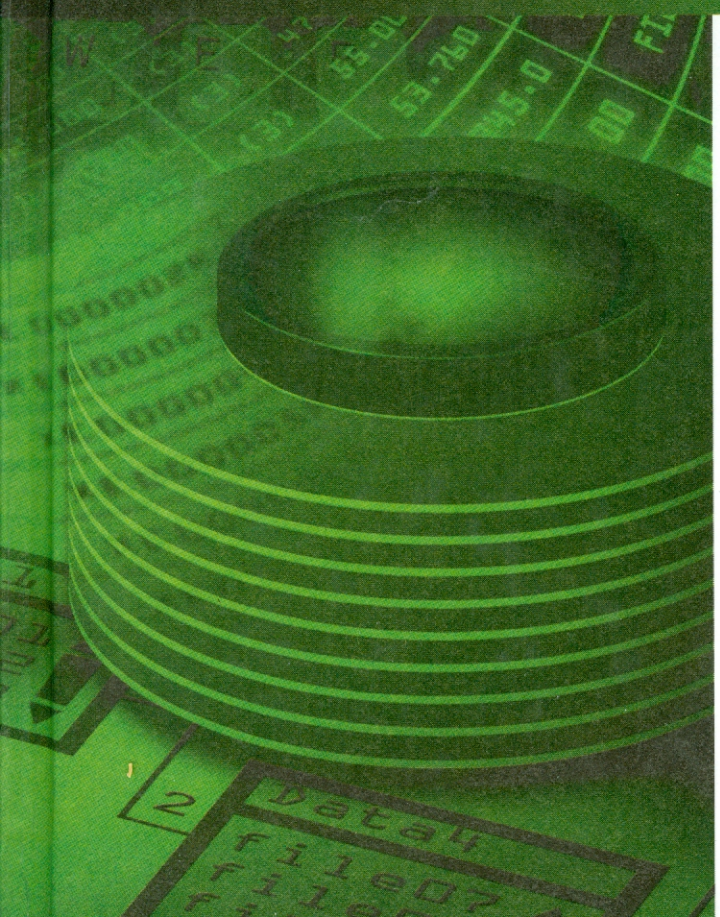
Т. Карпова

ПИТЕР®

БАЗЫ ДАННЫХ

МОДЕЛИ, РАЗРАБОТКА, РЕАЛИЗАЦИЯ

УЧЕБНИК



- студентам вузов технических специальностей
- автор книги — преподаватель с многолетним стажем
- теоретический материал в сочетании с примерами

Т. Карпова

БАЗЫ ДАННЫХ

МОДЕЛИ, РАЗРАБОТКА, РЕАЛИЗАЦИЯ

Санкт-Петербург
Москва • Харьков • Минск
2001

ПИТЕР®

ББК 32.973.233-018я7
УДК 681.3.016 (075)
К26

К26 **Базы данных: модели, разработка, реализация** / Т. С. Карпова. — СПб.: Питер, 2001. — 304 с.: ил.

ISBN 5–272–00278–4

Настоящее учебное пособие подготовлено по материалам лекционных курсов, посвященных основам теории баз данных, языку SQL и серверам баз данных, которые читались автором в течение последних десяти лет в Государственном Санкт-Петербургском университете аэрокосмического приборостроения и в Государственном техническом университете (Политехническом институте).

Учебное пособие полностью соответствует требованиям стандарта по дисциплине «Базы данных» для всех вычислительных специальностей, а также для бакалавров по направлению 5528 «Информатика и вычислительная техника».

ББК 32.973.233-018я7
УДК 681.3.016 (075)

ISBN 5–272–00278–4

Краткое содержание

Содержание

От автора

Теория баз данных — сравнительно молодая область знаний. Возраст ее составляет немногим более 30 лет. Однако изменился ритм времени, оно уже не бежит, а летит, и мы вынуждены подчиняться ему во всем. Поэтому столь молодая область знаний является практически обязательной для изучения студентами всех технических специальностей. В соответствии с новыми стандартами учебная дисциплина «Базы данных» включена в стандарты всех специальностей, связанных с подготовкой специалистов по вычислительной технике: это группа специальностей 22.01, 22.02, 22.03 и 22.04. В остальные технические специальности раздел, посвященный базам данных, включен в общий курс информатики и вычислительной техники.

И действительно, современный мир информационных технологий трудно представить себе без использования баз данных. Практически все системы в той или иной степени связаны с функциями долговременного хранения и обработки информации. Фактически информация становится фактором, определяющим эффективность любой сферы деятельности. Увеличились информационные потоки и повысились требования к скорости обработки данных, и теперь уже большинство операций не может быть выполнено вручную, они требуют применения наиболее перспективных компьютерных технологий. Любые административные решения требуют четкой и точной оценки текущей ситуации и возможных перспектив ее изменения. И если раньше в оценке ситуации участвовало несколько десятков факторов, которые могли быть вычислены вручную, то теперь таких факторов сотни и сотни тысяч, и ситуация меняется не в течение года, а через несколько минут, а обоснованность принимаемых решений требуется большая, потому что и реакция на неправильные решения более серьезная, более быстрая и более мощная, чем раньше. И, конечно, обойтись без информационной модели производства, хранимой в базе данных, в этом случае невозможно.

Настоящее учебное пособие подготовлено по материалам лекционных курсов, посвященных основам теории баз данных, языку SQL и серверам баз данных, которые читались мною в течение 10 последних лет в Государственном Санкт-Петербургском университете аэрокосмического приборостроения студентам дневной формы обучения и слушателям курсов повышения квалификации, а также слушателям второго высшего образования в данном университете и в Государственном техническом университете (Политехническом институте). В него включены разделы по распределенной обработке данных с использованием технологии «клиент—сервер», которые могут быть использованы для дальнейшего освоения современной теории и практики работы с базами данных.

Учебное пособие полностью соответствует требованиям стандарта по дисциплине «Базы данных» для всех вычислительных специальностей, а также для бакалавров по направлению 5528 «Информатика и вычислительная техника».

Пособие состоит из 14 глав. Оно может быть использовано для самостоятельного освоения курса «Базы данных» и подготовке к сдаче экзамена или как дополнение к лекционным курсам, читаемым в высших учебных заведениях.

Первая глава пособия посвящена истории возникновения области знаний, связанной с базами данных, в данной главе выделены основные этапы развития теории и практики баз данных, даются сравнительные характеристики этих этапов.

Вторая глава знакомит нас непосредственно с базами данных, здесь даются понятия и определения, являющиеся ключевыми для данной области знаний. Здесь рассматривается классическая трехуровневая архитектура, используемая в системах баз данных, упрощенный процесс прохождения запроса в базах данных и, наконец, приводится классификация моделей, используемых в системах баз данных.

Глава 3 посвящена первым теоретико-графовым моделям, которые использовались в ранних системах управления базами данных.

В главе 4 начинается обсуждение современной реляционной модели, которая является основой практически для всех коммерческих систем управления базами данных (СУБД) и наиболее распространена в настоящий момент. В этой же главе дается описание первого языка манипулирования данными, предложенного для данной модели ее создателем американским математиком Е.Ф. Коддом — реляционной алгебры.

Пятая глава полностью посвящена современному стандартному языку работы с базами данных, языку SQL. Язык SQL является сейчас стандартным базовым языком по работе с базами данных, любое собеседование, при приглашении вас на работу, связанную с информационными технологиями, включает решение нескольких запросов на языке SQL. Мне кажется, что пятая глава данного пособия может помочь научиться писать правильные запросы на языке SQL самостоятельно. Мои студенты учились на примерах, которые приведены в данной главе и те из них, кто успешно решал приведенные примеры, также успешно сдали собеседования в ведущих зарубежных и отечественных фирмах, связанных с разработкой информационных систем. Я желаю и вам, мои читатели, таких же успехов.

Глава 6 посвящена вопросам проектирования баз данных, в ней рассматриваются базовые понятия функциональных и многозначных зависимостей между свойствами объектов, которые моделируются в базе данных. База данных — это фундамент Вашей будущей информационной системы и от того, как он будет построен, зависит во многом успех всей информационной системы, которая будет строиться на данном фундаменте. От корректности и продуманности структуры спроектированной базы данных зависит успех решения не только текущих информационных задач, но и перспективы развития и наращивания системы.

Глава 7 посвящена семантическим или инфологическим моделям, используемым в современных программных системах поддержки проектирования, называемых CASE-системами (Computer Aided Software Engineering).

Глава 8 посвящена принципам поддержки целостности в базах данных. Понятие целостности — одно из базовых понятий в современных базах данных. Принципы целостности составляют набор некоторых правил, которые выполняются автоматически при работе с данной базой данных. При грамотном составлении этих правил мы избавляем операторов, которые работают с базой данных, а также разработчиков приложений от дополнительного контроля за правильностью и взаимосвязанностью вводимой в базу данных информации. Эти функции теперь выполняет сама СУБД, она контролирует вводимые и удаляемые данные, она не допускает ввода некорректной информации.

Глава 9 посвящена физическим моделям баз данных. В этом разделе описываются основные файловые конструкции, применяемые при создании баз данных.

Глава 10 посвящена вопросам распределенной обработки данных, здесь рассматриваются модели клиент-сервер, применяемые в системах баз данных.

Глава 11 посвящена понятию транзакции, которое является базовым при выполнении параллельных запросов к базам данных. Рассматриваются две базовые модели транзакций: модель ANSI и расширенная модель транзакций, подробно рассматриваются проблемы, выполняемые при параллельном выполнении транзакций.

В главе 12 рассматриваются дополнительные возможности языка SQL , которые используются при разработке хранимых процедур и приложений, работающих с базами данных, а также принципы трансляции операторов языка SQL, порядок трансляции и выполнения SQL-запросов.

Глава 13 посвящена вопросам защиты информации в базах данных. Понятие защиты информации в базах данных чаще всего связано с концепцией защиты от несанкционированного доступа. В данной главе обсуждается общая концепция защиты информации, которая применяется в базах данных, вводится понятие пользователя и рассматриваются вопросы определения прав и привилегий пользователей по работе с отдельными объектами в базе данных.

Глава 14 посвящена рассмотрению обобщенной архитектуры современных баз данных, здесь рассматривается структура системных каталогов, основные функциональные блоки в современных серверах баз данных и их назначение. В этой главе обобщаются все сведения о базах данных, рассмотренные в предыдущих разделах.

В Заключении кратко характеризуются перспективы развития современных систем баз данных, дается перечень тех тем и вопросов, которые рекомендуется рассмотреть для дальнейшего более глубокого знакомства с теорией и практикой баз данных.

В заключение мне бы хотелось поблагодарить тех, кто помог мне написать данную книгу. Прежде всего, это мои многочисленные студенты, которые решали многие задачи, задавали множество вопросов и постоянно будоражили меня, заставляя придумывать новые рисунки и способы объяснения запутанных понятий, упрощать наиболее сложные моменты изложения, делая их доступными и ясными. Однако переход от многолетнего лекционного материала к написанию на его основе учебного пособия оказался весьма непростым для меня, и здесь я благодарна сотрудникам редакции компьютерной литературы издательства «Питер», особенно Екатерине Вячеславовне Строгановой и Илье Александровичу Корнееву. Только их терпение, доброжелательность и настойчивость позволили мне завершить мой труд. И, конечно, я благодарна моим домашним: маме и дочке, которые весь период написания книги терпели мое плохое настроение, недовольство собой и текстом и полное устранение от домашних дел.

ГЛАВА 1 Введение

История развития баз данных

В истории вычислительной техники можно проследить развитие двух основных областей ее использования. Первая область — применение вычислительной техники для выполнения численных расчетов, которые слишком долго или вообще невозможно производить вручную. Развитие этой области способствовало интенсификации методов численного решения сложных математических задач, появлению языков программирования, ориентированных на удобную запись численных алгоритмов, становлению обратной связи с разработчиками новых архитектур ЭВМ. Характерной особенностью данной области применения вычислительной техники является наличие сложных алгоритмов обработки, которые применяются к простым по структуре данным, объем которых сравнительно невелик.

Вторая область, которая непосредственно относится к нашей теме, — это использование средств вычислительной техники в автоматических или автоматизированных информационных системах. Информационная система представляет собой программно-аппаратный комплекс, обеспечивающий выполнение следующих функций:

- надежное хранение информации в памяти компьютера;
- выполнение специфических для данного приложения преобразований информации и вычислений;
- предоставление пользователям удобного и легко осваиваемого интерфейса.

Обычно такие системы имеют дело с большими объемами информации, имеющей достаточно сложную структуру. Классическими примерами информационных систем являются банковские системы, автоматизированные системы управления предприятиями, системы резервирования авиационных или железнодорожных билетов, мест в гостиницах и т. д.

Вторая область использования вычислительной техники возникла несколько позже первой. Это связано с тем, что на заре вычислительной техники возможности компьютеров по хранению информации были очень ограниченными. Говорить о надежном и долговременном хранении информации можно только при наличии запоминающих устройств, сохраняющих информацию после выключения электрического питания. Оперативная (основная) память компьютеров этим свойством обычно не обладает. В первых компьютерах использовались два вида устройств внешней памяти — магнитные ленты и барабаны. Емкость магнитных лент была достаточно велика, но по своей физической природе они обеспечивали последовательный доступ к данным. Магнитные же барабаны (они ближе всего к современным магнитным дискам с фиксированными головками) давали возможность произвольного доступа к данным, но имели ограниченный объем хранимой информации.

Эти ограничения не являлись слишком существенными для чисто численных расчетов. Даже если программа должна обработать (или произвести) большой объем информации, при программировании можно продумать расположение этой информации во внешней памяти (например, на последовательной магнитной ленте), обеспечивающее эффективное выполнение этой программы. Однако в информационных системах совокупность взаимосвязанных информационных объектов фактически отражает модель объектов реального мира. А потребность пользователей в информации, адекватно отражающей состояние реальных объектов, требует сравнительно быстрой реакции системы на их запросы. И в этом случае наличие сравнительно медленных устройств хранения данных, к которым относятся магнитные ленты и барабаны, было недостаточным.

Можно предположить, что именно требования нечисловых приложений вызвали появление съемных магнитных дисков с подвижными головками, что явилось революцией в истории вычислительной техники. Эти устройства внешней памяти обладали существенно большей емкостью, чем магнитные барабаны, обеспечивали удовлетворительную скорость доступа к данным в режиме произвольной выборки, а возможность смены дискового пакета на устройстве позволяла иметь практически неограниченный архив данных.

С появлением магнитных дисков началась история систем управления данными во внешней памяти. До этого каждая прикладная программа, которой требовалось хранить данные во внешней памяти, сама определяла расположение каждой порции данных на магнитной ленте или барабане и выполняла обмены между оперативной памятью и устройствами внешней памяти с помощью программно-аппаратных средств низкого уровня (машинных команд или вызовов соответствующих программ операционной системы). Такой режим работы не позволяет или очень затрудняет поддержание на одном внешнем носителе нескольких архивов долговременно хранимой информации. Кроме того, каждой прикладной программе приходилось решать проблемы именования частей данных и структуризации данных во внешней памяти.

Файлы и файловые системы

Важным шагом в развитии именно информационных систем явился переход к использованию централизованных систем управления файлами. С точки зрения прикладной программы, файл — это именованная область внешней памяти, в которую можно записывать и из которой можно считывать данные. Правила именования файлов, способ доступа к данным, хранящимся в файле, и структура этих данных зависят от конкретной системы управления файлами и, возмож-

но, от типа файла. Система управления файлами берет на себя распределение внешней памяти, отображение имен файлов в соответствующие адреса во внешней памяти и обеспечение доступа к данным.

Конкретные модели файлов, используемые в системе управления файлами, мы рассмотрим далее, когда перейдем к физическим способам организации баз данных, а на этом этапе нам достаточно знать, что пользователи видят файл как линейную последовательность записей и могут выполнить над ним ряд стандартных операций:

- создать файл (требуемого типа и размера);
- открыть ранее созданный файл;
- прочитать из файла некоторую запись (текущую, следующую, предыдущую, первую, последнюю);
- записать в файл на место текущей записи новую, добавить новую запись в конец файла.

В разных файловых системах эти операции могли несколько отличаться, но общий смысл их был именно таким. Главное, что следует отметить, это то, что структура записи файла была известна только программе, которая с ним работала, система управления файлами не знала ее. И поэтому для того, чтобы извлечь некоторую информацию из файла, необходимо было точно знать структуру записи файла с точностью до бита. Каждая программа, работающая с файлом, должна была иметь у себя внутри структуру данных, соответствующую структуре этого файла. Поэтому при изменении структуры файла требовалось изменять структуру программы, а это требовало новой компиляции, то есть процесса перевода программы в исполняемые машинные коды. Такая ситуации характеризовалась как зависимость программ от данных. Для информационных систем характерным является наличие большого числа различных пользователей (программ), каждый из которых имеет свои специфические алгоритмы обработки информации, хранящейся в одних и тех же файлах. Изменение структуры файла, которое было необходимо для одной программы, требовало исправления и перекомпиляции и дополнительной отладки всех остальных программ, работающих с этим же файлом. Это было первым существенным недостатком файловых систем, который явился толчком к созданию новых систем хранения и управления информацией.

Далее, поскольку файловые системы являются общим хранилищем файлов, принадлежащих, вообще говоря, разным пользователям, системы управления файлами должны обеспечивать авторизацию доступа к файлам. В общем виде подход состоит в том, что по отношению к каждому зарегистрированному пользователю данной вычислительной системы для каждого существующего файла указываются действия, которые разрешены или запрещены данному пользователю. В большинстве современных систем управления файлами применяется подход к защите файлов, впервые реализованный в ОС UNIX. В этой ОС каждому зарегистрированному пользователю соответствует пара целочисленных идентификаторов: идентификатор группы, к которой относится этот пользователь, и его собственный идентификатор в группе. При каждом файле хранится полный идентификатор пользователя, который создал этот файл, и фиксируется, какие

действия с файлом может производить его создатель, какие действия с файлом доступны для других пользователей той же группы и что могут делать с файлом пользователи других групп. Администрирование режимом доступа к файлу в основном выполняется его создателем-владельцем. Для множества файлов, отражающих информационную модель одной предметной области, такой децентрализованный принцип управления доступом вызывал дополнительные трудности. И отсутствие централизованных методов управления доступом к информации послужило еще одной причиной разработки СУБД.

Следующей причиной стала необходимость обеспечения эффективной параллельной работы многих пользователей с одними и теми же файлами. В общем случае системы управления файлами обеспечивали режим многопользовательского доступа. Если операционная система поддерживает многопользовательский режим, вполне реальна ситуация, когда два или более пользователя одновременно пытаются работать с одним и тем же файлом. Если все пользователи собираются только читать файл, ничего страшного не произойдет. Но если хотя бы один из них будет изменять файл, для корректной работы этих пользователей требуется взаимная синхронизация их действий по отношению к файлу.

В системах управления файлами обычно применялся следующий подход. В операции открытия файла (первой и обязательной операции, с которой должен начинаться сеанс работы с файлом) среди прочих параметров указывался режим работы (чтение или изменение). Если к моменту выполнения этой операции некоторым пользовательским процессом PR1 файл был уже открыт другим процессом PR2 в режиме изменения, то в зависимости от особенностей системы процессу PR1 либо сообщалось о невозможности открытия файла, либо он блокировался до тех пор, пока в процессе PR2 не выполнялась операция закрытия файла.

