, обнаружив запись фиксации с отметкой времени меньше t;
- *создаст список незавершенных транзакций*, отыскав в журнале записи, созданные транзакциями, для которых отсутствуют записи фиксации или отката. Чтение журнала можно прекратить, когда обнаружится запись блокирующей контрольной точки или самая ранняя запись начала для транзакций, присутствующих в записи неблокирующей контрольной точки;
- *использует записи обновления для отмены изменений в копии b*. Встретив запись обновления для блока b, созданную пишущей транзакцией в любом из списков, упомянутых выше, диспетчер выполняет отмену. Чтение журнала можно прекратить, когда обнаружится запись начала для самой ранней транзакции в списках;

❍ вернет восстановленную копию блока b транзакции.

Другими словами, диспетчер конкуренции воссоздает версию блока на момент времени t, отменяя изменения, которые были сделаны транзакциями, не зафиксированными до этого момента. Три прохода в этом алгоритме использованы только для простоты объяснения. В упражнении 5.38 вам будет предложено переписать алгоритм, чтобы он выполнял только один проход.

Наконец, каждая транзакция должна явно указать, будет ли она только читать данные или нет, потому что диспетчер конкуренции по-разному обрабатывает эти два типа транзакций. В JDBC это требование выполняется методом `setReadOnly` в интерфейсе `Connection`. Например:

```
Connection conn = ... // получить соединение
conn.setReadOnly(true);
```

Вызов `setReadOnly` считается «подсказкой» для системы баз данных. Движок может игнорировать этот вызов, если многоверсионное блокирование не поддерживается.

5.4.7. Уровни изоляции транзакций

Обеспечение сериализуемости вызывает значительные потери времени на ожидание, потому что протокол блокирования требует, чтобы транзакции удерживали свои блокировки до завершения. То есть если транзакции T1 требуется только одна блокировка, которая конфликтует с блокировкой, удерживаемой транзакцией T2, тогда T1 не сможет продолжить работу, пока T2 не завершится.

Многоверсионное блокирование очень привлекательно в этом отношении, потому что позволяет выполнять читающие транзакции без установки блокировок и, следовательно, без необходимости ждать. Однако реализация многоверсионного блокирования сложна и требует дополнительных обращений к диску для воссоздания нужных версий. Кроме того, многоверсионное блокирование не оказывает положительного влияния на транзакции, обновляющие базу данных.

Однако есть еще один способ сократить время ожидания блокировок – транзакция может указать, что ей не требуется полная сериализуемость. В главе 2 мы

рассмотрели четыре *уровня изоляции транзакций* JDBC. Краткие характеристики этих уровней приводятся в табл. 5.1.

Таблица 5.1. Уровни изоляции транзакций

Уровень изоляции	Проблемы	Используемые блокировки	Комментарии
serializable (упорядоченное выполнение)	Нет	Разделяемые блокировки удерживаются до завершения, разделяемая блокировка для маркера конца файла	Единственный уровень, гарантирующий правильность
repeatable read (повторяемое чтение)	Фантомы	Разделяемые блокировки удерживаются до завершения, не используется разделяемая блокировка для маркера конца файла	Подходит для пишущих транзакций
read committed (чтение только подтвержденных данных)	Фантомы, значения могут изменяться	Разделяемые блокировки освобождаются раньше, не используется разделяемая блокировка для маркера конца файла	Подходит для концептуально разделяемых транзакций, изменения которых сохраняются по принципу «все или ничего»
read uncommitted (чтение неподтвержденных данных)	Фантомы, значения могут изменяться, доступны незафиксированные изменения	Разделяемые блокировки вообще не используются	Подходит для читающих транзакций, для которых допустимы неточные результаты

В главе 2 подробно описываются возможные проблемы, связанные с этими уровнями изоляции. Из нового в табл. 5.1 можно отметить связь уровней с использованием разделяемых блокировок. Уровень изоляции serializable предъявляет строгие требования к применению разделяемых блокировок, тогда как уровень изоляции read uncommitted вообще не предполагает их использования. Очевидно, что чем ниже требования к использованию блокировок, тем меньше времени тратится на ожидание. Но с ослаблением требований увеличивается вероятность получить неточный результат в ответ на запрос: транзакция может видеть фантомы, получить два разных значения из одного и того же места в разные моменты времени или увидеть значения, записанные незафиксированной транзакцией.

Хочу подчеркнуть, что эти уровни изоляции применимы только к *чтению* данных. Все пишущие транзакции, независимо от их уровня изоляции, должны действовать в соответствии с протоколом. Они должны получить все необходимые монопольные блокировки (включая монопольную блокировку для маркера конца файла) и удерживать их до своего завершения. Причина этого требования в том, что читающая транзакция вполне может решить, что некоторые неточности при выполнении запроса допустимы, но неточное изменение данных может повредить содержимое всей базы данных и поэтому недопустимо.

В чем сходство уровня изоляции read uncommitted и многоверсионного блокирования? Оба применяются только к читающим транзакциям, и в обоих случаях не используются блокировки. Однако транзакция, использующая уровень изоляции read uncommitted, видит текущее значение каждого читаемого блока, независимо от того, когда это значение было записано. Это даже близко не похоже на уровень изоляции serializable. С другой стороны, транзакция, ис-

пользующая многоверсионное блокирование, видит содержимое блоков, зафиксированное в определенный момент времени, и является сериализуемой.

5.4.8. Гранулярность элементов данных

В этой главе предполагается, что диспетчер конкуренции применяет блокировки к целым блокам. Однако возможны другие уровни гранулярности блокировок: диспетчер конкуренции может блокировать отдельные значения, файлы или даже базу данных целиком. Единица блокировки называется *элементом данных конкуренции*.

Гранулярность элементов данных не влияет на принципы управления конкурентным доступом. Все определения, протоколы и алгоритмы, представленные в этой главе, в равной степени применимы к элементу данных любого размера. То есть выбор гранулярности в значительной мере должен базироваться на компромиссах между эффективностью и гибкостью. Некоторые из этих компромиссов мы рассмотрим в данном разделе.

Диспетчер конкуренции хранит блокировки для всех элементов данных. Чем меньше размер элемента данных, тем большую степень параллелизма это обеспечивает. Например, представьте, что две транзакции одновременно изменяют разные части одного и того же блока. Эти изменения можно выполнить параллельно, если использовать блокировки на уровне отдельных значений, но нельзя с применением блокировок на уровне блока.

Однако чем мельче элементы, которые можно блокировать, тем большее количество блокировок требуется. Значения представляют слишком мелкие элементы данных, что приводит к созданию огромного количества блокировок. С другой стороны, если в качестве элемента данных использовать файлы, потребуется очень мало блокировок, но при этом существенно уменьшится степень параллелизма – клиент должен будет заблокировать весь файл, чтобы изменить его часть. Использование блоков в качестве элементов данных является разумным компромиссом.

Кроме того, обратите внимание, что для реализации примитивной формы управления одновременным доступом некоторые операционные системы (такие как MacOS и Windows) используют блокировки на уровне файлов. В частности, приложение не сможет выполнить запись в файл, не получив монопольную блокировку для файла, и не сможет получить монопольную блокировку, если этот файл в настоящее время используется другим приложением.

Некоторые диспетчеры конкуренции поддерживают несколько уровней гранулярности элементов данных, таких как блоки и файлы. Транзакция, которая планирует получить доступ лишь к нескольким блокам в файле, может заблокировать только их; но если транзакция планирует получить доступ ко всему (или большей части) файлу, она может установить одну блокировку для файла целиком. Этот подход помогает сочетать гибкость элементов данных небольшого размера с удобством использования объектов высокого уровня.

Также в роли элементов данных конкуренции можно использовать *записи данных*. Записи данных обрабатываются диспетчером записей, о котором рассказывается в следующей главе. Движок SimpleDB организован так, что диспетчер конкуренции не имеет представления о записях и поэтому не может их блокировать. Однако в некоторых коммерческих системах (таких как Oracle)

диспетчер конкуренции знает о существовании диспетчера записей и может вызывать его методы. В таких системах записи данных являются разумным выбором на роль элемента данных конкуренции.

Однако, несмотря на привлекательность гранулярности на уровне записей данных, этот выбор создает дополнительные проблемы с фантомами. В существующие блоки могут вставляться новые записи данных, поэтому транзакция, читающая все записи в блоке, должна иметь возможность предотвратить вставку записей другими транзакциями. Эту проблему можно решить с поддержкой в диспетчере конкуренции более крупных элементов данных, таких как блоки или файлы. Фактически некоторые коммерческие системы предотвращают появление фантомов, просто заставляя транзакцию получить блокировку для файла, прежде чем она выполнит вставку.

5.4.9. Диспетчер конкуренции в SimpleDB

Диспетчер конкуренции в SimpleDB реализован в виде класса `ConcurrencyMgr` в пакете `simpledb.tx.concurrency`. Этот класс реализует протокол блокирования с гранулярностью на уровне блоков. Его API показан в листинге 5.14.

Листинг 5.14. API диспетчера конкуренции в SimpleDB

ConcurrencyMgr

```
public ConcurrencyMgr(int txnum);
public void sLock(Block blk);
public void xLock(Block blk);
public void release();
```

Каждая транзакция имеет свой экземпляр диспетчера конкуренции. Методы диспетчера конкуренции аналогичны методам таблицы блокировок, но учитывают особенности конкретной транзакции. Каждый объект `ConcurrencyMgr` хранит ссылки на блокировки, удерживаемые его транзакцией. Методы `sLock` и `xLock` запрашивают блокировку из таблицы блокировок, только если транзакция еще не владеет ею. Метод `release` вызывается в конце транзакции и освобождает все ее блокировки.

Класс `ConcurrencyMgr` использует класс `LockTable`, реализующий таблицу блокировок в SimpleDB. В оставшейся части этого раздела мы рассмотрим реализации этих двух классов.

5.4.9.1. Класс LockTable

Определение класса `LockTable` показано в листинге 5.15. Объект `LockTable` содержит переменную типа `Map` с именем `locks`. Эта переменная хранит элементы для всех блоков, для которых в настоящий момент удерживаются блокировки. Значением элемента является объект типа `Integer`: число –1 соответствует монопольной блокировке, а положительное число – текущему количеству установленных разделяемых блокировок.

Методы `sLock` и `xLock` действуют подобно методу `pin` в `BufferMgr`. Они оба вызывают Java-метод `wait` внутри цикла, что равносильно помещению потока клиента в список ожидания, где он находится, пока выполняется условие цикла. Условие цикла в `sLock` вызывает метод `hasXlock`, который возвращает `true`, если для блока есть запись в `locks` со значением –1. Условие цикла в `xLock` вы-

зывает метод hasOtherLocks, который возвращает значение true, если для блока есть запись в locks со значением больше 1. Объясняется это просто: перед попыткой получить монопольную блокировку диспетчер конкуренции всегда устанавливает разделяемую блокировку для этого же блока, поэтому значение выше 1 указывает, что какая-то другая транзакция также удерживает разделяемую блокировку для этого блока.

Листинг 5.15. Реализация класса LockTable в SimpleDB

```
class LockTable {
    private static final long MAX_TIME = 10000; // 10 секунд

    private Map<Block,Integer> locks = new HashMap<Block,Integer>();

    public synchronized void sLock(Block blk) {
        try {
            long timestamp = System.currentTimeMillis();
            while (hasXlock(blk) && !waitingTooLong(timestamp))
                wait(MAX_TIME);
            if (hasXlock(blk))
                throw new LockAbortException();
            int val = getLockVal(blk); // не будет отрицательным
            locks.put(blk, val+1);
        }
        catch(InterruptedException e) {
            throw new LockAbortException();
        }
    }

    public synchronized void xLock(Block blk) {
        try {
            long timestamp = System.currentTimeMillis();
            while (hasOtherSLocks(blk) && !waitingTooLong(timestamp))
                wait(MAX_TIME);
            if (hasOtherSLocks(blk))
                throw new LockAbortException();
            locks.put(blk, -1);
        }
        catch(InterruptedException e) {
            throw new LockAbortException();
        }
    }

    public synchronized void unlock(Block blk) {
        int val = getLockVal(blk);
        if (val > 1)
            locks.put(blk, val-1);
        else {
            locks.remove(blk);
            notifyAll();
        }
    }

    private boolean hasXlock(Block blk) {
        return getLockVal(blk) < 0;
    }
```

```
    private boolean hasOtherSLocks(Block blk) {
        return getLockVal(blk) > 1;
    }

    private boolean waitingTooLong(long starttime) {
        return System.currentTimeMillis() - starttime > MAX_TIME;
    }

    private int getLockVal(Block blk) {
        Integer ival = locks.get(blk);
        return (ival == null) ? 0 : ival.intValue();
    }
}
```

Метод `unlock` либо удаляет указанную блокировку из коллекции `locks` (если это монопольная блокировка или разделяемая блокировка, установленная только одной транзакцией), либо уменьшает счетчик транзакций, владеющих разделяемой блокировкой. Если блокировка удаляется из коллекции, вызывается Java-метод `notifyAll`, который переносит все ожидающие потоки в список готовности для планирования. Внутренний планировщик потоков в Java возобновляет каждый поток в некотором неопределенном порядке. Может так получиться, что сразу несколько потоков ожидают освобождения одной и той же блокировки. К тому времени, когда поток возобновится, ожидаемая им блокировка может оказаться недоступной, в этом случае он снова поместит себя в список ожидания.

Этот код не особенно эффективно управляет уведомлением потоков. Метод `notifyAll` возобновит *все* ожидающие потоки, включая потоки, ожидающие освобождения других блокировок. Когда эти потоки возобновят работу, они (конечно же) обнаружат, что ожидаемые ими блокировки все еще недоступны, и вернутся в состояние ожидания. С одной стороны, эта стратегия обходится не слишком дорого, если одновременно работает лишь несколько конфликтующих потоков. С другой стороны, более эффективная реализация движка базы данных получилась бы более сложной. В упражнениях 5.53–5.54 вам будет предложено усовершенствовать механизм ожидания и уведомления.

5.4.9.2. Класс ConcurrencyMgr

Определение класса `ConcurrencyMgr` показано в листинге 5.16. Несмотря на то что каждая транзакция владеет своим экземпляром диспетчера конкуренции, все они должны использовать одну и ту же таблицу блокировок.

Листинг 5.16. Реализация класса ConcurrencyMgr в SimpleDB

```
public class ConcurrencyMgr {
    private static LockTable locktbl = new LockTable();
    private Map<Block,String> locks = new HashMap<Block,String>();

    public void sLock(Block blk) {
        if (locks.get(blk) == null) {
            locktbl.sLock(blk);
            locks.put(blk, "S");
        }
    }
```

```
    public void xLock(Block blk) {
        if (!hasXLock(blk)) {
            sLock(blk);
            locktbl.xLock(blk);
            locks.put(blk, "X");
        }
    }

    public void release() {
        for (Block blk : locks.keySet())
            locktbl.unlock(blk);
        locks.clear();
    }

    private boolean hasXLock(Block blk) {
        String locktype = locks.get(blk);
        return locktype != null && locktype.equals("X");
    }
}
```

Это требование реализуется путем использования в каждом объекте `ConcurrencyMgr` общей статической переменной `LockTable`. Перечень блокировок, удерживаемых транзакцией, хранится в локальной переменной `locks`. Эта переменная содержит ассоциативный массив, каждый элемент которого соответствует заблокированному блоку. Роль значений в этих элементах играют символы «S» и «X», в зависимости от типа установленной блокировки – разделяемой («S») или монопольной («X»).

Метод `sLock` сначала проверяет, владеет ли транзакция разделяемой блокировкой для заданного блока; если владеет, то нет необходимости обращаться к таблице блокировок. Иначе он вызывает метод `sLock` таблицы блокировки и ждет получения блокировки. Метод `xLock` не должен ничего делать, если транзакция уже владеет монопольной блокировкой для заданного блока. В противном случае он сначала устанавливает разделяемую блокировку для блока, а затем – монопольную. (Напомню: метод `xLock` таблицы блокировок предполагает, что транзакция уже владеет разделяемой блокировкой.) Обратите внимание, что монопольные блокировки «строже» разделяемых, в том смысле, что транзакция, владеющая монопольной блокировкой для блока, также владеет разделяемой блокировкой.

5.5. Реализация транзакций в SimpleDB

В разделе 5.2 был представлен API класса `Transaction`. Теперь мы готовы обсудить его реализацию. Класс `Transaction` использует класс `BufferList` для управления буферами, которые он закрепил. Обсудим каждый класс по очереди.

Класс Transaction

Определение класса `Transaction` показано в листинге 5.17. Каждый объект `Transaction` создает свои экземпляры диспетчеров восстановления и конкуренции. Он также создает объект `myBuffers` для управления закрепленными в данный момент буферами.

Листинг 5.17. Реализация класса Transaction в SimpleDB

```
public class Transaction {
    private static int nextTxNum = 0;
    private static final int END_OF_FILE = -1;
    private RecoveryMgr    recoveryMgr;
    private ConcurrencyMgr concurMgr;
    private BufferMgr bm;
    private FileMgr fm;
    private int txnum;
    private BufferList mybuffers;

    public Transaction(FileMgr fm, LogMgr lm, BufferMgr bm) {
        this.fm = fm;
        this.bm = bm;
        txnum = nextTxNumber();
        recoveryMgr = new RecoveryMgr(this, txnum, lm, bm);
        concurMgr = new ConcurrencyMgr();
        mybuffers = new BufferList(bm);
    }

    public void commit() {
        recoveryMgr.commit();
        concurMgr.release();
        mybuffers.unpinAll();
        System.out.println("transaction " + txnum + " committed");
    }

    public void rollback() {
        recoveryMgr.rollback();
        concurMgr.release();
        mybuffers.unpinAll();
        System.out.println("transaction " + txnum + " rolled back");
    }
    public void recover() {
        bm.flushAll(txnum);
        recoveryMgr.recover();
    }

    public void pin(BlockId blk) {
        mybuffers.pin(blk);
    }

    public void unpin(BlockId blk) {
        mybuffers.unpin(blk);
    }

    public int getInt(BlockId blk, int offset) {
        concurMgr.sLock(blk);
        Buffer buff = mybuffers.getBuffer(blk);
        return buff.contents().getInt(offset);
    }

    public String getString(BlockId blk, int offset) {
        concurMgr.sLock(blk);
        Buffer buff = mybuffers.getBuffer(blk);
        return buff.contents().getString(offset);
    }
```

```
    public void setInt(BlockId blk, int offset, int val,
                       boolean okToLog) {
        concurMgr.xLock(blk);
        Buffer buff = mybuffers.getBuffer(blk);
        int lsn = -1;
        if (okToLog)
            lsn = recoveryMgr.setInt(buff, offset, val);
        Page p = buff.contents();
        p.setInt(offset, val);
        buff.setModified(txnum, lsn);
    }

    public void setString(BlockId blk, int offset, String val,
                          boolean okToLog) {
        concurMgr.xLock(blk);
        Buffer buff = mybuffers.getBuffer(blk);
        int lsn = -1;
        if (okToLog)
            lsn = recoveryMgr.setString(buff, offset, val);
        Page p = buff.contents();
        p.setString(offset, val);
        buff.setModified(txnum, lsn);
    }

    public int size(String filename) {
        BlockId dummyblk = new BlockId(filename, END_OF_FILE);
        concurMgr.sLock(dummyblk);
        return fm.length(filename);
    }

    public BlockId append(String filename) {
        BlockId dummyblk = new BlockId(filename, END_OF_FILE);
        concurMgr.xLock(dummyblk);
        return fm.append(filename);
    }

    public int blockSize() {
        return fm.blockSize();
    }

    public int availableBuffs() {
        return bm.available();
    }

    private static synchronized int nextTxNumber() {
        nextTxNum++;
        System.out.println("new transaction: " + nextTxNum);
        return nextTxNum;
    }
}
```

Методы `commit` и `rollback` выполняют следующие действия:

- открепляют все буферы, использованные транзакцией;
- вызывают диспетчера восстановления для фиксации (или отката) транзакции;
- вызывают диспетчера конкуренции для освобождения блокировок.

Методы getInt и getString сначала устанавливают разделяемую блокировку для указанного блока, обращаясь к диспетчеру конкуренции, а затем возвращают запрошенное значение из буфера. Методы setInt и setString сначала устанавливают монопольную блокировку, обращаясь к диспетчеру конкуренции, а затем вызывают соответствующий метод диспетчера восстановления, чтобы создать журнальную запись и вернуть ее номер LSN. Этот номер LSN затем передается методу setModified буфера.

Методы size и append обрабатывают маркер конца файла как «фиктивный» блок с номером блока –1. Метод size устанавливает разделяемую блокировку для этого блока, а append – монопольную.

Класс BufferList

Класс BufferList управляет списком буферов, закрепленных транзакцией (листинг 5.18). Объект BufferList должен знать, какой буфер связан с указанным блоком и сколько раз каждый блок был закреплен. Класс использует ассоциативный массив для определения связей между буферами и блоками и список, в котором каждый объект BlockId присутствует столько раз, сколько он был закреплен; каждый раз, когда блок открепляется, из списка удаляется элемент с соответствующим ему экземпляром BlockId.

Листинг 5.18. Реализация класса BufferList в SimpleDB

```
class BufferList {
    private Map<BlockId,Buffer> buffers = new HashMap<>();
    private List<BlockId> pins = new ArrayList<>();
    private BufferMgr bm;
    public BufferList(BufferMgr bm) {
        this.bm = bm;
    }

    Buffer getBuffer(BlockId blk) {
        return buffers.get(blk);
    }

    void pin(BlockId blk) {
        Buffer buff = bm.pin(blk);
        buffers.put(blk, buff);
        pins.add(blk);
    }

    void unpin(BlockId blk) {
        Buffer buff = buffers.get(blk);
        bm.unpin(buff);
        pins.remove(blk);
        if (!pins.contains(blk))
            buffers.remove(blk);
    }

    void unpinAll() {
        for (BlockId blk : pins) {
            Buffer buff = buffers.get(blk);
            bm.unpin(buff);
        }
        buffers.clear();
        pins.clear();
    }
}
```

Метод `unpinAll` выполняет все необходимые действия с буферами, когда транзакция фиксируется или откатывается, – он обращается к диспетчеру буферов, чтобы сбросить на диск все буферы, измененные транзакцией, и открепляет все закрепленные буферы.

5.6. Итоги

- Данные можно потерять или повредить, если клиентские программы будут работать без оглядки друг на друга. Движки баз данных вынуждают клиентские программы выполнять *транзакции*.
- Транзакция – это группа операций, которая действует подобно одной операции. Она удовлетворяет ACID-свойствам: *атомарности* (atomicity), *согласованности* (consistency), *изоляции* (isolation) и *долговечности* (durability).
- *Диспетчер восстановления* отвечает за атомарность и долговечность. Этот компонент движка читает и обрабатывает журнал. Он выполняет три функции: сохраняет записи в журнал, выполняет откат транзакций и восстанавливает базу данных после сбоя системы.
- Каждая транзакция записывает в журнал *начальную запись*, чтобы обозначить момент своего начала, *записи обновления*, чтобы сообщить, какие изменения она выполнила, и *запись фиксации* или *отката*, чтобы обозначить момент своего завершения. Кроме того, в разные моменты времени диспетчер восстановления может выводить в журнал *записи контрольных точек*.
- Диспетчер восстановления *откатывает* транзакцию, читая журнал в обратном направлении. Он использует записи обновления транзакции, чтобы отменить изменения.
- После сбоя системы диспетчер восстановления *восстанавливает* базу данных.
- *Алгоритм восстановления с отменой и повторным выполнением* отменяет изменения, сделанные незафиксированными транзакциями, и повторно выполняет изменения, сделанные зафиксированными транзакциями.
- *Алгоритм восстановления только с отменой* предполагает, что изменения, сделанные зафиксированной транзакцией, записываются на диск перед фиксацией транзакции. Поэтому он только отменяет изменения, сделанные незафиксированными транзакциями.
- *Алгоритм восстановления только с повторным выполнением* предполагает, что измененные буферы не сбрасываются на диск, пока транзакция не будет зафиксирована. Этот алгоритм требует, чтобы транзакция удерживала измененные буферы в памяти до завершения, но он избавляет от необходимости отмены незафиксированных транзакций.
- Стратегия *журналирования с опережением* требует, чтобы записи обновления принудительно сбрасывались на диск до записи измененной страницы с данными. Журналирование с опережением гарантирует присутствие в журнале всех изменений, произведенных в базе данных, что обеспечит возможность их отмены.

- Записи контрольных точек позволяют уменьшить часть журнала, которую должен прочитать алгоритм восстановления. В моменты, когда нет ни одной активной транзакции, можно сохранить *запись блокирующей контрольной точки*; *запись неблокирующей контрольной точки* можно сохранить в любой момент. Если используется алгоритм восстановления с отменой и повторным выполнением (или только с повторным выполнением), то перед сохранением записи контрольной точки диспетчер восстановления должен сбросить на диск измененные буферы.
- Диспетчер восстановления может выбирать уровень детализации для сохранения значений в журнал: записи, страницы, файлы и т. д. Единица ведения журнала называется *элементом данных восстановления*. Выбор элемента данных предполагает компромисс: чем больше элемент данных, тем меньше будет записей обновления в журнале, но каждая запись будет больше.
- *Диспетчер конкуренции* – это компонент движка базы данных, отвечающий за правильное выполнение конкурирующих транзакций.
- Последовательность операций, выполняемых транзакциями в движке, называется *расписанием*. Расписание является *сериализуемым*, если оно эквивалентно последовательному расписанию. Только сериализуемые расписания являются правильными.
- Для гарантий сериализуемости расписаний диспетчер конкуренции использует блокировки. В частности, он требует, чтобы все транзакции следовали *протоколу блокирования*, который обязывает:
  - устанавливать разделяемую блокировку для блока перед его чтением;
  - устанавливать монопольную блокировку для блока перед его изменением;
  - освобождать все блокировки после фиксации или отката.
- Если образуется замкнутый цикл транзакций, в котором каждая транзакция ожидает освобождения блокировки, удерживаемой следующей транзакцией, может возникнуть состояние *взаимоблокировки*. Диспетчер конкуренции способен обнаруживать такие состояния, поддерживая граф *ожидания* и проверяя наличие циклов в нем.
- Диспетчер конкуренции также может использовать специальные алгоритмы для определения вероятных состояний взаимоблокировки. Алгоритм *ожидания-отмены* принудительно откатывает транзакцию, если она ждет освобождения блокировки, удерживаемой более старой транзакцией. Алгоритм *ограничения по времени* откатывает транзакцию, если она ждет дольше установленного предела. Оба алгоритма устранят взаимоблокировку, когда она возникает, но также могут откатить транзакцию, когда в этом нет реальной необходимости.
- Пока одна транзакция исследует содержимое файла, другая может добавить в него новые блоки. Значения в этих блоках называются *фантомами*. Фантомы нежелательны, потому что нарушают сериализуемость. Транзакция может предотвратить появление фантомов, заблокировав маркер конца файла.

- Блокирование, необходимое для обеспечения сериализуемости, значительно уменьшает возможности для одновременного выполнения. Стратегия *многоверсионного блокирования* позволяет выполнять читающие транзакции без блокировок (и, соответственно, без необходимости ждать их освобождения). Диспетчер конкуренции реализует многоверсионное блокирование, присваивая каждой транзакции отметку времени и используя эти отметки для воссоздания версий блоков, какими они были в определенные моменты времени.
- Еще один способ сократить время ожидания блокировок – устранить требование сериализуемости. Транзакция может выбрать один из четырех *уровней изоляции*: *serializable* (упорядоченное выполнение), *repeatable read* (повторяемое чтение), *read committed* (чтение только подтвержденных данных) или *read uncommitted* (чтение неподтвержденных данных). Все уровни изоляции, кроме serializable, ослабляют ограничения для разделяемых блокировок, определяемые протоколом блокирования, и способствуют уменьшению времени ожидания, но также увеличивают серьезность проблем, которые могут возникнуть при чтении. Разработчики, выбирающие уровни изоляции, отличные от serializable, должны оценить возможность получения неточных результатов и приемлемость этих неточностей.
- Диспетчер конкуренции, так же как диспетчер восстановления, может позволять устанавливать блокировки для значений, записей, страниц, файлов и т. д. Единица блокировки называется *элементом данных конкуренции*. Выбор элемента данных предполагает компромисс: чем больше элемент данных, тем меньше потребуется блокировок, но вероятность конфликтов с более крупными блокировками намного выше, а значит, возможностей для одновременного выполнения будет меньше.

5.7. Для дополнительного чтения

Транзакция – фундаментальное понятие во многих областях распределенных вычислений, не только в системах баз данных. Исследователи разработали обширный набор методов и алгоритмов, и в этой главе мы показали лишь верхушку гигантского айсберга. Прекрасный обзор этой темы можно найти в книгах: Bernstein and Newcomer (1997) и Grey and Reuter (1993). Подробное исследование многих алгоритмов управления конкуренцией и восстановлением можно найти в книге Bernstein et al. (1987). Один из широко распространенных алгоритмов восстановления называется ARIES и описывается в Mohan et al. (1992).

Реализация уровня изоляции serializable, разработанная в Oracle, называется *snapshot isolation* (изоляция мгновенных снимков), которая добавляет к многоверсионному управлению конкурентным доступом поддержку изменений. Подробности можно найти в главе 9 книги Ashdown et al. (2019). Обратите внимание, что в Oracle этот уровень изоляции называется serializable, хотя в действительности имеет некоторые отличия. Изоляция мгновенных снимков эффективнее протокола блокирования, но не гарантирует сериализуемость. Большинство расписаний будут сериализуемыми, но в некоторых

сценариях может проявляться несериализуемое поведение. В статье Fekete et al. (2005) автор анализирует эти сценарии и показывает, как модифицировать приложения, чтобы гарантировать сериализуемость.

В Oracle алгоритм восстановления с отменой и повторным выполнением реализован немного иначе: он отделяет информацию, используемую для отмены (то есть старые, перезаписанные значения), от информации для повторного выполнения (новые значения). Информация для повторного выполнения хранится в журнале (redo log), который управляется, как описано в этой главе. Однако информация для отмены сохраняется не в файле журнала, а в специальных *буферах отката* (undo buffers). Причина такого решения в том, что Oracle использует старые, перезаписанные значения не только для восстановления, но и для многоверсионного управления конкурентным доступом. Подробности можно найти в главе 9 книги Ashdown et al. (2019).

Часто транзакции полезно рассматривать как состоящие из нескольких меньших скоординированных транзакций. Например, при наличии поддержки *вложенных транзакций* родительская транзакция может запустить одну или несколько дочерних транзакций; когда дочерняя транзакция завершается, родительская решает, что делать дальше. Если дочерняя транзакция прерывается, родительская может прервать все остальные дочерние транзакции или запустить другую транзакцию взамен прерванной. Описание основ вложенных транзакций можно найти в Moss (1985). В статье Weikum (1991) дается определение *многоуровневых транзакций* (multilevel transactions), похожих на вложенные транзакции, с той разницей, что многоуровневая транзакция использует дочерние транзакции для повышения эффективности за счет параллельного выполнения.

Ashdown, L., et al. (2019). «Oracle database concepts». Document E96138-01, Oracle Corporation. Доступна по адресу: https://docs.oracle.com/en/database/oracle/oracle-database/19/cncpt/database-concepts.pdf.

Bernstein, P., Hadzilacos, V., & Goodman, N. (1987). «Concurrency control and recovery in database systems». Reading, MA: Addison-Wesley.

Bernstein, P., & Newcomer, E. (1997). «Principles of transaction processing». San Mateo: Morgan Kaufman.

Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P., & Shasha, D. (2005). «Making snapshot isolation serializable». ACM Transactions on Database Systems, 30 (2), 492–528.

Gray, J., & Reuter, A. (1993). «Transaction processing: concepts and techniques». San Mateo: Morgan Kaufman.

Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., & Schwartz, P. (1992). «ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging». ACM Transactions on Database Systems, 17 (1), 94–162.

Moss, J. (1985). «Nested transactions: An approach to reliable distributed computing». Cambridge, MA: MIT Press.

Weikum, G. (1991). «Principles and realization strategies of multilevel transaction management». ACM Transactions on Database Systems, 16 (1), 132–180.

5.8. Упражнения

Теория

5.1. Предположим, что код в листинге 5.1 одновременно выполняется двумя пользователями, но без транзакций. Опишите ситуацию, когда будет забронировано два места, но оплачено только одно.

5.2. Системы управления версиями программного обеспечения, такие как Git и Subversion, позволяют зафиксировать серию изменений в файле и затем откатить файл до предыдущего состояния. Они также позволяют нескольким пользователям изменять файлы одновременно.

a) Что понимается под транзакцией в таких системах?
b) Как такие системы обеспечивают сериализуемость?
c) Можно ли такой подход использовать в системах баз данных? Объясните, почему.

5.3. Представьте программу JDBC, которая выполняет несколько независимых друг от друга запросов SQL, но не изменяет базу данных. Программист решает, что идея транзакций в этом случае неважна, потому что программа ничего не изменяет; и реализует ее как одну большую транзакцию.

a) Объясните, почему идея транзакций *сохраняет* важность для программ, выполняющих только чтение.
b) Какие проблемы могут возникнуть в программе, которая выполняется в одной большой транзакции?
c) Какие накладные расходы вызывает фиксация читающей транзакции? Есть ли смысл выполнять фиксацию после каждого SQL-запроса?

5.4. С началом каждой транзакции диспетчер восстановления сохраняет в журнал начальную запись.

a) Какую практическую пользу дает присутствие начальных записей в журнале?
b) Представьте систему баз данных, которая не сохраняет начальные записи в журнале. Сможет ли диспетчер восстановления работать нормально? На какие возможности повлияет отсутствие начальных записей?

5.5. Метод rollback в SimpleDB сразу же сохраняет запись отката на диск. Действительно ли это необходимо? Это хорошая идея?

5.6. Представьте, что диспетчер восстановления не сохраняет запись отката в журнал после его выполнения. Породит ли это какие-либо проблемы? Это хорошая идея?

5.7. Исследуйте алгоритм фиксации только с отменой, показанный в разделе 5.3.4.1. Объясните, почему нельзя поменять местами шаги 1 и 2 алгоритма.

5.8. Покажите, что если система потерпит сбой во время отката или восстановления, то повторный откат (или восстановление) по-прежнему будет выполнен правильно.

5.9. Необходимо ли регистрировать в журнале изменения, внесенные в базу данных во время отката или восстановления? Объясните, почему.

5.10. Одна из разновидностей алгоритма неблокирующей контрольной точки предполагает упоминание только одной транзакции в журнальной записи контрольной точки – самой старой из активных на тот момент.
   a) Объясните, как в этом случае будет работать алгоритм восстановления.
   b) Сравните эту стратегию со стратегией, приведенной в книге. Какую из них проще реализовать? Какая будет действовать более эффективно?

5.12. Что должен делать метод `rollback`, встретив в журнале запись блокирующей контрольной точки? А встретив запись неблокирующей контрольной точки? Объясните.

5.13. Алгоритм неблокирующей контрольной точки не позволяет запускать новые транзакции, пока запись контрольной точки не будет сохранена в журнале. Объясните, почему это ограничение важно.

5.14. Другой способ создать неблокирующую контрольную точку – сохранить в журнал две записи. Первая запись содержит только `<BEGIN_NQCKPT>` и ничего больше. Вторая – стандартная запись `<NQCKPT ...>` со списком активных транзакций. Первая запись сохраняется, как только диспетчер восстановления решит создать контрольную точку. Вторая сохраняется позже, после создания списка активных транзакций.
   a) Объясните, почему эта стратегия решает проблему, описанную в упражнении 5.12.
   b) Напишите алгоритм восстановления, использующий эту стратегию.

5.15. Объясните, почему диспетчер восстановления никогда не встретит больше одной записи блокирующей контрольной точки.

5.16. Приведите пример, показывающий, что диспетчер восстановления может встретить несколько записей неблокирующих контрольных точек во время восстановления. Как лучше обрабатывать вторую и последующие записи?

5.17. Объясните, почему диспетчер восстановления никогда не встретит вместе записи блокирующей и неблокирующей контрольных точек.

5.18. Исследуйте алгоритм восстановления в разделе 5.3.3. Шаг 1c не восстанавливает значение для отмененных транзакций.
   a) Объясните, почему это правильное решение.
   b) Будет ли алгоритм работать правильно, восстанавливая эти значения? Объясните.

5.19. Когда метод `rollback` должен восстановить исходное значение, он выполняет запись на страницу непосредственно, не устанавливая никаких блокировок. Может ли это вызвать несериализуемый конфликт с другой транзакцией? Объясните.

5.20. Объясните, почему невозможен алгоритм восстановления, сочетающий методы восстановления только с отменой и только с повторным выполнением. То есть почему операции нужно или только отменять, или только повторно выполнять?

5.20. Предположим, что диспетчер восстановления нашел следующие записи в файле после сбоя системы:

```
<START, 1>
<START, 2>
<SETSTRING, 2, junk, 33, 0, abc, def>
<SETSTRING, 1, junk, 44, 0, abc, xyz>
<START, 3>
<COMMIT, 2>
<SETSTRING, 3, junk, 33, 0, def, joe>
<START, 4>
<SETSTRING, 4, junk, 55, 0, abc, sue>
<NQCKPT, 1, 3, 4>
<SETSTRING, 4, junk, 55, 0, sue, max>
<START, 5>
<COMMIT, 4>
```

a) Какие изменения будут выполнены в базе данных в случае использования алгоритма отмены и повторного выполнения?
b) Какие изменения будут выполнены в базе данных в случае использования алгоритма только с отменой?
c) Есть ли возможность фиксации транзакции T1 в отсутствие записи фиксации в журнале?
d) Есть ли возможность для транзакции T1 изменить буфер, содержащий блок 23?
e) Есть ли возможность для транзакции T1 изменить блок 23 на диске?
f) Есть ли возможность для транзакции T1 оставить неизменным буфер, содержащий блок 44?

5.21. Всегда ли последовательное расписание является сериализуемым? Всегда ли сериализуемое расписание является последовательным? Объясните.

5.22. В этом упражнении вам предлагается проверить необходимость использования непоследовательных расписаний.

a) Предположим, что размер базы данных намного больше размера пула буферов. Объясните, почему система базы данных будет обрабатывать транзакции быстрее, если сможет выполнять транзакции одновременно.
b) И наоборот, объясните, почему одновременное выполнение менее важно, если база данных целиком помещается в пул буферов.

5.23. Методы `get` и `set` класса `Transaction` в SimpleDB устанавливают блокировку для указанного блока. Почему они не освобождают блокировку по завершении?

5.24. Исследуйте листинг 5.3. Составьте историю транзакций для случая, когда элементом конкуренции является файл.

5.25. Взгляните на истории двух следующих транзакций:

```
T1: W(b1); R(b2); W(b1); R(b3); W(b3); R(b4); W(b2)
T2: R(b2); R(b3); R(b1); W(b3); R(b4); W(b4)
```

a) Приведите сериализуемое непоследовательное расписание для этих транзакций.
b) Добавьте в эти истории операции установки и освобождения блокировок, удовлетворяющие протоколу блокирования.
c) Приведите непоследовательное расписание, соответствующее этим блокировкам, вызывающее состояние взаимоблокировки.

d) Покажите, что для этих транзакций, следующих протоколу блокирования, не существует непоследовательных сериализуемых расписаний, не приводящих к состоянию взаимоблокировки.

5.26. Приведите пример сериализуемого расписания, которое имеет конфликт запись–запись, не влияющий на порядок фиксации транзакций. (*Совет:* некоторые конфликтующие операции не имеют соответствующих операций чтения.)

5.27. Покажите, что когда транзакции следуют протоколу двухфазного блокирования, все расписания являются сериализуемыми.

5.28. Покажите, что циклы в графе ожидания присутствуют тогда и только тогда, когда существует тупик.

5.29. Предположим, что диспетчер транзакций создает граф ожидания для обнаружения взаимоблокировок. В разделе 5.4.4 предлагалось откатывать транзакцию, запрос которой вызвал появление цикла в графе. Однако также можно откатить самую старую транзакцию в образовавшемся цикле; самую новую; удерживающую наибольшее количество блокировок; или удерживающую наименьшее количество блокировок. Какой вариант представляется вам наиболее разумным? Объясните.

5.30. Предположим, что в SimpleDB транзакция T в данный момент владеет разделяемой блокировкой для блока и вызывает его метод `setInt`. Приведите сценарий, когда этот вызов может привести к состоянию взаимоблокировки.

5.31. Изучите класс `ConcurrencyTest` в листинге 5.13. Приведите расписание, приводящее к состоянию взаимоблокировки.

5.32. Исследуйте сценарий использования блокировок, описанный для листинга 5.13. Нарисуйте разные состояния графа ожидания в моменты установки и освобождения блокировок.

5.33. Один из вариантов протокола wait-die (ожидание–отмена) называется *wound-wait* (*отмена–ожидание*), и суть его заключается в следующем:

- если T1 имеет меньший номер (старше), чем T2, тогда выполнение T2 прерывается (то есть T1 «отменяет», или «ранит» (wounds), транзакцию T2);
- если T1 имеет больший номер (новее), чем T2, тогда T1 ждет освобождения блокировки.

Идея в том, что если более старой транзакции потребовалась блокировка, удерживаемая более новой транзакцией, она просто отменяет новую транзакцию и устанавливает блокировку.

a) Покажите, что этот протокол предотвращает взаимоблокировки.
b) Сравните относительные преимущества протоколов wait-die и wound-wait.

5.34. В протоколе *wait-die* обнаружения взаимоблокировок выполнение транзакции прерывается, если она запросила блокировку, удерживаемую более старой транзакцией. Предположим, вы изменили протокол так, чтобы транзакция прерывалась при попытке установить блокировку, удерживаемую более молодой транзакцией. Этот протокол тоже будет обнаруживать взаимоблокировки. Имеет ли он преимущества

перед оригинальным протоколом wait-die? Какой из них вы предпочли бы использовать в диспетчере транзакций? Объясните.

5.35. Объясните, почему методы `lock` и `unlock` в классе `LockTable` объявлены синхронными? Что может случиться, если убрать это объявление?

5.36. Предположим, что система баз данных использует в качестве элементов конкуренции файлы. Объясните, почему в этом случае возможно появление фантомов.

5.37. Приведите алгоритм обнаружения взаимоблокировок, который также учитывает транзакции, ожидающие буферов.

5.38. Перепишите алгоритм многоверсионного блокирования, чтобы диспетчер конкуренции выполнял только один проход в файле журнала.

5.30. Уровень изоляции read committed призван сократить время ожидания транзакции за счет раннего освобождения разделяемых блокировок. На первый взгляд, не очевидно, почему время ожидания сократится, если транзакция будет раньше освобождать свои блокировки. Объясните преимущества раннего освобождения блокировок и приведите иллюстративные сценарии.

5.40. Метод `nextTransactionNumber` – единственный метод в классе `Transaction`, который объявлен синхронным. Объясните, почему другие методы не требуется объявлять синхронными.

5.41. Исследуйте класс `Transaction` в SimpleDB.

a) Может ли транзакция закрепить блок, не устанавливая блокировки для него?

b) Может ли транзакция установить блокировку для блока, не закрепляя его?

Практика

5.42. SimpleDB транзакция устанавливает разделяемую блокировку для блока всякий раз, когда вызывает метод `getInt` или `getString`. Однако транзакция могла бы устанавливать разделяемую блокировку при закреплении блока, при условии что вы не закрепляете блоки, если не собираетесь просматривать их содержимое.

a) Реализуйте эту стратегию.

b) Сравните преимущества этой стратегии с реализованной в SimpleDB. Какую вы предпочли бы и почему?

5.43. После выполнения процедуры восстановления журнал нужен разве только для архивных целей. Измените код SimpleDB так, чтобы после восстановления файл журнала перемещался в отдельный каталог и создавался новый пустой файл журнала.

5.44. Измените диспетчер восстановления в SimpleDB так, чтобы он отменял обновления только при необходимости.

5.45. Измените код SimpleDB так, чтобы в качестве элементов восстановления он использовал блоки. Одна из возможных стратегий: сохранять копию блока перед первым его изменением в транзакции. Копию можно сохранить в отдельном файле, а запись обновления в журнале может содержать номер блока с копией. Вам также понадобится написать методы, копирующие блоки между файлами.

5.46. Реализуйте статический метод в классе `Transaction`, создающий блокирующую контрольную точку. Определите, как будет вызываться этот метод (например, через каждые N транзакций, через каждые N секунд или вручную). Вам нужно будет изменить класс `Transaction` следующим образом:
- определить статическую переменную для хранения всех транзакций, активных в данный момент;
- переписать конструктор `Transaction` для проверки, выполняется ли в данный момент процедура контрольной точки; если выполняется, то транзакция должна быть добавлена в список ожидания до окончания процедуры.

5.47. Реализуйте создание неблокирующих контрольных точек, используя стратегию, описанную в книге.

5.48. Предположим, что транзакция добавляет в файл множество блоков, записывает в них массу значений и затем откатывается. Новые блоки будут приведены в исходное состояние, но сами они останутся в файле. Измените код SimpleDB так, чтобы он удалял эти блоки. (*Подсказка:* можно использовать тот факт, что в каждый конкретный момент добавлять блоки в файл может только одна транзакция, то есть файл можно усечь во время отката. Вам потребуется добавить в диспетчер файла возможность усечения файла.)

5.49. Записи в журнале можно использовать не только для восстановления, но и для аудита системы. Для этого запись должна хранить дату, когда произошло действие, а также IP-адрес клиента.
a) Измените код в SimpleDB для работы с журнальными записями, реализующими эту возможность.
b) Спроектируйте и реализуйте класс, методы которого решают типичные задачи аудита, такие как определение времени последнего изменения блока или получение списка действий, выполненных в конкретной транзакции или с определенного IP-адреса.

5.50. Каждый раз, когда запускается сервер, нумерация транзакций начинается с нуля. Это означает, что на протяжении существования базы данных будет создано много транзакций с одинаковыми номерами.
a) Объясните, почему эта неуникальность номеров транзакций не является существенной проблемой.
b) Измените код SimpleDB так, чтобы нумерация транзакций продолжалась непрерывно.

5.51. Перепишите код SimpleDB так, чтобы он использовал алгоритм восстановления с отменой и повторным выполнением.

5.52. Реализуйте обнаружение взаимоблокировок в SimpleDB, используя:
a) Протокол *wait-die (ожидание–отмена*), описанный в книге.
b) Протокол *wound-wait* (*отмена–ожидание*), описанный в упражнении 5.33.

5.53. Перепишите реализацию таблицы блокировок, чтобы она использовала отдельные списки ожидания для каждого блока. (Чтобы `notifyAll` затрагивал только потоки, ожидающие этой блокировки.)

5.54. Перепишите реализацию таблицы блокировок, чтобы она хранила свои явные списки ожидания и сама выбирала, какие транзакции следует уведомить, когда какая-то блокировка становится доступной. (То есть использовать Java-метод `notify` вместо `notifyAll`.)

5.55. Перепишите код диспетчера конкуренции в SimpleDB так, чтобы:

a) роль элементов конкуренции играли файлы;
b) роль элементов конкуренции играли значения. (*Внимание:* вы также должны предотвратить появление конфликтов в методах `size` и `append`.)

5.56. Напишите тестовые программы:

a) для проверки работы диспетчера восстановления (фиксация, откат и восстановление);
b) для более полного тестирования диспетчера блокировок;
c) для тестирования диспетчера транзакций.

Глава 6

Управление записями

Диспетчер транзакций может читать и сохранять значения в указанных местоположениях в дисковых блоках. Однако он не знает, какие значения находятся в блоке и где эти значения размещаются. За это отвечает диспетчер записей. Он организует файл в коллекцию записей и предлагает методы для обхода записей и размещения значений в них. В этой главе рассматриваются функциональные возможности диспетчера записей и методы, реализующие их.

6.1. Архитектура диспетчера записей

Диспетчер записей призван решать следующие вопросы:

- должна ли каждая запись целиком размещаться в одном блоке?
- принадлежат ли все записи в блоке одной таблице?
- ограничен ли размер каждого поля заранее определенным количеством байтов?
- где в записи располагается значение каждого ее поля?

В данном разделе обсуждаются все эти вопросы и сопутствующие им компромиссы.

6.1.1. Расщепление записей

Допустим, что диспетчеру записей потребовалось вставить четыре 300-байтные записи в файл с размером блока 1000 байт. Три записи умещаются в первые 900 байт блока. Но как быть с четвертой записью? На рис. 6.1 изображены два возможных варианта.

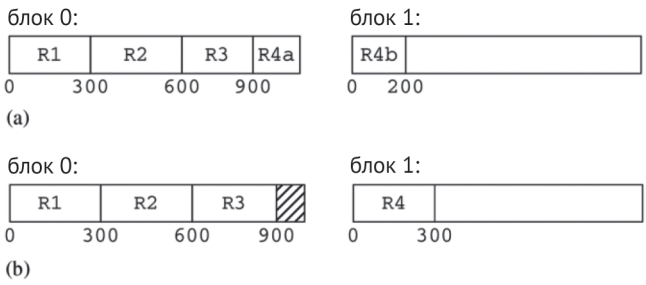


Рис. 6.1. Расщепление записей: (a) запись R4 расщеплена и размещается в двух блоках, 0 и 1; (b) запись R4 размещается целиком в блоке 1

На рис. 6.1a диспетчер записей *расщепляет запись* на два или более блоков. Он сохраняет первые 100 байт записи в текущем блоке, следующие 200 байт – в новом. На рис. 6.1b диспетчер записей сохраняет четвертую запись целиком в новом блоке.

Диспетчер записей должен решить, расщеплять записи или нет. Недостатком нерасщепляемых записей является увеличенный расход дискового пространства. На рис. 6.1b видно, что 100 байт (или 10 %) в каждом блоке тратятся впустую. Еще более тяжелый случай, когда каждая запись имеет размер 501 байт – тогда в блок можно сохранить только одну запись, и почти 50 % его пространства будут потрачены впустую. Другой недостаток – размер нерасщепляемой записи не сможет превысить размер блока. Если допустить, что записи могут иметь размер, превышающий размер блока, то без расщепления записей не обойтись.

Основной недостаток расщепляемых записей – увеличенная сложность доступа. Для того чтобы прочитать расщепленную запись, может потребоваться обратиться к нескольким блокам, в которых эта запись размещается. Кроме того, для воссоздания расщепленной записи может потребоваться скопировать ее части из этих блоков в отдельную область памяти.

6.1.2. Однородные и неоднородные файлы

Файл считается *однородным*, если все записи в нем принадлежат одной таблице. Диспетчер записей должен решить, могут ли файлы быть неоднородными. И снова решение определяется компромиссом между эффективностью и гибкостью.

Рассмотрим для примера таблицы STUDENT и DEPT, изображенные на рис. 1.1. Однородная реализация поместит все записи STUDENT в один файл, а все записи DEPT – в другой. Такое размещение позволяет легко формировать результаты в ответ на запросы SQL к одной таблице – диспетчеру записей достаточно просканировать только блоки из одного файла. Однако запросы к нескольким таблицам оказываются в этом случае менее эффективными. Рассмотрим запрос, выполняющий соединение этих двух таблиц, например: «Найти имена студентов и их основные кафедры». Диспетчер записей вынужден будет постоянно прыгать взад и вперед между блоками с записями STUDENT и DEPT (как будет обсуждаться в главе 8), пытаясь отыскать записи, соответствующие условию. И даже если запрос можно выполнить без лишних операций поиска (например, путем соединения индексов, как описывается в главе 12), дисковому накопителю все равно придется попеременно обращаться к блокам из таблиц STUDENT и DEPT.

При выборе неоднородной организации записи STUDENT и DEPT могут храниться в одном файле, причем запись о каждом студенте будет храниться рядом с записью, соответствующей его основной кафедре. На рис. 6.2 показаны первые два блока, соответствующие такой организации, где предполагается, что в одном блоке хранятся три записи. За каждой записью DEPT в файле следуют записи STUDENT, в которых эта кафедра указана как основная. Такая организация требует меньше обращений к блокам для вычисления соединения, потому что соединяемые записи *кластеризованы*, то есть находятся в том же (или в соседнем) блоке.

Рис. 6.2. Кластеризация записей при использовании неоднородных файлов

Кластеризация повышает эффективность запросов, соединяющих кластеризованные таблицы, потому что соответствующие записи хранятся вместе. Однако кластеризация может ухудшить эффективность обработки запросов к одной таблице, потому что записи, принадлежащие каждой таблице, разбросаны по большему количеству блоков. Аналогично, ухудшится эффективность операций соединения с другими таблицами. Таким образом, кластеризация эффективна, только если запросы, вычисляющие соединение, определяемое кластеризацией, используются чаще других[1].

6.1.3. Поля фиксированного и переменного размера

Каждое поле в таблице имеет определенный тип. Основываясь на этом типе, диспетчер записей решает, будет поле иметь *фиксированный* или *переменный* размер. Для хранения любых значений в поле фиксированного размера всегда используется одинаковое количество байтов, тогда как разные значения в полях переменного размера могут иметь разную длину.

Большинство типов по своей природе имеют фиксированный размер. Например, целые числа или числа с плавающей точкой можно хранить в виде 4-байтных двоичных значений. Фактически все числовые типы и типы даты и времени имеют естественные представления фиксированной длины. Тип `String` в Java, напротив, является ярким примером типа, имеющего представление переменного размера, потому что строки символов могут иметь произвольную длину.

Представления переменного размера могут вызвать серьезные осложнения. Допустим, к примеру, что вы изменили значение одного из полей в записи, находящейся в середине блока, который содержит также и другие записи. Если поле имеет фиксированную длину, размер записи останется прежним, и поле можно изменить на месте. Однако если поле имеет переменный размер, тогда размер записи может увеличиться. Чтобы освободить для нее место, диспетчеру записей, возможно, придется изменить расположение записей в блоке. А если измененная запись окажется слишком большой, то может потребоваться одну или несколько записей переместить в другой блок.

По этой причине диспетчер записей старается использовать представления фиксированной длины. Например, для строкового поля он может выбрать одно из трех представлений:

- с переменной длиной, когда диспетчер записей выделяет точное количество байтов, необходимое для сохранения строки в записи;

[1] Кластеризация является одним из фундаментальных организационных принципов ранних иерархических систем баз данных, таких как система *IMS*, созданная в компании IBM. Такие системы позволяют создавать очень эффективные реализации баз данных, иерархических по своей природе.

- с фиксированной длиной, когда диспетчер записей сохраняет строку в другом месте, за пределами записи, а в запись добавляет ссылку фиксированной длины на это место;
- с фиксированной длиной, когда для любой строки, независимо от ее длины, диспетчер записей выделяет одинаковое количество байтов.

Эти представления изображены на рис. 6.3. В верхней части (a) показаны три записи COURSE, в которых поле `Title` реализовано с использованием представления переменной длины. Эти записи занимают ровно столько места, сколько требуется, но имеют проблемы, которые только что обсуждались.

В середине (b) показаны те же три записи, но со строками для поля `Title`, хранящимися в отдельной «области для строк». Эта область может быть реализована как отдельный файл или (если строки очень большие) каталог, где каждая строка хранится в своем файле. В любом случае поле содержит только ссылку на местоположение строки в этой области. При использовании такого представления все записи имеют фиксированный и небольшой размер. Небольшие записи хороши тем, что их можно сохранить в меньшем количестве блоков, что способствует уменьшению количества обращений к диску. Однако для извлечения строкового значения из записи это представление требует дополнительного обращения к другим дисковым блокам.

В нижней части (c) показаны две записи, реализованные с использованием поля `Title` фиксированной длины. Преимущество этой реализации в том, что записи имеют фиксированную длину, а строки хранятся в самой записи. Однако некоторые записи будут иметь больший размер, чем необходимо. Если размеры фактически хранимых строк будут существенно отличаться от размера поля, потери пространства могут оказаться значительными, что приведет к увеличению размера файла и соответственно к увеличению числа обращений к блокам.

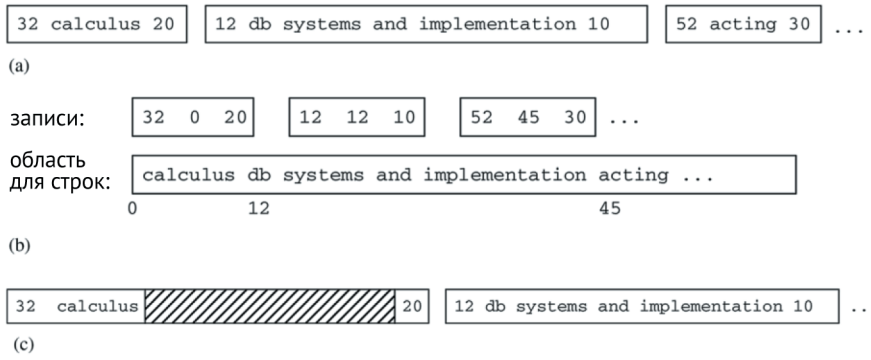


Рис. 6.3. Альтернативные представления поля Title в записях COURSE: (a) выделяется ровно столько места, сколько нужно для каждой строки; (б) строки хранятся в отдельном месте; (c) для всех строк выделяется одинаковое количество байтов

Ни одно из этих представлений не имеет явных преимуществ перед другими. Чтобы помочь диспетчеру записей выбрать наиболее подходящее представление, стандарт SQL определяет три разных строковых типа: `char`, `varchar` и `clob`. Тип `char(n)` определяет строки как последовательности из точ-

но n символов. Типы varchar(n) и clob(n) определяют строки, длина которых не превышает n символов. Разница – в ожидаемом размере n. В случае с типом varchar(n) параметр n имеет достаточно небольшую величину, скажем не более 4096. С другой стороны, в типе clob(n) значение n может достигать нескольких миллиардов символов. (Аббревиатура CLOB расшифровывается как «character large object» – «большой символьный объект».) В качестве примера поля clob представим, что в таблицу SECTION университетской базы данных добавлено поле Syllabus, которое должно хранить текст каждой учебной программы. Если предположить, что текст учебной программы не может содержать больше 8000 символов, это поле разумно определить как clob(8000).

Поля типа char наиболее естественно соответствуют ситуации на рис. 6.3c. Поскольку все строки имеют одинаковую длину, место в записях не расходуется напрасно, и представление с фиксированной длиной будет наиболее эффективным.

Поля типа varchar(n) наиболее естественно соответствуют ситуации на рис. 6.3a. Поскольку величина n будет относительно небольшой, размещение строк внутри записей не сделает последние слишком большими. Кроме того, в случаях, когда строки будут иметь разные размеры, использование представления фиксированной длины (рис. 6.3c) повлечет напрасное расходование пространства. Поэтому представление с переменной длиной является лучшей альтернативой.

Если величина параметра n окажется достаточно маленькой (скажем, меньше 20), тогда диспетчер записей может предпочесть реализовать поле varchar с использованием третьего представления (рис. 6.3c), потому что объем потраченного впустую пространства будет незначительным по сравнению с преимуществами, которые дает представление с фиксированной длиной.

Поля типа clob соответствуют ситуации на рис. 6.3b, потому что это представление лучше всего подходит для строк большого размера. При сохранении больших строк за пределами записей сами записи уменьшаются, и управлять ими становится проще.

6.1.4. Размещение полей в записях

Диспетчер записей определяет структуру своих записей. Для записей фиксированного размера он определяет местоположение каждого поля. Самая простая стратегия – хранить поля рядом друг с другом. Размер записи определяется суммой размеров полей, а каждое следующее поле начинается там, где заканчивается предыдущее.

Такая стратегия плотной упаковки полей в записях прекрасно подходит для систем, написанных на Java (таких как SimpleDB и Derby), но может вызвать проблемы в других языках. Причина связана с выравниванием значений в памяти. В большинстве компьютеров машинный код требует, чтобы целые числа размещались в памяти, начиная с адресов, кратных 4; иными словами, целое число должно быть *выровнено* по 4-байтной границе. Поэтому диспетчер записей должен гарантировать выравнивание всех целых чисел в каждой странице по 4-байтной границе. Поскольку операционная система выравнивает страницы памяти по 2^N-байтной границе для некоторого достаточно большого N, первый байт каждой страницы будет выровнен правильно. Таким образом,

диспетчер записей должен просто убедиться, что смещение каждого целого числа в каждой странице кратно 4. Если предыдущее поле закончилось в ячейке памяти с адресом, не кратным 4, то диспетчер записей должен дополнить его достаточным количеством байтов.

Например, рассмотрим таблицу STUDENT, записи которой состоят из трех целочисленных полей и строкового поля `varchar(10)`. Размеры целочисленных полей кратны 4, поэтому их не нужно дополнять. Для хранения строкового поля, однако, требуется 14 байт (если использовать представление SimpleDB, описанное в разделе 3.5.2); поэтому необходимо отступить на два байта, чтобы поле, следующее за ним, было выровнено по адресу, кратному 4.

В целом для разных типов может потребоваться разное количество дополняющих байтов. Например, числа с плавающей точкой двойной точности обычно выравниваются по 8-байтной границе, а короткие целые числа – по 2-байтной. Диспетчер записей несет ответственность за обеспечение этих выравниваний. Самая простая стратегия – расположить поля в порядке их объявления и дополнить каждое из них так, чтобы обеспечить правильное выравнивание следующего поля. Более разумная стратегия – переупорядочить поля, чтобы минимизировать количество отступов. Например, рассмотрим следующее объявление таблицы на языке SQL:

```
create table T (A smallint, B double precision, C smallint, D int, E int)
```

Предположим, что поля хранятся в указанном порядке. Тогда поле A придется дополнить 6 байтами, а поле C – 2 байтами, в результате чего размер записи составит 28 байт (см. рис. 6.4a). С другой стороны, если поля сохранить в порядке [B, D, A, C, E], дополнение вообще не потребуется и размер записи составит всего 20 байт, как показано на рис. 6.4b.

Помимо дополнения полей, диспетчер записей также должен дополнить каждую запись. Идея состоит в том, что каждая запись должна заканчиваться на границе k байт, где k – наибольшее поддерживаемое выравнивание. В этом случае каждая следующая запись на странице будет иметь такое же выравнивание, как и первая. Рассмотрим еще раз размещение полей на рис. 6.4a, где длина записи составляет 28 байт. Предположим, что первая запись начинается с 0-го байта в блоке. Тогда вторая запись начнется с 28-го байта, то есть поле B во второй записи начнется с 36-го байта, что является неправильным выравниванием. В данном примере важно, чтобы все записи начиналась с 8-байтной границы. В обоих случаях, изображенных на рис. 6.4, записи необходимо дополнить 4 байтами.

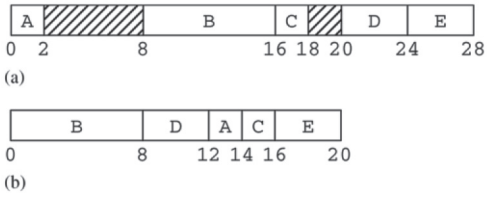


Рис. 6.4. Размещение полей в записи с учетом выравнивания: (а) размещение, требующее добавления дополняющих байтов; (б) размещение, не требующее дополняющих байтов

В программах на Java не требуется учитывать выравнивание, потому что они не могут напрямую обращаться к числовым значениям в байтовом массиве. Например, рассмотрим Java-метод `ByteBuffer.getInt` для чтения целого числа из страницы. Этот метод не вызывает машинных инструкций для получения целого числа, а просто создает это число из 4 указанных байтов в массиве. Данный метод действует не так эффективно, как машинная инструкция, зато позволяет избежать проблем с выравниванием.

6.2. Реализация файла с записями

В предыдущем разделе рассматривались различные решения, которые должен учитывать диспетчер записей. В этом разделе рассматриваются способы реализации этих решений. Он начинается с самой простой реализации: однородного файла, содержащего нерасщепляемые записи фиксированного размера. А затем описывается, как другие проектные решения влияют на эту реализацию.

6.2.1. Простая реализация

Предположим, вам нужно создать однородный файл с нерасщепляемыми записями фиксированной длины. Тот факт, что записи не расщепляются, означает, что файл можно рассматривать как последовательность блоков, каждый из которых хранит свои записи целиком. Тот факт, что файл является однородным и записи имеют фиксированную длину, означает, что для каждой записи в блоке можно выделить одинаковое количество байтов. Другими словами, каждый блок можно представить как *массив записей*. В SimpleDB такой блок называется *страницей записей*.

Для реализации страницы записей диспетчер записей делит блок на *слоты* достаточно большого размера, чтобы в каждый уместить запись, плюс один дополнительный байт. Этот байт будет служить флагом заполненности; пусть 0 обозначает пустой слот, а 1 – занятый[1].

Например, предположим, что размер блока равен 400, а размер записи – 26 байт; тогда каждый слот будет иметь размер, равный 27 байт, и в блок уместится 14 слотов с потерей 22 байт. Рисунок 6.5 изображает эту ситуацию. На этом рисунке показаны 4 из 14 слотов; в данный момент слоты 0 и 13 содержат записи, а слоты 1 и 2 – пустые.

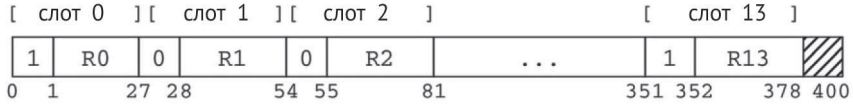


Рис. 6.5. Страница записей с местом для хранения 14 записей размером 26 байт

[1] Вы можете оптимизировать использование пространства в слоте, используя для хранения флага заполненности только бит. См. упражнение 6.7.

Диспетчер записей должен иметь возможность вставлять, удалять и изменять записи в странице записей. Для этого он использует следующую информацию о записях:

- размер слота;
- имя, тип, длину и смещение каждого поля в записи.

Эта информация составляет *компоновку* записи. Для примера рассмотрим таблицу STUDENT, как определено в листинге 2.4. Запись STUDENT содержит три целых числа и поле `varchar` размером в 10 символов. Допустим, что в данном случае используется стратегия хранения, принятая в SimpleDB, согласно которой для каждого целого числа требуется 4 байта, а для строки из десяти символов – 14 байт. Предположим также, что дополнение для выравнивания не требуется, поля `varchar` реализуются путем выделения фиксированного пространства для максимально возможной строки и флаг заполненности занимает один байт в начале каждого слота. В табл. 6.1 показана компоновка для такой таблицы.

Таблица 6.1. Компоновка записи STUDENT

Размер слота: 27
Информация о полях:

Имя	Тип	Длина	Смещение
SId	int	4	1
SName	varchar(10)	14	5
GradYear	int	4	19
MajorId	int	4	23

При использовании такой компоновки диспетчер записей может определить местоположение каждого значения на странице. Запись в слоте `k` начинается в местоположении `RL×k`, где `RL` – длина записи. Флаг заполненности для этой записи находится в местоположении `RL×k`, а значение поля `F` – в местоположении `RL×k + Смещение(F)`.

Диспетчер записей может легко обрабатывать операции вставки, удаления, изменения и извлечения:

- чтобы вставить новую запись, диспетчер должен последовательно проверить флаг заполненности в каждом слоте, пока не найдет слот с флагом 0. Затем он должен записать во флаг значение 1 и вернуть местоположение этого слота. Если все флаги имеют значение 1, значит, блок заполнен и вставка в него невозможна;
- чтобы удалить запись, диспетчеру достаточно установить флаг заполненности в значение 0;
- чтобы изменить значение поля в записи (или инициализировать поле в новой записи), диспетчер должен определить местоположение поля и записать туда требуемое значение;
- чтобы извлечь записи из страницы, диспетчер должен проверить флаг заполненности в каждом слоте, и каждый раз, обнаружив значение 1, он будет знать, что этот слот содержит запись.

Диспетчеру записей также необходима возможность идентифицировать записи в странице. Самый простой способ идентификации записей с фиксированной длиной – это номер слота.

6.2.2. Реализация полей переменного размера

Реализация полей фиксированного размера очень проста. В этом разделе рассказывается, как на нее влияет введение полей переменного размера.

Одна из проблем заключается в том, что смещения полей в записи перестают быть фиксированными. В частности, смещения всех полей, следующих за полем переменного размера, могут отличаться в разных записях. Единственный способ определить смещение этих полей – прочитать предыдущее поле и узнать, где оно заканчивается. Если первое поле в записи имеет переменный размер, тогда придется прочитать первые `n-1` полей, чтобы определить смещение `n`-го поля. По этой причине диспетчер записей обычно помещает поля фиксированного размера в начале каждой записи, чтобы упростить доступ к ним с использованием предварительно вычисленных смещений. Поля переменного размера помещаются в конец записи. Первое поле переменного размера в такой записи будет иметь фиксированное смещение, но остальные – нет.

Другая проблема обусловлена тем, что изменение значения поля может привести к изменению длины записи. Если новое значение больше прежнего, содержимое блока справа от измененного значения придется сместить, чтобы освободить дополнительное место. Иногда сдвинутые из-за этого поля записи будут оказываться вне блока; эта ситуация должна обрабатываться путем выделения *блока переполнения*.

Блок переполнения – это новый блок, выделенный в области, известной как *область переполнения*. Любая запись, вышедшая за границы исходного блока, удаляется из этого блока и добавляется в блок переполнения. Если происходит множество таких изменений в данных, то может потребоваться создать цепочку из нескольких блоков переполнения. Каждый такой блок будет содержать ссылку на следующий блок переполнения в цепочке. Концептуально исходные блоки и блоки переполнения образуют одну (большую) страницу записей.

Рассмотрим, например, таблицу COURSE и представим, что названия курсов сохраняются в виде строк переменной длины. На рис. 6.6a изображен блок, содержащий первые три записи из этой таблицы. (Поле `Title` перемещено в конец записи, потому что остальные поля имеют фиксированный размер.) На рис. 6.6b показан результат изменения заголовка «DbSys» на «Database Systems Implementation». Если предположить, что блок имеет размер 80 байт, третья запись не поместится в блок и будет перенесена в блок переполнения. После этого исходный блок будет содержать ссылку на блок переполнения.

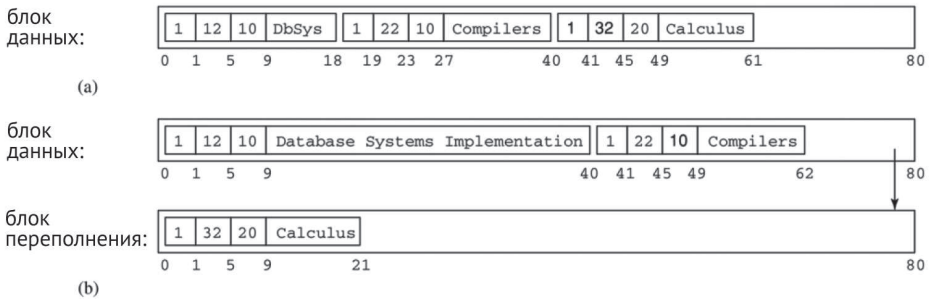


Рис. 6.6. Реализация записей переменной длины с использованием блока переполнения: (a) исходный блок; (b) результат изменения названия курса 12

Третья проблема касается использования номера слота для идентификации записи. Теперь нет возможности умножить номер слота на его размер, как с записями фиксированной длины. Единственный способ найти начало записи с заданным идентификатором – прочитать все записи, начиная с первой в блоке.

Использование номера слота в качестве идентификатора записи также усложняет вставку записи. Эту проблему иллюстрирует рис. 6.7.

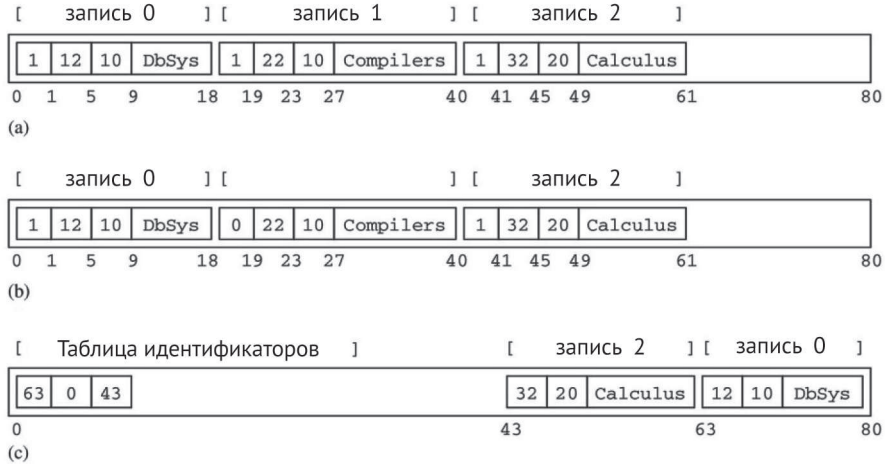


Рис. 6.7. Реализация записей переменной длины с использованием таблицы идентификаторов: (a) исходный блок; (b) прямолинейный способ удаления записи 1; (c) удаление записи 1 с использованием таблицы идентификаторов

На рис. 6.7a изображен блок, содержащий первые три записи COURSE, так же, как на рис. 6.6a. Операция удаления записи для курса 22 установит флаг в значение 0 («пустой») и оставит саму запись нетронутой, как показано на рис. 6.7b. После этого пространство будет доступно для вставки новой записи. Однако вставить в освободившееся место можно только запись, поле `Title` которой содержит не более девяти символов. Новая запись может не помещаться в блок, даже если в нем есть пустые места, оставленные меньшими удаленными записями. Такой блок называется *фрагментированным*.

Уменьшить эту фрагментацию можно смещением оставшихся записей так, чтобы они все оказались сгруппированными на одном конце блока. Однако при этом изменятся номера слотов смещенных записей, что, к сожалению, изменит их идентификаторы.

Решением этой проблемы является использование *таблицы идентификаторов*, чтобы разорвать прямую связь между номером слота записи и ее местоположением на странице. Таблица идентификаторов – это массив целых чисел, хранящийся в начале страницы. Каждый элемент массива обозначает идентификатор записи и хранит ее местоположение; значение 0 означает, что этому идентификатору не соответствует ни одна запись. На рис. 6.7c показаны те же данные, что и на рис. 6.7b, но с таблицей идентификаторов. Таблица идентификаторов содержит три элемента: два указывают на записи со смеще-

ниями 63 и 43 в блоке, а третий – пустой. Запись, начинающаяся со смещения 63, имеет идентификатор 0, а запись, начинающаяся со смещения 43, имеет идентификатор 2. В данный момент в блоке нет записи с идентификатором 1.

Таблица идентификаторов обеспечивает уровень косвенности, позволяющий диспетчеру записей перемещать записи в блоке. Если запись перемещается, корректируется соответствующий ей элемент в таблице идентификаторов; если запись удаляется, в соответствующий ей элемент записывается 0. Когда вставляется новая запись, диспетчер записей отыскивает доступный элемент и назначает его в качестве идентификатора новой записи. Таким образом, таблица идентификаторов позволяет перемещать записи переменной длины внутри блока и обеспечивает неизменность их идентификаторов.

Таблица идентификаторов расширяется по мере увеличения числа записей в блоке. Массив имеет переменный размер, потому что блок может содержать разное число записей переменной длины. Обычно таблица идентификаторов размещается на одном конце блока, а записи – на другом, и они растут навстречу друг другу. Эту ситуацию можно увидеть на рис. 6.7c, где первая запись в блоке находится в правом конце блока.

Таблица идентификаторов делает ненужными флаги заполненности. Запись занята, если на нее ссылается какой-либо из элементов таблицы идентификаторов. Пустым записям соответствуют элементы массива со значением 0 (на самом деле они могут даже не существовать). Таблица идентификаторов также позволяет диспетчеру записей быстро находить любую запись в блоке. Чтобы перейти к записи с определенным идентификатором, диспетчер просто использует смещение, хранящееся в соответствующем элементе таблицы идентификаторов; чтобы перейти к следующей записи, диспетчеру достаточно просмотреть таблицу идентификаторов и отыскать следующий ненулевой элемент.

6.2.3. Реализация расщепляемых записей

В этом разделе рассматриваются способы реализации расщепляемых записей. Если записи не расщеплять, первая запись в каждом блоке всегда начинается с одного и того же смещения. С расщепляемыми записями ситуация иная. Поэтому диспетчер записей должен хранить целое число в начале каждого блока, определяющее смещение первой записи.

Например, взгляните на рис. 6.8. Первое целое число в блоке 0 имеет значение 4, означающее, что первая запись R1 начинается со смещения 4 (то есть сразу после целого числа). Запись R2 охватывает блоки 0 и 1, и поэтому первая запись в блоке 1 – запись R3 – начинается со смещения 60. Запись R3 занимает полностью блок 2 и переходит в блок 3. Запись R4 является первой записью в блоке 3 и начинается со смещения 30. Обратите внимание, что первое целое число в блоке 2 равно 0 и указывает, что в этом блоке не начинается никакая запись.

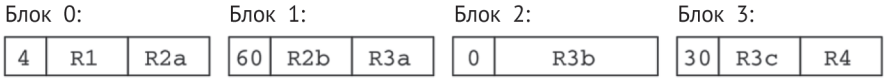


Рис. 6.8. Реализация расщепляемых записей

Диспетчер записей может расщепить запись двумя разными способами. Первый – заполнить блок до конца, расщепив запись по границе блока, а остальные байты поместить в следующий блок или блоки. Второй – записывать одно за другим поля записи, пока на странице достаточно места, а затем продолжить записывать поля в новую страницу. Преимущество первого способа состоит в том, что он не тратит пространство впустую, но при этом какие-то поля могут оказаться размещенными в разных блоках. Чтобы получить значение такого поля, диспетчеру придется объединить байты из двух или более блоков.

6.2.4. Реализация неоднородных файлов

Если диспетчер записей поддерживает неоднородные файлы, то он также должен поддерживать записи переменной длины, потому что записи из разных таблиц не обязательно будут иметь одинаковый размер. С обработкой блоков неоднородных файлов связаны две проблемы:

- диспетчер записей должен знать компоновку каждого типа записей в блоке;
- диспетчер записей должен знать, какой таблице принадлежит каждая запись.

Первую проблему можно решить, сохранив массив компоновок, по одному для каждой возможной таблицы. Вторая проблема решается добавлением дополнительного значения в начало каждой записи; это значение иногда называется *значением тега* – оно служит индексом в массиве компоновок и определяет таблицу, которой принадлежит запись.

Например, вернемся к рис. 6.2, где изображены неоднородные блоки с записями таблиц DEPT и STUDENT. Диспетчер записей будет хранить массив с компоновками обеих этих таблиц. Предположим, что компоновка таблицы DEPT находится в элементе массива с индексом 0, а компоновка STUDENT – в элементе с индексом 1. Тогда значение тега для каждой записи из DEPT будет равно 0, а из STUDENT – 1.

Поведение диспетчера записей при этом не сильно изменится. Обращаясь к записи, диспетчер прочитает значения ее тега и определит, какую компоновку использовать. Затем он сможет использовать эту компоновку для чтения или изменения любого поля, так же как в случае с однородными записями.

Журнал в SimpleDB является наглядным примером неоднородных файлов. Первым значением каждой журнальной записи является целое число, определяющее ее тип. Диспетчер восстановления использует это значение, чтобы определить, как читать остальную часть записи.

6.3. Страницы записей в SimpleDB

В следующих двух разделах рассматривается диспетчер записей в SimpleDB, реализующий базовый вариант, представленный в разделе 6.2.1. В этом разделе описывается реализация страниц записей, а в следующем рассказывается, как реализовать файл страниц. В некоторых упражнениях в конце главы вам будет предложено изменить его и реализовать другие решения.

6.3.1. Управление информацией о записях

Для управления информацией о записях диспетчер записей в SimpleDB использует классы `Schema` и `Layout`. Их API приводится в листинге 6.1.

Листинг 6.1. API классов в SimpleDB для управления информацией о записях

Schema

```
public Schema();
public void addField(String fldname, int type, int length);
public void addIntField(String fldname);
public void addStringField(String fldname, int length);
public void add(String fldname, Schema sch);
public void addAll(Schema sch);

public List<String> fields();
public boolean hasField(String fldname);
public int type(String fldname);
public int length(String fldname);
```

Layout

```
public Layout(Schema schema);
public Layout(Schema schema, Map<String,Integer> offsets, int slotSize);

public Schema schema();
public int offset(String fldname);
public int slotSize();
```

Объект `Schema` содержит *схему* записи, то есть имя и тип каждого поля, а также длину каждого строкового поля. Эти сведения соответствуют тому, что указал пользователь при создании таблицы, и не содержат физической информации. Например, длина строки – это максимально допустимое количество символов, а не размер в байтах.

Схему может рассматривать как список троек вида [имя поля, тип, длина]. Класс `Schema` имеет пять методов для добавления тройки в список. Метод `addField` добавляет тройку явно. Методы `addIntField`, `addStringField`, `add` и `addAll` являются вспомогательными: первые два вычисляют тройки, а последние копируют тройки из существующей схемы. В классе также есть методы доступа, позволяющие получить коллекцию имен полей, определить присутствие указанного поля в коллекции, а также получить тип и длину указанного поля.

Класс `Layout` представляет *компоновку* и дополнительно содержит физическую информацию о записи. Он вычисляет размеры полей и слотов, а также смещения полей в слоте. Класс имеет два конструктора, соответствующих двум способам создания объекта `Layout`. Первый конструктор вызывается при создании таблицы; он вычисляет информацию о компоновке на основе заданной схемы. Второй вызывается после создания таблицы; клиент просто передает предварительно рассчитанные значения этому конструктору.

Фрагмент кода в листинге 6.2 иллюстрирует использование этих двух классов. Первая часть кода создает схему с тремя полями из таблицы COURSE, а затем создает на ее основе компоновку. Вторая часть кода выводит имя и смещение каждого поля.

Листинг 6.2. Определение структуры записей COURSE

```
Schema sch = new Schema();
sch.addIntField("cid");
sch.addStringField("title", 20);
sch.addIntField("deptid");
Layout layout = new Layout(sch);

for (String fldname : layout.schema().fields()) {
    int offset = layout.offset(fldname);
    System.out.println(fldname + " has offset " + offset);
}
```

6.3.2. Реализация классов Schema и Layout

Класс `Schema` имеет простую реализацию, показанную в листинге 6.3. Класс хранит тройки в ассоциативном массиве, роль ключей в котором играют имена полей. Объект, связанный с именем поля, имеет тип приватного класса `FieldInfo`, который хранит длину и тип поля.

Листинг 6.3. Определение класса Schema в SimpleDB

```
public class Schema {
    private List<String> fields = new ArrayList<>();
    private Map<String,FieldInfo> info = new HashMap<>();

    public void addField(String fldname, int type, int length) {
        fields.add(fldname);
        info.put(fldname, new FieldInfo(type, length));
    }

    public void addIntField(String fldname) {
        addField(fldname, INTEGER, 0);
    }

    public void addStringField(String fldname, int length) {
        addField(fldname, VARCHAR, length);
    }

    public void add(String fldname, Schema sch) {
        int type = sch.type(fldname);
        int length = sch.length(fldname);
        addField(fldname, type, length);
    }

    public void addAll(Schema sch) {
        for (String fldname : sch.fields())
            add(fldname, sch);
    }

    public List<String> fields() {
        return fields;
    }

    public boolean hasField(String fldname) {
        return fields.contains(fldname);
    }
```

```
    public int type(String fldname) {
        return info.get(fldname).type;
    }

    public int length(String fldname) {
        return info.get(fldname).length;
    }

    class FieldInfo {
        int type, length;
        public FieldInfo(int type, int length) {
            this.type = type;
            this.length = length;
        }
    }
}
```

Типы обозначаются константами `INTEGER` и `VARCHAR`, как определено в JDBC-классе `Types`. Длина поля имеет смысл только для строковых полей; метод `addIntField` задает длину для целочисленных полей, равную 0, но это значение не имеет смысла, потому что никогда не будет использоваться.

Реализация класса `Layout` представлена в листинге 6.4. Первый конструктор размещает поля в порядке их появления в схеме. Он определяет длину каждого поля в байтах, вычисляет размер слота как сумму длин полей, добавляет четыре байта со смещением 0 для целочисленного флага заполненности и вычисляет смещение для каждого следующего поля, опираясь на конец предыдущего (то есть без дополнения).

Листинг 6.4. Определение класса Layout в SimpleDB

```
public class Layout {
    private Schema schema;
    private Map<String,Integer> offsets;
    private int slotsize;

    public Layout(Schema schema) {
        this.schema = schema;
        offsets = new HashMap<>();
        int pos = Integer.BYTES; // место для флага заполненности
        for (String fldname : schema.fields()) {
            offsets.put(fldname, pos);
            pos += lengthInBytes(fldname);
        }

        slotsize = pos;
    }

    public Layout(Schema schema, Map<String,Integer> offsets, int slotsize) {
        this.schema   = schema;
        this.offsets  = offsets;
        this.slotsize = slotsize;
    }

    public Schema schema() {
        return schema;
    }
```

```
    public int offset(String fldname) {
        return offsets.get(fldname);
    }

    public int slotSize() {
        return slotsize;
    }

    private int lengthInBytes(String fldname) {
        int fldtype = schema.type(fldname);
        if (fldtype == INTEGER)
            return Integer.BYTES;
        else // fldtype == VARCHAR
            return Page.maxLength(schema.length(fldname));
    }
}
```

6.3.3. Управление записями на странице

Класс RecordPage управляет записями на странице. Его API показан в листинге 6.5.

Листинг 6.5. API класса RecordPage в SimpleDB для управления записями на странице

RecordPage

```
public RecordPage(Transaction tx, BlockId blk, Layout layout);
public BlockId block();

public int    getInt (int slot, String fldname);
public String getString(int slot, String fldname);
public void   setInt (int slot, String fldname, int val);
public void   setString(int slot, String fldname, String val);
public void   format();
public void   delete(int slot);

public int    nextAfter(int slot);
public int    insertAfter(int slot);
```

Методы nextAfter и insertAfter ищут на странице нужные записи. Метод nextAfter возвращает первый занятый слот, следующий за указанным, пропуская все пустые слоты. Отрицательное возвращаемое значение указывает, что все оставшиеся слоты пустые. Метод insertAfter ищет первый пустой слот после указанного. Если такой слот будет найден, метод записывает в его флаг значение USED и возвращает номер слота. В противном случае возвращается значение –1.