При подобном способе организации одновременная работа нескольких пользователей, связанная с модификацией данных в файле, либо вообще не реализовывалась, либо была очень замедлена.

Эти недостатки послужили тем толчком, который заставил разработчиков информационных систем предложить новый подход к управлению информацией. Этот подход был реализован в рамках новых программных систем, названных впоследствии Системами Управления Базами Данных (СУБД), а сами хранилища информации, которые работали под управлением данных систем, назывались базами или банками данных (БД и БнД).

Первый этап — базы данных на больших ЭВМ

История развития СУБД насчитывает более 30 лет. В 1968 году была введена в эксплуатацию первая промышленная СУБД система IMS фирмы IBM. В 1975 году появился первый стандарт ассоциации по языкам систем обработки данных — Conference of Data System Languages (CODASYL), который определил ряд фундаментальных понятий в теории систем баз данных, которые и до сих пор являются основополагающими для сетевой модели данных.

В дальнейшее развитие теории баз данных большой вклад был сделан американским математиком Э. Ф. Коддом, который является создателем реляционной

модели данных. В 1981 году Э. Ф. Кодд получил за создание реляционной модели и реляционной алгебры престижную премию Тьюринга Американской ассоциации по вычислительной технике.

Менее двух десятков лет прошло с этого момента, но стремительное развитие вычислительной техники, изменение ее принципиальной роли в жизни общества, обрушившийся бум персональных ЭВМ и, наконец, появление мощных рабочих станций и сетей ЭВМ повлияло также и на развитие технологии баз данных. Можно выделить четыре этапа в развитии данного направления в обработке данных. Однако необходимо заметить, что все же нет жестких временных ограничений в этих этапах: они плавно переходят один в другой и даже сосуществуют параллельно, но тем не менее выделение этих этапов позволит более четко охарактеризовать отдельные стадии развития технологии баз данных, подчеркнуть особенности, специфичные для конкретного этапа.

Первый этап развития СУБД связан с организацией баз данных на больших машинах типа IBM 360/370, ЕС-ЭВМ и мини-ЭВМ типа PDP11 (фирмы Digital Equipment Corporation – DEC), разных моделях HP (фирмы Hewlett Packard).

Базы данных хранились во внешней памяти центральной ЭВМ, пользователями этих баз данных были задачи, запускаемые в основном в пакетном режиме. Интерактивный режим доступа обеспечивался с помощью консольных терминалов, которые не обладали собственными вычислительными ресурсами (процессором, внешней памятью) и служили только устройствами ввода-вывода для центральной ЭВМ. Программы доступа к БД писались на различных языках и запускались как обычные числовые программы. Мощные операционные системы обеспечивали возможность условно параллельного выполнения всего множества задач. Эти системы можно было отнести к системам распределенного доступа, потому что база данных была централизованной, хранилась на устройствах внешней памяти одной центральной ЭВМ, а доступ к ней поддерживался от многих пользователей-задач.

Особенности этого этапа развития выражаются в следующем:

- Все СУБД базируются на мощных мультипрограммных операционных системах (MVS, SVM, RTE, OSRV, RSX, UNIX), поэтому в основном поддерживается работа с централизованной базой данных в режиме распределенного доступа.
- Функции управления распределением ресурсов в основном осуществляются операционной системой (ОС).
- Поддерживаются языки низкого уровня манипулирования данными, ориентированные на навигационные методы доступа к данным.
- Значительная роль отводится администрированию данных.
- Проводятся серьезные работы по обоснованию и формализации реляционной модели данных, и была создана первая система (System R), реализующая идеологию реляционной модели данных.
- Проводятся теоретические работы по оптимизации запросов и управлению распределенным доступом к централизованной БД, было введено понятие транзакции.

- Результаты научных исследований открыто обсуждаются в печати, идет мощный поток общедоступных публикаций, касающихся всех аспектов теории и практики баз данных, и результаты теоретических исследований активно внедряются в коммерческие СУБД.

Появляются первые языки высокого уровня для работы с реляционной моделью данных. Однако отсутствуют стандарты для этих первых языков.

Эпоха персональных компьютеров

Персональные компьютеры стремительно ворвались в нашу жизнь и буквально перевернули наше представление о месте и роли вычислительной техники в жизни общества. Теперь компьютеры стали ближе и доступнее каждому пользователю. Исчез благоговейный страх рядовых пользователей перед непонятными и сложными языками программирования. Появилось множество программ, предназначенных для работы неподготовленных пользователей. Эти программы были просты в использовании и интуитивно понятны: это прежде всего различные редакторы текстов, электронные таблицы и другие. Простыми и понятными стали операции копирования файлов и перенос информации с одного компьютера на другой, распечатка текстов, таблиц и других документов. Системные программисты были отодвинуты на второй план. Каждый пользователь мог себя почувствовать полным хозяином этого мощного и удобного устройства, позволяющего автоматизировать многие аспекты деятельности. И, конечно, это сказалось и на работе с базами данных. Появились программы, которые назывались системами управления базами данных и позволяли хранить значительные объемы информации, они имели удобный интерфейс для заполнения данных, встроенные средства для генерации различных отчетов. Эти программы позволяли автоматизировать многие учетные функции, которые раньше велись вручную. Постоянное снижение цен на персональные компьютеры сделало их доступными не только для организаций и фирм, но и для отдельных пользователей. Компьютеры стали инструментом для ведения документации и собственных учетных функций. Это все сыграло как положительную, так и отрицательную роль в области развития баз данных. Кажущаяся простота и доступность персональных компьютеров и их программного обеспечения породила множество дилетантов. Эти разработчики, считая себя знатоками, стали проектировать недолговечные базы данных, которые не учитывали многих особенностей объектов реального мира. Много было создано систем-однодневок, которые не отвечали законам развития и взаимосвязи реальных объектов. Однако доступность персональных компьютеров заставила пользователей из многих областей знаний, которые ранее не применяли вычислительную технику в своей деятельности, обратиться к ним. И спрос на развитые удобные программы обработки данных заставлял поставщиков программного обеспечения поставлять все новые системы, которые принято называть настольными (desktop) СУБД. Значительная конкуренция среди поставщиков заставляла совершенствовать эти системы, предлагая новые возможности, улучшая интерфейс и быстродействие систем, снижая их стоимость. Наличие на рынке большого числа СУБД, выполняющих сходные функции, потребовало разработки методов экспорта-импорта данных для этих систем и открытия форматов хранения данных.

Но и в этот период появлялись любители, которые вопреки здравому смыслу разрабатывали собственные СУБД, используя стандартные языки программирования. Это был тупиковый вариант, потому что дальнейшее развитие показало, что перенести данные из нестандартных форматов в новые СУБД было гораздо труднее, а в некоторых случаях требовало таких трудозатрат, что легче было бы все разработать заново, но данные все равно надо было переносить на новую более перспективную СУБД. И это тоже было результатом недооценки тех функций, которые должна была выполнять СУБД.

Особенности этого этапа следующие:

- Все СУБД были рассчитаны на создание БД в основном с монопольным доступом. И это понятно. Компьютер персональный, он не был подсоединен к сети, и база данных на нем создавалась для работы одного пользователя. В редких случаях предполагалась последовательная работа нескольких пользователей, например, сначала оператор, который вводил бухгалтерские документы, а потом главбух, который определял проводки, соответствующие первичным документам.
- Большинство СУБД имели развитый и удобный пользовательский интерфейс. В большинстве существовал интерактивный режим работы с БД как в рамках описания БД, так и в рамках проектирования запросов. Кроме того, большинство СУБД предлагали развитый и удобный инструментарий для разработки готовых приложений без программирования. Инструментальная среда состояла из готовых элементов приложения в виде шаблонов экранных форм, отчетов, этикеток (Labels), графических конструкторов запросов, которые достаточно просто могли быть собраны в единый комплекс.
- Во всех настольных СУБД поддерживался только внешний уровень представления реляционной модели, то есть только внешний табличный вид структур данных.
- При наличии высокоуровневых языков манипулирования данными типа реляционной алгебры и SQL в настольных СУБД поддерживались низкоуровневые языки манипулирования данными на уровне отдельных строк таблиц.
- В настольных СУБД отсутствовали средства поддержки ссылочной и структурной целостности базы данных. Эти функции должны были выполнять приложения, однако скудость средств разработки приложений иногда не позволяла это сделать, и в этом случае эти функции должны были выполняться пользователем, требуя от него дополнительного контроля при вводе и изменении информации, хранящейся в БД.
- Наличие монопольного режима работы фактически привело к вырождению функций администрирования БД и в связи с этим — к отсутствию инструментальных средств администрирования БД.
- И, наконец, последняя и в настоящий момент весьма положительная особенность — это сравнительно скромные требования к аппаратному обеспечению со стороны настольных СУБД. Вполне работоспособные приложения, разработанные, например, на Clipper, работали на PC 286.

- В принципе, их даже трудно назвать полноценными СУБД. Яркие представители этого семейства — очень широко использовавшиеся до недавнего времени СУБД Dbase (DbaseIII+, DbaseIV), FoxPro, Clipper, Paradox.

Распределенные базы данных

Хорошо известно, что история развивается по спирали, поэтому после процесса «персонализации» начался обратный процесс — интеграция. Множится количество локальных сетей, все больше информации передается между компьютерами, остро встает задача согласованности данных, хранящихся и обрабатывающихся в разных местах, но логически друг с другом связанных, возникают задачи, связанные с параллельной обработкой *транзакций* — последовательностей операций над БД, переводящих ее из одного непротиворечивого состояния в другое непротиворечивое состояние. Успешное решение этих задач приводит к появлению *распределенных баз данных*, сохраняющих все преимущества настольных СУБД и в то же время позволяющих организовать параллельную обработку информации и поддержку целостности БД.

Особенности данного этапа:

- Практически все современные СУБД обеспечивают поддержку полной реляционной модели, а именно:
  - структурной целостности — допустимыми являются только данные, представленные в виде отношений реляционной модели;
  - языковой целостности, то есть языков манипулирования данными высокого уровня (в основном SQL);
  - ссылочной целостности, контроля за соблюдением ссылочной целостности в течение всего времени функционирования системы, и гарантий невозможности со стороны СУБД нарушить эти ограничения.
- Большинство современных СУБД рассчитаны на многоплатформенную архитектуру, то есть они могут работать на компьютерах с разной архитектурой и под разными операционными системами, при этом для пользователей доступ к данным, управляемым СУБД на разных платформах, практически неразличим.
- Необходимость поддержки многопользовательской работы с базой данных и возможность децентрализованного хранения данных потребовали развития средств администрирования БД с реализацией общей концепции средств защиты данных.
- Потребность в новых реализациях вызвала создание серьезных теоретических трудов по оптимизации реализаций распределенных БД и работе с распределенными транзакциями и запросами с внедрением полученных результатов в коммерческие СУБД.
- Для того чтобы не потерять клиентов, которые ранее работали на настольных СУБД, практически все современные СУБД имеют средства подключения клиентских приложений, разработанных с использованием настольных СУБД, и средства экспорта данных из форматов настольных СУБД второго этапа развития.

- Именно к этому этапу можно отнести разработку ряда стандартов в рамках языков описания и манипулирования данными начиная с SQL89, SQL92, SQL99 и технологий по обмену данными между различными СУБД, к которым можно отнести и протокол ODBC (Open DataBase Connectivity), предложенный фирмой Microsoft.
- Именно к этому этапу можно отнести начало работ, связанных с концепцией объектно-ориентированных БД — ООБД. Представителями СУБД, относящимся ко второму этапу, можно считать MS Access 97 и все современные серверы баз данных Oracle7.3,Oracle 8.4 MS SQL6.5, MS SQL7.0, System 10, System 11, Informix, DB2, SQL Base и другие современные серверы баз данных, которых в настоящий момент насчитывается несколько десятков.

Перспективы развития систем управления базами данных

Этот этап характеризуется появлением новой технологии доступа к данным — *интранет*. Основное отличие этого подхода от технологии клиент-сервер состоит в том, что отпадает необходимость использования специализированного клиентского программного обеспечения. Для работы с удаленной базой данных используется стандартный броузер Интернета, например Microsoft Internet Explorer или Netscape Navigator, и для конечного пользователя процесс обращения к данным происходит аналогично скольжению по Всемирной Паутине (см. рис. 1.1). При этом встроенный в загружаемые пользователем HTML-страницы код, написанный обычно на языке Java, Java-script, Perl и других, отслеживает все действия пользователя и транслирует их в низкоуровневые SQL-запросы к базе данных, выполняя, таким образом, ту работу, которой в технологии клиент-сервер занимается клиентская программа. Удобство данного подхода привело к тому, что он стал использоваться не только для удаленного доступа к базам данных, но и для пользователей локальной сети предприятия. Простые задачи обработки данных, не связанные со сложными алгоритмами, требующими согласованного изменения данных во многих взаимосвязанных объектах, достаточно просто и эффективно могут быть построены по данной архитектуре. В этом случае для подключения нового пользователя к возможности использовать данную задачу не требуется установка дополнительного клиентского программного обеспечения. Однако алгоритмически сложные задачи рекомендуется реализовывать в архитектуре «клиент-сервер» с разработкой специального клиентского программного обеспечения.

У каждого из вышеперечисленных подходов к работе с данными есть свои достоинства и свои недостатки, которые и определяют область применения того или иного метода, и в настоящее время все подходы широко используются.

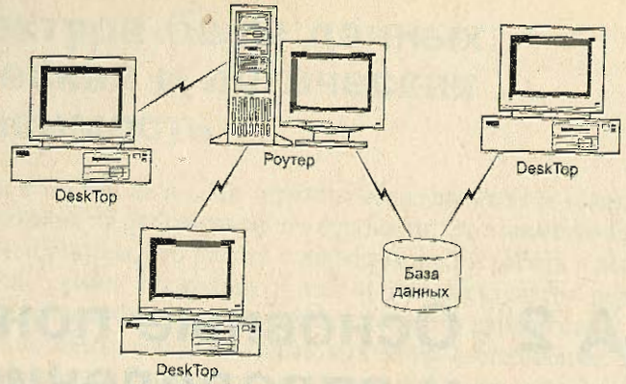


Рис. 1.1. Взаимодействие с базой данных в технологии интранет

Контрольные вопросы

1. Найдите сходства первого и четвертого этапов развития.
2. Найдите отличия первого и третьего этапов развития.
3. Если при использовании файловых систем для параллельного доступа пользователей создавать копии файлов для каждого пользователя, может ли это ускорить параллельную работу с информацией?

ГЛАВА 2 Основные понятия и определения

Современные авторы часто употребляют термины «банк данных» и «база данных» как синонимы, однако в общеотраслевых руководящих материалах по созданию банков данных Государственного комитета по науке и технике (ГКНТ), изданных в 1982 г., эти понятия различаются. Там приводятся следующие определения банка данных, базы данных и СУБД:

Банк данных (БнД) — это система специальным образом организованных данных — баз данных, программных, технических, языковых, организационно-методических средств, предназначенных для обеспечения централизованного накопления и коллективного многоцелевого использования данных.

База данных (БД) — именованная совокупность данных, отражающая состояние объектов и их отношений в рассматриваемой предметной области.

Система управления базами данных (СУБД) — совокупность языковых и программных средств, предназначенных для создания, ведения и совместного использования БД многими пользователями.

Сухой канцелярский язык труден для восприятия, но эти определения четко разграничивают назначение всех трех базовых понятий, и мы можем принять их за основу.

Программы, с помощью которых пользователи работают с базой данных, называются *приложениями*. В общем случае с одной базой данных могут работать множество различных приложений. Например, если база данных моделирует некоторое предприятие, то для работы с ней может быть создано приложение, которое обслуживает подсистему учета кадров, другое приложение может быть посвящено работе подсистемы расчета заработной платы сотрудников, третье приложение работает как подсистемы складского учета, четвертое приложение посвящено планированию производственного процесса. При рассмотрении приложений, работающих с одной базой данных, предполагается, что они могут работать параллельно и независимо друг от друга, и именно СУБД призвана обеспечить работу множества приложений с единой базой данных таким образом, чтобы каждое из них выполнялось корректно, но учитывало все изменения в базе данных, вносимые другими приложениями.

Архитектура базы данных. Физическая и логическая независимость

Терминология в СУБД, да и сами термины «база данных» и «банк данных» частично заимствованы из финансовой деятельности. Это заимствование — не случайно и объясняется тем, что работа с информацией и работа с денежными массами во многом схожи, поскольку и там и там отсутствует персонификация объекта обработки: две банкноты достоинством в сто рублей столь же неотличимы и взаимозаменяемы, как два одинаковых байта (естественно, за исключением серийных номеров). Вы можете положить деньги на некоторый счет и предоставить возможность вашим родственникам или коллегам использовать их для иных целей. Вы можете поручить банку оплачивать ваши расходы с вашего счета или получить их наличными в другом банке, и это будут уже другие денежные купюры, но их ценность будет эквивалентна той, которую вы имели, когда клали их на ваш счет.

В процессе научных исследований, посвященных тому, как именно должна быть устроена СУБД, предлагались различные способы реализации. Самым жизнеспособным из них оказалась предложенная американским комитетом по стандартизации ANSI (American National Standards Institute) трехуровневая система организации БД, изображенная на рис. 2.1:

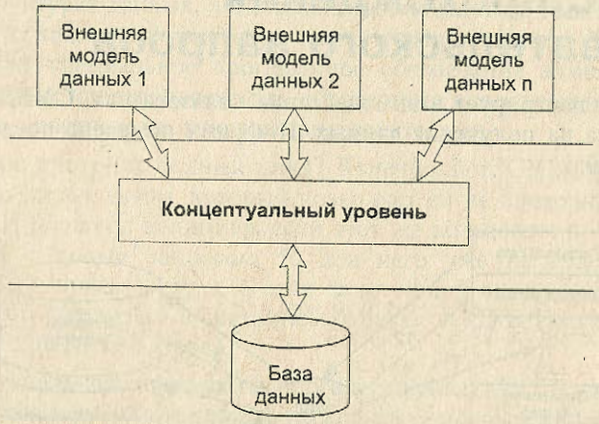


Рис. 2.1. Трехуровневая модель системы управления базой данных, предложенная ANSI

1. Уровень внешних моделей — самый верхний уровень, где каждая модель имеет свое «видение» данных. Этот уровень определяет точку зрения на БД отдельных приложений. Каждое приложение видит и обрабатывает только те данные, которые необходимы именно этому приложению. Например, система распределения работ использует сведения о квалификации сотрудника, но ее

не интересуют сведения об окладе, домашнем адресе и телефоне сотрудника, и наоборот, именно эти сведения используются в подсистеме отдела кадров.

2. Концептуальный уровень — центральное управляющее звено, здесь база данных представлена в наиболее общем виде, который объединяет данные, используемые всеми приложениями, работающими с данной базой данных. Фактически концептуальный уровень отражает обобщенную модель предметной области (объектов реального мира), для которой создавалась база данных. Как любая модель, концептуальная модель отражает только существенные, с точки зрения обработки, особенности объектов реального мира.
3. Физический уровень — собственно данные, расположенные в файлах или в страничных структурах, расположенных на внешних носителях информации.

Эта архитектура позволяет обеспечить логическую (между уровнями 1 и 2) и физическую (между уровнями 2 и 3) независимость при работе с данными. Логическая независимость предполагает возможность изменения одного приложения без корректировки других приложений, работающих с этой же базой данных. Физическая независимость предполагает возможность переноса хранимой информации с одних носителей на другие при сохранении работоспособности всех приложений, работающих с данной базой данных. Это именно то, чего не хватало при использовании файловых систем.

Выделение концептуального уровня позволило разработать аппарат централизованного управления базой данных.

Процесс прохождения пользовательского запроса

Рисунок 2.2 иллюстрирует взаимодействие пользователя, СУБД и ОС при обработке запроса на получение данных. Цифрами помечена последовательность взаимодействий:

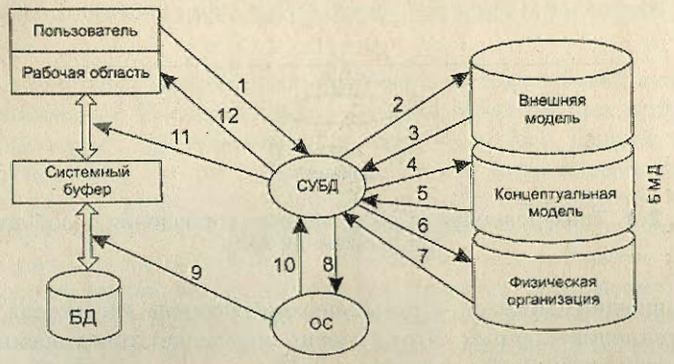


Рис. 2.2. Схема прохождения запроса к БД

1. Пользователь посылает СУБД запрос на получение данных из БД.
2. Анализ прав пользователя и внешней модели данных, соответствующей данному пользователю, подтверждает или запрещает доступ данного пользователя к запрошенным данным.
3. В случае запрета на доступ к данным СУБД сообщает пользователю об этом (стрелка 12) и прекращает дальнейший процесс обработки данных, в противном случае СУБД определяет часть концептуальной модели, которая затрагивается запросом пользователя.
4. СУБД получает информацию о запрошенной части концептуальной модели.
5. СУБД запрашивает информацию о местоположении данных на физическом уровне (файлы или физические адреса).
6. В СУБД возвращается информация о местоположении данных в терминах операционной системы.
7. СУБД вежливо просит операционную систему предоставить необходимые данные, используя средства операционной системы.
8. Операционная система осуществляет перекачку информации из устройств хранения и пересылает ее в системный буфер.
9. Операционная система оповещает СУБД об окончании пересылки.
10. СУБД выбирает из доставленной информации, находящейся в системном буфере, только то, что нужно пользователю, и пересылает эти данные в рабочую область пользователя.

БМД — это *База Метаданных*, именно здесь и хранится вся информация об используемых структурах данных, логической организации данных, правах доступа пользователей и, наконец, физическом расположении данных. Для управления БМД существует специальное программное обеспечение администрирования баз данных, которое предназначено для корректного использования единого информационного пространства многими пользователями.

Всегда ли запрос проходит полный цикл? Конечно, нет. СУБД обладает достаточно развитым интеллектом, который позволяет ей не повторять бессмысленных действий. И поэтому, например, если этот же пользователь повторно обратится к СУБД с новым запросом, то для него уже не будут проверяться внешняя модель и права доступа, а если дальнейший анализ запроса покажет, что данные могут находиться в системном буфере, то СУБД осуществит только 11 и 12 шаги в обработке запроса.

Разумеется, механизм прохождения запроса в реальных СУБД гораздо сложнее, но и эта упрощенная схема показывает, насколько серьезными и сложными должны быть механизмы обработки запросов, поддерживаемые реальными СУБД.

Пользователи банков данных

Как любой программно-организационно-технический комплекс, банк данных существует во времени и в пространстве. Он имеет определенные стадии своего развития:

1. Проектирование.
2. Реализация.
3. Эксплуатация;
4. Модернизация и развитие.
5. Полная реорганизация.

На каждом этапе своего существования с банком данных связаны разные категории пользователей.

Определим основные категории пользователей и их роль в функционировании банка данных:

- **Конечные пользователи.** Это основная категория пользователей, в интересах которых и создается банк данных. В зависимости от особенностей создаваемого банка данных круг его конечных пользователей может существенно различаться. Это могут быть случайные пользователи, обращающиеся к БД время от времени за получением некоторой информации, а могут быть регулярные пользователи. В качестве случайных пользователей могут рассматриваться, например, возможные клиенты вашей фирмы, просматривающие каталог вашей продукции или услуг с обобщенным или подробным описанием того и другого. Регулярными пользователями могут быть ваши сотрудники, работающие со специально разработанными для них программами, которые обеспечивают автоматизацию их деятельности при выполнении своих должностных обязанностей. Например, менеджер, планирующий работу сервисного отдела компьютерной фирмы, имеет в своем распоряжении программу, которая помогает ему планировать и распределять текущие заказы, контролировать ход их выполнения, заказывать на складе необходимые комплектующие для новых заказов. Главный принцип состоит в том, что от конечных пользователей не должно требоваться каких-либо специальных знаний в области вычислительной техники и языковых средств.
- **Администраторы банка данных.** Это группа пользователей, которая на начальной стадии разработки банка данных отвечает за его оптимальную организацию с точки зрения одновременной работы множества конечных пользователей, на стадии эксплуатации отвечает за корректность работы данного банка информации в многопользовательском режиме. На стадии развития и реорганизации эта группа пользователей отвечает за возможность корректной реорганизации банка без изменения или прекращения его текущей эксплуатации.
- **Разработчики и администраторы приложений.** Это группа пользователей, которая функционирует во время проектирования, создания и реорганизации банка данных. Администраторы приложений координируют работу разработчиков при разработке конкретного приложения или группы приложений, объединенных в функциональную подсистему. Разработчики конкретных приложений работают с той частью информации из базы данных, которая требуется для конкретного приложения.

Не в каждом банке данных могут быть выделены все типы пользователей. Мы уже знаем, что при разработке информационных систем с использованием настольных СУБД администратор банка данных, администратор приложений и разработчик часто существовали в одном лице. Однако при построении современных сложных корпоративных баз данных, которые используются для автоматизации всех или большей части бизнес-процессов в крупной фирме или корпорации, могут существовать и группы администраторов приложений, и отделы разработчиков. Наиболее сложные обязанности возложены на группу администратора БД.

Рассмотрим их более подробно.

В составе группы администратора БД должны быть:

- системные аналитики;
- проектировщики структур данных и внешнего по отношению к банку данных информационного обеспечения;
- проектировщики технологических процессов обработки данных;
- системные и прикладные программисты;
- операторы и специалисты по техническому обслуживанию.

Если речь идет о коммерческом банке данных, то важную роль здесь играют специалисты по маркетингу.

Основные функции группы администратора БД

1. **Анализ предметной области**: описание предметной области, выявление ограничений целостности, определение статуса (доступности, секретности) информации, определение потребностей пользователей, определение соответствия «данные—пользователь», определение объемно-временных характеристик обработки данных.
2. **Проектирование структуры БД**: определение состава и структуры файлов БД и связей между ними, выбор методов упорядочения данных и методов доступа к информации, описание БД на языке описания данных (ЯОД).
3. **Задание ограничений целостности при описании структуры БД и процедур обработки БД**:
   - задание декларативных ограничений целостности, присущих предметной области;
   - определение динамических ограничений целостности, присущих предметной области в процессе изменения информации, хранящейся в БД;
   - определение ограничений целостности, вызванных структурой БД;
   - разработка процедур обеспечения целостности БД при вводе и корректировке данных;
   - определение ограничений целостности при параллельной работе пользователей в многопользовательском режиме.
4. **Первоначальная загрузка и ведение БД**:
   - разработка технологии первоначальной загрузки БД, которая будет отличаться от процедуры модификации и дополнения данными при штатном использовании базы данных;

- разработка технологии проверки соответствия введенных данных реальному состоянию предметной области. База данных моделирует реальные объекты некоторой предметной области и взаимосвязи между ними, и на момент начала штатной эксплуатации эта модель должна полностью соответствовать состоянию объектов предметной области на данный момент времени;
- в соответствии с разработанной технологией первоначальной загрузки может понадобиться проектирование системы первоначального ввода данных.

5. **Защита данных:**
   - определение системы паролей, принципов регистрации пользователей, создание групп пользователей, обладающих одинаковыми правами доступа к данным;
   - разработка принципов защиты конкретных данных и объектов проектирования; разработка специализированных методов кодирования информации при ее циркуляции в локальной и глобальной информационных сетях;
   - разработка средств фиксации доступа к данным и попыток нарушения системы защиты;
   - тестирование системы защиты;
   - исследование случаев нарушения системы защиты и развитие динамических методов защиты информации в БД.
6. **Обеспечение восстановления БД**:
   - разработка организационных средств архивирования и принципов восстановления БД;
   - разработка дополнительных программных средств и технологических процессов восстановления БД после сбоев.
7. **Анализ обращений пользователей БД**: сбор статистики по характеру запросов, по времени их выполнения, по требуемым выходным документам
8. **Анализ эффективности функционирования БД:**
   - анализ показателей функционирования БД;
   - планирование реструктуризации (изменение структуры) БД и реорганизации БнД.
9. **Работа с конечными пользователями**:
   - сбор информации об изменении предметной области;
   - сбор информации об оценке работы БД;
   - обучение пользователей, консультирование пользователей;
   - разработка необходимой методической и учебной документации по работе конечных пользователей.
10. **Подготовка и поддержание системных средств:**
    - анализ существующих на рынке программных средств и анализ возможности и необходимости их использования в рамках БД;
    - разработка требуемых организационных и программно-технических мероприятий по развитию БД;
    - проверка работоспособности закупаемых программных средств перед подключением их к БД;
    - курирование подключения новых программных средств к БД.
11. **Организационно-методическая работа по проектированию БД**:
    - выбор или создание методики проектирования БД;
    - определение целей и направления развития системы в целом;
    - планирование этапов развития БД;
    - разработка общих словарей-справочников проекта БД и концептуальной модели;
    - стыковка внешних моделей разрабатываемых приложений;
    - курирование подключения нового приложения к действующей БД;
    - обеспечение возможности комплексной отладки множества приложений, взаимодействующих с одной БД.

Классификация моделей данных

Одними из основополагающих в концепции баз данных являются обобщенные категории «данные» и «модель данных».

Понятие «данные» в концепции баз данных — это набор конкретных значений, параметров, характеризующих объект, условие, ситуацию или любые другие факторы. Примеры данных: Петров Николай Степанович, $30 и т. д. Данные не обладают определенной структурой, данные становятся информацией тогда, когда пользователь задает им определенную структуру, то есть осознает их смысловое содержание. Поэтому центральным понятием в области баз данных является понятие модели. Не существует однозначного определения этого термина, у разных авторов эта абстракция определяется с некоторыми различиями, но тем не менее можно выделить нечто общее в этих определениях.

Модель данных — это некоторая абстракция, которая, будучи приложима к конкретным данным, позволяет пользователям и разработчикам трактовать их уже как информацию, то есть сведения, содержащие не только данные, но и взаимосвязь между ними.

На рис. 2.3 представлена классификация моделей данных.

В соответствии с рассмотренной ранее трехуровневой архитектурой мы сталкиваемся с понятием модели данных по отношению к каждому уровню. И действительно, физическая модель данных оперирует категориями, касающимися организации внешней памяти и структур хранения, используемых в данной операционной среде. В настоящий момент в качестве физических моделей используются различные методы размещения данных, основанные на файловых структурах: это организация файлов прямого и последовательного доступа, индексных файлов и инвертированных файлов, файлов, использующих различные методы хэширования, взаимосвязанных файлов. Кроме того, современные СУБД широко используют страничную организацию данных. Физические мо-

дели данных, основанные на страничной организации, являются наиболее перспективными.

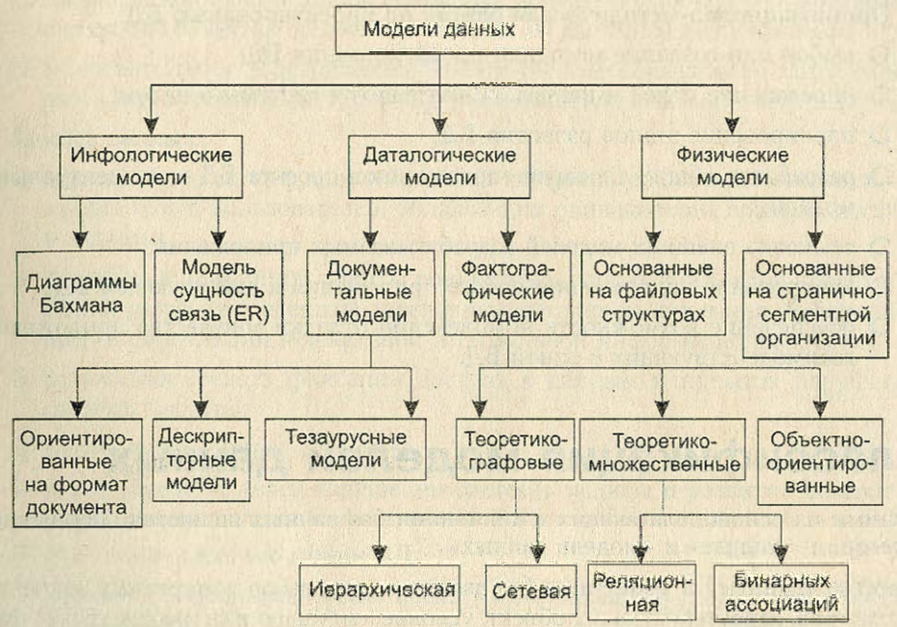


Рис. 2.3. Классификация моделей данных

Наибольший интерес вызывают модели данных, используемые на концептуальном уровне. По отношению к ним внешние модели называются подсхемами и используют те же абстрактные категории, что и концептуальные модели данных.

Кроме трех рассмотренных уровней абстракции при проектировании БД существует еще один уровень, предшествующий им. Модель этого уровня должна выражать информацию о предметной области в виде, независимом от используемой СУБД. Эти модели называются *инфологическими*, или *семантическими*, и отражают в естественной и удобной для разработчиков и других пользователей форме информационно-логический уровень абстрагирования, связанный с фиксацией и описанием объектов предметной области, их свойств и их взаимосвязей.

Инфологические модели данных используются на ранних стадиях проектирования для описания структур данных в процессе разработки приложения, а *даталогические* модели уже поддерживаются конкретной СУБД.

Документальные модели данных соответствуют представлению о слабоструктурированной информации, ориентированной в основном на свободные форматы документов, текстов на естественном языке.

Модели, основанные на языках разметки документов, связаны прежде всего со стандартным общим языком разметки — SGML (Standart Generalised Markup

Language), который был утвержден ISO в качестве стандарта еще в 80-х годах. Этот язык предназначен для создания других языков разметки, он определяет допустимый набор тегов (ссылок), их атрибуты и внутреннюю структуру документа. Контроль за правильностью использования тегов осуществляется при помощи специального набора правил, называемых DTD-описаниями, которые используются программой клиента при разборе документа. Для каждого класса документов определяется свой набор правил, описывающих грамматику соответствующего языка разметки. С помощью SGML можно описывать структурированные данные, организовывать информацию, содержащуюся в документах, представлять эту информацию в некотором стандартизованном формате. Но ввиду некоторой своей сложности SGML использовался в основном для описания синтаксиса других языков (наиболее известным из которых является HTML), и немногие приложения работали с SGML-документами напрямую.

Гораздо более простой и удобный, чем SGML, язык HTML позволяет определять оформление элементов документа и имеет некий ограниченный набор инструкций — тегов, при помощи которых осуществляется процесс разметки. Инструкции HTML в первую очередь предназначены для управления процессом вывода содержимого документа на экране программы-клиента и определяют этим самым способ представления документа, но не его структуру. В качестве элемента гипертекстовой базы данных, описываемой HTML, используется текстовый файл, который может легко передаваться по сети с использованием протокола HTTP. Эта особенность, а также то, что HTML является открытым стандартом и огромное количество пользователей имеет возможность применять возможности этого языка для оформления своих документов, безусловно, повлияли на рост популярности HTML и сделали его сегодня главным механизмом представления информации в Интернете.

Однако HTML сегодня уже не удовлетворяет в полной мере требованиям, предъявляемым современными разработчиками к языкам подобного рода. И ему на смену был предложен новый язык гипертекстовой разметки, мощный, гибкий и, одновременно с этим, удобный язык XML. В чем же заключаются его достоинства?

XML (Extensible Markup Language) — это язык разметки, описывающий целый класс объектов данных, называемых XML-документами. Он используется в качестве средства для описания грамматики других языков и контроля за правильностью составления документов. То есть сам по себе XML не содержит никаких тегов, предназначенных для разметки, он просто определяет порядок их создания.

Тезаурусные модели основаны на принципе организации словарей, содержат определенные языковые конструкции и принципы их взаимодействия в заданной грамматике. Эти модели эффективно используются в системах-переводчиках, особенно многоязыковых переводчиках. Принцип хранения информации в этих системах и подчиняется тезаурусным моделям.

Дескрипторные модели — самые простые из документальных моделей, они широко использовались на ранних стадиях использования документальных баз данных. В этих моделях каждому документу соответствовал дескриптор — описатель. Этот дескриптор имел жесткую структуру и описывал документ в соот-

ветствии с теми характеристиками, которые требуются для работы с документами в разрабатываемой документальной БД. Например, для БД, содержащей описание патентов, дескриптор содержал название области, к которой относился патент, номер патента, дату выдачи патента и еще ряд ключевых параметров, которые заполнялись для каждого патента. Обработка информации в таких базах данных велась исключительно по дескрипторам, то есть по тем параметрам, которые характеризовали патент, а не по самому тексту патента.

ГЛАВА 3 Теоретико-графовые модели данных

Как уже упоминалось ранее, эти модели отражают совокупность объектов реального мира в виде графа взаимосвязанных информационных объектов. В зависимости от типа графа выделяют иерархическую или сетевую модели. Исторически эти модели появились раньше, и в настоящий момент они используются реже, чем более современная реляционная модель данных. Однако до сих пор существуют системы, работающие на основе этих моделей, а одна из концепций развития объектно-ориентированных баз данных предполагает объединение принципов сетевой модели с концепцией реляционной.

Иерархическая модель данных

Иерархическая модель данных является наиболее простой среди всех даталогических моделей. Исторически она появилась первой среди всех даталогических моделей: именно эту модель поддерживает первая из зарегистрированных промышленных СУБД IMS фирмы IBM.

Появление иерархической модели связано с тем, что в реальном мире очень многие связи соответствуют иерархии, когда один объект выступает как родительский, а с ним может быть связано множество подчиненных объектов. Иерархия проста и естественна в отображении взаимосвязи между классами объектов.