Методы get и set обращаются к значению указанного поля в указанной записи. Метод delete устанавливает флаг слота в EMPTY. Метод format записывает значения по умолчанию во все слоты на странице. Он устанавливает флаг заполненности каждого слота в значение EMPTY, во все целочисленные поля записывает 0, а в строковые – пустую строку.

Класс RecordTest в листинге 6.6 иллюстрирует использование методов класса RecordPage. Он определяет схему записи с двумя полями: целочисленным полем A и строковым полем B. Затем создает объект RecordPage для нового блока и форматирует его. Потом в цикле for вызывает метод insertAfter для заполнения страницы записями со случайными значениями полей. (Каждое значение в поле A является случайным числом от 0 до 49, а каждое значение в поле B – строковой версией этого числа.) Два цикла while вызывают nextAfter для

поиска страницы. Первый цикл удаляет некоторые записи, а второй выводит содержимое оставшихся записей.

Листинг 6.6. Тестирование класса RecordPage

```
public class RecordTest {
    public static void main(String[] args) throws Exception {
        SimpleDB db = new SimpleDB("recordtest", 400, 8);
        Transaction tx = db.newTx();
        Schema sch = new Schema();
        sch.addIntField("A");
        sch.addStringField("B", 9);
        Layout layout = new Layout(sch);
        for (String fldname : layout.schema().fields()) {
            int offset = layout.offset(fldname);
            System.out.println(fldname + " has offset " + offset);
        }
        BlockId blk = tx.append("testfile");
        tx.pin(blk);
        RecordPage rp = new RecordPage(tx, blk, layout);
        rp.format();

        System.out.println("Filling the page with random records.");
        int slot = rp.insertAfter(-1);
        while (slot >= 0) {
            int n = (int) Math.round(Math.random() * 50);
            rp.setInt(slot, "A", n);
            rp.setString(slot, "B", "rec"+n);
            System.out.println("inserting into slot " + slot + ": {"
                               + n + ", " + "rec"+n + "}");
            slot = rp.insertAfter(slot);
        }
        System.out.println("Deleted these records with A-values < 25.");
        int count = 0;
        slot = rp.nextAfter(-1);
        while (slot >= 0) {
            int a = rp.getInt(slot, "A");
            String b = rp.getString(slot, "B");
            if (a < 25) {
                count++;
                System.out.println("slot " + slot + ": {" + a + ", " + b + "}");
                rp.delete(slot);
            }
            slot = rp.nextAfter(slot);
        }
        System.out.println(count + " values under 25 were deleted.\n");
        System.out.println("Here are the remaining records.");
        slot = rp.nextAfter(-1);
        while (slot >= 0) {
            int a = rp.getInt(slot, "A");
            String b = rp.getString(slot, "B");
            System.out.println("slot " + slot + ": {" + a + ", " + b + "}");
            slot = rp.nextAfter(slot);
        }
        tx.unpin(blk);
        tx.commit();
    }
}
```

6.3.4. Реализация страниц записей

В SimpleDB используются страницы, организованные в слоты, как показано на рис. 6.5. Единственное отличие: флаги заполненности реализованы в виде 4-байтных целых чисел, а не отдельных байтов (SimpleDB не поддерживает однобайтные значения). Определение класса `RecordPage` показано в листинге 6.7.

Приватный метод `offset` определяет смещение начала слота для данной записи, используя размер слотов. Методы `get` и `set` вычисляют местоположение указанного поля, добавляя смещение поля к смещению слота. Методы `nextAfter` и `insertAfter` вызывают приватный метод `searchAfter`, чтобы найти слот с флагом `USED` или `EMPTY` соответственно. Метод `searchAfter` поочередно просматривает слоты, пока не найдет слот с указанным флагом или в блоке не останется слотов. Метод `delete` устанавливает флаг указанного слота в значение `EMPTY`, а `insertAfter` устанавливает флаг найденного слота в значение `USED`.

Листинг 6.7. Определение класса RecordPage в SimpleDB

```
public class RecordPage {
    public static final int EMPTY = 0, USED = 1;
    private Transaction tx;
    private BlockId blk;
    private Layout layout;

    public RecordPage(Transaction tx, BlockId blk, Layout layout) {
        this.tx = tx;
        this.blk = blk;
        this.layout = layout;
        tx.pin(blk);
    }

    public int getInt(int slot, String fldname) {
        int fldpos = offset(slot) + layout.offset(fldname);
        return tx.getInt(blk, fldpos);
    }

    public String getString(int slot, String fldname) {
        int fldpos = offset(slot) + layout.offset(fldname);
        return tx.getString(blk, fldpos);
    }

    public void setInt(int slot, String fldname, int val) {
        int fldpos = offset(slot) + layout.offset(fldname);
        tx.setInt(blk, fldpos, val, true);
    }

    public void setString(int slot, String fldname, String val) {
        int fldpos = offset(slot) + layout.offset(fldname);
        tx.setString(blk, fldpos, val, true);
    }

    public void delete(int slot) {
        setFlag(slot, EMPTY);
    }
```

```
    public void format() {
        int slot = 0;
        while (isValidSlot(slot)) {
            tx.setInt(blk, offset(slot), EMPTY, false);
            Schema sch = layout.schema();
            for (String fldname : sch.fields()) {
                int fldpos = offset(slot) + layout.offset(fldname);
                if (sch.type(fldname) == INTEGER)
                    tx.setInt(blk, fldpos, 0, false);
                else
                    tx.setString(blk, fldpos, "", false);
            }
            slot++;
        }
    }

    public int nextAfter(int slot) {
        return searchAfter(slot, USED);
    }

    public int insertAfter(int slot) {
        int newslot = searchAfter(slot, EMPTY);
        if (newslot >= 0)
            setFlag(newslot, USED);
        return newslot;
    }

    public BlockId block() {
        return blk;
    }

    // Приватные вспомогательные методы
    private void setFlag(int slot, int flag) {
        tx.setInt(blk, offset(slot), flag, true);
    }

    private int searchAfter(int slot, int flag) {
        slot++;
        while (isValidSlot(slot)) {
            if (tx.getInt(blk, offset(slot)) == flag)
                return slot;
            slot++;
        }
        return -1;
    }

    private boolean isValidSlot(int slot) {
        return offset(slot+1) <= tx.blockSize();
    }

    private int offset(int slot) {
        return slot * layout.slotSize();
    }
}
```

6.4. Сканирование таблиц в SimpleDB

Страница записей управляет записями в блоке. В этом разделе рассматривается механизм сканирования таблиц, который сохраняет произвольное число записей в нескольких блоках в файле.

6.4.1. Сканирование таблиц

Класс `TableScan` управляет записями в таблице. Его API показан в листинге 6.8.

Листинг 6.8. API класса TableScan в SimpleDB

TableScan

```
public TableScan(Transaction tx, String tblname, Layout layout);

public void    close();
public boolean hasField(String fldname);

// методы для выбора текущей записи
public void    beforeFirst();
public boolean next();
public void    moveToRid(RID r);
public void    insert();

// методы для доступа к текущей записи
public int    getInt(String fldname);
public String getString(String fldname);
public void   setInt(String fldname, int val);
public void   setString(String fldname, String val);
public RID    currentRid();
public void   delete();
```

RID

```
public RID(int blknum, int slot);
public int     blockNumber();
public int     slot();
```

Объект `TableScan` запоминает позицию *текущей записи* и имеет методы для смены текущей записи и доступа к ее содержимому. Метод `beforeFirst` устанавливает позицию текущей записи перед первой записью в файле, а метод `next` устанавливает позицию текущей записи на следующую запись в файле. Если в текущем блоке больше нет записей, производится чтение следующих блоков из файла, пока не будет найдена следующая запись. Если в файле больше нет записей, то вызов `next` вернет `false`.

Методы `get`, `set` и `delete` применяются к текущей записи. Метод `insert` вставляет в файл новую запись, начиная поиск места для нее с блока, где находится текущая запись. В отличие от метода вставки в `RecordPage`, этот метод всегда выполняется успешно; если он не находит места для записи в существующих блоках, то добавляет в файл новый блок и вставляет запись туда.

Каждая запись в файле идентифицируется парой значений: номером блока в файле и номером слота внутри блока. Эти два значения известны как *идентификатор записи* (record identifier, или rid). Идентификаторы записей реализуются классом `RID`. Его конструктор сохраняет два значения и имеет два метода доступа к ним: `blockNumber` и `slot`.

Класс TableScan имеет два метода для работы с идентификаторами записей. Метод moveToRid назначает текущей запись с указанным идентификатором, а метод currentRid возвращает идентификатор текущей записи.

Класс TableScan обеспечивает уровень абстракции, значительно отличающийся от других классов, которые были представлены до сих пор. В отличие от классов Page, Buffer, Transaction и RecordPage, методы которых применяются к конкретному блоку, класс TableScan скрывает блочную структуру от клиентов. Как правило, клиент не знает (или ему это не интересно), к какому блоку в данный момент он обращается.

Класс TableScanTest в листинге 6.9 иллюстрирует использование класса TableScan. Он похож на класс RecordTest, за исключением того, что вставляет 50 записей в файл. Вызов ts.insert выделяет столько новых блоков, сколько необходимо для хранения записей. В данном случае будет выделено три блока (по 18 записей в блоке). Однако код ничего не знает об этом. Если запустить этот пример несколько раз, то можно заметить, что каждый раз в файл вставляется 50 новых записей, и они заполняют слоты, оставшиеся от ранее удаленных записей.

Листинг 6.9. Тестирование класса TableScan

```
public class TableScanTest {
    public static void main(String[] args) throws Exception {
        SimpleDB db = new SimpleDB("tabletest", 400, 8);
        Transaction tx = db.newTx();
        Schema sch = new Schema();
        sch.addIntField("A");
        sch.addStringField("B", 9);
        Layout layout = new Layout(sch);
        for (String fldname : layout.schema().fields()) {
            int offset = layout.offset(fldname);
            System.out.println(fldname + " has offset " + offset);
        }
        TableScan ts = new TableScan(tx, "T", layout);

        System.out.println("Filling the table with 50 random records.");
        ts.beforeFirst();
        for (int i=0; i<50; i++) {
            ts.insert();
            int n = (int) Math.round(Math.random() * 50);
            ts.setInt("A", n);
            ts.setString("B", "rec"+n);
            System.out.println("inserting into slot " + ts.getRid() + ": {"
                               + n + ", " + "rec"+n + "}");
        }

        System.out.println("Deleting records with A-values < 25.");
        int count = 0;
        ts.beforeFirst();
        while (ts.next()) {
            int a = ts.getInt("A");
            String b = ts.getString("B");
            if (a < 25) {
                count++;
                System.out.println("slot " + ts.getRid() + ": {" + a + ", " + b + "}");
                ts.delete();
            }
        }
        System.out.println(count + " values under 10 were deleted.\n");
```

```
        System.out.println("Here are the remaining records.");
        ts.beforeFirst();
        while (ts.next()) {
            int a = ts.getInt("A");
            String b = ts.getString("B");
            System.out.println("slot " + ts.getRid() +
                               ": {" + a + ", " + b + "}");
        }
        ts.close();
        tx.commit();
    }
}
```

6.4.2. Реализация TableScan

Определение класса `TableScan` показано в листинге 6.10. Объект `TableScan` хранит страницу записей, соответствующую текущему блоку. Методы `get`, `set` и `delete` просто вызывают соответствующий метод страницы записей. Приватный метод `moveToBlock` вызывается для смены текущего блока; он закрывает страницу с текущей записью и открывает другую, соответствующую указанному блоку, устанавливая указатель текущей записи перед первым слотом.

Листинг 6.10. Определение класса TableScan в SimpleDB

```
public class TableScan implements UpdateScan {
    private Transaction tx;
    private Layout layout;
    private RecordPage rp;
    private String filename;
    private int currentslot;

    public TableScan(Transaction tx, String tblname, Layout layout) {
        this.tx = tx;
        this.layout = layout;
        filename = tblname + ".tbl";
        if (tx.size(filename) == 0)
            moveToNewBlock();
        else
            moveToBlock(0);
    }

    // Реализация интерфейса Scan
    public void close() {
        if (rp != null)
            tx.unpin(rp.block());
    }
    public void beforeFirst() {
        moveToBlock(0);
    }

    public boolean next() {
        currentslot = rp.nextAfter(currentslot);
        while (currentslot < 0) {
            if (atLastBlock())
                return false;
            moveToBlock(rp.block().number()+1);
            currentslot = rp.nextAfter(currentslot);
        }
        return true;
    }
```

```
public int getInt(String fldname) {
    return rp.getInt(currentslot, fldname);
}

public String getString(String fldname) {
    return rp.getString(currentslot, fldname);
}

public Constant getVal(String fldname) {
    if (layout.schema().type(fldname) == INTEGER)
        return new IntConstant(getInt(fldname));
    else
        return new StringConstant(getString(fldname));
}

public boolean hasField(String fldname) {
    return layout.schema().hasField(fldname);
}

// Реализация интерфейса UpdateScan
public void setInt(String fldname, int val) {
    rp.setInt(currentslot, fldname, val);
}

public void setString(String fldname, String val) {
    rp.setString(currentslot, fldname, val);
}

public void setVal(String fldname, Constant val) {
    if (layout.schema().type(fldname) == INTEGER)
        setInt(fldname, (Integer)val.asJavaVal());
    else
        setString(fldname, (String)val.asJavaVal());
}

public void insert() {
    currentslot = rp.insertAfter(currentslot);
    while (currentslot < 0) {
        if (atLastBlock())
            moveToNewBlock();
        else
            moveToBlock(rp.block().number()+1);
        currentslot = rp.insertAfter(currentslot);
    }
}

public void delete() {
    rp.delete(currentslot);
}

public void moveToRid(RID rid) {
    close();
    BlockId blk = new BlockId(filename, rid.blockNumber());
    rp = new RecordPage(tx, blk, layout);
    currentslot = rid.slot();
}
```

```
public RID getRid() {
    return new RID(rp.block().number(), currentslot);
}

// Приватные вспомогательные методы
private void moveToBlock(int blknum) {
    close();
    BlockId blk = new BlockId(filename, blknum);
    rp = new RecordPage(tx, blk, layout);
    currentslot = -1;
}

private void moveToNewBlock() {
    close();
    BlockId blk = tx.append(filename);
    rp = new RecordPage(tx, blk, layout);
    rp.format();
    currentslot = -1;
}

private boolean atLastBlock() {
    return rp.block().number() == tx.size(filename) - 1;
}
```

Вот алгоритм работы метода `next`:

1. Перейти к следующей записи на текущей странице записей.
2. Если записей на странице больше нет, перейти к следующему блоку в файле и попытаться найти следующую запись в нем.
3. Продолжать, пока не будет найдена следующая запись или достигнут конец файла.

Возможно, что несколько блоков в файле будут пустыми (см. упражнение 6.2), поэтому методу `next` может потребоваться обойти несколько блоков.

Метод `insert` пытается вставить новую запись, начиная с текущего блока. Если текущий блок заполнен, то метод переходит к следующему блоку и пытается вставить новую запись туда. Так продолжается до тех пор, пока не будет найден пустой слот. Если все блоки заполнены, метод `insert` добавит в файл новый блок и вставит запись туда.

Класс `TableScan` реализует интерфейс `UpdateScan` (а также `Scan`, по цепочке наследования). Эти интерфейсы играют важную роль в обработке запросов и будут обсуждаться в главе 8. Методы `getVal` и `setVal` также обсуждаются в главе 8. Они извлекают и сохраняют объекты типа `Constant`. Константа – это абстракция типа-значения (например, `int` или `String`), которая упрощает выражение запроса, избавляя от необходимости знать тип заданного поля.

Объекты `RID` – это просто комбинации двух целых чисел: номера блока и номера слота. Соответственно, класс `RID` имеет очень простую реализацию, представленную в листинге 6.11.

Листинг 6.11. Определение класса RID в SimpleDB

```
public class RID {
    private int blknum;
    private int slot;

    public RID(int blknum, int slot) {
        this.blknum = blknum;
        this.slot = slot;
    }

    public int blockNumber() {
        return blknum;
    }

    public int slot() {
        return slot;
    }

    public boolean equals(Object obj) {
        RID r = (RID) obj;
        return blknum == r.blknum && slot==r.slot;
    }

    public String toString() {
        return "[" + blknum + ", " + slot + "]";
    }
}
```

6.5. Итоги

- Диспетчер записей – это компонент системы баз данных для хранения записей в файле. Он имеет три основные обязанности:
  - размещение полей внутри записи;
  - размещение записей внутри блока;
  - организация доступа к записям в файле.

Вот некоторые вопросы, которые необходимо решить при проектировании диспетчера записей.

- Прежде всего нужно решить, поддерживать ли *поля переменной длины*. Записи фиксированной длины легко реализуются, потому что поля можно изменять на месте. Изменение поля переменной длины может привести к тому, что запись выйдет за границы блока и ее придется перенести в *блок переполнения*.
- SQL поддерживает три разных типа для строковых значений: `char`, `varchar` и `clob`.
  - Тип `char` наиболее естественно реализуется с использованием представления фиксированной длины.
  - Тип `varchar` наиболее естественно реализуется с использованием представления переменной длины.
  - Тип `clob` наиболее естественно реализуется с использованием представления фиксированной длины и сохранением строки во вспомогательном файле.

- Типичный способ реализации записей переменной длины – использование *таблицы идентификаторов*. Каждый элемент в этой таблице ссылается на запись на странице. Запись может свободно перемещаться по странице, для чего достаточно просто изменить соответствующий элемент в таблице идентификаторов.
- Второй вопрос – *расщеплять ли записи*. Расщепляемые записи полезны тем, что экономят место и позволяют создавать записи большого размера, но они сложнее в реализации.
- Третий вопрос – поддерживать ли *неоднородные файлы*. Неоднородные файлы позволяют *кластеризовать* в странице связанные записи. Кластеризация может существенно ускорить вычисление соединений таблиц, но, как правило, ухудшает производительность других запросов. Диспетчер записей может реализовать поддержку неоднородных файлов, сохраняя *поле тега* в начале каждой записи, значение которого определяет таблицу, которой принадлежит запись.
- Четвертый вопрос – как определять смещение каждого поля в записи. Диспетчеру записей может потребоваться дополнять поля, чтобы *выровнять* их по соответствующим границам байтов. Поле фиксированной длины имеет одно и то же смещение в каждой записи. При использовании полей переменной длины может потребоваться выполнить обход записи, чтобы найти начало требуемого поля.

6.6. Для дополнительного чтения

Идеи и методы, представленные в этой главе, появились еще на заре реляционных баз данных. Раздел 3.3 в Stonebraker et al. (1976) описывает подход, предпринятый в первой версии INGRES; этот подход использует вариант таблицы идентификаторов, описанной в разделе 6.2.2 в этой главе. Раздел 3 в Astrahan et al. (1976) описывает структуру страницы для ранней системы баз данных System R (которая впоследствии превратилась в DB2 компании IBM), поддерживающую хранение неоднородных записей. Обе статьи обсуждают широкий спектр идей реализации, и их стоит прочесть полностью. Более подробное обсуждение этих методов вместе с примером реализации диспетчера записей на C можно найти в главе 14 в Grey and Reuter (1993).

Стратегия хранения записей на странице в виде непрерывной последовательности полей не обязательно является наилучшей. В статье Ailamaki et al. (2002) предлагается другой подход: разбивать записи по полям и хранить значения одноименных полей вместе. Такая организация записей не влияет на количество обращений к диску, выполняемых диспетчером записей, но она значительно повышает производительность, потому что при подобном подходе кеш данных процессора используется особенно эффективно. В статье Stonebraker et al. (2005) эта идея развивается еще дальше и предлагается организовывать по полям целые таблицы, то есть хранить вместе значения одноименных полей для всех записей. В статье показано, как хранение данных по полям может обеспечить большую компактность, чем хранение данных по записям, и увеличить эффективность обработки запросов.

Описание реализации стратегии поддержки очень больших записей можно найти в Carey et al. (1986).

Ailamaki, A., DeWitt, D., & Hill, M. (2002). «Data page layouts for relational databases on deep memory hierarchies». VLDB Journal, 11 (3), 198–215.

Astrahan, M., Blasgen, M., Chamberlin, D., Eswaren, K., Gray, J., Griffiths, P., King, W., Lorie, R., McJones, P., Mehl, J., Putzolu, G., Traiger, I., Wade, B., & Watson, V. (1976). «System R: Relational approach to database management». ACM Transactions on Database Systems, 1 (2), 97–137.

Carey, M., DeWitt, D., Richardson, J., & Shekita, E. (1986). «Object and file management in the EXODUS extendable database system». In Proceedings of the VLDB Conference (p. 91–100).

Gray, J., & Reuter, A. (1993). «Transaction processing: concepts and techniques». San Mateo, CA: Morgan Kaufman.

Stonebraker, M., Abadi, D., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., O'Neil, P., Rasin, A., Tran, N., & Zdonik, S. (2005). «C-Store: A column-oriented DBMS». In Proceedings of the VLDB Conference (p. 553–564).

Stonebraker, M., Kreps, P., Wong, E., & Held, G. (1976). «The design and implementation of INGRES». ACM Transactions on Database Systems, 1 (3), 189–222.

6.7. Упражнения

Теория

6.1. Пусть размер блока составляет 400 байт и записи не расщепляются. Подсчитайте, сколько записей поместится на странице SimpleDB и каков объем потерянного пространства на странице для каждого из следующих размеров слотов: 10 байт, 20 байт, 50 байт и 100 байт.

6.2. Объясните, как в файле с таблицей могут появиться пустые блоки.

6.3. Исследуйте определения всех таблиц в базе данных университета (кроме STUDENT).

a) Приведите компоновку каждой таблицы, как показано табл. 6.1. (Можете использовать объявления `varchar`, как в файлах демонстрационного клиента, или предположить, что все строковые поля определены как `varchar(20)`.)

b) Изобразите страницу записей (как на рис. 6.5) для каждой таблицы, используя записи, изображенные на рис. 1.1. Так же как на рис. 6.5, предположите, что флаг заполненности занимает один байт. Также предположите, что реализация строковых полей использует представление фиксированной длины.

c) Выполните п. «b», но в предположении, что реализация строковых полей использует представление переменной длины. Используйте рис. 6.7c в качестве основы.

d) Измените рисунки, выполненные в пп. «b» и «c», чтобы показать состояние страниц после удаления второй записи.

6.4. Другой способ обработки очень больших строк – хранить их за пределами таблицы. Такие строки можно размещать в файле, а путь к этому файлу сохранить в базе данных. Эта стратегия устраняет потребность в типе `clob`. Приведите несколько причин, почему эта стратегия не особенно хороша.

6.5. Предположим, что вам понадобилось вставить запись в блок, имеющий блок переполнения, как показано на рис. 6.6b. Правильно ли будет сохранить запись в блоке переполнения? Объясните.

6.6. Вот еще один способ реализации записей переменной длины. Каждый блок имеет две области: последовательность слотов фиксированной длины (как в SimpleDB) и область для хранения значений переменной длины. Запись хранится в слоте. Значения фиксированной длины сохраняются в самой записи; значения переменной длины сохраняются во второй области, а внутри записи находятся смещения таких значений в блоке. Например, записи, показанные на рис. 6.7a, могут храниться, как показано на рис. 6.9.

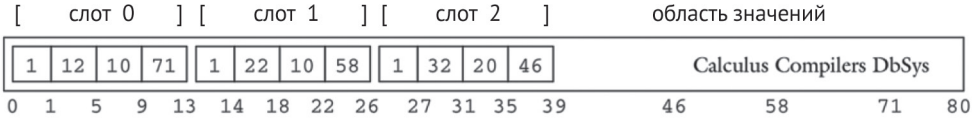


Рис. 6.9. Вариант хранения записей с отдельной областью в блоке для значений переменной длины

a) Объясните, что должно произойти при изменении значения переменной длины. Для этого может понадобиться блок переполнения? Если да, то как это должно выглядеть?
b) Сравните эту стратегию хранения со стратегией на основе таблицы идентификаторов. Объясните преимущества каждой.
c) Какую стратегию предпочли бы вы? Почему?

6.7. Использование целого байта для флага заполненности приводит к расходованию пространства впустую, потому что для флага достаточно одного бита. Альтернативная стратегия заключается в сохранении битов заполненности для слотов в отдельном битовом массиве в начале блока. Такой битовый массив можно реализовать как одно или несколько 4-байтных целых чисел.

a) Сравните этот битовый массив с таблицей идентификаторов на рис. 6.7c.
b) Предположим, что размер блока составляет 4 Кбайта, а размер записей будет не менее 15 байт. Сколько целых чисел потребуется для хранения битового массива?
c) Опишите алгоритм поиска пустого слота для вставки новой записи.
d) Опишите алгоритм поиска следующего непустого слота в блоке.

Практика

6.8. Измените класс `RecordPage` так, чтобы он закреплял блок не в конструкторе, а в начале каждого метода `get` и `set`. Точно так же блок должен открепляться в конце каждого из методов `get` и `set`, чтобы избавить от необходимости вызывать метод `close`. Как вы думаете, такая реализация лучше, чем в SimpleDB? Объясните.

6.9. Измените реализацию диспетчера записей, чтобы поля `varchar` имели переменную длину.

6.10. SimpleDB поддерживает чтение файлов только в прямом направлении.

a) Добавьте в классы `TableScan` и `RecordPage` метод `previous` для перехода к предыдущей записи, а также метод `afterLast`, который устанавливает позицию текущей записи после последней записи в файле (или в странице).

b) Измените программу `TableScanTest`, чтобы она выводила записи в обратном порядке.

6.11. Измените реализацию диспетчера записей, чтобы он поддерживал расщепление записей.

6.12. Измените класс `Layout` так, чтобы он дополнял строковые поля до размера, кратного 4.

6.13. Измените реализацию диспетчера записей в SimpleDB так, чтобы он поддерживал значения `null` в полях. Учитывая невозможность использования конкретного целого или строкового значения для обозначения `null`, используйте флаги, указывающие, что поле имеет неопределенное значение. Например, предположим, что запись содержит N полей. В этом случае в каждой записи можно сохранить N дополнительных бит; значение 1 в i-м бите соответствует неопределенному значению в i-м поле. Для этой цели, если допустить, что N < 32 и флаг заполненности реализован как целое число, можно использовать остальные биты во флаге. Бит 0 этого целого числа будет обозначать признак заполненности, как и раньше, а остальные биты – хранить признаки значения `null` в полях. Для этого потребуется внести следующие изменения в код:

- изменить класс `Layout`, добавив в него метод `bitLocation(fldname)`, который возвращает позицию null-бита во флаге для поля `fldname`;
- изменить классы `RecordPage` и `TableScan`, добавив два дополнительных публичных метода: `void setNull(fldname)`, записывающий 1 в соответствующий бит флага, и логический метод `isNull(fldname)`, возвращающий `true`, если null-бит для указанного поля в текущей записи содержит 1;
- изменить метод `format` в классе `RecordPage`, чтобы он явно сбрасывал null-биты для полей в новой записи;
- изменить методы `setString` и `setInt`, чтобы они сбрасывали null-бит для соответствующего поля в записи.

6.14. Предположим, что `setString` вызывается со строкой, которая длиннее, чем указано в схеме.

a) Объясните, что может пойти не так и когда ошибка будет обнаружена.

b) Исправьте код SimpleDB, чтобы ошибка обнаруживалась своевременно и обрабатывалась соответствующим образом.

Глава 7

Управление метаданными

В предыдущей главе рассматривалось, как диспетчер записей хранит записи в файлах. Однако, как вы видели, файл сам по себе бесполезен; диспетчеру записей также необходимо знать компоновку записей, чтобы «декодировать» содержимое каждого блока. Компоновка, описывающая структуру записей, является примером *метаданных*. В этой главе рассматриваются виды метаданных, поддерживаемых движком базы данных, их назначение, а также способы хранения метаданных в базе данных.

7.1. Диспетчер метаданных

Метаданные – это данные, описывающие организацию базы данных. Движок базы данных поддерживает широкий спектр метаданных. Например:

- метаданные *таблиц* описывают структуру записей в таблицах: длину, тип и смещение каждого поля. Примером этого вида метаданных может служить компоновка, используемая диспетчером записей;
- метаданные *представлений* описывают свойства каждого представления: определение и имя создавшего его пользователя. Эти метаданные помогают планировщику обрабатывать запросы, в которых упоминаются представления;
- метаданные *индексов* описывают индексы, определенные для таблиц (будут обсуждаться в главе 12). Планировщик использует эти метаданные, чтобы узнать, можно ли обработать запрос с помощью индексов;
- *статистические* метаданные описывают размер каждой таблицы и распределение значений ее полей. Оптимизатор использует эти метаданные для оценки стоимости запроса.

Метаданные из первых трех категорий создаются вместе с таблицей, представлением или индексом. Статистические метаданные генерируются при каждом обновлении базы данных.

Диспетчер метаданных – это компонент движка базы данных, который хранит и извлекает метаданные. Диспетчер метаданных в SimpleDB состоит из четырех отдельных диспетчеров, соответствующих каждому из четырех типов метаданных. Они подробно описываются в остальных разделах данной главы.

7.2. Метаданные таблиц

Метаданными таблиц в SimpleDB управляет класс `TableMgr`. Его API, представленный в листинге 7.1, состоит из конструктора и двух методов. Конструктор вызывается один раз, во время запуска системы. Метод `createTable` принимает имя таблицы и схему, вычисляет смещения полей в записи и сохраняет результаты в каталоге. Метод `getLayout` извлекает метаданные для указанной таблицы из каталога и возвращает объект `Layout` с этими метаданными.

Листинг 7.1. API диспетчера таблиц в SimpleDB

TableMgr

```
public TableMgr(boolean isnew, Transaction tx);
public void createTable(String tblname, Schema sch, Transaction tx);
public Layout getLayout(String tblname, Transactcion tx);
```

Класс `TableMgrTest` в листинге 7.2 иллюстрирует использование этих методов. Сначала он определяет схему, содержащую целочисленное поле с именем «A» и строковое поле с именем «B». Затем вызывает `createTable`, чтобы создать таблицу с именем «MyTable», имеющую эту схему, а потом `getLayout`, чтобы получить вычисленную компоновку.

Листинг 7.2. Использование методов диспетчера таблиц

```
public class TableMgrTest {
    public static void main(String[] args) throws Exception {
        SimpleDB db = new SimpleDB("tblmgrtest", 400, 8);
        Transaction tx = db.newTx();
        TableMgr tm = new TableMgr(true, tx);

        Schema sch = new Schema();
        sch.addIntField("A");
        sch.addStringField("B", 9);
        tm.createTable("MyTable", sch, tx);

        Layout layout = tm.getLayout("MyTable", tx);
        int size = layout.slotSize();
        Schema sch2 = layout.schema();
        System.out.println("MyTable has slot size " + size);
        System.out.println("Its fields are:");
        for (String fldname : sch2.fields()) {
            String type;
            if (sch2.type(fldname) == INTEGER)
                type = "int";
            else {
                int strlen = sch2.length(fldname);
                type = "varchar(" + strlen + ")";
            }
            System.out.println(fldname + ": " + type);
        }
        tx.commit();
    }
}
```

Диспетчер метаданных хранит свои метаданные в части базы данных, которая называется *каталогом*. Но как реализован каталог? На практике для этой цели чаще всего применяются специальные служебные таблицы в базе данных. SimpleDB использует две такие таблицы для хранения табличных метаданных: в `tblcat` хранятся метаданные для каждой таблицы, а в `fldcat` – метаданные для каждого поля каждой таблицы. Эти таблицы имеют следующие поля:

```
tblcat(TblName, SlotSize)
fldcat(TblName, FldName, Type, Length, Offset)
```

Каждой таблице в базе данных соответствует одна запись в `tblcat`, и каждому полю в каждой таблице – одна запись в `fldcat`. В поле `SlotSize` хранится длина слота в байтах, вычисленная с помощью класса `Layout`. Поле `Length` хранит длину поля в символах, как определено в схеме соответствующей таблицы. Например, в табл. 7.1 показаны таблицы каталога из университетской базы данных, изображенной на рис. 1.1. Обратите внимание, как информация о компоновке таблиц организована в серию записей `fldcat`. Поле `Type` в таблице `fldcat` содержит значения 4 и 12 – это коды типов `INTEGER` и `VARCHAR`, которые определены в JDBC-классе `Types`.

Таблица 7.1. Таблицы каталога из университетской базы данных

tblcat	TblName	SlotSize
	student	30
	dept	20
	course	36
	section	28
	enroll	22

fldcat	TblName	FldName	Type	Length	Offset
	student	sid	4	0	4
	student	sname	12	10	8
	student	majorid	4	0	22
	student	gradyear	4	0	26
	dept	did	4	0	4
	dept	dname	12	8	8
	course	cid	4	0	4
	course	title	12	20	8
	course	deptid	4	0	32
	section	sectid	4	0	4
	section	courseid	4	0	8
	section	prof	12	8	12
	section	year	4	0	24
	enroll	eid	4	0	4
	enroll	studentid	4	0	8
	enroll	sectionid	4	0	12
	enroll	grade	12	2	16

Доступ к таблицам каталога осуществляется так же, как к любым другим таблицам. Например, запрос SQL в листинге 7.3 извлекает имена и размеры всех полей в таблице STUDENT[1].

Листинг 7.3. Запрос SQL, извлекающий метаданные

```
select FldName, Length
from fldcat
where TblName = 'student'
```

Таблицы каталога даже содержат записи о собственных метаданных. Эти записи не показаны в табл. 7.1, но в упражнении 7.1 вам будет предложено определить их. В листинге 7.4 показан класс `CatalogTest`, который выводит длину записи для каждой таблицы и смещение каждого поля. Запустив этот код, вы увидите, что метаданные для таблиц каталога тоже выводятся на экран.

Листинг 7.4. Чтение таблиц каталога с помощью механизма сканирования таблиц

```
public class CatalogTest {
    public static void main(String[] args) throws Exception {
        SimpleDB db = new SimpleDB("catalogtest", 400, 8);
        Transaction tx = db.newTx();
        TableMgr tm = new TableMgr(true, tx);

        Schema sch = new Schema();
        sch.addIntField("A");
        sch.addStringField("B", 9);
        tm.createTable("MyTable", sch, tx);

        System.out.println("All tables and their lengths:");
        Layout layout = tm.getLayout("tblcat", tx);
        TableScan ts = new TableScan(tx, "tblcat", layout);
        while (ts.next()) {
            String tname = ts.getString("tblname");
            int size     = ts.getInt("slotsize");
            System.out.println(tname + " " + size);
        }
        ts.close();

        System.out.println("All fields and their offsets:");
        layout = tm.getLayout("fldcat", tx);
        ts = new TableScan(tx, "fldcat", layout);
        while (ts.next()) {
            String tname = ts.getString("tblname");
            String fname = ts.getString("fldname");
            int offset   = ts.getInt("offset");
            System.out.println(tname + " " + fname + " " + offset);
        }
        ts.close();
    }
}
```

[1] Обратите внимание, что константа «student» содержит только символы нижнего регистра, хотя имя таблицы состоит лишь из символов верхнего регистра. Причина в том, что все имена таблиц и полей в SimpleDB хранятся в нижнем регистре, а константы в операторах SQL чувствительны к регистру.

В листинге 7.5 приводится определение класса `TableMgr`. Конструктор создает схемы для таблиц каталога `tblcat` и `fldcat` и соответствующие им объекты `Layout`. Если база данных новая, он также создает две таблицы каталога.

Листинг 7.5. Определение класса TableMgr в SimpleDB

```
public class TableMgr {
    public static final int MAX_NAME = 16; // имя таблицы или поля
    private Layout tcatLayout, fcatLayout;

    public TableMgr(boolean isNew, Transaction tx) {
        Schema tcatSchema = new Schema();
        tcatSchema.addStringField("tblname", MAX_NAME);
        tcatSchema.addIntField("slotsize");
        tcatLayout = new Layout(tcatSchema);

        Schema fcatSchema = new Schema();
        fcatSchema.addStringField("tblname", MAX_NAME);
        fcatSchema.addStringField("fldname", MAX_NAME);
        fcatSchema.addIntField("type");
        fcatSchema.addIntField("length");
        fcatSchema.addIntField("offset");
        fcatLayout = new Layout(fcatSchema);

        if (isNew) {
            createTable("tblcat", tcatSchema, tx);
            createTable("fldcat", fcatSchema, tx);
        }
    }

    public void createTable(String tblname, Schema sch, Transaction tx) {
        Layout layout = new Layout(sch);

        // вставить одну запись в tblcat
        TableScan tcat = new TableScan(tx, "tblcat", tcatLayout);
        tcat.insert();
        tcat.setString("tblname", tblname);
        tcat.setInt("slotsize", layout.slotSize());
        tcat.close();

        // вставить одну запись в fldcat для каждого поля
        TableScan fcat = new TableScan(tx, "fldcat", fcatLayout);
        for (String fldname : sch.fields()) {
            fcat.insert();
            fcat.setString("tblname", tblname);
            fcat.setString("fldname", fldname);
            fcat.setInt   ("type", sch.type(fldname));
            fcat.setInt   ("length", sch.length(fldname));
            fcat.setInt   ("offset", layout.offset(fldname));
        }
        fcat.close();
    }
```

```
    public Layout getLayout(String tblname, Transaction tx) {
        int size = -1;
        TableScan tcat = new TableScan(tx, "tblcat", tcatLayout);
        while(tcat.next())
            if(tcat.getString("tblname").equals(tblname)) {
                size = tcat.getInt("slotsize");
                break;
            }

        tcat.close();

        Schema sch = new Schema();
        Map<String,Integer> offsets = new HashMap<String,Integer>();
        TableScan fcat = new TableScan(tx, "fldcat", fcatLayout);
        while(fcat.next())
            if(fcat.getString("tblname").equals(tblname)) {
                String fldname = fcat.getString("fldname");
                int fldtype    = fcat.getInt("type");
                int fldlen     = fcat.getInt("length");
                int offset     = fcat.getInt("offset");
                offsets.put(fldname, offset);
                sch.addField(fldname, fldtype, fldlen);
            }

        fcat.close();
        return new Layout(sch, offsets, size);
    }
}
```

Метод `createTable` вставляет записи в каталог с помощью механизма сканирования таблиц. Для каждой таблицы он вставляет одну запись в `tblcat` и для каждого поля – одну запись в `fldcat`.

Метод `getLayout` начинает сканирование двух таблиц каталога и проверяет наличие в них записей, соответствующих указанному имени таблицы. Затем на основе полученных записей он создает объект `Layout`.

7.3. Метаданные представлений

Представление – это таблица, записи которой вычисляются динамически с помощью запроса. Этот запрос называется *определением* представления и указывается при его создании. Диспетчер метаданных сохраняет определение каждого создаваемого представления и извлекает его, когда это необходимо.

Эту работу в SimpleDB выполняет класс `ViewMgr`. Класс сохраняет определения представлений в таблице каталога `viewcat`, создавая по одной записи для каждого представления. Таблица имеет следующие поля:

```
viewcat(ViewName, ViewDef)
```

Определение класса `ViewMgr` показано в листинге 7.6. Его конструктор вызывается во время запуска системы и, если база данных новая, создает таблицу `viewcat`. Методы `createView` и `getViewDef` используют механизм сканирования таблиц для доступа к таблице каталога: `createView` вставляет запись в таблицу, а `getViewDef` отыскивает в таблице запись, соответствующую указанному имени представления.

Листинг 7.6. Определение класса ViewMgr в SimpleDB

```
class ViewMgr {
    // максимальная длина определения представления в символах
    private static final int MAX_VIEWDEF = 100;
    TableMgr tblMgr;
    public ViewMgr(boolean isNew, TableMgr tblMgr, Transaction tx) {
        this.tblMgr = tblMgr;
        if (isNew) {
            Schema sch = new Schema();
            sch.addStringField("viewname", TableMgr.MAX_NAME);
            sch.addStringField("viewdef", MAX_VIEWDEF);
            tblMgr.createTable("viewcat", sch, tx);
        }
    }

    public void createView(String vname, String vdef,
                           Transaction tx) {
        Layout layout = tblMgr.getLayout("viewcat", tx);
        TableScan ts = new TableScan(tx, "viewcat", layout);
        ts.setString("viewname", vname);
        ts.setString("viewdef", vdef);
        ts.close();
    }

    public String getViewDef(String vname, Transaction tx) {
        String result = null;
        Layout layout = tblMgr.getLayout("viewcat", tx);
        TableScan ts = new TableScan(tx, "viewcat", layout);
        while (ts.next())
            if (ts.getString("viewname").equals(vname)) {
                result = ts.getString("viewdef");
                break;
            }
        ts.close();
        return result;
    }
}
```

Определения представлений хранятся в виде строк `varchar`, что накладывает ограничение на их длину. Текущее ограничение в 100 символов, конечно, очень непрактично, потому что определения представлений могут иметь длину в несколько тысяч символов. Разумнее было бы объявить поле `ViewDef` с типом `clob`, например `clob(9999)`.

7.4. Статистические метаданные

Другой разновидностью метаданных, поддерживаемых системами баз данных, является статистическая информация о каждой таблице, например количество записей в таблице и распределение значений полей. Эта информация используется планировщиком запросов для оценки затрат. Как показывает опыт, хороший набор статистик может значительно улучшить время выполнения запросов. По этой причине многие коммерческие диспетчеры метаданных

поддерживают сбор подробной и всеобъемлющей статистики, например гистограмму и диапазон значений для каждого поля в каждой таблице и информацию о корреляции между полями в разных таблицах.

Для простоты в этом разделе рассматриваются только три вида статистической информации:

- число блоков, использованных каждой таблицей T;
- число записей в каждой таблице T;
- количество уникальных значений в каждом поле F таблицы T.

Эти статистики обозначаются как *B(T)*, *R(T)* и *V(T, F)* соответственно.

В табл. 7.2 показан пример статистик для базы данных университета. Значения соответствуют университету, который принимает около 900 студентов в год и предлагает около 500 разделов учебных курсов в год. Эта информация была собрана за последние 50 лет. Значения для табл. 7.2 выбирались так, чтобы быть как можно ближе к реальности, и могут не соответствовать значениям, которые можно вычислить из данных на рис. 1.1. Здесь предполагается, что в каждый блок умещается 10 записей STUDENT, 20 записей DEPT и т. д.

Таблица 7.2. Пример статистик о базе данных университета

s	B(T)	R(T)	V(T,F)	
STUDENT	4 500	45 000	45 000	для F=SId
			44 960	для F=SName
			50	для F=GradYear
			40	для F=MajorId
DEPT	2	40	40	для F=DId, DName
COURSE	25	500	500	для F=CId, Title
			40	для F=DeptId
SECTION	2 500	25 000	25 000	для F=SectId
			500	для F=CourseId
			250	для F=Prof
			50	для F=YearOffered
ENROLL	50 000	1 500 000	1 500 000	для F=EId
			25 000	для F=SectionId
			45 000	для F=StudentId
			14	для F=Grade

Взгляните на значения V(T, F) для таблицы STUDENT. Поле `SId` является ключом в таблице STUDENT, поэтому V(STUDENT, `SId`) имеет значение 45 000. Значение 44 960 в V(STUDENT, `SName`) означает, что 40 из 45 000 студентов являются тезками. Значение 50 в V(STUDENT, `GradYear`) указывает, что по крайней мере один студент заканчивал обучение каждый год в течение последних 50 лет. А значение 40 в V(STUDENT, `MajorId`) означает, что каждая из 40 кафедр в какой-то момент выбиралась как основная.

В SimpleDB этой статистической информацией управляет класс `StatMgr`. Движок базы данных содержит один объект `StatMgr`. Этот объект имеет метод `getStatInfo`, который возвращает объект `StatInfo` для указанной таблицы. Объект `StatInfo` содержит статистику для этой таблицы и имеет методы `blocksAccessed`, `recordsOutput` и `distinctValues`, которые реализуют статистические функции B(T), R(T) и V(T, F) соответственно. API этих классов показаны в листинге 7.7.

Листинг 7.7. API классов поддержки статистик таблиц в SimpleDB

StatMgr

```
public StatMgr(TableMgr tm, Transaction tx);
public StatInfo getStatInfo(String tblname, Layout lo,
 Transaction tx);
```

StatInfo

```
public int blocksAccessed();
public int recordsOutput();
public int distinctValues(String fldname);
```

Фрагмент кода в листинге 7.8 иллюстрирует типичное использование данных методов. Этот код получает статистики для таблицы STUDENT и выводит значения B(STUDENT), R(STUDENT) и V(STUDENT, `MajorId`).

Листинг 7.8. Получение и вывод статистик о таблице

```
SimpleDB db = ...
Transaction tx = db.newTx();
TableMgr tblmgr = ...
StatMgr statmgr = new StatMgr(tblmgr, tx);
Layout layout = tblmgr.getLayout("student", tx);
StatInfo si = statmgr.getStatInfo("student", layout, tx);
System.out.println(si.blocksAccessed() + " " +
                   si.recordsOutput() + " " +
                   si.distinctValues("majorid"));
tx.commit();
```

Движок базы данных может управлять статистическими метаданными одним из двух способов. Первый: хранить информацию в каталоге и обновлять ее после каждого изменения базы данных. Второй: хранить информацию в памяти, вычисляя ее на этапе запуска движка.

Для первого подхода необходимо создать две новые таблицы в каталоге, `tblstats` и `fldstats`, со следующими полями:

```
tblstats(TblName, NumBlocks, NumRecords)
fldstats(TblName, FldName, NumValues)
```

Таблица `tblstats` будет хранить одну запись для каждой таблицы T со значениями B(T) и R(T), а таблица `fldstats` – одну запись для каждого поля F каждой таблицы T со значением V(T, F). Проблема этого подхода заключается в стоимости поддержания статистики в актуальном состоянии. Каждый вызов `insert`, `delete`, `setInt` и `setString` может потребовать обновления этих таблиц. Для записи измененных страниц потребуются дополнительные обращения к диску. Более того, степень параллелизма уменьшится – каждое обновление таблицы T будет блокировать в монопольном режиме блоки со статистическими записями для T, что заставит ждать транзакции, читающие статистику T (а также статистику других таблиц, имеющих записи на тех же страницах).

Одно из решений этой проблемы – позволить транзакциям читать статистику без установки разделяемых блокировок, как на уровне изоляции read uncommitted (чтение неподтвержденных данных), описанном в разделе 5.4.7. Потеря точности в данном случае допустима, потому что система баз данных

использует статистику для сравнения предполагаемого времени выполнения планов запросов. Поэтому статистические данные не обязательно должны быть точными, достаточно, чтобы оценки были приемлемыми.

Вторая стратегия реализации статистик не требует создания таблиц в каталоге и предполагает хранение статистик непосредственно в памяти. Статистические данные имеют относительно небольшой объем и должны легко умещаться в оперативной памяти. Единственная проблема заключается в том, что статистику нужно вычислять заново при каждом запуске сервера. Чтобы выполнить такие вычисления, нужно просканировать каждую таблицу в базе данных и подсчитать количество записей, блоков и значений. Если база данных не слишком велика, такие вычисления не будут слишком сильно задерживать запуск системы.

Стратегия хранения статистик в оперативной памяти предполагает два варианта обработки обновлений базы данных. Первый вариант: обновлять статистику при каждом обновлении базы данных, как и раньше. Второй: не обновлять статистику, но периодически пересчитывать ее с самого начала. Второй вариант основан на том факте, что движку не требуется точная статистическая информация, и поэтому допустимо, чтобы она немного устаревала в какие-то периоды времени.

В SimpleDB реализован второй вариант второго подхода. Класс `StatMgr` имеет переменную `tableStats`, хранящую статистики для каждой таблицы. Также в классе имеется публичный метод `statInfo`, возвращающий статистики для указанной таблицы, и приватные методы `refreshStatistics` и `refreshTableStats`, пересчитывающие статистики. Определение класса `StatMgr` показано в листинге 7.9.

Листинг 7.9. Определение класса StatMgr в SimpleDB

```
class StatMgr {
    private TableMgr tblMgr;
    private Map<String,StatInfo> tablestats;
    private int numcalls;

    public StatMgr(TableMgr tblMgr, Transaction tx) {
        this.tblMgr = tblMgr;
        refreshStatistics(tx);
    }

    public synchronized StatInfo getStatInfo(String tblname,
                                Layout layout, Transaction tx) {
        numcalls++;

        if (numcalls > 100)
            refreshStatistics(tx);
        StatInfo si = tablestats.get(tblname);
        if (si == null) {
            si = calcTableStats(tblname, layout, tx);
            tablestats.put(tblname, si);
        }
        return si;
    }
```

```
    private synchronized void refreshStatistics(Transaction tx) {
        tablestats = new HashMap<String,StatInfo>();
        numcalls = 0;
        Layout tcatlayout = tblMgr.getLayout("tblcat", tx);
        TableScan tcat = new TableScan(tx, "tblcat", tcatlayout);
        while(tcat.next()) {
            String tblname = tcat.getString("tblname");
            Layout layout = tblMgr.getLayout(tblname, tx);
            StatInfo si = calcTableStats(tblname, layout, tx);
            tablestats.put(tblname, si);
        }
        tcat.close();
    }

    private synchronized StatInfo calcTableStats(String tblname,
                        Layout layout, Transaction tx) {
        int numRecs = 0;
        int numblocks = 0;
        TableScan ts = new TableScan(tx, tblname, layout);
        while (ts.next()) {
            numRecs++;
            numblocks = ts.getRid().blockNumber() + 1;
        }
        ts.close();
        return new StatInfo(numblocks, numRecs);
    }
}
```

Класс `StatMgr` хранит счетчик, который увеличивается при каждом вызове `statInfo`. Когда счетчик достигает определенного значения (в данном случае 100), вызывается `refreshStatistics` для пересчета статистик всех таблиц. Если `statInfo` обнаружит, что для указанной таблицы нет известных значений, он вызовет `refreshTableStats`, чтобы вычислить статистики для этой таблицы.

Метод `refreshStatistics` перебирает записи в таблице `tblcat`, извлекая из каждой имя таблицы и вызывая `refreshTableStats` для вычисления статистик для этой таблицы. Метод `refreshTableStats` просматривает содержимое таблицы, подсчитывает записи и вызывает `size`, чтобы определить количество используемых блоков. Для простоты метод не подсчитывает значения полей. Вместо этого объект `StatInfo` пытается угадать количество уникальных значений для поля, исходя из количества записей в таблице.

Определение класса `StatInfo` показано в листинге 7.10. Обратите внимание, что `distinctValues` не использует имя поля из аргумента, так как наивно предполагает, что примерно 1/3 значений в любом поле уникальны. Излишне говорить, что это предположение не имеет ничего общего с действительностью. В упражнении 7.12 вам будет предложено исправить ситуацию.

Листинг 7.10. Определение класса StatInfo в SimpleDB

```
public class StatInfo {
    private int numBlocks;
    private int numRecs;

    public StatInfo(int numblocks, int numrecs) {
        this.numBlocks = numblocks;
        this.numRecs = numrecs;
    }
```

```
    public int blocksAccessed() {
        return numBlocks;
    }

    public int recordsOutput() {
        return numRecs;
    }

    public int distinctValues(String fldname) {
        return 1 + (numRecs / 3); // пальцем в небо
    }
}
```

7.5. Метаданные индексов

Метаданные индексов включают имя индекса, имя индексируемой таблицы и список индексированных полей. *Диспетчер индексов* – это компонент движка, хранящий и извлекающий эти метаданные. В SimpleDB он состоит из двух классов: `IndexMgr` и `IndexInfo`. Их API показан в листинге 7.11.

Листинг 7.11. API диспетчера индексов в SimpleDB

IndexMgr

```
public IndexMgr(boolean isnew, TableMgr tmgr, StatMgr smgr,
 Transaction tx);
public createIndex(String iname, String tname, String fname,
 Transaction tx);
public Map(String,IndexInfo> getIndexInfo(String tblname,
 Transaction tx);
```

IndexInfo

```
public IndexInfo(String iname, String tname, String fname,
 Transaction tx);
public int blocksAccessed();
public int recordsOutput();
public int distinctValues(String fldname);
public Index open();
```

Метод `createIndex` класса `IndexMgr` сохраняет эти метаданные в каталоге. Метод `getIndexInfo` извлекает метаданные для всех индексов указанной таблицы. В частности, он возвращает ассоциативный массив объектов `Indexinfo`, роль ключей в котором играют имена индексированных полей. Вызовом метода `keyset` ассоциативного массива можно узнать, по каким полям таблицы созданы индексы. Методы класса `IndexInfo` возвращают статистическую информацию о выбранном индексе, аналогично методам класса `StatInfo`. Метод `blocksAccessed` возвращает количество обращений к блокам, необходимых для поиска в индексе (а не размер индекса). Методы `recordsOutput` и `distinctValues` возвращают количество записей в индексе и количество уникальных значений в индексированном поле, которые совпадают с аналогичными статистиками данной таблицы.

Объект `IndexInfo` также имеет метод `open`, возвращающий объект `Index` для индекса. Класс `Index` определяет методы для поиска в индексе и обсуждается в главе 12.

Фрагмент кода в листинге 7.12 иллюстрирует использование этих методов. Код в этом примере создает два индекса для таблицы STUDENT, а затем извлекает их метаданные и для каждого выводит имя и стоимость поиска.

Листинг 7.12. Использование диспетчера индексов в SimpleDB

```
SimpleDB db = ...
Transaction tx = db.newTx();
TableMgr tblmgr = ...
StatMgr statmgr = new StatMgr(tblmgr, tx);
IndexMgr idxmgr = new IndexMgr(true, tblmgr, statmgr, tx);
idxmgr.createIndex("sidIdx", "student", "sid");
idxmgr.createIndex("snameIdx", "student", "sname");

Map<String,IndexInfo> indexes = idxmgr.getIndexInfo("student", tx);
for (String fldname : indexes.keySet()) {
    IndexInfo ii = indexes.get(fldname);
    System.out.println(fldname + "\t" + ii.blocksAccessed(fldname));
}
```

В листинге 7.13 показано определение класса `IndexMgr`. Он хранит метаданные индексов в таблице каталога `idxcat`. Для каждого индекса в этой таблице имеется отдельная запись с тремя полями: именем индекса, именем индексируемой таблицы и именем индексированного поля.

Листинг 7.13. Определение класса диспетчера индексов SimpleDB

```
public class IndexMgr {
    private Layout layout;
    private TableMgr tblmgr;
    private StatMgr statmgr;

    public IndexMgr(boolean isnew, TableMgr tblmgr, StatMgr statmgr, Transaction tx) {
        if (isnew) {
            Schema sch = new Schema();
            sch.addStringField("indexname", MAX_NAME);
            sch.addStringField("tablename", MAX_NAME);
            sch.addStringField("fieldname", MAX_NAME);
            tblmgr.createTable("idxcat", sch, tx);
        }
        this.tblmgr = tblmgr;
        this.statmgr = statmgr;
        layout = tblmgr.getLayout("idxcat", tx);
    }

    public void createIndex(String idxname, String tblname, String fldname, Transaction tx) {
        TableScan ts = new TableScan(tx, "idxcat", layout);
        ts.insert();
        ts.setString("indexname", idxname);
        ts.setString("tablename", tblname);
        ts.setString("fieldname", fldname);
        ts.close();
    }
```

```
    public Map<String,IndexInfo> getIndexInfo(String tblname, Transaction tx) {
        Map<String,IndexInfo> result = new HashMap<String,IndexInfo>();
        TableScan ts = new TableScan(tx, "idxcat", layout);
        while (ts.next())
            if (ts.getString("tablename").equals(tblname)) {
                String idxname = ts.getString("indexname");
                String fldname = ts.getString("fieldname");
                Layout tblLayout = tblmgr.getLayout(tblname, tx);
                StatInfo tblsi = statmgr.getStatInfo(tblname, tbllayout, tx);
                IndexInfo ii = new IndexInfo(idxname, fldname,
                                             tblLayout.schema(), tx, tblsi);
                result.put(fldname, ii);
            }
        ts.close();
        return result;
    }
}
```

Конструктор вызывается во время запуска системы и создает таблицу каталога idxcat, если она отсутствует. Методы `createIndex` и `getIndexInfo` имеют простую реализацию. Оба создают объект `TableScan`, и с его помощью метод `createIndex` вставляет новую запись, а метод `getIndexInfo` отыскивает записи с указанным именем таблицы и добавляет их в ассоциативный массив.

В листинге 7.14 показано определение класса `IndexInfo`. Конструктор принимает имя индекса и индексированного поля, а также переменные, содержащие компоновку и статистические метаданные соответствующей таблицы. Эти метаданные позволяют объекту `IndexInfo` создать схему для индексной записи и оценить размер файла индекса.

Листинг 7.14. Определение класса IndexInfo в SimpleDB

```
public class IndexInfo {
    private String idxname, fldname;
    private Transaction tx;
    private Schema tblSchema;
    private Layout idxLayout;
    private StatInfo si;

    public IndexInfo(String idxname, String fldname, Schema tblSchema,
                     Transaction tx, StatInfo si) {
        this.idxname = idxname;
        this.fldname = fldname;
        this.tx = tx;
        this.idxLayout = createIdxLayout();
        this.si = si;
    }

    public Index open() {
        Schema sch = schema();
        return new HashIndex(tx, idxname, idxLayout);
//      return new BTreeIndex(tx, idxname, idxLayout);
    }
```

```
    public int blocksAccessed() {
        int rpb = tx.blockSize() / idxLayout.slotSize();
        int numblocks = si.recordsOutput() / rpb;
        return HashIndex.searchCost(numblocks, rpb);
//      return BTreeIndex.searchCost(numblocks, rpb);
    }

    public int recordsOutput() {
        return si.recordsOutput() / si.distinctValues(fldname);
    }

    public int distinctValues(String fname) {
        return fldname.equals(fname) ? 1 : si.distinctValues(fldname);
    }

    private Layout createIdxLayout() {
        Schema sch = new Schema();
        sch.addIntField("block");
        sch.addIntField("id");
        if (layout.schema().type(fldname) == INTEGER)
            sch.addIntField("dataval");
        else {
            int fldlen = layout.schema().length(fldname);
            sch.addStringField("dataval", fldlen);
        }
        return new Layout(sch);
    }
}
```

Метод `open` открывает индекс, передавая имя индекса и схему конструктору `HashIndex`. Класс `HashIndex` реализует статический хеш-индекс и обсуждается в главе 12. Чтобы вместо него использовать индекс на основе B-деревьев, нужно заменить конструктор, выделенный жирным, закомментированным кодом. Метод `blocksAccessed` оценивает стоимость поиска по индексу. Сначала он использует информацию из объекта компоновки `Layout`, чтобы определить длину каждой индексной записи и оценить количество записей в блоке (records per block, RPB) и размер файла индекса. Затем вызывает метод `searchCost`, чтобы вычислить количество обращений к блоку для индекса этого типа. Метод `recordsOutput` оценивает количество индексных записей, соответствующих ключу поиска. И метод `distinctValues` возвращает то же количество уникальных значений, что и в индексируемой таблице.

7.6. Реализация диспетчера метаданных

SimpleDB упрощает интерфейс доступа к диспетчеру метаданных со стороны клиентов, скрывая четыре отдельных класса, составляющих реализацию диспетчера: `TableMgr`, `ViewMgr`, `StatMgr` и `IndexMgr`. Все клиенты должны использовать общий класс `MetadataMgr` в роли единственного источника метаданных. API класса `MetadataMgr` показан в листинге 7.15.

Листинг 7.15. API диспетчера метаданных в SimpleDB

MetadataMgr

```
public void createTable(String tblname, Schema sch, Transaction tx);
public Layout getLayout(String tblname, Transaction tx);

public void createView(String viewname, String viewdef, Transaction tx);
public String getViewDef(String viewname, Transaction tx);
public void createIndex(String idxname, String tblname, String fldname, Transaction tx);
public Map<String,IndexInfo> getIndexinfo(String tblname, Transaction tx);
public StatInfo getStatInfo(String tblname, Layout layout, Transaction tx);
```

API содержит по два метода для каждого типа метаданных: один метод генерирует и сохраняет метаданные, а другой извлекает их. Единственное исключение – статистические метаданные, метод создания которых вызывается только реализацией и поэтому объявлен приватным.

В листинге 7.16 показан класс `MetadataMgrTest`, иллюстрирующий использование этих методов.

Часть 1 в листинге 7.16 иллюстрирует работу с метаданными таблицы. Здесь создается таблица MyTable и выводится ее компоновка, как показано в листинге 7.2. В части 2 демонстрируется использование диспетчера статистики. Ее код вставляет несколько записей в MyTable и выводит итоговую статистику таблицы. В части 3 показан пример взаимодействия с диспетчером представлений: создание представления и получение его определения. В части 4 иллюстрируется работа с диспетчером индексов. Здесь код создает индексы для полей A и B и выводит свойства каждого индекса.

Листинг 7.16. Тестирование методов MetadataMgr

```
public class MetadataMgrTest {
    public static void main(String[] args) throws Exception {
        SimpleDB db = new SimpleDB("metadatamgrtest", 400, 8);
        Transaction tx = db.newTx();
        MetadataMgr mdm = new MetadataMgr(true, tx);

        Schema sch = new Schema();
        sch.addIntField("A");
        sch.addStringField("B", 9);

        // Часть 1: Метаданные таблиц
        mdm.createTable("MyTable", sch, tx);
        Layout layout = mdm.getLayout("MyTable", tx);
        int size = layout.slotSize();
        Schema sch2 = layout.schema();
        System.out.println("MyTable has slot size " + size);
        System.out.println("Its fields are:");
        for (String fldname : sch2.fields()) {
            String type;
            if (sch2.type(fldname) == INTEGER)
                type = "int";
            else {
                int strlen = sch2.length(fldname);
                type = "varchar(" + strlen + ")";
            }
            System.out.println(fldname + ": " + type);
        }
```

```
        // Часть 2: Статистические метаданные
        TableScan ts = new TableScan(tx, "MyTable", layout);
        for (int i=0; i<50; i++) {
            ts.insert();
            int n = (int) Math.round(Math.random() * 50);
            ts.setInt("A", n);
            ts.setString("B", "rec"+n);
        }
        StatInfo si = mdm.getStatInfo("MyTable", layout, tx);
        System.out.println("B(MyTable) = " + si.blocksAccessed());
        System.out.println("R(MyTable) = " + si.recordsOutput());
        System.out.println("V(MyTable,A) = " + si.distinctValues("A"));
        System.out.println("V(MyTable,B) = " + si.distinctValues("B"));

        // Часть 3: Метаданные представлений
        String viewdef = "select B from MyTable where A = 1";
        mdm.createView("viewA", viewdef, tx);
        String v = mdm.getViewDef("viewA", tx);
        System.out.println("View def = " + v);

        // Часть 4: Метаданные индексов
        mdm.createIndex("indexA", "MyTable", "A", tx);
        mdm.createIndex("indexB", "MyTable", "B", tx);
        Map<String,IndexInfo> idxmap = mdm.getIndexInfo("MyTable", tx);

        IndexInfo ii = idxmap.get("A");
        System.out.println("B(indexA) = " + ii.blocksAccessed());
        System.out.println("R(indexA) = " + ii.recordsOutput());
        System.out.println("V(indexA,A) = " + ii.distinctValues("A"));
        System.out.println("V(indexA,B) = " + ii.distinctValues("B"));

        ii = idxmap.get("B");
        System.out.println("B(indexB) = " + ii.blocksAccessed());
        System.out.println("R(indexB) = " + ii.recordsOutput());
        System.out.println("V(indexB,A) = " + ii.distinctValues("A"));
        System.out.println("V(indexB,B) = " + ii.distinctValues("B"));
        tx.commit();
    }
}
```

Классы, подобные классу `MetadataMgr`, называют *фасадными классами*. Конструктор этого класса создает четыре объекта диспетчеров и сохраняет их в приватных переменных. Методы класса `MetadataMgr` повторяют публичные методы отдельных диспетчеров. Когда клиент вызывает метод диспетчера метаданных, тот, в свою очередь, делегирует выполнение работы соответствующему локальному диспетчеру. Определение `MetadataMgr` приводится в листинге 7.17.

Листинг 7.17. Определение класса MetadataMgr в SimpleDB

```
public class MetadataMgr {
    private static TableMgr  tblmgr;
    private static ViewMgr   viewmgr;
    private static StatMgr   statmgr;
    private static IndexMgr  idxmgr;

    public MetadataMgr(boolean isnew, Transaction tx) {
        tblmgr  = new TableMgr(isnew, tx);
        viewmgr = new ViewMgr(isnew, tblmgr, tx);
        statmgr = new StatMgr(tblmgr, tx);
        idxmgr  = new IndexMgr(isnew, tblmgr, statmgr, tx);
    }
    public void createTable(String tblname, Schema sch, Transaction tx) {
        tblmgr.createTable(tblname, sch, tx);
    }

    public Layout getLayout(String tblname, Transaction tx) {
        return tblmgr.getLayout(tblname, tx);
    }

    public void createView(String viewname, String viewdef, Transaction tx) {
        viewmgr.createView(viewname, viewdef, tx);
    }

    public String getViewDef(String viewname, Transaction tx) {
        return viewmgr.getViewDef(viewname, tx);
    }

    public void createIndex(String idxname, String tblname,
                            String fldname, Transaction tx) {
        idxmgr.createIndex(idxname, tblname, fldname, tx);
    }

    public Map<String,IndexInfo> getIndexInfo(String tblname, Transaction tx){
       return idxmgr.getIndexInfo(tblname, tx);
    }

    public StatInfo getStatInfo(String tblname, Layout layout, Transaction tx){
        return statmgr.getStatInfo(tblname, layout, tx);
    }
}
```

Все тестовые программы, представленные выше в этой книге, вызывали конструктор SimpleDB с тремя аргументами. Этот конструктор использует указанный размер блока и размер пула буферов для настройки системных объектов `FileMgr`, `LogMgr` и `BufferMgr`. Его цель – помочь отладить низкоуровневые механизмы системы без создания объекта `MetadataMgr`.

В SimpleDB имеется еще один конструктор, принимающий один аргумент – имя базы данных. Этот конструктор используется для нормальной работы. Он сначала создает объекты диспетчеров файлов, журнала и буферов, используя значения по умолчанию. Затем вызывает диспетчера восстановления для восстановления базы данных (если это необходимо) и создает объект диспетчера метаданных (который, для новой базы данных, создает таблицы каталога). Код обоих конструкторов SimpleDB показан в листинге 7.18.

Листинг 7.18. Два конструктора SimpleDB

```
public SimpleDB(String dirname, int blocksize, int buffsize) {
    String homedir = System.getProperty(HOME_DIR);
    File dbDirectory = new File(homedir, dirname);
    fm = new FileMgr(dbDirectory, blocksize);
    lm = new LogMgr(fm, LOG_FILE);
    bm = new BufferMgr(fm, lm, buffsize);
}

public SimpleDB(String dirname) {
    this(dirname, BLOCK_SIZE, BUFFER_SIZE);
    Transaction tx = new Transaction(fm, lm, bm);
    boolean isnew = fm.isNew();
    if (isnew)
        System.out.println("creating new database");
    else {
        System.out.println("recovering existing database");
        tx.recover();
    }
    mdm = new MetadataMgr(isnew, tx);
    tx.commit();
}
```

Используя конструктор с одним аргументом, код `MetadataMgrTest` в листинге 7.16 можно упростить, как показано в листинге 7.19.

Листинг 7.19. Использование конструктора SimpleDB с одним аргументом

```
public class MetadataMgrTest {
    public static void main(String[] args) throws Exception {
        SimpleDB db = new SimpleDB("metadatamgrtest");
        MetadataMgr mdm = db.mdMgr();
        Transaction tx = db.newTx();
        ...
}
```

7.7. Итоги

- *Метаданные* – это информация о базе данных в дополнение к ее содержимому. *Диспетчер метаданных* – это компонент системы баз данных, хранящий и извлекающий ее метаданные.
- Метаданные в SimpleDB делятся на четыре категории:
  - метаданные *таблиц* – описывают структуру записей в таблицах: длину, тип и смещение каждого поля;
  - метаданные *представлений* – описывают свойства каждого представления: определение и имя пользователя, создавшего представление;
  - метаданные *индексов* – описывают индексы, определенные для таблиц;
  - *статистические* метаданные – описывают размеры таблиц и распределение значений их полей.
- Диспетчер метаданных хранит свои метаданные в *системном каталоге*. Каталог часто реализуется в виде таблиц в базе данных, называемых *таблицами каталога*. К таблицам каталога можно обращаться точно так же, как к любым другим таблицам в базе данных.

- Метаданные таблиц могут храниться в двух таблицах каталога: с информацией о таблицах (например, размер слота) и с информацией о полях (например, имя, длина и тип каждого поля).
- Метаданные представлений состоят в основном из определений и могут храниться в отдельной таблице каталога. Определение представления – это произвольная строка, поэтому для ее хранения уместно использовать поле переменной длины.
- Статистические метаданные содержат информацию о размере таблиц и распределении значений полей в каждой из них. Коммерческие системы баз данных обычно поддерживают подробную и всеобъемлющую статистику, например гистограмму и диапазон значений для каждого поля в каждой таблице и информацию о корреляции между полями в разных таблицах.
- Базовый набор статистик включает три функции:
  - B(T) возвращает количество блоков в таблице T;
  - R(T) возвращает количество записей в таблице T;
  - V(T, F) возвращает количество уникальных значений в поле F таблицы T.
- Статистики могут храниться в таблицах каталога или вычисляться заново при каждом запуске базы данных. Первый вариант позволяет сократить время запуска, но может замедлить выполнение транзакций.
- Метаданные индекса содержат имена всех индексов, а также соответствующих им таблиц и полей.

7.8. Для дополнительного чтения

Таблицы каталога, используемые в SimpleDB, устроены очень просто, имеют небольшой объем и похожи на аналогичные таблицы в ранней системе INGRES (Stonebraker et al. 1976). С другой стороны, в Oracle в настоящее время имеется настолько обширный каталог, что для его описания потребовалась 60-страничная книга (Kreines 2003).

Стандарт SQL определяет набор представлений для доступа к метаданным. Эти представления называются *информационной схемой* базы данных. Существует более 50 определенных представлений, которые дополняют метаданные, описанные в данной главе. Например, есть представления для отображения информации о триггерах, утверждениях (assertions), ограничениях целостности, пользовательских типах и т. д. Есть также несколько представлений с информацией о привилегиях и ролях. Идея состоит в том, что каждая система базы данных может хранить эти метаданные своим уникальным способом, но она обязана предоставить стандартный интерфейс для доступа к ним. Подробности можно найти в главе 16 в Gulutzan and Pelzer (1999).

Точные и подробные статистические метаданные имеют большое значение для оптимального планирования запросов. В этой главе описан грубый, упрощенный подход, но коммерческие системы используют гораздо более сложные методы. В статье Gibbons et al. (2002) описывается использование гистограмм и показано, как можно эффективно поддерживать их в условиях частых изменений. Информацию для гистограмм можно определить разными способами, наиболее интересным из которых является использование вейвлет-методов (Matias et al. 1998). Также можно собрать статистику по ранее выполненным запросам и использовать ее для планирования похожих запросов (Bruno and Chaudhuri 2004).

Bruno, N., & Chaudhuri, S. (2004). «Conditional selectivity for statistics on query expressions». In Proceedings of the ACM SIGMOD Conference (p. 311–322).

Gibbons, P., Matias, Y., & Poosala, V. (2002). «Fast incremental maintenance of incremental histograms». ACM Transactions on Database Systems, 27 (3), 261–298.

Gulutzan, P., & Pelzer, T. (1999). «SQL-99 complete, really». Lawrence, KA: R&D Books.

Kreines, D. (2003). «Oracle data dictionary pocket reference». Sebastopol, CA: O'Reilly.

Matias, Y., Vitter, J., & Wang, M. (1998). «Wavelet-based histograms for selectivity estimation». In Proceedings of the ACM SIGMOD Conference (p. 448–459).

Stonebraker, M., Kreps, P., Wong, E., & Held, G. (1976). «The design and implementation of INGRES». ACM Transactions on Database Systems, 1 (3), 189–222.

7.9. Упражнения

Теория

7.1. Приведите записи в `tblcat` и `fldcat`, которые SimpleDB создаст для таблиц `tblcat` и `fldcat`. (*Подсказка*: изучите определение класса `TableMgr`.)

7.2. Предположим, что выполняются две транзакции, T1 и T2. Транзакция T1 создает таблицу X, а транзакция T2 – таблицу Y.

a) Какие возможные конкурентные расписания могут иметь эти транзакции?

b) Могут ли T1 и T2 оказаться в состоянии взаимоблокировки? Объясните.

7.3. Стандарт SQL позволяет клиенту добавить новое поле в существующую таблицу. Приведите хороший алгоритм реализации этой возможности.

Практика

7.4. Стандарт SQL позволяет клиенту удалить поле из существующей таблицы. Предположим, что эта возможность реализована в методе `removeField` класса `TableMgr`.

a) Один из вариантов реализации этого метода – просто изменить запись в `fldcat` для поля, записав пустую строку в поле `FldName`. Напишите код данного метода.

b) Пункт «a» не предполагает изменения записей в самой таблице. Что произойдет со значениями удаленных полей? Почему к ним невозможно получить доступ?

c) Другой вариант реализации этого метода – удалить запись для поля из `fldcat` и изменить все существующие записи с данными в таблице. Это потребует значительно больше работы, чем в пункте «a». Стоит ли оно того? Объясните достоинства и недостатки этого варианта.

7.5. В таблицах каталога SimpleDB поле `tblname` является ключом таблицы `tblcat`, а поле `tblname` – соответствующим внешним ключом таблицы `fldcat`. Также можно было бы использовать суррогатный ключ (скажем, `tblId`) в `tblcat` и соответствующий внешний ключ в `fldcat` (скажем, с именем `tableId`).

a) Реализуйте это решение в SimpleDB.

b) Это решение лучше оригинального? (Экономит ли оно место или уменьшает число обращений к диску?)

7.6. Предположим, что в работе SimpleDB произошел сбой во время создания таблиц каталога для новой базы данных.
   a) Опишите, что произойдет во время процесса восстановления базы данных после перезапуска системы. Какая проблема возникнет?
   b) Измените код SimpleDB, чтобы исправить проблему.

7.7. Напишите клиента для SimpleDB, выполняющего каждую из следующих задач путем непосредственного обращения к таблицам `tblcat` и `fldcat`:
   a) вывести имена полей всех таблиц в базе данных (например, в формате: «T(A, B)»);
   b) воссоздать и вывести текст SQL-оператора `create table`, создающего конкретную таблицу (например, в виде: «create table T (A integer, B varchar (7))»).

7.8. Что случится, если вызвать метод `getLayout` с несуществующим именем таблицы? Исправьте код так, чтобы в этом случае возвращалось значение `null`.

7.9. Какая проблема может возникнуть, если клиент попытается создать таблицу с именем, совпадающим с именем таблицы, уже имеющейся в каталоге? Измените код так, чтобы предотвратить эту проблему.

7.10. Добавьте в класс `TableMgr` метод `dropTable`, удаляющий таблицу из базы данных. Нужно ли при этом изменить реализацию диспетчера файлов?

7.11. Измените код SimpleDB так, чтобы статистика сохранялась в таблицах каталога и обновлялась при каждом изменении базы данных.

7.12. Измените код SimpleDB так, чтобы функция V(T, F) вычислялась для каждой таблицы T и поля F. (*Подсказка*: для подсчета уникальных значений в каждом поле может потребоваться значительный объем памяти, потому что число таких значений может быть неограниченным. Целесообразнее выполнить подсчет значений для части таблицы и затем применить экстраполяцию. Например, можно посчитать количество записей, необходимых для получения 1000 разных значений.)

7.13. Предположим, что клиент создал таблицу, вставил в нее несколько записей, а затем выполнил откат.
   a) Что произойдет с метаданными таблицы в каталоге?
   b) Что произойдет с файлом, содержащим данные? Объясните, какая проблема может возникнуть, если клиент впоследствии создаст таблицу с тем же именем, но с другой схемой.
   c) Исправьте код SimpleDB так, чтобы решить эту проблему.

7.14. Измените диспетчер индексов так, чтобы он также сохранял в каталоге тип индекса. Предположим, что есть два типа индексов, реализованных в виде классов `BTreeIndex` и `HashIndex`. Конструкторы и статический метод `searchCost` обоих классов должны принимать одинаковые аргументы.

7.15. Диспетчер индексов в SimpleDB хранит необходимую ему информацию в таблице `idxcat`. Эту информацию также можно хранить в таблице каталога `fldcat`.
   a) Сравните эти два подхода. Какие преимущества имеет каждый из них?
   b) Реализуйте данный альтернативный подход.

Глава 8

Обработка запросов

В следующих трех главах рассказывается, как движки баз данных обрабатывают запросы SQL. Дело в том, что SQL-запрос описывает, какие данные должен вернуть движок, а не как их получить. По этой причине движок должен реализовать набор операторов извлечения данных, известных как *реляционная алгебра*. Движок может преобразовать запрос SQL в запрос реляционной алгебры и затем выполнить его. В этой главе рассматриваются запросы реляционной алгебры и их реализация. В двух следующих главах будет описан порядок преобразования SQL-запросов в запросы реляционной алгебры.

8.1. Реляционная алгебра

Реляционная алгебра состоит из набора *операторов*. Каждый оператор решает одну конкретную задачу, получая одну или несколько таблиц на входе и порождая одну таблицу на выходе. Комбинируя эти операторы, можно конструировать очень сложные запросы.

Версию SQL, используемую в SimpleDB, можно реализовать с применением трех операторов:

- *оператор селекции (фильтрации) select* возвращает таблицу, содержащую те же столбцы, что и входная таблица, но убирает некоторые строки;
- *оператор проекции project* возвращает таблицу, содержащую те же строки, что и входная таблица, но убирает некоторые столбцы;
- *оператор прямого (декартова) произведения product* возвращает таблицу, содержащую все возможные комбинации записей из двух входных таблиц.

Все эти операторы подробно рассматриваются в следующих подразделах.

8.1.1. Селекция

Оператор *select* принимает два аргумента: входную таблицу и предикат (условие). Выходная таблица включает все записи из входной таблицы, удовлетворяющие предикату. Запрос select всегда возвращает таблицу, имеющую ту же схему, что и входная таблица, но содержащую лишь подмножество записей.

Например, вот запрос Q1, возвращающий таблицу со списком студентов, закончивших обучение в 2019 году.

```
Q1 = select(STUDENT, GradYear=2019)
```

Предикатом может быть любое логическое выражение, соответствующее предложению where в SQL. Например, вот запрос Q2, который отыскивает студентов, закончивших учебу в 2019 году по специальности 10 или 20.

```
Q2 = select(STUDENT, GradYear=2019 and (MajorId=10 or MajorId=20))
```

Выходная таблица одного запроса может служить входной для другого запроса. Например, вот запросы Q3 и Q4, каждый из которых эквивалентен запросу Q2:

```
Q3 = select(select(STUDENT, GradYear=2019), MajorId=10 or MajorId=20)
Q4 = select(Q1, MajorId=10 or MajorId=20)
```

В Q3 первым аргументом самого внешнего запроса является другой запрос, идентичный Q1, который отыскивает студентов, окончивших обучение в 2019 году. Внешний запрос извлекает из его результатов записи со студентами, закончившими обучение по специальности 10 или 20. Запрос Q4 действует аналогично, но вместо определения внутреннего запроса использует имя Q1.

Запрос реляционной алгебры можно представить графически в виде *дерева запроса*. Дерево запроса содержит узлы для всех таблиц и операторов, упомянутых в запросе. Узлы таблиц, в отличие от узлов операторов, являются листовыми узлами дерева. Узел оператора имеет дочерний узел для каждой из его входных таблиц. Например, на рис. 8.1 показано дерево запроса для Q3.

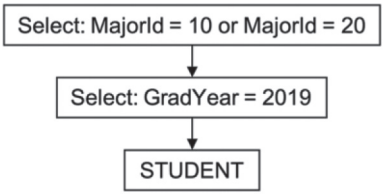


Рис. 8.1. Дерево запроса для Q3

8.1.2. Проекция

Оператор *project* принимает два аргумента: входную таблицу и набор имен полей. Выходная таблица включает все записи из входной таблицы, но ее схема содержит только указанные поля. Например, вот запрос Q5, возвращающий имена и годы выпуска всех студентов:

```
Q5 = project(STUDENT, {SName, GradYear})
```

Запросы могут включать оба оператора, project и select. Вот запрос Q6, возвращающий таблицу с именами всех студентов, обучающихся по специальности 10:

```
Q6 = project(select(STUDENT, MajorId=10), {SName})
```

Дерево запроса для Q6 показано на рис. 8.2.

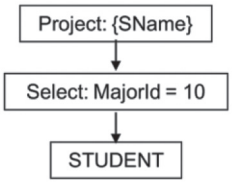


Рис. 8.2. Дерево запроса для Q6

Выходная таблица запроса project может содержать повторяющиеся записи. Например, если в исходной таблице имеется три записи с именем студента «pat», и все три тезки обучались по специальности 10, результат запроса Q6 будет содержать три записи с именем «pat».

Не все комбинации операторов имеют смысл. Например, вот запрос, полученный перестановкой операторов в запросе Q6:

```
Q7 = select(project(STUDENT, {SName}), MajorId=10) // Недопустим!
```

Этот запрос не имеет смысла, потому что результат внутреннего запроса не содержит поля `MajorId`, указанного в предикате оператора select.

8.1.3. Прямое произведение

Операторы select и project обрабатывают одну таблицу. Оператор *product* позволяет объединять и сравнивать информацию из нескольких таблиц. Он принимает две таблицы и возвращает таблицу, включающую все комбинации записей из входных таблиц, а схема выходной таблицы является объединением полей в схемах входных таблиц. Входные таблицы должны иметь уникальные имена полей, чтобы в выходной таблице не было двух полей с одинаковыми именами.

Вот запрос Q8, возвращающий результат применения оператора product к таблицам STUDENT и DEPT:

```
Q8 = product(STUDENT, DEPT)
```

Как показано на рис. 1.1, база данных университета содержит девять записей в таблице STUDENT и три записи в таблице DEPT. В табл. 8.1 показаны результаты запроса Q8, принимающего эти две таблицы. Выходная таблица содержит 27 записей, по одной для каждой возможной пары записей в STUDENT и DEPT. В общем случае если в STUDENT имеется `N` записей, а в DEPT – `M` записей, то выходная таблица будет содержать `N×M` записей (кстати, именно поэтому оператор называется «product» – произведение).

Таблица 8.1. Результаты запроса Q8

Q8	SId	SName	MajorId	GradYear	DId	DName
	1	joe	10	2021	10	compsci
	1	joe	10	2021	20	math
	1	joe	10	2021	30	drama
	2	amy	20	2020	10	compsci
	2	amy	20	2020	20	math
	2	amy	20	2020	30	drama
	3	max	10	2022	10	compsci

Окончание табл. 8.1

Q8	SId	SName	MajorId	GradYear	DId	DName
	3	max	10	2022	20	math
	3	max	10	2022	30	drama
	4	sue	20	2022	10	compsci
	4	sue	20	2022	20	math
	4	sue	20	2022	30	drama
	5	bob	30	2020	10	compsci
	5	bob	30	2020	20	math
	5	bob	30	2020	30	drama
	6	kim	20	2020	10	compsci
	6	kim	20	2020	20	math
	6	kim	20	2020	30	drama
	7	art	30	2021	10	compsci
	7	art	30	2021	20	math
	7	art	30	2021	30	drama
	8	pat	20	2019	10	compsci
	8	pat	20	2019	20	math
	8	pat	20	2019	30	drama
	9	lee	10	2021	10	compsci
	9	lee	10	2021	20	math
	9	lee	10	2021	30	drama

Запрос Q8 не имеет особого смысла, так как не учитывает специальность, выбранную студентом как основную. Принять этот аспект во внимание можно в предикате, как показано в запросе Q9 и на рис. 8.3:

```
Q9 = select(product(STUDENT, DEPT), MajorId=Did)
```

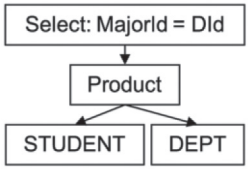


Рис. 8.3. Дерево запроса для Q9

Выходная таблица этого запроса содержит только комбинации записей из STUDENT и DEPT, удовлетворяющие предикату. То есть из 27 возможных комбинаций останутся только те, в которых номер специальности студента совпадает с номером специальности в таблице DEPT, – иными словами, таблица результатов будет содержать список студентов и соответствующих им специальностей. Теперь в выходной таблице вместо 27 записей осталось только 9.

8.2. Образ сканирования

Образ – это объект, представляющий набор записей, получаемых при сканировании результатов выполнения запроса реляционной алгебры. Образы скани-

рования в SimpleDB реализуют интерфейс Scan (см. листинг 8.1). Методы Scan являются подмножеством методов образа табличного сканирования TableScan и имеют одинаковое поведение. Это соответствие не должно вызывать удивления – результатом запроса является таблица, поэтому вполне естественно, что доступ к результатам запросов и таблицам осуществляется одинаково.

Листинг 8.1. Интерфейс Scan в SimpleDB

```
public interface Scan {
    public void     beforeFirst();
    public boolean  next();
    public int      getInt(String fldname);
    public String   getString(String fldname);
    public Constant getVal(String fldname);
    public boolean  hasField(String fldname);
    public void     close();
}
```

Для примера рассмотрим метод printNameAndGradYear в листинге 8.2. Этот метод выполняет сканирование записей и выводит значения их полей sname и gradyear.