Основными информационными единицами в иерархической модели являются: *база данных* (*БД*), *сегмент и поле*. Поле данных определяется как минимальная, неделимая единица данных, доступная пользователю с помощью СУБД. Например, если в задачах требуется печатать в документах адрес клиента, но не требуется дополнительного анализа полного адреса, то есть города, улицы, дома, квартиры, то мы можем принять весь адрес за элемент данных, и он будет храниться полностью, а пользователь сможет получить его только как полную строку символов из БД. Если же в наших задачах существует анализ частей, составляющих адрес, например города, где расположен клиент, то нам необходимо выделить город как отдельный элемент данных, только в этом случае пользователь может

получить к нему доступ и выполнить, например, запрос на поиск всех клиентов, которые проживают в конкретном городе, например в Париже. Однако если пользователю понадобится и полный адрес клиента, то остальную информацию по адресу также необходимо хранить в отдельном поле, которое может быть названо, например, Сокращенный адрес. В этом случае для каждого клиента в БД хранится как Город, так и Сокращенный адрес.

Сегмент в терминологии Американской Ассоциации по базам данных DBTG (Data Base Task Group) называется *записью*, при этом в рамках иерархической модели определяются два понятия: *тип сегмента* или *тип записи* и *экземпляр сегмента* или *экземпляр записи*.

Тип сегмента — это поименованная совокупность типов элементов данных, в него входящих. *Экземпляр сегмента* образуется из конкретных значений полей или элементов данных, в него входящих. Каждый тип сегмента в рамках иерархической модели образует некоторый набор однородных записей. Для возможности различия отдельных записей в данном наборе каждый тип сегмента должен иметь ключ или набор ключевых атрибутов (полей, элементов данных). Ключом называется набор элементов данных, однозначно идентифицирующих экземпляр сегмента. Например, рассматривая тип сегмента, описывающий сотрудника организации, мы должны выделить те характеристики сотрудника, которые могут его однозначно идентифицировать в рамках БД предприятия. Если предположить, что на предприятии могут работать однофамильцы, то, вероятно, наиболее надежным будет идентифицировать сотрудника по его табельному номеру. Однако если мы будем строить БД, содержащую описание множества граждан, например нашей страны, то, скорее всего, нам придется в качестве ключа выбрать совокупность полей, отражающих его паспортные данные.

В иерархической модели сегменты объединяются в ориентированный древовидный граф. При этом полагают, что направленные ребра графа отражают иерархические связи между сегментами: каждому экземпляру сегмента, стоящему выше по иерархии и соединенному с данным типом сегмента, соответствует несколько (множество) экземпляров данного (подчиненного) типа сегмента. Тип сегмента, находящийся на более высоком уровне иерархии, называется логически исходным по отношению к типам сегментов, соединенным с данным направленными иерархическими ребрами, которые в свою очередь называются логически подчиненными по отношению к этому типу сегмента. Иногда исходные сегменты называют сегментами-предками, а подчиненные сегменты называют сегментами-потомками.

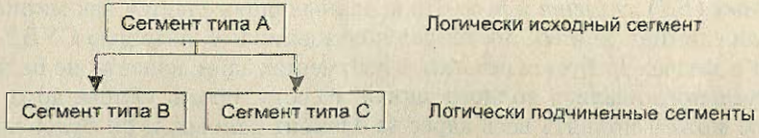


Рис. 3.1. Пример иерархических связей между сегментами

На концептуальном уровне определяется понятие схемы БД в терминологии иерархической модели.

Схема иерархической БД представляет собой совокупность отдельных деревьев, каждое дерево в рамках модели называется *физической базой данных*. Каждая физическая БД удовлетворяет следующим иерархическим ограничениям:

- в каждой физической БД существует один корневой сегмент, то есть сегмент, у которого нет логически исходного (родительского) типа сегмента;
- каждый логически исходный сегмент может быть связан с произвольным числом логически подчиненных сегментов;
- каждый логически подчиненный сегмент может быть связан только с одним логически исходным (родительским) сегментом.

Очень важно понимать различие между сегментом и типом сегмента — оно такое же, как между типом переменной и самой переменной: сегмент является экземпляром типа сегмента. Например, у нас может быть тип сегмента Группа (Номер, Староста) и сегменты этого типа, такие как (4305, Петров Ф. И.) или (383, Кустова Т. С.).

Между экземплярами сегментов также существуют иерархические связи. Рассмотрим, например, иерархический граф, представленный на рис. 3.2.

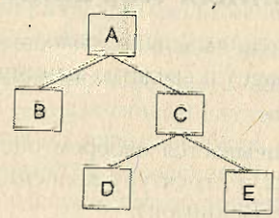


Рис. 3.2. Пример структуры иерархического дерева

Каждый тип сегмента может иметь множество соответствующих ему экземпляров. Между экземплярами сегментов также существуют иерархические связи.

На рис. 3.3 представлены 2 экземпляра иерархического дерева соответствующей структуры.

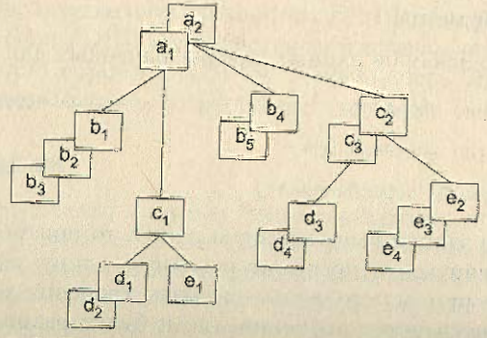


Рис. 3.3. Пример двух экземпляров данного дерева

Экземпляры-потомки одного типа, связанные с одним экземпляром сегмента-предка, называют «близнецами». Так, для нашего примера экземпляры b1, b2 и b3 являются «близнецами», но экземпляр b4 подчинен другому экземпляру родительского сегмента, и он не является «близнецом» по отношению к экземплярам b1, b2 и b3. Набор всех экземпляров сегментов, подчиненных одному экземпляру корневого сегмента, называется *физической записью*. Количество экземпляров-потомков может быть разным для разных экземпляров родительских сегментов, поэтому в общем случае физические записи имеют разную длину. Так, используя принцип линейной записи иерархических графов, пример на рис. 3.3 можно представить в виде двух записей:

$a_1\ b_1\ b_2\ b_3\ c_1\ d_1\ d_2\ e_1$	$a_2\ b_4\ b_5\ c_2\ d_3\ d_4\ e_2\ e_3\ e_4$
Запись 1	Запись 2

Как видно из нашего примера, физические записи в иерархической модели различаются по длине и структуре.

Язык описания данных иерархической модели

В рамках иерархической модели выделяют языковые средства описания данных (DDL, Data Definition Language) и средства манипулирования данными (DML, Data Manipulation Language).

Каждая физическая база описывается набором операторов, определяющих как ее логическую структуру, так и структуру хранения БД. Описание начинается с оператора DBD (Data Base Definition):

```
DBD Name = < имя БД>, ACCESS = < способ доступа>
```

Способ доступа определяет способ организации взаимосвязи физических записей. Определено 5 способов доступа: *HSAM — hierarchical sequential access method* (иерархически последовательный метод), *HISAM — hierarchical index sequential access method* (иерархически индексно-последовательный метод), *HDAM — hierarchical direct access method* (иерархически прямой метод), *HIDAM — hierarchical index direct access method* (иерархически индексно-прямой метод), *INDEX* — индексный метод.

Далее идет описание наборов данных, предназначенных для хранения БД:

```
DATA SET DD1 = < имя оператора, определяющего хранимый набор данных>,
DEVICE =< устройство хранения БД>,
[OVFLW = < имя области переполнения>]
```

Так как физические записи имеют разную длину, то при модификации данных запись может увеличиться и превысит исходную длину записи до модификации. В этом случае при определенных методах хранения может понадобиться дополнительное пространство хранения, где и будут размещены дополнительные данные. Это пространство и называется областью переполнения.

После описания всей физической БД идет описание типов сегментов, ее составляющих, в соответствии с иерархией. Описание сегментов всегда начинается с описания корневого сегмента. Общая схема описания типа сегмента такова:

```
SEGM NAME = < имя сегмента>, BYTES =< размер в байтах>,
    FREQ = <средняя частота реализаций сегмента под одним исходным>
    PARENT = <имя родительского сегмента>
```

Параметр FREQ определяет среднее количество экземпляров данного сегмента, связанных с одним экземпляром родительского сегмента. Для корневого сегмента это число возможных экземпляров корневого сегмента.

Для корневого сегмента параметр PARENT равен 0 (нулю).

Далее для каждого сегмента дается описание полей:

```
FIELD NAME = {(<имя поля> [, SEQ],{U | M}) | <имя поля> },
    START = < номер байта, с которого начинается значения поля >,
    BYTES = <размер поля в байтах>,
    TYPE = {X | P | C}
```

Признак SEQ — задается для ключевого поля, если экземпляры данного сегмента физически упорядочены в соответствии со значениями данного поля.

Параметр U задается, если значения ключевого поля уникальны для всех экземпляров данного сегмента, M — в противном случае. Если поле является ключевым, то его описание задается в круглых скобках, в противном случае имя поля задается без скобок. Параметр TYPE определяет тип данных. Для ранних иерархических моделей были определены только три типа данных: X — шестнадцатеричный, P —упакованный десятичный, C — символьный.

Заканчивается описание схемы вызовом процедуры генерации:

- DBDGEN — указывает на конец последовательности управляющих операторов описания БД;
- FINISH — устанавливает ненулевой код завершения при обнаружении ошибки;
- END — конец.

В системе может быть несколько физических БД (ФБД), но каждая из них описывается отдельно своим DBD и ей присваивается уникальное имя. Каждая ФБД содержит только один корневой сегмент. Совокупность ФБД образует концептуальную модель данных.

Внешние модели

При работе с иерархической моделью каждая программа, пользователь или приложение определяет свою внешнюю модель. Внешняя модель представляет собой совокупность подлеревьев для физических баз данных, с которыми работает данный пользователь. Каждый подграф внешней модели в обязательном порядке должен содержать корневой тип сегмента соответствующей физической базы данных концептуальной модели.

Представление внешней модели называется логической базой данных и определяется совокупностью блоков связи данного приложения с физическими БД, входящими в концептуальную схему БД. Блок связи — *PCB, program communication bloc* — описывает связь с одной физической БД по следующим правилам:

```
DBD NAME = < имя логической БД (подсхемы)> . ACCESS = LOGICAL
DATA SET = LOGICAL.
SEGM NAME = <имя сегмента в подсхеме>, PARENT =<имя родительского сегмента в
подсхеме>.   SOURSE =(Имя соответствующего сегмента ФБД, имя ФБД)
...
DBDGEN
FINISH
END
```

Совокупность блоков PCB образует полное внешнее представление данного приложения, называемое «блоком спецификации программ» (*PSB, program specification block*).

Рассмотрим пример иерархической БД.

Наша организация занимается производством и продажей компьютеров, в рамках производства мы комплектуем компьютеры из готовых деталей по индивидуальным заказам. У нас существует несколько базовых моделей, которые мы продаем без предварительных заказов по наличию на складе. В организации существуют несколько филиалов (рис. 3.4) и несколько складов, на которых хранятся комплектующие. Нам необходимо вести учет продаваемой продукции.

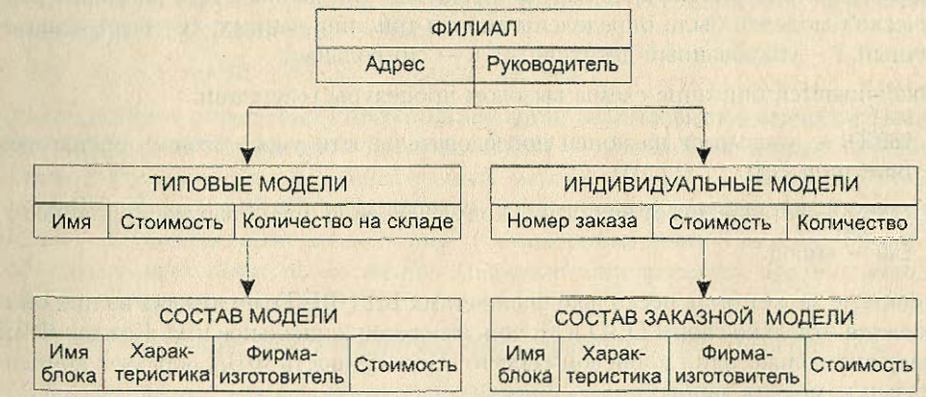


Рис. 3.4. Физическая БД «Филиалы»

Какие задачи нам надо решать в ходе разработки приложения?

- При приеме заказа мы должны выяснить, какую модель заказывает заказчик: типичную или индивидуальную комплектацию.
- Если заказывается типичная модель, то выясняется, какая модель и есть ли она в наличии, если модель есть, то надо уменьшить количество компьютеров данной модели в данном филиале на покупаемое количество. На этом будем считать заказ выполненным, однако при оформлении заказа может потребоваться задание полной спецификации покупаемого изделия.
- Если заказывается индивидуальная модель, то требуется описать весь состав новой модели.

Для того чтобы можно было бы принимать заказы на индивидуальные модели, нам понадобится информация о наличие конкретных деталей на складе, в этом случае нам необходимо второе дерево — Склады (см. рис. 3.5).

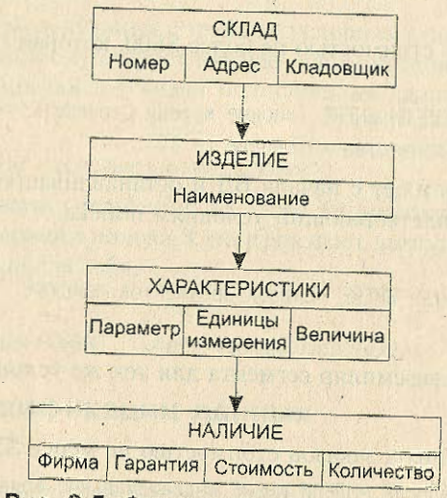


Рис. 3.5. Физическая модель «Склады»

Язык манипулирования данными в иерархических базах данных

Для доступа к базе данных у пользователя должна быть сформирована специальная среда окружения, поддерживающая в явном виде имеющиеся навигационные операции. Для этого в ней должны храниться:

- шаблоны всех записей логических баз данных, доступных пользователю;
- указатели на текущий экземпляр сегмента данного типа — для всех типов сегментов.

Язык манипулирования данными в иерархической модели поддерживает в явном виде навигационные операции. Эти операции связаны с перемещением указателя, который определяет текущий экземпляр конкретного сегмента.

Все операторы в языке манипулирования данными можно разделить на 3 группы. Первую группу составляют операторы поиска данных.

Операторы поиска данных

Синтаксис:

```
GET UNIQUE <имя сегмента> WHERE <список поиска>;
```

список поиска состоит из последовательности условий вида:

```
<имя сегмента>.<имя поля>ОС <constant или имя другого поля данного сегмента
или имя переменной>:
```

ОС — операция сравнения;

условия могут быть соединены логическими операциями И и ИЛИ {& , V}.

Назначение:

Получить единственное значение.

Пример:

Найти типовую модель стоимостью не более $600, которая существует не менее чем в 10 экземплярах.

```
GET UNIQUE ТИПОВЫЕ МОДЕЛИ WHERE Типовые модели.Стоимость <= $600
AND Типовые модели.Количество на складе >= 10
```

Данная команда всегда ищет с начала БД и останавливается, найдя первый экземпляр сегмента, удовлетворяющий условиям поиска.

Синтаксис:

```
GET NEXT <имя сегмента> WHERE <список аргументов поиска>
```

Назначение:

Получить следующий экземпляр сегмента для тех же условий.

Пример:

Напечатать полный список заказов стоимостью не менее $500.

```
GET UNIQUE ИНДИВИДУАЛЬНЫЕ МОДЕЛИ WHERE Индивидуальные модели.Стоимость >= $500
WHILE NOT FAIL (пока не конец поиска) DO
   PRINT № заказа, Стоимость, Количество
   GET NEXT ИНДИВИДУАЛЬНЫЕ МОДЕЛИ
END
```

Синтаксис:

```
GET NEXT <имя сегмента> WITHIN PARENT [ where <дополн.условия>]
```

Назначение:

Получить следующий для того же исходного.

Пример:

Получить перечень винчестеров, имеющихся на складе номер 1, в количестве не менее 10 с объемом 10 Гбайт.

```
GET UNIQUE СКЛАД WHERE Склад.Номер = 1
GET NEXT ИЗДЕЛИЕ WITHIN PARENT WHERE Изделие.Наименование = "Винчестер"
GET NEXT ХАРАКТЕРИСТИКИ WITHIN PARENT
WHERE ХАРАКТЕРИСТИКИ.Параметр = 10 AND
ХАРАКТЕРИСТИКИ.Единицы Измерения = Гб AND
ХАРАКТЕРИСТИКИ.Величина > 10
While Not Fail (пока поиск не завершен) DO
Get Next Within Parent
end
```


Операторы поиска данных с возможностью модификации

1. Найти и удержать единственный экземпляр сегмента. Эта операция подобна первой операции поиска GET UNIQUE, единственным отличием этой операции является то, что после выполнения этой операции над найденным экземпляром сегмента допустимы операции модификации (изменения) данных.

   Синтаксис:

   ```
   GET HOLD UNIQUE <имя сегмента> WHERE <список поиска>
   ```

2. Найти и удержать следующий с теми же условиями поиска. Аналогично операции 4 эта операция дублирует вторую операции поиска GET NEXT с возможностью выполнения последующей модификации данных.

   Синтаксис:

   ```
   GET HOLD NEXT [WHERE <дополнительные условия>]
   ```

3. Получить и удержать следующий для того же родителя. Эта операция является аналогом операции поиска 3, но разрешает выполнение операций модификации данных после себя.

   Синтаксис:

   ```
   GET HOLD NEXT WITHIN PARENT [ where <дополн.условия>]
   ```

Операторы модификации данных

1. Удалить

   Это первая из трех операций модификации.

   Синтаксис:

   ```
   DELETE
   ```

   Эта команда не имеет параметров. Почему? Потому что операции модификации действуют на экземпляр сегмента, найденный командами поиска с удержанием. А он всегда единственный текущий найденный и удерживаемый для модификации экземпляр конкретного сегмента. Поэтому при выполнении команды удаления будет удален именно этот экземпляр сегмента.

2. Обновить

   Синтаксис:

   ```
   UPDATE
   ```

   Как же происходит обновление, если мы и в этой команде не задаем никаких параметров. СУБД берет данные из рабочей области пользователя, где в шаблонах записей соответствующих внутренних переменных находятся значения полей каждого сегмента внешней модели, с которой работает данный пользователь. Именно этими значениями и обновляется текущий экземпляр сегмента. Значит, перед тем как выполнить операции модификации UPDATE, необходимо присвоить соответствующим переменным новые значения.

   Ввести новый экземпляр сегмента.

   ```
   INSERT <имя сегмента>
   ```

Эта команда позволяет ввести новый экземпляр сегмента, имя которого определено в параметре команды. Если мы вводим данные в сегмент, который является подчиненным некоторому родительскому экземпляру сегмента, то он будет внесен в БД и физически подключен к тому экземпляру родительского сегмента, который в данный момент является текущим.

Как видим, набор операций поиска и манипулирования данными в иерархической БД невелик, но он вполне достаточен для получения доступа к любому экземпляру любого сегмента БД. Однако следует отметить, что способ доступа, который применяется в данной модели, связан с последовательным перемещением от одного экземпляра сегмента к другому. Такой способ напоминает движение летательного аппарата или корабля по заданным координатам и называется навигационным.

Сетевая модель данных

Стандарт сетевой модели впервые был определен в 1975 году организацией CODASYL (Conference of Data System Languages), которая определила базовые понятия модели и формальный язык описания.

Базовыми объектами модели являются:

- элемент данных;
- агрегат данных;
- запись;
- набор данных.

Элемент данных — то же, что и в иерархической модели, то есть минимальная информационная единица, доступная пользователю с использованием СУБД.

Агрегат данных соответствует следующему уровню обобщения в модели. В модели определены агрегаты двух типов: агрегат типа *вектор* и агрегат типа *повторяющаяся группа*.

Агрегат данных имеет имя, и в системе допустимо обращение к агрегату по имени. Агрегат типа вектор соответствует линейному набору элементов данных. Например, агрегат Адрес может быть представлен следующим образом:

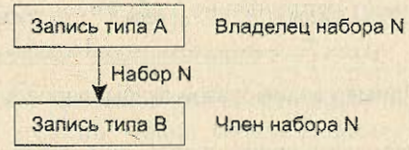
Адрес			
Город	Улица	дом	квартира

Агрегат типа повторяющаяся группа соответствует совокупности векторов данных. Например, агрегат Зарплата соответствует типу повторяющаяся группа с числом повторений 12.

Зарплата	
Месяц	Сумма

Записью называется совокупность агрегатов или элементов данных, моделирующая некоторый класс объектов реального мира. Понятие записи соответствует понятию «сегмент» в иерархической модели. Для записи, так же как и для сегмента, вводятся понятия типа записи и экземпляра записи.

Следующим базовым понятием в сетевой модели является понятие «Набор». Набором называется двухуровневый граф, связывающий отношением «один-ко-многим» два типа записи.



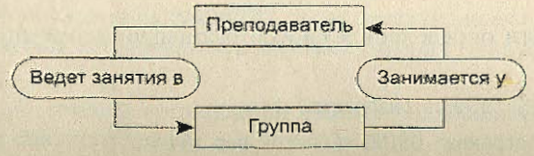
Набор фактически отражает иерархическую связь между двумя типами записей. Родительский тип записи в данном наборе называется владельцем набора, а дочерний тип записи — членом того же набора.

Для любых двух типов записей может быть задано любое количество наборов, которые их связывают. Фактически наличие подобных возможностей позволяет промоделировать отношение «многие-ко-многим» между двумя объектами реального мира, что выгодно отличает сетевую модель от иерархической. В рамках набора возможен последовательный просмотр экземпляров членов набора, связанных с одним экземпляром владельца набора.

Между двумя типами записей может быть определено любое количество наборов: например, можно построить два взаимосвязанных набора. Существенным ограничением набора является то, что один и тот же тип записи не может быть одновременно владельцем и членом набора.

В качестве примера рассмотрим таблицу, на основе которой организуем два набора и определим связь между ними:

Преподаватель	Группа	День недели	№ пары	Аудитория	Дисциплина
Иванов	4306	Понедельник	1	22-13	КИД
Иванов	4307	Понедельник	2	22-13	КИД
Карпова	4307	Вторник	2	22-14	БЗ и ЭС
Карпова	4309	Вторник	4	22-14	БЗ и ЭС
Карпова	84305	Вторник	1	22-14	БД
Смирнов	4306	Вторник	3	23-07	ГВП
Смирнов	4309	Вторник	4	23-07	ГВП



Экземпляров набора Ведет занятия будет 3 (по числу преподавателей), экземпляров набора Занимается у будет 4 (по числу групп). На рис. 3.6 представлены взаимосвязи экземпляров данных наборов.

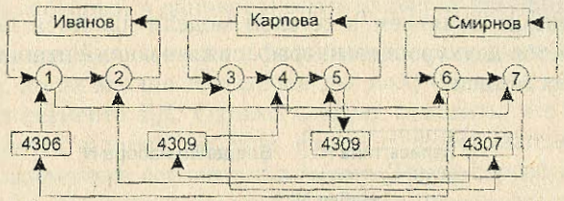


Рис. 3.6. Пример взаимосвязи экземпляров двух наборов

Среди всех наборов выделяют специальный тип набора, называемый «Сингулярным набором», владельцем которого формально определена вся система. Сингулярный набор изображается в виде входящей стрелки, которая имеет собственно имя набора и имя члена набора, но у которой не определен тип записи «Владелец набора». Например, сингулярный набор М.

Сингулярные наборы позволяют обеспечить доступ к экземплярам отдельных типов данных, поэтому если в задаче алгоритм обработки информации предполагает обеспечение произвольного доступа к некоторому типу записи, то для поддержки этой возможности необходимо ввести соответствующий сингулярный набор.

В общем случае сетевая база данных представляет совокупность взаимосвязанных наборов, которые образуют на концептуальном уровне некоторый граф.

Язык описания данных в сетевой модели

Язык описания данных в сетевой модели имеет несколько разделов:

- описание базы данных — области размещения;
- описания записей — элементов и агрегатов (каждого в отдельности);
- описания наборов (каждого в отдельности).

```
SCHEMA IS <Имя БД>.
AREA NAME IS <Имя физической области>.
RECORD NAME IS <Имя записи (уникальное)>
```

Для каждой записи определяется способ размещения экземпляров записи данного типа:

```
LOCATION MODE IS {DIRECT (напрямую) |
   CALC <Имя программы> USING <[Список пер.>]
```



```
   DUPLICATE ARE [NOT] ALLOWED
   VIA <Имя набора> SET (рядом с записями владельца)
   SYSTEM (решать будет система)}
```

Каждый тип записи должен быть приписан к некоторой физической области размещения:

```
WITHIN <Имя области размещения> AREA
```

После описания записи в целом идет описание внутренней структуры:

```
<Имя уровня> <Имя данного> <Шаблон> <Тип>
```

Номер уровня определяет уровень вложенности при описании элементов и агрегатов данных. Первый уровень — сама запись. Поэтому элементы или агрегаты данных имеют уровень начиная со второго. Если данное соответствует агрегату, то любая его составляющая добавляет один уровень вложенности.

Если агрегат является вектором, то он описывается как

```
<Номер уровня> <Имя агрегата>.<Номер уровня> <Имя 1-й сост.>
```

а если — повторяющейся группой, то следующим образом:

```
<Номер уровня> <Имя агрегата>.OCCURS <N> TIMES
```

где N — среднее количество элементов в группе.

Описание набора и порядка включения членов в него выглядит следующим образом:

```
SET NAME IS <Имя набора>;
OWNER IS (<Имя владельца> | SYSTEM).
```

Далее указывается порядок включения новых экземпляров члена данного набора в экземпляр набора:

```
ORDER PERMANENT INSERTION IS {SORTED | NEXT | PREV | LAST | FIRST}
```

После этого описывается член набора с указанием способа включения и способа исключения экземпляра — члена набора из экземпляра набора.

```
MEMBER IS <Имя члена набора> {AUTOMATIC | MANUAL} {MANDATORY | OPTIONAL} KEY
IS (ACCENDING | DESCENDING) <Имя элемента данных>
```

При автоматическом включении каждый новый экземпляр члена набора автоматически попадает в текущий экземпляр набора в соответствии с заданным ранее порядком включения. При ручном способе экземпляр члена набора сначала попадает в БД, а только потом командой CONNECT может быть включен в конкретный экземпляр набора.

Если задан способ исключения MANDATORY, то экземпляр записи, исключаемый из набора, автоматически исключается и из базы данных. Иначе просто разрываются связи.

Внешняя модель при сетевой организации данных поддерживается путем описания части общего связного графа.

Язык манипулирования данными в сетевой модели

Все операции манипулирования данными в сетевой модели делятся на *навигационные операции* и *операции модификации*.

Навигационные операции осуществляют перемещение по БД путем прохождения по связям, которые поддерживаются в схеме БД. В этом случае результатом является новый единичный объект, который получает статус *текущего объекта*.

Операции модификации осуществляют как добавление новых экземпляров отдельных типов записей, так и экземпляров новых наборов, удаление экземпляров записей и наборов, модификацию отдельных составляющих внутри конкретных экземпляров записей. Средства модификации данных сведены в табл. 3.1:

Таблица 3.1. Операторы манипулирования данными в сетевой модели

Операция	Назначение
READY	Обеспечение доступа данного процесса или пользователя к БД (сходна по смыслу с операцией открытия файла)
FINISH	Окончание работы с БД
FIND	Группа операций, устанавливающих указатель найденного объекта на текущий объект
GET	Передача найденного объекта в рабочую область. Допустима только после FIND
STORE	Помещение в БД записи, сформированной в рабочей области
CONNECT	Включение текущей записи в текущий экземпляр набора
DISCONNECT	Исключение текущей записи из текущего экземпляра набора
MODIFY	Обновление текущей записи данными из рабочей области пользователя
ERASE	Удаление экземпляра текущей записи

В рабочей области пользователя хранятся шаблоны записей, программные переменные и три типа указателей текущего состояния:

- текущая запись процесса (код или ключ последней записи, с которой работала данная программа);
- текущая запись типа записи (для каждого типа записи ключ последней записи, с которой работала программа);
- текущая запись типа набор (для каждого набора с владельцем T1 и членом T2 указывается, T1 или T2 были последней обрабатываемой записью).

На рис. 3.7 представлена концептуальная модель торгово-посреднической организации.

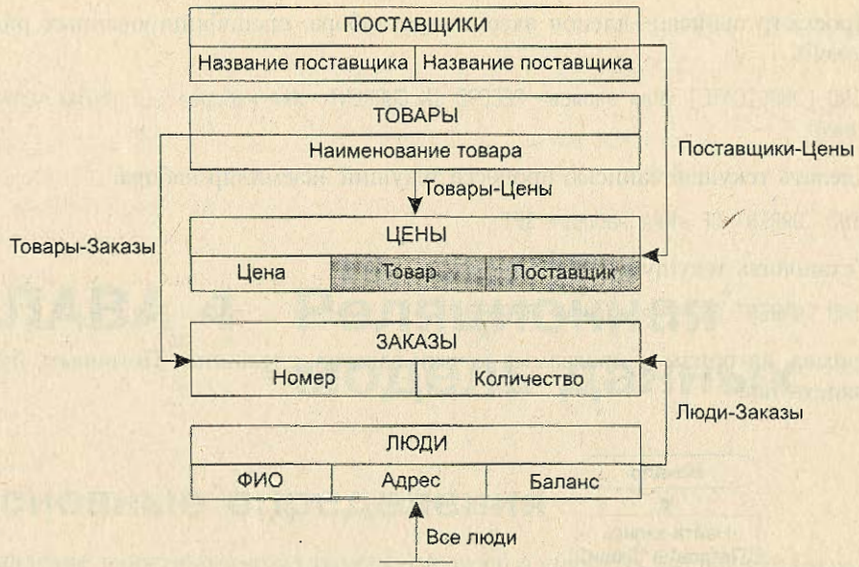


Рис. 3.7. Схема БД «Торговая фирма»

При необходимости возможно описание элементов данных, которые не принадлежат непосредственно данной записи, но при ее обработке часто используются. Для этого используется тип VIRTUAL с обязательным указанием источника данного элемента данных.

```
RECORD Цены
02 Цена TYPE REAL
02 Товар VIRTUAL
SOURCE IS Товары.НаименованиеТовара
OF OWNER OF Товар-Цены SET
```

Наиболее интересна операция поиска (FIND), так как именно она отражает суть навигационных методов, применяемых в сетевой модели. Всего существует семь типов операций поиска:

1. По ключу (запись должна быть описана через CALC USING ...):

   ```
   FIND <Имя записи> RECORD BY CALC KEY <Имя параметра>
   ```

2. Последовательный просмотр записей данного типа:

   ```
   FIND DUPLICATE <Имя записи> RECORD BY CALC KEY
   ```

3. Найти владельца текущего экземпляра набора:

   ```
   FIND OWNER OF CURRENT <Имя набора> SET
   ```

4. Последовательный просмотр записей—членов текущего экземпляра набора:

   ```
   FIND (FIRST | NEXT) <Имя записи> RECORD IN CURRENT <Имя набора> SET
   ```

5. Просмотр записей—членов экземпляра набора, специфицированных рядом полей:

   FIND [DUPLICATE] <Имя записи> RECORD IN CURRENT <Имя набора> SET USING <Список полей>

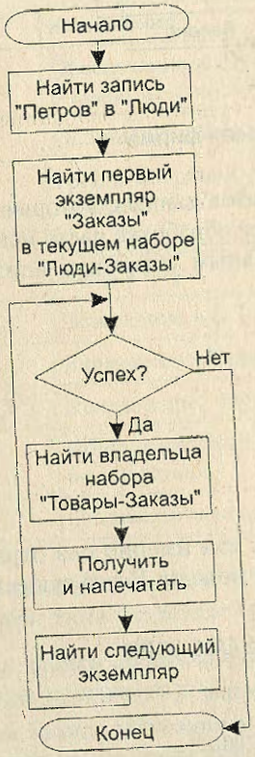
6. Сделать текущей записью процесса текущий экземпляр набора:

   FIND CURRENT OF <Имя набора> SET

7. Установить текущую запись процесса:

   FIND CURRENT OF <Имя записи> RECORD

Например, алгоритм и программа печати заказов, сделанных Петровым, будут выглядеть так:




```
ФИО = "Петров"
FIND Люди RECORD BY CALC KEY
FIND FIRST Заказы RECORD IN
           CURRENT Люди-Заказы SET
WHILE NOT FAIL DO
    FIND OWNER OF CURRENT
                  Товары-Заказы SET
    GET Товары
    PRINT НаимТовара
    FIND NEXT Заказы RECORD IN
              CURRENT Люди-Заказы SET
END
```

ГЛАВА 4 Реляционная модель данных

Основные определения

Появление теоретико-множественных моделей в системах баз данных было предопределено настоятельной потребностью пользователей в переходе от работы с элементами данных, как это делается в графовых моделях, к работе с некоторыми макрообъектами. Основной моделью в этом классе является реляционная модель данных. Простота и наглядность модели для пользователей-непрограммистов, с одной стороны, и серьезное теоретическое обоснование, с другой стороны, определили большую популярность этой модели. Кроме того, развитие формального аппарата представления и манипулирования данными в рамках реляционной модели сделали ее наиболее перспективной для использования в системах представления знаний, что обеспечивает качественно иной подход к обработке данных в больших информационных системах.

Теоретической основой этой модели стала теория отношений, основу которой заложили два логика – американец Чарльз Содерс Пирс (1839–1914) и немец Эрнст Шредер (1841–1902). В руководствах по теории отношений было показано, что множество отношений замкнуто относительно некоторых специальных операций, то есть образует вместе с этими операциями абстрактную алгебру. Это важнейшее свойство отношений было использовано в реляционной модели для разработки языка манипулирования данными, связанного с исходной алгеброй. Американский математик Э. Ф. Кодд в 1970 году впервые сформулировал основные понятия и ограничения реляционной модели, ограничив набор операций в ней семью основными и одной дополнительной операцией. Предложения Кодда были настолько эффективны для систем баз данных, что за эту модель он был удостоен престижной премии Тьюринга в области теоретических основ вычислительной техники.

Основной структурой данных в модели является отношение, именно поэтому модель получила название *реляционной* (от английского *relation* – отношение).

N-арным отношением R называют подмножество декартова произведения $D_1 \times D_2 \times \ldots \times D_n$ множеств $D_1, D_2, \ldots, D_n$ ($n \geq 1$), необязательно различных. Исходные множества $D_1, D_2, \ldots, D_n$ называют в модели *доменами*.

$R \subseteq D_1 \times D_2 \times ... \times D_n$,

где $D_1 \times D_2 \times ... \times D_n$ — полное декартово произведение.

Полное декартово произведение — это набор всевозможных сочетаний из n элементов каждое, где каждый элемент берется из своего домена. Например, имеем три домена: D_1 содержит три фамилии, D_2 — набор из двух учебных дисциплин и D_3 — набор из трех оценок. Допустим, содержимое доменов следующее:

- D_1 = {Иванов, Крылов, Степанов};
- D_2 = {Теория автоматов, Базы данных};
- D_3 = {3, 4, 5}

Тогда полное декартово произведение содержит набор из 18 троек, где первый элемент — это одна из фамилий, второй — это название одной из учебных дисциплин, а третий — одна из оценок.

<Иванов,Теория автоматов,3>; <Иванов,Теория автоматов,4>; <Иванов,Теория автоматов,5>
<Крылов,Теория автоматов,3>; <Крылов,Теория автоматов,4>; <Крылов,Теория автоматов,5>;
<Степанов,Теория автоматов,3>; <Степанов,Теория автоматов,4>; <Степанов,Теория автоматов,5>;
<Иванов,Базы данных,3>; <Иванов,Базы данных,4>; <Иванов,Базы данных,5>;
<Крылов,Базы данных,3>; <Крылов,Базы данных,4>; <Крылов,Базы данных,5>;
<Степанов,Базы данных,3>; <Степанов,Базы данных,4>; <Степанов,Базы данных,5>;

Отношение R моделирует реальную ситуацию и оно может содержать, допустим, только 5 строк, которые соответствуют результатам сессии (Крылов экзамен по «Базам данных» еще не сдавал):

<Иванов,Теория автоматов,4>; <Крылов,Теория автоматов,5>; <Степанов,Теория автоматов,5>;
<Иванов,Базы данных,3>; <Степанов,Базы данных,4>;

Отношение имеет простую графическую интерпретацию, оно может быть представлено в виде таблицы, столбцы которой соответствуют вхождениям доменов в отношение, а строки — наборам из n значений, взятых из исходных доменов, которые расположены в строго определенном порядке в соответствии с заголовком. Такие наборы из n значений часто называют n-ками.

R		
Фамилия	**Дисциплина**	**Оценка**
Иванов	Теория автоматов	4
Иванов	Базы данных	3
Крылов	Теория автоматов	5
Степанов	Теория автоматов	5
Степанов	Базы данных	4

Данная таблица обладает рядом специфических свойств:

1. В таблице нет двух одинаковых строк.
2. Таблица имеет столбцы, соответствующие атрибутам отношения.
3. Каждый атрибут в отношении имеет уникальное имя.
4. Порядок строк в таблице произвольный.

Вхождение домена в отношение принято называть *атрибутом*. Строки отношения называются *кортежами*.

Количество атрибутов в отношении называется степенью, или рангом, отношения.

Следует заметить, что в отношении не может быть одинаковых кортежей, это следует из математической модели: отношение — это подмножество декартова произведения, а в декартовом произведении все *n*-ки различны.

В соответствии со свойствами отношений два отношения, отличающиеся только порядком строк или порядком столбцов, будут интерпретироваться в рамках реляционной модели как одинаковые, то есть отношение R и отношение R_1, изображенное далее, одинаковы с точки зрения реляционной модели данных.

R1		
Дисциплина	**Фамилия**	**Оценка**
Теория автоматов	Крылов	5
Теория автоматов	Степанов	5
Теория автоматов	Иванов	4
Базы данных	Иванов	3
Базы данных	Степанов	4

Любое отношение является динамической моделью некоторого реального объекта внешнего мира. Поэтому вводится понятие экземпляра отношения, которое отражает состояние данного объекта в текущий момент времени, и понятие схемы отношения, которая определяет структуру отношения.

Схемой отношения R называется перечень имен атрибутов данного отношения с указанием домена, к которому они относятся:

$S_R = (A_1, A_2, A_n)$, $A_i \subseteq D_i$.