Листинг 8.2. Вывод имени и года выпуска для каждого студента в образе сканирования

```
public static void printNameAndGradyear(Scan s) {
    s.beforeFirst();
    while (s.next()) {
        String sname = s.getString("sname");
        String gradyr = s.getInt("gradyear");
        System.out.println(sname + "\t" + gradyr);
    }
    s.close();
}
```

Суть этого примера в том, что метод работает с образом, не зная, результатом сканирования какого запроса (или таблицы) он является. Это может быть таблица STUDENT или запрос, выбирающий студентов, обучающихся по конкретной специальности или прослушавших курс профессора Эйнштейна. Единственное, что требуется, – чтобы выходная таблица образа содержала имя и год выпуска студента.

Объект Scan соответствует узлу в дереве запроса. Для каждого реляционного оператора в SimpleDB определен свой класс Scan. Объекты этих классов образуют внутренние узлы дерева запроса, а объекты TableScan обозначают листья дерева. В листинге 8.3 показаны конструкторы классов Scan для таблиц и трех основных операторов, поддерживаемых в SimpleDB.

Листинг 8.3. API конструкторов классов в SimpleDB, реализующих интерфейс Scan

Scan

```
public TableScan(Transaction tx, String filename, Layout layout);
public SelectScan(Scan s, Predicate pred);
public ProjectScan(Scan s, List<String> fldlist);
public ProductScan(Scan s1, Scan s2);
```

Конструктор SelectScan принимает два аргумента: базовый образ сканирования и предикат. Базовый образ служит входом для оператора select. Поскольку Scan является интерфейсом, объект SelectScan не знает, является ли его аргумент хранимой таблицей или результатом другого запроса. То есть входом реляционного оператора может быть любая таблица или запрос.

Предикат селекции, переданный в конструктор SelectScan, имеет тип Predicate. Детали обработки предикатов в SimpleDB обсуждаются в разделе 8.6, а до тех пор я постараюсь не акцентировать внимание на этом вопросе.

Деревья запросов строятся путем объединения образов. Каждому узлу дерева соответствует свой образ сканирования. Например, в листинге 8.4 приводится код, реализующий дерево запросов на рис. 8.2 (без деталей, касающихся предиката). Переменные типа Scan – s1, s2 и s3 – соответствуют узлам в дереве запроса. Дерево строится снизу вверх: сначала создается образ сканирования таблицы, затем образ для оператора select и, наконец, образ для оператора project. Переменная s3 содержит окончательное дерево запроса. Цикл while выполняет обход содержимого s3 и выводит имя каждого студента.

Листинг 8.4. Представление дерева запроса на рис. 8.2 в виде образа сканирования

```
Transaction tx = db.newTx();
MetadataMgr mdm = db.MetadataMgr();

// узел STUDENT
Layout layout = mdm.getLayout("student", tx);
Scan s1 = new TableScan(tx, "student", layout);

// узел select
Predicate pred = new Predicate(. . .); // majorid=10
Scan s2 = new SelectScan(s1, pred);

// узел project
List<String> c = Arrays.asList("sname");
Scan s3 = new ProjectScan(s2, c);

while (s3.next())
    System.out.println(s3.getString("sname"));
s3.close();
```

В листинге 8.5 приводится код, реализующий дерево запросов на рис. 8.3. Код содержит четыре образа сканирования, потому что дерево запроса имеет четыре узла. Переменная s4 содержит окончательное дерево запроса. Обратите внимание, что цикл while практически идентичен предыдущему. Для экономии места цикл выводит значения только трех полей из каждой записи, но его легко изменить, чтобы включить вывод значений всех шести полей.

Листинг 8.5. Представление дерева запроса на рис. 8.3 в виде образа сканирования

```
Transaction tx = db.newTx();
MetadataMgr mdm = db.MetadataMgr();

// узел STUDENT
Layout layout1 = mdm.getLayout("student", tx);
Scan s1 = new TableScan(tx, "student", layout1);
```

```
// узел DEPT
Layout layout2 = mdm.getLayout("dept", tx);
Scan s2 = new TableScan(tx, "dept", layout2);

// узел product
Scan s3 = new ProductScan(s1, s2);

// узел select
Predicate pred = new Predicate(. . .); //majorid=did
Scan s4 = new SelectScan(s3, pred);

while (s4.next())
    System.out.println(s4.getString("sname")
                    + ", " + s4.getString("gradyear")
                    + ", " + s4.getString("dname") );
s4.close();
```

Наконец, обратите внимание, что метод `close` вызывается только для самого внешнего объекта `Scan` в дереве запроса. Когда закрывается внешний образ сканирования, он автоматически закроет все вложенные образы.

8.3. Обновляемые образы

Запрос определяет виртуальную таблицу. Интерфейс `Scan` имеет методы, позволяющие клиентам читать данные из этой виртуальной таблицы, но не изменять их. Не все образы сканирования поддерживают возможность обновления. Образ считается *обновляемым*, если каждой выходной записи `r` в образе соответствует запись `r'` в базовой таблице. В этом случае изменение в `r` определяется как изменение в `r'`.

Обновляемые образы сканирования поддерживают интерфейс `UpdateScan` (см. листинг 8.6). Первые пять методов интерфейса определяют основные операции обновления. Два других связаны с идентификатором хранимой записи, соответствующей текущей записи в образе. Метод `getRid` возвращает этот идентификатор, а `moveToRid` перемещает текущую позицию в образе к указанной хранимой записи.

Листинг 8.6. Интерфейс UpdateScan в SimpleDB

```
public interface UpdateScan extends Scan {
    public void setInt(String fldname, int val);
    public void setString(String fldname, String val);
    public void setVal(String fldname, Constant val);
    public void insert();
    public void delete();

    public RID getRid();
    public void moveToRid(RID rid);
}
```

В SimpleDB только два класса, реализующих интерфейс `UpdateScan`: `TableScan` и `SelectScan`. Примеры их использования приводятся в листинге 8.7. В части (a) этого листинга показан оператор SQL, который меняет оценку каждого учащегося, прослушавшего курс 53, а в части (b) – код, реализующий этот оператор.

Сначала код создает образ сканирования для всех зачисленных на курс 53, затем перебирает все записи в образе, изменяя оценку в каждой из них.

Листинг 8.7. Представление SQL-оператора update в виде обновляемого образа: (a) оператор SQL, изменяющий оценки студентов, прослушавших курс 53; (b) код, реализующий оператор

```
update ENROLL
set Grade = 'C' where SectionId = 53
```

(a)

```
Transaction tx = db.newTx();
MetadataMgr mdm = db.MetadataMgr();

Layout layout = mdm.getLayout("enroll", tx);
Scan s1 = new TableScan(tx, "enroll", layout);

Predicate pred = new Predicate(. . .); // SectionId=53
UpdateScan s2 = new SelectScan(s1, pred);

while (s2.next())
    s2.setString("grade", "C");
s2.close();
```

(b)

Для переменной `s2` вызывается метод `setString`, поэтому она должна быть объявлена с типом обновляемого образа сканирования. С другой стороны, первый аргумент конструктора `SelectScan` – это образ сканирования, то есть его не требуется объявлять как обновляемый образ. Вместо этого код, вызывающий метод `s2.setString`, преобразует базовый образ (т. е. `s1`) в обновляемый; если образ не может обновляться, будет сгенерировано исключение `ClassCastException`.

8.4. Реализация образов сканирования

Движок SimpleDB содержит четыре класса `Scan`: класс `TableScan` и три класса для операторов select, project и product. Класс `TableScan` был описан в главе 6. Три других класса обсуждаются в подразделах ниже.

8.4.1. Образы сканирования для оператора select

В листинге 8.8 приводится определение класса `SelectScan`. Его конструктор сохраняет образ сканирования базовой входной таблицы. Текущая запись образа сканирования для оператора select совпадает с текущей записью базового образа, благодаря чему большинство методов можно реализовать, просто вызывая соответствующие методы этого базового образа.

Единственный нетривиальный метод – `next`. Его задача – перейти к следующей записи. Метод перебирает базовый образ в поисках записи, удовлетворяющей предикату. Обнаружив такую запись, метод делает ее текущей и возвращает значение `true`. Если такой записи нет, цикл `while` завершится и метод вернет `false`.

Образ сканирования для оператора select поддерживает возможность обновления. Методы `UpdateScan` предполагают, что базовый образ тоже может

обновляться; в частности, они предполагают, что могут привести базовый образ сканирования к типу UpdateScan, не опасаясь исключения ClassCastException. Планировщик изменений в SimpleDB может создавать только образы сканирования таблиц и образы для оператора select, так что это исключение не должно генерироваться.

Листинг 8.8. Определение класса SelectScan в SimpleDB

```
public class SelectScan implements UpdateScan {
    private Scan s;
    private Predicate pred;

    public SelectScan(Scan s, Predicate pred) {
        this.s = s;
        this.pred = pred;
    }

    // Реализация интерфейса Scan
    public void beforeFirst() {
        s.beforeFirst();
    }

    public boolean next() {
        while (s.next())
            if (pred.isSatisfied(s))
                return true;
        return false;
    }

    public int getInt(String fldname) {
        return s.getInt(fldname);
    }

    public String getString(String fldname) {
        return s.getString(fldname);
    }

    public Constant getVal(String fldname) {
        return s.getVal(fldname);
    }

    public boolean hasField(String fldname) {
        return s.hasField(fldname);
    }

    public void close() {
        s.close();
    }

    // Реализация интерфейса UpdateScan
    public void setInt(String fldname, int val) {
        UpdateScan us = (UpdateScan) s;
        us.setInt(fldname, val);
    }
```

```
    public void setString(String fldname, String val) {
        UpdateScan us = (UpdateScan) s;
        us.setString(fldname, val);
    }

    public void setVal(String fldname, Constant val) {
        UpdateScan us = (UpdateScan) s;
        us.setVal(fldname, val);
    }

    public void delete() {
        UpdateScan us = (UpdateScan) s;
        us.delete();
    }

    public void insert() {
        UpdateScan us = (UpdateScan) s;
        us.insert();
    }

    public RID getRid() {
        UpdateScan us = (UpdateScan) s;
        return us.getRid();
    }

    public void moveToRid(RID rid) {
        UpdateScan us = (UpdateScan) s;
        us.moveToRid(rid);
    }
}
```

8.4.2. Образы сканирования для оператора project

В листинге 8.9 приводится определение класса `ProjectScan`. Конструктор получает список выходных полей, который используется в методе `hasField`. Другие методы просто делегируют работу соответствующим методам базового образа. Методы `getVal`, `getInt` и `getString` проверяют наличие поля с указанным именем в списке полей и, если его нет, генерируют исключение.

Листинг 8.9. Определение класса ProjectScan в SimpleDB

```
public class ProjectScan implements Scan {
    private Scan s;
    private Collection<String> fieldlist;

    public ProjectScan(Scan s, List<String> fieldlist) {
        this.s = s;
        this.fieldlist = fieldlist;
    }

    public void beforeFirst() {
        s.beforeFirst();
    }

    public boolean next() {
        return s.next();
    }
```

```
    public int getInt(String fldname) {
        if (hasField(fldname))
            return s.getInt(fldname);
        else
            throw new RuntimeException("field not found.");
    }

    public String getString(String fldname) {
        if (hasField(fldname))
            return s.getString(fldname);
        else
            throw new RuntimeException("field not found.");
    }

    public Constant getVal(String fldname) {
        if (hasField(fldname))
            return s.getVal(fldname);
        else
            throw new RuntimeException("field not found.");
    }

    public boolean hasField(String fldname) {
        return fieldlist.contains(fldname);
    }

    public void close() {
        s.close();
    }
}
```

Класс `ProjectScan` не реализует интерфейс `UpdateScan`, даже притом, что проекции могут быть обновляемыми. В упражнении 8.12 вам будет предложено дополнить реализацию.

8.4.3. Образы сканирования для оператора product

В листинге 8.10 приводится определение класса `ProductScan`. Образ сканирования для оператора product должен перебирать все возможные комбинации записей в базовых образах `s1` и `s2`. Сначала выбирается первая запись из `s1` и перебираются все записи в `s2`, затем выполняется переход ко второй записи в `s1` и снова перебираются все записи в `s2` и т. д. Концептуально это напоминает два вложенных цикла, где внешний выполняет итерации по записям в `s1`, а внутренний – по записям в `s2`.

Листинг 8.10. Определение класса ProductScan в SimpleDB

```
public class ProductScan implements Scan {
    private Scan s1, s2;

    public ProductScan(Scan s1, Scan s2) {
        this.s1 = s1;
        this.s2 = s2;
        s1.next();
    }
```

```
    public void beforeFirst() {
        s1.beforeFirst();
        s1.next();
        s2.beforeFirst();
    }

    public boolean next() {
        if (s2.next())
            return true;
        else {
            s2.beforeFirst();
            return s2.next() && s1.next();
        }
    }

    public int getInt(String fldname) {
        if (s1.hasField(fldname))
            return s1.getInt(fldname);
        else
            return s2.getInt(fldname);
    }

    public String getString(String fldname) {
        if (s1.hasField(fldname))
            return s1.getString(fldname);
        else
            return s2.getString(fldname);
    }

    public Constant getVal(String fldname) {
        if (s1.hasField(fldname))
            return s1.getVal(fldname);
        else
            return s2.getVal(fldname);
    }

    public boolean hasField(String fldname) {
        return s1.hasField(fldname) || s2.hasField(fldname);
    }

    public void close() {
        s1.close();
        s2.close();
    }
}
```

Метод `next` реализует идею «вложенных циклов» следующим образом. Каждый вызов `next` выполняет переход к следующей записи в `s2`. Если в `s2` есть следующая запись, то метод возвращает `true`. Если нет, значит, перебор содержимого `s2` завершился и метод переходит к следующей записи в `s1` и к первой записи в `s2`. Если переход прошел успешно, метод возвращает `true`; если в `s1` больше нет записей, то сканирование завершается и `next` возвращает `false`.

Методы `getVal`, `getInt` и `getString` просто получают доступ к полю соответствующего базового образа. Все эти методы проверяют наличие поля с указанным именем в `s1`. Если такое поле имеется, его значение извлекается из `s1`, иначе – из `s2`.

8.5. Конвейерная обработка запросов

Реализации этих трех операторов реляционной алгебры имеют два общих свойства:

- генерируют выходные записи по одной, по мере необходимости;
- не сохраняют ни выходные записи, ни результаты промежуточных вычислений.

Такие реализации называются *конвейерными*. Этот раздел посвящен анализу конвейерных реализаций и их свойств.

Рассмотрим объект `TableScan`. Он содержит страницу записей с буфером, хранящим страницу с текущей записью. Текущая запись – это просто местоположение в этой странице. Запись не должна удаляться из своей страницы; если клиент запросит значение поля, диспетчер записей просто извлечет его из страницы и вернет клиенту. Каждый вызов `next` производит переход к следующей записи в образе табличного сканирования, что может привести к загрузке другой страницы записей.

Теперь рассмотрим объект `SelectScan`. Каждый вызов его метода `next` многократно вызывает метод `next` базового образа, пока не найдет запись, удовлетворяющую предикату. Конечно, никакой фактической «текущей записи» не существует – если базовый образ является образом сканирования таблицы, то текущая запись – это просто местоположение в странице, хранящейся в объекте `TableScan`. А если базовый образ является объектом другого типа (например, образом для оператора product, такого как на рис. 8.3 и в листинге 8.5), то значения полей текущей записи определяются из текущих записей образов табличных сканирований, находящихся в поддереве этого узла.

Каждый раз, когда образ сканирования выполняет конвейерную обработку следующего вызова `next`, он начинает поиск с того места, где остановился. В результате образ запрашивает ровно столько записей из базового образа, сколько необходимо, чтобы добраться до следующей выходной записи.

Конвейерный образ не сохраняет выбранные записи, поэтому, если клиент запросит ту же запись еще раз, весь поиск придется выполнить с самого начала.

Термин «конвейерный» относится к последовательности вызовов методов вниз по дереву запросов и к последовательности результатов, возвращаемых в обратном направлении, вверх по дереву. Например, рассмотрим вызов метода `getInt`. Каждый узел в дереве передает этот вызов вниз, одному из своих дочерних узлов, пока не будет достигнут листовой узел. Этот листовой узел (который является образом табличного сканирования) извлекает искомое значение из своей страницы и возвращает его обратно, вверх по дереву. Или рассмотрим вызов метода `next`. Каждый узел выполняет один или несколько вызовов метода `next` (и, возможно, `beforeFirst`, если это узел оператора product) своих дочерних узлов, пока не убедится, что они ссылаются на следующую запись. Затем возвращает родительскому узлу признак успеха (или неудачи, если такой записи не существует).

Конвейерные реализации могут быть очень эффективными. Например, рассмотрим дерево запроса на рис. 8.4, который извлекает имена студентов, окончивших обучение в 2020 году по специальности 10.

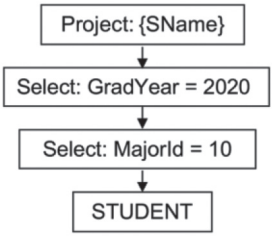


Рис. 8.4. Дерево запроса с несколькими узлами select

Узлы project и select в этом дереве не обращаются к другим блокам в таблице STUDENT, кроме тех, которые необходимы образу сканирования таблицы. Чтобы понять, почему, сначала рассмотрим узел project. Каждый вызов `next` этого узла будет просто вызывать метод `next` его дочернего узла и возвращать полученное значение. Другими словами, узел project не изменяет количество обращений к блокам, выполненных остальной частью запроса.

Теперь рассмотрим узлы select. Вызов метода `next` внешнего узла select вызовет `next` внутреннего узла select. Внутренний узел будет снова и снова вызывать `next` своего дочернего узла, пока не достигнет записи, удовлетворяющей предикату «MajorId = 10». Затем внутренний узел select вернет `true`, а внешний узел select проверит текущую запись. Если год выпуска не равен 2020, то внешний узел снова вызовет метод `next` внутреннего узла и будет ждать получения другой записи. Внешний узел select вернет `true`, только если полученная запись удовлетворяет обоим предикатам. Этот процесс выполняется каждый раз, когда вызывается метод `next` внешнего узла, при этом образ сканирования базовой таблицы постоянно перемещается от записи к записи, пока оба предиката не будут удовлетворены. Когда образ табличного сканирования обнаружит, что записи в таблице STUDENT закончились, его метод `next` вернет значение `false`, которое затем будет передано вверх по дереву. Другими словами, таблица STUDENT сканируется только один раз, как если бы запрос выполнил простое сканирование таблицы. Отсюда следует, что узлы select в этом запросе не привносят дополнительных затрат.

Несмотря на высокую эффективность конвейерных реализаций в таких случаях, в других ситуациях они не так хороши. Одна такая ситуация – когда узел select находится справа от узла product; в этом случае он будет выполнен несколько раз. Вместо того чтобы выполнять селекцию снова и снова, лучше использовать реализацию, которая материализует выходные записи и сохраняет их во временной таблице. Такие реализации рассматриваются в главе 13.

8.6. Предикаты

Предикат определяет условие, которое возвращает истину или ложь для каждой записи в данном образе сканирования. Если условие возвращает `true`, говорят, что запись *удовлетворяет* предикату. В SQL предикат имеет следующую структуру:

- *предикат* – это простое условие (терм) или логическая комбинация простых условий;
- *простое условие* – это операция сравнения двух выражений;

- *выражение* состоит из операций с константами и именами полей;
- *константа* – это значение предопределенного типа, например целое число или строка.

Рассмотрим следующий предикат на стандартном SQL:

```
( GradYear>2021 or MOD(GradYear,4)=0 ) and MajorId=DId
```

Этот предикат содержит три простых условия (выделены жирным). Первые два условия сравнивают поле `GradYear` (или функцию `GradYear`) с константой, а третье – два поля. Каждое простое условие содержит два выражения. Например, второе условие содержит выражения `MOD(GradYear,4)` и `0`.

SimpleDB значительно сужает круг допустимых констант, выражений, простых условий и предикатов. Константа в SimpleDB может быть только целым числом или строкой, выражение может быть только константой или именем поля, простое условие может проверять только равенство выражений, а предикат может объединять простые условия только логическим «И» (AND). В упражнениях 8.7–8.9 вам будет предложено расширить круг допустимых предикатов в SimpleDB для большей выразительности.

Рассмотрим следующий предикат:

```
SName = 'joe' and MajorId = DId
```

Фрагмент кода в листинге 8.11 демонстрирует, как определить этот предикат в SimpleDB. Обратите внимание, что предикат создается изнутри наружу: сначала определяются константы и выражения, затем простые условия, и наконец сам предикат.

Листинг 8.11. Реализация предиката в SimpleDB

```
Expression lhs1 = new Expression("SName");
Constant c = new Constant("joe");
Expression rhs1 = new Expression(c);
Term t1 = new Term(lhs1, rhs1);

Expression lhs2 = new Expression("MajorId");
Expression rhs2 = new Expression("DId");
Term t2 = new Term(lhs2, rhs2);

Predicate pred1 = new Predicate(t1);
Predicate pred2 = new Predicate(t2);
pred1.conjoinWith(pred2);
```

В листинге 8.12 показано определение класса `Constant`. Каждый объект `Constant` содержит переменную типа `Integer` и переменную типа `String`. Только одна из этих переменных будет содержать значение, отличное от `null`, в зависимости от того, какой конструктор был вызван. Методы `equals`, `compareTo`, `hasCode` и `toString` используют ту переменную, значение которой отлично от `null`.

Листинг 8.12. Определение класса Constant

```
public class Constant implements Comparable<Constant> {
    private Integer ival = null;
    private String sval = null;
```

```
    public Constant(Integer ival) {
        this.ival = ival;
    }

    public Constant(String sval) {
        this.sval = sval;
    }

    public int asInt() {
        return ival;
    }

    public String asString() {
        return sval;
    }

    public boolean equals(Object obj) {
        Constant c = (Constant) obj;
        return (ival != null) ? ival.equals(c.ival)
                              : sval.equals(c.sval);
    }

    public int compareTo(Constant c) {
        return (ival!=null) ? ival.compareTo(c.ival)
                            : sval.compareTo(c.sval);
    }

    public int hashCode() {
        return (ival != null) ? ival.hashCode() : sval.hashCode();
    }

    public String toString() {
        return (ival != null) ? ival.toString() : sval.toString();
    }
}
```

В листинге 8.13 показано определение класса `Expression`. Он тоже имеет два конструктора: один создает константное выражение, а другой – выражение с именем поля. Каждый конструктор присваивает значение своей переменной. Метод `isFieldName` помогает определить, является выражение именем поля или нет. Метод `evaluate` возвращает значение выражения относительно текущей выходной записи в образе сканирования. Если выражение является константой, то образ не используется и метод просто возвращает константу. Если выражение является именем поля, то метод возвращает значение поля из образа. Метод `appliesTo` используется планировщиком запросов для определения области применимости выражения.

Листинг 8.13. Определение класса Expression

```
public class Expression {
    private Constant val = null;
    private String fldname = null;

    public Expression(Constant val) {
        this.val = val;
    }
```

```
    public Expression(String fldname) {
        this.fldname = fldname;
    }

    public boolean isFieldName() {
        return fldname != null;
    }

    public Constant asConstant() {
        return val;
    }

    public String asFieldName() {
        return fldname;
    }

    public Constant evaluate(Scan s) {
        return (val != null) ? val : s.getVal(fldname);
    }

    public boolean appliesTo(Schema sch) {
        return (val != null) ? true : sch.hasField(fldname);
    }

    public String toString() {
        return (val != null) ? val.toString() : fldname;
    }
}
```

Простые условия в SimpleDB реализованы в виде класса `Term`, определение которого приводится в листинге 8.14. Его конструктор принимает два аргумента, обозначающих левое и правое выражения. Наиболее важным методом является `isSatisfied`, который возвращает `true`, если оба выражения имеют одно и то же значение в данном образе сканирования. Остальные методы помогают планировщику запросов определить вид и область применимости простого условия. Например, метод `reductionFactor` определяет ожидаемое количество записей, которые будут удовлетворять предикату (подробнее о нем рассказывается в главе 10). Методы `equatesWithConstant` и `equatesWithField` помогают планировщику запросов решить, когда использовать индексы, и обсуждаются в главе 15.

Листинг 8.14. Определение класса Term в SimpleDB

```
public class Term {
    private Expression lhs, rhs;

    public Term(Expression lhs, Expression rhs) {
        this.lhs = lhs;
        this.rhs = rhs;
    }

    public boolean isSatisfied(Scan s) {
        Constant lhsval = lhs.evaluate(s);
        Constant rhsval = rhs.evaluate(s);
        return rhsval.equals(lhsval);
    }
```

```
    public boolean appliesTo(Schema sch) {
        return lhs.appliesTo(sch) && rhs.appliesTo(sch);
    }

    public int reductionFactor(Plan p) {
        String lhsName, rhsName;
        if (lhs.isFieldName() && rhs.isFieldName()) {
            lhsName = lhs.asFieldName();
            rhsName = rhs.asFieldName();
            return Math.max(p.distinctValues(lhsName),
                            p.distinctValues(rhsName));
        }
        if (lhs.isFieldName()) {
            lhsName = lhs.asFieldName();
            return p.distinctValues(lhsName);
        }
        if (rhs.isFieldName()) {
            rhsName = rhs.asFieldName();
            return p.distinctValues(rhsName);
        }
        // иначе условие сравнивает константы
        if (lhs.asConstant().equals(rhs.asConstant()))
            return 1;
        else
            return Integer.MAX_VALUE;
    }

    public Constant equatesWithConstant(String fldname) {
        if ( lhs.isFieldName() &&
             lhs.asFieldName().equals(fldname) &&
             !rhs.isFieldName())
            return rhs.asConstant();
        else if (rhs.isFieldName() &&
                 rhs.asFieldName().equals(fldname) &&
                 !lhs.isFieldName())
            return lhs.asConstant();
        else
            return null;
    }

    public String equatesWithField(String fldname) {
        if ( lhs.isFieldName() &&
             lhs.asFieldName().equals(fldname) &&
             rhs.isFieldName())
            return rhs.asFieldName();
        else if (rhs.isFieldName() &&
                 rhs.asFieldName().equals(fldname) &&
                 lhs.isFieldName())
            return lhs.asFieldName();
        else
            return null;
    }

    public String toString() {
        return lhs.toString() + "=" + rhs.toString();
    }
}
```

В листинге 8.15 приводится определение класса Predicate. Предикат реализован как список простых условий и реагирует на вызовы своих методов, вызывая соответствующие методы этих условий. Класс имеет два конструктора. Один конструктор – без аргументов – создает предикат без условий. Такой предикат всегда возвращает *истину*, и ему соответствует любая запись. Другой конструктор создает предикат с одним простым условием. Метод conjoinWith добавляет простые условия из предиката-аргумента в данный предикат.

Листинг 8.15. Определение класса Predicate в SimpleDB

```
public class Predicate {
    private List<Term> terms = new ArrayList<Term>();

    public Predicate() {}

    public Predicate(Term t) {
        terms.add(t);
    }

    public void conjoinWith(Predicate pred) {
        terms.addAll(pred.terms);
    }

    public boolean isSatisfied(Scan s) {
        for (Term t : terms)
            if (!t.isSatisfied(s))
                return false;
        return true;
    }

    public int reductionFactor(Plan p) {
        int factor = 1;
        for (Term t : terms)
            factor *= t.reductionFactor(p);
        return factor;
    }

    public Predicate selectSubPred(Schema sch) {
        Predicate result = new Predicate();
        for (Term t : terms)
            if (t.appliesTo(sch))
                result.terms.add(t);
        if (result.terms.size() == 0)
            return null;
        else
            return result;
    }

    public Predicate joinSubPred(Schema sch1, Schema sch2) {
        Predicate result = new Predicate();
        Schema newsch = new Schema();
        newsch.addAll(sch1);
        newsch.addAll(sch2);
```

```
        for (Term t : terms)
            if ( !t.appliesTo(sch1) &&
                 !t.appliesTo(sch2) &&
                 t.appliesTo(newsch))
                result.terms.add(t);
        if (result.terms.size() == 0)
            return null;
        else
            return result;
    }

    public Constant equatesWithConstant(String fldname) {
        for (Term t : terms) {
            Constant c = t.equatesWithConstant(fldname);
            if (c != null)
                return c;
        }
        return null;
    }

    public String equatesWithField(String fldname) {
        for (Term t : terms) {
            String s = t.equatesWithField(fldname);
            if (s != null)
                return s;
        }
        return null;
    }

    public String toString() {
        Iterator<Term> iter = terms.iterator();
        if (!iter.hasNext())
            return "";
        String result = iter.next().toString();
        while (iter.hasNext())
            result += " and " + iter.next().toString();
        return result;
    }
}
```

8.7. Итоги

- Запрос реляционной алгебры состоит из *операторов*. Каждый оператор выполняет одну конкретную задачу. Комбинацию операторов в запросе можно записать в виде *дерева запроса*.
- В этой главе описываются три оператора, чтобы помочь понять особенности версии SQL в SimpleDB. Вот эти операторы:
  - *оператор селекции (фильтрации) select* возвращает таблицу, содержащую те же столбцы, что и входная таблица, но за исключением некоторых строк;
  - *оператор проекции project* возвращает таблицу, содержащую те же строки, что и входная таблица, но за исключением некоторых столбцов;

- *оператор прямого (декартова) произведения product* возвращает таблицу, содержащую все возможные комбинации записей из двух входных таблиц.

- *Образ сканирования* – это объект, представляющий дерево запроса реляционной алгебры. Каждому реляционному оператору в SimpleDB соответствует свой класс, реализующий интерфейс `Scan`; объекты этих классов составляют внутренние узлы дерева запроса. Существует также класс для образа табличного сканирования, экземпляры которого составляют листья дерева.
- Класс `Scan` имеет практически те же методы, что и `TableScan`. С помощью этого класса клиенты сканируют записи, перемещаясь от одной выходной записи к другой, и извлекают значения полей. Образ сканирования управляет реализацией запроса, перемещаясь по файлам с записями и сравнивая значения.
- Образ является *обновляемым*, если для каждой записи `r` в скане имеется соответствующая запись `r'` в некоторой таблице в базе данных. Изменение виртуальной записи `r` в этом случае сводится к изменению хранимой записи `r'`.
- Методы каждого класса `Scan` реализуют поведение своего оператора. Например:
  - образ сканирования для оператора select проверяет каждую запись в базовом образе и возвращает только те из них, которые удовлетворяют предикату;
  - образ сканирования для оператора product возвращает запись для каждой возможной комбинации записей из двух базовых образов;
  - образ сканирования таблицы открывает файл с записями, хранящий указанную таблицу, при необходимости закрепляя буферы и устанавливая блокировки.
- Такие реализации образов сканирования называются *конвейерными*. Конвейерная реализация не использует прием опережающего чтения, не кеширует данные, не сортирует и вообще никак не обрабатывает их.
- Конвейерная реализация не создает выходные записи. Каждый лист в дереве запроса является образом сканирования таблицы, содержащим буфер с текущей записью из этой таблицы. «Текущая запись» определяется по записям в каждом буфере. Запросы, извлекающие значения полей, направляются вниз по дереву до соответствующего образа сканирования таблицы; результаты возвращаются из этого образа обратно вверх, в направлении корня дерева.
- Образы сканирования с конвейерной реализацией выполняют только необходимые операции. Каждый образ запросит ровно столько дочерних записей, сколько потребуется для определения следующей записи.

8.8. Для дополнительного чтения

Реляционная алгебра описывается почти во всех вводных книгах о базах данных, хотя в каждой для примеров используется свой синтаксис. Подробное описание реляционной алгебры и ее выразительной силы можно найти в Atzeni and DeAntonellis (1992). В этой книге также представлено *реляционное исчисле-*

ние – язык запросов, основанный на логике предикатов. Интересно отметить, что реляционное исчисление можно расширить для поддержки рекурсивных запросов (то есть запросов, в которых выходная таблица упоминается в определении запроса). Рекурсивное реляционное исчисление называется Datalog и связано с языком программирования Prolog. Обсуждение Datalog и его выразительной силы также можно найти в Atzeni and DeAntonellis (1992).

Конвейерная обработка – это лишь малая часть мозаики обработки запросов, которая также включает темы, обсуждаемые в последующих главах. В статье Graefe (1993) приводится исчерпывающая информация о методах обработки запросов; в разделе 1 подробно обсуждаются образы сканирования и конвейерная обработка. В статье Chaudhuri (1998) рассматриваются деревья запросов, а также сбор статистик и оптимизация.

Atzeni, P., & DeAntonellis, V. (1992). «Relational database theory». Upper Saddle River, NJ: Prentice-Hall.

Chaudhuri, S. (1998). «An overview of query optimization in relational systems». In Proceedings of the ACM Principles of Database Systems Conference (p. 34–43).

Graefe, G. (1993). «Query evaluation techniques for large databases». ACM Computing Surveys, 25 (2), 73–170.

8.9. Упражнения

Теория

8.1. Какой результат вернет оператор product, если в одном из аргументов передать пустой набор данных?

8.2. Реализуйте следующий запрос в виде образа сканирования, взяв за основу листинг 8.5.

```
select sname, dname, grade
from STUDENT, DEPT, ENROLL, SECTION
where SId=StudentId and SectId=SectionId and DId=MajorId
and YearOffered=2020
```

8.3. Исследуйте код в листинге 8.5.

a) Какие блокировки должна установить транзакция в процессе выполнения этого кода?

b) Для каждой из этих блокировок приведите сценарий, который заставит код ждать ее освобождения.

8.4. Исследуйте определение класса `ProductScan`.

a) Какая проблема может возникнуть, если в первом базовом образе сканирования не обнаружится ни одной записи? Как исправить код?

b) Объясните, почему проблема не возникает, если ни одной записи не обнаружится во втором базовом образе.

8.5. Допустим, вам нужно найти все пары студентов, взяв прямое произведение таблицы STUDENT с самой собой.

a) Один из способов – создать образ сканирования таблицы STUDENT и передать его в обоих аргументах оператору product, как показано ниже:

```
Layout layout = mdm.getLayout("student", tx);
Scan s1 = new TableScan(tx, "student", layout);
Scan s2 = new ProductScan(s1, s1);
```

Объясните, почему это решение показывает некорректное (и странное) поведение.

b) Более удачное решение: создать два разных образа сканирования таблицы STUDENT и применить к ним оператор product. Это решение вернет все комбинации записей из таблицы STUDENT, но имеет проблему. Опишите, в чем она заключается.

Практика

8.6. Методы `getVal`, `getInt` и `getString` класса `ProjectScan` проверяют действительность имен полей в их аргументах. Но в других классах образов сканирований такая проверка отсутствует. Для всех других классов:

a) опишите, какая проблема возникнет (и где), если этим методам передать недействительное имя поля;

b) исправьте код SimpleDB так, чтобы он генерировал соответствующее исключение.

8.7. На данный момент SimpleDB поддерживает только целочисленные и строковые константы.

a) Добавьте в SimpleDB поддержку констант других типов, например короткие целые числа, массивы байтов и даты.

b) В упражнении 3.17 вам было предложено добавить в класс `Page` методы `get` и `set` для таких типов, как короткие целые числа, даты и т. д. Если вы выполнили это упражнение, добавьте аналогичные методы `get` и `set` в интерфейсы `Scan` и `UpdateScan` (и их реализации в соответствующих классах), а также в диспетчер записей, диспетчер транзакций и диспетчер буферов. Затем измените методы `getVal` и `setVal` соответственно.

8.8 Добавьте в реализацию выражений поддержку арифметических операторов с целыми числами.

8.9. Добавьте в класс `Term` поддержку операторов сравнения < и >.

8.10. Добавьте в класс `Predicate` обработку произвольных комбинаций логических операторов `and`, `or` и `not`.

8.11. В упражнении 6.13 вам было предложено добавить в диспетчер записей SimpleDB поддержку значений `null`. Теперь добавьте такую же поддержку в обработчик запросов. В частности:

- внесите необходимые изменения в класс `Constant`;
- измените методы `getVal` и `setVal` в классе `TableScan` так, чтобы они распознавали и обрабатывали значения `null`;
- определите, в какие из классов `Expression`, `Term` и `Predicate` нужно добавить обработку констант `null`.

8.12. Измените класс `ProjectScan` так, чтобы он стал обновляемым образом сканирования.

8.13. В упражнении 6.10 вам было предложено добавить в класс `TableScan` методы `previous` и `afterLast`.

a) Измените код SimpleDB так, чтобы все образы сканирований имели эти методы.
b) Напишите программу для проверки новых методов. Обратите внимание, что вы не сможете протестировать изменения в движке SimpleDB, не расширив его реализацию JDBC. См. упражнение 11.5.

8.14. Оператор *переименования rename* принимает три аргумента: имя таблицы, имя поля в таблице и новое имя поля, и возвращает таблицу, идентичную исходной, за исключением того, что в ней указанное поле имеет новое имя. Например, следующий запрос переименовывает поле `SName` в `StudentName`:

```
rename(STUDENT, SName, StudentName)
```

Напишите класс `RenameScan`, реализующий этот оператор. Этот класс понадобится в упражнении 10.13.

8.15. Оператор *расширения extend* принимает три аргумента: имя таблицы, выражение и новое имя поля, и возвращает таблицу, идентичную исходной, за исключением дополнительного нового поля, значение которого определяется указанным выражением. Например, следующий запрос добавляет в таблицу STUDENT новое поле `JuniorYear`, которое вычисляет год, когда студент обучался на предпоследнем курсе:

```
extend(STUDENT, GradYear-1, JuniorYear)
```

Напишите класс `ExtendScan`, реализующий этот оператор. Этот класс понадобится в упражнении 10.14.

8.16. Реляционный оператор *объединения union* принимает два аргумента – имена двух таблиц – и возвращает новую таблицу с записями, присутствующими в исходных таблицах. Оператор union требует, чтобы обе исходные таблицы имели одинаковые схемы; новая таблица также будет иметь эту схему. Напишите класс `UnionScan`, реализующий этот оператор. Данный класс понадобится в упражнении 10.15.

8.17. Оператор *полусоединения semijoin* принимает три аргумента: две таблицы и предикат – и возвращает записи из первой таблицы, для которых имеется «соответствующая» запись во второй таблице. Например, следующий запрос должен вернуть специальности, выбранные в качестве основных хотя бы одним студеном:

```
semijoin(DEPT, STUDENT, Did=MajorId)
```

Аналогично, оператор *антисоединения antijoin* возвращает записи из первой таблицы, для которых отсутствуют «соответствующие» записи во второй таблице. Например, следующий запрос должен вернуть специальности, не выбранные в качестве основных ни одним студеном:

```
antijoin(DEPT, STUDENT, Did=MajorId)
```

Напишите классы `SemijoinScan` и `AntijoinScan`, реализующие эти операторы. Эти классы понадобятся в упражнении 10.16.

Глава 9

Синтаксический анализ

Клиент JDBC посылает оператор SQL движку базы данных в виде строки. Движок должен извлечь из этой строки всю необходимую информацию и создать дерево запроса. Этот процесс извлечения состоит из двух этапов: синтаксического анализа и семантического анализа, известного как планирование. Данная глава посвящена этапу синтаксического анализа. Планирование рассматривается в главе 10.

9.1. Синтаксис и семантика

Синтаксис языка – это набор правил, описывающих строки, которые могут содержать значимые операторы. Например, взгляните на следующую строку:

```
select from tables T1 and T2 where b - 3
```

Ниже перечислены причины, почему эта строка не является синтаксически правильным оператором:

- предложение `select` должно что-то содержать;
- идентификатор `tables` не является ключевым словом и будет интерпретироваться как имя таблицы;
- имена таблиц должны отделяться друг от друга запятыми, а не ключевым словом `and`;
- подстрока «b – 3» не является допустимым предикатом.

Каждая из названных причин превращает эту строку в совершенно бессмысленный оператор SQL. Движок базы данных просто не сможет понять, как выполнить его, что бы ни означали идентификаторы `tables`, `T1`, `T2` и `b`.

Семантика языка определяет фактический смысл синтаксически правильной строки. Взгляните на следующую синтаксически допустимую строку:

```
select a from x, z where b = 3
```

Глядя на эту строку, можно сделать вывод, что этот оператор запрашивает одно поле (с именем `a`) из двух таблиц (с именами `x` и `z`) и имеет предикат `b = 3`. То есть данный оператор, возможно, имеет смысл.

Является ли оператор действительно значимым, зависит от *семантической информации* о `x`, `z`, `a` и `b`. В частности, `x` и `z` должны быть именами таблиц, эти таблицы должны содержать поле с именем `a` и числовое поле с именем `b`. Эту семантическую информацию можно получить из метаданных. Синтаксический

анализатор ничего не знает о метаданных и, следовательно, не может оценить значимость оператора SQL. Поэтому вся ответственность за проверку метаданных возлагается на планировщика, который будет обсуждаться в главе 10.

9.2. Лексический анализ

Первым делом синтаксический анализатор должен разбить входную строку на фрагменты, называемые *лексемами*, или *токенами*. Компонент, который решает эту задачу, называется *лексическим анализатором*.

Каждая лексема имеет *тип* и *значение*. Лексический анализатор в SimpleDB поддерживает пять типов лексем:

- односимвольные *разделители*, например запятые;
- *целочисленные константы*, такие как `123`;
- *строковые константы*, такие как `'joe'`;
- *ключевые слова*, такие как `select`, `from` и `where`;
- *идентификаторы*, такие как `STUDENT`, `x` и `glop34a`.

Пробельные символы (пробелы, табуляции и символы перевода строки), как правило, не являются частью лексем, за исключением строковых констант. Пробелы используются, чтобы улучшить читаемость и отделить лексемы друг от друга.

Вернемся к предыдущему оператору SQL:

```
select a from x, z where b = 3
```

Получив эту строку, лексический анализатор разобьет ее на десять лексем, перечисленных в табл. 9.1.

Таблица 9.1. Лексемы, произведенные лексическим анализатором

Тип	Значение
ключевое слово	select
идентификатор	a
ключевое слово	from
идентификатор	x
разделитель	,
идентификатор	z
ключевое слово	where
идентификатор	b
разделитель	=
целочисленная константа	3

Концептуально лексический анализатор действует просто: он читает входную строку по одному символу за раз, приостанавливаясь, когда выясняется, что очередная лексема прочитана. Сложность лексического анализатора прямо пропорциональна набору типов лексем: чем больше поддерживаемых типов, тем сложнее реализация.

Язык Java предлагает два встроенных лексических анализатора в виде классов `StringTokenizer` и `StreamTokenizer`. Класс `StringTokenizer` проще в использовании, но он поддерживает только два вида лексем: разделители и слова (под-

строки между разделителями). Он не подходит для SQL, отчасти потому, что не различает числа и строки в кавычках. Класс `StreamTokenizer`, напротив, поддерживает обширный набор типов лексем, включая все пять типов, используемых в SimpleDB.

В листинге 9.1 приводится определение класса `TokenizerTest`, демонстрирующего порядок использования `StreamTokenizer`. Код разбивает заданную строку на лексемы и выводит тип и значение каждой.

Листинг 9.1. Определение класса TokenizerTest

```
public class TokenizerTest {
    private static Collection<String> keywords =
        Arrays.asList("select", "from", "where", "and", "insert",
                      "into", "values", "delete", "update", "set",
                      "create", "table","int", "varchar", "view", "as",
                      "index", "on");

    public static void main(String[] args) throws IOException {
        String s = getStringFromUser();
        StreamTokenizer tok = new StreamTokenizer(new StringReader(s));
        tok.ordinaryChar('.');
        tok.wordChars('_', '_');
        tok.lowerCaseMode(true); // для преобразования идентификаторов и
                                 // ключевых слов в нижний регистр
        while (tok.nextToken() != TT_EOF)
            printCurrentToken(tok);
    }

    private static String getStringFromUser() {
        System.out.println("Enter tokens:");
        Scanner sc = new Scanner(System.in);
        String s = sc.nextLine();
        sc.close();
        return s;
    }

    private static void printCurrentToken(StreamTokenizer tok)
            throws IOException {
        if (tok.ttype == TT_NUMBER)
            System.out.println("IntConstant " + (int)tok.nval);
        else if (tok.ttype == TT_WORD) {
            String word = tok.sval;
            if (keywords.contains(word))
                System.out.println("Keyword " + word);
            else
                System.out.println("Id " + word);
        }
        else if (tok.ttype == '\'')
            System.out.println("StringConstant " + tok.sval);
        else
            System.out.println("Delimiter " + (char)tok.ttype);
    }
}
```

Вызов `tok.ordinaryChar('.')` требует от лексического анализатора, чтобы тот интерпретировал точку как разделитель. (Даже притом что точки не исполь-

зуются в SimpleDB, их важно идентифицировать как разделители, чтобы исключить возможность появления в идентификаторах.) Вызов `tok.wordChars('_', '_')`, напротив, требует, чтобы анализатор интерпретировал символы подчеркивания как часть идентификаторов. Вызов `tok.lowerCaseMode(true)` сообщает анализатору, что тот должен преобразовать все строковые лексемы (кроме строк в кавычках) в нижний регистр; это позволит не учитывать регистр символов в ключевых словах и идентификаторах.

Метод `nextToken` переносит указатель текущей позиции в начало следующей лексемы в потоке; возвращаемое значение `TT_EOF` указывает, что лексем больше нет. Общедоступная переменная `ttype` анализатора хранит тип текущей лексемы: значение `TT_NUMBER` соответствует числовой константе, `TT_WORD` обозначает идентификатор или ключевое слово, а целочисленное представление одиночной кавычки обозначает строковую константу. Тип односимвольной лексемы-разделителя обозначается целочисленным представлением соответствующего символа.

9.3. Лексический анализатор в SimpleDB

Класс `SteamTokenizer` – это универсальный лексический анализатор, но не всегда удобен в использовании. Класс `Lexer` в SimpleDB предоставляет синтаксическому анализатору более простой доступ к потоку лексем. В нем имеются методы двух видов, которые может вызвать синтаксический анализатор: методы, запрашивающие текущую лексему, и методы, требующие от лексического анализатора «поглотить» текущую лексему, вернуть ее значение и перейти к следующей. Для каждого типа лексем имеется соответствующая пара методов. API для этих десяти методов показан в листинге 9.2.

Листинг 9.2. API лексического анализатора в SimpleDB

Lexer

```
public boolean matchDelim(char d);
public boolean matchIntConstant();
public boolean matchStringConstant();
public boolean matchKeyword(String w);
public boolean matchId();

public void    eatDelim(char d);
public int     eatIntConstant();
public String  eatStringConstant();
public void    eatKeyword(String w);
public String  eatId();
```

Первые пять методов возвращают информацию о текущей лексеме. Метод `matchDelim` возвращает `true`, если текущая лексема является разделителем с указанным значением. Аналогично, `matchKeyword` возвращает `true`, если текущая лексема является ключевым словом с указанным значением. Три других метода `matchXXX` возвращают `true`, если текущая лексема имеет соответствующий тип.

Последние пять методов «поглощают» текущую лексему. Каждый метод вызывает соответствующий ему метод `matchXXX`. Если он вернет `false`, генерируется исключение; иначе текущей становится следующая лексема. Кроме того,

методы `eatIntConstant`, `eatStringConstant` и `eatId` возвращают значение текущей лексемы.

Порядок использования этих методов иллюстрирует класс `LexerTest` в листинге 9.3. Он читает входные строки и анализирует их, предполагая, что каждая строка имеет формат «A = c» или «c = A», где A – это идентификатор, а c – это целочисленная константа. Для любых других строк генерируется исключение.

Листинг 9.3. Класс LexerTest

```
public class LexerTest {
    public static void main(String[] args) {
        String x = "";
        int y = 0;
        Scanner sc = new Scanner(System.in);
        while (sc.hasNext()) {
            String s = sc.nextLine();
            Lexer lex = new Lexer(s);
            if (lex.matchId()) {
                x = lex.eatId();
                lex.eatDelim('=');
                y = lex.eatIntConstant();
            }
            else {
                y = lex.eatIntConstant();
                lex.eatDelim('=');
                x = lex.eatId();
            }
            System.out.println(x + " equals " + y);
        }
        sc.close();
    }
}
```

Определение класса `Lexer` приводится в листинге 9.4. Его конструктор создает экземпляр `StreamTokenizer`. Метод `initKeywords` создает коллекцию ключевых слов, используемых в версии SQL для SimpleDB.

Листинг 9.4. Определение класса Lexer в SimpleDB

```
public class Lexer {
    private Collection<String> keywords;
    private StreamTokenizer tok;

    public Lexer(String s) {
        initKeywords();
        tok = new StreamTokenizer(new StringReader(s));
        tok.ordinaryChar('.');
        tok.wordChars('_', '_');
        tok.lowerCaseMode(true);
        nextToken();
    }

    // Методы для проверки текущей лексемы

    public boolean matchDelim(char d) {
        return d == (char)tok.ttype;
    }
```

```
public boolean matchIntConstant() {
    return tok.ttype == StreamTokenizer.TT_NUMBER;
}

public boolean matchStringConstant() {
    return '\'' == (char)tok.ttype;
}

public boolean matchKeyword(String w) {
    return tok.ttype == StreamTokenizer.TT_WORD &&
                        tok.sval.equals(w);
}

public boolean matchId() {
    return tok.ttype == StreamTokenizer.TT_WORD &&
                        !keywords.contains(tok.sval);
}

// Методы для "поглощения" текущей лексемы

public void eatDelim(char d) {
    if (!matchDelim(d))
        throw new BadSyntaxException();
    nextToken();
}

public int eatIntConstant() {
    if (!matchIntConstant())
        throw new BadSyntaxException();
    int i = (int) tok.nval;
    nextToken();
    return i;
}

public String eatStringConstant() {
    if (!matchStringConstant())
        throw new BadSyntaxException();
    String s = tok.sval;
    nextToken();
    return s;
}

public void eatKeyword(String w) {
    if (!matchKeyword(w))
        throw new BadSyntaxException();
    nextToken();
}

public String eatId() {
    if (!matchId())
        throw new BadSyntaxException();
    String s = tok.sval;
    nextToken();
    return s;
}
```

```
    private void nextToken() {
        try {
            tok.nextToken();
        }
        catch(IOException e) {
            throw new BadSyntaxException();
        }
    }

    private void initKeywords() {
        keywords = Arrays.asList("select", "from", "where", "and",
                         "insert","into", "values", "delete", "update",
                         "set", "create", "table", "varchar",
                         "int", "view", "as", "index", "on");
    }
}
```

Метод `nextToken` класса `StreamTokenizer` может сгенерировать исключение `IOException`. Метод `nextToken` класса `Lexer` преобразует это исключение в исключение `BadSyntaxException`, которое возвращается клиенту (и превращается в исключение `SQLException`, как будет описано в главе 11).

9.4. Грамматика

Грамматика – это набор правил, описывающих порядок объединения лексем в допустимые сочетания. Вот пример грамматического правила:

```
<Field> := IdTok
```

В левой части правила определяется *синтаксическая категория*. Синтаксическая категория обозначает определенное понятие языка. В данном примере <*Field*> обозначает понятие имени поля. Правая часть правила – это шаблон, который определяет набор строк, принадлежащих синтаксической категории. Здесь это просто `IdTok` – строка, соответствующая любой лексеме-идентификатору. То есть <Field> – это множество строк, соответствующих идентификаторам.

Каждую синтаксическую категорию можно рассматривать как отдельный мини-язык. Например, «SName» и «Glop» являются элементами множества <Field>. Имейте в виду, что идентификаторы не должны иметь какого-то предопределенного смысла – они всего лишь идентификаторы. Таким образом, строка «Glop» – вполне допустимый представитель <Field>, даже в университетской базе данных SimpleDB. Однако строка «select» не годится на роль элемента <Field>, потому что представляет ключевое слово, а не идентификатор.

Шаблон справа в грамматическом правиле может содержать ссылки на лексемы и на синтаксические категории. Лексемы с общеизвестными значениями (например, ключевые слова и разделители) указываются явно. Другие лексемы (идентификаторы, целочисленные и строковые константы) записываются как `IdTok`, `IntTok` и `StrTok` соответственно. В шаблонах также встречаются три метасимвола («[», «]» и «|»); они не являются разделителями в языке, поэтому их можно использовать для создания шаблонов. Для иллюстрации рассмотрим четыре дополнительных правила грамматики:

```
<Constant>   := StrTok | IntTok
<Expression> := <Field> | <Constant>
<Term>       := <Expression> = <Expression>
<Predicate>  := <Term> [ AND <Predicate> ]
```

Первое правило определяет категорию <Constant>, обозначающую любую константу – строковую или целочисленную. Метасимвол «|» означает «или». Поэтому категория <Constant> соответствует строковым или целочисленным лексемам и ей (как языку) соответствуют любые строковые и целочисленные константы.

Второе правило определяет категорию <Expression> – выражения без операторов. Правило указывает, что выражением может быть или имя поля, или константа.

Третье правило определяет категорию <Term> – простые условия проверки равенства выражений (как в классе `Term` в SimpleDB). Например, следующие строки принадлежат <Term>:

```
DeptId = DId
'math' = DName
SName = 123
65 = 'abc'
```

Напомню, что синтаксический анализатор не проверяет согласованность типов; поэтому последняя строка синтаксически верна, даже притом что семантически она неправильна.

Четвертое правило определяет категорию <Predicate>, которая обозначает логическое объединение условий, подобно классу `Predicate` в SimpleDB. Метасимволы «[» и «]» обозначают что-то необязательное. Таким образом, правая часть правила соответствует любой последовательности лексем, которые составляют либо <Term>, либо <Term>, за которым следует лексема ключевого слова `AND` и (рекурсивно) другой <Predicate>. Например, все следующие строки соответствуют категории <Predicate>:

```
DName = 'math'
Id = 3 AND DName = 'math'
MajorId = DId AND Id = 3 AND DName = 'math'
```

Первая строка имеет простую форму <*Term*>. Две другие строки – форму <*Term*> *AND* <*Predicate*>.

Если строка принадлежит определенной синтаксической категории, это можно продемонстрировать, нарисовав *дерево разбора* (или *синтаксическое дерево*). Внутренние узлы в дереве разбора представлены синтаксическими категориями, а листья – лексемами. Дочерние узлы узла-категории соответствуют примененному грамматическому правилу. Например, на рис. 9.1 изображено дерево разбора для следующей строки:

```
DName = 'math' AND GradYear = SName
```

На этом рисунке листья дерева расположены вдоль нижнего края дерева, чтобы нагляднее изобразить входную строку. Начиная с корневого узла это дерево утверждает, что вся строка является <Predicate>, потому что `DName='math'` – это <Term>, а `GradYear=SName` – это <Predicate>. Каждое поддерево разворачивается аналогично. Например, `DName='math'` – это <Term>, потому что `DName` и `'math'` являются представителями <Expression>.

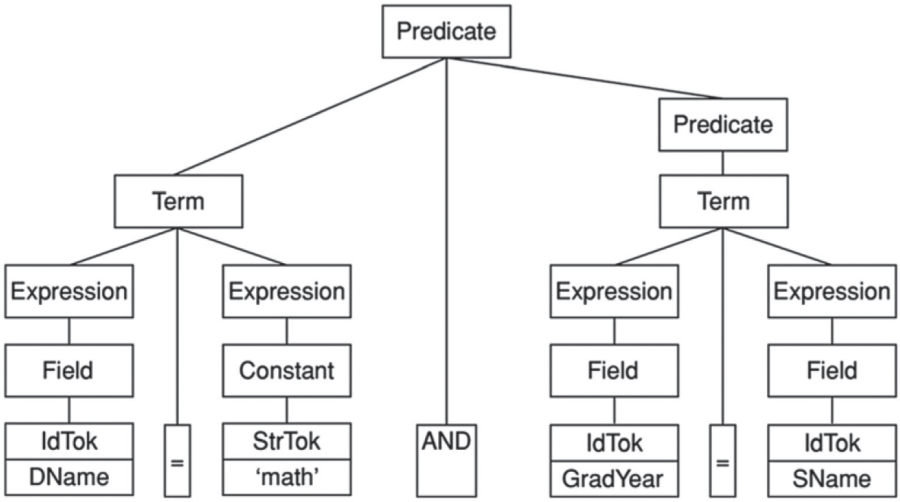


Рис. 9.1. Дерево разбора строки DName = 'math' AND GradYear = SName

В листинге 9.5 перечислены все грамматические правила подмножества SQL, поддерживаемого в SimpleDB. Правила разделены на девять разделов: один раздел для общих конструкций, таких как предикаты, выражения и поля, один раздел для запросов и семь разделов для разных операторов, производящих изменения.

Листинг 9.5. Грамматика подмножества SQL в SimpleDB

```
<Field>       := IdTok
<Constant>    := StrTok | IntTok
<Expression>  := <Field> | <Constant>
<Term>        := <Expression> = <Expression>
<Predicate>   := <Term> [ AND <Predicate> ]

<Query>       := SELECT <SelectList> FROM <TableList> [ WHERE <Predicate> ]
<SelectList>  := <Field> [ , <SelectList> ]
<TableList>   := IdTok [ , <TableList> ]

<UpdateCmd>   := <Insert> | <Delete> | <Modify> | <Create>
<Create>      := <CreateTable> | <CreateView> | <CreateIndex>

<Insert>      := INSERT INTO IdTok ( <FieldList> ) VALUES ( <ConstList> )
<FieldList>   := <Field> [ , <FieldList> ]
<ConstList>   := <Constant> [ , <ConstList> ]

<Delete>      := DELETE FROM IdTok [ WHERE <Predicate> ]

<Modify>      := UPDATE IdTok SET <Field> = <Expression> [ WHERE <Predicate> ]

<CreateTable> := CREATE TABLE IdTok ( <FieldDefs> )
<FieldDefs>   := <FieldDef> [ , <FieldDefs> ]
<FieldDef>    := IdTok <TypeDef>
<TypeDef>     := INT | VARCHAR ( IntTok )
```

```
<CreateView>  := CREATE VIEW IdTok AS <Query>

<CreateIndex> := CREATE INDEX IdTok ON IdTok ( <Field> )
```

Списки элементов широко используются в SQL. Например, предложение `select` может содержать список полей, разделенных запятыми, предложение `from` – список идентификаторов, так же разделенных запятыми, а предложение `where` – список условий, разделенных `AND`. Все списки определяются в грамматике с использованием одного и того же рекурсивного приема, который был продемонстрирован в определении категории <Predicate>. Также обратите внимание, как форма записи необязательных элементов в квадратных скобках используется в правилах <Query>, <Delete> и <Modify> для определения необязательного предложения `where`.

Выше я упоминал, что синтаксический анализатор не проверяет совместимость типов, потому что не знает типы идентификаторов, которые он видит. Также парсер не проверяет совместимость размеров списков. Например, в SQL-операторе `insert` количество значений в <ConstList> должно соответствовать количеству полей в <FieldList>, но грамматическое правило <Insert> требует только наличия списков <FieldList> и <ConstList> в строке. Проверка соответствия размеров списков и типов их элементов возлагается на планировщик[1].

9.5. Алгоритм рекурсивного спуска

Дерево разбора можно рассматривать как доказательство, что данная строка синтаксически допустима. Но как определить дерево разбора? Как движок базы данных сможет определить, является ли строка синтаксически допустимой?

Разработчики языков программирования создали для этой цели множество алгоритмов синтаксического анализа. Сложность таких алгоритмов обычно прямо пропорциональна сложности поддерживаемых грамматик. К счастью, наша грамматика SQL настолько проста, насколько это вообще возможно, поэтому для синтаксического анализа можно использовать простейший из возможных алгоритмов, который называется *рекурсивным спуском*.

В простейшем анализаторе на основе алгоритма рекурсивного спуска каждая синтаксическая категория реализуется методом, не возвращающим значение. Вызов этого метода «поглотит» те лексемы, которые составляют дерево разбора для этой категории. Метод сгенерирует исключение, встретив лексемы, не соответствующие дереву разбора для этой категории.

Рассмотрим первые пять правил грамматики в листинге 9.5, которые образуют подмножество SQL, соответствующее предикатам. Java-класс, реализующий эту грамматику, показан в листинге 9.6.

---

[1] Конечно, намного предпочтительнее было бы иметь грамматику, проверяющую равенство размеров списков. Однако такая грамматика невозможна, что можно доказать с помощью теории автоматов.

Листинг 9.6. Реализация простейшего разбора предикатов на основе алгоритма рекурсивного спуска

```
public class PredParser {

    private Lexer lex;

    public PredParser(String s) {
        lex = new Lexer(s);
    }

    public void field() {
        lex.eatId();
    }

    public void constant() {
        if (lex.matchStringConstant())
            lex.eatStringConstant();
        else
            lex.eatIntConstant();
    }

    public void expression() {
        if (lex.matchId())
            field();
        else
            constant();
    }

    public void term() {
        expression();
        lex.eatDelim('=');
        expression();
    }

    public void predicate() {
        term();
        if (lex.matchKeyword("and")) {
            lex.keyword("and");
            predicate();
        }
    }
}
```

Рассмотрим метод `field`, который вызывает лексический анализатор (и игнорирует любые возвращаемые значения). Если исследуемая лексема является идентификатором, то вызов завершится успешно и лексема будет поглощена. Если нет, метод сгенерирует исключение и вернет его вызывающей стороне. Аналогично действует метод `term`. Он вызовет сначала метод `expression`, который поглотит лексемы, соответствующие одному выражению SQL, затем метод `eatDelim`, который поглотит лексему со знаком равенства, а потом снова метод `expression`, который поглотит лексемы, соответствующие другому выражению SQL. Если какой-либо из этих вызовов не найдет ожидаемых лексем, он сгенерирует исключение, которое `term` передаст вызывающей стороне.

Грамматические правила с альтернативами реализуются с помощью операторов `if`. Условное выражение в операторе `if` проверит текущую лексему, чтобы решить, что делать. В качестве тривиального примера рассмотрим метод `constant`. Если текущая лексема является строковой константой, метод поглотит ее; иначе он попытается поглотить целочисленную константу. Если текущая лексема не является ни строковой, ни целочисленной константой, то вызов `lex.eatIntConstant` сгенерирует исключение. В качестве более сложного примера рассмотрим метод `expression`. Этот метод знает, что если текущая лексема является идентификатором, то он должен интерпретировать ее как имя поля; иначе – как константу[1].

Метод `predicate` иллюстрирует реализацию рекурсивного правила. Сначала он вызывает метод `term`, затем проверяет, является ли текущая лексема ключевым словом `AND`. Если это действительно лексема `AND`, то поглощает ее и рекурсивно вызывает самого себя. Если текущая лексема отличается от `AND`, то метод делает вывод, что только что проверил последнее условие в списке, и возвращает управление. Следовательно, вызов `predicate` поглотит ровно столько лексем, сколько сумеет, – если он увидит лексему `AND`, то продолжит выполнение, даже если перед этим он уже видел действительный предикат.

Интересно отметить, что при разборе методом рекурсивного спуска последовательность вызовов методов определяет дерево разбора для входной строки. В упражнении 9.4 вам будет предложено изменить код каждого метода, чтобы он выводил свое имя с соответствующим отступом; результат будет напоминать дерево разбора, повернутое набок.

9.6. Добавление действий в синтаксический анализатор

Рассмотренный алгоритм синтаксического анализа методом рекурсивного спуска просто благополучно возвращает управление, когда входная строка синтаксически допустима. Такое поведение интересно само по себе, но не особенно полезно. Мы должны изменить анализатор, чтобы он возвращал информацию, необходимую планировщику. Это изменение называется добавлением *действий* в анализатор.

Как правило, синтаксический анализатор SQL должен извлекать из оператора SQL такие сведения, как имена таблиц и полей, предикаты и константы. Что именно будет извлечено, зависит от вида оператора SQL:

- *для запроса*: список имен полей (из предложения `select`), коллекция имен таблиц (из предложения `from`) и предикат (из предложения `where`);
- *для операции вставки*: имя таблицы, список имен полей и список значений;
- *для операции удаления*: имя таблицы и предикат;

---

[1] Этот пример также демонстрирует ограничения синтаксического анализа на основе алгоритма рекурсивного спуска. Если грамматическое правило имеет две альтернативы с одинаковыми первыми лексемами, алгоритм не сможет определить, какую альтернативу выбрать, и рекурсивный спуск не будет работать. Фактически грамматика в листинге 9.5 имеет именно эту проблему. В упражнение 9.3 вам будет предложено решить ее.

- *для операции изменения*: имя таблицы, имя изменяемого поля, выражение, определяющее новое значение для поля, и предикат;
- *для операции создания таблицы*: имя таблицы и схема;
- *для операции создания представления*: имя таблицы и определение представления;
- *для операции создания индекса*: имя индекса, имя таблицы и имя индексируемого поля.

Эту информацию можно извлечь из потока лексем с помощью возвращаемых значений методов класса `Lexer`. Стратегия изменения методов синтаксического анализатора проста: получить значения, возвращаемые вызовами `eatId`, `eatStringConstant` и `eatIntConstant`, собрать их в подходящий объект и вернуть этот объект вызывающей стороне.

В листинге 9.7 представлено определение класса `Parser`, методы которого реализуют грамматику из листинга 9.5. Более подробно этот класс рассматривается в следующих подразделах.

Листинг 9.7. Определение класса Parser в SimpleDB

```
public class Parser {
    private Lexer lex;

    public Parser(String s) {
        lex = new Lexer(s);
    }

    // Методы для разбора предикатов и их компонентов

    public String field() {
        return lex.eatId();
    }

    public Constant constant() {
        if (lex.matchStringConstant())
            return new Constant(lex.eatStringConstant());
        else
            return new Constant(lex.eatIntConstant());
    }

    public Expression expression() {
        if (lex.matchId())
            return new Expression(field());
        else
            return new Expression(constant());
    }

    public Term term() {
        Expression lhs = expression();
        lex.eatDelim('=');
        Expression rhs = expression();
        return new Term(lhs, rhs);
    }
```

```
public Predicate predicate() {
    Predicate pred = new Predicate(term());
    if (lex.matchKeyword("and")) {
        lex.eatKeyword("and");
        pred.conjoinWith(predicate());
    }
    return pred;
}

// Методы для разбора запросов

public QueryData query() {
    lex.eatKeyword("select");
    List<String> fields = selectList();
    lex.eatKeyword("from");
    Collection<String> tables = tableList();
    Predicate pred = new Predicate();
    if (lex.matchKeyword("where")) {
        lex.eatKeyword("where");
        pred = predicate();
    }
    return new QueryData(fields, tables, pred);
}

private List<String> selectList() {
    List<String> L = new ArrayList<String>();
    L.add(field());
    if (lex.matchDelim(',')) {
        lex.eatDelim(',');
        L.addAll(selectList());
    }
    return L;
}

private Collection<String> tableList() {
    Collection<String> L = new ArrayList<String>();
    L.add(lex.eatId());
    if (lex.matchDelim(',')) {
        lex.eatDelim(',');
        L.addAll(tableList());
    }
    return L;
}

// Методы для разбора разных команд обновления

public Object updateCmd() {
    if (lex.matchKeyword("insert"))
        return insert();
    else if (lex.matchKeyword("delete"))
        return delete();
    else if (lex.matchKeyword("update"))
        return modify();
    else
        return create();
}
```

```
private Object create() {
    lex.eatKeyword("create");
    if (lex.matchKeyword("table"))
        return createTable();
    else if (lex.matchKeyword("view"))
        return createView();
    else
        return createIndex();
}

// Методы для разбора команд удаления
public DeleteData delete() {
    lex.eatKeyword("delete");
    lex.eatKeyword("from");
    String tblname = lex.eatId();
    Predicate pred = new Predicate();
    if (lex.matchKeyword("where")) {
        lex.eatKeyword("where");
        pred = predicate();
    }
    return new DeleteData(tblname, pred);
}

// Методы для разбора команд вставки
public InsertData insert() {
    lex.eatKeyword("insert");
    lex.eatKeyword("into");
    String tblname = lex.eatId();
    lex.eatDelim('(');
    List<String> flds = fieldList();
    lex.eatDelim(')');
    lex.eatKeyword("values");
    lex.eatDelim('(');
    List<Constant> vals = constList();
    lex.eatDelim(')');
    return new InsertData(tblname, flds, vals);
}

private List<String> fieldList() {
    List<String> L = new ArrayList<String>();
    L.add(field());
    if (lex.matchDelim(',')) {
        lex.eatDelim(',');
        L.addAll(fieldList());
    }
    return L;
}

private List<Constant> constList() {
    List<Constant> L = new ArrayList<Constant>();
    L.add(constant());
    if (lex.matchDelim(',')) {
        lex.eatDelim(',');
        L.addAll(constList());
    }
    return L;
}
```

```
// Метод для разбора команд изменения
public ModifyData modify() {
    lex.eatKeyword("update");
    String tblname = lex.eatId();
    lex.eatKeyword("set");
    String fldname = field();
    lex.eatDelim('=');
    Expression newval = expression();
    Predicate pred = new Predicate();
    if (lex.matchKeyword("where")) {
        lex.eatKeyword("where");
        pred = predicate();
    }
    return new ModifyData(tblname, fldname, newval, pred);
}

// Метод для разбора команд создания таблиц
public CreateTableData createTable() {
    lex.eatKeyword("table");
    String tblname = lex.eatId();
    lex.eatDelim('(');
    Schema sch = fieldDefs();
    lex.eatDelim(')');
    return new CreateTableData(tblname, sch);
}

private Schema fieldDefs() {
    Schema schema = fieldDef();
    if (lex.matchDelim(',')) {
        lex.eatDelim(',');
        Schema schema2 = fieldDefs();
        schema.addAll(schema2);
    }
    return schema;
}

private Schema fieldDef() {
    String fldname = field();
    return fieldType(fldname);
}

private Schema fieldType(String fldname) {
    Schema schema = new Schema();
    if (lex.matchKeyword("int")) {
        lex.eatKeyword("int");
        schema.addIntField(fldname);
    }
    else {
        lex.eatKeyword("varchar");
        lex.eatDelim('(');
        int strLen = lex.eatIntConstant();
        lex.eatDelim(')');
        schema.addStringField(fldname, strLen);
    }
    return schema;
}
```

```
    // Метод для разбора команд создания представлений

    public CreateViewData createView() {
        lex.eatKeyword("view");
        String viewname = lex.eatId();
        lex.eatKeyword("as");
        QueryData qd = query();
        return new CreateViewData(viewname, qd);
    }

    // Метод для разбора команд создания индексов

    public CreateIndexData createIndex() {
        lex.eatKeyword("index");
        String idxname = lex.eatId();
        lex.eatKeyword("on");
        String tblname = lex.eatId();
        lex.eatDelim('(');
        String fldname = field();
        lex.eatDelim(')');
        return new CreateIndexData(idxname, tblname, fldname);
    }
}
```

9.6.1. Разбор предикатов и выражений

В основе синтаксического анализатора лежат пять правил грамматики, которые определяют предикаты и выражения, потому что они используются для разбора разных видов операторов SQL. Эти методы в классе `Parser` реализованы точно так же, как в классе `PredParser` (листинг 9.6), за исключением того, что теперь они выполняют действия и возвращают значения. В частности, метод `field` извлекает и возвращает имя поля из текущей лексемы. Методы `constant`, `expression`, `term` и `predicate` действуют аналогично и возвращают объекты `Constant`, `Expression`, `Term` и `Predicate` соответственно.

9.6.2. Разбор запросов

Метод `query` реализует синтаксическую категорию <Query>. Разбирая запрос, анализатор получает три элемента, необходимых планировщику, – имена полей, имена таблиц и предикат – и сохраняет их в объекте `QueryData`. Класс `QueryData` открывает доступ к этим значениям посредством методов `fields`, `tables` и `pred` (см. листинг 9.8). Также этот класс имеет метод `toString`, который заново воссоздает строку запроса. Этот метод понадобится при обработке определений представлений.

Листинг 9.8. Определение класса QueryData в SimpleDB

```
public class QueryData {
    private List<String> fields;
    private Collection<String> tables;
    private Predicate pred;
```

```
    public QueryData(List<String> fields, Collection<String> tables, Predicate pred) {
        this.fields = fields;
        this.tables = tables;
        this.pred   = pred;
    }
    public List<String> fields() {
        return fields;
    }

    public Collection<String> tables() {
        return tables;
    }

    public Predicate pred() {
        return pred;
    }

    public String toString() {
        String result = "select ";
        for (String fldname : fields)
            result += fldname + ", ";
        result = result.substring(0, result.length()-2); //удалить последнюю запятую
        result += " from ";
        for (String tblname : tables)
            result += tblname + ", ";
        result = result.substring(0, result.length()-2); //удалить последнюю запятую
        String predstring = pred.toString();
        if (!predstring.equals(""))
            result += " where " + predstring;
        return result;
    }
}
```

9.6.3. Разбор операций обновления

Метод `updateCmd` анализатора реализует синтаксическую категорию <UpdateCmd>, которая представляет группу из нескольких SQL-операторов, выполняющих изменения. Этот метод будет вызываться JDBC-методом `executeUpdate`, чтобы определить вид SQL-оператора. Для идентификации фактической команды метод проверяет первую лексему в строке, а затем передает управление конкретному методу, соответствующему этой команде. Все методы, представляющие операторы изменения, имеют свой тип возвращаемого значения, поскольку все они извлекают разную информацию из входной строки; именно по этой причине метод `updateCmd` возвращает значение обобщенного типа `Object`.

9.6.4. Разбор операций вставки

Метод анализатора `insert` реализует синтаксическую категорию <Insert>. Он извлекает три элемента: имя таблицы, список полей и список значений. Эти значения сохраняются в экземпляре класса `InsertData` (листинг 9.9) и доступны посредством методов чтения.

Листинг 9.9. Определение класса InsertData в SimpleDB

```
public class InsertData {
    private String tblname;
    private List<String> flds;
    private List<Constant> vals;

    public InsertData(String tblname, List<String> flds, List<Constant> vals) {
        this.tblname = tblname;
        this.flds = flds;
        this.vals = vals;
    }

    public String tableName() {
        return tblname;
    }

    public List<String> fields() {
        return flds;
    }

    public List<Constant> vals() {
        return vals;
    }
}
```

9.6.5. Разбор операций удаления

Операторы удаления обрабатываются методом `delete`. Он возвращает объект класса `DeleteData` (листинг 9.10). Конструктор класса сохраняет имя таблицы и предикат из указанного оператора удаления и предоставляет методы `tableName` и `pred` для доступа к ним.

Листинг 9.10. Определение класса DeleteData в SimpleDB

```
public class DeleteData {
    private String tblname;
    private Predicate pred;

    public DeleteData(String tblname, Predicate pred) {
        this.tblname = tblname;
        this.pred = pred;
    }

    public String tableName() {
        return tblname;
    }

    public Predicate pred() {
        return pred;
    }
}
```

9.6.6. Разбор операций изменения

Операторы изменения обрабатываются методом `modify`. Метод возвращает объект класса `ModifyData` (листинг 9.11). Этот класс очень похож на класс `DeleteData`. Разница лишь в том, что этот класс также хранит информацию об операции изменения: имя поля, значение которого требуется изменить, и выражение, определяющее новое значение. Эту информацию возвращают дополнительные методы `targetField` и `newValue`.

Листинг 9.11. Определение класса ModifyData в SimpleDB

```
public class ModifyData {
    private String tblname;
    private String fldname;
    private Expression newval;
    private Predicate pred;

    public ModifyData(String tblname, String fldname, Expression newval, Predicate pred) {
        this.tblname = tblname;
        this.fldname = fldname;
        this.newval = newval;
        this.pred = pred;
    }

    public String tableName() {
        return tblname;
    }

    public String targetField() {
        return fldname;
    }

    public Expression newValue() {
        return newval;
    }

    public Predicate pred() {
        return pred;
    }
}
```

9.6.7. Разбор операций создания таблиц, представлений и индексов

Синтаксическая категория <Create> определяет три SQL-оператора создания, поддерживаемых в SimpleDB. Операторы создания таблиц обрабатываются синтаксической категорией <CreateTable> и ее методом `createTable`. Методы `fieldDef` и `fieldType` извлекают информацию об одном поле и сохраняют ее в объекте `Schema`. Затем метод `fieldDefs` добавляет эту схему в схему таблицы. Имя таблицы и схема возвращаются в виде экземпляра класса `CreateTableData`, определение которого показано в листинге 9.12.

Листинг 9.12. Определение класса CreateTableData в SimpleDB

```
public class CreateTableData {
    private String tblname;
    private Schema sch;

    public CreateTableData(String tblname, Schema sch) {
        this.tblname = tblname;
        this.sch = sch;
    }

    public String tableName() {
        return tblname;
    }

    public Schema newSchema() {
        return sch;
    }
}
```

Операторы создания представлений обрабатываются методом `createView`. Он извлекает имя и определение представления и возвращает их в виде экземпляра класса `CreateViewData` (листинг 9.13). Определение представления обрабатывается несколько необычно. Его необходимо проанализировать как <Query>, чтобы проверить синтаксическую допустимость. Однако диспетчер метаданных не сохраняет проанализированное определение представления, и ему необходимо воссоздать фактическую строку запроса. По этой причине конструктор `CreateViewData` заново воссоздает определение представления, вызывая метод `toString` возвращаемого объекта `QueryData`. Синтаксический анализатор разбирает запрос, а метод `toString` «собирает» его обратно.

Листинг 9.13. Определение класса CreateViewData в SimpleDB

```
public class CreateViewData {
    private String viewname;
    private QueryData qrydata;

    public CreateViewData(String viewname, QueryData qrydata) {
        this.viewname = viewname;
        this.qrydata = qrydata;
    }

    public String viewName() {
        return viewname;
    }

    public String viewDef() {
        return qrydata.toString();
    }
}
```

Индекс – это структура данных, которую система баз данных использует для ускорения обработки запросов; индексы будут рассматриваться в главе 12. Метод синтаксического анализатора `createIndex` извлекает имя индекса, имя таблицы и имя поля и сохраняет их в экземпляре класса `CreateIndexData` (листинг 9.14).

Листинг 9.14. Определение класса CreateIndexData в SimpleDB

```
public class CreateIndexData {
    private String idxname, tblname, fldname;

    public CreateIndexData(String idxname, String tblname, String fldname) {
        this.idxname = idxname;
        this.tblname = tblname;
        this.fldname = fldname;
    }

    public String indexName() {
        return idxname;
    }

    public String tableName() {
        return tblname;
    }

    public String fieldName() {
        return fldname;
    }
}
```

9.7. Итоги

- *Синтаксис* языка – это набор правил, описывающих строки, которые могут представлять собой значимые операторы.
- *Синтаксический анализатор* проверяет, является ли синтаксически правильной входная строка.
- *Лексический анализатор* – это часть синтаксического анализатора, которая разбивает входную строку на последовательность *лексем*.
- Каждая лексема имеет *тип* и *значение*. Лексический анализатор в SimpleDB поддерживает пять типов лексем:
  - односимвольные *разделители*, такие как запятая;
  - *целочисленные константы*, такие как `123`;
  - *строковые константы*, такие как `'joe'`;
  - *ключевые слова*, такие как `select`, `from` и `where`;
  - *идентификаторы*, такие как `STUDENT`, `x` и `glop34a`.
- Каждый тип лексем имеет два метода: возвращающий текущую лексему и требующий от лексического анализатора «поглотить» текущую лексему, вернуть ее значение и перейти к следующей лексеме.
- *Грамматика* – это набор правил, описывающих допустимые комбинации лексем. В каждом грамматическом правиле:
  - слева определяется *синтаксическая категория*, обозначающая определенное понятие в языке;
  - справа определяется состав этой категории – множество строк, соответствующих данному правилу.
- *Дерево разбора* состоит из внутренних узлов, представленных синтаксическими категориями, и листьев, представленных лексемами. Дочерние узлы категорий соответствуют применению правил грамматики. Строка принадлежит синтаксической категории, если она имеет дерево разбора с данной категорией в качестве корня.

- *Алгоритм разбора* создает дерево разбора из синтаксически допустимой строки. Сложность алгоритма разбора обычно прямо пропорциональна сложности поддерживаемых грамматик. Один из простейших алгоритмов синтаксического разбора известен как *алгоритм рекурсивного спуска*.
- Синтаксический анализатор, использующий алгоритм рекурсивного спуска, имеет отдельный метод для каждого грамматического правила. Каждый такой метод вызывает методы, соответствующие элементам в правой части правила.
- Методы синтаксического анализатора на основе алгоритма рекурсивного спуска извлекают и возвращают значения прочитанных лексем. Анализатор SQL должен извлекать из оператора SQL такие сведения, как имена таблиц и полей, предикаты и константы. Что именно будет извлечено, зависит от вида оператора SQL:
  - *для запроса*: список имен полей (из предложения `select`), коллекция имен таблиц (из предложения `from`) и предикат (из предложения `where`);
  - *для операции вставки*: имя таблицы, список имен полей и список значений;
  - *для операции удаления*: имя таблицы и предикат;
  - *для операции изменения*: имя таблицы, имя изменяемого поля, выражение, определяющее новое значение для поля, и предикат;
  - *для операции создания таблицы*: имя таблицы и схема;
  - *для операции создания представления*: имя таблицы и определение представления;
  - *для операции создания индекса*: имя индекса, имя таблицы и имя индексируемого поля.

9.8. Для дополнительного чтения

Области лексического и синтаксического анализа уделялось огромное внимание на протяжении более 60 лет. Отличное введение в различные алгоритмы, используемые в настоящее время, можно найти в книге (Scott, 2000). В интернете доступно большое количество синтаксических анализаторов SQL, таких как Zql (zql.sourceforge.net). Определение грамматики SQL можно найти в Date and Darwen (2004). Копия стандарта SQL-92, описывающего язык SQL и его грамматику, доступна по адресу www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt. Если прежде вам не доводилось читать документы со стандартами, загляните в него, чтобы получить хотя бы общее представление.

Date, C., & Darwen, H. (2004). «A guide to the SQL standard» (4th ed.). Boston, MA: Addison Wesley.

Scott, M. (2000). «Programming language pragmatics». San Francisco, CA: Morgan Kaufman.

9.9. Упражнения

Теория

9.1. Нарисуйте деревья разбора для следующих операторов SQL:

a) `select a from x where b = 3`
b) `select a, b from x,y,z`
c) `delete from x where a = b and c = 0`
d) `update x set a = b where c = 3`
e) `insert into x (a,b,c) values (3, 'glop', 4)`
f) `create table x ( a varchar(3), b int, c varchar(2) )`

9.2. Для каждой из следующих строк укажите, где будет сгенерировано исключение при ее анализе и почему. Затем выполните каждый запрос в клиенте JDBC и посмотрите, что произойдет:

a) `select from x`
b) `select x x from x`
c) `select x from y z`
d) `select a from where b=3`
e) `select a from y where b -=3`
f) `select a from y where`

9.3. Метод синтаксического анализатора `create` не соответствует грамматике SQL в листинге 9.5.

a) Объясните, почему грамматическое правило для <Create> слишком неоднозначно, чтобы использовать его для анализа методом рекурсивного спуска.
b) Измените грамматику так, чтобы она соответствовала фактической реализации метода `create`.

Практика

9.4. Измените все методы синтаксического анализатора на основе алгоритма рекурсивного спуска так, чтобы они использовали цикл `while` вместо рекурсии.

9.5. Измените класс `PredParser` (в листинге 9.6) так, чтобы он выводил дерево разбора, полученное в результате последовательности вызовов методов.

9.6. В упражнении 8.8 вам предлагалось добавить в реализацию выражений поддержку арифметических операторов с целыми числами.

a) Внесите соответствующие изменения в грамматику SQL.
b) Дополните синтаксический анализатор SimpleDB для реализации изменений в грамматике.
c) Напишите JDBC-клиента для тестирования сервера. Например, напишите программу, выполняющую SQL-запрос, который увеличивает год выпуска для всех учащихся, обучающихся по специальности 30.

9.7. В упражнении 8.9 вам было предложено добавить поддержку операторов сравнения < и >.

a) Внесите соответствующие изменения в грамматику SQL.
b) Дополните синтаксический анализатор SimpleDB для реализации изменений в грамматике.

c) Напишите JDBC-клиента для тестирования сервера. Например, напишите программу, выполняющую SQL-запрос, который извлекает имена всех студентов, выпустившихся до 2010 года.

9.8. В упражнении 8.10 вам было предложено добавить в класс `Predicate` обработку произвольных комбинаций логических операторов `and`, `or` и `not`.

a) Внесите соответствующие изменения в грамматику SQL.
b) Дополните синтаксический анализатор SimpleDB для реализации изменений в грамматике.
c) Напишите JDBC-клиента для тестирования сервера. Например, напишите программу, выполняющую SQL-запрос, который извлекает имена всех студентов, обучающихся по специальности 10 или 20.

9.9. SimpleDB не поддерживает круглые скобки в предикатах.

a) Внесите соответствующие изменения в грамматику SQL (вместе с упражнением 9.8 или без него).
b) Дополните синтаксический анализатор SimpleDB для реализации изменений в грамматике.
c) Напишите JDBC-клиента для тестирования ваших изменений.

9.10. Предикаты соединения таблиц в стандартном SQL можно указать с помощью ключевого слова `JOIN` в предложении `from`. Например, следующие два запроса эквивалентны:

```
select SName, DName
from STUDENT, DEPT
where MajorId = Did and GradYear = 2020

select SName, DName
from STUDENT join DEPT on MajorId = Did
where GradYear = 2020
```

a) Измените лексический анализатор SQL и добавьте поддержку ключевых слов «join» и «on».
b) Внесите изменения в грамматику SQL для поддержки явных соединений.
c) Дополните синтаксический анализатор SimpleDB для реализации изменений в грамматике. Добавьте предикат соединения в предикат, получаемый из предложения `where`.
d) Напишите JDBC-клиента для тестирования ваших изменений.