Если атрибуты принимают значения из одного и того же домена, то они называются θ-*сравнимыми*, где θ — множество допустимых операций сравнения, заданных для данного домена. Например, если домен содержит числовые данные , то для него допустимы все операции сравнения, тогда θ = {=, <>,>=,<=,<,>}. Однако и для доменов, содержащих символьные данные, могут быть заданы не только операции сравнения по равенству и неравенству значений. Если для данного домена задано лексикографическое упорядочение, то он имеет также полный спектр операций сравнения.

Схемы двух отношений называются *эквивалентными*, если они имеют одинаковую степень и возможно такое упорядочение имен атрибутов в схемах, что на одинаковых местах будут находиться сравнимые атрибуты, то есть атрибуты, принимающие значения из одного домена.

$S_{R1} = (A_1, A_2, ..., A_n)$ — схема отношения R_1.

$S_{R2} = (B_{i1}, B_{i2}, ..., B_{in})$ — схема отношения R_2 после упорядочения имен атрибутов.

Тогда

$$S_{R1} \sim S_{R2} \Leftrightarrow \begin{cases} 1. & n = m, \\ 2. & A_j, B_{ij} \subseteq D_j. \end{cases}$$

Как уже говорилось ранее, реляционная модель представляет базу данных в виде множества взаимосвязанных отношений. В отличие от теоретико-графовых моделей в реляционной модели связи между отношениями поддерживаются неявным образом. Какие же связи между отношениями поддерживаются в реляционной модели? В этой модели, так же как и в остальных, поддерживаются иерархические связи между отношениями. В каждой связи одно отношение может выступать как основное, а другое отношение выступает в роли подчиненного. Это означает, что один кортеж основного отношения может быть связан с несколькими кортежами подчиненного отношения. Для поддержки этих связей оба отношения должны содержать наборы атрибутов, по которым они связаны. В основном отношении это первичный ключ отношения (PRIMARY KEY), который однозначно определяет кортеж основного отношения. В подчиненном отношении для моделирования связи должен присутствовать набор атрибутов, соответствующий первичному ключу основного отношения. Однако здесь этот набор атрибутов уже является вторичным ключом, то есть он определяет множество кортежей подчиненного отношения, которые связаны с единственным кортежем основного отношения. Данный набор атрибутов в подчиненном отношении принято называть внешним ключом (FOREIGN KEY).

Например, рассмотрим ситуацию, когда надо описать карьеру некоторого индивидуума. Каждый человек в своей трудовой деятельности сменяет несколько мест работы в разных организациях, где он работает в разных должностях. Тогда мы должны создать два отношения: одно для моделирования всех работающих людей, а другое для моделирования записей в их трудовых книжках, если для нас важно не только отследить переход работника из одной организации в другую, но и прохождение его по служебной лестнице в рамках одной организации (рис. 4.1).

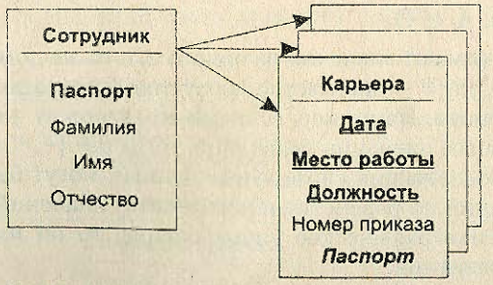


Рис. 4.1. Связь между основным и подчиненным отношениями

PRIMARY KEY отношения Сотрудник атрибут Паспорт является FOREIGN KEY для отношения «карьера».

Операции над отношениями. Реляционная алгебра

Напомним, что *алгеброй* называется множество объектов с заданной на нем совокупностью операций, замкнутых относительно этого множества, называемого *основным множеством*.

Основным множеством в реляционной алгебре является множество отношений. Всего Э. Ф. Коддом было предложено 8 операций. В общем это множество избыточное, так как одни операции могут быть представлены через другие, однако множество операций выбрано из соображений максимального удобства при реализации произвольных запросов к БД. Все множество операций можно разделить на две группы: теоретико-множественные операции и специальные операции. В первую группу входят 4 операции. Три первые теоретико-множественные операции являются бинарными, то есть в них участвуют два отношения и они требуют эквивалентных схем исходных отношений.

Теоретико-множественные операции реляционной алгебры

Объединением двух отношений называется отношение, содержащее множество кортежей, принадлежащих либо первому, либо второму исходным отношениям, либо обоим отношениям одновременно.

Пусть заданы два отношения $R_1 = \{ r_1 \}$, $R_2 = \{ r_2 \}$, где r_1 и r_2 — соответственно кортежи отношений R_1 и R_2, то объединение

$$R_1 \cup R_2 = \{ r \mid r \in R_1 \vee r \in R_2 \}.$$

Здесь r — кортеж нового отношения, $\vee$ — операция логического сложения «ИЛИ».

Пример применения операции объединения приведен на рис. 4.1. Исходными отношениями являются отношения R_1 и R_2, которые содержат перечни деталей, изготавливаемых соответственно на первом и втором участках цеха. Отношение R_3 содержит общий перечень деталей, изготавливаемых в цеху, то есть характеризует общую номенклатуру цеха.

R1	
Шифр детали	**Название детали**
00011073	Гайка М1
00011075	Гайка М2
00011076	Гайка М3
00011003	Болт М1
00011006	Болт М3
00013063	Шайба М1
00013066	Шайба М3

R2	
Шифр детали	**Название детали**
00011073	Гайка М1
00011076	Гайка М3
00011077	Гайка М4
00011004	Болт М2
00011006	Болт М3

R3	
Шифр детали	**Название детали**
00011073	Гайка М1
00011075	Гайка М2
00011076	Гайка М3
00011003	Болт М1
00011006	Болт М3
00013063	Шайба М1
00013066	Шайба М3
00011077	Гайка М4
00011004	Болт М2

Пересечением отношений называется отношение, которое содержит множество кортежей, принадлежащих одновременно и первому и второму отношениям. R_1 и R_2:

$$R_3 = R_1 \cap R_2 = \{ r \mid r \in R_1 \wedge r \in R_2 \}$$

здесь $\wedge$ — операция логического умножения (логическое «И»).

В отношении R_4 содержатся перечень деталей, которые выпускаются одновременно на двух участках цеха.

R4	
Шифр детали	**Название детали**
00011073	Гайка М1
00011076	Гайка М3
00011006	Болт М3

Разностью отношений R_1 и R_2 называется отношение, содержащее множество кортежей, принадлежащих R_1 и не принадлежащих R_2:

$$R_5 = R_1 \setminus R_2 = \{ r \mid r \in R_1 \wedge r \notin R_2 \}$$

Отношение R_5 содержит перечень деталей, изготавливаемых только на участке 1, отношение R_6 содержит перечень деталей, изготавливаемых только на участке 2.

$$R_6 = R_2 \setminus R_1 = \{ r \mid r \in R_2 \wedge r \notin R_1 \}$$

R5	
Шифр детали	**Название детали**
00011075	Гайка М2
00011003	Болт М1
00013063	Шайба М1
00013066	Шайба М3

R6	
Шифр детали	**Название детали**
00011077	Гайка М4
00011004	Болт М2

Следует отметить, что первые две операции, объединение и пересечение, являются коммутативными операциями, то есть результат операции не зависит от порядка аргументов в операции. Операция же разности является принципиально несимметричной операцией, то есть результат операции будет различным для разного порядка аргументов, что и видно из сравнения отношений R_5 и R_6.

В отличие от навигационных средств манипулирования данными в теоретико-графовых моделях операции реляционной алгебры позволяют получить сразу иной качественный результат, который является семантически гораздо более ценным и понятным пользователям. Например, сравнение результатов объединения и разности номенклатуры двух участков позволит оценить специфику производства: насколько оно уникально на каждом участке, и, в зависимости от необходимости, принять соответствующее решение по изменению номенклатуры.

Для демонстрации возможностей трех первых операций реляционной алгебры рассмотрим еще один пример — уже из другой предметной области. Исходными являются три отношения R_1, R_2 и R_3. Все они имеют эквивалентные схемы.

- R_1= (ФИО, Паспорт, Школа);
- R_2= (ФИО, Паспорт, Школа);
- R_3= (ФИО, Паспорт, Школа).

Рассмотрим ситуацию поступления в высшие учебные заведения, которая была характерна для периода, когда были разрешены так называемые репетиционные вступительные экзамены, которые сдавались раньше основных вступительных экзаменов в вуз. Отношение R_1 содержит список абитуриентов, сдававших репетиционные экзамены. Отношение R_2 содержит список абитуриентов, сдававших экзамены на общих условиях. И наконец, отношение R_3 содержит список абитуриентов, принятых в институт. Будем считать, что при неудачной сдаче репетиционных экзаменов абитуриент мог делать вторую попытку и сдавать экзамены в общем потоке, поэтому некоторые абитуриенты могут присутствовать как в первом, так и во втором отношении.

Ответим на следующие вопросы:

1. Список абитуриентов, которые поступали два раза и не поступили в вуз.
   $R = R_1 \cap R_2 \setminus R_3$
2. Список абитуриентов, которые поступили в вуз с первого раза, то есть они сдавали экзамены только один раз и сдали их так хорошо, что сразу были зачислены в вуз.
   $R = (R_1 \setminus R_2 \cap R_3) \cup (R_2 \setminus R_1 \cap R_3)$
3. Список абитуриентов, которые поступили в вуз только со второго раза.

   Прежде всего это те абитуриенты, которые присутствуют в отношениях R_1 и R_2, потому что они поступали два раза, и присутствуют в отношении R_3, потому что они поступили.
   $R = R_1 \cap R_2 \cap R_3$
4. Список абитуриентов, которые поступали только один раз и не поступили.

   Это прежде всего те абитуриенты, которые присутствуют в R_1 и не присутствуют в R_2, и те, кто присутствуют в R_2 и не присутствуют в R_1. И разумеется, никто из них не присутствует в R_3.

$$R = (R_1 \setminus R_2) \cup (R_2 \setminus R_1) \setminus R_3$$

В отсутствие скобок порядок выполнения операций реляционной алгебры естественный, поэтому сначала будут выполнены операции в скобках, а затем будет выполнена последняя операция вычитания отношения R_3.

Операции объединения, пересечения и разности применимы только к отношениям с эквивалентными схемами.

Кроме перечисленных трех теоретико-множественных операций в рамках реляционной алгебры определена еще одна теоретико-множественная операция — расширенное декартово произведение. Эта операция не накладывает никаких дополнительных условий на схемы исходных отношений, поэтому операция расширенного декартова произведения, обозначаемая $R_1 \otimes R_2$, допустима для любых двух отношений. Но прежде чем определить саму операцию, введем дополнительно понятие конкатенации, или сцепления, кортежей.

Сцеплением, или *конкатенацией*, кортежей $c = \langle c_1, c_2, ..., c_n \rangle$ и $q = \langle q_1, q_2, ..., q_m \rangle$ называется кортеж, полученный добавлением значений второго в конец первого. Сцепление кортежей c и q обозначается как (c , q).

$$(c, q) = \langle c_1, c_2, \ldots , c_n, q_1, q_2, ..., q_m \rangle$$

Здесь n — число элементов в первом кортеже c, m — число элементов во втором кортеже q.

Все предыдущие операции не меняли степени или арности отношений — это следует из определения эквивалентности схем отношений. Операция декартова произведения меняет степень результирующего отношения.

Расширенным декартовым произведением отношения R_1 степени n со схемой

$$S_{R1} = (A_1, A_2, \ldots , A_n)$$

и отношения R_2 степени m со схемой

$$S_{R2} = (B_1, B_2, \ldots , B_m)$$

называется отношение R_3 степени n+m со схемой

$$S_{R3} = (A_1, A_2, \ldots , A_n, B_1, B_2, ..., B_m),$$

содержащее кортежи, полученные сцеплением каждого кортежа r отношения R_1 с каждым кортежем q отношения R_2.

То есть если $R_1 = \{ r \}$, $R_2 = \{ q \}$

$$R_1 \otimes R_2 = \{(r, q) \mid r \in R_1 \wedge q \in R_2\}$$

Операцию декартова произведения с учетом возможности перестановки атрибутов в отношении можно считать симметричной. Очень часто операция расширенного декартова произведения используется для получения некоторого универсума — т. е. отношения, которое характеризует все возможные комбинации между элементами отдельных множеств. Однако самостоятельного значения результат выполнения операции обычно не имеет, он участвует в дальнейшей обработке. Например, на производстве в отношении 07 задана обязательная номенклатура деталей для всех цехов, а в отношении 08 дан перечень всех цехов.

R7

Шифр детали	Название детали
00011073	Гайка М1
00011075	Гайка М2
00011076	Гайка М3
00011003	Болт М1
00011006	Болт М3
00013063	Шайба М1
00013066	Шайба М3
00011077	Гайка М4
00011004	Болт М2
00011005	Болт М5
00011006	Болт М6
00013062	Шайба М2

R8

Цех
Цех 1
Цех 2
Цех 3

Тогда отношение R_9, которое соответствует ситуации, когда каждый цех изготавливает все требуемые детали, будет выглядеть следующим образом:

R9

Шифр детали	Название детали	Цех
00011073	Гайка М1	Цех 1
00011075	Гайка М2	Цех 1
00011076	Гайка М3	Цех 1
00011003	Болт М1	Цех 1
00011006	Болт М3	Цех 1
00013063	Шайба М1	Цех 1
00013066	Шайба М3	Цех 1
00011077	Гайка М4	Цех 1
00011004	Болт М2	Цех 1
00011005	Болт М5	Цех 1
00011006	Болт М6	Цех 1
00013062	Шайба М2	Цех 1
00011073	Гайка М1	Цех 2
00011075	Гайка М2	Цех 2

R10

Шифр детали	Название детали	Цех
00011073	Гайка М1	Цех 1
00011075	Гайка М2	Цех 1
00011076	Гайка М3	Цех 1
00011003	Болт М1	Цех 1
00011006	Болт М3	Цех 1
00013063	Шайба М1	Цех 1
00013066	Шайба М3	Цех 1
00011077	Гайка М4	Цех 1
00011004	Болт М2	Цех 1
00011006	Болт М3	Цех 2
00013063	Шайба М1	Цех 2
00013066	Шайба М3	Цех 2
00011077	Гайка М4	Цех 2
00011004	Болт М2	Цех 2

R9 (*продолжение*)		
00011076	Гайка М3	Цех 2
00011003	Болт М1	Цех 2
00011006	Болт М3	Цех 2
00013063	Шайба М1	Цех 2
00013066	Шайба М3	Цех 2
00011077	Гайка М4	Цех 2
00011004	Болт М2	Цех 2
00011005	Болт М5	Цех 2
00011006	Болт М6	Цех 2
00013062	Шайба М2	Цех 2
00011073	Гайка М1	Цех 3
00011075	Гайка М2	Цех 3
00011076	Гайка М3	Цех 3
00011003	Болт М1	Цех 3
00011006	Болт М3	Цех 3
00013063	Шайба М1	Цех 3
00013066	Шайба М3	Цех 3
00011077	Гайка М4	Цех 3
00011004	Болт М2	Цех 3
00011005	Болт М5	Цех 3
00011006	Болт М6	Цех 3
00013062	Шайба М2	Цех 3

R10 (*продолжение*)		
00011006	Болт М6	Цех 2
00013062	Шайба М2	Цех 2
00011073	Гайка М1	Цех 3
00011075	Гайка М2	Цех 3
00011076	Гайка М3	Цех 3
00011003	Болт М1	Цех 3
00011006	Болт М3	Цех 3
00013063	Шайба М1	Цех 3
00013066	Шайба М3	Цех 3
00011077	Гайка М4	Цех 3
00011005	Болт М5	Цех 3
00011006	Болт М6	Цех 3
00011005	Болт М5	Цех 1
00011006	Болт М6	Цех 1
00013062	Шайба М2	Цех 1

В каких запросах нужно использовать расширенное декартово произведение? Эта операция моделирует некоторую ситуацию, которая характеризуется словом «все». Поэтому если нам надо узнать, какие детали в каких цехах из общей обязательной номенклатуры не выпускаются, то мы можем вычесть из полученного отношения R_9 отношение R_{10}, характеризующее реальный выпуск деталей в каждом цехе.

Отношение R_{11}, которое является результатом выполнения этой операции, имеет вид:

$$R_{11} = R_9 \setminus R_{10}$$

R11		
Шифр детали	**Название детали**	**Цех**
00011073	Гайка М1	Цех 2
00011075	Гайка М2	Цех 2
00011076	Гайка М3	Цех 2
00011004	Болт М2	Цех 3
00013062	Шайба М2	Цех 3
00011003	Болт М1	Цех 2
00011005	Болт М5	Цех 3

Группа теоретико-множественных операций избыточна, так, например, операцию можно заменить сочетанием операций объединения и пересечения.

$$(R_1 \cup R_2) \setminus (R_1 \setminus R_2) \setminus (R_2 \setminus R_1)$$

Однако это достаточно сложная формула, и именно поэтому все три теоретико-множественные операции вошли в базовый набор операций реляционной алгебры.

Далее мы переходим к группе операций, названных специальными операциями реляционной алгебры.

Специальные операции реляционной алгебры

Первой специальной операцией реляционной алгебры является *горизонтальный выбор*, или *операция фильтрации*, или *операция ограничения отношений*. Для определения этой операции нам необходимо ввести дополнительные обозначения.

Пусть α — булевское выражение, составленное из термов сравнения с помощью связок И ($\wedge$), ИЛИ ($\vee$), НЕ (-) и, возможно, скобок. В качестве термов сравнения допускаются:

а) терм A ос a,

где A — имя некоторого атрибута, принимающего значения из домена D; a — константа, взятая из того же домена D, $a \in D$; ос — одна из допустимых для данного домена D операций сравнения;

б) терм A ос B,

где A, B — имена некоторых θ-сравнимых атрибутов, то есть атрибутов, принимающих значения из одного и то же домена D.

Тогда результатом операции выбора, или фильтрации, заданной на отношении R в виде булевского выражения, определенного на атрибутах отношения R, называется отношение $R[\alpha]$, включающее те кортежи из исходного отношения, для которых истинно условие выбора или фильтрации:

$$R[\alpha(r)] = \{r \mid r \in R \wedge \alpha(r) = \text{"Истина"}\}$$

Операция фильтрации является одной из основных при работе с реляционной моделью данных. Условие α может быть сколь угодно сложным.

Например, выбрать из R_{10} детали с шифром «0011003».

$R_{12} = R_{10} [\text{Шифр детали} = «0011003»]$

R12		
Шифр детали	**Название детали**	**Цех**
00011003	Болт М1	Цех 1
00011003	Болт М1	Цех 3

Следующей специальной операцией является *операция* проектирования.

Пусть R — отношение, $S_R = (A_1, \ldots, A_n)$ — схема отношения R.

Обозначим через B подмножество $[A_i]$; $B \subseteq \{ A_i \}$.

При этом пусть B^1 — множество атрибутов из $\{ A_i \}$, не вошедших в B.

Если $B = \{A_i^1, A_i^2, \ldots, A_i^k\}$, $B^1 = \{A_j^1, A_j^2, \ldots, A_j^k\}$ и $r = < a_i^1, a_i^2, \ldots, a_i^k >$, $a_i^k \in A_i^k$,

то $r[B]$, $s = < a_j^1, a_j^2, \ldots, a_j^m > ; a_j^m \in A_j^m$.