9.11. В стандартном SQL таблица может иметь связанную *переменную области значений*. Ссылки на поля из этой таблицы включают префикс с именем этой переменной. Например, следующий запрос эквивалентен обоим запросам из упражнения 9.10:

```
select s.SName, d.DName
from STUDENT s, DEPT d
where s.MajorId = d.Did and s.GradYear = 2020
```

a) Внесите в грамматику SQL изменения, необходимые для поддержки этой особенности.

b) Дополните синтаксический анализатор SimpleDB для реализации изменений в грамматике. Вам также нужно изменить информацию, возвращаемую анализатором. Обратите внимание, что вы не сможете протестировать свои изменения в сервере SimpleDB, пока не расширите планировщик, как предлагается в упражнении 10.13.

9.12. Стандартный SQL позволяет использовать ключевое слово `AS` для добавления вычисляемых значений в набор результатов. Например:

```
select SName, GradYear-1 as JuniorYear from STUDENT
```

a) Внесите изменения в грамматику SQL, чтобы разрешить использовать необязательное выражение `AS` после любого имени поля в предложении `select`.

b) Дополните лексический и синтаксический анализаторы SimpleDB для реализации изменений в грамматике. Как в анализаторе можно организовать доступ к этой дополнительной информации? Обратите внимание, что вы не сможете протестировать свои изменения в сервере SimpleDB, пока не расширите планировщик, как предлагается в упражнении 10.14.

9.13. Для объединения результатов двух запросов в стандартном SQL можно использовать ключевое слово `UNION`. Например:

```
select SName from STUDENT where MajorId = 10
union
select SName from STUDENT where MajorId = 20
```

a) Внесите изменения в грамматику SQL для поддержки запросов, объединяющих результаты двух других запросов.

b) Дополните лексический и синтаксический анализаторы SimpleDB для реализации изменений в грамматике. Обратите внимание, что вы не сможете протестировать свои изменения в сервере SimpleDB, пока не расширите планировщик, как предлагается в упражнении 10.15.

9.14. Стандартный SQL поддерживает вложенные запросы в предложении `where`. Например:

```
select SName from STUDENT
where MajorId in select Did from DEPT where DName = 'math'
```

a) Внесите изменения в грамматику SQL для поддержки условий в формате «*fieldname op query*», где *op* может быть «in» или «not in».

b) Дополните лексический и синтаксический анализаторы SimpleDB для реализации изменений в грамматике. Обратите внимание, что вы не сможете протестировать свои изменения в сервере SimpleDB, пока не расширите планировщик, как предлагается в упражнении 10.16.

9.15. В стандартном SQL разрешается использовать символ «*» в предложении `select` для обозначения всех полей таблицы. Если SQL поддерживает переменные области значений (см. упражнение 9.11), то символ «*» также может иметь префикс с именем такой переменной.

a) Внесите изменения в грамматику SQL, чтобы разрешить использовать «*» в запросах.
b) Дополните синтаксический анализатор SimpleDB для реализации изменений в грамматике. Обратите внимание, что вы не сможете протестировать свои изменения в сервере SimpleDB, пока не расширите планировщик, как предлагается в упражнении 10.17.

9.16. В стандартном SQL есть возможность вставки записей в таблицу с использованием следующего варианта оператора `insert`:

```
insert into MATHSTUDENT(SId, SName)
select SId, SName
from STUDENT, DEPT
where MajorId = DId and DName = 'math'
```

То есть в указанную таблицу вставляются записи, возвращаемые оператором `select`. (Оператор выше предполагает, что пустая таблица *MATHSTUDENT* уже была создана.)

a) Внесите изменения в грамматику SQL для поддержки таких операций вставки.
b) Дополните синтаксический анализатор SimpleDB для реализации изменений в грамматике. Обратите внимание, что вы не сможете выполнять JDBC-запросы, пока не расширите планировщик, как предлагается в упражнении 10.18.

9.17. В упражнении 8.7 вам было предложено добавить поддержку новых типов констант.

a) Внесите изменения в грамматику SQL для поддержки этих типов в операциях создания таблиц.
b) Необходимо ли для этого определить новые литералы констант? Если да, измените синтаксическую категорию <Constant>.
c) Дополните синтаксический анализатор SimpleDB для реализации изменений в грамматике.

9.18. В упражнении 8.11 вам было предложено добавить поддержку значений `null`. Теперь добавьте поддержку этих значений в SQL.

a) Внесите изменения в грамматику SQL для поддержки ключевого слова `null` в роли константы.
b) Дополните синтаксический анализатор SimpleDB для реализации изменений в грамматике, внесенных в п. «a».
c) В стандартном SQL условие может иметь вид `GradYear is null` и возвращает `true`, если выражение `GradYear` имеет неопределенное значение. Два ключевых слова `is null` интерпретируются как один оператор с одним аргументом. Внесите изменения в грамматику SQL для поддержки этого нового оператора.
d) Дополните синтаксический анализатор SimpleDB и класс `Term` для реализации изменений в грамматике, внесенных в п. «c».
e) Напишите JDBC-клиента для тестирования ваших изменений. Ваша программа должна записать в поле значение `null` (или использовать неприсвоенное значение в только что вставленной записи)

и затем выполнить запрос с условием `is null`. Обратите внимание, что ваша программа не сможет выводить неопределенные значения, пока вы не измените реализацию JDBC в SimpleDB (см. упражнение 11.6).

9.19. Пакет программного обеспечения с открытым исходным кодом *javacc* (см. javacc.github.io/javacc) создает синтаксические анализаторы на основе грамматических правил. Используйте javacc, чтобы создать анализатор для грамматики SimpleDB. Затем замените существующий анализатор новым.

9.20. Класс `Parser` имеет отдельный метод для каждой синтаксической категории в грамматике. Наша упрощенная грамматика SQL невелика, поэтому сопровождение класса не вызывает сложностей. Однако использование полноценной грамматики привело бы к значительному увеличению класса. Альтернативная стратегия – определить каждую синтаксическую категорию в отдельном классе. Конструктор такого класса может выполнять анализ этой категории. Также в таких классах могут быть методы, возвращающие значения проанализированных лексем. Эта стратегия приводит к созданию большого количества относительно небольших классов. Перепишите синтаксический анализатор SimpleDB, используя эту стратегию.

Глава 10

Планирование

На первом этапе обработки запросов синтаксический анализатор извлекает данные из оператора SQL. Следующий шаг – превращение этих данных в дерево запроса реляционной алгебры – называется *планированием*. В этой главе рассматривается базовый процесс планирования. Здесь описывается, что должен сделать планировщик, чтобы убедиться, что оператор SQL несет смысловую нагрузку, и рассматриваются два очень простых алгоритма построения плана.

Оператор SQL может иметь несколько эквивалентных деревьев запросов, часто с совершенно разной стоимостью. Система баз данных, более или менее конкурентоспособная, должна реализовать алгоритм планирования, отыскивающий эффективные планы. Создание оптимального плана – сложная тема, и она будет рассматриваться в главе 15.

10.1. Проверка

Первая задача планировщика – определить, является ли данный оператор SQL значимым. Планировщик должен проверить:

- наличие в каталоге упоминаемых таблиц и полей;
- однозначность интерпретации имен полей;
- соответствие действий с полями их типам;
- соответствие размеров и типов всех констант их полям.

Всю информацию, необходимую для этих проверок, можно найти в схемах, упомянутых в запросе таблиц. Например, отсутствие схемы означает, что упомянутая таблица не существует. Точно так же отсутствие поля в любой из схем указывает на то, что поле не существует, а его присутствие в нескольких схемах указывает на возможность неоднозначной интерпретации.

Планировщик должен также определить правильность типов предикатов и значений, присваиваемых полям, изучив тип и длину каждого упомянутого поля. Аргументы операторов в выражениях и простых условиях, относящихся к предикатам, должны иметь совместимые типы. Поля и выражения в операциях изменения и вставки также должны иметь совместимые типы.

Планировщик в SimpleDB может получить необходимые схемы таблиц с помощью метода `getLayout` диспетчера метаданных. Однако в данный момент планировщик не выполняет никаких явных проверок. В упражнениях 10.4–10.8 вам будет предложено исправить эту ситуацию.

10.2. Стоимость выполнения дерева запросов

Вторая задача планировщика – создать для оператора SQL дерево запроса реляционной алгебры. Сложность заключается в том, что оператор SQL может иметь много эквивалентных деревьев запросов, каждое из которых имеет свое время выполнения. Планировщик должен выбрать из них наиболее эффективное.

Но как оценить эффективность дерева запросов? Как известно, наиболее важным фактором, влияющим на время выполнения запроса, является количество обращений к блокам. Поэтому *стоимость дерева запросов* определяется как количество блоков, к которым необходимо обратиться, чтобы получить полный образ сканирования запроса.

Стоимость полного образа можно вычислить, выполнив рекурсивный расчет стоимости выполнения его дочерних элементов, а затем применив к ним формулы стоимости в зависимости от типа образа сканирования. В табл. 10.1 перечислены формулы трех функций стоимости. Каждый реляционный оператор имеет свои формулы для этих функций.

Таблица 10.1. Формулы стоимости для образов сканирования

s	B(s)	R(s)	V(s,F)
TableScan(T)	B(T)	R(T)	V(T,F)
SelectScan(s1,A=c)	B(s1)	R(s1) / V(s1,A)	1 если F = A V(s1,F) если F ≠ A
SelectScan(s1,A=B)	B(s1)	R(s1) / max{V(s1,A),V(s1,B)}	min{V(s1,A), V(s1,B)} если F = A,B V(s1,F) если F ≠ A,B
ProjectScan(s1,L)	B(s1)	R(s1)	V(s1,F)
ProductScan(s1,s2)	B(s1) + R(s1)×B(s2)	R(s1)×R(s2)	V(s1,F) если F присутствует в s1 V(s2,F) если F присутствует в s2

B(s) – количество блоков, к которым требуется обратиться, чтобы получить выходной образ s.

R(s) – количество записей в выходном образе s.

V(s, F) – количество уникальных значений F в выходном образе s.

Эти функции аналогичны методам `blocksAccessed`, `recordsOutput` и `distinctValues` диспетчера статистики. Разница лишь в том, что они применяются к образам сканирований вместо таблиц.

Уже при беглом обзоре табл. 10.1 можно заметить взаимосвязь между тремя функциями стоимости. Для любого образа s планировщик должен вычислить B(s). Но если s является произведением двух таблиц, то значение B(s) будет зависеть от суммарного количества блоков в этих двух таблицах, а также от количества записей в левом образе. А если левый образ создается оператором select, то количество записей в нем зависит от числа уникальных значений полей, упомянутых в предикате. Другими словами, планировщику нужны все три функции.

В следующих подразделах подробно рассматриваются функции стоимости, перечисленные в табл. 10.1, и приводятся примеры их использования для расчета стоимости дерева запросов.

10.2.1. Стоимость сканирования таблицы

Образ сканирования каждой таблицы, упомянутой в запросе, содержит текущую страницу записей (`RecordPage`), которая содержит буфер с закрепленной страницей. После чтения записей в этой странице ее буфер открепляется, и его место занимает страница с записями из следующего блока в файле. Таким образом, для одного прохода по образу сканирования таблицы потребуется обратиться к каждому блоку ровно один раз, с закреплением буферов по одному.

То есть если s – это образ сканирования таблицы, то значения B(s), R(s) и V(s, F) определяются как количество блоков, записей и уникальных значений в таблице соответственно.

10.2.2. Стоимость сканирования для оператора селекции

Образ сканирования для оператора селекции (select) опирается на один базовый образ; назовем его s1. Каждый вызов метода `next` образа селекции приведет к одному или нескольким вызовам `s1.next`; метод `s.next` вернет `false`, когда вызов `s1.next` вернет `false`. Каждый вызов `getInt`, `getString` или `getVal` просто запрашивает значение поля у образа `s1` и не требует доступа к блоку. Таким образом, один проход по образу селекции потребует ровно столько же обращений к блокам, сколько потребуется для одного прохода по базовому образу. То есть

```
B(s) = B(s1)
```

Значения R(s) и V(s, F) зависят от предиката селекции. В качестве примера проанализируем частые случаи, когда предикат сравнивает поле с константой или с другим полем.

Сравнение с константой

Допустим, что предикат имеет форму `A = c`, где A – некоторое поле. Предположим, что значения в A распределены равномерно, тогда предикату будет соответствовать R(s1)/V(s1, A) записей. То есть

```
R(s) = R(s1) / V(s1, A)
```

Предположение о равномерном распределении также подразумевает, что в других полях на выходе значения тоже будут распределены равномерно. То есть

```
V(s, A) = 1
V(s, F) = V(s1, F) для всех полей F, отличных от A
```

Сравнение с полем

Теперь допустим, что предикат имеет форму `A = B`, где A и B – имена полей. В этом случае разумно предположить, что значения в полях A и B каким-то связаны. В частности, предположим, что если уникальных значений в поле B больше, чем уникальных значений в поле A (то есть $V(s1, A) < V(s1, B)$), то каждое значение A появится в одной или нескольких записях в поле B; а если уникальных значений в поле A больше, чем уникальных значений в поле B, то верно обратное. (Это предположение типично для случая, когда A является ключом, а B – внешним ключом, или наоборот.) Поэтому предположим,

что уникальных значений в B больше, чем уникальных значений в A, и рассмотрим произвольную запись в s1. Значение ее поля A имеет вероятность 1/V(s1, B) совпасть со значением ее же поля B. Аналогично, если уникальных значений в A больше, чем уникальных значений в B, то значение ее поля B имеет вероятность 1/V(s1, A) совпасть со значением ее же поля A. То есть

```
R(s) = R(s1) / max{V(s1, A), V(s1, B)}
```

Предположение о равномерном распределении также подразумевает, что каждое значение в A будет одинаково вероятно совпадать со значением в B. В результате получаем:

```
V(s, F) = min(V(s1, A), V(s1, B)} для F = A или B
V(s, F) = V(s1, F) для всех полей F, отличных от A или B
```

10.2.3. Стоимость сканирования для оператора проекции

По аналогии с образом селекции, образ проекции (project) опирается на единственный базовый образ (назовем его s1) и не требует дополнительных обращений к блокам, кроме тех, которые требуются базовому образу. Кроме того, оператор проекции не меняет количества записей и не изменяет значений в записях. То есть

```
B(s) = B(s1)
R(s) = R(s1)
V(s, F) = V(s1, F) для всех полей F
```

10.2.4. Стоимость сканирования для оператора прямого произведения

Образ сканирования для оператора прямого произведения (product) опирается на два базовых образа: s1 и s2. Его содержимое определяется как набор всех комбинаций записей из s1 и s2. Для одного обхода образа прямого произведения обход базового образа s1 будет выполнен один раз, а обход базового образа s2 – один раз для каждой записи в s1. Из этого вытекают следующие формулы:

```
B(s) = B(s1) + (R(s1)B(s2))
R(s) = R(s1) × R(s2)
V(s, F) = V(s1, F) или V(s2, F), в зависимости от схемы, которой принадлежит F
```

Самое важное и интересное – формула для B(s) несимметрична относительно s1 и s2. То есть инструкция

```
Scan s3 = new ProductScan(s1, s2);
```

может выполнить другое количество обращений к блокам, чем логически эквивалентная инструкция

```
Scan s3 = new ProductScan(s2, s1);
```

Насколько большой может быть разница? Определим количество записей в блоке (records per block) для образа сканирования s как

```
RPB(s) = R(s) / B(s)
```

То есть RPB(s) – это среднее количество выходных записей, получающихся в результате обращения к каждому блоку s. Теперь формулу, приведенную выше, можно переписать так:

```
B(s) = B(s1) + (RPB(s1) × B(s1) × B(s2))
```

Наибольший вклад в результат вносит член `RPB(s1) × B(s1) × B(s2)`. Если сравнить это выражение с выражением, полученным заменой s1 на s2, то можно заметить, что стоимость получения образа прямого произведения, как правило, будет ниже, когда образ s1 имеет наименьшее значение RPB.

Например, предположим, что s1 – это образ таблицы STUDENT, а s2 – образ таблицы DEPT. Поскольку записи в STUDENT имеют больший размер, чем записи в DEPT, в один блок поместится больше записей DEPT. То есть STUDENT имеет меньшее значение RPB, чем DEPT. Согласно выводам, полученным выше, наименьшее количество обращений к диску произойдет, если роль первого образа будет играть образ таблицы STUDENT.

10.2.5. Конкретный пример

Рассмотрим запрос, возвращающий имена студентов, избравших основной специальностью математику. На рис. 10.1 изображено дерево этого запроса, а в листинге 10.1 представлен фрагмент кода для SimpleDB, получающий образ сканирования запроса.

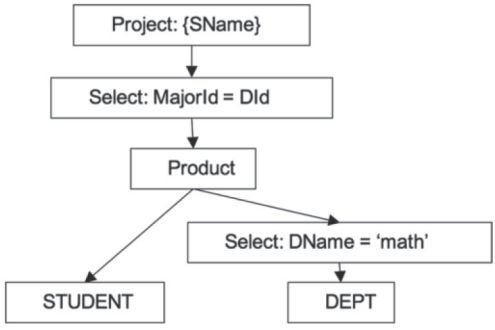


Рис. 10.1. Поиск имен студентов, изучающих математику. Дерево запроса

Листинг 10.1. Код, получающий образ сканирования запроса

```
SimpleDB db = new SimpleDB("studentdb");
Transaction tx = db.newTx();
MetadataMgr mdm = db.mdMgr();
Layout slayout = mdm.getLayout("student", tx);
Layout dlayout = mdm.getLayout("dept", tx);
Scan s1 = new TableScan(tx, "student", slayout);
Scan s2 = new TableScan(tx, "dept", dlayout);
Predicate pred1 = new Predicate(. . .); //dname='math'
Scan s3 = new SelectScan(s2, pred1);
Scan s4 = new ProductScan(s1, s3);
Predicate pred2 = new Predicate(. . .); //majorid=did
Scan s5 = new SelectScan(s4, pred2);
List<String> fields = Arrays.asList("sname");
Scan s6 = new ProjectScan(s5, fields);
```

В табл. 10.2 приводятся расчеты стоимости каждого образа сканирования, создаваемого в листинге 10.1, основанные на статистических метаданных из рис. 7.2. Строки для s1 и s2 просто повторяют статистики для STUDENT и DEPT на рис. 7.2. Строка для s3 сообщает, что выборка по полю `DName` вернет 1 запись, но потребует выполнить поиск в обоих блоках DEPT. Образ s4 вернет все комбинации из 45 000 записей в STUDENT с 1 выбранной записью из DEPT; на выходе получится 45 000 записей. Однако эта операция потребует 94 500 обращений к блокам, потому что одну запись с кафедрой математики потребуется найти 45 000 раз, и каждый раз потребуется выполнить обход образа DEPT с 2 блоками. (Дополнительные 4500 обращений к блокам обусловлены одним обходом образа STUDENT.) Выборка по `MajorId` в образе s5 уменьшает объем выходных результатов до 1125 записей (45 000 студентов / 40 кафедр), но не изменяет числа необходимых обращений к блокам. И как видите, оператор project ничего не меняет.

Таблица 10.2. Стоимости образов сканирования, создаваемых в листинге 10.1

s	B(s)	R(s)	V(s,F)	
s1	4500	45 000	45 000	для F=SId
			44 960	для F=SName
			50	для F=GradYear
			40	для F=MajorId
s2	2	40	40	для F=DId, DName
s3	2	1	1	для F=DId, DName
s4	94 500	45 000	45 000	для F=SId
			44 960	для F=SName
			50	для F=GradYear
			40	для F=MajorId
			1	для F=DId, DName
s5	94 500	1125	1125	для F=SId
			1124	для F=SName
			50	для F=GradYear
			1	для F=MajorId, DId, DName
s6	94 500	1125	1124	для F=SName

Может показаться странным, что система баз данных выполняет поиск записи, соответствующей кафедре математики, 45 000 раз, что требует значительных затрат; однако такова особенность конвейерной обработки запросов. (Фактически это один из примеров ситуаций, когда могут пригодиться неконвейерные реализации, рассматриваемые в главе 13.)

Глядя на цифры RPB для STUDENT и `s3`, можно заметить, что RPB(STUDENT) = 10 и RPB(s3) = 0,5. Поскольку прямое произведение образов выполняется быстрее, когда образ с меньшим значением RPB находится слева, эффективнее было бы определить s4 так:

```
s4 = new ProductScan(s3, STUDENT)
```

В упражнении 10.3 вам будет предложено показать, что в этом случае для выполнения операции потребовалось бы всего 4502 обращения к блокам. Разница обусловлена прежде всего тем, что теперь выборка производится только один раз.

10.3. Планы

Объект, вычисляющий стоимость дерева запроса в SimpleDB, называется *планом*. План реализует интерфейс `Plan`, определение которого показано в листинге 10.2.

Листинг 10.2. Определение интерфейса Plan в SimpleDB

```
public interface Plan {
    public Scan   open();
    public int    blocksAccessed();
    public int    recordsOutput();
    public int    distinctValues(String fldname);
    public Schema schema();
}
```

Этот интерфейс определяет методы `blockAccessed`, `recordsOutput` и `distinctValues`, которые вычисляют значения B(s), R(s) и V(s, F) для запроса. Метод `schema` возвращает схему выходной таблицы. Планировщик запросов может использовать эту схему для проверки типов и поиска способов оптимизации плана. Наконец, каждый план имеет метод `open`, который создает соответствующий образ сканирования.

Планы и образы концептуально схожи в том, что и те, и другие обозначают дерево запросов. Разница в том, что план обращается к метаданным таблиц, упомянутых в запросе, тогда как образ обращается к хранимым данным. Получив SQL-запрос, планировщик базы данных может создать несколько планов для запроса и использовать их метаданные для выбора наиболее эффективного. Затем он вызовет метод `open` выбранного плана, чтобы создать желаемый образ сканирования.

План конструируется аналогично образу. Для каждого оператора реляционной алгебры существует свой класс `Plan`, а также класс `TablePlan` для хранимых таблиц. Например, код в листинге 10.3 извлекает имена студентов, изучающих математику, и реализует тот же запрос, что показан на рис. 10.1. Единственное отличие состоит в том, что код в листинге 10.3 строит дерево запроса с использованием планов, преобразующее окончательный план в образ сканирования.

Листинг 10.3. Использование планов для выполнения запроса

```
SimpleDB db = new SimpleDB("studentdb");
MetadataMgr mdm = db.mdMgr();
Transaction tx = db.newTx();
Plan p1 = new TablePlan(tx, "student", mdm);
Plan p2 = new TablePlan(tx, "dept", mdm);
Predicate pred1 = new Predicate(...); //dname='math'
Plan p3 = new SelectPlan(p2, pred1);
Plan p4 = new ProductPlan(p1, p3);
Predicate pred2 = new Predicate(...); //majorid=did
Plan p5 = new SelectPlan(p4, pred2);
List<String> fields = Arrays.asList("sname");
Plan p6 = new ProjectPlan(p5, fields);
Scan s = p6.open();
```

В листингах 10.4, 10.5, 10.6 и 10.7 представлены определения классов `TablePlan`, `SelectPlan`, `ProjectPlan` и `ProductPlan`. Класс `TablePlan` получает свои оценки стоимости, обращаясь непосредственно к диспетчеру метаданных. Другие классы используют формулы из предыдущего раздела.

Листинг 10.4. Определение класса TablePlan в SimpleDB

```
public class TablePlan implements Plan {
    private Transaction tx;
    private String tblname;
    private Layout layout;
    private StatInfo si;

    public TablePlan(Transaction tx, String tblname, MetadataMgr md) {
        this.tx = tx;
        this.tblname = tblname;
        layout = md.getLayout(tblname, tx);
        si = md.getStatInfo(tblname, layout, tx);
    }

    public Scan open() {
        return new TableScan(tx, tblname, layout);
    }

    public int blocksAccessed() {
        return si.blocksAccessed();
    }

    public int recordsOutput() {
        return si.recordsOutput();
    }

    public int distinctValues(String fldname) {
        return si.distinctValues(fldname);
    }

    public Schema schema() {
        return layout.schema();
    }
}
```

Оценка стоимости плана для оператора select имеет самую сложную реализацию, потому что зависит от предиката. Для нее предикат предоставляет методы `reductionFactor` и `equatesWithConstant`. Метод `reductionFactor` используется методом `recordsOutput` для вычисления степени уменьшения размера входной таблицы предикатом. Метод `equatesWithConstant` используется методом `distinctValues`, чтобы определить, сравнивает ли предикат указанное поле с константой.

Листинг 10.5. Определение класса SelectPlan в SimpleDB

```
public class SelectPlan implements Plan {
    private Plan p;
    private Predicate pred;

    public SelectPlan(Plan p, Predicate pred) {
        this.p = p;
        this.pred = pred;
    }

    public Scan open() {
        Scan s = p.open();
        return new SelectScan(s, pred);
    }
```

```
    public int blocksAccessed() {
        return p.blocksAccessed();
    }

    public int recordsOutput() {
        return p.recordsOutput() / pred.reductionFactor(p);
    }

    public int distinctValues(String fldname) {
        if (pred.equatesWithConstant(fldname) != null)
            return 1;
        else {
            String fldname2 = pred.equatesWithField(fldname);
            if (fldname2 != null)
                return Math.min(p.distinctValues(fldname),
                                p.distinctValues(fldname2));
            else
                return p.distinctValues(fldname);
        }
    }

    public Schema schema() {
        return p.schema();
    }
}
```

Конструкторы `ProjectPlan` и `ProductPlan` создают свои схемы на основе схем своих базовых планов. В `ProjectPlan` схема создается путем поиска каждого поля в базовом списке и добавления этой информации в новую схему. В `ProductPlan` схема создается путем объединения базовых схем.

Листинг 10.6. Определение класса ProjectPlan в SimpleDB

```
public class ProjectPlan implements Plan {
    private Plan p;
    private Schema schema = new Schema();

    public ProjectPlan(Plan p, List<String> fieldlist) {
        this.p = p;
        for (String fldname : fieldlist)
            schema.add(fldname, p.schema());
    }

    public Scan open() {
        Scan s = p.open();
        return new ProjectScan(s, schema.fields());
    }

    public int blocksAccessed() {
        return p.blocksAccessed();
    }

    public int recordsOutput() {
        return p.recordsOutput();
    }
```

```
    public int distinctValues(String fldname) {
        return p.distinctValues(fldname);
    }

    public Schema schema() {
        return schema;
    }
}
```

Листинг 10.7. Определение класса ProductPlan в SimpleDB

```
public class ProductPlan implements Plan {
    private Plan p1, p2;
    private Schema schema = new Schema();

    public ProductPlan(Plan p1, Plan p2) {
        this.p1 = p1;
        this.p2 = p2;
        schema.addAll(p1.schema());
        schema.addAll(p2.schema());
    }

    public Scan open() {
        Scan s1 = p1.open();
        Scan s2 = p2.open();
        return new ProductScan(s1, s2);
    }

    public int blocksAccessed() {
        return p1.blocksAccessed()
            + (p1.recordsOutput() * p2.blocksAccessed());
    }

    public int recordsOutput() {
        return p1.recordsOutput() * p2.recordsOutput();
    }

    public int distinctValues(String fldname) {
        if (p1.schema().hasField(fldname))
            return p1.distinctValues(fldname);
        else
            return p2.distinctValues(fldname);
    }

    public Schema schema() {
        return schema;
    }
}
```

Методы `open` во всех этих классах планов имеют простую реализацию. Процесс конструирования образа сканирования на основе плана выполняется в общем случае в два этапа. Сначала метод `open` рекурсивно создает образы для всех базовых планов. А затем передает полученные образы конструктору класса `Scan` для данного оператора.

10.4. Планирование запроса

Как уже рассказывалось, синтаксический анализатор получает строку с оператором SQL и возвращает объект `QueryData`. В этом разделе рассматривается задача построения плана из этого объекта `QueryData`.

10.4.1. Алгоритм планирования запросов в SimpleDB

Движок SimpleDB поддерживает ограниченное подмножество SQL и не предусматривает возможность вычислений, сортировки, группировки, вложенности и переименования. Благодаря этому все SQL-запросы можно реализовать с помощью дерева запросов, в котором используются только три оператора: select, project и product. Последовательность этапов создания такого плана приводится в алгоритме 10.1.

Алгоритм 10.1. Простой алгоритм создания плана запроса с учетом поддержки ограниченного подмножества SQL в SimpleDB

1. Сконструировать план для каждой таблицы T в предложении from.
   a) Если T – хранимая таблица, то планом является план для таблицы T.
   b) Если T – представление, то планом является результат рекурсивного вызова этого алгоритма для определения T.
2. Получить прямое произведение (product) планов таблиц в указанном порядке.
3. Получить план для оператора селекции (select) по предикату в предложении `where`.
4. Получить план для оператора проекции (project) с учетом полей в предложении `select`.

Рассмотрим работу этого алгоритма планирования на примере SQL-запроса, возвращающего имена студентов, которые получили оценку «A» у профессора Эйнштейна. Запрос показан в листинге 10.8, а на рис. 10.2 изображено дерево этого запроса, созданное алгоритмом.

Листинг 10.8. Запрос SQL, возвращающий имена студентов, которые получили оценку «A» у профессора Эйнштейна

```
select SName
from STUDENT, ENROLL, SECTION
where SId = StudentId
and SectionId = SectId
and Grade = 'A'
and Prof = 'einstein'
```

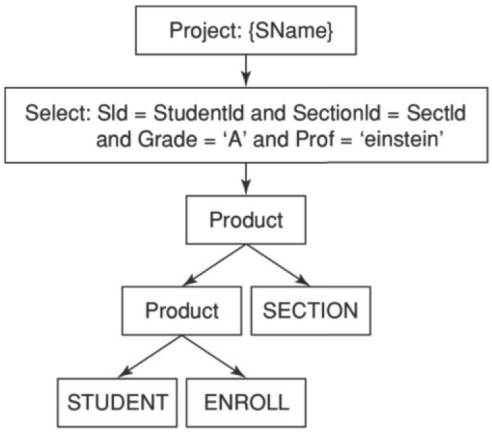


Рис. 10.2 . Результат применения алгоритма планирования к SQL-запросу

На рис. 10.3 изображен результат применения алгоритма планирования к эквивалентному запросу, который использует представление. В листинге 10.9 показаны определение представления и сам запрос, а на рис. 10.3 изображено дерево запроса для представления (a) и дерево для всего запроса (b).

Листинг 10.9. Запрос SQL, использующий представление

```
create view EINSTEIN as
select SectId from SECTION where Prof = 'einstein'

select SName
from STUDENT, ENROLL, EINSTEIN
where SId = StudentId and SectionId = SectId and Grade = 'A'
```

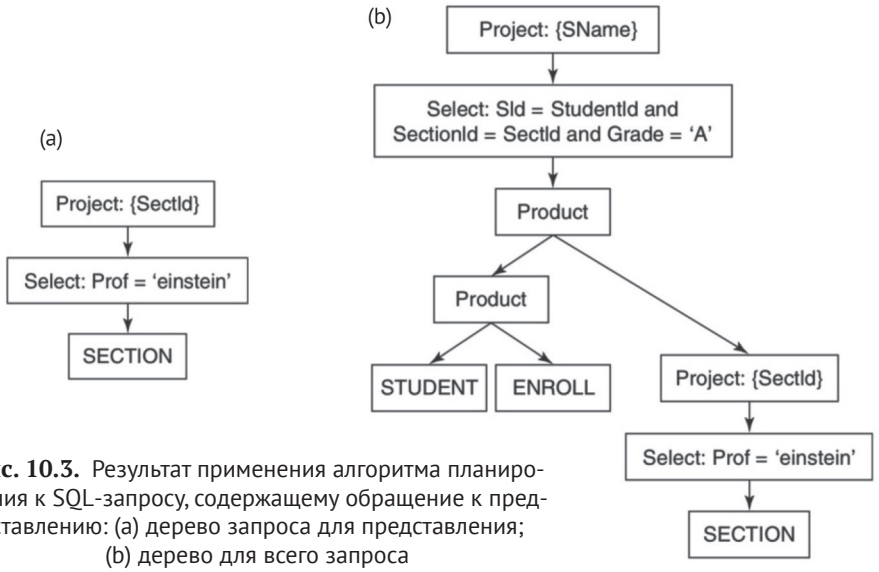


Рис. 10.3. Результат применения алгоритма планирования к SQL-запросу, содержащему обращение к представлению: (a) дерево запроса для представления; (b) дерево для всего запроса

Обратите внимание, что окончательное дерево состоит из произведения двух таблиц и дерева представления, за которым следуют операции селекции и проекции. Дерево на рис. 10.3c эквивалентно дереву на рис. 10.2b, но имеет некоторые отличия. В частности, предикат селекции был «сдвинут» вниз по дереву и добавилась промежуточная проекция. В главе 15 будут представлены методы оптимизации запросов, использующие такие эквивалентности.

10.4.2. Реализация алгоритма планирования запросов

Базовый алгоритм планирования запросов в движке SimpleDB реализует класс `BasicQueryPlanner`; его определение показано в листинге 10.10. Каждый из четырех шагов в коде реализует соответствующий шаг в этом алгоритме.

Листинг 10.10. Определение класса BasicQueryPlanner в SimpleDB

```
public class BasicQueryPlanner implements QueryPlanner {
    private MetadataMgr mdm;

    public BasicQueryPlanner(MetadataMgr mdm) {
        this.mdm = mdm;
    }

    public Plan createPlan(QueryData data, Transaction tx) {
        // Шаг 1: создать планы для всех таблиц и представлений, упомянутых в запросе.
        List<Plan> plans = new ArrayList<Plan>();
        for (String tblname : data.tables()) {
            String viewdef = mdm.getViewDef(tblname, tx);
            if (viewdef != null) { // Рекурсивно создать план для представления.
                Parser parser = new Parser(viewdef);
                QueryData viewdata = parser.query();
                plans.add(createPlan(viewdata, tx));
            }
            else
                plans.add(new TablePlan(tblname, tx, mdm));
        }

        // Шаг 2: получить прямое произведение всех планов таблиц
        Plan p = plans.remove(0);
        for (Plan nextplan : plans)
            p = new ProductPlan(p, nextplan);

        // Шаг 3: добавить план для селекции по предикату
        p = new SelectPlan(p, data.pred());

        // Шаг 4: выполнить проекцию с учетом имен полей
        return new ProjectPlan(p, data.fields());
    }
}
```

Базовый алгоритм планирования запросов является негибким и наивным. Он генерирует планы операторов прямого произведения в порядке, определяемом методом `QueryData.tables`. Обратите внимание, что порядок может быть произвольным: любой другой порядок следования таблиц будет производить эквивалентный образ сканирования. По этой причине алгоритм

показывает неустойчивую (и часто плохую) производительность, так как не использует метаданные плана для выбора оптимального порядка прямых произведений планов.

В листинге 10.11 показана немного усовершенствованная версия алгоритма планирования. Она по-прежнему рассматривает таблицы в одном и том же порядке, но для каждой таблицы создает два плана: один для случая, когда в прямом произведении таблица находится слева, а другой – справа, и сохраняет план с наименьшей стоимостью.

Листинг 10.11. Определение класса BetterQueryPlanner в SimpleDB

```
public class BetterQueryPlanner implements QueryPlanner {
    ...
    public Plan createPlan(QueryData data, Transaction tx) {
        ...
        // Шаг 2: получить прямое произведение всех планов таблиц
        // Каждый раз на этом шаге выбирается план с наименьшей стоимостью
        Plan p = plans.remove(0);
        for (Plan nextplan : plans) {
            Plan p1 = new ProductPlan(nextplan, p);
            Plan p2 = new ProductPlan(p, nextplan);
            p = (p1.blocksAccessed() < p2.blocksAccessed() ? p1 : p2;
        }
        ...
    }
}
```

Этот алгоритм планирования действует лучше базового, но все еще слишком сильно зависит от порядка следования таблиц в запросе. Алгоритмы планирования в коммерческих системах баз данных намного сложнее. Они не только анализируют стоимость эквивалентных планов, но также используют дополнительные реляционные операции в особых обстоятельствах. Их цель – выбрать наиболее эффективный план (и, как следствие, получить конкурентные преимущества). Эти приемы обсуждаются в главах 12, 13, 14 и 15.

10.5. Планирование операций изменения

В этом разделе рассказывается, как планировщик должен обрабатывать операторы изменения. Планировщик изменений в SimpleDB реализует класс `BasicUpdatePlanner`; его определение показано в листинге 10.12. Этот класс имеет отдельные методы для каждой операции изменения. Они обсуждаются в следующих подразделах.

Листинг 10.12. Определение класса BasicUpdatePlanner в SimpleDB

```
public class BasicUpdatePlanner implements UpdatePlanner {
    private MetadataMgr mdm;

    public BasicUpdatePlanner(MetadataMgr mdm) {
        this.mdm = mdm;
    }
```

```
    public int executeDelete(DeleteData data, Transaction tx) {
        Plan p = new TablePlan(data.tableName(), tx, mdm);
        p = new SelectPlan(p, data.pred());
        UpdateScan us = (UpdateScan) p.open();
        int count = 0;
        while(us.next()) {
            us.delete();
            count++;
        }
        us.close();
        return count;
    }

    public int executeModify(ModifyData data, Transaction tx) {
        Plan p = new TablePlan(data.tableName(), tx, mdm);
        p = new SelectPlan(p, data.pred());
        UpdateScan us = (UpdateScan) p.open();
        int count = 0;
        while(us.next()) {
            Constant val = data.newValue().evaluate(us);
            us.setVal(data.targetField(), val);
            count++;
        }
        us.close();
        return count;
    }

    public int executeInsert(InsertData data, Transaction tx) {
        Plan p = new TablePlan(data.tableName(), tx, mdm);
        UpdateScan us = (UpdateScan) p.open();
        us.insert();
        Iterator<Constant> iter = data.vals().iterator();
        for (String fldname : data.fields()) {
            Constant val = iter.next();
            us.setVal(fldname, val);
        }
        us.close();
        return 1;
    }

    public int executeCreateTable(CreateTableData data, Transaction tx) {
        mdm.createTable(data.tableName(), data.newSchema(), tx);
        return 0;
    }

    public int executeCreateView(CreateViewData data, Transaction tx) {
        mdm.createView(data.viewName(), data.viewDef(), tx);
        return 0;
    }

    public int executeCreateIndex(CreateIndexData data, Transaction tx) {
        mdm.createIndex(data.indexName(), data.tableName(),
                        data.fieldName(), tx);
        return 0;
    }
}
```

10.5.1. Планирование удаления и обновления

Образ сканирования для оператора delete (и update) – это образ для оператора select, который извлекает записи, подлежащие удалению (или изменению). Например, рассмотрим следующий оператор update:

```
update STUDENT
set MajorId = 20
where MajorId = 30 and GradYear = 2020
```

и оператор delete:

```
delete from STUDENT
where MajorId = 30 and GradYear = 2020
```

Эти операторы имеют одинаковые образы, включающие всех студентов, обучавшихся на кафедре 30 и выпустившихся в 2020 году. Методы `executeDelete` и `executeModify` создают этот образ и выполняют обход записей в нем, производя соответствующее действие с каждой из них: оператор update изменяет, а оператор delete удаляет.

Заглянув в код, можно заметить, что оба метода создают один и тот же план, похожий на план, созданный планировщиком запросов (кроме того что планировщик запросов добавляет план проекции). Также оба метода одинаково открывают образ сканирования и выполняют обход записей. Метод `executeDelete` вызывает `delete` для каждой записи в образе, тогда как `executeModify` вызывает `setVal` для изменяемого поля в каждой записи в образе. Оба метода также подсчитывают число затронутых записей и возвращают его вызывающему коду.

10.5.2. Планирование вставки

Образ сканирования для оператора insert – это простой образ базовой таблицы. Метод `executeInsert` сначала вставляет новую запись в этот образ, а затем выполняет параллельный обход списков `fields` и `vals`, вызывая `setInt` или `setString` для установки значений в указанные поля записи. Метод возвращает число 1, сообщая, что была вставлена одна запись.

10.5.3. Планирование создания таблиц, представлений и индексов

Методы `executeCreateTable`, `executeCreateView` и `executeCreateIndex` отличаются от других, поскольку им не нужен доступ ни к каким другим записям с данными и, следовательно, не нужны образы сканирований. Они просто вызывают методы диспетчера метаданных `createTable`, `createView` и `createIndex`, используя информацию, полученную от синтаксического анализатора. Все эти методы возвращают 0, чтобы указать, что никакие записи не были затронуты.

10.6. Планировщик в SimpleDB

Планировщик – это компонент движка базы данных, преобразующий оператор SQL в план. Планировщик в SimpleDB реализуется классом `Planner`, API которого представлен в листинге 10.13.

Листинг 10.13. API планировщика в SimpleDB

Planner

```
public Plan createQueryPlan(String query, Transaction tx);
public int  executeUpdate(String cmd, Transaction tx);
```

В первом аргументе оба метода принимают строковое представление оператора SQL. Метод `createQueryPlan` создает и возвращает план для входного запроса. Метод `executeUpdate` создает план для входного оператора, выполняет его и возвращает количество затронутых записей (подобно методу `executeUpdate` в JDBC).

Клиент может получить объект `Planner`, вызвав статический метод `planner` класса `SimpleDB`. В листинге 10.14 приводится определение класса `PlannerTest`, который иллюстрирует использование планировщика. Часть 1 в этом листинге демонстрирует обработку запроса SQL. Здесь вызывается метод `createQueryPlan` планировщика со строкой запроса, который возвращает план. Затем код вызывает метод `open` плана, получает образ сканирования и выполняет обход записей в нем. Вторая часть кода иллюстрирует использование SQL-оператора update. Строка с оператором передается в метод `executeUpdate` планировщика, который выполняет всю необходимую работу.

Листинг 10.14. Класс PlannerTest

```
public class PlannerTest {
    public static void main(String[] args) {
        SimpleDB db = new SimpleDB("studentdb");
        Planner planner = db.planner();
        Transaction tx  = db.newTx();

        // часть 1: обработка запроса
        String qry = "select sname, gradyear from student";
        Plan p = planner.createQueryPlan(qry, tx);
        Scan s = p.open();
        while (s.next())
            System.out.println(s.getString("sname") + " " +
                    s.getInt("gradyear"));
        s.close();

        // часть 2: обработка команды изменения
        String cmd = "delete from STUDENT where MajorId = 30";
        int num = planner.executeUpdate(cmd, tx);
        System.out.println(num + " students were deleted");

        tx.commit();
    }
}
```

Планировщик в SimpleDB имеет два метода: один для обработки запросов и один для обработки операторов изменения. Оба метода обрабатывают данные схожим образом. В алгоритме 10.2 перечислены выполняемые ими шаги. В частности, оба метода выполняют шаги 1–3. Основное отличие между ними заключается в том, как они используют созданные ими планы. Метод `createQueryPlan` просто возвращает свой план, тогда как `executeUpdate` открывает и выполняет план.

Алгоритм 10.2. Шаги, выполняемые двумя методами планировщика

1. *Разбор оператора SQL*. Метод вызывает синтаксический анализатор и передает ему входную строку. В ответ анализатор возвращает объект, содержащий данные из оператора SQL. Например, объект `QueryData` для запроса, объект `InsertData` для оператора вставки и т. д.
2. *Проверка оператора SQL*. Метод исследует объект `QueryData` (или `InsertData` и т. д.) и определяет его семантическую осмысленность.
3. *Создание плана для оператора SQL*. Метод использует алгоритм планирования, чтобы получить дерево запроса, соответствующее оператору, и создает план, соответствующий этому дереву.
4. *Обработка плана*.
    a) Метод `createQueryPlan` просто возвращает план.
    b) Метод `executeUpdate` выполняет план. Этот метод открывает план и получает образ сканирования; затем выполняет его обход, изменяя каждую запись образа, как определено оператором, и возвращает количество затронутых записей.

В листинге 10.15 приводится определение класса `Planner` в SimpleDB. Его методы являются прямой реализацией алгоритма 10.2. Метод `createQueryPlan` создает экземпляр синтаксического анализатора для разбора своего SQL-запроса, вызывает его метод `query`, проверяет полученный в ответ объект `QueryData` (по крайней мере, он должен это сделать) и возвращает план, созданный планировщиком запросов. Метод `executeUpdate` действует аналогично: разбирает строку SQL-оператора, проверяет объект, возвращаемый синтаксическим анализатором, и вызывает соответствующий метод планировщика для выполнения изменений.

Объект, возвращаемый синтаксическим анализатором операторов изменения, имеет тип `InsertData`, `DeleteData` и т. д., в зависимости от типа оператора. Метод `executeUpdate` проверяет этот тип, чтобы определить, какой метод планировщика вызвать.

Листинг 10.15. Определение класса Planner в SimpleDB

```
public class Planner {
    private QueryPlanner  qplanner;
    private UpdatePlanner uplanner;

    public Planner(QueryPlanner qplanner, UpdatePlanner uplanner) {
        this.qplanner = qplanner;
        this.uplanner = uplanner;
    }

    public Plan createQueryPlan(String cmd, Transaction tx) {
        Parser parser = new Parser(cmd);
        QueryData data = parser.query();
        // здесь должен быть код, проверяющий семантику запроса...
        return qplanner.createPlan(data, tx);
    }
```

```
    public int executeUpdate(String cmd, Transaction tx) {
        Parser parser = new Parser(cmd);
        Object obj = parser.updateCmd();
        // здесь должен быть код, проверяющий семантику оператора изменения...
        if (obj instanceof InsertData)
            return uplanner.executeInsert((InsertData)obj, tx);
        else if (obj instanceof DeleteData)
            return uplanner.executeDelete((DeleteData)obj, tx);
        else if (obj instanceof ModifyData)
            return uplanner.executeModify((ModifyData)obj, tx);
        else if (obj instanceof CreateTableData)
            return uplanner.executeCreateTable((CreateTableData)obj, tx);
        else if (obj instanceof CreateViewData)
            return uplanner.executeCreateView((CreateViewData)obj, tx);
        else if (obj instanceof CreateIndexData)
            return uplanner.executeCreateIndex((CreateIndexData)obj, tx);
        else
            return 0; // эта ветка никогда не должна выполняться
    }
}
```

Для фактического планирования объект `Planner` использует планировщика запросов и планировщика изменений. Эти объекты передаются в конструктор `Planner`, что позволяет настроить планирование с использованием разных алгоритмов. Например, в главе 15 демонстрируется хитроумный планировщик запросов под названием `HeuristicQueryPlanner`; этот планировщик можно использовать вместо `BasicQueryPlanner`, просто передав объект `HeuristicQueryPlanner` в конструктор `Planner`.

Для того чтобы можно было легко менять планировщики, используются интерфейсы Java. Аргументы конструктора `Planner` имеют типы интерфейсов `QueryPlanner` и `UpdatePlanner`, определение которых показано в листинге 10.16. Классы `BasicQueryPlanner` и `BasicUpdatePlanner`, как и другие, более сложные планировщики запросов и изменений в главе 15, реализуют эти интерфейсы.

Листинг 10.16. Определение интерфейсов QueryPlanner и UpdatePlanner в SimpleDB

```
public interface QueryPlanner {
    public Plan createPlan(QueryData data, Transaction tx);
}

public interface UpdatePlanner {
    public int executeInsert(InsertData data, Transaction tx);
    public int executeDelete(DeleteData data, Transaction tx);
    public int executeModify(ModifyData data, Transaction tx);
    public int executeCreateTable(CreateTableData data, Transaction tx);
    public int executeCreateView(CreateViewData data, Transaction tx);
    public int executeCreateIndex(CreateIndexData data, Transaction tx);
}
```

Объекты `Planner` создаются конструктором класса `SimpleDB`, который создает экземпляры базовых планировщиков запросов и изменений и передает их конструктору `Planner`, как показано в листинге 10.17. Чтобы настроить механизм на использование другого планировщика запросов, достаточно

изменить конструктор SimpleDB, чтобы он создавал необходимые объекты QueryPlanner и UpdatePlanner.

Листинг 10.17. Код создания планировщика в SimpleDB

```
public SimpleDB(String dirname) {
    ...
    mdm = new MetadataMgr(isnew, tx);
    QueryPlanner qp = new BasicQueryPlanner(mdm);
    UpdatePlanner up = new BasicUpdatePlanner(mdm);
    planner = new Planner(qp, up);
    ...
}
```

10.7. Итоги

- Чтобы для данного запроса получить образ сканирования с минимальной стоимостью, система баз данных должна оценить количество обращений к дисковым блокам при сканировании. Для образа s определены следующие функции оценки стоимости:
  - B(s) – количество блоков, к которым требуется обратиться, чтобы получить выходной образ s;
  - R(s) – количество записей в выходном образе s;
  - V(s, F) – количество уникальных значений F в выходном образе s.
- Если s – это образ сканирования таблицы, то эти функции эквивалентны статистическим метаданным этой таблицы. В других случаях для каждого оператора существуют свои формулы вычисления этих функций, основанные на значениях функций входных образов.
- Оператор SQL может иметь несколько эквивалентных деревьев запросов, каждому из которых соответствует свой образ сканирования. В таких случаях планировщик базы данных должен выбрать образ с наименьшей оценочной стоимостью. Для этого ему может потребоваться построить несколько деревьев запросов, сравнить их стоимость и выбрать дерево с самой низкой стоимостью.
- Дерево запроса, построенное для сравнения стоимостей, называется *планом*. Планы и образы сканирований концептуально схожи в том, что оба обозначают дерево запросов. Разница в том, что план имеет методы оценки затрат и обращается к метаданным, а не к фактическим данным. Чтобы создать план, не требуется обращаться к диску. Планировщик создает несколько планов, сравнивает их, выбирает план с наименьшей стоимостью и открывает его.
- Планировщик – это компонент движка базы данных, который преобразует оператор SQL в план.
- Кроме того, планировщик *удостоверяется* в семантической осмысленности оператора, проверяя:
  - наличие в каталоге упоминаемых таблиц и полей;
  - однозначность интерпретации имен полей;
  - соответствие действий с полями их типам;
  - соответствие размеров и типов всех констант их полям.

- *Базовый алгоритм планирования запросов* создает элементарный план следующим образом:
  1. Сконструировать план для каждой таблицы T в предложении `from`.
     a) Если T – это хранимая таблица, то планом является план для таблицы T.
     b) Если T – это представление, то планом является результат рекурсивного вызова этого алгоритма для определения T.
  2. Получить прямое произведение (product) планов таблиц из предложения `from` в указанном порядке.
  3. Получить план для оператора селекции (select) по предикату в предложении `where`.
  4. Получить план для оператора проекции (project) с учетом полей в предложении `select`.
- Базовый алгоритм планирования запросов генерирует наивный и часто неэффективный план. Алгоритмы планирования в коммерческих системах баз данных выполняют всесторонний анализ имеющихся эквивалентных планов, как описывается в главе 15.
- Операторы удаления и изменения обрабатываются аналогично. Планировщик создает план оператора select для извлечения записей, которые необходимо удалить (или изменить). Методы `executeDelete` и `executeModify` открывают план и выполняют обход полученного образа сканирования, производя соответствующее действие с каждой записью в нем: в случае оператора изменения производится изменение каждой записи; в случае оператора удаления – удаление.
- План оператора insert – это план базовой таблицы. Метод `executeInsert` открывает план и вставляет новую запись в полученный образ.
- Для операторов создания планы не конструируются, потому что они не обращаются ни к каким данным. Вместо этого вызываются соответствующие методы диспетчера метаданных.

10.8. Для дополнительного чтения

Планировщик, представленный в этой главе, понимает только небольшое подмножество SQL, и я лишь кратко затронул вопросы планирования более сложных конструкций. Статья (Kim, 1982) описывает проблемы, свойственные вложенным запросам, и предлагает некоторые решения. В статье (Chaudhuri, 1998) обсуждаются стратегии реализации более сложных аспектов SQL, включая внешние соединения и вложенные запросы.

Chaudhuri, S. (1998). «An overview of query optimization in relational systems». *In Proceedings of the ACM Principles of Database Systems Conference* (p. 34–43).

Kim, W. (1982). «On optimizing an SQL-like nested query». *ACM Transactions on Database Systems*, 7 (3), 443–469.

10.9. Упражнения

Теория

10.1. Взгляните на следующий запрос реляционной алгебры:

```
T1 = select(DEPT, DName='math')
T2 = select(STUDENT, GradYear=2018)
product(T1, T2)
```

Основываясь на предположениях, изложенных в разделе 10.2:

a) подсчитайте, сколько обращений к диску потребуется, чтобы выполнить операцию;

b) подсчитайте, сколько обращений к диску потребуется, чтобы выполнить операцию, если аргументы в операторе *product* поменять местами.

10.2. Вычислите оценки B(s), R(s) и V(s, F) для запросов на рис. 10.2 и 10.3.

10.3. Покажите, что если поменять местами аргументы в операторе product в разделе 10.2.5, то для выполнения всей операции потребуется 4502 обращения к блокам.

10.4. В разделе 10.2.4 говорилось, что прямое произведение таблиц STUDENT и DEPT можно выполнить более эффективно, если во внешнем цикле выполнять обход записей в STUDENT. Используя статистики в табл. 7.2, подсчитайте, сколько обращений к блокам потребуется, чтобы получить результат.

10.5. Для каждого из следующих операторов SQL нарисуйте план, который будет сгенерирован базовым планировщиком из этой главы.

```
(a) select SName, Grade
    from STUDENT, COURSE, ENROLL, SECTION
(b) where SId ¼ StudentId and SectId ¼ SectionId
    and CourseId ¼ CId and Title ¼ 'Calculus'
    select SName
    from STUDENT, ENROLL
    where MajorId ¼ 10 and SId ¼ StudentId and Grade ¼ 'C'
```

10.6. Для каждого запроса в упражнении 10.5 объясните, что планировщик должен проверить, чтобы убедиться в его правильности.

10.7. Для каждого из следующих операторов изменения объясните, что планировщик должен проверить, чтобы убедиться в его правильности.

```
(a) insert into STUDENT(SId, SName, GradYear, MajorId)
    values(120, 'abigail', 2012, 30)
(b) delete from STUDENT
    where MajorId = 10 and SID in (select StudentId
                                   from ENROLL
                                   where Grade = 'F')
(c) update STUDENT
    set GradYear ¼ GradYear + 3
    where MajorId in (select DId from DEPT
                  where DName = 'drama')
```

Практика

10.8. Планировщик в SimpleDB не проверяет значимость имен таблиц.

a) Какая проблема возникнет, если в запросе упомянуть несуществующую таблицу?

b) Исправьте класс `Planner`, чтобы он проверял имена таблиц и генерировал исключение `BadSyntaxException`, если таблица не существует.

10.9. Планировщик в SimpleDB не проверяет существование и уникальность имен полей.

a) Какая проблема возникнет, если в запросе упомянуть несуществующее имя поля?

b) Какая проблема возникнет, если в запросе упомянуть таблицы, имеющие поля с одинаковыми именами?

c) Исправьте код, чтобы он выполнял соответствующие проверки.

10.10. Планировщик в SimpleDB не проверяет типы операндов в предикатах.

a) Какая проблема возникнет, если в предикате выполнить операцию, не соответствующую типам операндов?

b) Исправьте код, чтобы он выполнял соответствующие проверки.

10.11. Планировщик изменений в SimpleDB не проверяет соответствие типов и размеров строковых констант указанным полям в операторе insert, а также не проверяет совпадение размеров списков констант и полей. Исправьте код соответствующим образом.

10.12. Планировщик изменений SimpleDB не проверяет соответствие типа значения, присваиваемого указанному полю в операторе update. Исправьте код соответствующим образом.

10.13. В упражнении 9.11 предлагалось изменить синтаксический анализатор и добавить поддержку переменных области значений, а в упражнении 8.14 – реализовать класс `RenameScan` для оператора переименования. Переменные области значений можно реализовать посредством переименования – сначала планировщик переименовывает каждое поле таблицы, добавляя префикс с именем переменной области значений; затем добавляет операторы прямого произведения (product), селекции (select) и проекции (project); и, наконец, переименовывает поля обратно, присваивая им имена без префиксов.

a) Напишите класс `RenamePlan`.

b) Добавьте в базовый планировщик поддержку переименования.

c) Напишите программу JDBC для тестирования ваших изменений. В частности, программа должна выполнять соединение таблицы с самой собой, например отыскивать студентов, обучающихся на той же специальности, что и Joe.

10.14. В упражнении 9.12 предлагалось изменить синтаксический анализатор и добавить поддержку ключевого слова `AS` в предложение `select`, а в упражнении 8.15 – реализовать класс `ExtendScan` для оператора расширения.

a) Напишите класс `ExtendPlan`.
b) Измените реализацию базового планировщика, чтобы он добавлял объекты `ExtendPlan` в план запроса. Они должны следовать за планами операторов прямого произведения, но перед планом оператора проекции.
c) Напишите программу JDBC для тестирования ваших изменений.

10.15. В упражнении 9.13 предлагалось изменить синтаксический анализатор и добавить поддержку ключевого слова `UNION`, а в упражнении 8.16 – реализовать класс `UnionScan` для оператора объединения.

a) Напишите класс `UnionPlan`.
b) Измените реализацию базового планировщика, чтобы он добавлял объекты `UnionPlan` в план запроса. Они должны следовать за планом оператора проекции.
c) Напишите программу JDBC для тестирования ваших изменений.

10.16. В упражнении 9.14 предлагалось изменить синтаксический анализатор и добавить поддержку вложенных запросов, а в упражнении 8.17 – реализовать классы `SemijoinScan` и `AntijoinScan` для операторов полу- и антисоединения.

a) Напишите классы `SemijoinPlan` и `AntijoinPlan`.
b) Измените реализацию базового планировщика, чтобы он добавлял объекты этих классов в план запроса. Они должны следовать за планами операторов прямого произведения, но перед планами операторов расширения.
c) Напишите программу JDBC для тестирования ваших изменений.

10.17. В упражнении 9.15 предлагалось изменить синтаксический анализатор и добавить поддержку символа «*» в предложении `select`.

a) Внесите необходимые изменения в планировщик.
b) Напишите программу JDBC для тестирования ваших изменений.

10.18. В упражнении 9.16 предлагалось изменить синтаксический анализатор в SimpleDB и добавить поддержку нового варианта оператора insert.

a) Внесите необходимые изменения в планировщик.
b) Напишите программу JDBC для тестирования ваших изменений.

10.19. Базовый планировщик изменений вставляет новые записи в начало таблицы.

a) Разработайте и реализуйте модифицированный вариант оператора insert, вставляющий записи в конец таблицы или, может быть, после предыдущей вставки.
b) Сравните преимущества двух стратегий. Какая предпочтительнее, на ваш взгляд?

10.20. Базовый планировщик изменений в SimpleDB предполагает, что таблица, упомянутая в операторе update, является хранимой таблицей. Стандартный SQL также позволяет использовать имя представления, при условии что представление является обновляемым.

a) Добавьте в планировщик изменений возможность обновления представлений. Планировщик не должен проверять недоступность представления для обновлений. Он может просто попытаться произвести обновление и сгенерировать исключение, если что-то пойдет не так. Обратите внимание, что вам потребуется изменить класс `ProjectScan`, реализовав в нем интерфейс `UpdateScan`, как в упражнении 8.12.
b) Объясните, как планировщик может проверить доступность представления для обновлений.

10.21. Базовый планировщик изменений в SimpleDB обрабатывает представления, выполняя разбор строки запроса и сохраняя ее в каталоге. Впоследствии базовый планировщик запросов должен анализировать определение представления каждый раз, когда оно используется в запросе. Альтернативное решение: сохранять проанализированную версию запроса в каталоге и использовать ее в планировщике запросов.

a) Реализуйте эту стратегию. (*Подсказка:* используйте механизм сериализации объектов в Java. Сериализуйте объект `QueryData` и используйте `StringWriter` для преобразования объекта в строку. В этом случае метод `getViewDef` диспетчера метаданных сможет восстанавливать объект `QueryData` из сохраненной строки.)
b) Как эта реализация соотносится с подходом, принятым в SimpleDB?

10.22. Измените сервер SimpleDB так, чтобы каждый раз, выполняя запрос, он выводил в консоль текст запроса и соответствующий ему план; эта информация поможет понять, как сервер обрабатывает запросы. Для этого необходимо:

a) изменить все классы, реализующие интерфейс `Plan`, добавив в них метод `toString`. Этот метод должен возвращать отформатированное строковое представление плана, напоминающее запрос реляционной алгебры;
b) изменить метод `executeQuery` (в классах `simpledb.jdbc.network.RemoteStatementImpl` и `simpledb.jdbc.embedded.EmbeddedStatement`), чтобы он выводил в консоль исходный запрос и строку, сгенерированную методом `toString`, как описано в п. «a»).

Глава 11

Интерфейсы JDBC

В этой главе рассказывается, как создавать интерфейсы JDBC для движка базы данных. Создать встроенный интерфейс относительно просто – достаточно написать классы JDBC, используя соответствующие классы движка. Для создания серверного интерфейса тоже необходимо написать дополнительный код, реализующий обработку запросов JDBC на сервере. Здесь показано, как использование Java RMI может упростить разработку этого дополнительного кода.

11.1. SimpleDB API

В главе 2 был представлен JDBC – стандартный интерфейс подключения к движкам баз данных – и продемонстрировано несколько примеров JDBC-клиентов. Однако последующие главы не использовали JDBC – в них были показаны тестовые программы, иллюстрирующие различные возможности движка SimpleDB. Тем не менее эти тестовые программы тоже являются клиентами баз данных, просто для доступа к движку SimpleDB они используют SimpleDB API вместо JDBC API.

SimpleDB API состоит из общедоступных классов SimpleDB (таких как `SimpleDB`, `Transaction`, `BufferMgr`, `Scan` и т. д.) и их общедоступных методов. Этот API гораздо обширнее, чем JDBC, и позволяет обращаться к низкоуровневым механизмам движка. Такой низкоуровневый доступ дает прикладным программам возможность настраивать функциональность, предлагаемую движком. Например, тестовый код в главе 4 напрямую обращается к диспетчерам журнала и буферов в обход диспетчера транзакций.

Однако низкоуровневый доступ влечет за собой определенные сложности. Автор приложения должен глубоко знать API целевого движка, и, чтобы перенастроить приложение на использование другого движка (или даже просто использовать серверную версию того же движка), ему придется переписать свою программу и использовать в ней другой API. Цель JDBC – предоставить стандартный API, одинаковый для любых движков баз данных, за исключением незначительных различий в настройках.

Для реализации JDBC API в SimpleDB достаточно отметить соответствия между двумя API. Например, рассмотрим листинг 11.1. В части (а) находится приложение JDBC, которое обращается к базе данных, выводит полученный набор результатов и закрывает соединение. В части (б) находится эквивалентное приложение, использующее SimpleDB API. Код запускает новую транзак-

цию, вызывает планировщик, чтобы получить план для запроса SQL, открывает план для сканирования, выполняет сканирование и закрывает план.

Листинг 11.1. Два способа доступа к движку базы данных: (a) с использованием JDBC API; (b) с использованием SimpleDB API

```
Driver d = new EmbeddedDriver();
Connection conn = d.connect("studentdb", null);

Statement stmt = conn.createStatement();
String qry = "select sname, gradyear from student";
ResultSet rs = stmt.executeQuery(qry);

while (rs.next())
    System.out.println(rs.getString("sname") + " " + rs.getInt("gradyear"));
rs.close();
```

(a)

```
SimpleDB db = new SimpleDB("studentdb");
Transaction tx = db.newTx();

Planner planner = db.planner();
String qry = "select sname, gradyear from student";
Plan p = planner.createQueryPlan(qry, tx);
Scan s = p.open();

while (s.next())
    System.out.println(s.getString("sname") + " "+ s.getInt("gradyear"));
s.close();
```

(b)

Код в листинге 11.1b использует пять классов из SimpleDB: `SimpleDB`, `Transaction`, `Planner`, `Plan` и `Scan`. Код в листинге 11.1a использует интерфейсы `Driver`, `Connection`, `Statement` и `ResultSet`. В табл. 11.1 показано, как эти классы и интерфейсы взаимосвязаны между собой.

Таблица 11.1. Соответствия между интерфейсами JDBC и классами SimpleDB

Интерфейс JDBC	Классы в SimpleDB
Driver	SimpleDB
Connection	Transaction
Statement	Planner, Plan
ResultSet	Scan
ResultSetMetaData	Schema

Компоненты в каждой строке табл. 11.1 преследуют общую цель. Например, `Connection` и `Transaction` управляют текущей транзакцией, классы `Statement` и `Planner` обрабатывают операторы SQL, а `ResultSet` и `Scan` выполняют итерации по результатам запроса. Это соответствие является ключом к реализации JDBC API для SimpleDB.

11.2. Встроенный интерфейс JDBC

В пакете simpledb.jdbc.embedded находятся классы, реализующие все интерфейсы JDBC. В листинге 11.2 показано определение класса EmbeddedDriver.

Листинг 11.2. Класс EmbeddedDriver

```
public class EmbeddedDriver extends DriverAdapter {
    public EmbeddedConnection connect(String dbname, Properties p) throws SQLException {
        SimpleDB db = new SimpleDB(dbname);
        return new EmbeddedConnection(db);
    }
}
```

Этот класс имеет пустой конструктор. Его единственный метод connect создает новый объект SimpleDB для подключения к указанной базе данных, передает его конструктору EmbeddedConnection и возвращает новый объект EmbeddedConnection. Обратите внимание, что интерфейс Driver в JDBC требует, чтобы метод был объявлен как способный генерировать исключение SQLException, даже если он никогда не будет этого делать.

Интерфейс Driver в JDBC на самом деле имеет больше методов, но ни один из них, кроме connect, не имеет смысла для SimpleDB. Чтобы обеспечить полную реализацию интерфейса Driver в классе, он наследует класс DriverAdapter, реализующий недостающие методы. Определение DriverAdapter показано в листинге 11.3.

Листинг 11.3. Класс DriverAdapter

```
public abstract class DriverAdapter implements Driver {
    public boolean acceptsURL(String url) throws SQLException {
        throw new SQLException("operation not implemented");
    }

    public Connection connect(String url, Properties info) throws SQLException {
        throw new SQLException("operation not implemented");
    }

    public int getMajorVersion() {
        return 0;
    }

    public int getMinorVersion() {
        return 0;
    }

    public DriverPropertyInfo[] getPropertyInfo(String url, Properties info) {
        return null;
    }

    public boolean jdbcCompliant() {
        return false;
    }

    public Logger getParentLogger() throws SQLFeatureNotSupportedException {
        throw new SQLFeatureNotSupportedException("operation not implemented");
    }
}
```

Класс DriverAdapter реализует все методы интерфейса Driver, возвращая значения по умолчанию или генерируя исключения. Класс EmbeddedDriver переопределяет метод, необходимый движку SimpleDB (а именно connect), и использует реализации других методов из DriverAdapter.

В листинге 11.4 показано определение класса EmbeddedConnection. Этот класс управляет транзакциями. Бóльшая часть работы выполняется объектом currentTx типа Transaction. Например, метод commit вызывает currentTx.commit, а затем создает и сохраняет в переменной currentTx новый экземпляр транзакции. Метод createStatement передает конструктору класса EmbeddedStatement объект Planner и ссылку на свой экземпляр EmbeddedConnection.

Листинг 11.4. Класс EmbeddedConnection

```
class EmbeddedConnection extends ConnectionAdapter {
    private SimpleDB db;
    private Transaction currentTx;
    private Planner planner;

    public EmbeddedConnection(SimpleDB db) {
        this.db = db;
        currentTx = db.newTx();
        planner = db.planner();
    }

    public EmbeddedStatement createStatement() throws SQLException {
        return new EmbeddedStatement(this, planner);
    }

    public void close() throws SQLException {
        commit();
    }

    public void commit() throws SQLException {
        currentTx.commit();
        currentTx = db.newTx();
    }

    public void rollback() throws SQLException {
        currentTx.rollback();
        currentTx = db.newTx();
    }

    Transaction getTransaction() {
        return currentTx;
    }
}
```

EmbeddedConnection не реализует интерфейс Connection непосредственно, а наследует класс ConnectionAdapter, который включает реализации по умолчанию всех методов Connection и здесь не показан.

В листинге 11.5 показано определение класса EmbeddedStatement. Этот класс отвечает за выполнение операторов SQL. Его метод executeQuery получает план от планировщика и передает его новому объекту RemoteResultSet для выполнения. Метод executeUpdate просто вызывает соответствующий метод планировщика.

Листинг 11.5. Класс EmbeddedStatement

```
class EmbeddedStatement extends StatementAdapter {
    private EmbeddedConnection conn;
    private Planner planner;

    public EmbeddedStatement(EmbeddedConnection conn, Planner planner) {
        this.conn = conn;
        this.planner = planner;
    }

    public EmbeddedResultSet executeQuery(String qry) throws SQLException {
        try {
            Transaction tx = conn.getTransaction();
            Plan pln = planner.createQueryPlan(qry, tx);
            return new EmbeddedResultSet(pln, conn);
        }
        catch(RuntimeException e) {
            conn.rollback();
            throw new SQLException(e);
        }
    }

    public int executeUpdate(String cmd) throws SQLException {
        try {
            Transaction tx = conn.getTransaction();
            int result = planner.executeUpdate(cmd, tx);
            conn.commit();
            return result;
        }
        catch(RuntimeException e) {
            conn.rollback();
            throw new SQLException(e);
        }
    }

    public void close() throws SQLException {
    }
}
```

Эти два метода также отвечают за реализацию семантики автоматической фиксации транзакций в JDBC. Если оператор SQL выполнился без ошибок, он должен быть зафиксирован. Метод `executeUpdate` сообщает соединению о необходимости зафиксировать текущую транзакцию сразу после завершения оператора изменения. Метод `executeQuery`, напротив, не может немедленно зафиксировать транзакцию, потому что его набор результатов все еще используется. Поэтому он возвращает объекту `Connection` экземпляр `EmbeddedResultSet`, который фиксирует транзакцию в своем методе `close`.

Если во время выполнения оператора SQL что-то пойдет не так, планировщик сгенерирует исключение времени выполнения. Методы `executeQuery` и `executeUpdate` перехватят это исключение, откатят транзакцию и сгенерируют исключение SQL.

Класс `EmbeddedResultSet` имеет методы для выполнения плана запроса; его определение показано в листинге 11.6. Конструктор класса открывает пере-

данный объект Plan и сохраняет полученный образ сканирования. Методы next, getInt, getString и close просто вызывают соответствующие методы образа. Метод close также фиксирует текущую транзакцию, как того требует семантика автоматической фиксации в JDBC. Класс EmbeddedResultSet получает объект Schema из своего плана. Метод getMetaData передает этот объект в конструктор EmbeddedMetaData.

Листинг 11.6. Класс EmbeddedResultSet

```
public class EmbeddedResultSet extends ResultSetAdapter {
    private Scan s;
    private Schema sch;
    private EmbeddedConnection conn;

    public EmbeddedResultSet(Plan plan, EmbeddedConnection conn) throws SQLException {
        s = plan.open();
        sch = plan.schema();
        this.conn = conn;
    }

    public boolean next() throws SQLException {
        try {
            return s.next();
        }
        catch(RuntimeException e) {
            conn.rollback();
            throw new SQLException(e);
        }
    }

    public int getInt(String fldname) throws SQLException {
        try {
            fldname = fldname.toLowerCase(); // для нечувствительности к регистру
            return s.getInt(fldname);
        }
        catch(RuntimeException e) {
            conn.rollback();
            throw new SQLException(e);
        }
    }

    public String getString(String fldname) throws SQLException {
        try {
            fldname = fldname.toLowerCase(); // для нечувствительности к регистру
            return s.getString(fldname);
        }
        catch(RuntimeException e) {
            conn.rollback();
            throw new SQLException(e);
        }
    }

    public ResultSetMetaData getMetaData() throws SQLException {
        return new EmbeddedMetaData(sch);
    }
```

```
    public void close() throws SQLException {
        s.close();
        conn.commit();
    }
}
```

Класс `EmbeddedMetaData` хранит объект `Schema`, переданный конструктору; его определение показано в листинге 11.7. Класс `Schema` имеет методы, аналогичные методам в интерфейсе `ResultSetMetaData`; разница лишь в том, что методы в `ResultSetMetaData` ссылаются на поля по их порядковым номерам, а методы в `Schema` – по именам. По этой причине `EmbeddedMetaData` вынужден преобразовывать порядковые номера в имена полей.

Листинг 11.7. Класс EmbeddedMetaData

```
public class EmbeddedMetaData extends ResultSetMetaDataAdapter {
    private Schema sch;

    public EmbeddedMetaData(Schema sch) {
        this.sch = sch;
    }

    public int getColumnCount() throws SQLException {
        return sch.fields().size();
    }

    public String getColumnName(int column) throws SQLException {
        return sch.fields().get(column-1);
    }

    public int getColumnType(int column) throws SQLException {
        String fldname = getColumnName(column);
        return sch.type(fldname);
    }

    public int getColumnDisplaySize(int column) throws SQLException {
        String fldname = getColumnName(column);
        int fldtype = sch.type(fldname);
        int fldlength = (fldtype == INTEGER) ? 6 : sch.length(fldname);
        return Math.max(fldname.length(), fldlength) + 1;
    }
}
```

11.3. Вызов удаленных методов

В оставшейся части этой главы рассматриваются вопросы реализации серверного интерфейса JDBC. Самая сложная часть серверной реализации JDBC – разработка кода для сервера. К счастью, библиотека Java содержит классы, выполняющие основную работу; эти классы объединены в механизм, известный как механизм *вызова удаленных методов* (Remote Method Invocation, RMI). Данный раздел дает общее представление о RMI, а в следующих разделах демонстрируется, как его использовать в серверном интерфейсе JDBC.

11.3.1. Удаленные интерфейсы

Механизм RMI позволяет Java-программе на одном компьютере (*клиенте*) взаимодействовать с объектами, находящимися на другом компьютере (*сервере*). Чтобы воспользоваться механизмом RMI, необходимо определить один или несколько интерфейсов, расширяющих Java-интерфейс `Remote`; они называются *удаленными интерфейсами*. Также для каждого интерфейса нужно написать класс реализации; эти классы будут находиться на сервере и называются *классами удаленной реализации*. Механизм RMI автоматически создаст необходимые экземпляры классов реализации на стороне клиента; они называются *классами-заглушками*. Когда клиент вызывает метод объекта-заглушки, этот вызов передается по сети на сервер и выполняется там соответствующим удаленным объектом реализации, а результат отправляется обратно в объект-заглушку на клиенте. Проще говоря, удаленный метод вызывается на стороне клиента (с помощью объекта-заглушки), но выполняется на сервере (удаленным объектом реализации).

Движок SimpleDB реализует пять удаленных интерфейсов в своем пакете `simpledb.jdbc.network`: `RemoteDriver`, `RemoteConnection`, `RemoteStatement`, `RemoteResultSet` и `RemoteMetaData`; их определения показаны в листинге 11.8. Эти удаленные интерфейсы являются зеркальным отражением соответствующих интерфейсов JDBC с двумя отличиями:

- реализуются только самые основные методы JDBC, перечисленные в листинге 2.1;
- генерируется исключение `RemoteException` (как того требует механизм RMI) вместо `SQLException` (как того требует JDBC).

Листинг 11.8. Удаленные интерфейсы в SimpleDB

```
public interface RemoteDriver extends Remote {
    public RemoteConnection connect() throws RemoteException;
}

public interface RemoteConnection extends Remote {
    public RemoteStatement createStatement() throws RemoteException;
    public void            close()           throws RemoteException;
}

public interface RemoteStatement extends Remote {
    public RemoteResultSet executeQuery(String qry)  throws RemoteException;
    public int             executeUpdate(String cmd) throws RemoteException;
}

public interface RemoteResultSet extends Remote {
    public boolean        next()                    throws RemoteException;
    public int            getInt(String fldname)    throws RemoteException;
    public String         getString(String fldname) throws RemoteException;
    public RemoteMetaData getMetaData()             throws RemoteException;
    public void           close()                   throws RemoteException;
}
```

```
public interface RemoteMetaData extends Remote {
    public int    getColumnCount()                throws RemoteException;
    public String getColumnName(int column)       throws RemoteException;
    public int    getColumnType(int column)       throws RemoteException;
    public int    getColumnDisplaySize(int column) throws RemoteException;
}
```

Чтобы понять, как работает RMI, рассмотрим фрагмент клиентского кода (листинг 11.9). Каждая переменная в этом фрагменте обозначает удаленный интерфейс. Однако поскольку этот код выполняется на стороне клиента, мы знаем, что фактические объекты, на которые ссылаются эти переменные, являются экземплярами классов-заглушек. Здесь не показано, как переменная `rdvr` получает ссылку на свою заглушку, – данный объект извлекается из реестра RMI, о котором рассказывается в разделе 11.3.2.

Листинг 11.9. Использование удаленных интерфейсов на стороне клиента

```
RemoteDriver rdvr = ...
RemoteConnection rconn = rdvr.connect();
RemoteStatement rstmt = rconn.createStatement();
```

Рассмотрим вызов `rdvr.connect`. Заглушка реализует свой метод `connect`, который отправляет запрос по сети соответствующему объекту реализации `RemoteDriver` на сервере. Этот удаленный объект реализации выполняет свой метод `connect` на сервере и там же на сервере создает новый объект реализации `RemoteConnection`. Затем заглушка для этого нового удаленного объекта отправляется обратно клиенту, который сохраняет ее в переменной `rconn`.

Теперь рассмотрим вызов `rconn.createStatement`. Объект-заглушка посылает запрос соответствующему объекту реализации `RemoteConnection` на сервере. Этот удаленный объект выполняет свой метод `createStatement`, там же на сервере создает новый объект реализации `RemoteStatement`, а его заглушка возвращается клиенту.

11.3.2. Реестр RMI

Каждый объект-заглушка на стороне клиента хранит ссылку на соответствующий удаленный объект реализации на стороне сервера. Клиент, имея объект-заглушку, может с его помощью взаимодействовать с сервером, и эти взаимодействия могут создавать другие объекты-заглушки для использования клиентом. Но остается вопрос: как клиент получает первую заглушку? Механизм RMI решает эту проблему с помощью программы, называемой *реестром RMI*. Сервер помещает объекты-заглушки в реестр RMI, а клиенты получают их из него.

Сервер SimpleDB помещает в реестр только один объект типа `RemoteDriver`. Делается это всего тремя строками кода в пакете `simpledb.server.StartServer`:

```
Registry reg = LocateRegistry.createRegistry(1099);
RemoteDriver d = new RemoteDriverImpl();
reg.rebind("simpledb", d);
```

Метод `createRegistry` запускает реестр RMI на локальном компьютере и назначает ему заданный порт. (По соглашению назначается порт 1099.) Вызов метода `reg.rebind` создает заглушку для удаленного объекта реализации `d`, сохраняет ее в реестре RMI и делает доступной для клиентов под именем «simpledb».

Клиент может запросить заглушку из реестра, вызвав метод `lookup` реестра. В SimpleDB такой запрос выполняется следующими строками в классе `NetworkDriver`:

```
String host = url.replace("jdbc:simpledb://", "");
Registry reg = LocateRegistry.getRegistry(host, 1099);
RemoteDriver rdvr = (RemoteDriver) reg.lookup("simpledb");
```

Метод `getRegistry` принимает имя узла и номер порта и возвращает ссылку на реестр RMI, находящийся на этом узле. Вызов `reg.lookup` подключается к реестру RMI, извлекает из него заглушку с именем «simpledb» и возвращает ее вызывающему коду.

11.3.3. Особенности многопоточного выполнения

Разрабатывая большие Java-программы, всегда полезно иметь четкое представление о том, какие потоки выполнения действуют в каждый момент времени. При использовании серверной версии SimpleDB всегда будет выполняться два набора потоков: потоки на клиентских компьютерах и потоки на сервере.

Каждый клиент имеет свой поток выполнения на своем компьютере. Этот поток продолжает действовать до завершения клиента; все объекты-заглушки клиента вызываются из этого потока. С другой стороны, каждый удаленный объект на сервере выполняется в отдельном потоке. Удаленный объект на стороне сервера можно рассматривать как «мини-сервер», который ожидает подключения своей заглушки. После установки соединения удаленный объект выполняет запрошенную операцию, отправляет результат клиенту и терпеливо ждет другого соединения. Объект `RemoteDriver`, созданный при помощи `simpledb.server.Startup`, запускается в потоке, который можно считать потоком «сервера базы данных».

Всякий раз, когда клиент вызывает удаленный метод, клиентский поток приостанавливается, пока соответствующий серверный поток не выполнит порученную операцию, и возобновляется, получив возвращаемое значение. Точно так же поток на стороне сервера будет простаивать, пока не будет вызван один из его методов, и вновь приостановится после выполнения метода. То есть в каждый конкретный момент времени действует только один поток – клиентский или серверный. Это создает впечатление, что при выполнении удаленных вызовов поток клиента как бы перемещается туда-обратно между клиентом и сервером. Такое представление помогает наглядно представить поток управления в клиент-серверном приложении, однако важно понимать, что происходит на самом деле.

Один из способов отличить клиентский поток от серверного – вывести что-нибудь. При вызове `System.out.println` из потока клиента вывод появится на компьютере клиента, а при вызове из потока сервера – на сервере.

11.4. Реализация удаленных интерфейсов

Для реализации каждого удаленного интерфейса требуются два класса: класс-заглушка и класс удаленной реализации. По соглашению классам удаленной реализации присваивается имя интерфейса с окончанием «Impl». Знать имена классов-заглушек вам никогда не понадобится.

К счастью, взаимодействия между всеми объектами на стороне сервера и их заглушками не зависят от конкретных удаленных интерфейсов, поэтому всю коммуникацию берет на себя библиотека RMI. Программист должен написать только код, специфичный для каждого конкретного интерфейса. Проще говоря, программист вообще не должен писать классы-заглушки и пишет только те части классов удаленной реализации, которые определяют ответ сервера на вызов каждого метода.

Класс `RemoteDriverImpl` – это точка входа в сервер SimpleDB; его определение показано в листинге 11.10. Класс начальной загрузки `simpledb.server.Startup` создает лишь один объект `RemoteDriverImpl` и помещает его единственную заглушку в реестр RMI. Каждый раз, когда вызывается метод `connect` этого объекта (через заглушку), он создает новый удаленный объект `RemoteConectionImpl` на сервере и запускает его в новом потоке. Механизм RMI прозрачно создает соответствующий объект-заглушку `RemoteConnection` и возвращает его клиенту.

Листинг 11.10. Класс RemoteDriverImpl в SimpleDB

```
public class RemoteDriverImpl extends UnicastRemoteObject
                              implements RemoteDriver {

    public RemoteDriverImpl() throws RemoteException {
    }

    public RemoteConnection connect() throws RemoteException {
        return new RemoteConnectionImpl();
    }
}
```

Обратите внимание, что этот класс только создает серверные объекты – он не содержит сетевого кода или ссылок на связанный с ним объект-заглушку, и когда ему нужно создать новый удаленный объект, он создает только удаленный объект реализации (но не объект-заглушку). Все другие задачи решает RMI-класс `UnicastRemoteObject`.

Фактически `RemoteDriverImpl` действует точно так же, как `EmbeddedDriver` в листинге 11.2, отличаясь только отсутствием аргументов в методе `connect`. Причина этого отличия в том, что драйвер встраиваемой версии SimpleDB может выбирать базу данных для подключения, тогда как драйвер серверной версии должен подключаться к базе данных, связанной с удаленным объектом `SimpleDB`.

В общем случае каждый JDBC-класс удаленной реализации функционально эквивалентен соответствующему JDBC-классу встраиваемой версии. В качестве еще одного примера рассмотрим класс `RemoteConnectionImpl`, определение которого показано в листинге 11.11. Обратите внимание на близкое сходство с классом `EmbeddedConnection` в листинге 11.4. Определения классов `RemoteStatementImpl`, `RemoteResultsetImpl` и `RemoteMetaDataImpl` точно так же похожи на свои встраиваемые эквиваленты и поэтому не приводятся в книге.

Листинг 11.11. Класс RemoteConnectionImpl в SimpleDB

```
class RemoteConnectionImpl extends UnicastRemoteObject
                           implements RemoteConnection {
    private SimpleDB db;
    private Transaction currentTx;
    private Planner planner;

    RemoteConnectionImpl(SimpleDB db) throws RemoteException {
        this.db = db;
        currentTx = db.newTx();
        planner = db.planner();
    }

    public RemoteStatement createStatement() throws RemoteException {
        return new RemoteStatementImpl(this, planner);
    }

    public void close() throws RemoteException {
        currentTx.commit();
    }

    Transaction getTransaction() {
        return currentTx;
    }

    void commit() {
        currentTx.commit();
        currentTx = db.newTx();
    }

    void rollback() {
        currentTx.rollback();
        currentTx = db.newTx();
    }
}
```

11.5. Реализация интерфейсов JDBC

Реализация удаленных классов RMI в SimpleDB поддерживает все особенности, которых требуют интерфейсы JDBC в `java.sql`, за исключением двух: методы RMI не генерируют исключений SQL и реализуют не все методы интерфейса. То есть нам доступны классы, реализующие интерфейсы `RemoteDriver`, `RemoteConnection` и т. д., но в действительности нам нужны классы, реализующие интерфейсы `Driver`, `Connection` и т. д. Это распространенная проблема в объектно-ориентированном программировании, которая часто решается реализацией необходимых классов в виде клиентских *оберток* для соответствующих объектов-заглушек.

Чтобы увидеть, как работает обертывание, рассмотрим класс `NetworkDriver`, определение которого показано в листинге 11.12. Его метод `connect` должен вернуть объект типа `Connection`, который в данном случае является объектом `NetworkConnection`. Для этого метод сначала получает заглушку `RemoteDriver` из реестра RMI, затем вызывает метод `connect` заглушки, чтобы получить заглушку

RemoteConnection, и, наконец, создает нужный объект NetworkConnection, передавая заглушку RemoteConnection в конструктор.