Проекцией отношения R на набор атрибутов B, обозначаемой R[B], называется отношение со схемой, соответствующей набору атрибутов B $S_{R[B]} = B$, содержащему кортежи, получаемые из кортежей исходного отношения R путем удаления из них значений, не принадлежащих атрибутам из набора B.

$R[B] = \{ r[B] \}$

По определению отношений все дублирующие кортежи удаляются из результирующего отношения.

Операция проектирования, называемая иногда также операцией вертикального выбора, позволяет получить только требуемые характеристики моделируемого объекта. Чаще всего операция проектирования употребляется как промежуточный шаг в операциях горизонтального выбора, или фильтрации. Кроме того, она используется самостоятельно на заключительном этапе получения ответа на запрос.

Например, выберем все цеха, которые изготавливают деталь «Болт М1».

Для этого нам необходимо из отношения R_{10} выбрать детали с заданным названием, а потом полученное отношение спроектировать на столбец «Цех». Результатом выполнения этих операций будет отношение R_{14}:

$R_{13} = R_{10} [\text{Название детали} = «\text{Болт М1}»]$

$R_{14} = R_{13} [\text{Цех}]$

R13		
Шифр детали	**Название детали**	**Цех**
00011003	Болт М1	Цех 1
00011003	Болт М1	Цех 3

R14
Цех
Цех 1
Цех 3

Следующей специальной операцией реляционной алгебры является операция *условного соединения*.

В отличие от рассмотренных специальных операций реляционной алгебры: фильтрации и проектирования, которые являются унарными, то есть производятся над одним отношением, операция условного соединения является бинарной, то есть исходными для нее являются два отношения, а результатом — одно.

Пусть $R = \{ r \}$, $Q = \{ q \}$ — исходные отношения,

S_R, S_Q — схемы отношений R и Q соответственно,

$S_R = (A_1, A_2, \ldots, A_k)$; $S_Q = (B_1, B_2, \ldots, B_m)$,

где A_i, B_j — имена атрибутов в схемах отношений R и Q соответственно.

При этом полагаем, что заданы наборы атрибутов A и B

$A \subseteq \{ A_i \}_{i=1,k}$; $B \subseteq \{ B_j \}_{j=1,m}$

и эти наборы состоят из θ-сравнимых атрибутов.

Тогда соединением отношений R и Q при условии β будет подмножество декартова произведения отношений R и Q, кортежи которого удовлетворяют условию β, рассматриваемому как одновременное выполнение условий:

- $r.A_i\ \theta_i\ B_i : i=1,k$, где k — число атрибутов, входящих в наборы A и B, а θ_i — конкретная операция сравнения.
- $A_i\ \theta_i\ B_i\ D$; θ_i — i-й предикат сравнения, определяемый из множества допустимых на домене D_i операций сравнения.

$R [\beta] Q = \{ (r,q) \mid (r, q) \mid r.A\ \theta_i\ q.B_i = «\text{Истина}», i=1,k \}$

Например, рассмотрим следующий запрос. Пусть отношение R_{15} содержит перечень деталей с указанием материалов, из которых эти детали изготавливаются, и оно имеет вид:

R15		
Шифр детали	**Название детали**	**Материал**
00011073	Гайка М1	сталь-ст1
00011075	Гайка М2	сталь-ст2
00011076	Гайка М3	сталь-ст1
00011003	Болт М1	сталь-ст3
00011006	Болт М3	сталь-ст3
00013063	Шайба М1	сталь-ст1
00013066	Шайба М3	сталь-ст1
00011077	Гайка М4	сталь-ст2
00011004	Болт М2	сталь-ст3
00011005	Болт М5	сталь-ст3
00013062	Шайба М2	сталь-ст1

R16
Название детали
Гайка М1
Гайка М3
Шайба М1
Шайба М3
Шайба М2

Получим перечень деталей, которые изготавливаются в цеху 1 из материала «сталь-ст1»

R_{16} = (R_{15}[(R_{15}.Шифр детали =R_{10}.Шифр детали) Λ R_{10}.Цех = «Цех1» Λ Λ R_{15}.Материал =«сталь-ст1»] R_{10})[Название детали]

Последней операцией, включаемой в набор операций реляционной алгебры, является операция *деления*.

Для определения операции деления рассмотрим сначала понятие множества образов.

Пусть R — отношение со схемой $S_R = (A_1, A_2, ..., A_k)$;

Пусть A — некоторый набор атрибутов $A \subset \{ A_i \}$ i=1,k , A^1 — набор атрибутов, не входящих в множество A.

Пересечение множеств A и A^1 пусто: $A \cap A^1 = \varnothing$; объединение множеств равно множеству всех атрибутов исходного отношения: $A \cup A^1 = S_R$.

Тогда множеством образов элемента y проекции R[A] называется множество таких элементов y проекции $R[A^1]$, для которых сцепление (x, y) является кортежами отношения R, то есть

$QA(x) = \{y \mid y \in R[A^1] \Lambda (x, y) \in R\}$ — множество образов.

Например, множеством образов отношения R_{15} по материалу «сталь-ст2» будет множество кортежей

R_{15}.Материал = {< 00011075, Гайка М2, «сталь-ст2»>,
< 00011077, Гайка М4, «сталь-ст2»>}

Дадим теперь определение операции *деления*.

Пусть даны два отношения R и T соответственно со схемами:

$S_R = (A_1, A_2, ..., A_k)$; $S_T = (B_1, B_2, ..., B_m)$;

A и B — наборы атрибутов этих отношений, одинаковой длины (без повторений);

$A \subset S_R$; $B \subset S_T$.

Атрибуты A^1 — это атрибуты из R, не вошедшие в множество A.

Пересечение множеств $A \cap A^1 = \varnothing$ — пусто и $A \cup A^1 = S^R$. Проекции R[A] и T[B] совместимы по объединению, то есть имеют эквивалентные схемы: $S_{R[A]} \sim S_{T[B]}$.

Тогда операция деления ставит в соответствие отношениям R и T отношение Q = R[A:B]T, кортежи которого являются теми элементами проекции $R[A^1]$, для которых T[B] входит в построенные для них множество образов:

$R[A:B]T = \{r \mid r \in R[A^1] \Lambda T[B] \subseteq \{y \mid y \in R[A] \Lambda (r, y) \in R\}\}$.

Операция деления удобна тогда, когда требуется сравнить некоторое множество характеристик отдельных атрибутов. Например, пусть у нас есть отношение R_7, которое содержит номенклатуру всех выпускаемых деталей на нашем предприятии, а в отношении R_{10} хранятся сведения о том, что и в каких цехах действительно выпускается. Поставим задачу определить перечень цехов, в которых выпускается вся номенклатура деталей.

Тогда решением этой задачи будет операция деления отношения R_{10} на отношение R_7 по набору атрибутов (Шифр детали, Наименование детали).

R_{17} = R_{10}[Шифр детали, Наименование детали: Шифр детали, Наименование детали]R_7

R 17
Цех
Цех1

Операция деления достаточно сложна для абстрактного представления. Она может быть заменена последовательностью других операций. Действительно, выполним тот же запрос с использованием других операций. Для этого определим последовательность промежуточных запросов, которая приведет нас к конечному результату:

1. Построим отношение, которое моделирует ситуацию, когда в каждом цеху изготавливается вся номенклатура, это уже построенное нами ранее расширенное декартово произведение отношений R_7 и R_8. Это отношение R_9:

   $R_9 = R_7 \otimes R_8$

2. Теперь найдем перечень того, что из обязательной номенклатуры не выпускается в некоторых цехах

   $R_{11} = R_9 \backslash R_{10}$

3. Далее найдем те цеха, в которых не все детали выпускаются, для этого нам надо отношение R_{11} спроектировать на столбец «Цех»:

   $R_{18} = R_{11}$[Цех]

R18
Цех
Цех 2
Цех 3

4. А теперь из перечня всех цехов вычтем те, кто выпускает не все детали, и получим ответ на запрос, и это будет тот же результат, что и в отношении R_{17}.

Посмотрим, как работают операции реляционной алгебры для другого примера. Возьмем набор отношений, которые моделируют сдачу сессии студентами некоторого учебного заведения. Тема весьма понятная и привычная.

R_1 = <ФИО, Дисциплина, Оценка>; R_2 = <ФИО, Группа>;

R_3 = < Группы, Дисциплина>,

где R_1 — информация о попытках (как успешных, так и неуспешных) сдачи экзаменов студентами; R_2 — состав групп; R_3 — список дисциплин, которые надо сдавать каждой группе. Домены для атрибутов формально задавать не будем, но, ориентируясь на здравый смысл, будем считать, что доменом для атрибута Дисциплина будет множество всех дисциплин, преподающихся в ВУЗе, доменом для атрибута Группа будет множество всех групп ВУЗа и т. д.

Покажем, каким образом можно получить из этих таблиц интересующие нас сведения с помощью реляционной алгебры. В каждом из приведенных примеров путем операций над исходными отношениями R_1, R_2, R_3 формируются промежуточные отношения и результирующее отношение S, содержащее требуемую информацию.

- Список студентов, которые сдали экзамен по БД на «отлично». Результат может быть получен применением операции фильтрации по сложному условию к отношению R_1 и последующим проектированием на атрибут «ФИО» (нам ведь требуется только список фамилий).

  $S = (R_1[\text{Оценка} = 5 \wedge \text{Дисциплина} = \text{«БД»}])[\text{ФИО}];$

- Список тех, кто должен был сдавать экзамен по БД, но пока еще не сдавал. Сначала найдем всех, кто должен был сдавать экзамен по БД. В отношении R_3 находится список всех дисциплин, по которым каждая группа должна была сдавать экзамены, ограничим перечень дисциплин только «БД». Для того чтобы получить список студентов, нам надо соединить отношение R_3 с отношением R_2, в котором определен список студентов каждой группы.

  $R_4 = (R_2[R_3.\text{НомерГруппы} = R_2.\text{НомерГруппы} \wedge$

  $R_3.\text{Дисциплина} = \text{«БД»}]\ R_3)[\text{ФИО}];$

- Теперь получим список всех, кто сдавал экзамен по «БД» (нас пока не интересует результат сдачи, а интересует сам факт попытки сдачи, то есть присутствие в отношении R_1):

  $R_5 = (R_1[\text{Дисциплина} = \text{«БД»}])[\text{ФИО}];$

  и, наконец, результат — все, кто есть в первом множестве, но не во втором:

  $S = R_4 \setminus R_5;$

- Список несчастных, имеющих несколько двоек:

  $S = (R_1[R_1.\text{ФИО} = R'_1.\text{ФИО} \wedge R_1.\text{Дисциплина} \neq R'_1.\text{Дисциплина} \wedge$

  $R_1.\text{Оценка} \leq 2 \wedge R'_1.\text{Оценка} \leq 2]\ R'_1)[\text{ФИО}]$

  Этот пример весьма интересен: для поиска строк, удовлетворяющих в совокупности условию *больше одного*, применяется операция соединения отношения с самим собой. Поэтому мы как бы взяли копию отношения R_1 и назвали ее R'_1.

- Список круглых отличников. Строим список всех пар <студент—дисциплина>, которые в принципе должны быть сданы:

  $R_4 = (R_2[R_2.\text{Группа} = R_3.\text{Группа}]\ R_3)[\text{ФИО, Дисциплина}];$

  Строим список пар <студент—дисциплина>, где получена оценка «отлично»:

  $R_5 = (R_1[\text{Оценка} = 5])[\text{ФИО, Дисциплина}];$

  Строим список студентов, что-либо не сдавших на «отлично»:

  $R_6 = (R_4 \setminus R_5)[\text{ФИО}].$

Наконец, исключив последнее отношение из общего списка студентов, получаем результат:

$R_2[\text{ФИО}] \setminus R_6$

Обратите внимание, что для получения множества студентов, что-либо не сдавших на «отлично» (R_6), мы осуществили «инверсию» множества всех отлично сданных пар <студент—дисциплина> (R_5) путем вычитания его из предварительного построенного универсального множества (R_4). Рекомендуем очень внимательно разобрать этот пример и вникнуть в смысл каждого действия — это очень пригодится для понимания реляционной алгебры.

Задания для самостоятельной работы

Задание 1

Даны отношения, моделирующие работу банка и его филиалов. Клиент может иметь несколько счетов, при этом они могут быть размещены как в одном, так и в разных филиалах банка. В отношении R_1 содержится информация обо всех клиентах и их счетах в филиалах нашего банка. Каждый клиент, в соответствии со своим счетом, может рассчитывать на некоторый кредит от нашего банка, сумма допустимого кредита также зафиксирована.

R_1				
ФИО клиента	№ филиала	№ счета	Остаток	Кредит

R_2	
№ филиала	Район

С использованием языка реляционной алгебры составить запросы, позволяющие выбрать:

1. Филиалы, клиенты которых имеют счета с остатком, превышающим $1000.
2. Клиентов, которые имеют счета во всех филиалах данного банка.
3. Клиентов, которые имеют только по одному счету в разных филиалах банка. То есть в общем у этих клиентов может быть несколько счетов, но в одном филиале не более одного счета.
4. Клиенты, которые имеют счета в нескольких филиалах банка, расположенных только в одном районе.
5. Филиалы, которые не имеют ни одного клиента.
6. Филиалы, которые имеют клиентов с остатком на счету 0 (ноль).

7. Филиалы, у которых есть клиенты с кредитом, превышающим остаток на счету в 2 раза.

Задание 2

Даны отношения, моделирующие работу международной фирмы, имеющей несколько филиалов. Филиалы фирмы могут быть расположены в разных странах, это отражено в отношении R_1. Клиенты фирмы также могут быть из разных стран, и это отражено в отношении R_4. По каждому конкретному заказу клиент мог заказать несколько разных товаров.

R_1	
Филиал	Страна

R_2		
Филиал	Заказчик	№ заказа

R_3		
N заказа	Товар	Количество

R_4	
Заказчик	Страна

С использованием реляционной алгебры составить запросы, позволяющие выбрать:

1. Заказчиков, которые работают со всеми филиалами фирмы, но покупают только один товар.
2. Филиалы фирмы, которые торгуют всеми товарами.
3. Товары, которые фирма продает только в одной стране.
4. Заказчиков, которые работают с филиалами фирмы, которые расположены только в одной стране.
5. Филиалы, с которыми не работает ни один заказчик.
6. Заказчиков, которые работают только с филиалами, расположенными в той же стране, что и заказчик.
7. Заказчиков, которые покупают все товары, представленные в отношении R_3.

Задание 3

Даны отношения, моделирующие работу фирмы, занимающейся разработкой программных систем. Каждый сотрудник административно закреплен только за одним отделом. Файлы хранятся на разных серверах. На разных серверах файлы могут иметь одинаковые имена. Создатель файла является его владельцем, поэтому у каждого файла только один владелец, но владелец файла может разрешить пользоваться файлом другим сотрудникам. Существует множество системного программного обеспечения, каждая программа может работать с одним или с несколькими файлами, расположенными на одном или нескольких серверах:

R_1	
Название файла	Имя владельца файла

R_4	
Сотрудник	Отдел

R_2		
Название программы	Название файла	Сервер

R_3	
Название файла	Название сервера

С использованием реляционной алгебры и языка составить запросы, позволяющие выбрать:

1. Файлы, которые имеют нескольких пользователей из разных отделов.
2. Программы, которые работают только с одним файлом.
3. Файлы, которые имеют одно и тоже имя, но расположены на различных серверах и используются сотрудниками разных отделов.
4. Файлы, с которыми работают сотрудники всех отделов.
5. Файлы, пользователями которых являются сотрудники только одного отдела.
6. Программы, которые работают со всеми серверами.
7. Отделы, сотрудники которых не работают ни с одним файлом. То есть отделы, в которых нет ни одного сотрудника, работающего с каким-нибудь файлом.
8. Отделы, сотрудники которых работают со всеми серверами.
9. Серверы, с которыми работают сотрудники только одного отдела.

ГЛАВА 5 Язык SQL. Формирование запросов к базе данных

История развития SQL

SQL (*Structured Query Language*) — *Структурированный Язык Запросов* — стандартный язык запросов по работе с реляционными БД. Язык SQL появился после реляционной алгебры, и его прототип был разработан в конце 70-х годов в компании IBM Research. Он был реализован в первом прототипе реляционной СУБД фирмы IBM System R. В дальнейшем этот язык применялся во многих коммерческих СУБД и в силу своего широкого распространения постепенно стал стандартом «де-факто» для языков манипулирования данными в реляционных СУБД.

Первый международный стандарт языка SQL был принят в 1989 г. (далее мы будем называть его SQL/89 или SQL1). Иногда стандарт SQL1 также называют стандартом ANSI/ISO, и подавляющее большинство доступных на рынке СУБД поддерживают этот стандарт полностью. Однако развитие информационных технологий, связанных с базами данных, и необходимость реализации переносимых приложений потребовали в скором времени доработки и расширения первого стандарта SQL.

В конце 1992 г. был принят новый международный стандарт языка SQL, который в дальнейшим будем называть SQL/92 или SQL2. И он не лишен недостатков, но в то же время является существенно более точным и полным, чем SQL/89. В настоящий момент большинство производителей СУБД внесли изменения в свои продукты так, чтобы они в большей степени удовлетворяли стандарту SQL2.

В 1999 году появился новый стандарт, названный SQL3. Если отличия между стандартами SQL1 и SQL2 во многом были количественными, то стандарт SQL3 соответствует качественным серьезным преобразованиям. В SQL3 введены новые типы данных, при этом предполагается возможность задания сложных

структурированных типов данных, которые в большей степени соответствуют объектной ориентации. Наконец, добавлен раздел, который вводит стандарты на события и триггеры, которые ранее не затрагивались в стандартах, хотя давно уже широко использовались в коммерческих СУБД. В стандарте определены возможности четкой спецификации триггеров как совокупности события и действия. В качестве действия могут выступать не только последовательность операторов SQL, но и операторы управления ходом выполнения программы. В рамках управления транзакциями произошел возврат к старой модели транзакций, допускающей *точки сохранения* (*savepoints*), и возможность указания в операторе отката ROOLBACK точек возврата позволит откатывать транзакцию не в начало, а в промежуточную ранее сохраненную точку. Такое решение повышает гибкость реализации сложных алгоритмов обработки информации.