Листинг 11.12. Определение класса NetworkDriver в SimpleDB

```
public class NetworkDriver extends DriverAdapter {
    public Connection connect(String url, Properties prop) throws SQLException {
        try {
            String host = url.replace("jdbc:simpledb://", "");
            Registry reg = LocateRegistry.getRegistry(host, 1099);
            RemoteDriver rdvr = (RemoteDriver) reg.lookup("simpledb");
            RemoteConnection rconn = rdvr.connect();
            return new NetworkConnection(rconn);
        }
        catch (Exception e) {
            throw new SQLException(e);
        }
    }
}
```

Другие интерфейсы JDBC реализуются аналогично. Например, в листинге 11.13 показано определение класса NetworkConnection. Его конструктор принимает объект RemoteConnection, который он использует в своих методах. Метод createStatement передает вновь созданный объект RemoteStatement в конструктор NetworkStatement и возвращает полученный объект. Всякий раз, когда в этих классах объекты-заглушки генерируют RemoteException, это исключение перехватывается и преобразуется в SQLException.

Листинг 11.13. Определение класса NetworkConnection в SimpleDB

```
public class NetworkConnection extends ConnectionAdapter {
    private RemoteConnection rconn;

    public NetworkConnection(RemoteConnection c) {
        rconn = c;
    }

    public Statement createStatement() throws SQLException {
        try {
            RemoteStatement rstmt = rconn.createStatement();
            return new NetworkStatement(rstmt);
        }
        catch(Exception e) {
            throw new SQLException(e);
        }
    }

    public void close() throws SQLException {
        try {
            rconn.close();
        }
        catch(Exception e) {
            throw new SQLException(e);
        }
    }
}
```

11.6. Итоги

- Прикладная программа может получить доступ к базе данных двумя способами: через встроенное соединение и через соединение с сервером. SimpleDB, как и большинство движков баз данных, реализует JDBC API для обоих типов соединений.
- Встроенное JDBC-соединение в SimpleDB использует тот факт, что для каждого интерфейса JDBC имеется соответствующий класс SimpleDB.
- Поддержка соединений с сервером в SimpleDB реализована с использованием Java-механизма *вызова удаленных методов (Remote Method Invocation, RMI)*. Для всех интерфейсов JDBC имеются соответствующие удаленные интерфейсы RMI. Основное отличие последних заключается в том, что они генерируют исключение `RemoteException` (как того требует RMI) вместо `SQLException` (как это требует JDBC).
- Каждый удаленный объект реализации выполняется в собственном потоке на стороне сервера, ожидая, пока заглушка не свяжется с ним. Код запуска SimpleDB создает удаленный объект реализации типа `RemoteDriver` и сохраняет его заглушку в *реестре RMI*. Когда JDBC-клиенту понадобится установить соединение с системой баз данных, он получит заглушку из реестра и вызовет ее метод `connect`.
- Метод `connect` имеет реализацию, типичную для удаленных методов RMI. Он создает новый объект `RemoteConnectionImpl` на сервере, который выполняется в своем потоке и возвращает заглушку для этого объекта клиенту. После этого клиент может использовать полученную заглушку для вызова методов интерфейса `Connection`, которые будут транслироваться в вызовы соответствующих методов объекта реализации на стороне сервера.
- JDBC-клиенты, подключающиеся к серверу, не используют удаленные заглушки непосредственно, потому что они реализуют удаленные интерфейсы вместо интерфейсов JDBC. Вместо этого объекты на стороне клиента *обертывают* свои объекты-заглушки.

11.7. Для дополнительного чтения

Существует масса книг, описывающих механизм RMI, такие как Grosso (2001). Кроме того, имеется руководство по RMI, опубликованное компанией Oracle по адресу: https://docs.oracle.com/javase/tutorial/rmi/index.html.

Реализация драйвера, используемая в SimpleDB, формально называется драйвером 4-го типа. Описание и сравнение драйверов четырех различных типов вы найдете в статье Nanda (2002). Сопутствующая статья Nanda et al. (2002) проведет вас через процесс создания аналогичного драйвера 3-го типа.

Grosso, W. (2001). «Java RMI». Sebastopol, CA: O'Reilly.

Nanda, N. (202). «Drivers in the wild». JavaWorld. Доступна по адресу: https://www.infoworld.com/article/2076117/jdbc-drivers-in-the-wild.html.

Nanda, N., & Kumar, S. (2002). «Create your own Type 3 JDBC driver». JavaWorld. Доступна по адресу: https://www.infoworld.com/article/2074249/create-your-own-type-3-jdbc-driver--part-1.html.

11.8. Упражнение

Теория

11.1. Посмотрите, как демонстрационный клиент StudentMajor.java использует классы из `simpledb.jdbc.network`. Какие объекты на стороне сервера он создает? Какие объекты на стороне клиента он создает? Какие потоки выполнения создаются?

11.2. Методам `executeQuery` и `executeUpdate` в классе `RemoteStatementImpl` необходимы транзакции. Объект `RemoteStatementImpl` получает их, вызывая `rconn.getTransaction` каждый раз, когда вызывается `executeQuery` или `executeUpdate`. Проще было бы передавать транзакцию каждому объекту `RemoteStatementImpl` через его конструктор при создании. Однако это очень плохая идея. Приведите сценарий, в котором такой подход может вызвать ошибку.

11.3. Мы знаем, что удаленные объекты реализации находятся на сервере. Но нужны ли классы удаленной реализации клиенту? Нужны ли удаленные интерфейсы клиенту? Создайте конфигурацию клиента с папками `sql` и `remote` из SimpleDB. Какие файлы классов можно удалить из этих папок, не нарушив работу клиента? Объясните свои результаты.

Практика

11.4. Измените JDBC-классы во встроенной и серверной версиях SimpleDB и добавьте в них реализации следующих методов интерфейса `ResultSet`:

a) метода `beforeFirst`, который перемещает указатель текущей позиции перед первой записью в наборе результатов (т. е. устанавливает его в исходное состояние). Используйте тот факт, что образ сканирования имеет метод `beforeFirst`, который делает то же самое;

b) метода `absolute(int n)`, который перемещает указатель текущей позиции на `n`-ю запись. (В образе сканирования нет соответствующего метода `absolute`.)

11.5. В упражнении 8.13 предлагалось реализовать методы образа сканирования `afterLast` и `previous`.

a) Измените реализацию `ResultSet`, добавив эти методы.

b) Протестируйте свой код, изменив класс `SimpleIJ` в демонстрационном JDBC-клиенте так, чтобы он печатал строки выходных таблиц в обратном порядке.

11.6. В упражнении 9.18 предлагалось реализовать поддержку неопределенных значений (`null`) в SimpleDB. JDBC-методы `getInt` и `getString` не возвращают неопределенных значений. Клиент JDBC может определить, было ли последнее полученное значение неопределенным, только с помощью метода `wasNull` класса `ResultSet`, как объяснялось в упражнении 2.8.

a) Добавьте этот метод в реализацию `ResultSet`.

b) Напишите программу JDBC, проверяющую ваш код.

11.7. JDBC-интерфейс `Statement` имеет метод `close`, который закрывает возможно еще открытый набор результатов, полученный данным оператором. Реализуйте этот метод.

11.8. Стандарт JDBC требует, чтобы метод `Connection.close` закрывал все операторы, открытые в данном соединении (как в упражнении 11.7). Реализуйте это требование.

11.9. Стандарт JDBC указывает, что соединение должно закрываться автоматически, когда объект `Connection` утилизируется сборщиком мусора (например, когда клиентская программа завершает работу). Эта важная особенность позволяет системе баз данных высвобождать ресурсы, которые не были освобождены забывчивыми клиентами. Используйте Java-конструкцию `finalizer` для реализации этой особенности.

11.10. SimpleDB реализует режим автоматической фиксации, в котором система сама решает, когда зафиксировать транзакцию. Стандарт JDBC позволяет клиентам отключать режим автоматической фиксации и фиксировать или откатывать свои транзакции явно. JDBC-интерфейс `Connection` имеет метод `setAutoCommit(boolean ac)`, с помощью которого клиент может включать или выключать режим автоматической фиксации, метод `getAutoCommit`, возвращающий текущее состояние режима автоматической фиксации, а также методы `commit` и `rollback`. Реализуйте эти методы.

11.11. Сервер SimpleDB позволяет подключиться к нему любому желающему. Измените класс `NetworkDriver` так, чтобы его метод `connect` выполнял аутентификацию пользователей. Он должен извлечь имя пользователя и пароль из переданного ему объекта `Properties`, сравнить их с содержимым текстового файла на стороне сервера и сгенерировать исключение, если совпадение не найдено. Для простоты можно считать, что новые имена пользователей и пароли добавляются (или удаляются) простым редактированием файла на сервере.

11.12. Измените `RemoteConnectionImpl` так, чтобы он ограничивал число одновременно обслуживаемых соединений. Что должна делать система при попытке клиента подключиться к ней, если не осталось доступных соединений?

11.13. В разделе 2.2.4 отмечалось, что JDBC имеет интерфейс `PreparedStatement`, который отделяет этап планирования запроса от его выполнения. Запрос может планироваться один раз и выполняться многократно, возможно, с разными значениями некоторых констант в нем. Рассмотрим следующий фрагмент кода:

```
String qry = "select SName from STUDENT where MajorId = ?";
PreparedStatement ps = conn.prepareStatement(qry);
ps.setInt(1, 20);
ResultSet rs = ps.executeQuery();
```

Символ «?» в запросе обозначает неизвестную константу, значение которой будет указано непосредственно перед выполнением. Запрос может иметь несколько неизвестных констант. Метод `setInt` (или `setString`) присваивает значение *i*-й неизвестной константе.

а) Предположим, что параметризованный запрос не содержит неизвестных констант. Тогда конструктор `PreparedStatement` должен получить план от планировщика, а метод `executeQuery` – передать этот

план в конструктор `ResultSet`. Реализуйте этот особый случай, внеся изменения в пакет `jdbc`, но не изменяя синтаксический анализатор или планировщик.

b) Теперь измените свою реализацию так, чтобы она обрабатывала неизвестные константы. Для этого вам придется добавить в синтаксический анализатор распознавание символов «?». Планировщик должен иметь возможность получить список неизвестных констант от анализатора; также должна иметься возможность присвоить значения этим константам с помощью методов `setInt` и `setString`.

11.14. Предположим, вы запускаете клиентскую программу JDBC; однако для завершения ей требуется слишком много времени, и вы прерываете ее комбинацией клавиш <CTRL-C>.

a) Как это действие повлияет на других клиентов JDBC, работающих с сервером?

b) Когда и как сервер заметит, что ваша клиентская программа JDBC прекратила выполнение? Что он сделает, когда узнает об этом?

c) Как лучше обработать эту ситуацию на сервере?

d) Спроектируйте и реализуйте свой ответ на вопрос в п. «c».

11.15. Напишите Java-класс `Shutdown`, метод `main` которого корректно завершает работу сервера. Он должен запретить прием новых соединений и дождаться, пока будут закрыты существующие. Когда не останется ни одной активной транзакции, код должен записать блокирующую контрольную точку в журнал и вывести в консоль сообщение «ok to shutdown» (готов к остановке). (*Подсказка:* самый простой способ завершить работу – удалить объект `SimpleDB` из реестра RMI. Также помните, что этот метод будет выполняться не в серверной виртуальной машине Java, поэтому вам необходимо внести какие-то изменения в сервер, чтобы он распознавал вызов `Shutdown`.)

Глава 12

Индексирование

При выполнении запроса к таблице пользователя часто интересуют только некоторые записи в ней, например записи, имеющие определенное значение в некотором поле. *Индекс* – это файл, помогающий движку базы данных быстро найти такие записи, не просматривая всю таблицу. В этой главе рассматриваются три распространенных способа реализации индексов: статическое хеширование, расширяемое хеширование и B-деревья.

12.1. Ценность индексирования

До сих пор в этой книге предполагалось, что записи в таблицах не имеют конкретной организации. Однако подходящая организация таблиц может значительно повысить эффективность выполнения некоторых запросов. Чтобы лучше понять проблему, представьте телефонный справочник.

Телефонный справочник – это, по сути, большая таблица, записи которой содержат имена, адреса и номера телефонов абонентов. Эта таблица отсортирована сначала по фамилиям, а затем по именам. Предположим, что вы хотите узнать номер телефона конкретного человека. Ускорить поиск вам поможет тот факт, что записи отсортированы по имени. Например, можно выполнить бинарный поиск, который в худшем случае потребует просмотреть $\log_2 N$ записей, где N – общее число записей в справочнике. Это очень быстро. Например, допустим, что $N = 1\,000\,000$. Тогда $\log_2 N < 20$, то есть вам никогда не придется просматривать больше 20 записей, чтобы найти нужного человека в справочнике, содержащем номера миллиона человек.

Телефонный справочник отлично приспособлен для поиска абонента по имени, но не подходит для быстрого поиска, например по номеру телефона или по адресу. Единственный способ получить эту информацию из телефонной книги – просмотреть каждую запись в ней. Такой поиск может оказаться очень медленным.

Для эффективного поиска абонентов по номеру телефона нужен справочник, отсортированный по номерам телефонов (такие справочники еще называют «обратными телефонными справочниками»). Однако такой справочник удобен, только если известен номер телефона. Если вы решите найти в таком справочнике номер телефона конкретного абонента, то вам снова придется просмотреть каждую запись.

Этот пример наглядно иллюстрирует важное ограничение организации таблиц: *таблицу можно организовать только каким-то одним способом*. Если

необходимо, чтобы поиск выполнялся быстро по номеру телефона или по имени абонента, вам потребуются две отдельные копии телефонной книги, каждая со своей организацией. А если понадобится возможность быстро находить номер телефона по известному адресу, вам потребуется третий экземпляр телефонного справочника, отсортированный по адресу.

Описанное выше в равной степени относится и к таблицам в базе данных. Чтобы иметь возможность эффективно находить в таблице записи с определенным значением некоторого поля, нужна версия таблицы, организованная по этому полю. Механизмы баз данных удовлетворяют эту потребность, поддерживая *индексы*. Таблица может иметь один или несколько индексов, каждый из которых определен для отдельного поля. Каждый индекс действует подобно версии таблицы, организованной по соответствующему полю. Например, индекс для поля `MajorId` в таблице STUDENT облегчает поиск записей STUDENT с определенным идентификатором кафедры.

Если говорить конкретнее, индекс – это файл с *индексными записями*. Файл индекса имеет одну индексную запись для каждой записи в соответствующей таблице. Каждая индексная запись имеет два поля, которые хранят идентификатор соответствующей записи в таблице и значение индексированного поля в этой записи. В SimpleDB эти поля в индексной записи имеют имена *datarid* и *dataval*. На рис. 12.1 изображена таблица STUDENT и два ее индекса: один для поля `Sid` и другой для поля `MajorId`. Прямоугольники обозначают записи в таблице STUDENT. Значения поля dataval в индексных записях показаны в квадратиках, а значения поля datarid изображены в виде стрелок, указывающих на соответствующие записи в STUDENT.

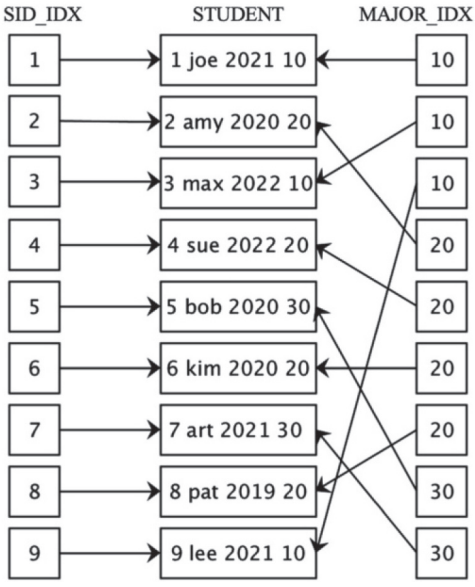


Рис. 12.1. Индексы SID_IDX и MAJOR_IDX

Движок организует записи в файле индекса по полю dataval. В разделах 12.3–12.5 будут рассмотрены некоторые более сложные способы организа-

ции записей. А пока для простоты будем считать, что записи в индексе отсортированы по полю dataval, как показано на рис. 12.1. Эту организацию можно использовать, как описывается далее.

Допустим, вы хотите найти запись STUDENT со значением 6 в поле `Sid`. Для этого вы выполняете бинарный поиск в SID_IDX и находите индексную запись с dataval = 6, а затем из поля datarid извлекаете идентификатор искомой записи STUDENT (которая в данном случае соответствует студенту с именем Kim).

Теперь предположим, что вы решили найти все записи STUDENT, в которых поле `MajorId` содержит значение 20. Для этого вы выполняете бинарный поиск в MAJOR_IDX, чтобы найти первую индексную запись с dataval = 20. Обратите внимание, что благодаря сортировке остальные три индексные записи с dataval = 20 следуют сразу же за первой. Затем вы в цикле выбираете эти четыре индексные записи и, используя значение в поле datarid, для каждой извлекаете соответствующую запись STUDENT.

Насколько эффективно такое применение индексов? В отсутствие индексов лучшее, что можно предпринять при обработке любого запроса, – выполнить последовательный поиск в таблице STUDENT. Вспомните статистику в табл. 7.2, где указано, что в таблице STUDENT хранится 45 000 записей и в один дисковый блок умещается 10 записей. Таким образом, для последовательного сканирования таблицы STUDENT может потребоваться обратиться к 4500 блокам.

Стоимость использования индекса SID_IDX можно оценить следующим образом. Индекс будет иметь 45 000 записей, то есть в ходе бинарного поиска в индексе потребуется просмотреть не более 16 индексных записей (потому что $\log_2(45\,000) < 16$); в худшем случае каждая из этих индексных записей будет находиться в отдельном блоке. Дополнительно потребуется еще одно обращение к блоку, чтобы по значению datarid из найденной индексной записи получить искомую запись STUDENT, что дает в результате всего 17 обращений к блокам – значительная экономия по сравнению с последовательным сканированием.

Аналогично можно рассчитать стоимость использования индекса MAJOR_IDX. Согласно статистике в табл. 7.2, всего в университете насчитывается 40 кафедр, то есть на каждой кафедре в среднем обучается 1125 студентов; следовательно, MAJOR_IDX будет иметь около 1125 записей для каждого уникального значения dataval. Индексные записи невелики, поэтому примем, что в одном блоке умещается 100 таких записей. Таким образом, 1125 индексных записей уместятся в 12 блоков. И снова бинарный поиск в индексе потребует до 16 обращений к блокам, чтобы найти первую индексную запись. Поскольку все индексные записи с одинаковым значением dataval следуют в файле друг за другом, для извлечения этих 1125 индексных записей потребуется еще 12 обращений к блокам. Суммируя, получаем, что для выполнения запроса понадобится 16 + 12 = 28 обращений к блокам в MAJOR_IDX. Это очень эффективно. Проблема, однако, заключается в количестве обращений к блокам, где хранятся записи STUDENT. Так как каждая из 1125 индексных записей хранит свою ссылку datarid, для извлечения каждой соответствующей записи STUDENT придется один раз обратиться к блоку. В результате для выполнения запроса потребуется 1125 обращений к блокам с таблицей STUDENT и всего 1125 + 28 = 1153 обращения. Это число намного больше, чем в случае с индексом SID_IDX, тем не менее

использование MAJOR_IDX увеличивает скорость примерно в четыре раза по сравнению с последовательным поиском.

Теперь предположим, что в университете имеется только 9 кафедр, а не 40. Тогда на каждой кафедре обучалось бы 5000 студентов, а это значит, что в MAJOR_IDX имелось бы около 5000 индексных записей для каждого уникального значения dataval. Давайте подсчитаем стоимость выполнения предыдущего запроса. Теперь мы получим 5000 записей в MAJOR_IDX, ссылающихся на искомые записи STUDENT, а это значит, что нам понадобится 5000 раз обратиться к блокам с таблицей STUDENT! То есть использование индекса приведет к большему количеству обращений к блокам, чем при последовательном сканировании STUDENT. В этом случае использование индекса замедлит обработку запроса по сравнению с простым сканированием таблицы STUDENT. Индекс окажется совершенно бесполезным.

Эти наблюдения можно обобщить в виде следующего правила: *полезность индекса для поля A пропорциональна количеству уникальных значений в этом поле в таблице*[1]. Согласно этому правилу, индекс наиболее полезен, когда индексируемое поле является ключом таблицы (например, SID_IDX), потому что каждая запись имеет свое уникальное значение ключа. И наоборот, согласно правилу индекс будет бесполезен, если число уникальных значений в поле A меньше количества записей в блоке (см. упражнение 12.15).

12.2. Индексы в SimpleDB

Предыдущий раздел проиллюстрировал способы использования индексов: поиск в индексе первой записи с указанным значением в поле dataval; выборка всех последующих индексных записей с тем же значением dataval; извлечение значений datarid из найденных индексных записей. В SimpleDB все эти операции формализует интерфейс `Index`. Его определение показано в листинге 12.1.

Листинг 12.1. Определение интерфейса Index в SimpleDB

```
public interface Index {
    public void    beforeFirst(Constant searchkey);
    public boolean next();
    public RID     getDataRid();
    public void    insert(Constant dataval, RID datarid);
    public void    delete(Constant dataval, RID datarid);
    public void    close();
}
```

Эти методы похожи на методы в `TableScan`: клиент может устанавливать текущую позицию в индексе в начало и перемещаться по записям, извлекать содержимое текущей индексной записи, а также вставлять и удалять индексные записи. Однако, поскольку индексы используются хорошо известными конкретными способами, методы в `Index` более специфичны, чем в `TableScan`.

В частности, клиент SimpleDB всегда выполняет поиск в индексе, указывая определенное значение (называемое *ключом поиска*) и извлекая индексные

[1] Следует учитывать, что это правило действует только в случае равномерного распределения и только для условий вида «A = константа» (см. также упражнение 12.19). – *Прим. ред.*

записи, имеющие соответствующее значение в поле dataval. Метод `beforeFirst` принимает аргумент с ключом поиска. Последующие вызовы `next` перемещают индекс к следующей записи, в которой значение dataval равно ключу поиска, и возвращают `false`, если таких записей больше не существует.

Кроме того, для работы с индексом не требуются универсальные методы `getInt` и `getString`, потому что все индексные записи имеют одни и те же два поля. Более того, клиенту никогда не придется извлекать dataval из индексной записи, потому что это значение всегда будет равно ключу поиска. Таким образом, единственный метод, который понадобится, – это `getDataRid`, возвращающий значение datarid текущей индексной записи.

Класс `IndexRetrievalTest` в листинге 12.2 иллюстрирует использование индекса. Он открывает индекс для поля `MajorId`, чтобы найти всех, кто обучается на кафедре с идентификатором 20, извлекает соответствующие записи STUDENT и выводит имена студентов. Обратите внимание, что для извлечения записей STUDENT код использует механизм сканирования таблицы, хотя на самом деле таблица не «сканируется». Вместо этого вызывается метод `moveToRid` образа сканирования, чтобы перейти к нужной записи.

Листинг 12.2. Использование индексов в SimpleDB

```
public class IndexRetrievalTest {
    public static void main(String[] args) {
        SimpleDB db = new SimpleDB("studentdb");
        Transaction tx = db.newTx();
        MetadataMgr mdm = db.mdMgr();

        // Открыть и просканировать таблицу с данными.
        Plan studentplan = new TablePlan(tx, "student", mdm);
        Scan studentscan = studentplan.open();

        // Открыть индекс для поля MajorId.
        Map<String,IndexInfo> indexes = mdm.getIndexInfo("student", tx);
        IndexInfo ii = indexes.get("majorid");
        Index idx = ii.open();

        // Извлечь все индексные записи, где dataval = 20.
        idx.beforeFirst(new Constant(20));
        while (idx.next()) {
            // Использовать datarid для получения соответствующей записи STUDENT.
            RID datarid = idx.getDataRid();
            studentscan.moveToRid(datarid);
            System.out.println(studentscan.getString("sname"));
        }

        // Закрыть индекс и таблицу.
        idx.close();
        studentscan.close();
        tx.commit();
    }
}
```

API классов метаданных, связанных с индексом, был показан в листинге 7.11. В частности, метод `getIndexInfo` в `IndexMgr` возвращает ассоциативный массив,

содержащий метаданные IndexInfo всех имеющихся индексов для указанной таблицы. Чтобы получить нужный объект Index, достаточно выбрать соответствующий объект IndexInfo из ассоциативного массива и вызвать его метод open.

Класс IndexUpdateTest в листинге 12.3 демонстрирует, как движок базы данных обрабатывает изменения в таблице. Код выполняет две операции: сначала вставляет новую запись в таблицу STUDENT, а затем удаляет существующую запись. Обрабатывая вставку новой записи, код должен вставить соответствующую запись в каждый индекс. Удаление обрабатывается аналогично. Обратите внимание, что код начинается с открытия всех индексов для STUDENT и сохранения их в ассоциативном массиве. В дальнейшем код может обращаться к этому массиву каждый раз, когда возникает необходимость выполнить какую-то операцию с каждым индексом.

Код в листингах 12.2 и 12.3 манипулирует индексами, не зная и не заботясь об их фактической реализации. Единственное требование – индексы должны реализовать интерфейс Index. В разделе 12.1 предполагалось, что индексы имеют простую реализацию, поддерживающую сортировку и бинарный поиск. Однако на практике такая реализация не применяется, потому что она не использует преимущества блочной структуры индексного файла. В разделах 12.3–12.5 представлены три более эффективные реализации – две из них основаны на хешировании и одна на сортированных деревьях.

Листинг 12.3. Изменение индексов при изменении записей с данными

```
public class IndexUpdateTest {
    public static void main(String[] args) {
        SimpleDB db = new SimpleDB("studentdb");
        Transaction tx = db.newTx();
        MetadataMgr mdm = db.mdMgr();
        Plan studentplan = new TablePlan(tx, "student", mdm);
        UpdateScan studentscan = (UpdateScan) studentplan.open();

        // Создать ассоциативный массив со всеми индексами для STUDENT.
        Map<String,Index> indexes = new HashMap<>();
        Map<String,IndexInfo> idxinfo = mdm.getIndexInfo("student", tx);
        for (String fldname : idxinfo.keySet()) {
            Index idx = idxinfo.get(fldname).open();
            indexes.put(fldname, idx);
        }

        // Задача 1: вставить новую запись STUDENT для студента Sam.
        //    Сначала вставить запись в STUDENT.
        studentscan.insert();
        studentscan.setInt("sid", 11);
        studentscan.setString("sname", "sam");
        studentscan.setInt("gradyear", 2023);
        studentscan.setInt("majorid", 30);
        //    Затем вставить запись в каждый индекс.
        RID datarid = studentscan.getRid();
        for (String fldname : indexes.keySet()) {
            Constant dataval = studentscan.getVal(fldname);
            Index idx = indexes.get(fldname);
            idx.insert(dataval, datarid);
        }
```

```
        // Задача 2: найти и удалить запись для студента Joe.
        studentscan.beforeFirst();
        while (studentscan.next()) {
            if (studentscan.getString("sname").equals("joe")) {
                // Сначала удалить индексную запись для Joe.
                RID joeRid = studentscan.getRid();
                for (String fldname : indexes.keySet()) {
                    Constant dataval = studentscan.getVal(fldname);
                    Index idx = indexes.get(fldname);
                    idx.delete(dataval, joeRid);
                }
                // Затем удалить запись для Joe из STUDENT.
                studentscan.delete();
                break;
            }
        }

        // Вывести записи для проверки.
        studentscan.beforeFirst();
        while (studentscan.next()) {
            System.out.println(studentscan.getString("sname") + " "
                               + studentscan.getInt("sid"));
        }
        studentscan.close();
        for (Index idx : indexes.values())
            idx.close();
        tx.commit();
    }
}
```

12.3. Статические хеш-индексы

Статическое хеширование – это, пожалуй, самый простой способ реализации индексов. Это не самая эффективная стратегия, зато достаточно простая и понятная для иллюстрации основных принципов. Благодаря этому своему свойству она послужит нам отличным началом.

12.3.1. Статическое хеширование

Статический хеш-индекс использует фиксированное число N *ячеек*, пронумерованных от 0 до N – 1. Индекс также использует *хеш-функцию*, отображающую значения в ячейки. По результатам хеширования поля dataval каждая индексная запись помещается в свою ячейку. Статический хеш-индекс действует следующим образом:

- чтобы сохранить индексную запись, ее нужно поместить в ячейку, вычисленную хеш-функцией;
- чтобы найти индексную запись, нужно вычислить хеш ключа поиска и просмотреть соответствующую ячейку;
- чтобы удалить индексную запись, нужно сначала найти ее (как указано выше), а затем удалить из ячейки.

Стоимость поиска с помощью хеш-индекса обратно пропорциональна количеству ячеек. Если индекс содержит B блоков и N ячеек, то на каждую ячейку будет приходиться около B/N блоков, поэтому для поиска в ячейке потребуется обратиться к B/N блоков.

Например, рассмотрим индекс для поля SName. Предположим для простоты, что N = 3 и хеш-функция отображает строку s в количество букв в этой строке, которые в алфавите предшествуют букве «m», по модулю N[1]. Предположим также, что в один блок умещается три индексные записи. На рис. 12.2 показано содержимое трех ячеек индекса. Обозначение ri на рисунке используется для представления идентификатора *i*-й записи STUDENT.

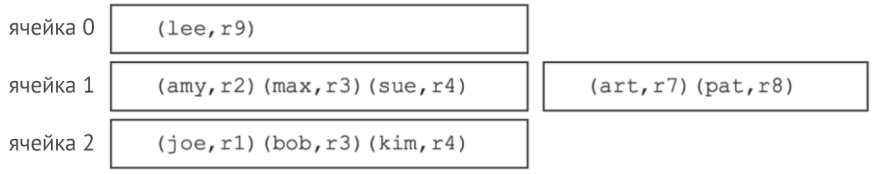


Рис. 12.2. Статический хеш-индекс с тремя ячейками

Предположим, что вам понадобилось найти datarid для всех студентов с именем «sue». Вы хешируете строку «sue», получаете номер ячейки 1 и ищете записи в этой ячейке. Поиск требует двух обращений к блокам. Аналогично, чтобы убедиться в отсутствии учеников с именем «ron», потребуется обратиться всего к одному блоку, потому что имя «ron» хешируется в номер ячейки 0.

В этом примере используются смехотворно маленькие значения для размера блока и количества ячеек. Чтобы получить более реалистичную картину, предположим, что для индекса используется 1024 ячейки, то есть (при условии что записи хешируются равномерно между ячейками):

- для поиска в индексе, занимающем до 1024 блоков, потребуется всего одно обращение к диску;
- для поиска в индексе, занимающем до 2048 блоков, потребуется два обращения к диску;

и так далее. Чтобы понять смысл этих чисел, учтите, что индексная запись для SName занимает 22 байта (14 байт для поля davavar, имеющего тип *varchar(10)*, и 8 байт для поля datarid); то есть если добавить 1 байт на запись для хранения флага заполненности, то в блок с размером 4096 байт поместится 178 индексных записей. Следовательно, индекс, занимающий 2048 блоков, будет соответствовать файлу данных, содержащему около 364 544 записей. Для поиска в таком большом количестве записей достаточно всего двух обращений к диску!

12.3.2. Реализация статического хеширования

Статическое хеширование в SimpleDB реализует класс HashIndex, определение которого показано в листинге 12.4.

[1] Это на удивление плохая хеш-функция, но она делает пример интереснее.

Листинг 12.4. Определение класса HashIndex в SimpleDB

```
public class HashIndex implements Index {
    public static int NUM_BUCKETS = 100; // количество ячеек
    private Transaction tx;
    private String idxname;
    private Layout layout;
    private Constant searchkey = null;
    private TableScan ts = null;

    public HashIndex(Transaction tx, String idxname, Layout layout) {
        this.tx = tx;
        this.idxname = idxname;
        this.layout = layout;
    }

    public void beforeFirst(Constant searchkey) {
        close();
        this.searchkey = searchkey;
        int bucket = searchkey.hashCode() % NUM_BUCKETS;
        String tblname = idxname + bucket;
        ts = new TableScan(tx, tblname, layout);
    }

    public boolean next() {
        while (ts.next())
            if (ts.getVal("dataval").equals(searchkey))
                return true;
        return false;
    }

    public RID getDataRid() {
        int blknum = ts.getInt("block");
        int id = ts.getInt("id");
        return new RID(blknum, id);
    }

    public void insert(Constant val, RID rid) {
        beforeFirst(val);
        ts.insert();
        ts.setInt("block", rid.blockNumber());
        ts.setInt("id", rid.slot());
        ts.setVal("dataval", val);
    }

    public void delete(Constant val, RID rid) {
        beforeFirst(val);
        while(next())
            if (getDataRid().equals(rid)) {
                ts.delete();
                return;
            }
    }

    public void close() {
        if (ts != null)
            ts.close();
    }
```

```
    public static int searchCost(int numblocks, int rpb) {
        return numblocks / HashIndex.NUM_BUCKETS;
    }
}
```

Этот класс сохраняет каждую ячейку в отдельной таблице с именем, состоящим из имени индекса и номера ячейки. Например, таблица для ячейки № 35 индекса SID_INDEX получает имя «SID_INDEX35». Метод `beforeFirst` хеширует ключ поиска и открывает образ сканирования таблицы для полученной ячейки. Метод `next` начинает с текущей позиции в образе сканирования и читает записи, пока не найдет соответствующую ключу поиска; если такая запись не будет найдена, он вернет `false`. Значение идентификатора записи данных (datarid) хранится в индексной записи в виде двух целых чисел в полях `block` и `id`. Метод `getDataRid` читает этих два значения из текущей индексной записи и конструирует объект rid; метод `insert` выполняет противоположную операцию.

Помимо методов интерфейса `Index`, класс `HashIndex` реализует также статический метод `searchCost`. Этот метод вызывается методом `IndexInfo.blocksAccessed`, как было показано в листинге 7.13. Объект `IndexInfo` передает в вызов метода `searchCost` два аргумента: количество блоков в индексе и количество индексных записей в блоке. Это решение объясняется тем, что он не знает, как индексы вычисляют стоимость своего использования. В случае статической индексации стоимость поиска зависит только от размера индекса, поэтому второй аргумент игнорируется.

12.4. Расширяемое хеширование

Стоимость поиска при использовании индексов на основе статического хеширования обратно пропорциональна количеству ячеек – чем больше ячеек используется, тем меньше блоков в каждой из них. Наиболее оптимальной считается ситуация, когда количество ячеек настолько велико, что на каждую приходится ровно один блок.

Если бы размер индекса никогда не изменялся, то было бы легко рассчитать это идеальное количество ячеек. Но на практике индексы растут по мере добавления новых записей в базу данных. Так как же выбрать правильное количество ячеек? Если исходить из текущего размера индекса, то впоследствии, при его увеличении, каждая ячейка будет содержать несколько блоков. А если выбрать большее количество ячеек, ориентируясь на потребности в будущем, то пустые и почти пустые в данный момент ячейки будут напрасно расходовать значительный объем дискового пространства, пока индекс не вырастет и не заполнит их.

Эту проблему решает стратегия, известная как *расширяемое хеширование*. Суть ее заключается в использовании достаточно большого количества ячеек, чтобы гарантировать, что каждая ячейка никогда не будет содержать более одного блока[1]. Проблема неиспользуемого пространства решается в расширя-

[1] Исключение должно быть сделано для случая, когда имеется слишком много записей с одинаковым значением dataval. Поскольку эти записи всегда будут хешироваться в одну и ту же ячейку, стратегия хеширования никак не сможет распределить их по

емом хешировании за счет возможности совместного использования одного и того же блока несколькими ячейками. Идея состоит в том, чтобы позволить большому количеству ячеек совместно использовать меньшее количество блоков и тем самым не допустить напрасного расходования дискового пространства. Это очень удачное решение.

Совместное использование блоков ячейками обеспечивается с помощью двух файлов: *файла ячеек* и *каталога ячеек*. Файл ячеек содержит блоки индекса. Каталог ячеек отображает ячейки в блоки. Каталог можно рассматривать как массив целых чисел, по одному для каждой ячейки. Назовем этот массив `Dir`. Тогда если индексная запись хешируется в ячейку `b`, то запись будет сохранена в блоке `Dir[b]` файла ячеек.

Например, на рис. 12.3 показано возможное содержимое расширяемого хеш-индекса для поля `SId` таблицы STUDENT, при условии (для удобства чтения) что:

- в блок умещаются три индексные записи;
- используется восемь ячеек;
- хеш-функция имеет вид h(x) = x mod 8;
- таблица STUDENT содержит семь записей с идентификаторами 1, 2, 4, 5, 7, 8 и 12.

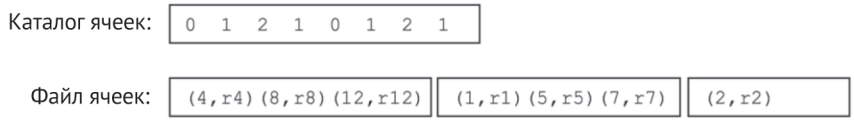


Рис. 12.3. Расширяемый хеш-индекс для поля SId таблицы STUDENT

Как и прежде, `ri` обозначает `rid` – идентификатор *i*-й записи в таблице STUDENT.

Обратите внимание, как используется каталог ячеек `Dir`. Тот факт, что `Dir[0] = 0` и `Dir[4] = 0`, означает, что если запись хешируется в число 0 (например, r8) или 4 (например, r4 и r12), она будет помещена в блок 0. Аналогично, записи, которые хешируются в число 1, 3, 5 или 7, будут помещены в блок 1, а записи, хеш которых равен 2 или 6, будут помещены в блок 2. То есть этот каталог ячеек позволяет хранить индексные записи в трех блоках вместо восьми.

Конечно, существует много других способов организации каталога ячеек для совместного использования трех блоков всеми ячейками. Особенности логики, на которой основан каталог на рис. 12.3, обсуждаются далее.

12.4.1. Совместное использование индексных блоков

Каталоги, используемые в стратегии расширяемого хеширования, всегда имеют 2^M ячеек, где целое число `M` называется *максимальной глубиной* индекса. Каталог, содержащий 2^M ячеек, может поддерживать хеш-значения длиной `M` бит. В примере на рис. 12.3 используется `M` = 3. На практике разумным выбором считается `M` = 32, потому что целочисленные значения имеют размер 32 бита.

нескольким ячейкам. В этом случае ячейка будет содержать столько блоков, сколько потребуется для хранения этих записей.

Первоначально пустой файл ячеек содержит один блок, и все элементы каталога будут ссылаться на этот блок. Другими словами, этот блок является общим для всех ячеек. Любая новая индексная запись будет вставлена в этот блок.

Каждый блок в файле ячеек имеет *локальную глубину*. Локальная глубина `L` – это количество крайних правых битов хеш-значения, одинаковых для всех записей в блоке. Первый блок в файле изначально имеет локальную глубину 0, потому что записи в нем могут иметь любые значения хеша.

Предположим, что новая вставляемая индексная запись не помещается в назначенный ей блок. Тогда этот блок *расщепляется*, то есть в файл ячеек добавляется еще один блок, и записи из заполненного блока перераспределяются между этим и новым блоками. Алгоритм перераспределения основан на локальной глубине блока. Поскольку в настоящий момент все записи в блоке имеют значения хеша с `L` одинаковыми правыми битами, алгоритм выбирает (`L` + 1)-й бит справа, и все записи со значением 0 в этом бите сохраняются в текущем блоке, а записи со значением 1 переносятся в новый блок. Обратите внимание, что после этого записи в каждом из этих двух блоков будут иметь `L` + 1 одинаковых правых битов. То есть после расщепления локальная глубина каждого блока увеличивается на 1.

После расщепления блока необходимо скорректировать каталог. Пусть `b` будет значением хеша вновь вставленной индексной записи, то есть `b` является номером ячейки. Предположим, что правые `L` бит в числе `b` имеют значения $b_L \ldots b_2 b_1$. Тогда можно показать (см. упражнение 12.10), что номера ячеек (включая `b`), имеющие те же самые правые `L` бит, ссылаются на только что расщепленный блок. То есть каталог следует модифицировать так, чтобы каждый элемент, имеющий в правых `L` + 1 битах значения $1b_L \ldots b_2 b_1$, ссылался на новый блок.

Например, предположим, что ячейка 17 в настоящее время отображается в блок `B`, имеющий локальную глубину 2. Поскольку число 17 имеет двоичное представление `1001`, правые 2 бита в нем равны `01`. Из этого следует, что все ячейки, номера которых имеют правые два бита `01`, отображаются в `B`. К ним относятся, например, ячейки с номерами 1, 5, 9, 13, 17 и 21. Теперь предположим, что блок `B` заполнен и его следует расщепить. Система выделяет новый блок `B'` и для обоих блоков, `B` и `B'`, устанавливает локальную глубину 3, а затем корректирует каталог ячеек. Те ячейки, в номерах которых правые 3 бита равны `001`, продолжают отображаться в блок `B` (то есть их элементы в каталоге остаются неизменными). А отображение ячеек, в номерах которых правые 3 бита равны `101`, изменяется на `B'`. То есть ячейки 1, 9, 17, 25 и т. д. будут продолжать отображаться в `B`, тогда как ячейки 5, 13, 21, 29 и т. д. теперь будут отображаться в `B'`.

В алгоритме 12.1 представлена последовательность действий при вставке записи в расширяемый хеш-индекс. Для примера снова рассмотрим расширяемый хеш-индекс для поля Sid. Предположим, что каталог ячеек содержит 2^{10} ячеек (то есть максимальная глубина равна 10) и хеш-функция отображает каждое целое число `n` в `n%1024`. Первоначально файл ячеек состоит из одного блока, и все элементы каталога ссылаются на этот блок. Эта ситуация изображена на рис. 12.9а.

Алгоритм 12.1. Вставка записи в расширяемый хеш-индекс

1. Вычислить хеш dataval, чтобы получить номер ячейки `b`.
2. Найти `B = Dir[b]`. Пусть `L` – локальная глубина блока `B`.
   Если запись:

3a) умещается в блок `B`, вставить ее и вернуть управление;

3b) не умещается в блок `B`, то:

- выделить новый блок `B'` в файле ячеек;
- установить локальную глубину обоих блоков, `B` и `B'`, равной `L+1`;
- скорректировать каталог ячеек так, чтобы все ячейки с номерами, в которых правые `L+1` бит имеют значения $1b_L \ldots b_2b_1$, ссылались на `B'`;
- повторно вставить в индекс все записи из `B` (эти записи будут хешированы либо в блок `B`, либо в блок `B'`);
- повторить попытку вставки новой записи в индекс.

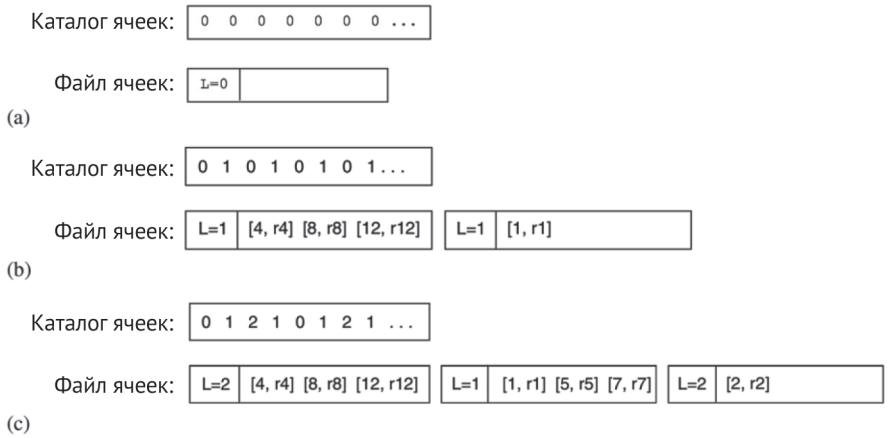


Рис. 12.4. Вставка записей в расширяемый хеш-индекс: (a) индекс содержит один блок; (b) после первого расщепления; (c) после второго расщепления

Предположим теперь, что вы вставляете индексные записи для студентов 4, 8, 1 и 12. Первые три попадут в блок 0, а четвертая вызовет расщепление. Это расщепление вызовет следующую последовательность событий: будет выделен новый блок, локальная глубина увеличится с 0 до 1, скорректируются элементы каталога, выполнится повторная вставка в индекс записей из блока 0, и затем будет вставлена запись для студента 12. Результат показан на рис. 12.4b. Обратите внимание, что нечетные элементы в каталоге ячеек теперь ссылаются на новый блок. Индекс сейчас организован так, что все записи с нечетными значениями хеша (то есть те, крайний правый бит в которых равен 0), находятся в блоке 0 файла ячеек, а все записи с нечетным значением (правый бит равен 1) находятся в блоке 1.

Затем вставляются индексные записи для студентов 5, 7 и 2. Первые две помещаются в блок 1, но вставка третьей вызывает повторное расщепление блока 0. Результат показан на рис. 12.4c. Блок 0 файла ячеек теперь содержит все индексные записи, значение хеша которых заканчивается на `00`, а блок 2 содержит все записи, значение хеша которых заканчивается на `10`. Блок 1 по-прежнему содержит записи, значение хеша которых заканчивается на `1`.

Одна из проблем заключается в том, что никакая стратегия хеширования не гарантирует равномерного распределения записей по ячейкам. Когда блок расщепляется, все его записи могут снова оказаться в одном и том же блоке; если и новая запись тоже хешируется в тот же блок, она опять не поместится в нем, и блок придется расщепить еще раз. Если локальная глубина окажется равной максимальной, расщепление станет невозможно, и тогда необходимо будет создать блок переполнения для хранения дополнительных индексных записей.

12.4.2. Компактное хранение каталога ячеек

При использовании стратегии расширяемого хеширования мы все еще должны учитывать размер каталога ячеек. Для хешей с максимальной глубиной 10 необходим каталог, состоящий из 2^{10} ячеек, который может уместиться в один блок, если исходить из предположения, что размер блока равен 4 Кбайт. Однако для хешей с максимальной глубиной 20 необходим каталог, состоящий из 2^{20} ячеек, для которого необходимо 1024 блока, независимо от размера индекса. Вы уже видели, как размер файла ячеек увеличивается пропорционально размеру индекса. В этом разделе вы увидите, что каталог ячеек тоже может изначально иметь небольшой размер и затем увеличиваться по мере необходимости.

Обратите внимание, что элементы каталога ячеек, как показано на рис. 12.4, следуют определенному шаблону. Если блок имеет локальную глубину 1, то все остальные элементы каталога ссылаются на этот блок. Если блок имеет локальную глубину 2, то каждый четвертый элемент ссылается на этот блок. И вообще, если блок имеет локальную глубину `L`, то каждый 2^L-й элемент ссылается на этот блок. В соответствии с этим шаблоном наибольшая локальная глубина определяет «период» каталога. Например, поскольку наибольшая локальная глубина на рис. 12.4c равна 2, содержимое каталога ячеек повторяется через каждые 2^2 записи.

Из-за того, что элементы каталога повторяются, нет необходимости хранить весь каталог ячеек; достаточно хранить только 2^d записей, где `d` – это наибольшая локальная глубина. Мы называем `d` *глобальной глубиной* индекса.

Алгоритм поиска по индексу нужно немного изменить, чтобы приспособить его к изменившейся организации каталога ячеек. В частности, после хеширования ключа поиска алгоритм должен использовать только крайние правые `d` бит в значении хеша для выбора соответствующего элемента каталога ячеек.

Алгоритм вставки новой индексной записи тоже следует изменить. Как и при поиске, он должен вычислить хеш значения dataval записи и с помощью крайних правых `d` бит хеша выбрать элемент каталога, чтобы определить, куда вставлять индексную запись. Если возникнет необходимость расщепления блока, то алгоритм должен действовать как обычно. Единственное исключение – когда из-за расщепления локальная глубина блока становится больше текущей глобальной глубины индекса. В этом случае глобальную глубину следует увеличить до повторного хеширования записей.

Увеличение глобальной глубины означает удвоение размера каталога ячеек, которое реализуется на удивление просто: поскольку элементы каталога повторяются, достаточно скопировать первую половину каталога во вторую. После удвоения можно продолжить процесс расщепления. Для иллюстрации

алгоритма вернемся к рис. 12.4. В начальный момент индекс будет иметь глобальную глубину 0, то есть в каталоге ячеек будет находиться единственный элемент, ссылающийся на блок 0. После вставки записей для студентов с идентификаторами 4, 8 и 1 глобальная глубина останется равной 0.

Поскольку глобальная глубина равна 0, для выбора элемента каталога используются только крайние правые 0 бит в значении хеша; другими словами, независимо от значения хеша всегда будет выбираться элемент 0. Однако после вставки записи для студента с идентификатором 12 расщепление вызовет увеличение локальной глубины блока 0, соответственно, увеличится глобальная глубина индекса, и размер каталога ячеек удвоится с одного до двух элементов. Первоначально оба элемента ссылаются на блок 0; затем все элементы с правым крайним битом, равным 1, корректируются и начинают ссылаться на новый блок. Получившийся в результате каталог имеет глобальную глубину 1 и элементы `Dir[0] = 0` и `Dir[1] = 1`.

Теперь, когда глобальная глубина равна 1, для вставки записей с идентификаторами студентов 5 и 7 анализируется один крайний правый бит хеша, который в обоих случаях имеет значение 1, поэтому выбирается ячейка `Dir[1]` и обе записи вставляются в блок 1. Расщепление, которое происходит после вставки записи с идентификатором 2, увеличивает до 2 локальную глубину блока 0, соответственно, увеличивается и глобальная глубина. Удвоение каталога увеличивает его размер до четырех элементов, которые сразу после этого получают значения 0, 1, 0, 1. Затем элементы с крайними правыми битами `10` корректируются так, чтобы они ссылались на новый блок, и каталог приобретает вид: 0 1 2 1.

Расширяемое хеширование не пригодно для случаев, когда индекс содержит больше записей с одним и тем же значением dataval, чем может уместиться в блок. В таких случаях не поможет никакое расщепление, и каталог ячеек вырастет до своего максимального размера, даже если в индексе будет относительно мало записей. Чтобы избежать этой проблемы, нужно модифицировать алгоритм вставки, добавив проверку этой ситуации и создавая цепочки блоков переполнения для данной ячейки без расщепления.

12.5. Индексы на основе B-дерева

Предыдущие две стратегии индексирования были основаны на хешировании. Теперь рассмотрим подход на основе сортировки. Основная идея заключается в сортировке индексных записей по значениям dataval.

12.5.1. Как усовершенствовать словарь

Если подумать, то отсортированный индексный файл очень похож на словарь. Индексный файл – это последовательность индексных записей с полями dataval и datarid. Словарь – это последовательность элементов, каждый из которых содержит слово и определение. При работе со словарем для нас важно иметь возможность как можно быстрее находить определения слов. При работе с индексным файлом для нас важно иметь возможность как можно быстрее находить идентификаторы записей (datarid) по значениям индексированного поля (dataval). Это соответствие показано в табл. 12.1.

Таблица 12.1. Соответствие между словарем и отсортированным индексным файлом

	Словарь	Отсортированный индексный файл
Элемент:	[слово, определение]. Слово может иметь несколько определений	[dataval, datarid]. Значению dataval может соответствовать несколько идентификаторов записей datarid
Используется для:	Поиск определений по словам	Поиск datarid по dataval

Близкое сходство словарей и отсортированных индексов подсказывает, что приемы реализации словарей можно применить для реализации отсортированного индекса. Посмотрим, так ли это.

Словарь, лежащий на моем столе, имеет около 1000 страниц. На каждой странице имеется заголовок, в котором указаны первое и последнее слова на этой странице. Когда я ищу слово, заголовок помогает мне выбрать правильную страницу – мне достаточно просматривать только заголовки, игнорируя содержимое страниц. Отыскав правильную страницу, я ищу на ней нужное мне слово.

В словаре также имеется оглавление, в котором перечислены начальные буквы и соответствующие им страницы. Однако я никогда не использую оглавление, потому что информация в нем не особенно полезна. Я бы предпочел, чтобы содержание включало заголовки всех страниц, как показано на рис. 12.5a. Это оглавление намного лучше, поскольку мне больше не нужно пролистывать страницы; вся информация из заголовков находится в одном месте.

Рис. 12.5. Усовершенствованное оглавление словаря: (a) одна строка соответствует одной странице; (b) одна строка соответствует одной странице в оглавлении

1000-страничный словарь будет иметь 1000 заголовков. Если предположить, что на одной странице оглавления поместится 100 заголовков, оглавление займет 10 страниц. Поиск по 10 страницам выполняется намного быстрее, чем по 1000, но все еще требует слишком много работы. Хотелось бы иметь еще и содержание оглавления, как на рис. 12.5b, помогающее найти нужную страницу в оглавлении. «Содержание оглавления» описывает, какие заголовки содержит каждая страница в оглавлении. Таким образом, содержание оглавления с десятью строками легко поместится на одной странице.

Благодаря такой организации я смогу быстро найти любое слово в своем словаре, просмотрев ровно три страницы:

- на странице содержания оглавления я найду нужную страницу оглавления;
- на странице оглавления я узнаю номер страницы словаря с определением искомого слова;
- на странице словаря я найду определение требуемого слова.

Если распространить эту стратегию на очень большой словарь (скажем, насчитывающий более 10 000 страниц), то его оглавление займет более 100 страниц, а содержание оглавления – более 1 страницы.

В этом случае можно было бы создать страницу «содержание содержания», которая избавила бы меня от необходимости длительного поиска в содержании оглавления. Тогда чтобы найти слово, потребуется просмотреть четыре страницы.

Взглянув на рис. 12.5, нетрудно заметить, что оглавление и его содержание имеют совершенно одинаковую структуру. Назовем эти страницы *каталогом* словаря. Оглавление – это каталог уровня 0, содержание оглавления – каталог уровня 1, содержание содержания – каталог уровня 2 и т. д.

Этот усовершенствованный словарь имеет следующую структуру:

- большое количество страниц с определениями слов, следующих в алфавитном порядке;
- каждая страница каталога уровня 0 содержит заголовки нескольких страниц с определениями слов;
- каждая страница каталога уровня (N + 1) содержит заголовки нескольких страниц каталога уровня N;
- на самом верхнем уровне находится единственная страница каталога.

Эту структуру можно изобразить в виде дерева страниц, в котором страница каталога самого верхнего уровня является корнем, а страницы с определениями слов – листьями. Пример такого дерева изображен на рис. 12.6.

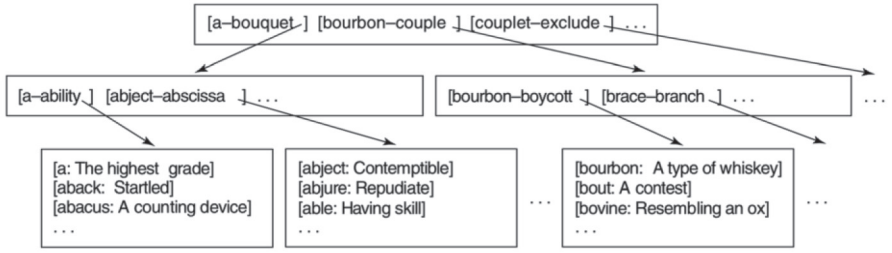


Рис. 12.6. Представление усовершенствованного словаря в виде дерева

12.5.2. Каталог на основе B-дерева

Идею организации каталога в виде дерева можно также применить к отсортированным индексам. Индексные записи будут храниться в индексном файле. Каталог уровня 0 будет хранить записи, ссылающиеся на блоки индексного файла. Эти каталожные записи будут иметь форму [dataval, block#], где *dataval* – это dataval первой индексной записи в блоке, а *block#* – номер блока.

Например, на рис. 12.7a изображен индексный файл для отсортированного индекса по полю `SName` таблицы STUDENT. Этот файл состоит из трех блоков, в каждом из которых содержится некоторое количество записей. На рис. 12.7b показан каталог уровня 0 для этого индексного файла, состоящий из трех записей, по одной на каждый индексный блок.

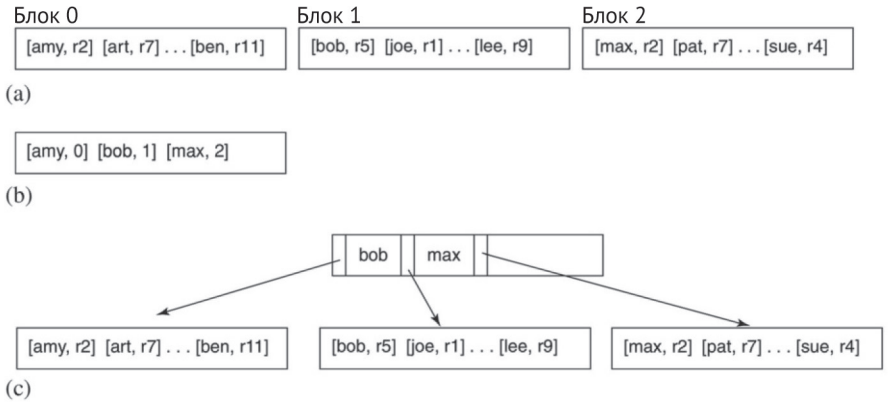


Рис. 12.7. Индекс на основе B-деревьев для поля SName:
(a) отсортированный индексный файл; (b) отсортированный каталог уровня 0; (c) древовидное представление индекса и его каталога

Если записи в каталоге отсортировать по значению dataval, то диапазон значений в каждом индексном блоке можно определить, сравнив смежные элементы каталога. Например, из трех записей в каталоге на рис. 12.7b можно узнать, что:

- блок 0 индексного файла содержит индексные записи со значением dataval от «amy» до (не включая) «bob»;
- блок 1 индексного файла содержит индексные записи от «bob» до (не включая) «max»;
- блок 2 индексного файла содержит индексные записи от «max» до конца.

В общем случае конкретное значение dataval в первой каталожной записи не представляет большого интереса и обычно заменяется специальным значением (например, `null`), обозначающим «самое начало».

Каталог и его индексные блоки обычно представляют графически в виде дерева, как показано на рис. 12.7c. Это дерево является примером *сбалансированного*, или *B-дерева*[1]. Обратите внимание, что получить фактические каталожные записи можно, объединив стрелку со значением dataval, которое ей предшествует. Значение dataval перед самой левой стрелкой в дереве опущено, потому что оно не нужно.

Этот каталог можно использовать для поиска индексных записей, соответствующих, например, значению `v` в поле dataval, или для вставки новой индексной записи с этим dataval. Порядок выполнения данных операций описывается в алгоритмах 12.2 и 12.3.

---

[1] В свое время были разработаны две немного отличающиеся версии B-деревьев. Версия, которую мы используем, на самом деле называется *B+-дерево*, потому что была разработана позже, а *B-деревом* правильнее называть первую версию, которую я не буду рассматривать. Однако, поскольку вторая версия используется на практике гораздо чаще, я буду использовать более простой (хотя и немного неточный) термин для ее обозначения.

Алгоритм 12.2. Поиск индексной записи со значением v в поле dataval в дереве на рис. 12.7

1. Найти в блоках каталога запись с диапазоном значений dataval, в который попадает значение v.
2. Прочитать индексный блок, на который ссылается каталожная запись.
3. Исследовать содержимое этого блока, чтобы найти требуемые индексные записи.

Алгоритм 12.3. Вставка индексной записи со значением v в поле dataval в дерево на рис. 12.7

1. Найти в блоках каталога запись с диапазоном значений dataval, в который попадает значение v.
2. Прочитать индексный блок, на который ссылается каталожная запись.
3. Вставить новую индексную запись в этот блок.

Отмечу два важных момента относительно этих алгоритмов. Во-первых, шаги 1 и 2 в них идентичны. Иначе говоря, алгоритм вставки выбирает для новой индексной записи тот же блок, который выберет алгоритм поиска, что, конечно же, вполне очевидно. Во-вторых, каждый алгоритм идентифицирует отдельный индексный блок, к которому относятся интересующие записи; то есть все индексные записи с одинаковым значением dataval должны находиться в одном и том же блоке.

B-дерево, изображенное на рис. 12.7, очень простое, потому что индекс мал. Но с увеличением индекса алгоритм неизбежно столкнется со следующими тремя сложностями:

- для каталога может потребоваться несколько блоков;
- новая индексная запись может не уместиться в соответствующий ей блок;
- в индексе может оказаться очень много записей с одинаковым значением dataval.

Решения этих проблем описываются в следующих подразделах.

12.5.3. Дерево каталога

Продолжая пример на рис. 12.7, допустим, что в базу данных были добавлены сведения о большом количестве новых студентов, поэтому теперь индексный файл занимает восемь блоков. Если предположить (в качестве примера), что в блок помещается не более трех каталожных записей, то для каталога на основе B-дерева потребуется как минимум три блока. Эти блоки можно поместить в файл и сканировать их последовательно; однако подобное решение не очень эффективно. Более удачная идея – поступить так, как в примере с усовершенствованным словарем: добавить в B-дерево «содержание» каталога уровня 0.

То есть теперь каталог будет состоять из блоков двух уровней. Блоки уровня 0 будут ссылаться на индексные блоки, а блоки уровня 1 – на блоки уровня 0. Наглядно B-дерево можно изобразить, как показано на рис. 12.8. Поиск по индексу начинается с уровня 1. Предположим, например, что выполняется поиск по ключу «jim». Этот ключ поиска находится между «eli» и «lee», поэтому мы следуем за средней стрелкой и находим блок уровня 0, содержащий «joe». Ключ поиска меньше строки «joe», поэтому мы следуем по стрелке слева и просматриваем индексный блок, содержащий «eli». Все индексные записи для «jim» (если они есть) будут находиться в этом блоке.

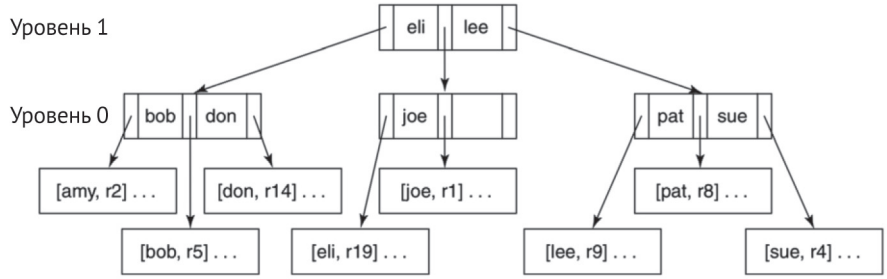


Рис. 12.8. B-дерево с двухуровневым каталогом

В общем случае, когда некоторый уровень каталога содержит несколько блоков, должен иметься более высокий уровень, блоки которого будут ссылаться на блоки этого уровня. В конечном итоге самый верхний уровень будет содержать единственный блок. Этот блок называется *корнем* B-дерева.

Теперь приостановитесь ненадолго и проверьте себя – сможете ли вы самостоятельно пройти через B-дерево. Используя рис. 12.8, выберите несколько имен и проверьте, сумеете ли вы найти соответствующие им индексные блоки. Не должно быть никакой неопределенности – для каждого значения dataval должен существовать ровно один индексный блок, содержащий индексные записи с этим значением dataval.

Также обратите внимание на распределение имен в каталожных записях в B-дереве. Например, значение «eli» в узле первого уровня означает, что «eli» – это первое имя в поддереве, на которое указывает средняя стрелка, то есть это первая запись в первом индексном блоке, на который указывает блок каталога уровня 0. И хотя значение «eli» явно не указано в блоке уровня 0, оно присутствует в блоке уровня 1. Фактически первое значение dataval в каждом индексном блоке (кроме самого первого блока) появляется ровно один раз в некотором блоке каталога на некотором уровне B-дерева.

Поиск в B-дереве требует обращения к одному блоку каталога на каждом уровне плюс к одному индексному блоку. Таким образом, стоимость поиска равна числу уровней каталога плюс 1. Чтобы увидеть практическое влияние этой формулы, вернемся к примеру в конце раздела 12.3.1, где вычисляется стоимость поиска в статическом хеш-индексе для поля `SName` с использованием четырехкилобайтных блоков. Как и прежде, каждая индексная запись занимает 22 байта и в блок помещается 178 индексных записей. Каждая каталожная запись занимает 18 байт (14 байт для dataval и 4 байта для номера блока), соответственно, в блок поместится 227 каталожных записей. Таким образом:

- одноуровневое B-дерево, поиск в котором выполняется за два обращения к диску, может содержать до 227 × 178 = 40 406 индексных записей;
- двухуровневое B-дерево, поиск в котором выполняется за три обращения к диску, может содержать до 227 × 227 × 178 = 9 172 162 индексных записей;
- трехуровневое B-дерево, поиск в котором выполняется за четыре обращения к диску, может содержать до 227 × 227 × 227 × 178 = 2 082 080 774 индексных записей.

Иными словами, организация индексов в виде B-дерева исключительно эффективна. Любую запись с данными можно получить не более чем за пять обращений к диску, за исключением совсем уж огромных таблиц[1]. Если коммерческая система баз данных реализует только одну стратегию индексирования, она почти наверняка использует B-дерево.

12.5.4. Вставка записей

Алгоритм 12.3 вставки новой индексной записи подразумевает, что существует ровно один индексный блок, куда ее можно вставить. Но как быть, если в этом блоке больше нет места? Как и в стратегии расширяемого хеширования, решение состоит в том, чтобы расщепить блок. Расщепление индексного блока влечет за собой следующие действия:

- добавление нового блока в файл индекса;
- перемещение половины записей с бóльшими значениями dataval в этот новый блок;
- создание каталожной записи для нового блока;
- вставка новой каталожной записи в тот же блок каталога уровня 0, который указывал на исходный индексный блок.

Например, допустим, что все индексные блоки на рис. 12.8 заполнены. Чтобы вставить новую индексную запись (hal, r55), алгоритм спускается по B-дереву каталога и определяет, что запись следует вставить в индексный блок, который содержит «eli». Затем он расщепляет этот блок, перемещая половину записей с наибольшими значениями dataval в новый блок. Если предположить, что новый блок – это блок 8 в индексном файле, и первой в нем следует запись (jim, r48), то в блок уровня 0 будет вставлена каталожная запись (jim, 8). Получившееся поддерево показано на рис. 12.9.

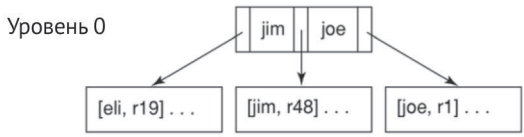


Рис. 12.9. Результат расщепления индексного блока

В данном случае в блоке уровня 0 нашлось место для новой каталожной записи. Если бы блок оказался заполнен, его также пришлось бы расщепить. Например, вернемся к рис. 12.8 и предположим, что нужно вставить индексную запись (zoe, r56). Вставка этой записи приведет к расщеплению правого крайнего индексного блока – пусть новый блок имеет номер 9, а первая запись в нем имеет значение «tom» в поле dataval. Тогда запись (tom, 9) должна быть вставлена в правый крайний блок каталога уровня 0. Однако в этом блоке нет места, поэтому он тоже расщепляется. Две первые каталожные записи остаются в исходном блоке, а две последние перемещаются в новый блок (например, в блок 4

[1] А если учесть буферизацию, то ситуация выглядит еще лучше. Если индекс используется активно, то корневой блок и многие блоки нижних уровней часто уже будут находиться в буферах, благодаря чему потребуется еще меньше обращений к диску.

файла каталога). Получившиеся в результате блоки каталога и индекса показаны на рис. 12.10. Обратите внимание, что каталожная запись для «sue» все еще существует, но не показана на рисунке, потому что это первая запись в блоке.

Но это еще не все. Для нового блока уровня 0 необходимо вставить запись в блок каталога уровня 1, поэтому процесс вставки записи продолжается рекурсивно. На этот раз будет вставлена новая каталожная запись (sue, 4). Значение «sue» использовано потому, что это наименьшее значение dataval в поддереве нового блока каталога. Рекурсивная вставка каталожных записей продолжается вверх по B-дереву. Если возникает необходимость расщепить корневой блок, то создается новый корневой блок и B-дерево получает дополнительный уровень. Именно это происходит на рис. 12.10. В блоке уровня 1 не оказалось свободного места, поэтому он тоже расщепляется. В результате создается новый блок уровня 1 и новый блок уровня 2, который становится корневым. Получившееся в итоге B-дерево показано на рис. 12.11.

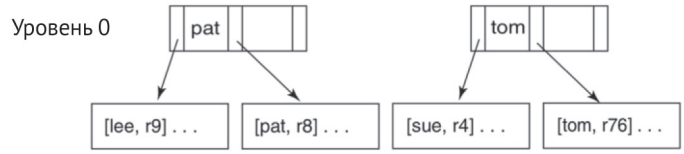


Рис. 12.10. Расщепление блока каталога

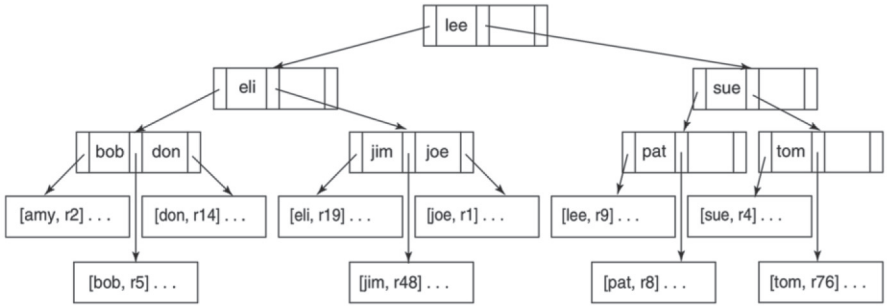


Рис. 12.11. Расщепление корневого блока B-дерева

Обратите внимание, что расщепление блока превращает полный блок в два полупустых блока. По этой причине заполненность B-дерева варьируется в диапазоне от 50 до 100 %.

12.5.5. Одинаковые значения dataval

Пример в разделе 12.1 показал, что индекс полезен, только когда он избирательный. То есть даже притом что в индексе может быть любое количество записей с одним и тем же значением dataval, на практике их будет не так много и, скорее всего, не настолько много, чтобы для них потребовалось несколько блоков. Тем не менее B-дерево должно уметь обрабатывать такие случаи.

Чтобы понять, в чем заключается проблема, предположим, что в дереве на рис. 12.11 присутствует несколько записей со значением «ron» в dataval. Обратите внимание, что все эти записи должны находиться в одном листо-

вом блоке B-дерева, а именно в блоке, содержащем «pat». Содержимое этого блока показано на рис. 12.12a. Предположим, что мы вставляем запись со значением «peg» в dataval, и эта операция вызывает расщепление блока. На рис. 12.12b показан результат расщепления блока ровно пополам: записи со значением «ron» оказываются в разных блоках.

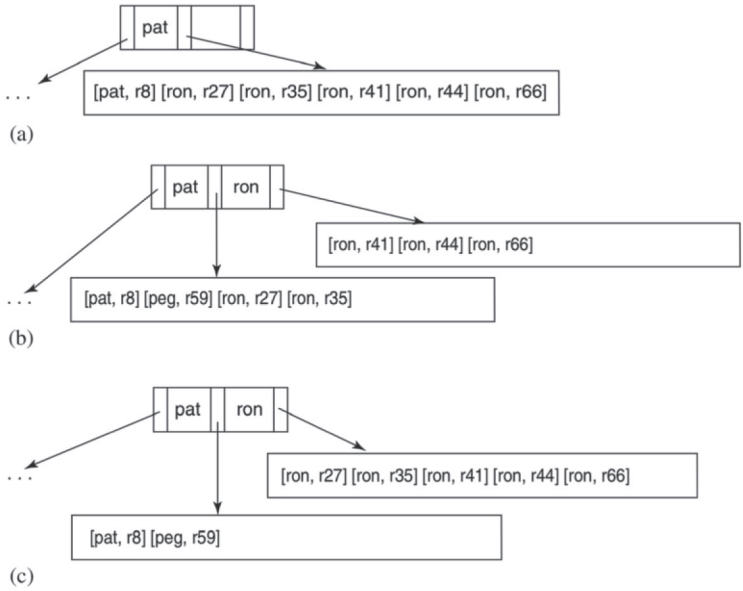


Рис. 12.12. Расщепление листового блока, имеющего записи с одинаковыми значениями: (a) исходный листовой блок и его родитель; (b) неправильный способ расщепления блока; (c) правильный способ расщепления блока

B-дерево на рис. 12.12b явно недопустимо, потому что записи со значением «ron», оставшиеся в блоке «pat», окажутся недоступными. У нас есть следующее правило: *при расщеплении блока все записи с одинаковым значением dataval должны помещаться в один и тот же блок*. Это правило достаточно очевидно. Когда для поиска индексных записей используется каталог в виде B-дерева, каталог всегда будет указывать на один листовой блок. Если в других блоках тоже окажутся индексные записи с заданным ключом, они никогда не будут найдены.

Следствием этого правила является невозможность расщепления индексного блока на равные половины. На рис. 12.12c показан единственный разумный вариант расщепления – поместить пять записей «ron» в новый блок.

Индексный блок всегда можно расщепить, если в нем находятся записи с как минимум двумя разными значениями dataval. Но когда все записи в блоке имеют одинаковое значение dataval, расщепление выполнить не удастся. Лучший выход из такой ситуации – использовать блок переполнения.

Например, вернемся вновь к рис. 12.12c и вставим записи для еще нескольких студентов с именем «ron». Теперь вместо расщепления блока мы должны создать новый листовой блок и переместить в него все записи «ron», кроме одной. Этот новый блок является блоком переполнения. Старый блок связан с блоком переполнения, как показано на рис. 12.13.

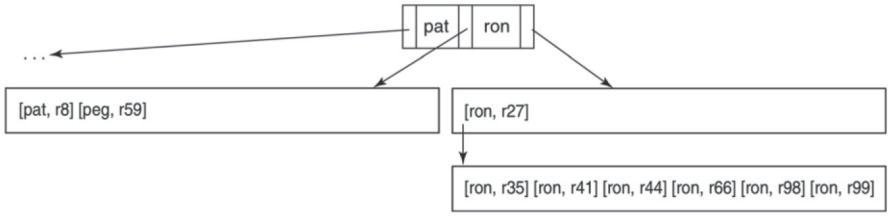


Рис. 12.13. Использование цепочки блоков переполнения для хранения записей с одинаковыми значениями dataval

Обратите внимание, что старый блок почти полностью опустел, что позволяет вставлять в него новые записи для студентов с именем «ron» (если таковые появятся). Когда блок снова заполнится, у нас будет два пути:

- если в блоке будут находиться записи с как минимум двумя разными значениями dataval, то мы расщепим его;
- если в блоке будут находиться только записи «ron», то мы создадим еще один блок переполнения и свяжем его с существующим.

В общем случае листовой блок может содержать неограниченную цепочку блоков переполнения. При этом все блоки переполнения будут полностью заполнены. Записи в цепочке переполнения всегда будут иметь одно и то же значение dataval, совпадающее со значением dataval в первой записи в основном блоке.

Предположим, нам нужно найти индексные записи, имеющие определенный ключ поиска. Следуя по ссылкам в каталоге B-дерева, мы добираемся до конкретного листового блока. Если ключ поиска не является первым ключом в блоке, то проверяем остальные записи в этом блоке, как и раньше. Если ключ поиска соответствует первой записи, то нам также необходимо извлечь записи из цепочки переполнения, если она существует.

Несмотря на то что индексные записи в B-дереве могут иметь одинаковые значения dataval, для каталожных записей это невозможно. Причина в том, что единственный способ добавить новую каталожную запись с конкретным значением dataval – это расщепить листовой блок и взять dataval первой записи из нового блока. Но первый dataval в блоке никогда не расщепится снова – если блок заполнится записями с одинаковым значением dataval, вместо расщепления будет создан блок переполнения.

12.5.6. Реализация страниц B-дерева

Реализация B-деревьев в SimpleDB находится в пакете `simpledb.index.btree`. Этот пакет содержит четыре основных класса: `BTreeIndex`, `BTreeDir`, `BTreeLeaf` и `BTPage`. Классы `BTreeDir` и `BTreeLeaf` реализуют блоки каталога и индекса соответственно[1]. Несмотря на то что блоки каталога и листовые блоки хранят записи разных типов и используются по-разному, они имеют общие черты, такие как поддержка вставки записей в порядке сортировки и расщепления. Код, реализу-

[1] Термин *листовой* (leaf) используется для обозначения индексных блоков, потому что они образуют листья в B-дереве. Реализация в SimpleDB использует суффикс «leaf», чтобы избежать путаницы с классом `BTreeIndex`, который реализует интерфейс `Index`.

ющий эти общие черты, находится в классе BTPage. Класс BTreeIndex реализует фактические операции B-дерева, как определено интерфейсом Index.

Рассмотрим сначала класс BTPage. Записи в странице B-дерева должны соответствовать следующим требованиям:

- записи должны храниться в порядке сортировки;
- записям не требуется иметь постоянного идентификатора, что позволяет перемещать их внутри страницы по мере необходимости;
- страница должна поддерживать возможность расщепления для перемещения части своих записей в другую страницу;
- в каждой странице дополнительно должно храниться целое число, служащее флагом (страница каталога использует флаг для хранения номера своего уровня, а листовая страница – для хранения ссылки на свой блок переполнения).

То есть страницу B-дерева можно представить как отсортированный список записей (в отличие от страницы записей (RecordPage), хранящей несортированный массив записей). Когда в страницу добавляется новая запись, определяется ее местоположение в соответствии с порядком сортировки, и записи, которые должны следовать за ней, сдвигаются на одну позицию вправо, чтобы освободить место. Точно так же, когда запись удаляется, записи, следующие за ней, сдвигаются влево, чтобы заполнить освободившееся место. Для реализации такого поведения в странице также должен храниться целочисленный счетчик записей в странице.

Определение класса BTPage показано в листинге 12.5. Наибольший интерес в этом классе представляет метод findSlotBefore. Он принимает ключ поиска k, отыскивает элемент x, соответствующий условию $k \leq dataval(x)$, и возвращает номер предшествующего ему элемента. Такое поведение реализовано потому, что оно учитывает все способы поиска на страницах. Например, этот метод действует как операция beforeFirst в листовых страницах, поэтому последующий вызов next вернет первую запись, имеющую заданный ключ поиска.

Листинг 12.5. Определение класса BTPage в SimpleDB

```
public class BTPage {
    private Transaction tx;
    private BlockId currentblk;
    private Layout layout;

    public BTPage(Transaction tx, BlockId currentblk, Layout layout) {
        this.tx = tx;
        this.currentblk = currentblk;
        this.layout = layout;
        tx.pin(currentblk);
    }

    public int findSlotBefore(Constant searchkey) {
        int slot = 0;
        while (slot < getNumRecs() &&
               getDataVal(slot).compareTo(searchkey) < 0)
            slot++;
        return slot-1;
    }
```

```
public void close() {
    if (currentblk != null)
        tx.unpin(currentblk);
    currentblk = null;
}

public boolean isFull() {
    return slotpos(getNumRecs()+1) >= tx.blockSize();
}

public BlockId split(int splitpos, int flag) {
    BlockId newblk = appendNew(flag);
    BTPage newpage = new BTPage(tx, newblk, layout);
    transferRecs(splitpos, newpage);
    newpage.setFlag(flag);
    newpage.close();
    return newblk;
}

public Constant getDataVal(int slot) {
    return getVal(slot, "dataval");
}

public int getFlag() {
    return tx.getInt(currentblk, 0);
}

public void setFlag(int val) {
    tx.setInt(currentblk, 0, val, true);
}

public BlockId appendNew(int flag) {
    BlockId blk = tx.append(currentblk.fileName());
    tx.pin(blk);
    format(blk, flag);
    return blk;
}

public void format(BlockId blk, int flag) {
    tx.setInt(blk, 0, flag, false);
    tx.setInt(blk, Integer.BYTES, 0, false); // число записей = 0
    int recsize = layout.slotSize();
    for (int pos = 2*Integer.BYTES; pos+recsize <= tx.blockSize();
                                    pos += recsize)
        makeDefaultRecord(blk, pos);
}

private void makeDefaultRecord(BlockId blk, int pos) {
    for (String fldname : layout.schema().fields()) {
        int offset = layout.offset(fldname);
        if (layout.schema().type(fldname) == INTEGER)
            tx.setInt(blk, pos + offset, 0, false);
        else
            tx.setString(blk, pos + offset, "", false);
    }
}
```

```
// Методы, вызываемые только классом BTreeDir
public int getChildNum(int slot) {
    return getInt(slot, "block");
}

public void insertDir(int slot, Constant val, int blknum) {
    insert(slot);
    setVal(slot, "dataval", val);
    setInt(slot, "block", blknum);
}

// Методы, вызываемые только классом BTreeLeaf
public RID getDataRid(int slot) {
    return new RID(getInt(slot, "block"), getInt(slot, "id"));
}

public void insertLeaf(int slot, Constant val, RID rid) {
    insert(slot);
    setVal(slot, "dataval", val);
    setInt(slot, "block", rid.blockNumber());
    setInt(slot, "id", rid.slot());
}

public void delete(int slot) {
    for (int i=slot+1; i<getNumRecs(); i++)
        copyRecord(i, i-1);
    setNumRecs(getNumRecs()-1);
    return;
}

public int getNumRecs() {
    return tx.getInt(currentblk, Integer.BYTES);
}

// Приватные методы
private int getInt(int slot, String fldname) {
    int pos = fldpos(slot, fldname);
    return tx.getInt(currentblk, pos);
}

private String getString(int slot, String fldname) {
    int pos = fldpos(slot, fldname);
    return tx.getString(currentblk, pos);
}

private Constant getVal(int slot, String fldname) {
    int type = layout.schema().type(fldname);
    if (type == INTEGER)
        return new Constant(getInt(slot, fldname));
    else
        return new Constant(getString(slot, fldname));
}

private void setInt(int slot, String fldname, int val) {
    int pos = fldpos(slot, fldname);
    tx.setInt(currentblk, pos, val, true);
}
```

```
    private void setString(int slot, String fldname, String val) {
        int pos = fldpos(slot, fldname);
        tx.setString(currentblk, pos, val, true);
    }

    private void setVal(int slot, String fldname, Constant val) {
        int type = layout.schema().type(fldname);
        if (type == INTEGER)
            setInt(slot, fldname, val.asInt());
        else
            setString(slot, fldname, val.asString());
    }

    private void setNumRecs(int n) {
        tx.setInt(currentblk, Integer.BYTES, n, true);
    }

    private void insert(int slot) {
        for (int i=getNumRecs(); i>slot; i--)
            copyRecord(i-1, i);
        setNumRecs(getNumRecs()+1);
    }

    private void copyRecord(int from, int to) {
        Schema sch = layout.schema();
        for (String fldname : sch.fields())
            setVal(to, fldname, getVal(from, fldname));
    }

    private void transferRecs(int slot, BTPage dest) {
        int destslot = 0;
        while (slot < getNumRecs()) {
            dest.insert(destslot);
            Schema sch = layout.schema();
            for (String fldname : sch.fields())
                dest.setVal(destslot, fldname, getVal(slot, fldname));
            delete(slot);
            destslot++;
        }
    }

    private int fldpos(int slot, String fldname) {
        int offset = layout.offset(fldname);
        return slotpos(slot) + offset;
    }

    private int slotpos(int slot) {
        int slotsize = layout.slotSize();
        return Integer.BYTES + Integer.BYTES + (slot * slotsize);
    }
}
```

Теперь рассмотрим листовые блоки B-дерева. Определение класса `BTreeLeaf` показано в листинге 12.6.

Листинг 12.6. Определение класса BTreeLeaf в SimpleDB

```
public class BTreeLeaf {
    private Transaction tx;
    private Layout layout;
    private Constant searchkey;
    private BTPage contents;
    private int currentslot;
    private String filename;

    public BTreeLeaf(Transaction tx, BlockId blk, Layout layout,
                     Constant searchkey) {
        this.tx = tx;
        this.layout = layout;
        this.searchkey = searchkey;
        contents = new BTPage(tx, blk, layout);
        currentslot = contents.findSlotBefore(searchkey);
        filename = blk.fileName();
    }

    public void close() {
        contents.close();
    }

    public boolean next() {
        currentslot++;
        if (currentslot >= contents.getNumRecs())
            return tryOverflow();
        else if (contents.getDataVal(currentslot).equals(searchkey))
            return true;
        else
            return tryOverflow();
    }

    public RID getDataRid() {
        return contents.getDataRid(currentslot);
    }

    public void delete(RID datarid) {
        while(next())
            if(getDataRid().equals(datarid)) {
                contents.delete(currentslot);
                return;
            }
    }

    public DirEntry insert(RID datarid) {
        if (contents.getFlag() >= 0 &&
            contents.getDataVal(0).compareTo(searchkey) > 0) {
            Constant firstval = contents.getDataVal(0);
            BlockId newblk = contents.split(0, contents.getFlag());
            contents.setFlag(-1);
            currentslot = 0;
            contents.insertLeaf(currentslot, searchkey, datarid);
            return new DirEntry(firstval, newblk.number());
        }
```

```
        currentslot++;
        contents.insertLeaf(currentslot, searchkey, datarid);
        if (!contents.isFull())
            return null;

        // если страница заполнена, расщепить ее
        Constant firstkey = contents.getDataVal(0);
        Constant lastkey = contents.getDataVal(contents.getNumRecs()-1);
        if (lastkey.equals(firstkey)) {
            // создать блок переполнения для хранения всех записей, кроме первой
            BlockId newblk = contents.split(1, contents.getFlag());
            contents.setFlag(newblk.number());
            return null;
        }
        else {
            int splitpos = contents.getNumRecs() / 2;
            Constant splitkey = contents.getDataVal(splitpos);
            if (splitkey.equals(firstkey)) {
                // двигаться вправо, пока не будет найден другой ключ
                while (contents.getDataVal(splitpos).equals(splitkey))
                    splitpos++;
                splitkey = contents.getDataVal(splitpos);
            }
            else {
                // двигаться влево, до первой записи с этим ключом
                while (contents.getDataVal(splitpos-1).equals(splitkey))
                    splitpos--;
            }
            BlockId newblk = contents.split(splitpos, -1);
            return new DirEntry(splitkey, newblk.number());
        }
    }

    private boolean tryOverflow() {
        Constant firstkey = contents.getDataVal(0);
        int flag = contents.getFlag();
        if (!searchkey.equals(firstkey) || flag < 0)
            return false;
        contents.close();
        BlockId nextblk = new BlockId(filename, flag);
        contents = new BTPage(tx, nextblk, layout);
        currentslot = 0;
        return true;
    }
}
```

Конструктор сначала создает страницу B-дерева для указанного блока, а затем вызывает `findSlotBefore`, чтобы занять позицию перед первой записью, содержащей ключ поиска. Вызов `next` выполняет переход к следующей записи и возвращает `true`, если запись хранит заданный ключ поиска, и `false` в противном случае. Вызов `tryOverflow` выполняется на тот случай, когда листовой блок содержит цепочку блоков переполнения.

Методы delete и insert предполагают, что текущий элемент уже был выбран вызовом findSlotBefore. Метод delete последовательно вызывает next до тех пор, пока не встретит индексную запись с указанным идентификатором (rid), а затем удаляет ее. Метод insert переходит к следующей записи, то есть к первой записи, значение dataval которой больше или равно ключу поиска, и вставляет в это место новую запись. Обратите внимание, что если страница уже содержит записи с заданным ключом поиска, то новая запись будет вставлена в начало списка. Метод insert возвращает объект типа DirEntry (т. е. каталожную запись). Если вставка не приводит к расщеплению блока, возвращается значение null, иначе возвращается запись (dataval, blocknumber), соответствующая новому индексному блоку.

Класс BTreeDir реализует блоки каталога; его определение показано в листинге 12.7.

Листинг 12.7. Определение класса BTreeDir в SimpleDB

```
public class BTreeDir {
    private Transaction tx;
    private Layout layout;
    private BTPage contents;
    private String filename;

    BTreeDir(Transaction tx, BlockId blk, Layout layout) {
        this.tx = tx;
        this.layout = layout;
        contents = new BTPage(tx, blk, layout);
        filename = blk.fileName();
    }

    public void close() {
        contents.close();
    }

    public int search(Constant searchkey) {
        BlockId childblk = findChildBlock(searchkey);
        while (contents.getFlag() > 0) {
            contents.close();
            contents = new BTPage(tx, childblk, layout);
            childblk = findChildBlock(searchkey);
        }
        return childblk.number();
    }

    public void makeNewRoot(DirEntry e) {
        Constant firstval = contents.getDataVal(0);
        int level = contents.getFlag();
        BlockId newblk = contents.split(0, level); // т. е. переместить все записи
        DirEntry oldroot = new DirEntry(firstval, newblk.number());
        insertEntry(oldroot);
        insertEntry(e);
        contents.setFlag(level+1);
    }
```

```
    public DirEntry insert(DirEntry e) {
        if (contents.getFlag() == 0)
            return insertEntry(e);
        BlockId childblk = findChildBlock(e.dataVal());
        BTreeDir child = new BTreeDir(tx, childblk, layout);
        DirEntry myentry = child.insert(e);
        child.close();
        return (myentry != null) ? insertEntry(myentry) : null;
    }

    private DirEntry insertEntry(DirEntry e) {
        int newslot = 1 + contents.findSlotBefore(e.dataVal());
        contents.insertDir(newslot, e.dataVal(), e.blockNumber());
        if (!contents.isFull())
            return null;

        // иначе страница заполнена, расщепить ее
        int level = contents.getFlag();
        int splitpos = contents.getNumRecs() / 2;
        Constant splitval = contents.getDataVal(splitpos);
        BlockId newblk = contents.split(splitpos, level);
        return new DirEntry(splitval, newblk.number());
    }

    private BlockId findChildBlock(Constant searchkey) {
        int slot = contents.findSlotBefore(searchkey);
        if (contents.getDataVal(slot+1).equals(searchkey))
            slot++;
        int blknum = contents.getChildNum(slot);
        return new BlockId(filename, blknum);
    }
}
```

Методы `search` и `insert` начинают поиск с корня и двигаются вниз по дереву до блока каталога уровня 0, связанного с ключом поиска. Метод `search` использует простой цикл while для перемещения вниз по дереву; добравшись до блока уровня 0, он отыскивает в нем нужную страницу и возвращает номер листового блока, содержащего ключ поиска. Метод `insert` использует рекурсию для перемещения вниз по дереву. Возвращаемое значение рекурсивного вызова сообщает, вызвала ли операция вставки расщепление дочерней страницы; если расщепление произошло, то вызывается метод `insertEntry` для вставки в страницу новой каталожной записи. Если эта операция вставки тоже вызвала расщепление страницы, каталожная запись для новой страницы передается родителю страницы. Значение `null` сообщает, что расщепления не произошло.

Метод `makeNewRoot` вызывается, когда вызов метода `insert` корневой страницы возвращает значение, отличное от `null`. Поскольку корень всегда должен находиться в блоке 0 файла каталога, этот метод размещает новый блок, копирует в него содержимое блока 0 и инициализирует блок 0 как новый корень. Новый корень всегда будет иметь две записи: первая будет ссылаться на старый корень, а вторая – на новый блок, созданный в ходе расщепления и переданный методу `makeNewRoot` в виде аргумента.

12.5.7. Реализация индекса на основе B-дерева

Теперь, после знакомства с реализацией страниц B-дерева, пришло время посмотреть, как они используются. Класс `BTreeIndex` реализует методы интерфейса `Index` и координирует использование листовых страниц и страниц каталога; его определение показано в листинге 12.8. Основную работу выполняет конструктор. Он строит компоновку листовых записей на основе переданного ему объекта `Schema`. Затем создает схему каталожных записей, извлекая соответствующую информацию из листовой схемы, и на ее основе создает их компоновку. Наконец, он форматирует корень, если необходимо, вставляя запись, которая ссылается на блок 0 листового файла.

Листинг 12.8. Определение класса BTreeIndex в SimpleDB

```
public class BTreeIndex implements Index {
    private Transaction tx;
    private Layout dirLayout, leafLayout;
    private String leaftbl;
    private BTreeLeaf leaf = null;
    private BlockId rootblk;

    public BTreeIndex(Transaction tx, String idxname, Layout leafLayout) {
        this.tx = tx;

        // подготовка для работы с листьями
        leaftbl = idxname + "leaf";
        this.leafLayout = leafLayout;
        if (tx.size(leaftbl) == 0) {
            BlockId blk = tx.append(leaftbl);
            BTPage node = new BTPage(tx, blk, leafLayout);
            node.format(blk, -1);
        }

        // подготовка для работы с каталогом
        Schema dirsch = new Schema();
        dirsch.add("block", leafLayout.schema());
        dirsch.add("dataval", leafLayout.schema());
        String dirtbl = idxname + "dir";
        dirLayout = new Layout(dirsch);
        rootblk = new BlockId(dirtbl, 0);
        if (tx.size(dirtbl) == 0) {
            // создать новый корневой блок
            tx.append(dirtbl);
            BTPage node = new BTPage(tx, rootblk, dirLayout);
            node.format(rootblk, 0);
            // вставить начальную каталожную запись
            int fldtype = dirsch.type("dataval");
            Constant minval = (fldtype == INTEGER) ?
                    new Constant(Integer.MIN_VALUE) :
                    new Constant("");
            node.insertDir(0, minval, 0);
            node.close();
        }
    }
```

```
    public void beforeFirst(Constant searchkey) {
        close();
        BTreeDir root = new BTreeDir(tx, rootblk, dirLayout);
        int blknum = root.search(searchkey);
        root.close();
        BlockId leafblk = new BlockId(leaftbl, blknum);
        leaf = new BTreeLeaf(tx, leafblk, leafLayout, searchkey);
    }

    public boolean next() {
        return leaf.next();
    }

    public RID getDataRid() {
        return leaf.getDataRid();
    }

    public void insert(Constant dataval, RID datarid) {
        beforeFirst(dataval);
        DirEntry e = leaf.insert(datarid);
        leaf.close();
        if (e == null)
            return;
        BTreeDir root = new BTreeDir(tx, rootblk, dirLayout);
        DirEntry e2 = root.insert(e);
        if (e2 != null)
            root.makeNewRoot(e2);
        root.close();
    }

    public void delete(Constant dataval, RID datarid) {
        beforeFirst(dataval);
        leaf.delete(datarid);
        leaf.close();
    }

    public void close() {
        if (leaf != null)
            leaf.close();
    }

    public static int searchCost(int numblocks, int rpb) {
        return 1 + (int)(Math.log(numblocks) / Math.log(rpb));
    }
}
```

Каждый объект `BTreeIndex` содержит открытый объект `BTreeLeaf`. Этот объект листовой страницы хранит ссылку на текущую индексную запись: она инициализируется вызовом метода `beforeFirst`, увеличивается вызовами `next` и используется методами `getDataRid`, `insert` и `delete`. Метод `beforeFirst` инициализирует объект листовой страницы, вызывая метод `search` корневой страницы каталога. Обратите внимание, что когда листовая страница будет найдена, каталог становится ненужным и его страницы можно закрыть.

Метод `insert` состоит из двух частей. Первая находит подходящую листовую страницу и вставляет в нее индексную запись. Если происходит расщепление

листовой страницы, то метод рекурсивно вставляет в каталог индексную запись для нового листа, начиная с корня. Если вызов `insert` вернул для корня значение, отличное от `null`, это означает, что произошло расщепление корня, и в таком случае вызывается `makeNewRoot`.

12.6. Реализации операторов с поддержкой индексов

В этом разделе рассказывается, как планировщик может использовать индексы для ускорения обработки запросов. Получив SQL-запрос, планировщик должен решить две задачи: сконструировать дерево запроса и выбрать план для каждого оператора в этом дереве. С точки зрения базового планировщика, представленного в главе 10, вторая задача выглядела тривиально, потому что он знал только об одной реализации для каждого оператора. Например, он всегда выполнял оператор селекции (select), используя `SelectPlan`, независимо от наличия подходящего индекса.

Чтобы построить план, использующий индексы, планировщик должен иметь реализации операторов, учитывающие наличие индексов. В этом разделе разрабатываются такие реализации для операторов селекции и соединения (join), которые планировщик сможет включить в свой план.

Процесс планирования существенно усложняется, когда реляционные операторы могут иметь более одной реализации. Планировщик должен проанализировать несколько планов выполнения запроса, часть из которых использует индексы, а часть – нет, и решить, какой план является наиболее эффективным. Эта часть задачи рассматривается в главе 15.

12.6.1. Реализация оператора селекции с поддержкой индексов

Оператор селекции в SimpleDB реализуется классом `IndexSelectPlan`. Его определение показано в листинге 12.9. Конструктор принимает три аргумента: план базовой таблицы, который, как предполагается, имеет тип `TablePlan`; информацию о применяемом индексе; и константу селекции. Метод `open` открывает индекс и передает его (и константу) в вызов конструктора `IndexSelectScan`. Методы `blockAccessed`, `recordsOutput` и `distinctValues` реализуют формулы оценки стоимости, используя методы класса `IndexInfo`.

Листинг 12.9. Определение класса IndexSelectPlan в SimpleDB

```
public class IndexSelectPlan implements Plan {
    private Plan p;
    private IndexInfo ii;
    private Constant val;

    public IndexSelectPlan(Plan p, IndexInfo ii, Constant val) {
        this.p = p;
        this.ii = ii;
        this.val = val;
    }
```

```
    public Scan open() {
        // Сгенерирует исключение, если p не является табличным планом.
        TableScan ts = (TableScan) p.open();
        Index idx = ii.open();
        return new IndexSelectScan(idx, val, ts);
    }

    public int blocksAccessed() {
        return ii.blocksAccessed() + recordsOutput();
    }

    public int recordsOutput() {
        return ii.recordsOutput();
    }

    public int distinctValues(String fldname) {
        return ii.distinctValues(fldname);
    }

    public Schema schema() {
        return p.schema();
    }
}
```

Определение класса `IndexSelectScan` показано в листинге 12.10. Переменная `idx` типа `Index` хранит текущую индексную запись, а переменная `ts` типа `TableScan` – текущую запись данных. Метод `next` производит переход к следующей индексной записи, соответствующей указанной константе поиска, и в случае успеха выполняет переход к записи данных в образе сканирования таблицы с идентификатором datarid из текущей индексной записи.

Листинг 12.10. Определение класса IndexSelectScan в SimpleDB

```
public class IndexSelectScan implements Scan {
    private TableScan ts;
    private Index idx;
    private Constant val;

    public IndexSelectScan(TableScan ts, Index idx, Constant val) {
        this.ts = ts;
        this.idx = idx;
        this.val = val;
        beforeFirst();
    }

    public void beforeFirst() {
        idx.beforeFirst(val);
    }

    public boolean next() {
        boolean ok = idx.next();
        if (ok) {
            RID rid = idx.getDataRid();
            ts.moveToRid(rid);
        }
        return ok;
    }
```

```
    public int getInt(String fldname) {
        return ts.getInt(fldname);
    }

    public String getString(String fldname) {
        return ts.getString(fldname);
    }

    public Constant getVal(String fldname) {
        return ts.getVal(fldname);
    }

    public boolean hasField(String fldname) {
        return ts.hasField(fldname);
    }

    public void close() {
        idx.close();
        ts.close();
    }
}
```

Обратите внимание, что табличный образ не сканируется; его текущая запись выбирается по значению datarid из индексной записи. Остальные методы класса (`getVal`, `getInt` и др.) взаимодействуют с текущей записью данных и поэтому непосредственно обращаются к образу сканирования таблицы.

12.6.2. Реализация оператора соединения с поддержкой индексов

Оператор *соединения* принимает три аргумента: две таблицы – T1 и T2 – и предикат p в форме «*A* = *B*», где *A* – поле из T1, а *B* – поле из T2. Предикат определяет, какие комбинации записей из T1 и T2 должны присутствовать в выходной таблице. Формально операция соединения определяется следующим образом: `join(T1, T2, p) ≡ select(product(T1, T2), p)`.

Соединение с использованием индекса – это реализация соединения в особом случае, когда T2 – это хранимая таблица, имеющая индекс для поля B. Работа этой реализации описывается алгоритмом 12.4.

Алгоритм 12.4. Реализация соединения с использованием индекса

Для каждой записи t1 в T1:

1. Пусть x – это значение поля A в t1.
2. Использовать индекс для B, чтобы найти индексные записи, где dataval = x.
3. Для каждой индексной записи:
   a) получить значение datarid;
   b) перейти непосредственно к записи t2 в таблице T2 с идентификатором datarid;
   c) сконструировать выходную запись (t1, t2).

Обратите внимание, что соединение с использованием индекса реализовано аналогично прямому произведению, только вместо многократного скани-

рования внутренней таблицы производится многократный поиск по индексу. По этой причине оно может выполняться намного эффективнее прямого произведения двух таблиц.

Соединение с помощью индекса реализуется классами `IndexJoinPlan` и `IndexJoinScan`. Определение `IndexJoinPlan` показано в листинге 12.11.

Листинг 12.11. Определение класса IndexJoinPlan в SimpleDB

```
public class IndexJoinPlan implements Plan {
    private Plan p1, p2;
    private IndexInfo ii;
    private String joinfield;
    private Schema sch = new Schema();

    public IndexJoinPlan(Plan p1, Plan p2, IndexInfo ii, String joinfield) {
        this.p1 = p1;
        this.p2 = p2;
        this.ii = ii;
        this.joinfield = joinfield;
        sch.addAll(p1.schema());
        sch.addAll(p2.schema());
    }

    public Scan open() {
        Scan s = p1.open();
        // сгенерирует исключение, если p2 не является табличным планом
        TableScan ts = (TableScan) p2.open();
        Index idx = ii.open();
        return new IndexJoinScan(s, idx, joinfield, ts);
    }

    public int blocksAccessed() {
        return p1.blocksAccessed()
            + (p1.recordsOutput() * ii.blocksAccessed())
            + recordsOutput();
    }

    public int recordsOutput() {
        return p1.recordsOutput() * ii.recordsOutput();
    }

    public int distinctValues(String fldname) {
        if (p1.schema().hasField(fldname))
            return p1.distinctValues(fldname);
        else
            return p2.distinctValues(fldname);
    }

    public Schema schema() {
        return sch;
    }
}
```

В аргументах `p1` и `p2` конструктор получает планы таблиц, обозначенных в алгоритме 12.4 как T1 и T2 соответственно. Через аргумент `ii` передается индекс для поля B в таблице T2, а через аргумент `joinfield` – поле A. Метод `open`

преобразует планы в образы сканирования, объект IndexInfo – в индекс и затем передает их конструктору IndexJoinScan.

Определение класса IndexJoinScan показано в листинге 12.12. Метод beforeFirst переходит к первой записи в T1, получает значение поля A и переходит к первой индексной записи, поле dataval которой имеет это значение. Метод next переходит к следующей индексной записи, если она существует. Если нет, то метод переходит к следующей записи в T1 и сбрасывает индекс, чтобы использовать новое значение dataval.

Листинг 12.12. Определение класса IndexJoinScan в SimpleDB

```
public class IndexJoinScan implements Scan {
    private Scan lhs;
    private Index idx;
    private String joinfield;
    private TableScan rhs;

    public IndexJoinScan(Scan lhs, Index idx, String joinfld, TableScan rhs) {
        this.lhs = lhs;
        this.idx = idx;
        this.joinfield = joinfld;
        this.rhs = rhs;
        beforeFirst();
    }

    public void beforeFirst() {
        lhs.beforeFirst();
        lhs.next();
        resetIndex();
    }

    public boolean next() {
        while (true) {
            if (idx.next()) {
                rhs.moveToRid(idx.getDataRid());
                return true;
            }
            if (!lhs.next())
                return false;
            resetIndex();
        }
    }

    public int getInt(String fldname) {
        if (rhs.hasField(fldname))
            return rhs.getInt(fldname);
        else
            return lhs.getInt(fldname);
    }

    public Constant getVal(String fldname) {
        if (rhs.hasField(fldname))
            return rhs.getVal(fldname);
        else
            return lhs.getVal(fldname);
    }
```

```
    public String getString(String fldname) {
        if (rhs.hasField(fldname))
            return rhs.getString(fldname);
        else
            return lhs.getString(fldname);
    }
    public boolean hasField(String fldname) {
        return rhs.hasField(fldname) || lhs.hasField(fldname);
    }

    public void close() {
        lhs.close();
        idx.close();
        rhs.close();
    }

    private void resetIndex() {
        Constant searchkey = lhs.getVal(joinfield);
        idx.beforeFirst(searchkey);
    }
}
```

12.7. Планирование обновления индекса

Если движок базы данных поддерживает индексирование, его планировщик должен вносить соответствующие изменения во все индексные записи при обновлении записи данных. Фрагмент кода в листинге 12.3 иллюстрирует, какие действия должен выполнить планировщик. В этом разделе показано, как эти действия реализуются в планировщике.

В пакете `simpledb.index.planner` имеется класс `IndexUpdatePlanner`, реализующий усовершенствованную версию базового планировщика обновлений; его определение показано в листинге 12.13.

Метод `executeInsert` извлекает информацию об индексах указанной таблицы. Как и в базовом планировщике, этот метод вызывает `setVal`, чтобы обновить значение каждого указанного поля. После каждого вызова `setVal` планировщик проверяет, имеется ли индекс для этого поля, и если есть, то вставляет новую запись в этот индекс.

Метод `executeDelete` создает образ сканирования для получения записей, подлежащих удалению, так же как в базовом планировщике. Но перед удалением каждой из этих записей метод использует значения их полей, чтобы определить, какие индексные записи нужно удалить. Затем он удаляет эти индексные записи, а потом и саму запись с данными.

Метод `executeModify` создает образ сканирования для получения записей, которые следует изменить, так же как в базовом планировщике. Но перед изменением каждой записи метод сначала корректирует индекс для каждого измененного поля, если он существует. Для этого он удаляет старую индексную запись и вставляет новую.

Методы создания таблиц, представлений и индексов действуют так же, как в базовом планировщике.

Чтобы заставить SimpleDB использовать планировщик обновлений, учитывающий индексы, необходимо изменить метод `planner` в классе `SimpleDB`: создать в нем экземпляр `IndexUpdatePlanner` вместо `BasicUpdatePlanner`.

Листинг 12.13 Определение класса IndexUpdatePlanner в SimpleDB

```
public class IndexUpdatePlanner implements UpdatePlanner {
    private MetadataMgr mdm;

    public IndexUpdatePlanner(MetadataMgr mdm) {
        this.mdm = mdm;
    }

    public int executeInsert(InsertData data, Transaction tx) {
        String tblname = data.tableName();
        Plan p = new TablePlan(tx, tblname, mdm);

        // сначала вставить запись
        UpdateScan s = (UpdateScan) p.open();
        s.insert();
        RID rid = s.getRid();

        // затем изменить каждое поле, попутно вставляя индексные записи
        Map<String,IndexInfo> indexes = mdm.getIndexInfo(tblname, tx);
        Iterator<Constant> valIter = data.vals().iterator();
        for (String fldname : data.fields()) {
            Constant val = valIter.next();
            System.out.println("Modify field " + fldname + " to val " + val);
            s.setVal(fldname, val);

            IndexInfo ii = indexes.get(fldname);
            if (ii != null) {
                Index idx = ii.open();
                idx.insert(val, rid);
                idx.close();
            }
        }
        s.close();
        return 1;
    }

    public int executeDelete(DeleteData data, Transaction tx) {
        String tblname = data.tableName();
        Plan p = new TablePlan(tx, tblname, mdm);
        p = new SelectPlan(p, data.pred());
        Map<String,IndexInfo> indexes = mdm.getIndexInfo(tblname, tx);

        UpdateScan s = (UpdateScan) p.open();
        int count = 0;
        while(s.next()) {
            // сначала удалить из всех индексов записи с данным значением RID
            RID rid = s.getRid();
            for (String fldname : indexes.keySet()) {
                Constant val = s.getVal(fldname);
                Index idx = indexes.get(fldname).open();
                idx.delete(val, rid);
                idx.close();
            }
```

```
            // затем удалить запись с данными
            s.delete();
            count++;
        }
        s.close();
        return count;
    }

    public int executeModify(ModifyData data, Transaction tx) {
        String tblname = data.tableName();
        String fldname = data.targetField();
        Plan p = new TablePlan(tx, tblname, mdm);
        p = new SelectPlan(p, data.pred());

        IndexInfo ii = mdm.getIndexInfo(tblname, tx).get(fldname);
        Index idx = (ii == null) ? null : ii.open();

        UpdateScan s = (UpdateScan) p.open();
        int count = 0;
        while(s.next()) {
            // сначала обновить запись с данными
            Constant newval = data.newValue().evaluate(s);
            Constant oldval = s.getVal(fldname);
            s.setVal(data.targetField(), newval);

            // затем обновить соответствующий индекс, если имеется
            if (idx != null) {
                RID rid = s.getRid();
                idx.delete(oldval, rid);
                idx.insert(newval, rid);
            }
            count++;
        }
        if (idx != null) idx.close();
        s.close();
        return count;
    }

    public int executeCreateTable(CreateTableData data, Transaction tx) {
        mdm.createTable(data.tableName(), data.newSchema(), tx);
        return 0;
    }

    public int executeCreateView(CreateViewData data, Transaction tx) {
        mdm.createView(data.viewName(), data.viewDef(), tx);
        return 0;
    }

    public int executeCreateIndex(CreateIndexData data, Transaction tx) {
        mdm.createIndex(data.indexName(), data.tableName(),
        data.fieldName(), tx);
        return 0;
    }
}
```

12.8. Итоги

- *Индекс* для поля A в таблице T – это файл, содержащий одну индексную запись для каждой записи в таблице T. Каждая индексная запись имеет два поля: поле *dataval* содержит значение поля A из соответствующей записи в T, а поле *datarid* – идентификатор этой табличной записи.
- Индекс может повысить эффективность операций селекции и соединения. Вместо сканирования каждого блока таблицы данных система может поступить проще:
  - найти в индексе все индексные записи с указанным значением в поле dataval;
  - из каждой найденной индексной записи извлечь значение ее поля datarid, чтобы получить доступ к нужной записи с данными.

  При таком подходе система баз данных может обращаться только к тем блокам данных, которые содержат искомые записи.
- Индексы не всегда полезны. Как правило, полезность индекса для поля A пропорциональна количеству уникальных значений в этом поле.
- Запрос может выполняться с использованием индексов разными способами. Обработчик запросов должен определить, какой из них является лучшим.
- Движок базы данных должен обновлять индексы при изменении таблиц. Он должен вставлять записи в каждый индекс (или удалять их из индекса), когда в таблицу вставляется (или удаляется) запись с данными. Учитывая эти дополнительные накладные расходы на поддержку индексов, желательно создавать только самые полезные индексы.
- Индексы реализованы так, что поиск выполняется за небольшое количество обращений к диску. В этой главе обсуждались три стратегии реализации индекса: *статическое хеширование*, *расширяемое хеширование* и *B-деревья*.
- В стратегии статического хеширования индексные записи хранятся в фиксированном количестве *ячеек*, и для каждой ячейки создается отдельный файл. *Хеш-функция* определяет ячейку для каждой индексной записи. Чтобы найти индексную запись с использованием статического хеширования, диспетчер индексов хеширует ключ поиска и проверяет соответствующую ячейку. Если индекс содержит `B` блоков и `N` ячеек, то каждая ячейка будет размещаться в среднем в `B/N` блоках, то есть для обхода ячейки в среднем потребуется `B/N` обращений к блокам.
- В стратегии расширяемого хеширования ячейки могут совместно использовать одни и те же блоки. Этот подход имеет более высокую эффективность по сравнению со статическим хешированием, потому что позволяет использовать очень много ячеек при относительно небольшом размере индексного файла. Совместное использование блоков обеспечивается с помощью *каталога ячеек*. Каталог ячеек можно рассматривать как массив `Dir` целых чисел; если индексная запись хешируется в ячейку `b`, то она будет сохранена в блоке `Dir[b]` в файле ячеек. Если новая индексная запись не помещается в свой блок, то блок *расщепляется*, каталог ячеек обновляется и записи в блоке повторно хешируются.

- В стратегии с использованием B-дерева индексные записи хранятся в файле в порядке сортировки по значению dataval. Для B-дерева также создается файл с каталожными записями. Для каждого блока в индексе имеется соответствующая каталожная запись, которая содержит данные из первой индексной записи в блоке и ссылку на этот блок. Эти каталожные записи образуют уровень 0 B-дерева. Аналогично для каждого блока каталога имеется своя каталожная запись, хранящаяся на следующем уровне каталога. Самый верхний уровень состоит из единственного блока, который называется *корнем* B-дерева. Имея искомое значение dataval, мы можем выполнить поиск по каталогу, исследовав один блок на каждом уровне дерева каталога; в результате мы получим блок индекса, содержащий нужные индексные записи.
- Индексы на основе B-дерева очень эффективны. Любую запись данных можно получить не более чем за пять обращений к диску, за исключением совсем уж огромных таблиц. Если коммерческая система баз данных реализует только одну стратегию индексирования, она почти наверняка использует B-дерево.

12.9. Для дополнительного чтения

Эта глава рассматривает индексы как вспомогательные файлы. В статье Sieg and Sciore (1990) показано, что индексы можно рассматривать как особый тип таблиц, а операторы селекции и соединения с поддержкой индексов – как операторы реляционной алгебры. Такой подход позволяет планировщику использовать индексы гораздо более гибко.

B-деревья и хеш-файлы – это универсальные индексные структуры, которые лучше всего подходят для запросов с единственным селективным ключом поиска. Они хуже подходят для обработки запросов, имеющих несколько ключей поиска, которые, например, часто используются в географических и пространственных базах данных. (Например, B-дерево не сможет помочь обработать такой запрос: «найти все рестораны не дальше 2 миль от моего дома».) Для работы с подобными базами данных были разработаны *многомерные индексы*. Обзор этих индексов приводится в статье Gaede and Gunther (1998).

Стоимость поиска в B-дереве определяется его высотой, которая зависит от размера индексных и каталожных записей. В статье Bayer and Unteraurer (1977) приводятся методы уменьшения размера этих записей. Например, если поле dataval в листовом узле является строковым и все строки имеют общий префикс, то этот префикс можно сохранить один раз в начале страницы, а в индексных записях хранить только окончания. Кроме того, обычно нет необходимости хранить в каталожной записи строковое значение dataval целиком; можно сохранить лишь префикс, имеющий достаточную длину, чтобы можно было однозначно определить дочернюю запись для выбора.

Статья Graefe (2004) описывает новую реализацию B-деревьев, в которых узлы никогда не переопределяются; вместо обновления существующих создаются новые узлы. Статья показывает, как эта реализация позволяет ускорить операции обновления за счет немного более медленного чтения.

Эта глава сосредоточилась исключительно на минимизации количества обращений к диску, выполняемых в ходе поиска по B-дереву. Хотя затраты процессорного времени при поиске по B-дереву менее важны, они часто оказываются значительными и должны учитываться коммерческими реализациями. В статье Lomet (2001) обсуждается, как структурировать узлы B-дерева, чтобы минимизировать количество просматриваемых узлов при поиске. В статье Chen et al. (2002) показано, как структурировать узлы B-дерева, чтобы максимизировать производительность кеша процессора.

Также в этой главе не рассматривался вопрос о блокировке узлов B-дерева. SimpleDB блокирует узлы так же, как любые другие блоки с данными, и удерживает блокировку до завершения транзакции. Однако, как оказывается, B-деревья не должны удовлетворять протоколу блокировки, описанному в главе 5, чтобы гарантировать сериализуемость, и могут освобождать блокировки раньше. Эта проблема освещается в статье Bayer and Schkolnick (1977).

Системы веб-поиска хранят базы данных веб-страниц, которые в основном являются текстовыми. Запросы к этим базам данных, как правило, основаны на сопоставлении строк и шаблонов, для которых традиционные структуры индексации обычно бесполезны. Методы индексации текстов рассматриваются в статье Faloutsos (1985).

Одна необычная стратегия индексирования основана на хранении битовой карты для каждого значения поля; битовая карта содержит один бит для каждой записи с данными и указывает, содержит ли запись это значение. Интересно отметить, что индексы с битовыми картами легко приспособить для обработки нескольких ключей поиска. В статье O'Neil and Quass (1997) объясняется, как работают индексы на основе битовых карт.

В главе 6 предполагается, что таблицы хранятся последовательно и в основном никак не организованы. Однако таблицы тоже можно организовать с применением B-деревьев, хеширования или любой другой стратегии индексирования. Однако это сопряжено с некоторыми сложностями. Например, в B-дереве запись может переместиться в другой блок, если этот блок расщепится; по этой причине обращение с идентификаторами записей требует большой осторожности. Кроме того, стратегия индексирования должна также поддерживать последовательное сканирование таблиц (и фактически всех интерфейсов `Scan` и `UpdateScan`). Но основные принципы остаются неизменными. Статья Batory (1982) описывает, как на основе базовых стратегий индексирования можно строить сложные файловые организации.

Batory, D., & Gotlieb, C. (1982). «A unifying model of physical databases». *ACM Transactions of Database Systems*, 7 (4), 509–539.

Bayer, R., & Schkolnick, M. (1977). «Concurrency of operations on B-trees». *Acta Informatica*, 9 (1), 1–21.

Bayer, R., & Unterauer, K. (1977). «Prefix B-trees». *ACM Transactions of Database Systems*, 2 (1), 11–26.

Chen, S., Gibbons, P., Mowry, T., & Valentin, G. (2002). «Fractal prefetching B+-trees: Optimizing both cache and disk performance». *Proceedings of the ACM SIGMOD Conference*, p. 157–168.

Faloutsos, C. (1985). «Access methods for text». *ACM Computing Surveys*, 17 (1), 49–74.

Graede, V., & Gunther, O. (1998). «Multidimensional access methods». *ACM Computing Surveys*, 30 (2), 170–231.

Graefe, G. (2004) «Write-optimized B-trees». *Proceedings of the VLDB Conference*, p. 672–683.

Lomet, D. (2001). «The evolution of effective B-tree: Page organization and techniques: A personal account». *ACM SIGMOD Record*, 30 (3), 64–69.

O'Neil, P., & Quass, D. (1997). «Improved query performance with variant indexes». *Proceedings of the ACM SIGMOD Conference*, p. 38–49.

Sieg, J., & Sciore, E. (1990). «Extended relations». *Proceedings of the IEEE Data Engineering Conference*, p. 488–494.

12.10. Упражнения

Теория

12.1. Взгляните на университетскую базу данных в табл. 1.1. Какие поля не подходят для индексации? Объясните, почему.

12.2. Объясните, какие индексы могут пригодиться для обработки каждого из следующих запросов:

(a)
```
select SName
   from STUDENT, DEPT
   where MajorId = DId and DName = 'math' and GradYear <> 2001
```

(b)
```
select Prof
   from ENROLL, SECTION, COURSE
   where SectId = SectionId and CourseId = CId
   and Grade = 'F' and Title = 'calculus'
```

12.3. Допустим, вы решили создать индекс для поля *GradYear* в таблице STUDENT.

a) Рассмотрите следующий запрос:

```
select from STUDENT where GradYear=2020
```

Вычислите стоимость использования индекса при обработке этого запроса, основываясь на статистике, приведенной в табл. 7.2, и предположив, что студенты равномерно распределены по 50 выпускным годам.

b) Выполните те же вычисления, что и в п. «a», но исходя из предположения, что выпускных лет было не 50, а 2, 10, 20 и 100.

12.4. Покажите, что индекс для поля `A` бесполезен, если число уникальных значений в `A` меньше числа записей таблицы, умещающихся в один блок.

12.5. Есть ли смысл в создании индекса для другого индекса? Объясните, почему.

12.6. Пусть блоки имеют размер 120 байт и таблица DEPT содержит 60 записей. Вычислите, сколько блоков потребуется для хранения индексных записей для каждого поля DEPT.

12.7. Интерфейс `Index` определяет метод `delete`, который удаляет индексную запись с указанными значениями dataval и datarid. Есть ли смысл определить также метод `deleteAll`, удаляющий все индексные записи с указанным значением dataval? Как и когда планировщик мог бы использовать этот метод?

12.8. Взгляните на следующее соединение двух таблиц:

```
select SName, DName
from STUDENT, DEPT
where MajorId = DId
```

Если предположить, что таблица STUDENT содержит индекс для поля `MajorId` и таблица DEPT содержит индекс для `Did`, то имеется два способа выполнить это соединение с использованием индекса – одного или другого. Используя информацию из табл. 7.2, сравните стоимости этих двух планов. Какой общий вывод можно сделать из результатов сравнения?

12.9. Пример расширяемого хеширования в разделе 12.4 вставляет всего семь записей. Продолжите пример, вставив записи для студентов с идентификаторами 28, 9, 16, 24, 36, 48, 64 и 56.

12.10. Пусть используется стратегия расширяемого хеширования и имеется индексный блок с локальной глубиной `L`. Покажите, что все хеши, ссылающиеся на этот блок, имеют одинаковые правые `L` бит.

12.11. В стратегии расширяемого хеширования файл ячеек увеличивается, когда происходит расщепление блока. Разработайте алгоритм удаления, который позволяет объединить два блока, полученных в результате расщепления. Насколько это практично?

12.12. Пусть используется стратегия расширяемого хеширования и в блок помещается 100 индексных записей. Допустим также, что в настоящее время индекс пуст.

a) Сколько записей можно вставить до того, как глобальная глубина индекса станет равной 1?
b) Сколько записей можно вставить до того, как глобальная глубина индекса станет равной 2?

12.13. Предположим, что вставка в расширяемый хеш-индекс привела к увеличению глобальной глубины с 3 до 4.

a) Сколько записей в каталоге ячеек теперь будет находиться?
b) Сколько блоков в файле ячеек будут иметь ровно одну каталожную запись, ссылающуюся на них?

12.14. Объясните, почему в стратегии расширяемого хеширования достаточно двух обращений к блокам, чтобы получить любую индексную запись, независимо от размера индекса.

12.15. Предположим, вы создали индекс на основе B-дерева для поля `SId`. Допустим также, что в один блок помещается три индексные и три каталожные записи. Нарисуйте B-дерево, которое получится в результате вставки записей для студентов 8, 12, 1, 20, 5, 7, 2, 28, 9, 16, 24 36, 48, 64 и 56.

12.16. Пусть имеется индекс на основе B-дерева для поля `StudentId` в таблице ENROLL и в один блок помещается 100 индексных или каталожных записей. Опираясь на статистику в табл. 7.2:

a) вычислите размер индексного файла в блоках;
b) вычислите размер файла каталога.

12.17. Пусть имеется индекс на основе B-дерева для поля `StudentId` в таблице ENROLL и в один блок помещается 100 индексных или каталожных записей. Предположим, что в настоящее время индекс пуст.

a) Сколько операций вставки нужно выполнить, чтобы произошло расщепление корневого блока (с образованием нового корня на уровне 1)?

b) Какое наименьшее количество операций вставки приведет к новому расщеплению корня (с образованием нового корня на уровне 2)?

12.18. Рассмотрим реализацию B-деревьев в SimpleDB.

a) Какое максимальное количество буферов будет закреплено одновременно во время сканирования индекса?

b) Какое максимальное количество буферов будет закреплено одновременно во время вставки?

12.19. Классы `IndexSelectPlan` и `IndexSelectScan` в SimpleDB предполагают, что предикат селекции выполняет проверку на равенство, например «*GradYear = 2019*». Однако индекс также можно было бы использовать с предикатом диапазона, таким как «*GradYear > 2019*».

a) Объясните в общих чертах, как можно было бы использовать индекс на основе B-дерева для поля `GradYear`, чтобы обработать следующий запрос:

```
select SName from STUDENT where GradYear > 2019
```

b) Какие изменения необходимо внести в реализацию B-дерева в SimpleDB, чтобы воплотить ваш ответ в п. «a»?

c) Пусть имеется индекс на основе B-дерева для поля `GradYear`. Объясните, почему этот индекс может оказаться бесполезным для обработки запроса. В каких случаях он может пригодиться?

d) Объясните, почему статические и расширяемые хеш-индексы никогда не будут использоваться для обработки этого запроса.

Практика

12.20. Методы `executeDelete` и `executeUpdate` в планировщике обновлений в SimpleDB используют образ сканирования для оператора селекции (`SelectScan`), чтобы найти требуемые записи. Однако образ можно построить и с поддержкой индекса (`IndexSelectScan`), если подходящий индекс существует.

a) Объясните, как для этого следует изменить алгоритмы планирования.

b) Внесите необходимые изменения в SimpleDB.

12.21. Реализуйте расширяемое хеширование. Выберите максимальную глубину, при которой создается каталог не более чем в двух дисковых блоках.

12.22. Представьте следующую модификацию индексных записей: вместо идентификатора соответствующей записи данных индексная запись хранит только номер блока, где находится эта запись данных. Индексных записей при такой организации может быть меньше, чем записей данных, – если блок данных содержит несколько записей с одним

и тем же ключом поиска, им всем будет соответствовать единственная индексная запись.

a) Объясните, почему эта модификация может уменьшить количество обращений к диску при обработке запросов с использованием индекса.
b) Как следует изменить методы удаления и вставки индекса, чтобы адаптировать их к этой модификации? Потребуется ли им больше обращений к диску, чем существующим методам? Напишите код, реализующий эту модификацию, как для B-деревьев, так и для стратегии статического хеширования.
c) Считаете ли вы эту модификацию заслуживающей внимания?

12.23. Многие коммерческие системы баз данных позволяют указывать индексы в SQL-операторе `create table`. Например, вот как выглядит такой оператор в MySQL:

```
create table T (A int, B varchar(9), index(A), C int, index(B))
```

То есть элементы вида `index(<field>)` могут появляться в любом месте в списке имен полей.

a) Добавьте в синтаксический анализатор SimpleDB обработку этого дополнительного синтаксиса.
b) Добавьте в планировщик SimpleDB возможность создания соответствующего плана.

12.24. Одна из проблем, связанных с методом `executeCreateIndex` планировщика обновлений, заключается в том, что он создает пустой индекс, даже если индексируемая таблица содержит записи. Переделайте метод так, чтобы он автоматически вставлял индексные записи для всех существующих записей в индексируемой таблице.

12.25. Добавьте в SimpleDB поддержку оператора `drop index`. Определите свой собственный синтаксис и внесите необходимые изменения в синтаксический анализатор и планировщик.

12.26. Измените SimpleDB так, чтобы пользователь мог указать тип вновь создаваемого индекса.

a) Разработайте новый синтаксис для оператора `create index` и определите его грамматику.
b) Измените синтаксический (и, возможно, лексический) анализатор, чтобы реализовать ваш новый синтаксис.

12.27. Реализуйте статическое хеширование, используя единственный индексный файл. В первых *N* блоках этого файла будут находиться первые блоки всех ячеек. Остальные блоки ячеек связаны в цепочку: каждый следующий блок будет определяться по целому числу, хранимому в предыдущем блоке. (Например, если в блоке 1 хранится число 173, то следующим блоком в цепочке будет блок 173. Конец цепочки отмечается числом 1.) Для простоты это число можно хранить в первой записи в каждом блоке.

12.28. В SimpleDB расщепление блоков в B-дереве происходит сразу при заполнении всего доступного места в них. Другое возможное решение

состоит в том, чтобы позволить блокам оставаться заполненными и расщеплять их в методе `insert`. Например, когда код перемещается вниз по дереву в поисках листового блока, он может расщеплять любые заполненные блоки, встретившиеся ему на пути.

a) Реализуйте этот алгоритм.
b) Объясните, как этот код уменьшает потребность метода `insert` в буферах.

Глава 13

Материализация и сортировка

В этой главе рассматриваются следующие операторы реляционной алгебры: *материализации (materialize)*, *сортировки (sort)*, *группировки (groupby)* и *соединения слиянием (mergejoin)*[1]. Эти операторы материализуют свои входные записи, сохраняя их во временных таблицах. Материализация позволяет операторам многократно обращаться к своим записям без повторного их вычисления, что может оказаться существенно эффективнее просто конвейерной обработки.

13.1. Цель материализации

Все операторы, которые вы видели до сих пор, имели *конвейерную* реализацию, которая обладает следующими характеристиками:

- записи вычисляются по мере необходимости и не сохраняются;
- получить ранее просмотренные записи можно, только выполнив всю операцию с самого начала.

В этой главе рассматриваются операторы, которые *материализуют* входные записи. Образы сканирования для этих операторов читают входные записи на этапе открытия и сохраняют выходные записи в одной или нескольких временных таблицах. Такие реализации называют *реализациями с предварительной обработкой* входных данных, потому что они выполняют все необходимые вычисления до любых обращений к их выходным записям. Целью материализации является повышение эффективности последующего сканирования.

Например, рассмотрим оператор *groupby*, который будет представлен в разделе 13.5. Он группирует входные записи по указанным полям группировки и для каждой группы вычисляет агрегатные функции. Самый простой способ определить группы – отсортировать входные записи по полям группировки, в результате чего записи каждой группы расположатся рядом друг с другом. Поэтому хорошей стратегией реализации является материализация во временной таблице входных записей, отсортированных по полям группировки. После этого можно вычислить агрегатные функции, выполнив всего один проход по временной таблице.

Материализация – это палка о двух концах. С одной стороны, использование временной таблицы значительно повышает эффективность сканирования. С другой стороны, создание временной таблицы сопряжено со значительными

---

[1] Эти операторы, конечно, не являются частью реляционной алгебры, но используются в планах выполнения SQL-запросов. – *Прим. ред.*

накладными расходами на запись и чтение. Более того, создание временной таблицы означает предварительную обработку всех входных записей, даже если клиента интересует лишь несколько выходных записей.

Материализация полезна, только когда эти затраты компенсируются более высокой эффективностью сканирования. Все четыре оператора, описываемых в этой главе, имеют высокоэффективные материализующие реализации.

13.2. Временные таблицы

Материализующие реализации сохраняют входные записи во *временных таблицах*. Временная таблица имеет три отличия от обычной таблицы:

- временная таблица создается без использования метода `createTable` диспетчера таблиц, и ее метаданные не сохраняются в системном каталоге. В SimpleDB каждая временная таблица сама поддерживает свои метаданные и имеет свой метод `getLayout`;
- временные таблицы автоматически удаляются системой баз данных, когда они больше не нужны. В SimpleDB временные таблицы удаляются диспетчером файлов во время инициализации системы;
- диспетчер восстановления не журналирует изменения во временных таблицах. После обработки запроса временная таблица больше никогда не будет использоваться, поэтому нет необходимости восстанавливать предыдущее ее состояние.

В SimpleDB временные таблицы реализованы в виде класса `TempTable`, определение которого показано в листинге 13.1. Конструктор создает пустую таблицу и присваивает ей уникальное имя (в форме «tempN», где N – некоторое целое число). Класс имеет три общедоступных метода. Метод `open` создает образ сканирования, а методы `tableName` и `getLayout` возвращают метаданные временной таблицы.

Листинг 13.1. Определение класса TempTable в SimpleDB

```
public class TempTable {
    private static int nextTableNum = 0;
    private Transaction tx;
    private String tblname;
    private Layout layout;

    public TempTable(Transaction tx, Schema sch) {
        this.tx = tx;
        tblname = nextTableName();
        layout = new Layout(sch);
    }

    public UpdateScan open() {
        return new TableScan(tx, tblname, layout);
    }

    public String tableName() {
        return tblname;
    }
```

```
    public Layout getLayout() {
        return layout;
    }
    private static synchronized String nextTableName() {
        nextTableNum++;
        return "temp" + nextTableNum;
    }
}
```

13.3. Материализация

В этом разделе описывается новый оператор реляционной алгебры: *материализация (materialize)*. Этот оператор не имеет видимой функциональности. Он принимает единственный аргумент с именем таблицы и возвращает набор записей, точно совпадающих с записями в исходной таблице. То есть

```
materialize(T) ≡ T
```

Цель оператора материализации – сохранить выходные данные подзапроса во временной таблице, чтобы не вычислять их по нескольку раз. В этом разделе рассматривается порядок использования и реализация этого оператора.

13.3.1. Пример материализации

Рассмотрим дерево запроса на рис. 13.1a. Как мы уже знаем, операция прямого произведения проверяет все записи в правом поддереве для каждой записи в левом поддереве. Как следствие обращение к записям в левом поддереве происходит только один раз, а к записям в правом поддереве – много раз.

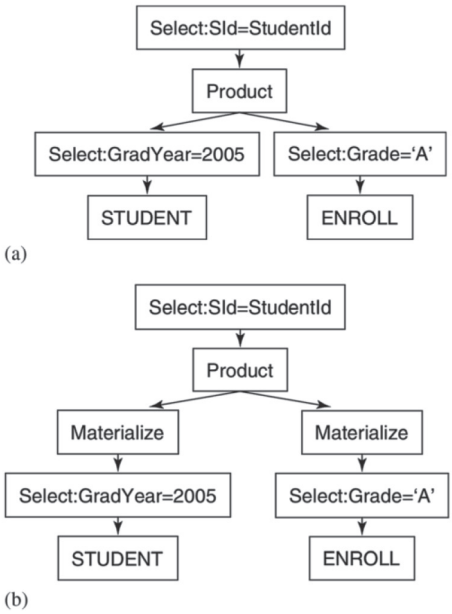


Рис. 13.1. Где использовать оператор материализации: a) исходный запрос; b) материализация левой и правой сторон прямого произведения

Проблема с многократным доступом к записям в правом поддереве заключается в том, что их придется снова и снова вычислять заново. На рис. 13.1a реализация должна будет прочитать всю таблицу ENROLL несколько раз, чтобы каждый раз найти в ней записи с оценкой «A». Используя статистику в табл. 7.2, можно вычислить стоимость прямого произведения. В 2005 году было выпущено 900 студентов. Конвейерная реализация будет читать всю таблицу ENROLL, занимающую 50 000 блоков, для каждого из этих 900 учеников, то есть всего будет выполнено 45 000 000 обращений к блокам ENROLL. Добавим сюда 4500 обращений к блокам STUDENT и получим общее число обращений 45 004 500.

Дерево запроса на рис. 13.1b имеет два узла materialize. Сначала рассмотрим узел materialize над узлом select справа. Этот узел создает временную таблицу с записями из ENROLL, имеющими оценку «A». Каждый раз, когда узел product запрашивает запись с правой стороны, узел materialize будет извлекать ее непосредственно из этой временной таблицы вместо поиска по таблице ENROLL.

Такая материализация значительно снижает затраты на вычисление прямого произведения. Проанализируем ситуацию. Временная таблица в 14 раз меньше таблицы ENROLL и занимает 3572 блока. Правому узлу materialize потребуется выполнить 53 572 обращения к блокам, чтобы создать таблицу (50 000 обращений, чтобы прочитать ENROLL, и 3572 обращения, чтобы записать данные во временную таблицу). После создания временная таблица будет прочитана 900 раз, что потребует 3 214 800 обращений к блокам. Добавим сюда 4500 обращений к блокам таблицы STUDENT и получим всего 3 272 872 обращения. Другими словами, материализация снизила стоимость выполнения дерева запроса по сравнению с оригиналом на 82 % (если принять, что одно обращение к блоку занимает 1 мс, то экономия составит более 11 часов). Стоимость создания временной таблицы незначительна по сравнению с экономией, которую она дает.

Теперь рассмотрим узел materialize слева на рис. 13.1b. Он отсканирует таблицу STUDENT и создаст временную таблицу, содержащую студентов, выпущенных в 2005 году. Узел product проверит эту временную таблицу только один раз. Однако узел product в оригинальном дереве запроса тоже проверит таблицу STUDENT лишь один раз. Поскольку записи STUDENT проверяются только один раз в каждом случае, левый узел materialize фактически увеличивает стоимость запроса. В общем случае материализация полезна, лишь когда ее результаты будут просматриваться несколько раз.

13.3.2. Стоимость материализации

На рис. 13.2 изображена структура дерева запроса, содержащего узел materialize. Входными данными для узла служат результаты, возвращаемые подзапросом, обозначенным как T2. Когда пользователь открывает план для запроса T1, его корневой план откроет дочерние планы, находящиеся ниже в дереве. В момент открытия план оператора материализации обработает свои входные данные. В частности, он откроет образ сканирования для T2, выполнит обход его содержимого, сохранит результаты во временной таблице и закроет образ. Во время сканирования запроса T1 образ сканирования оператора материализации вернет соответствующую запись из своей временной таблицы.

Обратите внимание, что подзапрос T2 будет выполнен только один раз, чтобы заполнить временную таблицу; после этого он станет ненужным.

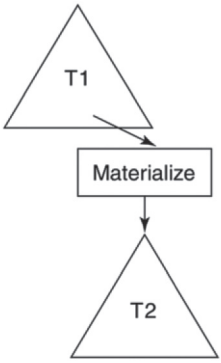


Рис. 13.2. Дерево запроса, содержащее узел materialize

Два слагаемых стоимости материализации – стоимость предварительной обработки входных данных и стоимость сканирования. Стоимость предварительной обработки – это стоимость подзапроса T2 плюс стоимость сохранения его результатов во временную таблицу. Стоимость сканирования – это стоимость чтения записей из временной таблицы. Если предположить, что временная таблица занимает `B` блоков, то эти стоимости можно выразить так:

- стоимость предварительной обработки = `B` + стоимость подзапроса;
- стоимость сканирования = `B`.

13.3.3. Реализация оператора материализации

В SimpleDB оператор материализации реализован в виде класса `MaterializePlan`, определение которого показано в листинге 13.2. Метод `open` выполняет предварительную обработку входных данных: создает новую временную таблицу, открывает образы сканирования для временной таблицы и входного набора данных, копирует входные записи в образ сканирования временной таблицы, закрывает образ входного набора данных и возвращает образ временной таблицы. Метод `blocksAccessed` возвращает приблизительную оценку размера материализованной таблицы. Для этого он вычисляет количество новых записей в блоке (Records Per Block, RPB) и делит общее количество выходных записей на это значение RPB. Методы `recordsOutput` и `distinctValues` возвращают значения, полученные из базового плана.

Листинг 13.2. Определение класса MaterializePlan в SimpleDB

```
public class MaterializePlan implements Plan {
    private Plan srcplan;
    private Transaction tx;

    public MaterializePlan(Transaction tx, Plan srcplan) {
        this.srcplan = srcplan;
        this.tx = tx;
    }
```

```
    public Scan open() {
        Schema sch = srcplan.schema();
        TempTable temp = new TempTable(tx, sch);
        Scan src = srcplan.open();
        UpdateScan dest = temp.open();
        while (src.next()) {
            dest.insert();
            for (String fldname : sch.fields())
                dest.setVal(fldname, src.getVal(fldname));
        }
        src.close();
        dest.beforeFirst();
        return dest;
    }

    public int blocksAccessed() {
        // создать фиктивный объект Layout, чтобы вычислить размер слота
        Layout y = new Layout(srcplan.schema());
        double rpb = (double) (tx.blockSize() / y.slotSize());
        return (int) Math.ceil(srcplan.recordsOutput() / rpb);
    }

    public int recordsOutput() {
        return srcplan.recordsOutput();
    }

    public int distinctValues(String fldname) {
        return srcplan.distinctValues(fldname);
    }

    public Schema schema() {
        return srcplan.schema();
    }
}
```

Обратите внимание, что `blocksAccessed` не включает стоимость предварительной обработки, потому что временная таблица создается один раз, но сканироваться может неоднократно. Если вы решите включить стоимость создания таблицы в свои формулы затрат, то добавьте новый метод (например, `preprocessingCost`) в интерфейс `Plan` и переделайте все формулы оценки плана, включив данный метод. Это задание будет предложено выполнить в упражнении 13.9. Также вполне допустимо предположить, что стоимость предварительной обработки незначительна, и просто игнорировать ее в своих оценках.

Обратите внимание на отсутствие класса `MaterializeScan`: метод `open` возвращает образ сканирования для временной таблицы.

13.4. Сортировка

Другой полезный оператор реляционной алгебры – оператор *сортировки (sort)*. Он принимает два аргумента: входную таблицу и список полей. Выходная таблица содержит те же записи, что и входная таблица, но отсортирована по указанным полям. Например, следующий запрос сортирует таблицу STUDENT по полю `GradYear`, а внутри каждого года студенты дополнительно сортируются по

именам. Если два студента имеют одинаковое имя и год выпуска, их записи могут следовать в любом порядке.

```
sort(STUDENT, [GradYear, SName])
```

Планировщик использует сортировку для реализации предложения `order by` в SQL-запросах. Сортировка также будет использоваться для реализации операторов *группировки* и *соединения слиянием* далее в этой главе. Движок базы данных должен уметь эффективно сортировать записи. В этом разделе рассматривается проблема сортировки и ее решение в SimpleDB.

13.4.1. Почему сортировке необходима материализация

Сортировку можно реализовать без материализации. Например, рассмотрим узел sort в дереве запроса на рис. 13.3. Входными данными для этого узла служит набор студентов и их специальности, а выходные данные сортируются по именам студентов. Для простоты предположим, что все студенты имеют уникальные имена, поэтому входные записи имеют разные значения поля сортировки.

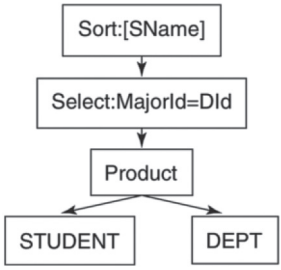


Рис. 13.3. Дерево запроса, содержащее узел sort

В нематериализующей реализации оператора sort метод `next` должен переместиться в образе сканирования к входной записи, имеющей следующее наибольшее значение `SName`. Для этого метод должен дважды просмотреть входные записи: сначала найти следующее наибольшее значение, а затем перейти к записи с этим значением. Такая реализация возможна, но она исключительно неэффективна и совершенно непрактична для больших таблиц.

В материализующей реализации оператора sort метод `open` предварительно обработает входные записи, сохранив их в отсортированном порядке во временной таблице. После этого каждый вызов `next` будет просто извлекать следующую запись из временной таблицы. Эта реализация обеспечивает очень эффективное сканирование за счет некоторой предварительной обработки. Если предположить, что создание и сортировка временной таблицы выполняются относительно эффективно (что вполне возможно), то материализующая реализация будет значительно дешевле нематериализующей.

13.4.2. Простой алгоритм сортировки слиянием

Стандартные алгоритмы сортировки, которые преподаются на начальных курсах по программированию (например, сортировка вставкой и быстрая сортировка), называются алгоритмами *внутренней* сортировки, потому что они

требуют, чтобы все записи находились в памяти одновременно. Движок базы данных, однако, не может положиться на то, что таблица целиком поместится в память, поэтому он должен использовать алгоритмы *внешней* сортировки. Самый простой и распространенный алгоритм внешней сортировки называется *сортировка слиянием (mergesort)*.

Алгоритм сортировки слиянием основан на понятии *серии*. Серия – это отсортированная часть таблицы. Несортированная таблица имеет несколько серий, а отсортированная – ровно одну. Например, предположим, что требуется отсортировать студентов по их идентификаторам, причем значения `SId` в записях STUDENT в настоящее время располагаются в следующем порядке:

```
2 6 20 4 1 16 19 3 18
```

Эта таблица имеет четыре серии. Первая серия содержит [2, 6, 20], вторая – [4], третья – [1, 16, 19], и четвертая – [3, 18].

Сортировка слиянием выполняется в два этапа. На первом этапе, который называется *расщеплением*, алгоритм сканирует входные записи и помещает каждую серию в свою временную таблицу. На втором этапе, который называется *слиянием*, алгоритм многократно объединяет полученные серии, пока не останется одна; эта последняя серия и представляет результат сортировки.

Этап слияния выполняется как последовательность *итераций*. Во время каждой итерации текущий набор серий делится на пары; затем каждая пара серий объединяется в одну серию. Получившиеся в результате этого серии формируют новый текущий набор серий. Этот новый набор будет содержать вдвое меньше серий, чем предыдущий. Итерации продолжаются до тех пор, пока текущий набор не будет содержать ровно одну серию.

Для демонстрации сортировки слиянием отсортируем записи в таблице STUDENT, как было определено выше. Фаза расщепления определяет четыре серии и сохраняет каждую в своей временной таблице:

```
Серия 1: 2 6 20
Серия 2: 4
Серия 3: 1 16 19
Серия 4: 3 18
```

В первой итерации этап слияния объединит серии 1 и 2 в серию 5 и серии 3 и 4 в серию 6:

```
Серия 5: 2 4 6 20
Серия 6: 1 3 16 18 19
```

Во второй итерации этап слияния объединит серии 5 и 6 в серию 7:

```
Серия 7: 1 2 3 4 6 16 18 19 20
```

Получив единственную серию, алгоритм остановится. Он отсортировал таблицу, используя всего две итерации слияния.

Предположим, что изначально таблица имеет 2^N серий. Каждая итерация слияния преобразует пары серий в новые объединенные серии, то есть сокращает количество серий в 2 раза. Таким образом, для сортировки файла потребуется `N` итераций: первая итерация сократит его до 2^{N-1} серий, вторая до 2^{N-2} серий, и `N`-я до $2^0 = 1$ серии. В общем случае таблица с `R` начальными сериями будет отсортирована за $\log_2 R$ итераций слияния.

13.4.3. Оптимизация алгоритма сортировки слиянием

Есть три способа повысить эффективность этого простого алгоритма сортировки слиянием:

- увеличить число серий, объединяемых в одной итерации;
- уменьшить число первоначальных серий;
- исключить операцию записи получившейся отсортированной таблицы.

Все эти способы рассматриваются далее в этом разделе.

Увеличение числа серий, объединяемых в одной итерации

Вместо двух алгоритм может объединять в каждой итерации сразу три или даже больше серий. Предположим, что алгоритм объединяет в одной итерации сразу k серий. Для этого он должен открыть образ сканирования для каждой из *k* временных таблиц. На каждом шаге он просматривает текущую запись из каждого образа, копирует запись с наименьшим значением в выходную таблицу и переходит к следующей записи в этом образе. Этот шаг повторяется до тех пор, пока записи из всех k серий не будут скопированы в выходную таблицу.

Объединение сразу нескольких серий уменьшает количество итераций, необходимых для сортировки таблицы. Если в таблице изначально имеется R начальных серий и в каждой итерации объединяется k, то для сортировки файла потребуется $\log_k R$ итераций. Но какое значение k выбрать? Почему бы просто не объединить все серии в одной итерации? Ответ на этот вопрос зависит от количества доступных буферов. Чтобы объединить сразу k серий, необходимо k+1 буферов, по одному для каждого из k входных образов сканирования и один для выходного образа. Пока предположим, что алгоритм выбирает произвольное значение, а в главе 14 мы рассмотрим алгоритм выбора наилучшего значения для k.

Уменьшение числа первоначальных серий

Чтобы уменьшить количество начальных серий, нужно увеличить количество записей в одной серии. Есть два алгоритма, которые можно использовать для этого.

Первый из них – алгоритм 13.1. Он игнорирует серии, получившиеся из входных записей, и создает серии, длина которых всегда равна одному блоку, многократно сохраняя входные записи из блока во временной таблице. Поскольку блок записей будет находиться в странице буфера в памяти, алгоритм может использовать сортировку в памяти (например, быструю сортировку), чтобы упорядочить записи без обращений к диску. Отсортированные записи в блоке образуют серию, и алгоритм сохраняет этот блок на диск.

Алгоритм 13.1. Создание начальных серий с длиной в один блок

Повторять, пока не останется входных записей:

1. Прочитать блок с входными записями в новую временную таблицу.
2. Отсортировать эти записи с использованием алгоритма сортировки в памяти.
3. Сохранить на диск получившуюся одноблочную временную таблицу.

Второй алгоритм действует аналогично, но использует дополнительный блок памяти в качестве «промежуточного хранилища» входных записей. Сначала

он заполняет промежуточное хранилище записями. Затем последовательно переносит очередную запись из промежуточного хранилища в текущую серию и добавляет в промежуточное хранилище новую входную запись. Процесс заканчивается, когда все записи в промежуточном хранилище окажутся меньше последней записи в серии. После этого текущая серия закрывается, и создается новая. Описание этой процедуры приводится в алгоритме 13.2.

Алгоритм 13.2. Создание больших начальных серий

1. Заполнить входными записями одноблочное промежуточное хранилище.
2. Создать новую серию.
3. Повторять, пока промежуточное хранилище не опустеет:
   a) Если в промежуточном хранилище не осталось записей, которые можно было бы перенести в текущую серию, то:
   b) закрыть текущую серию и создать новую.
   c) Найти в промежуточном хранилище запись с наименьшим значением, превышающим значение последней записи в текущей серии.
   d) Скопировать эту запись в текущую серию.
   e) Удалить эту запись из промежуточного хранилища.
   f) Добавить в промежуточное хранилище следующую входную запись (если имеется).
4. Закрыть текущую серию.

Преимущество использования промежуточного хранилища заключается в возможности снова и снова добавлять в нее записи, а это означает, что всегда можно выбрать следующую запись в серию из пула кандидатов размером с блок. Таким образом, каждая серия, скорее всего, будет содержать больше записей, чем один блок.

В следующем примере сравниваются эти два способа создания начальных серий. Вернемся к предыдущему примеру сортировки записей STUDENT по значениям в поле `SId`. Допустим, что блок может содержать три записи, и записи изначально хранятся в следующем порядке:

```
2 6 20 4 1 16 19 3 18
```

Эти записи образуют четыре серии, как было показано выше. Применим алгоритм 13.1, чтобы уменьшить количество начальных серий. Он прочитает записи группами по три и отсортирует каждую группу по отдельности. В результате получится три начальные серии, как показано ниже:

```
Серия 1: 2 6 20
Серия 2: 1 4 16
Серия 3: 3 18 19
```

Теперь применим алгоритм 13.2. Сначала он прочитает три первые записи в промежуточное хранилище.

```
Промежуточное хранилище: 2 6 20
Серия 1:
```

Затем выберет запись с наименьшим значением (2), добавит ее в серию, удалит из промежуточного хранилища и прочитает следующую входную запись в промежуточное хранилище.

```
Промежуточное хранилище: 6 20 4
Серия 1: 2
```

Следующее наименьшее значение в промежуточном хранилище – 4. Алгоритм добавит эту запись в серию, удалит из промежуточного хранилища и прочитает следующую входную запись.

```
Промежуточное хранилище: 6 20 1
Серия 1: 2 4
```

Следующее наименьшее значение – 1, но оно слишком мало, чтобы стать частью текущей серии, поэтому выбирается следующее по величине значение 6. Алгоритм перенесет эту запись в серию и прочитает в промежуточное хранилище следующую входную запись.

```
Промежуточное хранилище: 20 1 16
Серия 1: 2 4 6
```

Продолжая цикл, алгоритм добавит в серию записи со значениями 16, 19 и 20 в поле `SId`. После этого промежуточное хранилище окажется заполнено записями, которые нельзя добавить в серию.

```
Промежуточное хранилище: 1 3 18
Серия 1: 2 4 6 16 19 20
```

Поэтому текущая серия закрывается, и создается новая. Поскольку входных записей для анализа не осталось, в эту новую серию будут помещены все три записи, оставшиеся в промежуточном хранилище.

```
Промежуточное хранилище:
Серия 1: 2 4 6 16 19 20
Серия 2: 1 3 18
```

Этот алгоритм создает всего две начальные серии. Первая серия занимает два блока.

Исключение операции записи получившейся отсортированной таблицы

Напомню, что каждая материализующая реализация имеет два этапа: этап предварительной обработки, в ходе которой исходные записи материализуются в одну или несколько временных таблиц, и этап сканирования, использующий временные таблицы для определения следующей выходной записи.

На этапе предварительной обработки простой алгоритм сортировки слиянием создает отсортированную временную таблицу, а на этапе сканирования читает ее. Это простая, но не оптимальная стратегия.

Пусть вместо этого этап предварительной обработки останавливается перед последней итерацией слияния до создания отсортированной временной таблицы, то есть когда количество временных таблиц ≤ `k`. Этап сканирования получает эти `k` таблиц и выполняет окончательное объединение. В частности, он открывает образы сканирования для всех `k` таблиц. Каждый вызов метода `next` проверит все текущие записи во всех этих образах и выберет запись с наименьшим значением в поле сортировки.

В каждый момент времени этап сканирования должен помнить, какому из `k` образов сканирования принадлежит текущая запись. Этот образ называется *текущим образом сканирования*. Когда клиент запросит следующую запись,

реализация перейдет к следующей записи в текущем образе, определит образ сканирования, содержащий наименьшую запись, и назначит его новым текущим образом сканирования.

В итоге задача этапа сканирования сводится к возврату записей в порядке сортировки, как если бы они хранились в одной отсортированной таблице. Однако для этого не требуется создавать общую временную таблицу. Вместо этого этап сканирования использует k таблиц, полученных от этапа предварительной обработки. Таким способом можно избавиться от необходимости обращаться к дисковым блокам, чтобы записать (и прочитать) окончательную отсортированную таблицу.

13.4.4. Стоимость сортировки слиянием

Рассчитаем стоимость сортировки по аналогии с расчетом стоимости оператора материализации. На рис. 13.4 показана структура дерева запроса, содержащего узел sort.

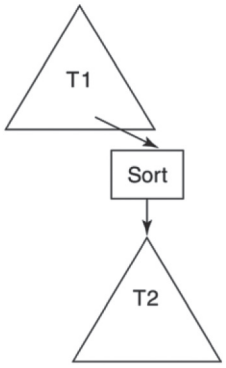


Рис. 13.4. Дерево запроса, содержащее узел sort

Стоимость сортировки состоит из двух слагаемых – стоимости предварительной обработки и стоимости сканирования:

- стоимость предварительной обработки равна стоимости подзапроса T2, плюс стоимость расщепления записей на серии, плюс стоимость всех итераций слияния, кроме последней;
- стоимость сканирования равна стоимости последнего объединения записей из временных таблиц.

Чтобы сделать рассуждения более предметными, допустим, что:

- в каждой итерации алгоритм объединяет k серий;
- всего имеется R начальных серий;
- для материализации входных записей требуется обратиться к B блокам.

На этапе расщепления каждый из блоков записывается ровно один раз, поэтому стоимость расщепления складывается из обращений к B блокам и стоимости чтения входных записей. Для сортировки записей требуется $\log_k R$ итераций. Одна из этих итераций будет выполняться на этапе сканирования, а остальные – на этапе предварительной обработки. Во время каждой итерации

на этапе предварительной обработки записи из каждой серии будут читаться и записываться один раз; то есть в каждой итерации потребуется 2B обращений к блокам. На этапе сканирования записи из каждой серии будут читаться один раз, что потребует B обращений к блокам. Объединив и упростив эти вычисления, получаем следующие формулы стоимости:

- Стоимость предварительной обработки = $2B \times \log_k R - B$ + стоимость чтения входных записей;
- Стоимость сканирования = B.

Рассмотрим конкретный пример. Допустим, что необходимо отсортировать хранимую таблицу, занимающую 1000 блоков, с начальными сериями длиной в 1 блок (то есть B = R = 1000). Поскольку таблица хранимая, стоимость чтения входных записей составит 1000 блоков. Если в каждой итерации объединять 2 серии, то для полной сортировки записей понадобится 10 итераций слияния (потому что $\log_2 1000 \leqslant 10$). Согласно приведенной выше формуле, для предварительной обработки записей потребуется 20 000 обращений к блокам, и еще 1000 для завершающего сканирования. Если в каждой итерации объединять по 10 серий (то есть k = 10), то всего потребуется 3 итерации, и для предварительной обработки – лишь 6000 обращений к блокам.

Продолжая пример, предположим, что в каждой итерации объединяется 1000 серий (то есть k = 1000). Тогда $\log_k R = 1$, и стоимость предварительной обработки составит B, плюс стоимость чтения входных записей, или 2000 обращений к блокам. Обратите внимание, что стоимость сортировки в этом случае идентична стоимости материализации. В частности, на этапе предварительной обработки не потребуется выполнять объединение, потому что этап расщепления уже даст k серий. Поэтому стоимость предварительной обработки будет равна стоимости чтения таблицы и расщепления записей, или 2B обращений к блокам.

13.4.5. Реализация сортировки слиянием

Оператор сортировки в SimpleDB реализуют классы SortPlan и SortScan.

Класс SortPlan

Определение класса SortPlan приводится в листинге 13.3.

Листинг 13.3. Определение класса SortPlan в SimpleDB

```
public class SortPlan implements Plan {
    private Plan p;
    private Transaction tx;
    private Schema sch;
    private RecordComparator comp;

 public SortPlan(Plan p, List<String> sortfields, Transaction tx) {
        this.p = p;
        this.tx = tx;
        sch = p.schema();
        comp = new RecordComparator(sortfields);
    }
```

```
public Scan open() {
    Scan src = p.open();
    List<TempTable> runs = splitIntoRuns(src);
    src.close();
    while (runs.size() > 2)
        runs = doAMergeIteration(runs);
    return new SortScan(runs, comp);
}

public int blocksAccessed() {
    // не включает стоимость однократной сортировки
    Plan mp = new MaterializePlan(tx, p);
    return mp.blocksAccessed();
}

public int recordsOutput() {
    return p.recordsOutput();
}

public int distinctValues(String fldname) {
    return p.distinctValues(fldname);
}

public Schema schema() {
    return sch;
}

private List<TempTable> splitIntoRuns(Scan src) {
    List<TempTable> temps = new ArrayList<>();
    src.beforeFirst();
    if (!src.next())
        return temps;
    TempTable currenttemp = new TempTable(tx, sch);
    temps.add(currenttemp);
    UpdateScan currentscan = currenttemp.open();
    while (copy(src, currentscan))
        if (comp.compare(src, currentscan) < 0) {
        // создать новую серию
        currentscan.close();
        currenttemp = new TempTable(tx, sch);
        temps.add(currenttemp);
        currentscan = (UpdateScan) currenttemp.open();
    }
    currentscan.close();
    return temps;
}

private List<TempTable> doAMergeIteration(List<TempTable> runs) {
    List<TempTable> result = new ArrayList<>();
    while (runs.size() > 1) {
        TempTable p1 = runs.remove(0);
        TempTable p2 = runs.remove(0);
        result.add(mergeTwoRuns(p1, p2));
    }
    if (runs.size() == 1)
        result.add(runs.get(0));
    return result;
}
```

```
    private TempTable mergeTwoRuns(TempTable p1, TempTable p2) {
        Scan src1 = p1.open();
        Scan src2 = p2.open();
        TempTable result = new TempTable(tx, sch);
        UpdateScan dest = result.open();

        boolean hasmore1 = src1.next();
        boolean hasmore2 = src2.next();
        while (hasmore1 && hasmore2)
            if (comp.compare(src1, src2) < 0)
                hasmore1 = copy(src1, dest);
            else
                hasmore2 = copy(src2, dest);

        if (hasmore1)
            while (hasmore1)
                hasmore1 = copy(src1, dest);
        else
            while (hasmore2)
                hasmore2 = copy(src2, dest);

        src1.close();
        src2.close();
        dest.close();
        return result;
    }

    private boolean copy(Scan src, UpdateScan dest) {
        dest.insert();
        for (String fldname : sch.fields())
            dest.setVal(fldname, src.getVal(fldname));
        return src.next();
    }
}
```

Метод `open` выполняет алгоритм сортировки слиянием. В каждой итерации он объединяет две серии (т. е. `k` = 2) и не пытается уменьшить количество начальных серий. (В упражнениях с 13.10 по 13.13 вам будет предложено добавить эти улучшения.)

Приватный метод `splitIntoRuns` выполняет этап расщепления в алгоритме сортировки слиянием, а метод `doAMergeIteration` – одну итерацию этапа слияния; этот метод вызывается несколько раз, пока не останется одна или две серии. После этого `open` передает список серий конструктору `SortScan`, который выполнит последнюю итерацию слияния.

Сначала метод `splitIntoRuns` создает временную таблицу и открывает ее образ сканирования («образ назначения»). Затем выполняет итерации по записям во входном образе сканирования. Каждая входная запись вставляется в образ назначения. Каждый раз, когда создается новая серия, образ назначения закрывается, после чего создается и открывается другая временная таблица. К завершению этого метода будет создано несколько временных таблиц, по одной серии в каждой.

Метод doAMergeIteration получает список текущих временных таблиц. Для каждой пары таблиц из этого списка он вызывает метод mergeTwoRuns и возвращает список, содержащий объединенные временные таблицы.

Метод mergeTwoRuns открывает образы сканирования обеих таблиц и создает временную таблицу для сохранения результата. Он последовательно выбирает из входных образов запись с наименьшим значением и копирует ее в результат. Достигнув конца одного из образов, метод просто добавляет в результат оставшиеся записи из другого образа.

Стоимость методов определяется просто. Методы recordsOutput и distinctValues возвращают тот же результат, что и одноименные методы входной таблицы, потому что отсортированная таблица содержит те же записи и характеризуется тем же распределением значений. Метод blocksAccessed оценивает количество обращений к блокам в итерации отсортированного образа сканирования как равное количеству блоков в отсортированной таблице. Поскольку отсортированные и материализованные таблицы имеют одинаковый размер, выполнение вычислений делегируется классу MaterializePlan. Для этого метод создает «фиктивный» материализованный план с единственной целью – вызвать его метод blocksAccessed. Стоимость предварительной обработки не учитывается методом blocksAccessed по тем же причинам, что и в MaterializePlan.

Сравнение записей выполняется классом RecordComparator, определение которого приводится в листинге 13.4. Класс сравнивает текущие записи из двух образов сканирования. Его метод compare перебирает поля сортировки в текущих записях двух образов сканирования и сравнивает их значения с помощью compareTo. Если все значения равны, тогда compareTo возвращает 0.

Листинг 13.4. Определение класса RecordComparator в SimpleDB

```
public class RecordComparator implements Comparator<Scan> {
    private Collection<String> fields;

    public RecordComparator(Collection<String> fields) {
        this.fields = fields;
    }

    public int compare(Scan s1, Scan s2) {
        for (String fldname : fields) {
            Constant val1 = s1.getVal(fldname);
            Constant val2 = s2.getVal(fldname);
            int result = val1.compareTo(val2);
            if (result != 0)
                return result;
        }
        return 0;
    }
}
```

Класс SortScan

Класс SortScan реализует образ сканирования; его определение показано в листинге 13.5. Конструктор принимает список с одной или двумя сериями; инициализирует их, открывая соответствующие таблицы, и переходит к первым записям. (Если в списке имеется только одна серия, то переменной hasmore2 присваивается значение false и вторая серия не учитывается.)

Листинг 13.5. Определение класса SortScan в SimpleDB

```
public class SortScan implements Scan {
    private UpdateScan s1, s2=null, currentscan=null;
    private RecordComparator comp;
    private boolean hasmore1, hasmore2=false;
    private List<RID> savedposition;

    public SortScan(List<TempTable> runs, RecordComparator comp) {
        this.comp = comp;
        s1 = (UpdateScan) runs.get(0).open();
        hasmore1 = s1.next();
        if (runs.size() > 1) {
            s2 = (UpdateScan) runs.get(1).open();
            hasmore2 = s2.next();
        }
    }

    public void beforeFirst() {
        s1.beforeFirst();
        hasmore1 = s1.next();
        if (s2 != null) {
            s2.beforeFirst();
            hasmore2 = s2.next();
        }
    }

    public boolean next() {
        if (currentscan == s1)
            hasmore1 = s1.next();
        else if (currentscan == s2)
            hasmore2 = s2.next();

        if (!hasmore1 && !hasmore2)
            return false;
        else if (hasmore1 && hasmore2) {
            if (comp.compare(s1, s2) < 0)
                currentscan = s1;
            else
                currentscan = s2;
        }
        else if (hasmore1)
            currentscan = s1;
        else if (hasmore2)
            currentscan = s2;
        return true;
    }

    public void close() {
        s1.close();
        if (s2 != null)
            s2.close();
    }

    public Constant getVal(String fldname) {
        return currentscan.getVal(fldname);
    }
```

```
    public int getInt(String fldname) {
        return currentscan.getInt(fldname);
    }

    public String getString(String fldname) {
        return currentscan.getString(fldname);
    }

    public boolean hasField(String fldname) {
        return currentscan.hasField(fldname);
    }

    public void savePosition() {
        RID rid1 = s1.getRid();
        RID rid2 = s2.getRid();
        savedposition = Arrays.asList(rid1,rid2);
    }

    public void restorePosition() {
        RID rid1 = savedposition.get(0);
        RID rid2 = savedposition.get(1);
        s1.moveToRid(rid1);
        s2.moveToRid(rid2);
    }
}
```

Переменная `currenttscan` ссылается на образ сканирования, содержащий самую последнюю запись в слиянии. Методы `get` получают свои значения из этого образа. Метод `next` переходит к следующей записи в текущем образе, а затем из двух образов сканирования выбирает запись с наименьшим значением. После этого переменной `currenttscan` присваивается ссылка на этот образ.

Класс также имеет два общедоступных метода: `savePosition` и `restorePosition`. Эти методы позволяют клиенту (в частности, в реализации образа сканирования оператора соединения слиянием, который описывается в разделе 13.6) вернуться к ранее просмотренной записи и продолжить сканирование, начав с нее.

13.5. Группировка и агрегирование

Оператор реляционной алгебры *группировка (groupby)* принимает три аргумента: входную таблицу, множество полей группировки и множество выражений агрегирования. Он организует входные записи в группы, объединяя записи с одинаковыми значениями в полях группировки. Выходная таблица содержит одну запись для каждой группы; запись включает поля группировки и результаты вычисления выражений агрегирования.

Например, следующий запрос вернет самый ранний и самый поздний год, когда университет выпускал студентов, обучавшихся по каждой основной специальности.

```
groupby (STUDENT, {MajorID}, {Min(GradYear), Max(GradYear)})
```

В табл. 13.1 показаны результаты этого запроса для таблицы STUDENT в табл. 1.1.

Таблица 13.1. Результаты запроса с оператором группировки

MajorId	MinOfGradYear	MaxOfGradYear
10	2021	2022
20	2019	2022
30	2020	2021

В общем случае выражение агрегирования определяет *агрегатную функцию* и поле. В запросе выше выражение агрегирования `Min(GradYear)` возвращает минимальное значение `GradYear` для записей в группе. В число доступных агрегатных функций в SQL входят: `MIN`, `MAX`, `COUNT`, `SUM` и `AVG`.

Основная сложность реализации оператора группировки заключается в создании групп записей. Лучшее решение – создать временную таблицу с записями, отсортированными по полям группировки. Записи, принадлежащие одной группе, будут находиться в такой таблице рядом друг с другом, благодаря чему реализация сможет вычислить информацию о каждой группе, выполнив всего один проход по отсортированной таблице. Порядок действий описан в алгоритме 13.3.

Алгоритм 13.3. Агрегирование

1. Создать временную таблицу с входными записями, отсортированными по полям группировки.
2. Переместиться к первой записи в таблице.
3. Повторять до исчерпания временной таблицы:
   a) Принять за «значение группы» значения полей группировки в текущей записи.
   b) Для каждой записи, в которой значения полей группировки совпадают со значением группы:
      прочитать запись в список группы.
   c) Вычислить соответствующую агрегатную функцию для записей в списке группы.

Стоимость алгоритма агрегирования складывается из стоимости предварительной обработки и стоимости сканирования. Вычислить эти затраты просто. Стоимость предварительной обработки – это стоимость сортировки, а стоимость сканирования – стоимость одной итерации по отсортированным записям. Иначе говоря, оператор группировки имеет ту же стоимость, что и оператор сортировки.

Оператор группировки в SimpleDB реализуют классы `GroupByPlan` и `GroupByScan`; их определения показаны в листингах 13.6 и 13.7.

Листинг 13.6. Определение класса GroupByPlan в SimpleDB

```
public class GroupByPlan implements Plan {
    private Plan p;
    private List<String> groupfields;
    private List<AggregationFn> aggfns;
    private Schema sch = new Schema();
```

```
    public GroupByPlan(Transaction tx, Plan p, List<String> groupfields,
                       List<AggregationFn> aggfns) {
        this.p = new SortPlan(tx, p, groupfields);
        this.groupfields = groupfields;
        this.aggfns = aggfns;
        for (String fldname : groupfields)
            sch.add(fldname, p.schema());
        for (AggregationFn fn : aggfns)
            sch.addIntField(fn.fieldName());
    }

    public Scan open() {
        Scan s = p.open();
        return new GroupByScan(s, groupfields, aggfns);
    }

    public int blocksAccessed() {
        return p.blocksAccessed();
    }

    public int recordsOutput() {
        int numgroups = 1;
        for (String fldname : groupfields)
            numgroups *= p.distinctValues(fldname);
        return numgroups;
    }

    public int distinctValues(String fldname) {
        if (p.schema().hasField(fldname))
            return p.distinctValues(fldname);
        else
            return recordsOutput();
    }

    public Schema schema() {
        return sch;
    }
}
```

Листинг 13.7. Определение класса GroupByScan в SimpleDB

```
public class GroupByScan implements Scan {
    private Scan s;
    private List<String> groupfields;
    private List<AggregationFn> aggfns;
    private GroupValue groupval;
    private boolean moregroups;
    public GroupByScan(Scan s, List<String> groupfields, List<AggregationFn> aggfns) {

        this.s = s;
        this.groupfields = groupfields;
        this.aggfns = aggfns;
        beforeFirst();
    }
```

```
    public void beforeFirst() {
        s.beforeFirst();
        moregroups = s.next();
    }

    public boolean next() {
        if (!moregroups)
            return false;
        for (AggregationFn fn : aggfns)
            fn.processFirst(s);
        groupval = new GroupValue(s, groupfields);
        while(moregroups = s.next()) {
            GroupValue gv = new GroupValue(s, groupfields);
            if (!groupval.equals(gv))
                break;
            for (AggregationFn fn : aggfns)
                fn.processNext(s);
        }
        return true;
    }

    public void close() {
        s.close();
    }

    public Constant getVal(String fldname) {
        if (groupfields.contains(fldname))
            return groupval.getVal(fldname);
        for (AggregationFn fn : aggfns)
            if (fn.fieldName().equals(fldname))
                return fn.value();
        throw new RuntimeException("no field " + fldname)
    }

    public int getInt(String fldname) {
        return getVal(fldname).asInt();
    }

    public String getString(String fldname) {
        return getVal(fldname).asString();
    }

    public boolean hasField(String fldname) {
        if (groupfields.contains(fldname))
            return true;
        for (AggregationFn fn : aggfns)
            if (fn.fieldName().equals(fldname))
                return true;
        return false;
    }
}
```

Метод `open` в `GroupByPlan` создает и открывает план сортировки для входных записей. Полученный в результате образ сканирования передается в конструктор `GroupByScan`. Образ сканирования оператора группировки читает записи из образа сканирования оператора сортировки по мере необходимости. В част-

ности, каждый вызов метода next читает записи, принадлежащие следующей группе. Конец группы определяется этим методом по первой встретившейся записи, принадлежащей другой группе (или когда обнаруживается, что в образе сканирования оператора сортировки больше нет записей); поэтому каждый раз, когда вызывается next, текущей в базовом образе сканирования всегда оказывается первая запись, принадлежащая следующей группе.

Класс GroupValue хранит информацию о текущей группе; его определение показано в листинге 13.8. Его конструктор принимает образ сканирования вместе с полями группировки. Значения полей в текущей записи определяют группу. Метод getVal возвращает значение указанного поля. Метод equals возвращает true, если два объекта GroupValue имеют одинаковые значения полей группировки, а метод hashCode присваивает хеш-значение каждому объекту GroupValue.

Листинг 13.8. Определение класса GroupValue в SimpleDB

```
public class GroupValue {
    private Map<String,Constant> vals = new HashMap<>();

    public GroupValue(Scan s, List<String> fields) {
        for (String fldname : fields)
            vals.put(fldname, s.getVal(fldname));
    }

    public Constant getVal(String fldname) {
        return vals.get(fldname);
    }

    public boolean equals(Object obj) {
        GroupValue gv = (GroupValue) obj;
        for (String fldname : vals.keySet()) {
            Constant v1 = vals.get(fldname);
            Constant v2 = gv.getVal(fldname);
            if (!v1.equals(v2))
                return false;
        }
        return true;
    }

    public int hashCode() {
        int hashval = 0;
        for (Constant c : vals.values())
            hashval += c.hashCode();
        return hashval;
    }
}
```

Агрегатные функции (такие как MIN, COUNT и др.) в SimpleDB реализованы в виде классов. Экземпляр класса отвечает за хранение соответствующей информации о записях в группе, за вычисление агрегатного значения для этой группы и за определение имени вычисляемого поля. Эти методы определяются интерфейсом AggregationFn, как показано в листинге 13.9. Метод processFirst начинает новую группу, используя текущую запись как первую запись этой группы. Метод processNext добавляет еще одну запись в существующую группу.

Листинг 13.9. Определение интерфейса AggregationFn в SimpleDB

```
public interface AggregationFn {
    void processFirst(Scan s);
    void processNext(Scan s);
    String fieldName();
    Constant value();
}
```

В листинге 13.10 представлен пример класса `MaxFn`, реализующего агрегатную функцию `MAX`. Клиент передает имя агрегатного поля в конструктор. Объект использует это имя для проверки значения поля в каждой записи внутри группы и сохраняет максимальное значение в своей переменной `val`.

Листинг 13.10. Определение класса MaxFn в SimpleDB

```
public class MaxFn implements AggregationFn {
    private String fldname;
    private Constant val;

    public MaxFn(String fldname) {
        this.fldname = fldname;
    }

    public void processFirst(Scan s) {
        val = s.getVal(fldname);
    }

    public void processNext(Scan s) {
        Constant newval = s.getVal(fldname);
        if (newval.compareTo(val) > 0)
            val = newval;
    }

    public String fieldName() {
        return "maxof" + fldname;
    }

    public Constant value() {
        return val;
    }
}
```

13.6. Соединение слиянием

В главе 12 был разработан эффективный оператор соединения двух таблиц с использованием индексов (indexjoin), использующий предикат соединения в форме «A = B», где A – поле из таблицы слева, а B – из таблицы справа. Эти поля называются *полями соединения*. Оператор соединения с помощью индекса можно использовать, только когда таблица справа – хранимая и на ее поле соединения построен индекс. В этом разделе рассматривается эффективный оператор *соединения слиянием (mergejoin)*, который применим к любым таблицам. Порядок его работы описан в алгоритме 13.4.

Алгоритм 13.4. Соединение слиянием

1. Для каждой входной таблицы:
   отсортировать таблицу, используя поле соединения в качестве ключа сортировки.
2. Сканировать отсортированные таблицы параллельно, отыскивая совпадения в полях соединения.

Рассмотрим шаг 2 алгоритма. Если предположить, что таблица в соединении слева не имеет повторяющихся значений в своем поле соединения, то алгоритм действует подобно сканированию прямого произведения. То есть сканирует левую таблицу только один раз и для каждой записи слева отыскивает соответствующие записи справа. Однако тот факт, что записи отсортированы, значительно упрощает поиск. В частности, обратите внимание, что:

- соответствующие записи справа должны следовать за записями, соответствующими предыдущей записи слева;
- соответствующие записи следуют в таблице друг за другом.

Как следствие после перехода к следующей записи слева достаточно продолжить сканирование таблицы справа с того места, где оно перед этим было остановлено, и вновь остановить по достижении значения поля соединения, превышающего значение поля соединения слева. То есть правая таблица должна быть отсканирована только один раз.

13.6.1. Пример соединения слиянием

Следующий запрос применяет оператор соединения слиянием к таблицам DEPT и STUDENT.

```
mergejoin(DEPT, STUDENT, DId=MajorId)
```

На первом шаге алгоритм соединения слиянием создает временные таблицы для хранения содержимого DEPT и STUDENT, отсортированного по полям `DId` и `MajorId` соответственно. Эти отсортированные таблицы показаны в табл. 13.2. Основой послужили записи из табл. 1.1, дополненные новой кафедрой Basketry (`DId` = 18).

На втором шаге алгоритм сканирует отсортированные таблицы. Первая запись в DEPT имеет значение `DId` = 10. Для этой записи выполняется сканирование таблицы STUDENT и обнаруживается совпадение с первыми тремя записями. После перехода к четвертой записи (для Amy) алгоритм обнаруживает другое значение в поле `MajorId` и понимает, что обработка кафедры с идентификатором 10 завершена. Он перемещается к следующей записи в DEPT (соответствующей кафедре Basketry) и сравнивает ее значение `DId` со значением `MajorId` в текущей записи в STUDENT (т. е. Amy). Поскольку значение `MajorId` для Amy больше, алгоритм понимает, что для этой кафедры нет совпадений, и переходит к следующей записи в DEPT (соответствующей кафедре Math). Эта запись соответствует записи для Amy, а также следующим трем записям в STUDENT. Перебирая записи в STUDENT, алгоритм достигает записи для Bob, которая не соответствует текущей кафедре. Поэтому он переходит к следующей записи в DEPT (соответствующей кафедре Drama) и продолжает поиск

в STUDENT, где находит совпадение с записями для Bob и Art. Операция соединения завершается сразу после исчерпания всех записей в одной из таблиц.

Таблица 13.2. Отсортированные таблицы DEPT и STUDENT

DEPT	Did	DName
	10	compsci
	18	basketry
	20	math
	30	drama

STUDENT	SId	SName	MajorId	GradYear
	1	joe	10	2021
	3	max	10	2022
	9	lee	10	2021
	2	amy	20	2020
	4	sue	20	2022
	6	kim	20	2020
	8	pat	20	2019
	5	bob	30	2020
	7	art	30	2021

Что случится, если в таблице слева обнаружатся повторяющиеся значения в поле соединения? Вспомните, что алгоритм переходит к следующей записи слева, когда обнаруживает, что следующая запись, прочитанная из таблицы справа, больше не соответствует значению в поле соединения. Если следующая запись в таблице слева имеет такое же значение, то алгоритм должен вернуться к первой соответствующей записи справа, то есть повторно прочитать все блоки справа, содержащие совпадающие записи, что увеличивает стоимость соединения.

К счастью, ситуации с повторяющимися значениями слева редки. В большинстве случаев соединение таблиц выполняется по полям, служащим ключами и внешними ключами. Так, в примере соединения выше поле `DId` является ключом таблицы DEPT, а `MajorId` – внешним ключом таблицы STUDENT. Поскольку ключи и внешние ключи объявляются при создании таблицы, планировщик запросов может использовать эту информацию и убедиться, что таблица, имеющая ключ, находится слева в соединении.

Теперь оценим стоимость алгоритма соединения слиянием. Обратите внимание, что этап предварительной обработки сортирует обе входные таблицы, а этап сканирования выполняет обход отсортированных таблиц. Если в таблице слева нет повторяющихся значений, то обе таблицы сканируются один раз и стоимость соединения складывается из стоимостей двух операций сортировки. Если в таблице слева имеются повторяющиеся значения, то при сканировании таблицы справа соответствующие записи будут прочитаны несколько раз.

Например, определим стоимость соединения слиянием таблиц DEPT и STUDENT, используя статистику из табл. 7.2. Предположим, что алгоритм

объединяет пары серий и каждая начальная серия имеет длину в 1 блок. Стоимость предварительной обработки включает сортировку таблицы STUDENT из 4500 блоков ($9000 \times \log_2(4500) - 4500 = 112\,500$ обращений к блокам, плюс 4500 обращений для чтения исходной таблицы) и таблицы DEPT из 2 блоков ($4 \times \log_2(2) - 2 = 2$ обращения к блокам, плюс 2 обращения для чтения исходной таблицы). То есть суммарная стоимость предварительной обработки составляет 117 004 блока. Стоимость сканирования определяется как сумма размеров отсортированных таблиц, которая составляет 4502 блока. Таким образом, общая стоимость соединения составляет 121 506 обращений к блокам.

Сравните эту стоимость со стоимостью соединения, выполняемого путем прямого произведения с последующей селекцией, как было показано в главе 8. Стоимость этого способа соединения вычисляется по формуле `B1 + R1 × B2`, которая дает 184 500 обращений к блокам.

13.6.2. Реализация оператора соединения слиянием

Алгоритм соединения слиянием в SimpleDB реализуют классы `MergeJoinPlan` и `MergeJoinScan`.

Класс MergeJoinPlan

Определение класса `MergeJoinPlan` показано в листинге 13.11. Метод `open` открывает образ сканирования оператора сортировки для каждой из входных таблиц, используя указанные поля соединения, и передает их в конструктор `MergeJoinScan`.

Листинг 13.11. Определение класса MergeJoinPlan в SimpleDB

```
public class MergeJoinPlan implements Plan {
    private Plan p1, p2;
    private String fldname1, fldname2;
    private Schema sch = new Schema();

    public MergeJoinPlan(Transaction tx, Plan p1, Plan p2,
                         String fldname1, String fldname2) {

        this.fldname1 = fldname1;
        List<String> sortlist1 = Arrays.asList(fldname1);
        this.p1 = new SortPlan(tx, p1, sortlist1);

        this.fldname2 = fldname2;
        List<String> sortlist2 = Arrays.asList(fldname2);
        this.p2 = new SortPlan(tx, p2, sortlist2);

        sch.addAll(p1.schema());
        sch.addAll(p2.schema());
    }

 public Scan open() {
        Scan s1 = p1.open();
        SortScan s2 = (SortScan) p2.open();
        return new MergeJoinScan(s1, s2, fldname1, fldname2);
    }
```

```
    public int blocksAccessed() {
        return p1.blocksAccessed() + p2.blocksAccessed();
    }

    public int recordsOutput() {
        int maxvals = Math.max(p1.distinctValues(fldname1),
                               p2.distinctValues(fldname2));
        return (p1.recordsOutput()* p2.recordsOutput()) / maxvals;
    }

    public int distinctValues(String fldname) {
        if (p1.schema().hasField(fldname))
            return p1.distinctValues(fldname);
        else
            return p2.distinctValues(fldname);
    }

    public Schema schema() {
        return sch;
    }
}
```

Метод `blocksAccessed` предполагает, что обход каждого образа будет выполнен только один раз. Идея состоит в том, что даже если в таблице слева имеются повторяющиеся значения, соответствующие записи из таблицы справа будут находиться в том же или в недавно использованном блоке, и весьма вероятно, что для повторного обхода этих записей потребуется очень мало (возможно, ноль) дополнительных обращений к дисковым блокам.

Метод `recordsOutput` вычисляет количество записей в соединении как число записей в прямом произведении, деленное на число записей, отфильтрованных предикатом соединения. Код метода `distinctValues` прост. Поскольку соединение не увеличивает и не уменьшает количество уникальных значений полей, результат соответствует оценке в базовом запросе.

Класс MergeJoinScan

Определение класса `MergeJoinScan` показано в листинге 13.12. Основную работу по поиску совпадений выполняет метод `next`, используя переменную класса `joinval`, где хранится самое последнее значение для соединения. Сразу после вызова метод `next` читает следующую запись справа. Если значение поля соединения этой записи равно `joinval`, то совпадение найдено, и метод возвращает управление. Если нет, то метод переходит к следующей записи слева. Если значение поля соединения в этой записи снова равно `joinval`, значит, встречено повторяющееся значение. В таком случае метод переходит к первой записи в образе сканирования справа, имеющей это значение, и возвращает управление. В противном случае `next` последовательно читает записи из образа сканирования, имеющего наименьшее значение поля соединения, пока не найдет совпадение или не достигнет конца образа. Если совпадение найдено, переменной `joinval` присваивается текущее значение поля соединения и сохраняется текущая позиция в образе справа. Если достигнут конец образа, метод `next` возвращает `false`.

Листинг 13.12. Определение класса MergeJoinScan в SimpleDB

```
public class MergeJoinScan implements Scan {
    private Scan s1;
    private SortScan s2;
    private String fldname1, fldname2;
    private Constant joinval = null;

    public MergeJoinScan(Scan s1, SortScan s2, String fldname1, String fldname2) {
        this.s1 = s1;
        this.s2 = s2;
        this.fldname1 = fldname1;
        this.fldname2 = fldname2;
        beforeFirst();
    }

    public void close() {
        s1.close();
        s2.close();
    }

    public void beforeFirst() {
        s1.beforeFirst();
        s2.beforeFirst();
    }

    public boolean next() {
        boolean hasmore2 = s2.next();
        if (hasmore2 && s2.getVal(fldname2).equals(joinval))
            return true;

        boolean hasmore1 = s1.next();
        if (hasmore1 && s1.getVal(fldname1).equals(joinval)) {
            s2.restorePosition();
            return true;
        }

        while (hasmore1 && hasmore2) {
            Constant v1 = s1.getVal(fldname1);
            Constant v2 = s2.getVal(fldname2);
            if (v1.compareTo(v2) < 0)
                hasmore1 = s1.next();
            else if (v1.compareTo(v2) > 0)
                hasmore2 = s2.next();
            else {
                s2.savePosition();
                joinval = s2.getVal(fldname2);
                return true;
            }
        }
        return false;
    }
```

```
    public int getInt(String fldname) {
        if (s1.hasField(fldname))
            return s1.getInt(fldname);
        else
            return s2.getInt(fldname);
    }

    public String getString(String fldname) {
        if (s1.hasField(fldname))
            return s1.getString(fldname);
        else
            return s2.getString(fldname);
    }

    public Constant getVal(String fldname) {
        if (s1.hasField(fldname))
            return s1.getVal(fldname);
        else
            return s2.getVal(fldname);
    }

    public boolean hasField(String fldname) {
        return s1.hasField(fldname) || s2.hasField(fldname);
    }
}
```

13.7. Итоги

- *Материализующая реализация* оператора предварительно обрабатывает исходные записи, сохраняя их в одной или в нескольких временных таблицах. Благодаря этому сканирование выполняется эффективнее, потому что требуется всего лишь просмотреть временные таблицы.
- Материализующие реализации вычисляют свои исходные данные один раз и могут использовать преимущества сортировки. Однако они должны вычислять всю входную таблицу, даже если пользователя интересуют только некоторые записи. Написать материализующую реализацию можно для любого реляционного оператора, но такая реализация будет полезна, только если стоимость ее предварительной обработки компенсируется экономией на сканирование результата.
- Оператор *материализации* создает временную таблицу, содержащую все входные записи. Это особенно полезно, когда обход входных данных выполняется многократно, например когда они находятся справа от узла прямого произведения product.
- Для сортировки записей во временную таблицу система баз данных использует алгоритм *внешней* сортировки. Самый простой и распространенный алгоритм внешней сортировки называется *сортировкой слиянием*. Этот алгоритм расщепляет входные записи на серии, а затем последовательно объединяет серии до получения отсортированного набора записей.

- Сортировка слиянием действует тем эффективнее, чем меньше число начальных серий. Один из простых подходов заключается в создании начальных серий длиной в один блок путем чтения входных записей в блок и применения к этому блоку алгоритма внутренней сортировки. Другой подход заключается в чтении входных записей в одноблочное *промежуточное хранилище* и построении серий путем многократного выбора записи с наименьшим значением в этом хранилище.
- Также сортировка слиянием действует тем эффективнее, чем больше серий объединяется в одной итерации, поскольку при этом требуется меньше итераций. Для размещения каждой серии при объединении необходим буфер, поэтому максимальное количество серий ограничено количеством доступных буферов.
- Сортировка слиянием требует $2B \times \log_{k(R)} {}_{-B}$ обращений к блокам (плюс стоимость чтения входных записей) для предварительной обработки входных данных, где `B` – количество блоков, необходимых для хранения отсортированной таблицы, `R` – количество начальных серий и `k` – количество серий, объединяемых в одной итерации.
- Реализация оператора *группировки* сортирует записи по полям группировки, чтобы записи, принадлежащие одной группе, оказались рядом друг с другом. Затем он вычисляет агрегатные значения для каждой группы, выполняя один проход по отсортированным записям.
- Алгоритм *соединения слиянием* реализует соединение двух таблиц. Сначала он сортирует обе таблицы по полю соединения. Затем сканирует отсортированные таблицы параллельно. Каждый вызов метода `next` переходит к следующей записи в образе сканирования с меньшим значением.

13.8. Для дополнительного чтения

Сортировка файлов оставалась важной (и даже важнейшей) операцией на протяжении всей долгой истории вычислений, предшествовавшей системам баз данных. Существует масса литературы, посвященной этому вопросу, и многочисленные варианты сортировки слиянием, которые здесь не рассматривались. Подробный обзор различных алгоритмов представлен в Knuth (1998).

Класс `SortPlan` в SimpleDB представляет простую реализацию алгоритма сортировки слиянием. В статье Graefe (2006) описывается несколько интересных и полезных методов улучшения этой реализации.

В статье Graefe (2003) исследуется общность алгоритмов сортировки и B-деревьев. В ней показано, как использовать B-дерево для хранения промежуточных серий в сортировке слиянием и как с помощью итераций слияния создать B-дерево индекса для существующей таблицы.

В статье Graefe (1993) обсуждаются материализующие алгоритмы и приводится сравнение с нематериализующими алгоритмами.

Graefe, G. (1993) «Query evaluation techniques for large databases». *ACM Computing Surveys*, 25 (2), 73–170.

Graefe, G. (2003) «Sorting and indexing with partitioned B-trees». *Proceedings of the CIDR Conference*.

Graefe, G. (2006) «Implementing sorting in database systems». *ACM Computing Surveys*, 38 (3), 1–37.

Knuth, D. (1998) «The art of computer programming, Vol 3: Sorting and searching». Addison-Wesley[1].

13.9. Упражнения

Теория

13.1. Рассмотрите дерево запроса на рис. 13.1b.

a) Если предположить, что в 2005 году был выпущен только один студент, даст ли экономию правый узел materialize?
b) Если предположить, что в 2005 году было выпущено два студента, даст ли экономию правый узел materialize?
c) Допустим, что правое и левое поддеревья узла product поменялись местами. Вычислите экономию, которую даст материализация нового правого узла select.

13.2. Базовый алгоритм сортировки слиянием, описанный в разделе 13.4, объединяет серии итеративно. В примере, показанном в этом разделе, алгоритм объединил серии 1 и 2 в серию 5 и серии 3 и 4 – в серию 6; затем он объединил серии 5 и 6, получив конечную серию. Предположим теперь, что алгоритм объединяет серии последовательно. То есть сначала объединяются серии 1 и 2 в серию 5, затем серии 3 и 5 – в серию 6, а потом серии 4 и 6 – в конечную серию.

a) Объясните, почему для создания конечной серии таким «последовательным объединением» всегда требуется ровно то же количество объединений, что и при итеративном объединении.
b) Объясните, почему последовательное объединение требует больше (и обычно намного) обращений к блокам, чем итеративное объединение.

13.3. Рассмотрите алгоритмы 13.1 и 13.2 создания серий.

a) Если предположить, что входные записи уже отсортированы, какой алгоритм создаст меньше начальных серий? Объясните.
b) Предположим, что входные записи отсортированы в обратном порядке. Объясните, почему алгоритмы создадут то же число начальных серий.

13.4. Рассмотрите университетскую базу данных и ее статистики в табл. 7.2.

a) Оцените стоимость сортировки для каждой таблицы при использовании 2, 10 или 100 вспомогательных таблиц. Предположите, что каждая начальная серия имеет длину в один блок.
b) Для каждой пары таблиц, которую можно осмысленно соединить, оцените стоимость соединения слиянием (опять же, при условии использования 2, 10 или 100 вспомогательных таблиц).

---

[1] *Дональд Э. Кнут.* Искусство программирования. Т. 3: Сортировка и поиск. 2-е изд. Вильямс, 2017. ISBN 978-5-8459-0082-1. – *Прим. перев.*

13.5. Метод `splitIntoRuns` в классе `SortPlan` возвращает список объектов `TempTable`. Если база данных очень большая, этот список может получиться довольно длинным.

a) Объясните, как этот список может неожиданно ухудшить эффективность.
b) Предложите лучшее решение.

Практика

13.6. В разделе 13.4 описана нематериализующая реализация сортировки.

a) Спроектируйте и реализуйте классы `NMSortPlan` и `NMSortScan`, предоставляющие доступ к записям в порядке сортировки без создания временных таблиц.
b) Сколько обращений к блокам потребуется для полного сканирования результатов?
c) Допустим, что JDBC-клиент решил найти запись с наименьшим значением в некотором поле; для этого он выполняет запрос, который сортирует таблицу по этому полю, а затем выбирает первую запись. Сравните количество обращений к блокам, необходимых для этого, при использовании материализующей и нематериализующей реализаций.

13.7. Когда сервер перезапускается, имена временных таблиц снова будут начинаться с 0. Конструктор диспетчера файлов SimpleDB удаляет все временные файлы.

a) Объясните, какая проблема возникнет в SimpleDB, если файлы временных таблиц не будут удалены после перезапуска системы.
b) Временные файлы могут удаляться не после перезапуска системы, а сразу после фиксации транзакции, создавшей их. Добавьте необходимый для этого код в SimpleDB.

13.8. Какая проблема возникнет, если предложить классам `SortPlan` и `SortScan` отсортировать пустую таблицу? Исправьте эту проблему.

13.9. Добавьте в интерфейс `Plan` в SimpleDB (и во все классы, реализующие его) метод `preprocessingCost`, который оценивает стоимость материализации таблицы. Измените другие формулы оценки соответствующим образом.

13.10. Измените реализацию `SortPlan` так, чтобы она создавала начальные серии длиной в один блок, как описано в алгоритме 13.1.

13.11. Измените реализацию `SortPlan` так, чтобы она создавала начальные серии с помощью промежуточного хранилища, как описано в алгоритме 13.2.

13.12. Измените реализацию `SortPlan` так, чтобы в одной итерации она объединяла три серии.

13.13. Измените реализацию `SortPlan` так, чтобы в одной итерации она объединяла `k` серий, где целое число `k` передается в вызов конструктора.

13.14. Измените классы `Plan` в SimpleDB так, чтобы они запоминали факт сортировки их записей, и если записи отсортированы, то по каким полям. Затем измените реализацию `SortPlan` так, чтобы она сортировала записи, только если это необходимо.

13.15. Предложение `order by` в SQL-запросе не является обязательным. Если оно указано, то состоит из двух ключевых слов, «order» и «by», за которыми следует список имен полей, разделенных запятыми.

a) Измените грамматику SQL в листинге 9.5 и добавьте в нее предложение `order by`.
b) Измените лексический и синтаксический анализаторы запросов в SimpleDB, чтобы учесть изменения в синтаксисе.
c) Измените планировщик запросов SimpleDB, чтобы он генерировал соответствующую операцию сортировки для запросов с предложением `order by`. Объект `SortPlan` должен быть самым верхним узлом в дереве запросов.

13.16. В SimpleDB реализованы только две агрегатные функции: `COUNT` и `MAX`. Добавьте классы, реализующие функции `MIN`, `AVG` и `SUM`.

13.17. Ознакомьтесь с синтаксисом SQL-операторов агрегирования.

a) Измените грамматику SQL в листинге 9.5 и добавьте в нее этот синтаксис.
b) Измените лексический и синтаксический анализаторы запросов в SimpleDB и добавьте в них поддержку нового синтаксиса.
c) Измените планировщик запросов SimpleDB, чтобы он генерировал соответствующую операцию группировки для запросов с предложением `group by`. Объект `GroupBy` находится в плане запроса выше узлов селекции (select) и полусоединения (semijoin), но ниже узлов расширения (extend) и прямого произведения (project).

13.18. Определите реляционный оператор *устранения дубликатов (nodups)*, который возвращает таблицу, включающую только уникальные записи из входной таблицы.

a) Определите классы `NoDupsPlan` и `NoDupsScan` по аналогии с классами `GroupByPlan` и `GroupByScan`.
b) Удаление дубликатов также может выполняться оператором группировки в отсутствие агрегатной функции. Определите класс `GBNoDupsPlan`, реализующий оператор nodups путем создания соответствующего объекта `GroupByPlan`.

13.19. В предложении `select` SQL-запроса может присутствовать ключевое слово «distinct». Если оно присутствует, обработчик запросов должен удалить повторяющиеся записи из выходной таблицы.

a) Измените грамматику SQL в листинге 9.5 и добавьте в нее ключевое слово `distinct`.
b) Добавьте в лексический и синтаксический анализаторы запросов SimpleDB поддержку нового синтаксиса.
c) Измените базовый планировщик запросов, чтобы он генерировал подходящую операцию nodups для запросов с ключевым словом `distinct` в предложении `select`.

13.20. Для сортировки таблицы по одному полю также можно использовать индекс в виде B-дерева. Конструктор `SortPlan` сначала должен создать для материализованной таблицы индекс по полю сортировки. Затем

для каждой записи данных добавить в B-дерево соответствующую индексную запись. А потом прочитать записи в порядке сортировки путем обхода листовых узлов B-дерева.

a) Реализуйте эту версию `SortPlan`. (Вам понадобится изменить реализацию B-дерева так, чтобы она связывала все индексные блоки в цепочку.)
b) Сколько обращений к блокам потребуется для такой сортировки? Этот вариант эффективнее сортировки слиянием?

Глава 14

Эффективное использование буферов

Разные реализации операторов имеют разные потребности в буферах. Например, конвейерная реализация оператора селекции очень эффективно использует один буфер и не нуждается в дополнительных буферах. Однако материализующая реализация оператора сортировки объединяет несколько серий в одной итерации, и для хранения каждой серии ей нужен отдельный буфер.

В этой главе рассматриваются различные способы использования дополнительных буферов и приводятся эффективные алгоритмы для операторов сортировки, прямого произведения и соединения, использующие несколько буферов.

14.1. Использование буферов в планах запросов

Реализации операторов реляционной алгебры, обсуждавшиеся выше, имеют очень скромные потребности в буферах. Например, образ сканирования таблицы закрепляет блоки по одному; закончив просматривать записи в блоке, он сначала открепляет его и только потом закрепляет следующий. Образы сканирования для операторов селекции, проекции и прямого произведения вообще не закрепляют дополнительных блоков. Как следствие, обрабатывая запрос к N таблицам, базовый планировщик SimpleDB одновременно закрепляет только N буферов.

Рассмотрим реализацию индексов в главе 12. Статический хеш-индекс реализует каждую ячейку в виде файла и последовательно сканирует ее, закрепляя блоки по одному. Индекс на основе B-дерева тоже закрепляет блоки каталога по одному, начиная с корня. Он сканирует блок, находит в нем дочерний элемент, открепляет текущий блок и закрепляет дочерний; сканирование продолжается до тех пор, пока не будет найден листовой блок[1].

Теперь рассмотрим материализующие реализации, представленные в главе 13. Оператору материализации нужен лишь один дополнительный буфер

[1] Разумеется, это верно только для запросов. При вставке записи в B-дерево может потребоваться закрепить сразу несколько буферов для рекурсивного расщепления блоков и вставки элементов в дерево. В упражнении 12.16 вам предлагалось проанализировать потребность операции вставки в буферах.

для работы с временной таблицей, кроме буферов, необходимых для обработки входного запроса. Этапу расщепления оператора сортировки нужен один или два буфера (в зависимости от того, используется ли промежуточное хранилище), а этапу объединения нужны k+1 буферов: по одному для каждого из k объединяемых серий и один буфер для выходной таблицы. Операторам группировки и соединения слиянием не требуются дополнительные буферы, кроме тех, что используются для сортировки.

Из вышесказанного следует, что, за исключением сортировки, количество буферов, одновременно закрепляемых планировщиком, примерно равно числу таблиц, упомянутых в запросе; обычно это число меньше 10 и почти наверняка меньше 100. Общее количество доступных буферов часто намного больше. Современные серверы обычно имеют не меньше 16 Гбайт физической памяти. Если для буферов выделить всего лишь 400 Мбайт, то в них можно хранить 100 000 четырехкилобайтных буферов. Как видите, даже если система баз данных будет обслуживать сотни (или тысячи) одновременных подключений, в ней все равно найдется достаточное количество буферов для выполнения любых запросов, при условии эффективного их использования планировщиком. В этой главе рассказывается, как операторы сортировки, соединения и прямого произведения могут извлечь выгоду из этого изобилия буферов.

14.2. Многобуферная сортировка

Как рассказывалось выше, алгоритм сортировки слиянием выполняется в два этапа: на первом этапе записи делятся на серии, а на втором серии объединяются до тех пор, пока таблица не будет отсортирована. В главе 13 обсуждались преимущества использования нескольких буферов на этапе слияния. Как оказывается, на этапе расщепления тоже можно использовать дополнительные буферы.

Допустим, что имеется k буферов. На этапе расщепления алгоритм может прочитать сразу k блоков таблицы в k буферов, отсортировать их в одну серию, используя алгоритм внутренней сортировки, и записать результат во временную таблицу. То есть записи распределяются по сериям длиной не в один, а в k блоков. Если число k достаточно велико (например, если $k \geqslant \sqrt{B}$), то этап расщепления произведет не более k начальных серий и на этапе предварительной обработки не придется ничего делать. Порядок выполнения *многобуферной сортировки слиянием* описан в алгоритме 14.1.

Алгоритм 14.1. Многобуферная сортировка слиянием

// Этап расщепления, использующий k буферов
1. Повторять до исчерпания входных записей:
   a) Закрепить k буферов и прочитать в них k блоков с входными записями.
   b) Отсортировать эти записи с использованием алгоритма внутренней сортировки.
   c) Записать содержимое буферов во временную таблицу.
   d) Открепить буферы.
   e) Добавить временную таблицу в список серий.

// Этап слияния, использующий k+1 буферов
2. Повторять, пока в списке серий не останется одна временная таблица:
   // Выполнить итерацию
   a) Повторять до исчерпания списка серий:
      i. Изъять из списка серий k временных таблиц и открыть для них образы сканирования.
      ii. Открыть образ сканирования для новой временной таблицы.
      iii. Объединить k образов в новый образ.
      iv. Добавить новую временную таблицу в список L.
   b) Добавить содержимое L в список серий.

Шаг 1 этого алгоритма произведет B/k начальных серий. Согласно формуле затрат в разделе 13.4.4, многобуферная сортировка слиянием выполнит $\log_k(B/k)$ итераций слияния, то есть на одну меньше, чем простая сортировка слиянием (когда начальные серии имеют длину в 1 блок). Другими словами, многобуферная сортировка слиянием экономит 2B обращений к блокам на этапе предварительной обработки; соответственно, сортировка таблицы, занимающей B блоков, с использованием k буферов имеет следующую стоимость:

- стоимость предварительной обработки = $2B\log_k B - 3B$ + стоимость чтения входных записей;
- стоимость сканирования = B.

Какое значение k можно считать наилучшим? Значение k определяет количество итераций слияния. В частности, количество итераций, выполняемых во время предварительной обработки, равно $(\log_k B)-2$. Эта формула вытекает из следующих соотношений:

- требуется 0 итераций, если $k = \sqrt{B}$;
- требуется 1 итерация, если $k = \sqrt[3]{B}$;
- требуется 2 итерации, если $k = \sqrt[4]{B}$.

И т. д.

Этот расчет должен быть вам понятен. Если $k = \sqrt{B}$, то этап расщепления произведет k серий размером k. Эти серии можно объединить на этапе сканирования, то есть во время предварительной обработки не потребуется выполнять итераций слияния. Если $k = \sqrt[3]{B}$, то этап расщепления произведет k^2 серий размером k. Одна итерация слияния произведет k серий (размером k^2), которые затем можно будет объединить на этапе сканирования.

В качестве конкретного примера допустим, что нам нужно отсортировать таблицу размером 4 Гбайт. Если предположить, что блоки имеют размер 4 Кбайт, то таблица содержит около миллионов блоков. В табл. 14.1 показано, сколько буферов необходимо, чтобы получить определенное количество итераций слияния во время предварительной обработки.

Таблица 14.1. Количество итераций во время предварительной обработки, необходимое для сортировки таблицы размером 4 Гбайт

Количество буферов	1000	100	32	16	10	8	6	5	4	3	2
Количество итераций	0	1	2	3	4	5	6	7	8	11	18

Как можно видеть в нижней строке в табл. 14.1, добавление всего нескольких буферов дает существенное улучшение: при использовании двух буферов требуется 18 итераций, а при использовании 10 буферов требуется всего 4 итерации. Такая огромная разница в стоимости явно показывает, что выделить менее десяти буферов для сортировки этой таблицы – очень плохая идея.

Верхняя строка в табл. 14.1 показывает, насколько эффективной может быть сортировка. В системе баз данных вполне может иметься 1000 свободных буферов или, по крайней мере, 100. При использовании 1000 буферов (что эквивалентно 4 Мбайт памяти) можно отсортировать таблицу размером 4 Гбайт, выполнив 1000 внутренних сортировок на этапе предварительной обработки, с последующим объединением 1000 серий на этапе сканирования. Общая стоимость в этом случае составит три миллиона обращений к блокам: один миллион для чтения несортированных блоков, один миллион для записи во временные таблицы и один миллион для чтения временных таблиц. Неожиданная и впечатляющая эффективность!

Этот пример также показывает, что многобуферная сортировка слиянием таблицы размером `B` эффективно использует только определенное количество буферов, а именно $\sqrt{B}$, $\sqrt[3]{B}$, $\sqrt[4]{B}$ и т. д. В табл. 14.1 перечислены эти значения для `B = 1 000 000`. А как насчет другого количества буферов? Что получится, если движку базы данных доступно, скажем, 500 буферов? Мы знаем, что при использовании 100 буферов необходимо выполнить 1 итерацию слияния на этапе предварительной обработки. Давайте посмотрим, дадут ли какой-то эффект дополнительные 400 буферов. При наличии 500 буферов этап расщепления произведет 2000 серий по 500 блоков в каждой. Первая итерация слияния объединит 500 серий и произведет 4 серии (по 250 000 блоков в каждой), которые затем можно объединить на этапе сканирования. То есть дополнительные 400 буферов не дают никакого эффекта, потому что все равно необходимо выполнить такое же количество итераций, как и при использовании 100 буферов.

Это наблюдение можно выразить следующим правилом: *при использовании k буферов для сортировки таблицы длиной B блоков число k следует выбирать равным корню из B*.

14.3. Многобуферное прямое произведение

Базовая реализация оператора прямого произведения выполняет множество обращений к блокам. Например, посмотрим, как в SimpleDB обрабатывается следующий запрос:

```
product(T1, T2)
```

Текущая реализация прочитает все записи в T2 для каждой записи в T1, используя единственный буфер для хранения записей из T2. То есть как только код проверит последнюю запись в очередном блоке таблицы T2, он открепит этот блок и закрепит следующий. Открепление позволяет диспетчеру буферов вытеснить каждый блок таблицы T2, то есть при проверке следующей записи в T1 может потребоваться прочитать с диска все блоки заново. В худшем слу-

чае каждый блок из T2 будет прочитан столько раз, сколько записей в T1. Если предположить, что T1 и T2 содержат по 1000 блоков, с 20 записями в каждом, то для обработки запроса потребуется 20 001 000 обращений к блокам.

Предположим теперь, что реализация не открепляет блоки таблицы T2. Тогда диспетчер буферов должен будет поместить каждый блок из T2 в отдельный буфер. В результате блоки из T2 будут прочитаны с диска один раз и останутся в памяти на протяжении всего времени обработки запроса. Такое сканирование было бы исключительно эффективным, потому что все блоки из T1 и T2 потребовалось бы прочитать только один раз.

Однако подобная стратегия возможна, только если имеется достаточное количество буферов для хранения всей таблицы T2. Но как быть, если T2 слишком велика? Например, предположим, что T2 занимает 1000 блоков, а доступно только 500 буферов. В таком случае таблицу T2 лучше обработать в два этапа. Сначала прочитать первые 500 блоков в доступные буферы, вычислить прямое произведение T1 с этими блоками; затем прочитать оставшиеся 500 блоков в эти же буферы и вычислить их произведение с T1.

Это очень эффективная стратегия. Первый этап потребует прочитать один раз T1 и первую половину T2, а второй этап – еще раз прочитать T1 и один раз вторую половину T2. В результате таблица T1 будет прочитана дважды, а T2 – только один раз, то есть всего потребуется 3000 обращений к блокам.

Эти идеи обобщены в алгоритме *многобуферного прямого произведения* (см. алгоритм 14.2). В этом алгоритме блоки T1 будут прочитаны один раз для каждого фрагмента T2, помещающегося в `k` буферов. Так как число фрагментов равно `B2/k`, то для получения прямого произведения потребуется `B2 + (B1×B2/k)` обращений к блокам.

Алгоритм 14.2. Многобуферное прямое произведение

Пусть T1 и T2 – две входные таблицы. Предположим, что T2 – хранимая таблица (определена пользователем или является материализованной временной таблицей) и содержит `B2` блоков.

1. Пусть `k = B2/i` для некоторого целого числа `i`.
2. Считать, что T2 состоит из `i` *фрагментов* по `k` блоков в каждом. Для каждого фрагмента C:
    a) прочитать все блоки из C в `k` буферов;
    b) найти прямое произведение T1 и C;
    c) открепить блоки фрагмента C.

Обратите внимание, что реализация многобуферного прямого произведения обрабатывает таблицы T1 и T2 не так, как базовая реализация, представленная в главе 8. Реализация в главе 8 несколько раз сканирует таблицу T2, тогда как эта реализация несколько раз сканирует таблицу T1.

Снова предположим, что имеются таблицы T1 и T2, занимающие по 1000 блоков каждая. В табл. 14.2 показано, сколько обращений к блокам потребуется алгоритму многобуферного прямого произведения при использовании разного количества буферов. Если доступно 1000 буферов, то T2 целиком уместится в один фрагмент и потребуется всего 2000 обращений к блокам. Если доступно 250 буферов, то алгоритм многобуферного произведения разо-

бьет T2 на 4 фрагмента по 250 блоков в каждом. В результате таблица T1 будет просканирована 4 раза, а T2 – один раз, то есть всего потребуется 5000 обращений к блокам. Если доступно только 100 буферов, то алгоритм разобьет T2 на 10 фрагментов и выполнит 11 000 обращений к блокам. Все эти значения намного меньше, чем в базовой реализации прямого произведения.

Таблица 14.2. Количество обращений к блокам при вычислении прямого произведения таблиц, занимающих по 1000 блоков каждая

Количество буферов	1000	500	334	250	200	167	143	125	112	100
Количество фрагментов	1	2	3	4	5	6	7	8	9	10
Количество обращений к блокам	2000	3000	4000	5000	6000	7000	8000	9000	10 000	11 000

Как и в случае с сортировкой, табл. 14.2 также показывает, что не все значения k дают эффект. Если допустить, что в описанном примере доступно 300 буферов, то многобуферный алгоритм прямого произведения сможет использовать только 250 из них.

14.4. Определение необходимого количества буферов

Многобуферные алгоритмы, представленные выше, используют k буферов, но не определяют точного значения k. Правильное значение k определяется количеством доступных буферов, размером входных таблиц и конкретным оператором. Для сортировки k определяется как корень некоторой степени из размера входной таблицы; для прямого произведения k определяется как частное от деления размера таблицы на количество фрагментов.

Цель состоит в том, чтобы выбрать для k наибольший корень (или частное), который меньше количества доступных буферов. В SimpleDB методы для вычисления этих значений помещены в класс `BufferNeeds`, определение которого представлено в листинге 14.1.

Листинг 14.1. Определение класса BufferNeeds в SimpleDB

```
public class BufferNeeds {
    public static int bestRoot(int available, int size) {
        int avail = available - 2; // зарезервировать пару буферов
        if (avail <= 1)
            return 1;
        int k = Integer.MAX_VALUE;
        double i = 1.0;
        while (k > avail) {
            i++;
            k = (int)Math.ceil(Math.pow(size, 1/i));
        }
        return k;
    }
```

```
    public static int bestFactor(int available, int size) {
        int avail = available - 2; // зарезервировать пару буферов
        if (avail <= 1)
            return 1;
        int k = size;
        double i = 1.0;
        while (k > avail) {
            i++;
            k = (int)Math.ceil(size / i);
        }
        return k;
    }
}
```

Класс содержит общедоступные статические методы `bestRoot` и `bestFactor`. Эти два метода практически идентичны. Оба принимают количество доступных буферов и размер таблицы в блоках, и оба вычисляют оптимальное количество буферов как наибольший корень или как наибольшее частное, которое меньше количества доступных буферов. Метод `bestRoot` инициализирует переменную `k` значением `MAX_VALUE`, чтобы заставить цикл выполниться хотя бы один раз (чтобы `k` не могло быть больше $\sqrt{B}$).

Обратите внимание, что методы в классе `BufferNeeds` не резервируют буферы в диспетчере буферов – они просто получают количество доступных буферов как параметр и выбирают значение `k` меньше этого числа. Когда многобуферные алгоритмы будут пытаться закрепить эти `k` блоков, некоторые буферы могут оказаться недоступными. В этом случае алгоритмы будут ждать, пока вновь не станет доступно необходимое количество буферов.

14.5. Реализация многобуферной сортировки

Методы `splitIntoRuns` и `doAMergeIteration` класса `SortPlan` в SimpleDB определяют количество используемых буферов. Текущая реализация `splitIntoRuns` создает серии поочередно, используя один буфер, связанный с временной таблицей, а `doAMergeIteration` использует три буфера (два для входных серий и один – для выходных). В этом разделе рассматривается, как следует изменить эти методы для реализации многобуферной сортировки.

Рассмотрим `splitIntoRuns`. Этот метод не знает, насколько большой получится отсортированная таблица, потому что она еще не создана. Однако он может использовать метод `blocksAccessed`, чтобы получить оценку этого числа. В частности, `splitIntoRuns` может выполнить такой код:

```
int size = blocksAccessed();
int available = tx.availableBuffs();
int numbuffs = BufferNeeds.bestRoot(available, size);
```

Затем закрепить `numbuffs` буферов, заполнить их входными записями, применить к ним алгоритм внутренней сортировки и записать во временную таблицу, согласно алгоритму 14.1.

Теперь рассмотрим метод `doAMergeIteration`. Лучшая стратегия для метода – изымать из списка серий сразу `k` временных таблиц, где `k` – корень из числа начальных серий:

```
int available = tx.availableBuffs();
int numbuffs = BufferNeeds.bestRoot(available, runs.size());
List<TempTable> runsToMerge = new ArrayList<>();
for (int i=0; i<numbuffs; i++)
    runsToMerge.add(runs.remove(0));
```

Затем метод может передать список `runsToMerge` методу `mergeTwoRuns` (который можно переименовать в `mergeSeveralRuns`) для объединения его содержимого в одну серию.

В исходном коде SimpleDB, сопровождающем книгу, нет версии `SortPlan`, реализующей многобуферную сортировку. В упражнениях 14.15–14.17 вам будет предложено решить эту задачу.

Наконец, обратите внимание, что код, использующий класс `SortPlan`, например `GroupByPlan` и `MergeJoinPlan`, не знает, какой алгоритм сортировки используется – простой или многобуферный. Поэтому эти классы не нужно изменять. (Однако есть некоторые незначительные проблемы, связанные с количеством буферов, используемых в `MergeJoinPlan`; см. упражнение 14.5.)

14.6. Реализация многобуферного прямого произведения

Чтобы реализовать алгоритм многобуферного прямого произведения, необходимо реализовать также понятие *фрагмента*. Как рассказывалось выше, фрагмент – это часть материализованной таблицы из k блоков, которая помещается в доступные буферы. В листинге 14.2 представлен класс `ChunkScan`, реализующий фрагмент как образ сканирования соответствующих записей.

Листинг 14.2. Определение класса ChunkScan в SimpleDB

```
public class ChunkScan implements Scan {
    private List<RecordPage> buffs = new ArrayList<>();
    private Transaction tx;
    private String filename;
    private Layout layout;
    private int startbnum, endbnum, currentbnum;
    private RecordPage rp;
    private int currentslot;

    public ChunkScan(Transaction tx, String filename,
                     Layout layout, int startbnum, int endbnum) {

        this.tx = tx;
        this.filename = filename;
        this.layout = layout;
        this.startbnum = startbnum;
        this.endbnum   = endbnum;
        for (int i=startbnum; i<=endbnum; i++) {
            BlockId blk = new BlockId(filename, i);
            buffs.add(new RecordPage(tx, blk, layout));
        }
        moveToBlock(startbnum);
    }
```

```
    public void close() {
        for (int i=0; i<buffs.size(); i++) {
            BlockId blk = new BlockId(filename, startbnum+i);
            tx.unpin(blk);
        }
    }

    public void beforeFirst() {
        moveToBlock(startbnum);
    }

    public boolean next() {
        currentslot = rp.nextAfter(currentslot);
        while (currentslot < 0) {
            if (currentbnum == endbnum)
                return false;
            moveToBlock(rp.block().number()+1);
            currentslot = rp.nextAfter(currentslot);
        }
        return true;
    }

    public int getInt(String fldname) {
        return rp.getInt(currentslot, fldname);
    }

    public String getString(String fldname) {
        return rp.getString(currentslot, fldname);
    }

    public Constant getVal(String fldname) {
        if (layout.schema().type(fldname) == INTEGER)
            return new Constant(getInt(fldname));
        else
            return new Constant(getString(fldname));
    }

    public boolean hasField(String fldname) {
        return layout.schema().hasField(fldname);
    }

    private void moveToBlock(int blknum) {
        currentbnum = blknum;
        rp = buffs.get(currentbnum - startbnum);
        currentslot = -1;
    }
}
```

Конструктор `ChunkScan` получает метаданные хранимой таблицы вместе с номерами первого и последнего блоков в фрагменте. Он открывает страницы записей для каждого блока в фрагменте и сохраняет их в списке. Образ сканирования также запоминает текущую страницу; в первый момент текущей становится первая страница в списке. Метод `next` переходит к следующей записи в текущей странице. Если записи в текущей странице исчерпаны, то текущей становится следующая страница в списке. В отличие от сканирова-

ния таблиц, при перемещении между блоками при сканировании фрагментов предыдущая страница не закрывается (иначе это привело бы к откреплению ее буфера) – страницы в фрагменте открепляются, только когда закрывается сам фрагмент.

Алгоритм многобуферного прямого произведения реализуется классом `MultibufferProductPlan`; его определение показано в листинге 14.3. Метод `open` материализует записи слева и справа: левая сторона материализуется как `MaterializeScan`, а правая – как временная таблица. Метод `blockAccessed` должен знать размер материализованной таблицы справа, чтобы вычислить количество фрагментов. Поскольку эта таблица не существует, пока план не открыт, в качестве размера метод использует оценку из `MaterializePlan`. Методы `recordsOutput` и `distinctValues` идентичны одноименным методам в `ProductPlan` и имеют простую реализацию.

Листинг 14.3. Определение класса MultibufferProductPlan в SimpleDB

```
public class MultibufferProductPlan implements Plan {
    private Transaction tx;
    private Plan lhs, rhs;
    private Schema schema = new Schema();

    public MultibufferProductPlan(Transaction tx, Plan lhs, Plan rhs) {
        this.tx = tx;
        this.lhs = new MaterializePlan(tx, lhs);
        this.rhs = rhs;
        schema.addAll(lhs.schema());
        schema.addAll(rhs.schema());
    }

    public Scan open() {
        Scan leftscan = lhs.open();
        TempTable t = copyRecordsFrom(rhs);
        return new MultibufferProductScan(tx, leftscan, t.tableName(), t.getLayout());
    }

    public int blocksAccessed() {
        // получить оценочное количество фрагментов
        int avail = tx.availableBuffs();
        int size = new MaterializePlan(tx, rhs).blocksAccessed();
        int numchunks = size / avail;
        return rhs.blocksAccessed() +
            (lhs.blocksAccessed() * numchunks);
    }

    public int recordsOutput() {
        return lhs.recordsOutput() * rhs.recordsOutput();
    }

    public int distinctValues(String fldname) {
        if (lhs.schema().hasField(fldname))
            return lhs.distinctValues(fldname);
        else
            return rhs.distinctValues(fldname);
    }
```

```
    public Schema schema() {
        return schema;
    }

    private TempTable copyRecordsFrom(Plan p) {
        Scan src = p.open();
        Schema sch = p.schema();
        TempTable tt = new TempTable(tx, sch);
        UpdateScan dest = (UpdateScan) tt.open();
        while (src.next()) {
            dest.insert();
            for (String fldname : sch.fields())
                dest.setVal(fldname, src.getVal(fldname));
        }
        src.close();
        dest.close();
        return tt;
    }
}
```

Определение класса `MultibufferProductScan` показано в листинге 14.4. Его конструктор получает размер фрагмента, вызывая `BufferNeeds.bestFactor` с размером правой таблицы. Затем в образе сканирования слева устанавливает позицию текущей записи перед первой записью, открывает `ChunkScan` для первого фрагмента в таблице справа и создает `ProductScan` из этих двух образов. То есть переменная `prodscan` содержит обычный образ сканирования для прямого произведения образа слева и текущего фрагмента. Этот образ сканирования прямого произведения используется в большинстве методов класса. Исключение составляет метод `next`.

Листинг 14.4. Определение MultibufferProductScan класса в SimpleDB

```
public class MultibufferProductScan implements Scan {
    private Transaction tx;
    private Scan lhsscan, rhsscan=null, prodscan;
    private String filename;
    private Layout layout;
    private int chunksize, nextblknum, filesize;

    public MultibufferProductScan(Transaction tx, Scan lhsscan,
                                  String filename, Layout layout) {

        this.tx = tx;
        this.lhsscan = lhsscan;
        this.filename = filename;
        this.layout = layout;
        filesize = tx.size(filename);
        int available = tx.availableBuffs();
        chunksize = BufferNeeds.bestFactor(available, filesize);
        beforeFirst();
    }

    public void beforeFirst() {
        nextblknum = 0;
        useNextChunk();
    }
```

```
    public boolean next() {
        while (!prodscan.next())
            if (!useNextChunk())
                return false;
        return true;
    }

    public void close() {
        prodscan.close();
    }

    public Constant getVal(String fldname) {
        return prodscan.getVal(fldname);
    }

    public int getInt(String fldname) {
        return prodscan.getInt(fldname);
    }

    public String getString(String fldname) {
        return prodscan.getString(fldname);
    }

    public boolean hasField(String fldname) {
        return prodscan.hasField(fldname);
    }

    private boolean useNextChunk() {
        if (rhsscan != null)
            rhsscan.close();
        if (nextblknum >= filesize)
            return false;
        int end = nextblknum + chunksize - 1;
        if (end >= filesize)
            end = filesize - 1;
        rhsscan = new ChunkScan(tx, filename, layout, nextblknum, end);
        lhsscan.beforeFirst();
        prodscan = new ProductScan(lhsscan, rhsscan);
        nextblknum = end + 1;
        return true;
    }
}
```

Метод `next` выполняет переход к следующей записи в текущем образе прямого произведения. Достигнув конца образа, он закрывает его, создает новый образ сканирования прямого произведения для следующего фрагмента и переходит к первой записи в нем. Если был обработан последний фрагмент, то метод возвращает `false`.

14.7. Соединение хешированием

В разделе 13.6 был рассмотрен алгоритм соединения слиянием. Поскольку этот алгоритм сортирует обе входные таблицы, его стоимость определяется размером большей из них. В этом разделе рассматривается другой алгоритм соединения, который называется *соединение хешированием (hashjoin)*. Он обладает

ценным свойством – его стоимость определяется размером меньшей входной таблицы. То есть этот алгоритм предпочтительнее, чем соединение слиянием, когда входные таблицы имеют очень разные размеры.

14.7.1. Алгоритм соединения хешированием

Идею, лежащую в основе алгоритма многобуферного прямого произведения, можно распространить на вычисление соединения двух таблиц. *Соединение хешированием* представлено в алгоритме 14.3.

Алгоритм 14.3. Соединение хешированием

Пусть T1 и T2 – таблицы, соединение которых требуется вычислить.
1. Выбрать значение k меньше числа доступных буферов.
2. Если размер T2 в блоках не превышает k, то:
    a) вычислить соединение T1 и T2, используя многобуферное прямое соединение и последующую селекцию по предикату соединения;
    b) вернуть результат.

// Иначе:

3. Выбрать хеш-функцию, возвращающую значение между 0 и k-1.
4. Для таблицы T1:
    a) открыть образы сканирования для k временных таблиц;
    b) для каждой записи в T1:
        i. получить хеш-значение h для поля соединения в текущей записи;
        ii. скопировать запись в h-ю временную таблицу;
    c) закрыть образы сканирования временных таблиц.
5. Повторить шаг 4 для таблицы T2.
6. Для каждого i от 0 до k-1:
    a) пусть V_i – это i-я временная таблица, полученная на шаге 4 для T1;
    b) пусть W_i – это i-я временная таблица, полученная на шаге 5 для T2;
    c) рекурсивно вычислить соединение хешированием для V_i и W_i.

Алгоритм соединения хешированием может действовать рекурсивно, в зависимости от размера таблицы T2. Если T2 достаточно мала и умещается в доступные буферы, то алгоритм вычисляет соединение T1 и T2, используя алгоритм многобуферного прямого произведения. Если T2 слишком велика, чтобы уместиться в памяти, то алгоритм использует хеширование, чтобы уменьшить размер T2. В этом случае он создает два набора временных таблиц: набор $\{V_0, \ldots, V_{k-1}\}$ для T1 и набор $\{W_0, \ldots, W_{k-1}\}$ для T2. Эти временные таблицы играют роль ячеек хеш-таблицы. Каждая запись из T1 хешируется по своему полю соединения и распределяется в соответствующую ячейку. Аналогично распределяются записи из T2. Затем рекурсивно производится соединение соответствующих таблиц (V_i, W_i).

Очевидно, что все записи, имеющие одинаковое значение в поле соединения, будут распределяться в одну и ту же ячейку. То есть общее соединение T1 и T2 можно получить, независимо вычисляя соединения V_i и W_i для каждого i. Поскольку каждая временная таблица W_i будет меньше T2, рекурсия гарантированно прекратится.

Обратите внимание, что каждый рекурсивный вызов алгоритма соединения хешированием должен использовать другую хеш-функцию. Причина в том, что все записи в текущей временной таблице оказались в ней потому, что для них хеш-функция вернула одно и то же значение. Использование другой хеш-функции гарантирует равномерное распределение записей между новыми временными таблицами.

Алгоритм 14.3 также требует повторного выбора значения k в каждом рекурсивном вызове. Однако вы можете выбрать k один раз и использовать его во всех вызовах. В упражнении 14.11 вам будет предложено рассмотреть достоинства и недостатки этих двух вариантов.

Есть возможность немного увеличить эффективность многобуферного прямого произведения, проявив смекалку при реализации поиска соответствующих записей в блоках. Для данной записи в T1 алгоритм должен найти соответствующие записи в T2. Стратегия многобуферного прямого произведения заключается в простом поиске по всей таблице T2. Даже притом что этот поиск не требует дополнительных обращений к диску, его эффективность можно повысить, если использовать подходящие внутренние структуры данных. Например, ссылки на записи в T2 можно хранить в хеш-таблице или в двоичном дереве поиска. (На самом деле подойдет любая реализация интерфейса `Map` в Java.) По значению поля соединения данной записи в T1 алгоритм будет получать в структуре данных ссылки на записи в T2, имеющие то же значение поля соединения, что позволит ему избежать необходимости поиска в T2.

14.7.2. Пример соединения хешированием

В качестве конкретного примера применим алгоритм соединения хешированием для вычисления соединения таблиц ENROLL и STUDENT, используя записи из табл. 1.1. Будем исходить из следующих предположений:

- таблица STUDENT находится справа от оператора соединения;
- в один блок умещается две записи STUDENT, и точно так же в один блок умещается две записи ENROLL;
- хеш-таблица состоит из трех ячеек, то есть `k = 3`;
- хеш-функция имеет вид `h(n) = n%3`.

Девять записей STUDENT занимают пять блоков. Поскольку `k = 3`, все записи STUDENT не поместятся в памяти, поэтому будем использовать хеширование. Полученные ячейки показаны на рис. 14.1.

Значения 3, 6 и 9 идентификаторов студентов хешируются в значение 0. Поэтому записи ENROLL, соответствующие этим студентам, помещаются в V_0, а соответствующие записи STUDENT – в W_0. Аналогично записи для студентов 1, 4 и 7 помещаются в V_1 и W_1, а записи для студентов 2, 5 и 8 – в V_2 и W_2. Теперь можно рекурсивно вычислить соединение для каждой пары таблиц V_i и W_i.

Поскольку каждая таблица W_i занимает два блока, любая из них уместится в памяти; поэтому каждое из трех рекурсивных соединений можно вычислить как многобуферное прямое произведение. В этом случае, чтобы вычислить соединение V_i и W_i, нужно прочитать все содержимое W_i в память. Затем просканировать V_i и для каждой записи в V_i найти все соответствующие записи в W_i.

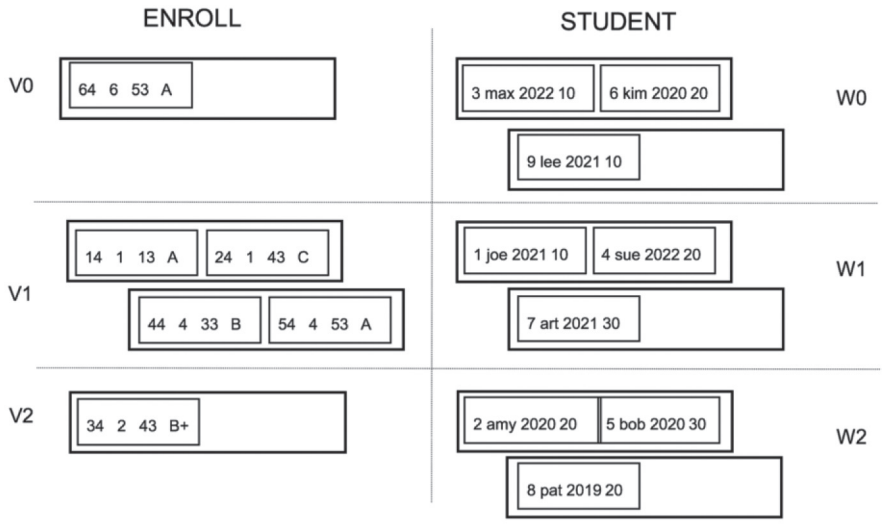


Рис. 14.1. Использование алгоритма соединения хешированием для вычисления соединения таблиц ENROLL и STUDENT

14.7.3. Анализ стоимости

Чтобы оценить стоимость применения алгоритма соединения хешированием для соединения T1 и T2, предположим, что материализованные записи в T1 занимают B1 блоков, а записи в T2 занимают B2 блоков. Выберем k как корень n-й степени из B2; то есть $B2 = k^n$. Затем, исходя из предположения о равномерном хешировании записей, рассчитаем затраты, как описывается далее.

Первый раунд распределения записей создаст k временных таблиц; каждая временная таблица, полученная из T2, будет занимать k^{n-1} блоков. После рекурсивного распределения записей из этих временных таблиц всего будет создано k^2 временных таблиц, каждая из которых будет занимать k^{n-2} блоков. В конечном итоге для T2 будет создано k^{n-1} временных таблиц, каждая из которых занимает k блоков. Для этих таблиц затем можно вычислить соединение (с соответствующими временными таблицами, полученными из T1) с использованием многобуферного прямого произведения.

Из вышесказанного следует, что будет выполнен n-1 раунд распределения записей. Стоимость первого раунда равна B1 + B2, плюс стоимость чтения входных таблиц. В последующих раундах каждый блок каждой временной таблицы будет прочитан один раз и записан один раз, поэтому стоимость этих раундов составит 2(B1 + B2). Многобуферное прямое произведение выполняется на этапе сканирования. Каждый блок каждой временной таблицы будет прочитан один раз, поэтому стоимость этого этапа составит B1 + B2.

Объединив эти стоимости, получаем стоимость соединения хешированием таблиц с размерами B1 и B2 и при наличии k буферов:

- стоимость предварительной обработки = $(2B_1 \log_k B_2 - 3B_1) + (2B_2 \log_k B_2 - 3B_2)$ + стоимость чтения входных записей;
- стоимость сканирования = $B_1 + B_2$.

Как ни удивительно, эта стоимость почти равна стоимости многобуферного соединения слиянием! Но есть одно отличие: в этой формуле аргументами обоих логарифмов является значение B2, тогда как в формуле стоимости соединения слиянием аргументом первого логарифма является значение B1. Причина этого отличия в том, что количество раундов распределения в соединении хешированием определяется только размером таблицы T2, тогда как в соединении слиянием количество итераций слияния на этапе сортировки определяется размерами обеих таблиц, T1 и T2.

Это отличие объясняет разную производительность двух алгоритмов соединения. Алгоритм соединения слиянием должен отсортировать обе входные таблицы, прежде чем вычислить их соединение. Алгоритм соединения хешированием, напротив, не зависит от размера T1; он прекращает раунды распределения, когда фрагменты T2 станут достаточно маленькими. Стоимость соединения слиянием не зависит от того, какая таблица находится слева или справа. Но алгоритм соединения хешированием более эффективен, когда меньшая таблица находится справа.

Если T1 и T2 имеют близкие размеры, то, вероятно, лучше использовать соединение слиянием, даже притом что алгоритм соединения хешированием имеет ту же формулу стоимости. Причина в том, что формула стоимости соединения хешированием зависит от предположения о равномерном хешировании записей. Но если хеширование происходит неравномерно, алгоритму потребуется больше буферов и больше итераций, чем указано в формуле. Алгоритм соединения слиянием, напротив, ведет себя гораздо более предсказуемо.

14.8. Сравнение алгоритмов соединения

В этой главе были рассмотрены два способа реализации соединения двух таблиц, соединение слиянием и соединение хешированием, а также в главе 12 была исследована реализация соединения с использованием индекса. Для оценки относительных преимуществ этих трех реализаций в этом разделе используется следующий запрос:

```
select SName, Grade from STUDENT, ENROLL where SId=StudentId
```

Допустим, что таблицы имеют размеры, как указано в табл. 7.2, доступно 200 буферов и таблица ENROLL имеет индекс для поля `StudentId`.

Рассмотрим алгоритм соединения слиянием. Он должен отсортировать обе таблицы, ENROLL и STUDENT, перед вычислением соединения. Таблица ENROLL занимает 50 000 блоков. Квадратный корень из 50 000 равен 244, что превышает количество доступных буферов. Поэтому используем корень кубический и выделим для алгоритма 37 блоков. Этап расщепления создаст 1352 серии, каждая из которых состоит из 37 блоков. Одна итерация слияния создаст 37 серий, по 1352 блока в каждой. То есть для предварительной обработки таблицы ENROLL потребуется дважды прочитать и записать каждую запись, что составит 200 000 обращений к блокам. Таблица STUDENT занимает 4500 блоков. Квадратный корень из 4500 равен 68, и это количество буферов нам доступно. Поэтому используем 68 буферов, чтобы расщепить 4500 блоков

таблицы STUDENT на 68 серий по 68 блоков в каждой. Чтобы выполнить это расщепление, потребуется 9000 обращений к блокам, и это все, что необходимо для предварительной обработки. Для вычисления соединения двух отсортированных таблиц потребуется еще 54 500 обращений к блокам, что составит 263 500 обращений к блокам.

Рассмотрим теперь алгоритм соединения хешированием. Этот алгоритм наиболее эффективен, когда наименьшая таблица находится справа; поэтому поставим таблицу ENROLL слева, а STUDENT – справа. Для распределения записей STUDENT можно использовать 68 буферов и получить 68 ячеек, каждая из которых будет содержать примерно 68 блоков. Те же 68 буферов можно использовать для распределения ENROLL и получить те же 68 ячеек примерно по 736 блоков в каждой. Затем рекурсивно соединить соответствующие ячейки. Соединение каждой пары ячеек можно выполнить с использованием многобуферного прямого произведения. То есть выделить 68 буферов для хранения всей ячейки STUDENT и один буфер для последовательного сканирования ячейки ENROLL. Все ячейки сканируются один раз. Суммируем затраты и получаем следующее: записи из таблиц ENROLL и STUDENT будут прочитаны один раз, и каждая ячейка будет прочитана и записана один раз, в общей сложности получаем 163 500 обращений к блокам.

Реализация соединения с использованием индекса сканирует таблицу STUDENT. В процессе сканирования она извлекает из каждой записи значение `SId` для поиска по индексу и отыскивает соответствующие записи в ENROLL. То есть таблица STUDENT будет прочитана один раз (всего 4500 обращений к блокам), а таблица ENROLL будет читаться один раз для каждого найденного совпадения. Однако, поскольку любая запись ENROLL соответствует некоторой записи STUDENT, для чтения таблицы ENROLL может потребоваться до 1 500 000 обращений к блокам. Соответственно, стоимость обработки запроса составит 1 504 500 обращений к блокам.

Этот пример показывает, что при исходных допущениях самым быстрым является алгоритм соединения хешированием, за ним следует алгоритм соединения слиянием и, наконец, алгоритм соединения с использованием индекса. Такая высокая эффективность алгоритма соединения хешированием объясняется тем, что одна из таблиц (STUDENT) достаточно мала, чтобы уместиться в доступные буферы, а другая (ENROLL) намного больше. Предположим теперь, что доступно 1000 буферов. Тогда алгоритм соединения слиянием сможет отсортировать ENROLL без выполнения итераций слияния, и общая стоимость составит 163 500 обращений к блокам, как и в алгоритме соединения хешированием. Алгоритм соединения с использованием индекса является наименее эффективным для этого запроса. Причина в том, что индексы практически бесполезны, когда имеется много совпадающих записей данных, а в этом запросе каждая запись в ENROLL имеет соответствующую запись в STUDENT.

Теперь рассмотрим вариант этого запроса с дополнительным предикатом, сравнивающим поле `GradYear` с константой:

```
select SName, Grade from STUDENT, ENROLL
where SId=StudentId and GradYear=2020
```

Рассмотрим сначала реализацию соединения слиянием. В таблице STUDENT имеется всего 900 записей, соответствующих этому предикату, которые уместятся в 90 блоков. Таким образом, для сортировки записей STUDENT достаточно прочитать их в 90 буферов и использовать алгоритм внутренней сортировки. В этом случае потребуется всего 4500 обращений к блокам. Но стоимость обработки ENROLL останется прежней, поэтому в общей сложности для обработки запроса потребуется 204 500 обращений к блокам, что лишь немного лучше, по сравнению с соединением слиянием в исходном запросе.

Реализация соединения хешированием определит, что 90 блоков с записями STUDENT поместятся в 90 буферов без всякого распределения, и соединение можно выполнить однократным сканированием обеих таблиц, что составит 54 500 обращений к блокам.

Реализация соединения с использованием индекса прочитает все 4500 записей STUDENT, чтобы найти 900 студентов, выпущенных в 2020 году. Этим записям будет соответствовать 1/50 часть (или 50 000) записей ENROLL, для чтения которых потребуется примерно 50 000 обращений к блокам ENROLL, или всего 54 500 обращений.

В этом случае алгоритмы соединения хешированием и с использованием индекса имеют одинаковую стоимость, но стоимость соединения слиянием значительно выше. Причина в том, что алгоритм соединения слиянием вынужден предварительно обработать обе таблицы, даже притом что одна из них значительно меньше.

И для заключительного примера изменим запрос выше, еще больше ограничив размер выборки из таблицы STUDENT:

```
select SName, Grade from STUDENT, ENROLL
where SId=StudentId and SId=3
```

Теперь выходная таблица будет содержать 34 записи, соответствующие единственному студенту. В этом случае наиболее эффективным будет алгоритм соединения с использованием индекса. Он просканирует все 4500 блоков STUDENT, просмотрит индекс и отыщет 34 записи ENROLL, для чего в сумме потребуется около 4534 обращений к блокам (не считая затрат на поиск в индексе). Реализация соединения хешированием имеет ту же стоимость, что и раньше. Ей потребуется один раз просканировать таблицу STUDENT (чтобы материализовать единственную запись) и один раз – таблицу ENROLL (чтобы найти все совпадающие записи), что в сумме составит 54 500 обращений к блокам. А алгоритму соединения слиянием придется предварительно обработать ENROLL и STUDENT так же, как раньше, что в общей сложности составит 204 500 обращений к блокам.

Этот пример показывает, что соединение слиянием наиболее эффективно, когда входные таблицы имеют примерно одинаковые размеры. Соединение хешированием эффективнее, когда входные таблицы имеют сильно отличающиеся размеры. И соединение с использованием индекса эффективнее, когда количество выходных записей невелико.

14.9. Итоги

- Нематериализующие образы сканирования очень экономны с точки зрения использования буферов. В частности:
  - табличный образ использует только один буфер;
  - образы для операторов селекции, проекции и прямого произведения не используют дополнительных буферов;
  - статическое хеширование и B-деревья используют один дополнительный буфер (для запросов).
- Алгоритм *сортировки слиянием* может с выгодой использовать дополнительные буферы при создании начальных серий и при их слиянии. Он выбирает `k = ⁿ√B`, где `B` – размер входной таблицы, а `n` – наименьшее целое число, при котором значение `k` получается меньше количества доступных буферов. Такой алгоритм называется *многобуферной сортировкой слиянием* и определяется следующим образом:
  - получить `k` буферов у диспетчера буферов;
  - прочитать `k` блоков из таблицы в `k` буферов и использовать алгоритм внутренней сортировки для получения серии из `k` блоков;
  - выполнить итерации слияния исходных серий, используя `k` временных таблиц, пока не останется `k` или меньше серий. Поскольку этап расщепления создает `B/k` серий, он выполнит `n-2` итераций слияния;
  - объединить полученные `k` серий на этапе сканирования.
- Алгоритм *многобуферного прямого произведения* является более эффективной реализацией оператора прямого произведения и работает следующим образом:
  1. Материализовать таблицу справа во временную таблицу T2. Пусть B2 – это количество блоков в T2.
  2. Пусть `i` – наименьшее число, такое, что `B2/i` меньше количества доступных буферов.
  3. Считать, что T2 состоит из `i` *фрагментов* по `k` блоков в каждом. Для каждого фрагмента *C*:
     a) прочитать все блоки из *C* в `k` буферов;
     b) найти прямое произведение T1 и *C*;
     c) открепить блоки фрагмента *C*.

То есть блоки T1 будут прочитаны один раз для каждого фрагмента, и общее количество обращений к блокам составит:

```
B2 + B1×B2/k
```

- Но не всякое количество используемых буферов дает выгоды. Количество эффективно используемых буферов для многобуферной сортировки равно корню некоторой степени из размера таблицы, а для многобуферного прямого произведения – частному от деления размера таблицы справа на целое число.
- Алгоритм *соединения хешированием* является расширением алгоритма многобуферного прямого произведения и работает следующим образом:

1. Выбрать значение k, меньшее количества доступных буферов.
2. Если T2 умещается в k буферов, использовать многобуферное прямое произведение для соединения T1 и T2.
3. Иначе распределить записи каждой из T1 и T2 в k временных таблиц.
4. Рекурсивно выполнить соединение хешированием соответствующих хеш-ячеек.

14.10. Для дополнительного чтения

В статье Shapiro (1986) описывается и анализируется несколько алгоритмов соединения и их требования к буферам. В статье Yu & Cornell (1993) рассматривается эффективность использования буферов с точки зрения стоимости. В ней утверждается, что буферы являются ценным глобальным ресурсом и что вместо выделения как можно большего числа буферов (что и делает SimpleDB), для обработки запроса следует использовать количество, наиболее выгодное для системы в целом. В статье приводится алгоритм, который можно использовать для определения оптимального количества буферов.

Shapiro, L. (1986) «Join processing in database systems with large main memories». ACM Transactions on Database Systems, 11 (3), 239–264.

Yu, P., & Cornell, D. (1993) «Buffer management based on return on consumption in a multi-query environment». VLDB Journal, 2 (1), 1–37.

14.11. Упражнения

Теория

14.1. Предположим, что система баз данных имеет так много буферов, что все они никогда не закрепляются одновременно. Можно ли считать это бесполезной тратой физической памяти, или наличие избыточного количества буферов имеет свои преимущества?

14.2. ОЗУ постоянно дешевеет. Предположим, что в системе баз данных больше буферов, чем блоков в базе данных. Можно ли эффективно использовать все буферы?

14.3. Предположим, что система баз данных имеет достаточно буферов для хранения всех блоков базы данных. Такая система называется системой баз данных в *основной памяти*, потому что она может прочитать всю базу данных в буферы и затем обрабатывать запросы без дополнительных обращений к блокам.

a) Становится ли ненужным какой-либо компонент системы баз данных в этом случае?

b) Должен ли какой-либо компонент функционировать совершенно иначе?

c) В такой системе баз данных функции оценки плана запроса, безусловно, потребуется изменить, потому что не имеет смысла оценивать запросы по количеству обращений к блокам. Предложите лучшую функцию, которая более точно моделирует затраты на обработку запроса.

14.4. Изучите реализацию многобуферной сортировки в разделе 14.5, которая предполагает, что методы `splitIntoRuns` и `doAMergeIteration` должны определять, сколько буферов выделить.

a) Как вариант, метод `open` может определить значение `numbuffs` и передать его в оба метода. Объясните, почему это нежелательный вариант.
b) Еще один вариант – выделить буферы в конструкторе `SortPlan`. Объясните, почему этот вариант еще хуже.

14.5. Предположим, что класс `SortPlan` был изменен и теперь реализует алгоритм 14.1 многобуферной сортировки, и рассмотрим первый пример соединения слиянием в разделе 14.8.

a) Сколько буферов будет использовано на этапе сканирования в этом примере?
b) Предположим, что вместо 200 доступно только 100 буферов, и буферы выделяются сначала для таблицы ENROLL, а потом для STUDENT. Как они будут распределены?
c) Предположим, что вместо 200 доступно только 100 буферов, и буферы выделяются сначала для таблицы STUDENT, а потом для ENROLL. Как они будут распределены?
d) Другой вариант – полностью материализовать любую из отсортированных таблиц перед их соединением. Определите стоимость этого варианта.

14.6. Исследуйте следующий алгоритм реализации оператора группировки.
1. Создать и открыть `k` временных таблиц.
2. Каждую входную запись:
   a) хешировать по ее полю группировки;
   b) скопировать в соответствующую временную таблицу.
3. Закрыть временные таблицы.
4. Для каждой временной таблицы выполнить алгоритм группировки сортировкой.
   a) Объясните, почему этот алгоритм работает.
   b) Вычислите стоимости этапов предварительной обработки и сканирования в этом алгоритме.
   c) Объясните, почему в общем случае этот алгоритм менее эффективен, чем реализация алгоритма группировки сортировкой в листинге 13.6.
   d) Объясните, почему этот алгоритм может пригодиться в окружениях с поддержкой параллельной обработки.

14.7. Исследуйте пример многобуферного прямого произведения в разделе 14.3, который находил произведение двух таблиц, занимающих 1000 блоков каждая. Предположим, что для таблицы T2 доступен только один буфер, то есть `k = 1`.

a) Вычислите, сколько обращений к блокам потребуется для получения прямого произведения.
b) Это число обращений к блокам значительно меньше, чем при применении базового алгоритма прямого произведения, описанного

в главе 8, хотя используется такое же количество буферов. Объясните, почему?

14.8. Алгоритм многобуферного прямого произведения требует материализации таблицы справа (чтобы ее можно было разбить на фрагменты). Однако класс `MultibufferProductPlan` также материализует образ сканирования слева. Отказ от материализации левого операнда может вызвать проблемы с использованием буфера и эффективностью. Объясните причины и приведите примеры обеих проблем.

14.9. Перепишите алгоритм 14.3 соединения хешированием так, чтобы он стал нерекурсивным. Хеширование в этом алгоритме должно производиться только на этапе предварительной обработки, а слияние – лишь на этапе сканирования.

14.10. Алгоритм 14.3 соединения хешированием использует одно и то же значение `k` для хеширования записей из T1 и T2. Объясните, почему использование разных значений `k` невозможно.

14.11. Алгоритм 14.3 соединения хешированием повторно выбирает значение `k` при каждом вызове.

a) Объясните, почему можно выбрать значение `k` один раз и передавать его в каждый рекурсивный вызов.

b) Проанализируйте достоинства и недостатки этих двух вариантов. Какой бы вы предпочли?

14.12. Предположим, что вы изменили алгоритм 14.3 соединения хешированием так, что на шаге 6 для объединения отдельных фрагментов вместо рекурсивного вызова соединения хешированием выполняется соединение слиянием. Вычислите стоимость этого алгоритма и сравните с исходным алгоритмом соединения хешированием.

14.13. Предположим, что таблица STUDENT имеет индексы для полей `SId` и `MajorId`. Для каждого из следующих SQL-запросов, используя статистику в табл. 7.2, вычислите стоимость обработки с использованием соединения слиянием, хешированием или с применением индекса.

```
(a) select SName, DName from STUDENT, DEPT
     where MajorId=DId
(b) select SName, DName from STUDENT, DEPT
     where MajorId=DId and GradYear=2020
(c) select DName from STUDENT, DEPT
     where MajorId=DId and SId=1
(d) select SName from STUDENT, ENROLL
     where SId=StudentId and Grade='F'
```

Практика

14.14. Класс `BufferNeeds` в SimpleDB не обращается к диспетчеру буферов для резервирования буферов.

a) Перечислите некоторые возможные проблемы в существующей реализации SimpleDB, которые можно решить за счет такого резервирования буферов. Есть ли какие-то преимущества в отказе от резервирования буферов?

b) Подумайте, как можно изменить архитектуру диспетчера буферов в SimpleDB, чтобы он позволял транзакциям резервировать буферы. (Обязательно рассмотрите ситуацию, когда транзакция T1 закрепляет блок b в зарезервированный буфер, а затем транзакция T2 пытается закрепить b. Что делать в этом случае?)
c) Реализуйте свою архитектуру и внесите соответствующие изменения в `BufferNeeds`.

14.15. В упражнении 13.10 предлагалось изменить класс `SortPlan` так, чтобы он создавал одноблочные начальные серии. Теперь измените его так, чтобы он создавал начальные серии длиной `k` блоков, как описано в разделе 14.5.
14.16. В упражнении 13.11 предлагалось изменить класс `SortPlan` так, чтобы для вычисления начальных серий он использовал одноблочное промежуточное хранилище. Теперь измените его так, чтобы он использовал промежуточное хранилище объемом в `k` блоков.
14.17. В упражнении 13.13 предлагалось изменить класс `SortPlan` так, чтобы в каждой итерации он объединял сразу `k` серий, при этом значение `k` передавалось в конструктор. Теперь измените его так, чтобы значение `k` определялось количеством начальных серий, как описано в разделе 14.5.
14.18. Алгоритм многобуферного прямого произведения обычно действует эффективнее, когда наименьшая входная таблица находится справа.
a) Объясните, почему?
b) Измените класс `MultiBufferProductPlan` так, чтобы он всегда помещал наименьшую входную таблицу справа.
14.19. Измените класс `MultiBufferProductPlan` так, чтобы он материализовал таблицы слева и справа, только если это необходимо.
14.20. Реализуйте в SimpleDB алгоритм соединения хешированием.

Глава 15

Оптимизация запросов

Базовый планировщик, представленный в главе 10, создает планы запросов, используя простой алгоритм. К сожалению, часто эти планы требуют намного большего количества обращений к блокам, чем необходимо, по двум основным причинам: операции выполняются в неоптимальном порядке и не используют преимуществ индексированных, материализующих или многобуферных реализаций, описанных в главах 12–14.

В этой главе рассказывается, как планировщик может решать такие проблемы и создавать эффективные планы. Эта задача называется *оптимизацией запроса*. Самый эффективный план выполнения запроса может оказаться на несколько порядков быстрее плана, построенного простым алгоритмом. Это отличает движки баз данных, способные отвечать на запросы за разумное время, от движков, полностью непригодных для практического использования. Поэтому хорошая стратегия оптимизации запросов является жизненно важной частью любой коммерческой системы баз данных.

15.1. Эквивалентные деревья запросов

Две таблицы *эквивалентны*, если с точки зрения SQL-запроса они неотличимы. То есть две эквивалентные таблицы содержат одни и те же записи, хотя и не обязательно в том же порядке. Два запроса *эквивалентны*, если их выходные таблицы всегда эквивалентны, независимо от содержимого базы данных. В этом разделе рассматривается эквивалентность запросов реляционной алгебры. Эти запросы можно представить в виде деревьев, поэтому эквивалентность двух запросов часто можно рассматривать как преобразование между их деревьями. Эти преобразования рассматриваются в следующих подразделах.

15.1.1. Перестановка операндов в прямом произведении

Пусть T1 и T2 – две таблицы. Напомню, что результатом прямого произведения T1 и T2 является таблица, и содержащая все комбинации записей из T1 и T2. То есть если в T1 есть запись r1 в T2 есть запись r2, то в выходной таблице имеется объединенная запись (r1, r2). Обратите внимание, что эта объединенная запись фактически такая же, как (r2, r1), потому что порядок следования полей в записи не имеет значения. А так как (r2, r1) – это запись, созданная произве-

дением T2 и T1, то оператор прямого произведения обладает свойством коммутативности. То есть:

```
product(T1, T2) ≡ product(T2, T1)
```

Аналогично (см. упражнение 15.1) можно показать, что оператор прямого произведения обладает свойством ассоциативности. То есть:

```
product(product(T1, T2), T3) ≡ product(T1, product(T2, T3))
```

С точки зрения деревьев запросов, первая эквивалентность меняет местами левый и правый дочерние элементы узла product. Вторая эквивалентность соответствует ситуации, когда два узла product следуют друг за другом. В этом случае внутренний узел product перемещается из левого дочернего элемента внешнего узла product в правый дочерний элемент; порядок остальных дочерних элементов остается прежним. Обе эквивалентности показаны на рис. 15.1.

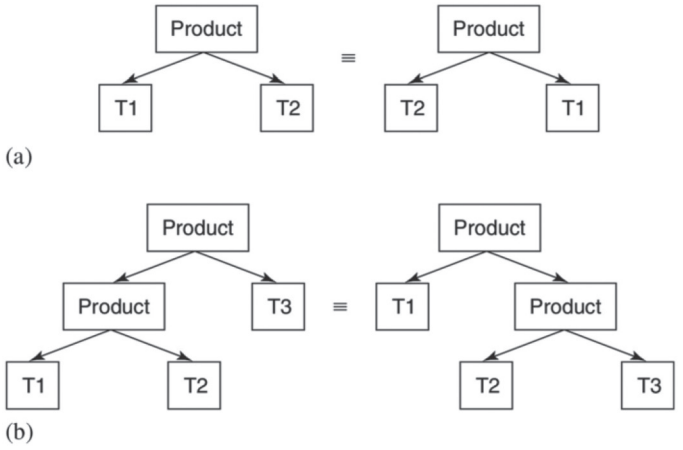


Рис. 15.1. Эквивалентности с оператором прямого произведения: (a) коммутативность; (b) ассоциативность.

Эти две эквивалентности можно многократно использовать для преобразования деревьев с узлами product. Например, взгляните на рис. 15.2, где показаны два дерева, соответствующие запросу:

```
select SName
from STUDENT, ENROLL, SECTION, COURSE, DEPT
```

Дерево на рис. 15.2a создано базовым планировщиком. Чтобы преобразовать его в дерево, показанное на рис. 15.2b, нужно выполнить два шага. На первом шаге применяется правило коммутативности к узлу product над SECTION; на втором шаге применяется ассоциативное правило к узлу product над DEPT.

Фактически можно показать (см. упражнение 15.2), что эти два правила можно использовать для преобразования любого дерева с узлами product в любое другое дерево с такими же узлами. То есть операции прямого произведения можно выполнять в любом порядке.

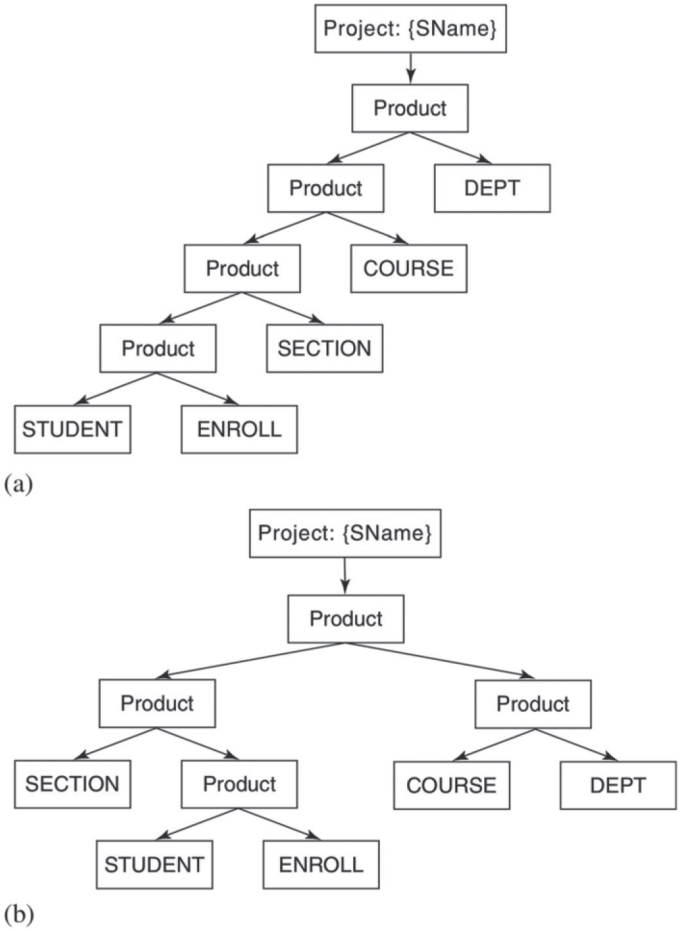


Рис. 15.2. Переупорядочивание узлов product для создания эквивалентного дерева запроса: (а) дерево, созданное базовым планировщиком; (б) результат применения ассоциативных и коммутативных преобразований

15.1.2. Расщепление селекции

Предположим, что предикат p в операторе селекции является конъюнкцией двух предикатов, p1 и p2 (p = p1 and p2). Тогда поиск записей, удовлетворяющих предикату p, можно выполнить в два этапа: сначала найти записи, удовлетворяющие предикату p1, а затем в полученном наборе найти записи, удовлетворяющие предикату p2. Иначе говоря, имеет место следующая эквивалентность:

```
select(T, p1 and p2) ≡ select(select(T, p1), p2)
```

С точки зрения деревьев запросов, эта эквивалентность заменяет один узел select двумя узлами, как показано на рис. 15.3.

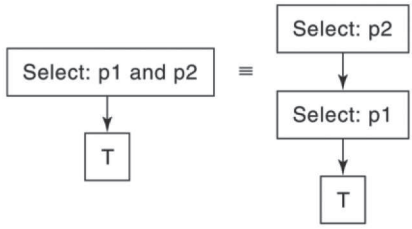


Рис. 15.3. Расщепление узла select

Многократно применяя эту эквивалентность, можно заменить единственный узел select в дереве запроса несколькими узлами, по одному для каждого конъюнктивного члена предиката. Более того, поскольку конъюнктивные члены предиката допускается переставлять как угодно, узлы select могут следовать в любом порядке.

Возможность расщепления узла select чрезвычайно полезна для оптимизации запросов, потому что каждый из «меньших» узлов select можно разместить независимо в наиболее подходящем месте в дереве запроса. По этой причине оптимизатор запросов стремится расщепить предикаты на как можно больше конъюнктивных членов. Это достигается путем преобразования каждого предиката в *конъюнктивную нормальную форму (КНФ)*. Предикат находится в КНФ, если представляет собой конъюнкцию предикатов, ни один из которых не содержит оператора `AND`.

Операторы `AND` в предикате, имеющем конъюнктивную нормальную форму, всегда являются самыми внешними. Например, рассмотрим следующий SQL-запрос:

```
select SName from STUDENT
where (MajorId=10 and SId=3) or (GradYear=2018)
```

В данном случае предикат в предложении `where` не является предикатом в конъюнктивной нормальной форме, потому что оператор `AND` находится внутри оператора `OR`. Однако всегда можно использовать законы де Моргана и сделать оператор `AND` внешним. В этом случае получится такой эквивалентный запрос:

```
select SName from STUDENT
where (MajorId=10 or GradYear=2018) and (SId=3 or GradYear=2018)
```

Предикат в этом запросе имеет два конъюнктивных члена, которые теперь можно расщепить.

15.1.3. Перемещение операторов селекции внутри дерева

Следующий запрос извлекает имена всех студентов, изучающих математику:

```
select SName from STUDENT, DEPT
where DName = 'math' and MajorId = DId
```

Предикат в предложении `where` здесь имеет конъюнктивную нормальную форму и содержит два конъюнктивных члена. На рис. 15.4а изображено дерево запроса, созданное базовым планировщиком и модифицированное так,

что в нем теперь два узла select. Сначала рассмотрим селекцию по DName. Узел product, следующий ниже в дереве, выводит все комбинации записей STUDENT и DEPT, а узел select сохраняет только те комбинации, в которых DName имеет значение «math». В результате получается тот же набор записей, как если бы сначала была выбрана запись DEPT, соответствующая кафедре математики, а затем получены все комбинации записей STUDENT с этой записью. Другими словами, поскольку селекция применяется только к таблице DEPT, эту операцию можно «протолкнуть» вниз по дереву, внутрь прямого произведения, и получить эквивалентное дерево, как показано на рис. 15.4b.

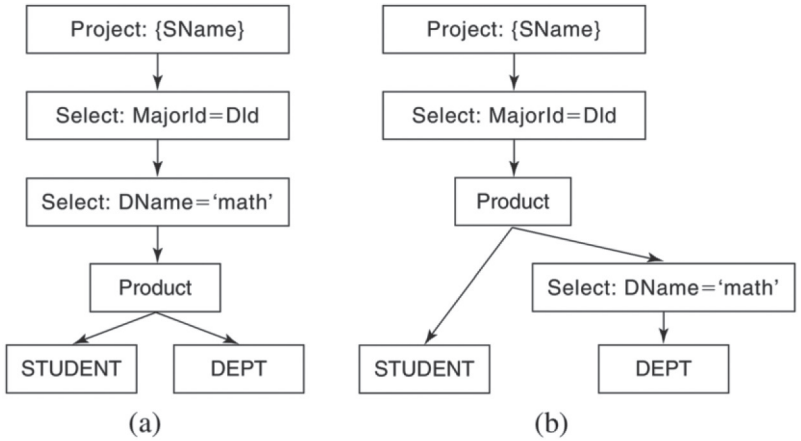


Рис. 15.4. Проталкивание узла select вниз по дереву запроса

Теперь рассмотрим предикат соединения MajorId=DId. Эту операцию селекции нельзя втолкнуть в прямое произведение, потому что в предикате упоминаются поля из обеих таблиц, STUDENT и DEPT. Например, если протолкнуть эту операцию вниз, поместив над узлом STUDENT, получится бессмысленный запрос, потому что операция селекции будет ссылаться на поле, которого нет в таблице STUDENT.

Следующая эквивалентность обобщает сказанное выше. Она справедлива, только когда предикат p ссылается на поля в T1:

```
select(product(T1, T2), p) ≡ product(select(T1, p), T2)
```

Эта эквивалентность изображена на рис. 15.5.

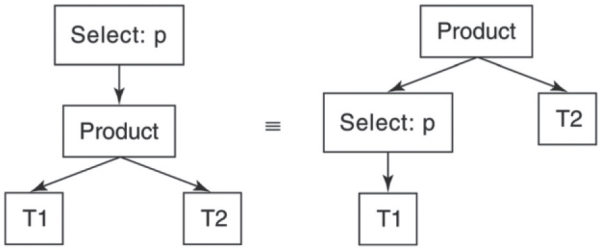


Рис. 15.5. Проталкивание узла select внутрь прямого произведения

Эту эквивалентность можно многократно применять к выбранному узлу, продвигая его вниз по дереву запроса, пока это возможно. Например, взгляните на рис. 15.6, где изображено дерево запроса из листинга 15.1:

Листинг 15.1. Запрос SQL, возвращающий имена студентов, не сдавших экзамен за курс математики в 2018 году

```
select SName
from STUDENT, ENROLL, SECTION, COURSE, DEPT
where SId=StudentId and SectionId=SectId and CourseId=CId
      and DeptId=DId and DName='math' and Grade='F'
      and YearOffered=2018
```

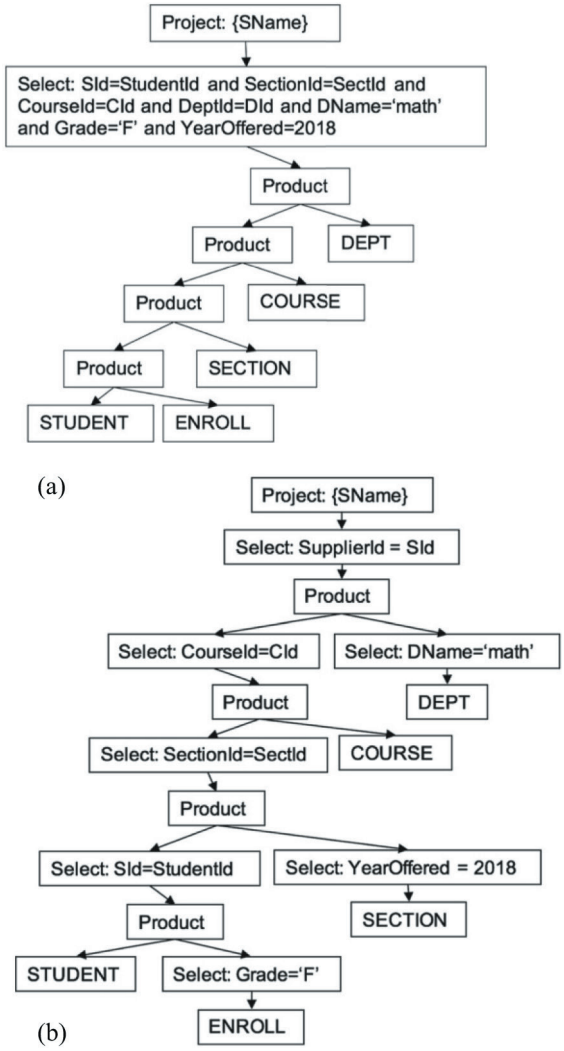


Рис. 15.6. Проталкивание некоторых операций селекции вниз по дереву запроса: (a) дерево запроса, созданное базовым планировщиком; (b) дерево запроса, полученное после проталкивания узлов select

Этот запрос возвращает имена студентов, которые не смогли сдать экзамен по курсу математики в 2018 году. На рис. 15.6a и 15.6b изображены два эквивалентных дерева для этого запроса. На рис. 15.6a показано дерево запроса, созданное базовым планировщиком, а на рис. 15.6b – полученное в результате расщепления узла select и перемещения меньших узлов select вниз по дереву.

Эквивалентность, показанная на рис. 15.5, действует также в обратном направлении и позволяет перемещать узлы select вверх по дереву, выше узлов product. Более того, легко показать, что узел select всегда можно переместить относительно другого узла select в любом направлении, а переместить его ниже узла project или groupby можно, только если это имеет смысл (см. упражнение 15.4). Отсюда следует, что узел select можно поместить в любое место в дереве запроса, при условии что его предикат упоминает только поля из нижележащего поддерева.

15.1.4. Выявление операций соединения

Как рассказывалось выше, оператор соединения определяется в терминах операторов селекции и прямого произведения:

```
join(T1, T2, p) ≡ select(product(T1, T2), p)
```

Согласно этой эквивалентности, пару узлов select и product можно преобразовать в один узел join. Например, на рис. 15.7 показан результат такого преобразования дерева на рис. 15.6b.

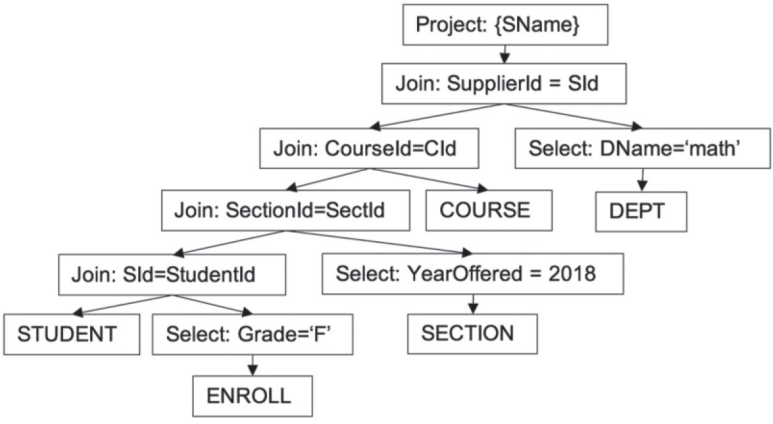


Рис. 15.7. Замена узлов select-product в дереве на рис. 15.6b узлами join

15.1.5. Добавление проекций

Узел project можно добавить выше любого узла в дереве запроса, при условии что все поля в его проекции упоминаются в узлах-предках. Это преобразование обычно используется для уменьшения объема входных данных в узлах дерева запроса перед материализацией.

Например, на рис. 15.8 изображено дерево запроса из рис. 15.7, в которое добавлены дополнительные узлы project, чтобы исключить ненужные поля как можно раньше.

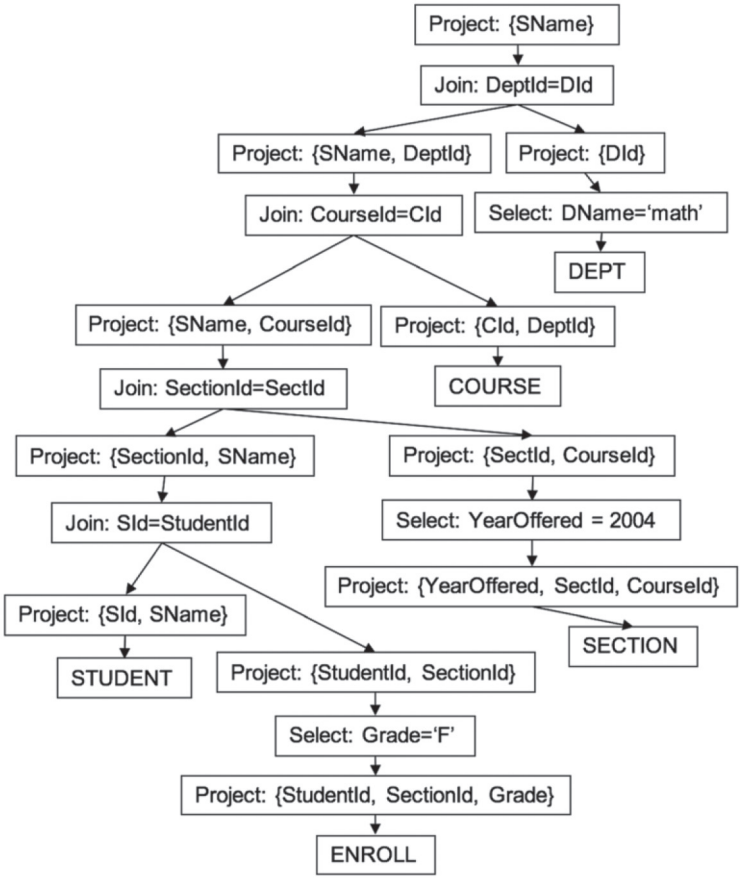


Рис. 15.8. Добавление проекций в дерево запроса из рис. 15.7

15.2. Необходимость оптимизации запросов

Получив SQL-запрос, планировщик должен выбрать подходящий план для его обработки. Процедура составления плана включает в себя два этапа:

- выбор дерева запроса реляционной алгебры, соответствующего SQL-запросу;
- выбор реализации для каждого узла в дереве запроса.

В общем случае SQL-запрос может иметь много эквивалентных деревьев запроса, и каждый узел каждого дерева может быть реализован несколькими способами. То есть планировщик может выбирать между разными потенциальными планами, и было бы неплохо, если бы он выбрал наиболее эффективный. Но нужно ли это? Ведь для поиска лучшего плана может потребоваться выполнить много работы, и, прежде чем ее начинать, стоит убедиться, что она действительно стоит затраченных усилий. Что плохого в использовании простого алгоритма планирования, представленного в главе 10?

Как оказывается, разные планы для одного и того же запроса могут чрезвычайно сильно отличаться по количеству обращений к блокам. Рассмотрим, например, два дерева запроса, соответствующих SQL-запросу в листинге 15.2, который извлекает оценки, полученные студентом Joe в течение 2020 года. На рис. 15.9a изображено дерево запроса, созданное базовым планировщиком, а на рис. 15.9b – эквивалентное дерево.

Листинг 15.2. Запрос SQL, извлекающий оценки, полученные студентом Joe в течение 2020 года

```
select Grade from STUDENT, SECTION, ENROLL
where SId=StudentId and SectId=SectionId
      and SName='joe' and YearOffered=2020
```

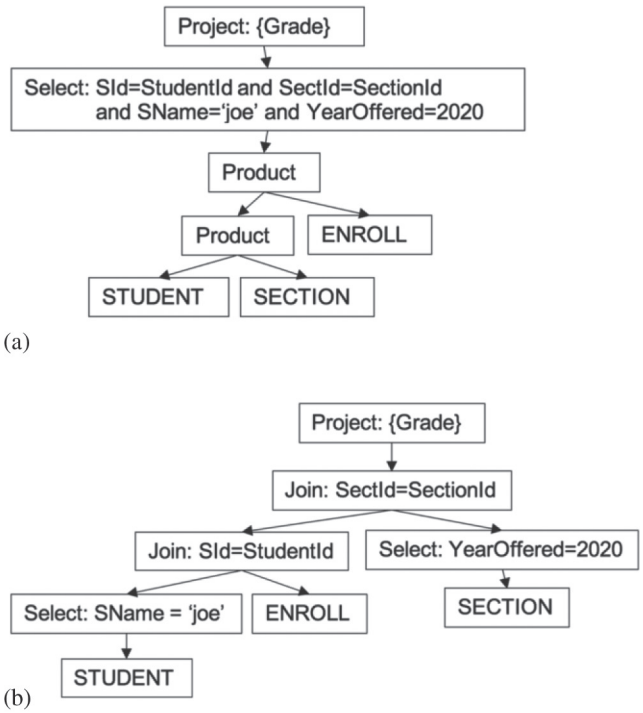


Рис. 15.9. Какое дерево запроса дает лучший план: (a) дерево запроса, созданное базовым планировщиком; (b) эквивалентное дерево?

Рассмотрим план в части (a). В соответствии со статистиками в табл. 7.2 стоимость этого плана вычисляется следующим образом. Чтобы найти прямое произведение между STUDENT и SECTION, потребуется прочитать 45 000 × 25 000 = 1 125 000 000 записей и выполнить 4500 + (45 000 × 2500) = 112 504 500 обращений к блокам. Чтобы найти прямое произведение с ENROLL, потребуется выполнить 112 504 500 + (1 125 000 000 × 50 000) = 56 250 112 504 500 обращений к блокам. Узлы select и project не требуют дополнительных обращений к блокам. Таким образом, этот план требует более 56 триллионов обращений к блокам! Если допустить, что для обращения каждому блоку требуется всего 1 мс,

движку базы данных потребуется около 1780 лет, чтобы получить ответ на этот запрос.

Теперь рассмотрим дерево запроса в части (b). Допустим, что есть только один студент с именем «Joe». В этом случае селекция из таблицы STUDENT потребует выполнить 4500 обращений к блокам и выведет 1 запись. Соединение с ENROLL потребует 4500 + (1 50 000) = 54 500 обращений к блокам и выведет 34 записи. А соединение с SECTION потребует 54 500 + (34 × 2500) = 139 500 обращений к блокам. При том же времени обращения к блоку в 1 мс для выполнения этого плана потребуется около 2,3 минуты.

Стоимость уменьшилась с 1780 лет до 2,3 минуты! Эта разница наглядно показывает, насколько безнадежен базовый алгоритм планирования. Никто не может позволить себе ждать тысячу лет, чтобы получить ответ на свой вопрос[1]. Чтобы движок базы данных имел практическую ценность, его планировщик должен быть достаточно сложным и уметь строить разумные деревья запросов.

Время выполнения 2,3 минуты вполне приемлемо, но планировщик может добиться большего, используя другие реализации узлов в дереве запроса. Снова рассмотрим дерево запроса в части (b) и предположим, что для ENROLL построен индекс по полю `StudentId`. В этом случае становится возможным план, представленный в листинге 15.3.

Листинг 15.3. Эффективный план для дерева, изображенного на рис. 15.9b

```
SimpleDB db = new SimpleDB("studentdb");
MetadataMgr mdm = db.mdMgr();
Transaction tx = db.newTx();

// план для узла STUDENT
Plan p1 = new TablePlan(tx, "student", mdm);

// план для узла select над узлом STUDENT
Predicate joepred = new Predicate(...); //sname='joe'
Plan p2 = new SelectPlan(p1, joepred);

// план для узла ENROLL
Plan p3 = new TablePlan(tx, "enroll", mdm);

// план соединения STUDENT и ENROLL с использованием индекса
Map<String,IndexInfo> indexes = mdm.getIndexInfo("enroll", tx);
IndexInfo ii = indexes.get("studentid");
Plan p4 = new IndexJoinPlan(p2, p3, ii, "sid");

// план для узла SECTION
Plan p5 = new TablePlan(tx, "section", mdm);

// план для узла select над узлом SECTION
Predicate sectpred = new Predicate(...); //yearoffered=2020
Plan p6 = new SelectPlan(p5, sectpred);
```

[1] Все, конечно, зависит от вопроса. Ответа на некоторые приходится ждать семь с половиной миллионов лет, как вы наверняка знаете из книг Дугласа Адамса. – *Прим. ред.*

```
// план многобуферного прямого произведения результата соединения и SECTION
Plan p7 = new MultiBufferProductPlan(tx, p4, p6);

// план для узла select над узлом многобуферного произведения
Predicate sectpred = new Predicate(...); //sectid=sectionid
Plan p8 = new SelectPlan(p7, sectpred);

// план для узла project
List<String> fields = Arrays.asList("grade");
Plan p9 = new ProjectPlan(p8, fields);
```

Большинство планов в листинге 15.3 использует базовые классы планов, представленные в главе 10. Исключениями являются `p4` и `p7`. План `p4` выполняет соединение с использованием индекса. Для каждой выбранной записи STUDENT выполняется поиск по индексу `StudentId`, чтобы найти соответствующие записи в ENROLL. План `p7` выполняет соединение с использованием многобуферного прямого произведения. Он материализует таблицу справа (подмножество разделов курсов, которые начали предлагаться с 2020 года), делит их на фрагменты и находит прямое произведение между `p4` и этими фрагментами.

Давайте посчитаем, сколько обращений к блокам потребует этот план. План `p2` выполнит 4500 обращений к блокам и выведет 1 запись. При вычислении соединения с использованием индекса потребуется прочитать таблицу ENROLL один раз для каждой из 34 записей, соответствующих записи Joe в таблице STUDENT; то есть для вычисления соединения понадобится 34 дополнительных обращения к блокам, и в результате получится 34 записи. План `p6` (который находит разделы курсов, которые предлагаются в 2020 году) потребует 2500 обращений к блокам и выведет 500 записей. Многобуферное прямое произведение материализует эти записи, что потребует еще 50 обращений, чтобы создать временную таблицу, занимающую 50 блоков. Если предположить, что в системе баз данных имеется не менее 50 доступных буферов, то эта временная таблица уместится в один фрагмент, поэтому прямому произведению потребуется еще 50 обращений к блокам для сканирования временной таблицы, в дополнение к затратам на вычисление записей слева. Остальные узлы плана не требуют дополнительных обращений к блокам. Таким образом, для выполнения плана необходимо всего 7134 обращения к блокам, что займет чуть больше 7 секунд.

Другими словами, внимательное отношение к выбору реализаций узлов сократило время выполнения запроса почти в 20 раз при использовании того же дерева запроса. Это сокращение не выглядит таким впечатляющим, как при выборе другого дерева запроса, но оно существенно и важно. Коммерческая система баз данных, которая в 20 раз медленнее конкурентов, недолго просуществует на рынке.

15.3. Структура оптимизатора запросов

Получив SQL-запрос, планировщик должен попытаться найти для него план, требующий наименьшего количества обращений к блокам. Этот процесс называется *оптимизацией запроса*.

Но как планировщик сможет определить этот план? Полный перечень всех возможных планов может получиться ошеломляющим: если запрос содержит `n` операций прямого произведения, то всего будет доступно `(2n)!/n!` способов их упорядочения, то есть количество эквивалентных планов растет сверхэкспоненциально с увеличением размера запроса. И это даже без учета разных способов размещения узлов других операторов и выбора разных реализаций для каждого узла.

Один из способов справиться с этой сложностью – выполнить оптимизацию в два независимых этапа:

- этап 1: найти наиболее перспективное дерево запроса, то есть такое дерево, которое вероятнее всего даст наиболее эффективный план;
- этап 2: выбрать лучшую реализацию для каждого узла в этом дереве.

Выполняя эти этапы независимо, планировщик сокращает количество вариантов, доступных для выбора, что позволяет сделать каждый этап более простым и целенаправленным.

На каждом из этих этапов оптимизации планировщик может еще больше уменьшить сложность, используя *эвристики*, помогающие ограничить набор рассматриваемых деревьев и планов. Например, обычно планировщики запросов используют эвристику «выполнить селекции как можно раньше». Как показывает опыт, в оптимальном плане запроса узлы select всегда (или почти всегда) размещаются как можно раньше (ниже в дереве). Следуя этой эвристике, планировщик запросов не должен учитывать любые другие варианты размещения узлов select в рассматриваемых деревьях запросов.

Оба этапа оптимизации запросов и соответствующие эвристики рассматриваются в следующих двух разделах.

15.4. Поиск наиболее перспективного дерева запроса

15.4.1. Стоимость дерева запроса

Первый этап оптимизации запроса – поиск «наиболее перспективного» дерева запроса, то есть дерева, которое, по мнению планировщика, имеет план с наименьшей стоимостью. Причина, по которой планировщик не может фактически определить лучшее дерево, заключается в том, что информация о стоимости недоступна на первом этапе. Количество обращений к блокам – это характеристика плана, но до второго этапа не существует никаких планов. Следовательно, планировщику нужен способ сравнения деревьев запросов без фактического вычисления количества обращений к блокам. Важно отметить, что:

- почти все обращения к блокам в запросе связаны с операциями прямого произведения и соединения;
- количество обращений к блокам в этих операциях зависит от размера входных данных[1].

[1] Исключением является соединение с использованием индексов, стоимость которого практически не зависит от размера индексированной таблицы. Но на этом этапе наш планировщик игнорирует это исключение.

Поэтому планировщик определяет *стоимость* дерева запроса как сумму размеров входных данных для каждого узла прямого произведения и соединения в дереве.

Например, давайте посчитаем стоимость двух деревьев запроса на рис. 15.9. Эти деревья имеют два узла product, поэтому суммируем размеры входных данных в каждом из них. Результаты представлены в табл. 15.1. Согласно этим результатам, второе дерево запросов намного лучше первого.

Таблица 15.1. Вычисление стоимости двух деревьев запроса

Дерево запроса	Размер входных данных для нижнего узла product	Размер входных данных для верхнего узла product	Общая стоимость дерева
Рис. 15.9a	45 000 + 25 000	1 125 000 000 + 1 500 000	1 126 570 000
Рис. 15.9b	1 + 1 500 000	34 + 25 000	1 525 035

Стоимость дерева запроса можно рассматривать как «первую прикидку» времени его выполнения. Стоимость не оценивает количество обращений к блокам, но помогает определить относительную стоимость двух деревьев. В частности, для двух рассматриваемых деревьев запроса можно ожидать, что наиболее эффективным получится план, построенный на основе дерева с более низкой стоимостью. Это не всегда верно (см. упражнение 15.8). Однако опыт показывает, что в большинстве случаев это предположение выполняется, и даже если это не так, план для самого дешевого дерева обычно оказывается достаточно хорошим.

15.4.2. Проталкивание узлов select вниз по дереву

Для поиска наиболее перспективного дерева запроса планировщик использует эвристики. Первая эвристика касается размещения узлов select в дереве. Предикат селекции определяется в предложении `where` SQL-запроса. Напомню, что формулы эквивалентности в разделе 15.1.2 позволяют планировщику разместить узел select в любом месте в дереве, при условии что предикат будет иметь смысл в этом месте.

При каком размещении узлов select получится дерево с наименьшей стоимостью? На выходе узла select не может быть больше записей, чем на входе. Поэтому, если поместить узел select под узлом product или join, эти узлы, вероятно, получат меньше данных, и общая стоимость дерева уменьшится. Это рассуждение ведет нас к следующей эвристике:

- эвристика 1: планировщик должен учитывать только те деревья запроса, узлы select в которых сдвинуты вниз, насколько это возможно.

Допустим, что после проталкивания всех узлов select вниз по дереву у нас появились два узла select, следующие друг за другом. Эвристика 1 не определяет порядок следования этих узлов. Однако их порядок не влияет на стоимость дерева, поэтому планировщик может выбрать любой вариант или объединить эти два узла в один.

Эвристика 1 упрощает задачу планировщика – следуя ей, ему не нужно беспокоиться о взаимном размещении узлов select. Учитывая положение других операторов в плане запроса, размещение этих узлов четко определено.

15.4.3. Замена пар узлов select и product узлами join

Рассмотрим предикат соединения, включающий поля из таблиц T1 и T2. Когда узел select, содержащий этот предикат, перемещается вниз по дереву, он достигает определенной точки в дереве, а именно узла product, для которого T1 находится в одном поддереве, а T2 – в другом. Эту пару узлов select и product можно заменить одним узлом join. В результате получаем вторую эвристику:

- эвристика 2: планировщик должен заменить каждую пару узлов select и product в дереве запроса одним узлом join.

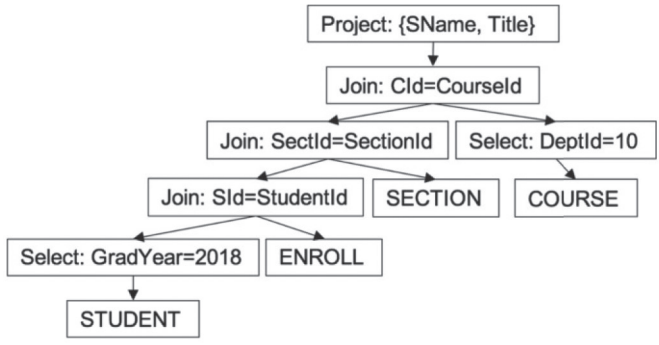
Эта эвристика не меняет стоимости дерева запроса, но является важным шагом в поиске лучшего плана. В этой книге было исследовано несколько эффективных реализаций оператора соединения. Выявляя соединения в дереве запросов, планировщик сможет выбрать лучшую реализацию на втором этапе оптимизации.

15.4.4. Использование односторонних левых деревьев запросов

Планировщик должен выбрать порядок выполнения операций произведения (соединения). Для примера рассмотрим запрос в листинге 15.4. Этот SQL-запрос извлекает имена студентов, окончивших обучение в 2018 году, и названия разделов в курсе математики, которые они прослушали. На рис. 15.10a–15.10e изображены пять эквивалентных деревьев для этого запроса.

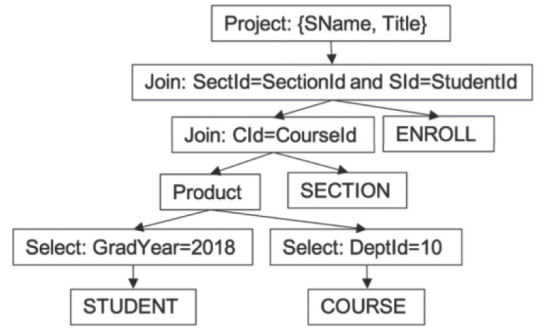
Листинг 15.4. SQL-запрос, возвращающий имена студентов, окончивших обучение в 2018 году, и названия разделов в курсе математики, которые они прослушали

```
select SName, Title
from STUDENT, ENROLL, SECTION, COURSE
where SId=StudentId and SectId=SectionId
      and CId=CourseId and GradYear=2018 and DeptId=10
```

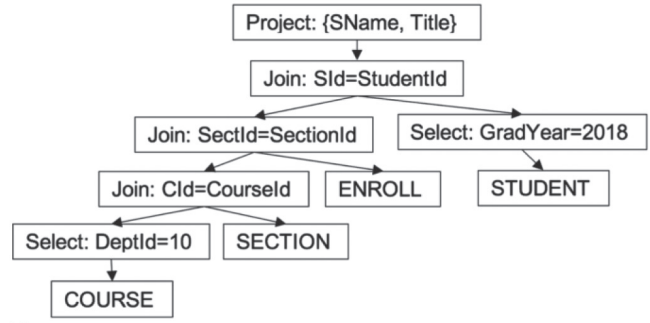


(a)

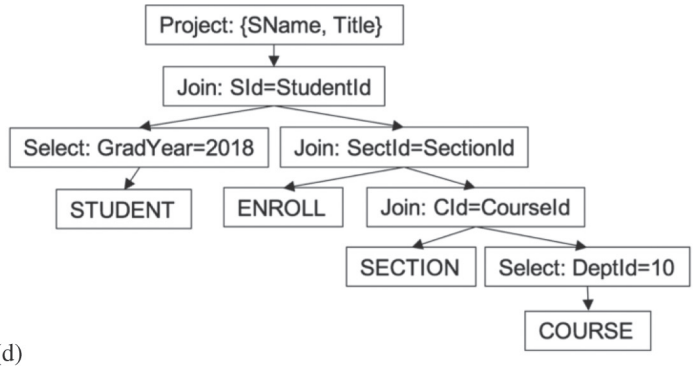
Рис. 15.10. Эквивалентные деревья запроса с разным строением: (a) одностороннее левое дерево запроса



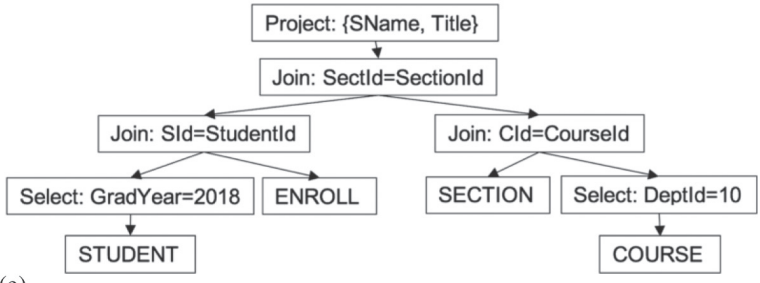
(b)



(c)



(d)



(e)

Рис. 15.10. (b) Еще одно одностороннее левое дерево запроса; (c) и еще одно односторонее левое дерево запроса; (d) одностороннее правое дерево запроса; (e) кустистое дерево запроса

Эти деревья имеют разное *строение*. Деревья (a)–(c) называются *односторонними левыми*, потому что правые ветви узлов product (join) не содержат других узлов product (join). Аналогично, дерево на рис. 15.10d называется *односторонним правым*. Дерево на рис. 15.10e называется *кустистым*, потому что оно не является односторонним. Многие планировщики запросов используют следующую эвристику:

- эвристика 3: планировщик должен рассматривать только односторонние левые деревья запросов.

Обоснование этой эвристики неочевидно. Рассмотрим для примера табл. 15.2, где показаны расчеты стоимости каждого дерева с использованием статистик из табл. 7.2. Наименьшую стоимость из деревьев на рис. 15.10 имеет кустистое дерево. Более того, это дерево оказывается наиболее многообещающим (см. упражнение 15.9). Так почему же планировщик должен игнорировать большое множество деревьев, среди которых может оказаться самое перспективное? Тому есть две причины.

Таблица 15.2. Стоимость деревьев на рис. 15.10

Дерево	Стоимость нижнего соединения	Стоимость среднего соединения	Стоимость верхнего соединения	Общая стоимость
(a)	1 500 900	55 000	30 013	1 585 913
(b)	913	36 700	3 750 000	3 787 613
(c)	25 013	1 500 625	38 400	1 564 038
(d)	25 013	1 500 625	38 400	1 564 038
(e)	1 500 900 (левое соединение)	25 013 (правое соединение)	30 625	1 556 538

Первая причина заключается в том, что односторонние левые деревья часто имеют наиболее эффективные планы, даже при не самой низкой стоимости. Вспомните алгоритмы соединения, которые вы видели. Все они показывают лучшую производительность, когда справа находится хранимая таблица. Например, многобуферное прямое произведение материализует таблицу справа, поэтому, если справа указана хранимая таблица, дополнительная материализация не потребуется. А соединение с использованием индексов возможно только тогда, когда правый операнд представлен хранимой таблицей. Следовательно, выбирая одностороннее левое дерево, планировщик увеличивает вероятность использования более эффективных реализаций при создании окончательного плана. Как показывает опыт, лучший план для одностороннего левого дерева запроса обычно оказывается самым оптимальным или достаточно близким к нему.

Вторая причина – удобство. Если в запросе есть `n` узлов product (join), то для такого запроса имеется всего `n!` односторонних левых деревьев, что намного меньше `(2n)!/n!` всех возможных деревьев. То есть эвристика 3 позволяет планировщику работать гораздо быстрее (что важно) с небольшим риском получить плохой план.

Односторонее левое дерево можно получить, перечислив таблицы по порядку. Первой в этом списке следует таблица, находящаяся слева от самого нижнего узла product (join), а последующие берутся с правой стороны каждого узла

product (join), размещенного выше в дереве. Этот порядок называется *порядком соединения* одностороннего левого дерева.

Например, дерево на рис. 15.10a имеет порядок соединения (STUDENT, ENROLL, SECTION, COURSE), а дерево на рис. 15.10b имеет порядок соединения (STUDENT, COURSE, SECTION, ENROLL). Так эвристика 3 упрощает работу планировщика запросов – ему остается только определить наилучший порядок соединения. Эвристики 1–3 полностью определяют дерево запроса, оптимальное с точки зрения планировщика.

15.4.5. Выбор порядка соединения с использованием эвристик

Задача поиска лучшего порядка соединения для данного запроса является наиболее важной частью процесса оптимизации запроса. Под «наиболее важным» я имею в виду следующее:

- выбор порядка соединения существенно влияет на стоимость получаемого дерева запроса. Пример представлен на рис. 15.10, где дерево (a) намного лучше, чем дерево (b);
- существует так много разных вариантов порядка соединения, что обычно невозможно изучить их все. В частности, запрос, в котором упоминается `n` таблиц, может иметь `n!` разных вариантов порядка соединения.

Таким образом, планировщик должен тщательно выбирать, какие порядки соединений рассматривать, чтобы не остаться с плохим планом. Для определения хорошего порядка соединения были разработаны два общих подхода: с использованием эвристик и перебором всех возможных порядков. В этом разделе исследуется подход на основе эвристик, а в следующем – полный перебор.

В подходе с использованием эвристик порядок соединения строится постепенно. То есть сначала планировщик выбирает одну из таблиц, которая будет первой в порядке соединения. Затем выбирает следующую таблицу – и повторяет этот процесс до получения полного порядка соединения.

Следующая эвристика помогает планировщику отсеять «явно плохие» варианты порядка соединения:

- эвристика 4: каждая таблица в порядке соединения должна по возможности соединяться с ранее выбиравшимися таблицами.

Проще говоря, эта эвристика утверждает, что все узлы product в дереве запроса должны соответствовать соединениям. Дерево запроса на рис. 15.10b нарушает эту эвристику, потому что начинается с вычисления прямого произведения таблиц STUDENT и COURSE.

Чем плохи варианты порядка соединения, нарушающие эвристику 4? Напомню, что роль предиката соединения состоит в том, чтобы отфильтровать ненужные выходные записи, возвращаемые операцией прямого произведения. Поэтому, если дерево запроса содержит узел product, не являющийся соединением, ненужные записи из его выходной таблицы будут подниматься выше по дереву, пока не встретится предикат соединения. Например, снова рассмотрим дерево запросов на рис. 15.10b. Прямое произведение таблиц STUDENT и COURSE дает 11 700 выходных записей, потому что каждая из 13 записей COURSE, соответ-

ствующих кафедре математики, повторяется 900 раз (по одному для каждого студента, выпущенного в 2018 году). Когда эта выходная таблица соединяется с SECTION, каждой записи COURSE соответствует своя запись SECTION; однако эти совпадения повторяются 900 раз. В результате на выходе этого соединения получается в 900 раз больше записей, чем должно быть. И только когда в порядок соединения добавляется таблица ENROLL, предикат соединения со STUDENT наконец удаляет ненужные записи.

Этот пример наглядно показывает, что выходные данные дерева запроса, включающего узел product, могут иметь небольшой объем, но в конечном итоге повторение, вызванное прямым произведением, дает дерево с очень высокой стоимостью. По этой причине эвристика 4 требует избегать операций прямого произведения, если это возможно. Конечно, если пользователь отправит запрос, в котором соединены не все таблицы, то избежать появления узла product не удастся. В этом случае эвристика гарантирует, что такой узел будет находиться максимально высоко в дереве и повторение даст минимально возможный негативный эффект.

Эвристика 4 получила широкое применение. Конечно, можно найти запросы, для которых наиболее перспективное дерево нарушает эту эвристику (см. упражнение 15.11), но на практике такие запросы встречаются редко.

Пришло время выбирать, какую таблицу поставить первой и какую из таблиц, участвующих в соединениях, поставить следующей. Это сложные вопросы. Разработчики баз данных предложили множество эвристик, но так и не пришли к единому мнению – какая из них лучше. Я рассмотрю два возможных варианта, которые назову эвристиками 5a и 5b:

- эвристика 5a: первой следует выбирать таблицу, которая дает наименьшее количество выходных записей.

Эта эвристика – наиболее простая и понятная. Стоимость дерева запроса определяется как сумма размеров промежуточных выходных таблиц, поэтому желательно минимизировать эту сумму, минимизируя каждую из этих таблиц.

Применим эту эвристику к запросу в листинге 15.4. Первой таблицей в порядке соединения будет COURSE, потому что предикат селекции сокращает ее до 13 записей. Остальные таблицы определяются эвристикой 4. SECTION – это единственная таблица, которая соединяется с COURSE. Затем следует ENROLL – единственная таблица, которая соединяется с SECTION. В результате таблица STUDENT остается последней. Получившееся в результате дерево запросов представлено на рис. 15.10c.

Альтернативная эвристика:

- эвристика 5b: первой следует выбирать таблицу с наиболее селективным предикатом.

Эвристика 5b вытекает из понимания того, что чем ниже в дереве запроса находится предикат селекции, тем большее влияние он будет оказывать. Например, рассмотрим дерево запроса на рис. 15.10a и его предикат селекции по полю в таблице STUDENT. Этот предикат имеет очевидное преимущество: он уменьшает количество записей STUDENT и снижает стоимость узла join, расположенного непосредственно над ним. Но есть еще более важное преимущество – этот предикат также сокращает результат этого соединения с

1 500 000 до 30 000 записей, уменьшая стоимость каждого последующего узла join в дереве. Другими словами, экономия, полученная узлом select, увеличивается на всем протяжении движения вверх по дереву. Напротив, предикат селекции по полю в таблице COURSE в верхней части дерева оказывает гораздо меньшее влияние.

Поскольку нижние предикаты селекции в дереве запроса оказывают наибольшее влияние на стоимость, имеет смысл первой выбрать таблицу, предикат которой сокращает размер выходных данных в большее количество раз. Именно это и делает эвристика 5b. Например, дерево запросов на рис. 15.10a удовлетворяет этой эвристике. Первой таблицей в порядке соединения следует STUDENT, потому что ее предикат селекции уменьшает размер входной таблицы в 50 раз, тогда как предикат селекции по полю в таблице COURSE уменьшает ее только в 40 раз. Остальные таблицы в порядке соединения, как и раньше, определяются с помощью эвристики 4.

Как показывает этот пример, использование эвристики 5b дает более дешевое дерево запроса, чем эвристика 5a. Это типично. Исследования (такие как Swami [1989]) показали, что хотя эвристика 5a имеет интуитивный смысл и дает разумные деревья запросов, эти деревья, как правило, имеют более высокую стоимость, чем те, которые дает эвристика 5b.

15.4.6. Выбор порядка соединения полным перебором

Эвристики 4 и 5 обычно дают хороший порядок соединения, но не гарантируют получение лучшего. Если разработчик движка баз данных хочет быть уверенным, что его планировщик найдет оптимальный порядок соединения, у него на выбор есть только одна альтернатива – выполнить полный перебор всех вариантов. Эта стратегия рассматривается в данном разделе.

Запрос, ссылающийся на n таблиц, может иметь до $n!$ вариантов порядка соединения. Сократить время поиска наиболее перспективного порядка соединения до $O(2^n)$ можно с помощью алгоритмического метода, известного как *динамическое программирование*. Если число n достаточно мало (скажем, не более 15 или 20 таблиц), то этот алгоритм достаточно эффективен для практического применения.

Для иллюстрации экономии времени с помощью этого метода рассмотрим запрос, соединяющий все пять таблиц из университетской базы данных. Вот четыре из 120 возможных вариантов порядка соединения:

```
(STUDENT, ENROLL, SECTION, COURSE, DEPT)
(STUDENT, SECTION, ENROLL, COURSE, DEPT)
(STUDENT, ENROLL, SECTION, DEPT, COURSE)
(STUDENT, SECTION, ENROLL, DEPT, COURSE)
```

Первые два варианта отличаются друг от друга только таблицами на втором и третьем местах. Допустим, мы определили, что неполный порядок соединения (STUDENT, ENROLL, SECTION) имеет меньшую стоимость, чем (STUDENT, SECTION, ENROLL). Из этого, не выполняя дальнейших вычислений, можно заключить, что первый вариант должен иметь меньшую стоимость, по сравнению со вторым. Более того, нам известно, что третий вариант требует меньшего количества обращений к блокам, чем четвертый. И вообще, мы знаем, что любой порядок соединения, начинающийся с таблиц (STUDENT, SECTION, ENROLL), не стоит рассматривать.

Алгоритм динамического программирования использует переменную-массив с именем lowest, содержащую информацию обо всех возможных множествах таблиц. Если принять, что S – это множество таблиц, то элемент lowest[S] содержит три значения:

- порядок соединения таблиц в S с наименьшей стоимостью;
- стоимость дерева запроса, соответствующего этому порядку соединения;
- количество записей на выходе этого дерева запроса.

Алгоритм начинается с вычисления lowest[S] для всех множеств из двух таблиц, затем для всех множеств из трех таблиц и завершается после вычисления характеристик для множества, включающего все таблицы, упоминаемые в запросе. Оптимальным считается порядок соединения, соответствующий значению lowest[S], где S – это множество всех таблиц.

Вычисления для множеств из двух таблиц

Рассмотрим множество из двух таблиц {T1, T2}. Значение lowest[{T1, T2}] определяется вычислением стоимости дерева запроса, включающего соединение (или прямое произведение, если предикат соединения отсутствует) двух таблиц и предикаты селекции. Стоимость дерева запроса – это сумма размеров двух входных наборов записей для узла произведения или соединения. Обратите внимание, что в данном случае стоимость не зависит от порядка следования таблиц. Поэтому планировщик должен использовать какой-то другой критерий выбора первой таблицы. В качестве такого критерия разумно использовать эвристику 5a или 5b.

Вычисления для множеств из трех таблиц

Рассмотрим множество из трех таблиц {T1, T2, T3}. Порядок их соединения с наименьшей стоимостью можно получить, сравнив следующие возможные варианты порядка соединения:

```
lowest[{T2, T3}] соединяется с T1
lowest[{T1, T3}] соединяется с T2
lowest[{T1, T2}] соединяется с T3
```

Порядок соединения с наименьшей стоимостью будет сохранен как значение lowest[{T1, T2, T3}].

Вычисления для множеств из n таблиц

Теперь предположим, что в массиве lowest хранятся вычисленные значения для всех множеств из n-1 таблиц. Анализируя множество {T1, T2, ..., Tn}, алгоритм рассмотрит следующие порядки соединения:

```
lowest[{T2, T3 ,. . ., Tn}] соединяется с T1
lowest[{T1, T3 ,. . ., Tn}] соединяется с T2
...
lowest[{T1, T2 ,. . ., Tn-1}] соединяется с Tn
```

Порядок соединения с наименьшей стоимостью будет лучшим для данного запроса.

Рассмотрим пример использования алгоритмом динамического программирования для анализа запроса в листинге 15.4. На первом шаге алгоритм исследует все шесть множеств из двух таблиц, перечисленных в табл. 15.3.

Таблица 15.3. Выбор лучшего порядка соединения для запроса в листинге 15.4. Результаты для множеств из двух таблиц

S	Порядок соединения	Стоимость	Количество записей
{ENROLL,STUDENT}	(STUDENT,ENROLL) (ENROLL,STUDENT)	1 500 900 1 500 900	30 000
{ENROLL,SECTION}	(SECTION,ENROLL) (ENROLL,SECTION)	1 525 000 1 525 000	1 500 000
{COURSE,SECTION}	(COURSE,SECTION) (SECTION,COURSE)	25 500 25 500	25 000
{SECTION,STUDENT}	(STUDENT,SECTION) (SECTION,STUDENT)	25 900 25 900	22 500 000
{COURSE,STUDENT}	(COURSE,STUDENT) (STUDENT,COURSE)	1 400 1 400	450 000
{COURSE,ENROLL}	(COURSE,ENROLL) (ENROLL,COURSE)	1 500 500 1 500 500	450 000 000

Каждое множество из двух таблиц имеет два варианта соединения. Варианты соединения для каждого множества перечислены в порядке желательности. В этом примере они имеют одинаковые стоимости, поэтому степень желательности определяется в соответствии с эвристикой 5а. Первый порядок соединения для каждого множества выбирается как представитель этого множества в последующих вычислениях.

Затем алгоритм исследует четыре множества из трех таблиц. Порядки соединения для этих множеств и их стоимости перечислены в табл. 15.4. Для каждого множества имеется три варианта соединения. Первые две таблицы в каждом порядке соединения представляют порядок с наименьшей стоимостью из табл. 15.3. Варианты соединения для каждого множества перечислены в порядке увеличения стоимости, поэтому первый порядок соединения для каждого множества выбирается как лучший его представитель.

Таблица 15.4. Выбор лучшего порядка соединения для запроса в листинге 15.4. Результаты для множеств из трех таблиц

S	Порядок соединения	Стоимость	Количество записей
{ENROLL,SECTION,STUDENT}	(STUDENT,ENROLL,SECTION) (SECTION,ENROLL,STUDENT) (STUDENT,SECTION,ENROLL)	1 555 900 3 025 900 24 025 900	30 000
{COURSE,ENROLL,STUDENT}	(STUDENT,ENROLL,COURSE) (COURSE,STUDENT,ENROLL) (COURSE,ENROLL,STUDENT)	1 531 400 1 951 400 451 501 400	15 000 000
{COURSE,ENROLL,SECTION}	(SECTION,ENROLL,COURSE) (COURSE,SECTION,ENROLL) (COURSE,ENROLL,SECTION)	1 500 500 1 550 500 450 025 000	1 500 000
{COURSE,SECTION,STUDENT}	(COURSE,SECTION,STUDENT) (COURSE,STUDENT,SECTION) (STUDENT,SECTION,COURSE)	25 900 475 000 22 500 500	22 500 000

В табл. 15.5 представлены результаты исследования множеств из четырех таблиц. На этом шаге алгоритм должен рассмотреть четыре порядка соединения. Первые три таблицы в каждом варианте представляют порядок соединения с наименьшей стоимостью из табл. 15.4, а четвертая – недостающая таблица. Согласно табл. 15.5, оптимальным является порядок соединения (STUDENT, ENROLL, SECTION, COURSE).

Таблица 15.5. Выбор лучшего порядка соединения для запроса в листинге 15.4. Результаты для множеств из четырех таблиц

Порядок соединения	Стоимость
(STUDENT,ENROLL,SECTION,COURSE)	1 586 400
(COURSE,SECTION,ENROLL,STUDENT)	3 051 400
(STUDENT,ENROLL,COURSE,SECTION)	16 556 400
(COURSE,SECTION,STUDENT,ENROLL)	24 051 400

Обратите внимание, что на каждом шаге алгоритм должен вычислить значение `lowest` для каждого подмножества таблиц, которое может использоваться как префикс, потому что нет иного способа узнать, как изменится стоимость на следующем шаге. Может случиться так, что префикс с наибольшей стоимостью, полученный на одном шаге, будет приводить к порядку соединения с наименьшей стоимостью из-за особенностей таблиц, добавляемых в соединение далее.

15.5. Поиск наиболее эффективного плана

Первый этап оптимизации запроса – поиск наиболее перспективного дерева запроса. Второй этап – превращение этого дерева в эффективный план. Составляя план, планировщик должен выбрать реализацию для каждого узла в дереве запроса. Анализ узлов дерева производится снизу вверх, начиная с листьев. Преимущество восходящего движения заключается в том, что, приступая к исследованию текущего узла, планировщик уже будет иметь план с наименьшей стоимостью для каждого из поддеревьев. Благодаря этому планировщик сможет оценить каждую возможную реализацию узла, используя для вычисления стоимости ее метод `blocksAccessed`, и выбрать реализацию с наименьшей стоимостью.

Обратите внимание, что выбор реализации для каждого узла производится независимо от реализаций других узлов. В частности, планировщика не волнует, как реализованы поддеревья, его интересует только стоимость этой реализации. Подход на основе независимого выбора реализаций значительно снижает вычислительную сложность генерации плана. Если дерево запроса имеет `n` узлов и каждый узел имеет не более `k` реализаций, то планировщику необходимо проверить не более `k×n` планов, что не очень много.

Однако планировщик может ускорить создание плана, воспользовавшись эвристиками, которые обычно зависят от конкретной операции. Например, такими:

- эвристика 6: если возможно, использовать для оператора селекции реализацию с поддержкой индексов;
- эвристика 7: выбор реализации для узла join производить в следующем порядке убывания предпочтений:
  - если возможно, выбрать реализацию соединения с использованием индексов;
  - если одна из таблиц меньше, выбрать реализацию соединения хешированием;
  - иначе выбрать соединение слиянием.

Также важно учесть еще одно обстоятельство. Каждый раз, выбирая материализующую реализацию, планировщик должен следовать правилу:

- эвристика 8: планировщик должен добавить дочерний узел project в каждый узел с материализующей реализацией, чтобы удалить ненужные поля.

Эвристика 8 гарантирует, что временные таблицы, созданные материализующей реализацией, будут иметь минимальный размер. Это важно, поскольку для создания и для сканирования более крупной таблицы требуется больше обращений к блокам. Поэтому планировщик должен определить, какие поля потребуются материализованному узлу и его предкам, и вставить узел project, чтобы удалить ненужные поля из входных данных.

Например, рассмотрим дерево запроса на рис. 15.11. Это дерево возвращает оценки, полученные студентом Joe в 2020 году, и эквивалентно деревьям на рис. 15.9.

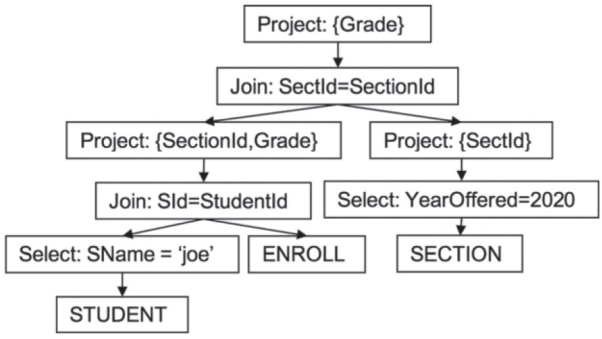


Рис. 15.11. Дерево для запроса в листинге 15.2 с дополнительными узлами project

Согласно плану в листинге 15.3, для верхнего узла join выбрана материализующая реализация соединения с многобуферным прямым произведением. Эвристика 8 требует добавлять в такие узлы join дочерние узлы project, как показано на рис. 15.11. Правый узел project особенно важен, потому что он уменьшает размер временной таблицы примерно на 75 %, что позволяет алгоритму работать с меньшим количеством фрагментов.

15.6. Объединение двух этапов оптимизации

Понять оптимизацию запросов проще, если разбить ее на два этапа: этап построения дерева запроса и этап построения плана на основе этого дерева. Однако на практике эти этапы часто совмещаются. Есть две веские причины в пользу объединения этапов оптимизации:

- *удобство*: план можно создать напрямую, без создания дерева запроса;
- *точность*: поскольку планы создаются одновременно с деревом запросов, стоимость дерева можно вычислить в терминах фактических обращений к блокам.

В этом разделе рассматриваются два примера объединения этапов оптимизации: оптимизатор SimpleDB, основанный на эвристиках, и переборный оптимизатор Селинджер.

15.6.1. Оптимизатор SimpleDB на основе эвристик

Оптимизатор запросов в SimpleDB реализован в пакете `simpledb.opt` в двух классах: `HeuristicQueryPlanner` и `TablePlanner`. Чтобы задействовать этот оптимизатор в SimpleDB, достаточно изменить метод `SimpleDB.planner` в пакете `simpledb.server`, чтобы он создавал экземпляр `HeuristicQueryPlanner` вместо `BasicQueryPlanner`.

Класс HeuristicQueryPlanner

Класс `HeuristicQueryPlanner` использует эвристику 5а для выбора порядка соединения. Для каждой таблицы, участвующей в запросе, создается объект `TablePlanner`. Когда таблица добавляется в порядок соединения, ее объект `TablePlanner` создает соответствующий план, добавляя подходящие предикаты селекции и соединения и по возможности используя индексы. Благодаря этому план строится одновременно с порядком соединения.

Определение класса `HeuristicQueryPlanner` приводится в листинге 15.5. Коллекция `tableinfo` содержит объекты `TablePlanner` для всех таблиц, упоминаемых в запросе. Планировщик начинает с выбора (и удаления) из этой коллекции объекта, соответствующего самой маленькой таблице, и использует план селекции для этой таблицы в качестве *текущего*. Затем он выбирает (и удаляет) из коллекции таблицу, дающую соединение с наименьшей стоимостью. Планировщик передает текущий план объекту `TablePlanner` выбранной таблицы, который создает и возвращает план соединения, который становится новым текущим планом. Процесс продолжается до исчерпания коллекции, после чего текущий план становится заключительным.

Листинг 15.5. Определение класса HeuristicQueryPlanner в SimpleDB

```
public class HeuristicQueryPlanner implements QueryPlanner {
    private Collection<TablePlanner> tableplanners = new ArrayList<>();
    private MetadataMgr mdm;

    public HeuristicQueryPlanner(MetadataMgr mdm) {
        this.mdm = mdm;
    }
```

```
public Plan createPlan(QueryData data, Transaction tx) {

    // Шаг 1: создать объект TablePlanner для каждой упоминаемой таблицы
    for (String tblname : data.tables()) {
        TablePlanner tp = new TablePlanner(tblname, data.pred(), tx, mdm);
        tableplanners.add(tp);
    }
    // Шаг 2: выбрать план с наименьшим размером выходной таблицы
    //        на роль начального плана для определения порядка соединения
    Plan currentplan = getLowestSelectPlan();

    // Шаг 3: повторять добавление планов в порядок соединения
    while (!tableplanners.isEmpty()) {

        Plan p = getLowestJoinPlan(currentplan);
        if (p != null)
            currentplan = p;
        else // соединение неприменимо
            currentplan = getLowestProductPlan(currentplan);
    }

    // Шаг 4: спроецировать имена полей и вернуть план
    return new ProjectPlan(currentplan, data.fields());
}

private Plan getLowestSelectPlan() {
    TablePlanner besttp = null;
    Plan bestplan = null;
    for (TablePlanner tp : tableplanners) {
        Plan plan = tp.makeSelectPlan();
        if (bestplan == null ||
              plan.recordsOutput() < bestplan.recordsOutput()) {
            besttp = tp;
            bestplan = plan;
        }
    }
    tableplanners.remove(besttp);
    return bestplan;
}

private Plan getLowestJoinPlan(Plan current) {
    TablePlanner besttp = null;
    Plan bestplan = null;
    for (TablePlanner tp : tableplanners) {
        Plan plan = tp.makeJoinPlan(current);
        if (plan != null && (bestplan == null ||
              plan.recordsOutput() < bestplan.recordsOutput())) {
            besttp = tp;
            bestplan = plan;
        }
    }
    if (bestplan != null)
        tableplanners.remove(besttp);
    return bestplan;
}
```

```
    private Plan getLowestProductPlan(Plan current) {
        TablePlanner besttp = null;
        Plan bestplan = null;
        for (TablePlanner tp : tableplanners) {
            Plan plan = tp.makeProductPlan(current);
            if (bestplan == null ||
                  plan.recordsOutput() < bestplan.recordsOutput()) {
                besttp = tp;
                bestplan = plan;
            }
        }
        tableplanners.remove(besttp);
        return bestplan;
    }

    public void setPlanner(Planner p) {
        // для использования при планировании представлений,
        // для простоты в этом коде не используется.
    }
}
```

Класс TablePlanner

Объект класса `TablePlanner` отвечает за создание плана для одной таблицы; его определение приводится в листинге 15.6. Конструктор `TablePlanner` создает план для указанной таблицы, получает информацию об индексах и сохраняет предикат запроса. Класс имеет общедоступные методы `makeSelectPlan`, `makeProductPlan` и `makeJoinPlan`.

Листинг 15.6. Определение класса TablePlanner в SimpleDB

```
class TablePlanner {
    private TablePlan myplan;
    private Predicate mypred;
    private Schema myschema;
    private Map<String,IndexInfo> indexes;
    private Transaction tx;

    public TablePlanner(String tblname, Predicate mypred,
                        Transaction tx, MetadataMgr mdm) {
        this.mypred = mypred;
        this.tx  = tx;
        myplan   = new TablePlan(tx, tblname, mdm);
        myschema = myplan.schema();
        indexes  = mdm.getIndexInfo(tblname, tx);
    }

    public Plan makeSelectPlan() {
        Plan p = makeIndexSelect();
        if (p == null)
            p = myplan;
        return addSelectPred(p);
    }
```

```
public Plan makeJoinPlan(Plan current) {
    Schema currsch = current.schema();
    Predicate joinpred = mypred.joinSubPred(myschema, currsch);

    if (joinpred == null)
        return null;
    Plan p = makeIndexJoin(current, currsch);
    if (p == null)
        p = makeProductJoin(current, currsch);
    return p;
}

public Plan makeProductPlan(Plan current) {
    Plan p = addSelectPred(myplan);
    return new MultiBufferProductPlan(current, p, tx);
}

private Plan makeIndexSelect() {
    for (String fldname : indexes.keySet()) {
        Constant val = mypred.equatesWithConstant(fldname);
        if (val != null) {
            IndexInfo ii = indexes.get(fldname);
            return new IndexSelectPlan(myplan, ii, val, tx);
        }
    }
    return null;
}

private Plan makeIndexJoin(Plan current, Schema currsch) {
    for (String fldname : indexes.keySet()) {
        String outerfield = mypred.equatesWithField(fldname);
        if (outerfield != null && currsch.hasField(outerfield)) {
            IndexInfo ii = indexes.get(fldname);
            Plan p = new IndexJoinPlan(current, myplan, ii, outerfield, tx);
            p = addSelectPred(p);
            return addJoinPred(p, currsch);
        }
    }
    return null;
}

private Plan makeProductJoin(Plan current, Schema currsch) {
    Plan p = makeProductPlan(current);
    return addJoinPred(p, currsch);
}

private Plan addSelectPred(Plan p) {
    Predicate selectpred = mypred.selectSubPred(myschema);
    if (selectpred != null)
        return new SelectPlan(p, selectpred);
    else
        return p;
}
```

```
    private Plan addJoinPred(Plan p, Schema currsch) {
        Predicate joinpred = mypred.joinSubPred(currsch, myschema);
        if (joinpred != null)
            return new SelectPlan(p, joinpred);
        else
            return p;
    }
}
```

Метод `makeSelectPlan` создает план селекции для своей таблицы. Сначала он вызывает `makeIndexSelect`, чтобы определить, можно ли использовать индекс; если да, то создает план `IndexSelect`. Затем вызывает `addSelectPred`, чтобы определить часть предиката, которая применяется к таблице, и создать для нее план селекции.

Метод `makeProductPlan` добавляет план селекции в план таблицы, а затем создает `MultiBufferProductPlan`, реализующий операцию многобуферного прямого произведения указанного плана с этим планом[1].

Метод `makeJoinPlan` сначала вызывает метод `joinPred` предиката, чтобы определить, должно ли выполняться соединение указанного плана с этим планом. Если предикат соединения отсутствует, метод возвращает `null`. Если предикат соединения существует, метод проверяет возможность создания `IndexJoinScan` для соединения с использованием индекса. Если нет, то соединение реализуется с использованием многобуферного прямого произведения с последующей селекцией.

Количество выходных записей и количество обращений к блокам

Класс `HeuristicQueryPlanner` находит план с наименьшей стоимостью, используя метод `recordsOutput`. То есть он пытается найти план с наименьшим количеством обращений к блокам, не проверяя, сколько обращений к блокам понадобится его дочерним планам. Это требует пояснений.

Как вы уже знаете, с оптимизацией на основе эвристик есть сложность: выбор первых соединений с самой низкой стоимостью может привести к очень дорогому окончательному порядку, а лучший окончательный порядок соединений может иметь очень дорогое начало. Поэтому важно, чтобы оптимизатор не отвлекался на соединения, которые кажутся лучше, чем есть на самом деле. Листинг 15.7 и соответствующий ему рис. 15.12 иллюстрируют эту проблему.

Листинг 15.7. Запрос, возвращающий все оценки, выставленные профессором Эйнштейном, и названия всех курсов, которые он преподает

```
select Title, Grade
from ENROLL, SECTION, COURSE
where SectId=SectionId and CId=CourseId and Prof='einstein'
```

[1] В идеале метод должен создавать план соединения хешированием, но этот вид соединений не поддерживается в SimpleDB. См. упражнение 15.17.

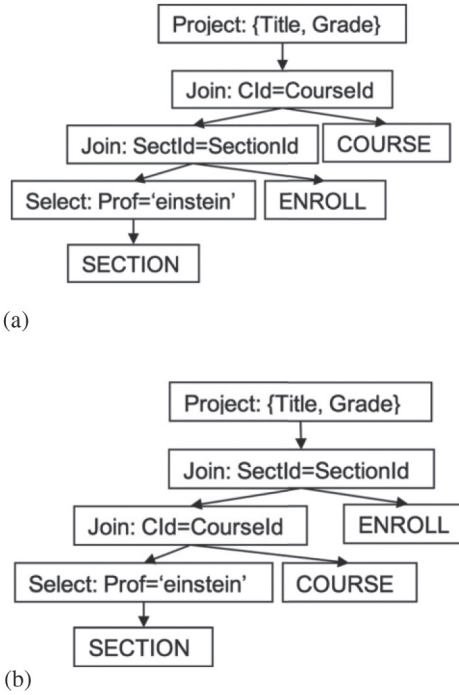


Рис. 15.12. Какую таблицу следует выбрать на роль второй в порядке соединения: (a) результат выбора таблицы ENROLL; (b) результат выбора таблицы COURSE?

Запрос в листинге 15.7 возвращает все оценки, выставленные профессором Эйнштейном, и названия всех курсов, которые он преподает. Возьмем за основу статистики из табл. 7.2 и предположим, что ENROLL имеет индекс для поля `SectionId`. Определяя порядок соединения, оптимизатор SimpleDB выберет таблицу SECTION первой, потому что она самая маленькая (а также имеет самый селективный предикат). Вопрос в том, какую таблицу выбрать следующей. Если использовать критерий, требующий минимизировать количество выходных записей, то тогда следует выбрать COURSE. А если требуется минимизировать количество обращений к блокам, то следует выбрать ENROLL, потому что соединение с использованием индексов выполняется более эффективно. Однако в данном случае ENROLL – неправильный выбор, потому что получение на выходе большого количества записей приведет к существенному удорожанию последующего соединения с COURSE.

Как показывает этот пример, большое количество записей ENROLL, соответствующих предикату соединения, оказывает существенное влияние на стоимость последующих соединений, поэтому ENROLL должна включаться в порядок соединения как можно позже. Минимизируя количество выходных записей, оптимизатор гарантирует, что ENROLL окажется в самом конце. Тот факт, что соединение с ENROLL реализуется быстро, лишь вводит в заблуждение и не имеет значения.

15.6.2. Оптимизатор Селинджер

Для выбора порядка соединения таблиц оптимизатор SimpleDB использует эвристики. В начале 1970-х исследователи из IBM написали оптимизатор для прототипа системы баз данных System-R. Этот оптимизатор оказал значительное влияние на современные реляционные системы; он выбирал порядок соединения с помощью метода динамического программирования. Эту стратегию оптимизации часто называют именем Патрисии Селинджер (Patricia Selinger), возглавлявшей команду разработки оптимизатора.

Алгоритм оптимизации Селинджер сочетает метод динамического программирования с генерированием планов. В частности, алгоритм вычисляет `lowest[S]` для каждого множества таблиц `S`, но сохраняет в `lowest[S]` не порядок соединения, а план с наименьшей стоимостью.

На первом шаге алгоритм вычисляет план с наименьшей стоимостью для каждой пары таблиц. Затем использует эти планы для вычисления плана с наименьшей стоимостью для каждой тройки таблиц и т. д., пока не будет вычислен общий план с самой низкой стоимостью.

В этом алгоритме планом с наименьшей стоимостью считается план с наименьшим количеством обращений к блокам, а не с наименьшим количеством выходных записей. То есть этот алгоритм – единственный в данной книге, который фактически учитывает обращения к блокам при выборе порядка соединения. Поэтому его оценки, вероятно, будут более точными, чем при использовании других алгоритмов.

Почему алгоритм оптимизации Селинджер может учитывать количество обращений к блокам? Причина в том, что, в отличие от оптимизации с использованием эвристик, алгоритм рассматривает все односторонние левые деревья и не отбрасывает частичный порядок соединения до тех пор, пока не будет уверен, что этот порядок бесполезен. Вернемся к примеру на рис. 15.12. Алгоритм Селинджер вычислит и сохранит все планы с наименьшей стоимостью для (SECTION, ENROLL) и (SECTION, COURSE), даже притом что план для (SECTION, ENROLL) имеет более низкую стоимость. Он учтет оба плана при вычислении самого недорогого плана для множества (ENROLL, SECTION, COURSE). Обнаружив, что соединение COURSE с (ENROLL, SECTION) обходится слишком дорого, он попробует альтернативный план.

Еще одно преимущество использования количества обращений к блокам для сравнения планов состоит в возможности более детального анализа затрат. Например, оптимизатор может учесть стоимость сортировки. Рассмотрим дерево запроса на рис. 15.13.

Предположим, что планировщик соединяет ENROLL и STUDENT, используя соединение хешированием. Перейдя к группировке, планировщику потребуется материализовать результат и отсортировать его по `StudentId`. В качестве альтернативы предположим, что планировщик использует соединение слиянием. В этом случае отпадает необходимость в предварительной обработке выходных данных, потому что они уже будут отсортированы по `StudentId`. Иначе говоря, вполне возможно, что использование соединения слиянием даст лучший окончательный план, даже притом что оно менее эффективно, чем соединение хешированием!

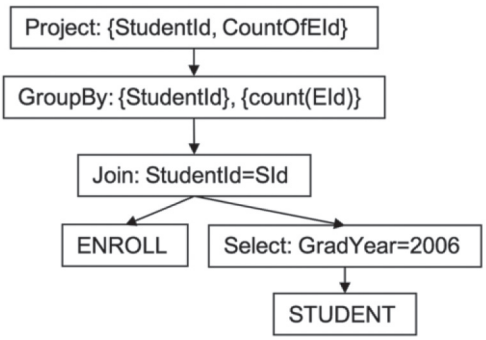


Рис. 15.13. Какой способ соединения ENROLL и STUDENT лучше?

Из этого примера следует, что планировщику также необходимо запоминать порядок сортировки, если перед ним стоит цель создать лучший план. Оптимизатор Селинджер может получить такую возможность, сохраняя план с наименьшей стоимостью для каждого порядка сортировки в `lowest[S]`. В примере выше значение `lowest[{ENROLL, STUDENT}]` будет хранить и планы соединения слиянием, и планы соединения хешированием, потому что каждый из них имеет свой порядок сортировки.

15.7. Объединение блоков запроса

В этом разделе исследуется оптимизация запросов, использующих представления. Рассмотрим, например, запрос в листинге 15.8, который использует представление для получения имен студентов, получивших оценку «А» по курсу профессора Эйнштейна. Базовый планировщик, представленный в главе 10, создаст план для такого запроса, запланировав представление и запрос по отдельности, а затем вставит план представления в план запроса. Полученный в результате план показан на рис. 15.14.

Листинг 15.8. Определение представления и запроса, который его использует

```
create view EINSTEIN as
select SectId from SECTION
where Prof = 'einstein'

select SName from STUDENT, ENROLL, EINSTEIN
where SId = StudentId and SectionId = SectId
      and Grade = 'A'
```

Планы, соответствующие запросу и представлению, называются *блоками запроса*. План на рис. 15.14 иллюстрирует простейший способ оптимизации запросов с представлениями – планировщик оптимизирует каждый блок запроса отдельно, а потом объединяет их в окончательный план. Реализовать оптимизацию каждого блока в отдельности просто, но окончательные планы не обязательно будут получаться самыми оптимальными. План на рис. 15.14 является тому примером. Наилучший порядок соединения – (SECTION, ENROLL, STUDENT), но он невозможен с учетом деления запроса на блоки.

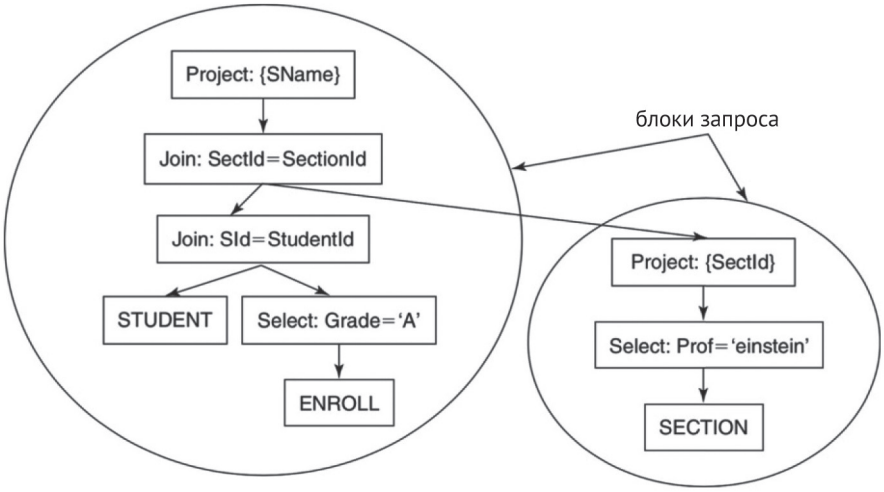


Рис. 15.14 . Планирование запроса с представлением. Каждый блок запроса планируется отдельно

Чтобы решить эту проблему, можно *объединить* блоки в один запрос и вычислить план для него. Например, планировщик может игнорировать проекцию в блоке определения представления на рис. 15.14 и добавить селекцию и таблицу в основной запрос. Однако такая стратегия возможна, только если представление имеет достаточно простое определение. Ситуация значительно усложняется, если представление включает группировку или удаление дубликатов; в подобном случае объединение может оказаться невозможным.

15.8. Итоги

- Два запроса *эквивалентны*, если их выходные таблицы содержат те же записи (пусть и в разном порядке), независимо от содержимого базы данных.
- SQL-запрос может иметь много эквивалентных деревьев запроса. Их эквивалентность вытекает из свойств операторов реляционной алгебры.
  - Оператор прямого произведения обладает свойствами коммутативности и ассоциативности. Эти свойства подразумевают, что узлы product в дереве запроса могут вычисляться в любом порядке.
  - Узел селекции для предиката p можно расщепить на несколько узлов селекции, по одному для каждого конъюнктивного члена в p. Преобразование p в конъюнктивную нормальную форму (КНФ) позволяет разделить предикат на наименьшие составляющие. Узлы для каждого конъюнктивного члена можно разместить в дереве запроса где угодно, при условии что предикат селекции сохранит смысл.
  - Пару узлов select-product можно заменить одним узлом соединения join.
  - Узел проекции project можно добавить выше любого узла в дереве запроса, при условии что все поля в его проекции упоминаются в узлах-предках.

- Планы для двух эквивалентных деревьев могут радикально отличаться временем выполнения. Поэтому планировщик должен попытаться найти план, требующий наименьшего количества обращений к блокам. Этот процесс называется *оптимизацией запроса*.
- Оптимизация запроса – сложная задача, потому что SQL-запрос может иметь гораздо больше планов, чем планировщик сможет перебрать. Справиться с этой сложностью можно, выполнив оптимизацию в два независимых этапа:
  - этап 1: найти *наиболее перспективное* дерево запроса, то есть такое дерево, которое вероятнее всего даст наиболее эффективный план;
  - этап 2: выбрать лучший план для этого дерева.
- На этапе 1 планировщик не имеет возможности оценить количество обращений к блокам, потому что не знает, какие планы используются. Поэтому стоимость дерева запроса определяется как сумма размеров входных данных для каждого узла product и join в дереве. Интуитивно понятно, что дерево запросов с наименьшей стоимостью минимизирует размер промежуточных соединений. Идея состоит в том, что каждое соединение формирует входные данные для последующего соединения, поэтому чем больше размер промежуточных выходных данных, тем выше будет стоимость выполнения запроса.
- Планировщик также использует эвристики, чтобы ограничить множество рассматриваемых деревьев и планов:
  - помещать узлы select как можно глубже в дереве запроса;
  - заменить каждую пару узлов select-product узлом join;
  - над входными данными каждого материализующего плана поместить узел project;
  - учитывать только односторонние левые деревья;
  - если возможно, избегать операций прямого произведения, которые не являются соединениями.
- Каждое одностороннее левое дерево имеет оптимальный *порядок соединения*. Поиск такого оптимального порядка соединения – самая сложная часть оптимизации запроса.
- Один из способов определить порядок соединения – использовать эвристики. Вот две разумные (но противоречивые) эвристики:
  - выбрать таблицу, производящую наименьший объем выходных данных;
  - выбрать таблицу с наиболее ограничивающим предикатом.

Смысл второй эвристики – создать дерево запроса, в котором самые ограничивающие узлы select находятся максимально глубоко. Как подсказывает интуиция, такие деревья часто имеют наименьшую стоимость.

- Другой способ выбрать порядок соединения – перебрать все возможные порядки соединений с использованием метода динамического программирования. Алгоритм динамического программирования вычисляет порядок соединения с наименьшей стоимостью для каждого множества таблиц, начиная с множеств из двух таблиц, затем из трех таблиц и т. д., пока не будет получено множество, включающее все таблицы.

- На втором этапе оптимизации планировщик составляет план, выбирая реализацию для каждого узла в дереве запроса. Выбор реализации для каждого узла производится независимо от реализаций других узлов. Стоимость реализаций определяется количеством обращений к блокам. Планировщик может определить план с наименьшей стоимостью, либо исследовав все возможные реализации для каждого узла, либо применив следующие эвристики:
  - всегда использовать индексы, если возможно;
  - если соединение с использованием индексов невозможно, то применить соединение хешированием, если одна из входных таблиц значительно меньше другой, иначе применить соединение слиянием.
- Реализация оптимизатора запросов может объединить два этапа и создавать план вместе с деревом запроса. Оптимизатор в SimpleDB использует эвристики для определения порядка соединения и постепенно строит план по мере выбора каждой таблицы. *Оптимизатор Селинджер* использует метод динамического программирования – для каждого множества таблиц он сохраняет не порядок соединения, а план с наименьшей стоимостью. Преимущество оптимизатора Селинджер перед любыми другими методами состоит в том, что для расчета наилучшего порядка соединения он может использовать оценки количества обращений к блокам.
- Запрос, использующий представление, будет иметь план, состоящий из нескольких *блоков запроса*. Самый простой способ обработать несколько блоков запроса – оптимизировать каждый из них отдельно, а затем объединить их. Однако если блоки запроса оптимизировать вместе, можно получить более эффективный план. Такая стратегия возможна, только если представление имеет достаточно простое определение.

15.9. Для дополнительного чтения

В этой главе было представлено упрощенное введение в оптимизацию запросов. Гораздо более подробные сведения можно найти в статьях Graefe (1993) и Chaudhuri (1998). В статье Swami (1989) приводится сравнение различных эвристик порядка соединения на основе экспериментальных данных. Описание оптимизатора в System-R можно найти в Selinger et al. (1979).

Один из недостатков традиционных планировщиков запросов заключается в том, что эвристики и стратегия оптимизации жестко «зашиты» в их реализации. Поэтому изменить эвристики или добавить новые реляционные операторы можно, только переписав код. Альтернативный подход состоит в том, чтобы выразить операторы и соответствующие им преобразования в виде *правил вывода* и заставить планировщик использовать эти правила для преобразования исходного запроса в оптимальный. Чтобы изменить такой планировщик, достаточно изменить набор правил. Описание этой стратегии можно найти в Pirahesh (1992).

Стратегии оптимизации, описанные в данной главе, четко различают этапы планирования и выполнения запросов – после того как план будет открыт и выполнен, пути назад уже не будет. Если планировщик по ошибке выберет неэффективный план, изменить ничего не получится. В статье Kabra & DeWitt

(1998) рассказывается, как система баз данных может отслеживать выполнение плана и собирать статистику о его поведении. Если она посчитает, что план выполняется менее эффективно, чем должно быть, то сможет использовать статистику для создания лучшего плана и «горячей замены» старого плана на новый.

Chaudhuri, S. (1998). «An overview of query optimization in relational systems». *Proceedings of the ACM Principles of Database Systems Conference*. P. 34–43.

Graefe, G. (1993). «Query evaluation techniques for large databases». *ACM Computing Surveys*, 25 (2). P. 73–170.

Kabra, N., & DeWitt, D. (1998). «Efficient mid-query re-optimization of sub-optimal query execution plans». *Proceedings of the ACM SIGMOD Conference*. P. 106–117.

Pirahesh, H., Hellerstein, J., & Hasan, W. (1992). «Extendable/rule based query rewrite in starburst». *Proceedings of the ACM SIGMOD Conference*. P. 39–48.

Selinger, P., Astrahan, M., Chamberlin, D., Lorie, R., & Price, T. (1979). «Access-path selection in a relational database management system». *Proceedings of the ACM SIGMOD Conference*. P. 23–34.

Swami, A. (1989) «Optimization of large join queries: Combining heuristics and combinatorial techniques». *ACM SIGMOD Record*, 18 (2), 367–376.

15.10. Упражнения

Теория

15.1. Покажите, что оператор прямого произведения обладает свойством ассоциативности.

15.2. Представьте запрос, который выполняет прямое произведение нескольких таблиц и два любых дерева запроса, эквивалентных этому запросу. Покажите, что с помощью выражений эквивалентности из раздела 15.1.1 можно преобразовать одно дерево в другое.

15.3. Исследуйте дерево запроса на рис. 15.2а.

а) Приведите последовательность преобразований, которая создаст дерево на рис. 15.15.

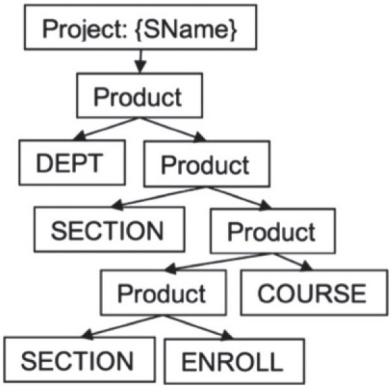


Рис. 15.15. Дерево, которое требуется получить

b) Приведите последовательность преобразований, которая создаст одностороннее левое дерево с порядком соединения (COURSE, SECTION, ENROLL, STUDENT, DEPT).

15.4. Представьте дерево запроса с узлами select.

a) Покажите, что перемещение узлов select относительно друг друга дает в результате эквивалентное дерево запроса.
b) Когда узел select можно поместить выше узла project?
c) Покажите, что при необходимости узел select можно поместить выше или ниже узла группировки groupby.

15.5. Исследуйте оператор *объединения union* из упражнения 8.16.

a) Покажите, что этот оператор обладает свойствами ассоциативности и коммутативности, и опишите преобразования для этих эквивалентностей.
b) Опишите преобразование, которое позволяет поместить оператор селекции в объединение.

15.6. Исследуйте операторы *антисоединения antijoin* и *полусоединения semijoin* из упражнения 8.17.

a) Обладают ли эти операторы свойством ассоциативности? Обладают ли они свойством коммутативности? Приведите подходящие преобразования.
b) Опишите преобразования, которые позволяют поместить оператор селекции в антисоединение или полусоединение.

15.7. Подумайте, можно ли добавить предикат селекции, показанный на рис. 15.6a, в дерево запросов на рис. 15.2b. Нарисуйте дерево запроса, которое является результатом перемещения узлов select на максимально возможную глубину.

15.8. Нарисуйте два эквивалентных дерева запроса таких, что план с наименьшей стоимостью получается из дерева с наибольшей стоимостью.

15.9. Покажите, что кустистое дерево на рис. 15.10d является наиболее многообещающим деревом для его SQL-запроса.

15.10. Дерево запросов на рис. 15.6b имеет порядок соединения (STUDENT, ENROLL, SECTION, COURSE, DEPT). Существует еще 15 вариантов порядка соединения, которые не требуют операции прямого произведения. Перечислите их.

15.11. Напишите такой запрос, чтобы эвристика 4 не давала дерева запросов с наименьшей стоимостью.

15.12. Рассмотрите рис. 15.6.

a) Вычислите стоимость обоих деревьев.
b) Вычислите наиболее перспективное дерево запроса с помощью эвристик, сначала 5a, а затем 5b.
c) Вычислите наиболее перспективное дерево запроса, используя алгоритм динамического программирования.
d) Вычислите план с наименьшей стоимостью для самого перспективного дерева запроса.

15.13. Взгляните на следующий запрос:

```
select Grade
from ENROLL, STUDENT, SECTION
where SId=StudentId and SectId=SectionId and SId=1 and SectId=53
```

a) Покажите, что порядок соединения (ENROLL, STUDENT, SECTION) имеет меньшую стоимость, чем (ENROLL, SECTION, STUDENT).
b) Рассчитайте наиболее перспективное дерево запроса с помощью эвристик, сначала 5a, а затем 5b.
c) Вычислите наиболее перспективное дерево запроса, используя алгоритм динамического программирования.
d) Вычислите план с наименьшей стоимостью для самого перспективного дерева запроса.

15.14. Алгоритм динамического программирования, представленный в разделе 15.4, рассматривает только односторонние левые деревья. Дополните его, чтобы он рассматривал все возможные деревья соединений.

Практика

15.15. Измените эвристический планировщик в SimpleDB, чтобы для выбора порядка соединения таблиц он использовал эвристику 5b.

15.16. Реализуйте для SimpleDB планировщик Селинджер.

15.17. В упражнении 14.15 предлагалось реализовать в SimpleDB алгоритм соединения хешированием. Теперь измените класс `TablePlanner` так, чтобы по возможности вместо многобуферного прямого произведения он использовал соединение хешированием.

Предметный указатель