А зачем вообще нужны эти стандарты? Зачем их изобретают и почему надо изучать их? Текст стандарта SQL2 занимает 600 станиц сухого формального текста, это очень много, и кажется, что это просто происки разработчиков стандартов, а не то, что необходимо рядовым разработчикам. Однако ни один серьезный разработчик, работающий с базами данных, не должен игнорировать стандарт, и для этого существуют весьма веские причины. Разработка любой информационной системы, ориентированной на технологию баз данных (а других информационных систем на настоящий момент и не бывает), является трудоемким процессом, занимающим несколько десятков и даже сотен человеко-месяцев. Следует отдавать себе отчет, что нельзя разработать сколько-нибудь серьезную систему за несколько дней. Кроме того, развитие вычислительной техники, систем телекоммуникаций и программного обеспечения столь стремительно, что проект может устареть еще до момента внедрения. Но развивается не только вычислительная техника, изменяются и реальные объекты, поведение которых моделируется использованием как самой БД, так и процедур обработки информации в ней, то есть конкретных приложений, которые составляют реальное наполнение разрабатываемой информационной системы. Именно поэтому проект информационной системы должен быть рассчитан на расширяемость и переносимость на другие платформы. Большинство поставщиков аппаратуры и программного обеспечения следуют стратегии поддержки стандартов, в противном случае пользователи просто не будут их покупать. Однако каждый поставщик стремится улучшить свой продут введением дополнительных возможностей, не входящих в стандарт. Выбор разработчиков, следовательно, таков: ориентироваться только на экзотические особенности данного продукта либо стараться в основном придерживаться стандарта. Во втором случае весь интеллектуальный труд, вкладываемый в разработку, становится более защищенным, так как система приобретает свойства переносимости. И в случае появления более перспективной платформы проект, ориентированный в большей степени на стандарты, может быть легче перенесен на нее, чем тот, который в основном ориентировался на особенности конкретной платформы. Кроме того, стандарты — это верный ориентир для разработчиков, так как все поставщики СУБД в своих перспективных разработках обязательно следуют стандарту, и можно быть уверенным, что в конце концов стандарт будет реализован практически во всех перспективных СУБД. Так произошло со стандартом SQL1, так происходит со стандартом SQL2 и так будет происходить со стандартом SQL3.

Для поставщиков СУБД стандарт — это путеводная звезда, которая гарантирует правильное направление работ. А вот эффективность реализации стандарта — это гарантия успеха.

SQL нельзя в полной мере отнести к традиционным языкам программирования, он не содержит традиционные операторы, управляющие ходом выполнения программы, операторы описания типов и многое другое, он содержит только набор стандартных операторов доступа к данным, хранящимся в базе данных. Операторы SQL встраиваются в базовый язык программирования, которым может быть любой стандартный язык типа C++, PL, COBOL и т. д. Кроме того, операторы SQL могут выполняться непосредственно в интерактивном режиме.

Структура SQL

В отличие от реляционной алгебры, где были представлены только операции запросов к БД, SQL является полным языком, в нем присутствуют не только операции запросов, но и операторы, соответствующие DDL — Data Definition Language — языку описания данных. Кроме того, язык содержит операторы, предназначенные для управления (администрирования) БД.

SQL содержит разделы, представленные в таблице 5.1:

Таблица 5.1. Операторы определения данных DDL

Оператор	Смысл	Действие
CREATE TABLE	Создать таблицу	Создает новую таблицу в БД
DROP TABLE	Удалить таблицу	Удаляет таблицу из БД
ALTER TABLE	Изменить таблицу	Изменяет структуру существующей таблицы или ограничения целостности, задаваемые для данной таблицы
CREATE VIEW	Создать представление	Создает виртуальную таблицу, соответствующую некоторому SQL-запросу
ALTER VIEW	Изменить представление	Изменяет ранее созданное представление
DROP VIEW	Удалить представление	Удаляет ранее созданное представление
CREATE INDEX	Создать индекс	Создает индекс для некоторой таблицы для обеспечения быстрого доступа по атрибутам, входящим в индекс
DROP INDEX	Удалить индекс	Удаляет ранее созданный индекс

Таблица 5.2. Операторы манипулирования данными Data Manipulation Language (DMP)

Оператор	Смысл	Действие
DELETE	Удалить строки	Удаляет одну или несколько строк, соответствующих условиям фильтрации, из базовой таблицы. Применение оператора согласуется с принципами поддержки целостности, поэтому этот оператор не всегда может быть выполнен корректно, даже если синтаксически он записан правильно
INSERT	Вставить строку	Вставляет одну строку в базовую таблицу. Допустимы модификации оператора, при которых сразу несколько строк могут быть перенесены из одной таблицы или запроса в базовую таблицу
UPDATE	Обновить строку	Обновляет значения одного или нескольких столбцов в одной или нескольких строках, соответствующих условиям фильтрации

Таблица 5.3. Язык запросов Data Query Language (DQL)

Оператор	Смысл	Действие
SELECT	Выбрать строки	Оператор, заменяющий все операторы реляционной алгебры и позволяющий сформировать результирующее отношение, соответствующее запросу

Таблица 5.4. Средства управления транзакциями

Оператор	Смысл	Действие
COMMIT	Завершить транзакцию	Завершить комплексную взаимосвязанную обработку информации, объединенную в транзакцию
ROLLBACK	Откатить транзакцию	Отменить изменения, проведенные в ходе выполнения транзакции
SAVEPOINT	Сохранить промежуточную точку выполнения транзакции	Сохранить промежуточное состояние БД, помстить его для того, чтобы можно было в дальнейшем к нему вернуться

Таблица 5.5. Средства администрирования данных

Оператор	Смысл	Действие
ALTER DATABASE	Изменить БД	Изменить набор основных объектов в базе данных, ограничений, касающихся всей базы данных
ALTER DBAREA	Изменить область хранения БД	Изменить ранее созданную область хранения

Таблица 5.5 (*продолжение*)

Оператор	Смысл	Действие
ALTER PASSWORD	Изменить пароль	Изменить пароль для всей базы данных
CREATE DATABASE	Создать БД	Создать новую базу данных, определив основные параметры для нее
CREATE DBAREA	Создать область хранения	Создать новую область хранения и сделать ее доступной для размещения данных
DROP DATABASE	Удалить БД	Удалить существующую базу данных (только в том случае, когда вы имеете право выполнить это действие)
DROP DBAREA	Удалить область хранения БД	Удалить существующую область хранения (если в ней на настоящий момент не располагаются активные данные)
GRANT	Предоставить права	Предоставить права доступа на ряд действий над некоторым объектом БД
REVOKE	Лишить прав	Лишить прав доступа к некоторому объекту или некоторым действиям над объектом

Таблица 5.6. Программный SQL

Оператор	Смысл	Действие
DECLARE	Определяет курсор для запроса	Задает некоторое имя и определяет связанный с ним запрос к БД, который соответствует виртуальному набору данных
OPEN	Открыть курсор	Формирует виртуальный набор данных, соответствующий описанию указанного курсора и текущему состоянию БД
FETCH	Считать строку из множества строк, определенных курсором	Считывает очередную строку, заданную параметром команды из виртуального набора данных, соответствующего открытому курсору
CLOSE	Закрыть курсор	Прекращает доступ к виртуальному набору данных, соответствующему указанному курсору
PREPARE	Подготовить оператор SQL к динамическому выполнению	Сгенерировать план выполнения запроса, соответствующего заданному оператору SQL

Оператор	Смысл	Действие
EXECUTE	Выполнить оператор SQL, ранее подготовленный к динамическому выполнению	Выполняет ранее подготовленный план запроса

В коммерческих СУБД набор основных операторов расширен. В большинство СУБД включены операторы определения и запуска хранимых процедур и операторы определения триггеров.

Типы данных

В языке SQL/89 поддерживаются следующие типы данных:

- CHARACTER(n) или CHAR(n) — символьные строки постоянной длины в n символов. При задании данного типа под каждое значение всегда отводится n символов, и если реальное значение занимает менее, чем n символов, то СУБД автоматически дополняет недостающие символы пробелами.
- NUMERIC[(n,m)] — точные числа, здесь n — общее количество цифр в числе, m — количество цифр слева от десятичной точки.
- DECIMAL[(n,m)] — точные числа, здесь n — общее количество цифр в числе, m — количество цифр слева от десятичной точки.
- DEC[(n,m)] — то же, что и DECIMAL[(n,m)].
- INTEGER или INT — целые числа.
- SMALLINT — целые числа меньшего диапазона.

Несмотря на то, что в стандарте SQL1 не определяется точно, что подразумевается под типом INT и SMALLINT (это отдано на откуп реализации), указано только соотношение между этими типами данных, в большинстве реализаций тип данных INTEGER соответствует целым числам, хранимым в четырех байтах, а SMALLINT — соответствует целым числам, хранимым в двух байтах. Выбор одного из этих типов определяется размером числа.

- FLOAT[(n)] — числа большой точности, хранимые в форме с плавающей точкой. Здесь n — число байтов, резервируемое под хранение одного числа. Диапазон чисел определяется конкретной реализацией.
- REAL — вещественный тип чисел, который соответствует числам с плавающей точкой, меньшей точности, чем FLOAT.
- DOUBLE PRECISION специфицирует тип данных с определенной в реализации точностью большей, чем определенная в реализации точность для REAL.

В стандарте SQL92 добавлены следующие типы данных:

- VARCHAR(n) — строки символов переменной длины.
- NCHAR(N) — строки локализованных символов постоянной длины.

- NCHAR VARYING(n) — строки локализованных символов переменной длины.
- BIT(n) — строка битов постоянной длины.
- BIT VARYING(n) — строка битов переменной длины.
- DATE — календарная дата.
- TIMESTAMP(точность) — дата и время.
- INTERVAL — временной интервал.

Большинство коммерческих СУБД поддерживают еще дополнительные типы данных, которые не специфицированы в стандарте. Так, например, практически все СУБД в том или ином виде поддерживают тип данных для представления неструктурированного текста большого объема. Этот тип аналогичен типу MEMO в настольных СУБД. Называются эти типы по-разному, например в ORACLE этот тип называется LONG, в DB2 — LONG VARCHAR, в SYBASE и MS SQL Server — TEXT.

Однако следует отметить, что специфика реализации отдельных типов данных серьезным образом влияет на результаты запросов к БД. Особенно это касается реализации типов данных DATE и TIMESTAMP. Поэтому при переносе приложений будьте внимательны, на разных платформах они могут работать по-разному, и одной из причин может быть различие в интерпретации типов данных.

При выполнении сравнений в операциях фильтрации могут использоваться константы заданных типов. В стандарте определены следующие константы. Для числовых типов данных определены константы в виде последовательности цифр с необязательным заданием знака числа и десятичной точкой. То есть правильными будут константы:

```
213-314   612.716      + 551.702
```

Константы с плавающей запятой задаются, как и в большинстве языков программирования, путем задания мантиссы и порядка, разделенных символом E, например:

```
2.9E-4    -134.235E7   0.54267E18
```

Строковые константы должны быть заключены в одинарные кавычки:

```
'Крылов Ю.Д.'          'Санкт-Петербург'
```

В некоторых реализациях, например MS SQL Server и Informix, допустимы двойные кавычки в строковых константах:

```
"Москва"               "New York"
```

Однако следует отметить, что использование двойных кавычек может вызвать дополнительные проблемы при переносе приложений на другую платформу, поэтому мы рекомендуем по возможности избегать такого представления символьных констант.

Константы даты, времени и временного интервала в реляционных СУБД представляются в виде строковых констант. Форматы этих констант отличаются в различных СУБД. Кроме того, формат представления даты различен в разных странах. В большинстве СУБД реализованы способы настройки форматов представления дат или специальные функции преобразования форматов дат, как сделано, например, в СУБД ORACLE. Приведем примеры констант в MS SQL Server:


```
March 15, 1999  Mar 15 1999  3/15/1999 3-15-99  1999 MAR 15
```

В СУБД ORACLE та же константа запишется как

```
15-MAR-99
```

Кроме пользовательских констант в СУБД могут существовать и специальные системные константы. Стандарт SQL1 определяет только одну системную константу USER, которая соответствует имени пользователя, под которым вы подключились к БД.

В операторах SQL могут использоваться выражения, которые строятся по стандартным правилам применения знаков арифметических операций сложения (+), вычитания (−), умножения (*) и деления (/). Однако в ряде СУБД операция деления (/) интерпретируется как деление нацело, поэтому при построении сложных выражений вы можете получить результат, не соответствующий традиционной интерпретации выражения. В стандарт SQL2 включена возможность выполнения операций сложения и вычитания над датами. В большинстве СУБД также определена операция конкатенации над строковыми данными, обозначается она, к сожалению, по-разному. Так, например, для DB2 операция конкатенации обозначается двойной вертикальной чертой, в MS SQL Server — знаком сложения (+), поэтому два выражения, созданные в разных СУБД, эквивалентны:

```
'Mr./Mrs. ' || NAME || ' ' LAST_NAME
'Mr./Mrs. ' + NAME + ' ' LAST_NAME
```

В стандарте SQL1 не были определены встроенные функции, однако в большинстве коммерческих СУБД такие функции были реализованы, и в стандарт SQL2 уже введен ряд стандартных встроенных функций:

- BIT_LENGTH(строка) — количество битов в строке;
- CAST(значение AS тип данных) — значение, преобразованное в заданный тип данных;
- CHAR_LENGTH(строка) — длина строки символов;
- CONVERT(строка USING функция) — строка, преобразованная в соответствии с указанной функцией;
- CURRENT_DATE — текущая дата;
- CURRENT_TIME(точность) — текущее время с указанной точностью;
- CURRENT_TIMESTAMP(точность) — текущие дата и время с указанной точностью;
- LOWER(строка) — строка, преобразованная к верхнему регистру;
- OCTED_LENGTH(строка) — число байтов в строке символов;
- POSITION(первая строка IN вторая строка) — позиция, с которой начинается вхождение первой строки во вторую;

- SUBSTRING(строка FROM n FOR длина) — часть строки, начинающаяся с n-го символа и имеющая указанную длину;
- TRANSLATE(строка USING функция) — строка, преобразованная с использованием указанной функции;
- TRIM(BOTH символ FROM строка) — строка, у которой удалены все первые и последние символы;
- TRIM(LEADING символ FROM строка) — строка, в которой удалены все первые указанные символы;
- TRIM(TRAILING символ FROM строка) — строка, в которой удалены последние указанные символы;
- UPPER(строка) — строка, преобразованная к верхнему регистру.

Оператор выбора SELECT

Язык запросов (Data Query Language) в SQL состоит из единственного оператора SELECT. Этот единственный оператор поиска реализует все операции реляционной алгебры. Как просто, всего один оператор. Однако писать запросы на языке SQL (грамотные запросы) сначала совсем не просто. Надо учиться, так же как надо учиться решать математические задачки или составлять алгоритмы для решения непростых комбинаторных задач. Один и тот же запрос может быть реализован несколькими способами, и, будучи все правильными, они, тем не менее, могут существенно отличаться по времени исполнения, и это особенно важно для больших баз данных.

Синтаксис оператора SELECT имеет следующий вид:

```
SELECT [ALL|DISTINCT](<Список полей>|*)
FROM        <Список таблиц>
[WHERE      <Предикат-условие выборки или соединения>]
[GROUP BY   <Список полей результата>]
[HAVING     <Предикат-условие для группы>]
[ORDER BY   <Список полей, по которым упорядочить вывод>]
```

Здесь ключевое слово ALL означает, что в результирующий набор строк включаются все строки, удовлетворяющие условиям запроса. Значит, в результирующий набор могут попасть одинаковые строки. И это нарушение принципов теории отношений (в отличие от реляционной алгебры, где по умолчанию предполагается отсутствие дубликатов в каждом результирующем отношении). Ключевое слово DISTINCT означает, что в результирующий набор включаются только различные строки, то есть дубликаты строк результата не включаются в набор.

Символ *, (звездочка) означает, что в результирующий набор включаются все столбцы из исходных таблиц запроса.

В разделе FROM задается перечень исходных отношений (таблиц) запроса.

В разделе WHERE задаются условия отбора строк результата или условия соединения кортежей исходных таблиц, подобно операции условного соединения в реляционной алгебре.

В разделе GROUP BY задается список полей группировки.

В разделе HAVING задаются предикаты-условия, накладываемые на каждую группу.

В части ORDER BY задается список полей упорядочения результата, то есть список полей, который определяет порядок сортировки в результирующем отношении. Например, если первым полем списка будет указана Фамилия, а вторым Номер группы, то в результирующем отношении сначала будут собраны в алфавтном порядке студенты, и если найдутся однофамильцы, то они будут расположены в порядке возрастания номеров групп.

В выражении условий раздела WHERE могут быть использованы следующие предикаты:

- Предикаты сравнения { =, <>, >,<, >=,<= }, которые имеют традиционный смысл.
- Предикат Between A and B — принимает значения между A и B. Предикат истинен, когда сравниваемое значение попадает в заданный диапазон, включая границы диапазона. Одновременно в стандарте задан и противоположный предикат Not Between A and B, который истинен тогда, когда сравниваемое значение не попадает в заданный интервал, включая его границы.
- Предикат вхождения в множество IN (множество) истинен тогда, когда сравниваемое значение входит в множество заданных значений. При этом множество значений может быть задано простым перечислением или встроенным подзапросом. Одновременно существует противоположный предикат NOT IN (множество), который истинен тогда, когда сравниваемое значение не входит в заданное множество.
- Предикаты сравнения с образцом LIKE и NOT LIKE. Предикат LIKE требует задания шаблона, с которым сравнивается заданное значение, предикат истинен, если сравниваемое значение соответствует шаблону, и ложен в противном случае. Предикат NOT LIKE имеет противоположный смысл.

  По стандарту в шаблон могут быть включены специальные символы:

  - символ подчеркивания (_) — для обозначения любого одиночного символа;
  - символ процента (%) — для обозначения любой произвольной последовательности символов;
  - остальные символы, заданные в шаблоне, обозначают самих себя.
- Предикат сравнения с неопределенным значением IS NULL. Понятие неопределенного значения было внесено в концепции баз данных позднее. Неопределенное значение интерпретируется в реляционной модели как значение, неизвестное на данный момент времени. Это значение при появлении дополнительной информации в любой момент времени может быть заменено на некоторое конкретное значение. При сравнении неопределенных значений не действуют стандартные правила сравнения: одно неопределенное значение никогда не считается равным другому неопределенному значению. Для вы-

явления равенства значения некоторого атрибута неопределенному применяют специальные стандартные предикаты:

<имя атрибута>IS NULL и <имя атрибута> IS NOT NULL.

Если в данном кортеже (в данной строке) указанный атрибут имеет неопределенное значение, то предикат IS NULL принимает значение «Истина» (TRUE), а предикат IS NOT NULL — «Ложь» (FALSE), в противном случае предикат IS NULL принимает значение «Ложь», а предикат IS NOT NULL принимает значение «Истина».

Введение Null-значений вызвало необходимость модификации классической двузначной логики и превращения ее в трехзначную. Все логические операции, производимые с неопределенными значениями, подчиняются этой логике в соответствии с заданной таблицей истинности:

A	B	Not A	A ∧ B	A ∨ B
TRUE	TRUE	FALSE	TRUE	TRUE
TRUE	FALSE	FALSE	FALSE	TRUE
TRUE	Null	FALSE	Null	TRUE
FALSE	TRUE	TRUE	FALSE	TRUE
FALSE	FALSE	TRUE	FALSE	FALSE
FALSE	Null	TRUE	FALSE	Null
Null	TRUE	Null	Null	TRUE
Null	FALSE	Null	FALSE	Null
Null	Null	Null	Null	Null

- Предикаты существования EXIST и несуществования NOT EXIST. Эти предикаты относятся к встроенным подзапросам, и подробнее мы рассмотрим их, когда коснемся вложенных подзапросов.

В условиях поиска могут быть использованы все рассмотренные ранее предикаты.

Отложив на время знакомство с группировкой, рассмотрим детально первые три строки оператора SELECT:

- SELECT — ключевое слово, которое сообщает СУБД, что эта команда — запрос. Все запросы начинаются этим словом с последующим пробелом. За ним может следовать способ выборки — с удалением дубликатов (DISTINCT) или без удаления (ALL, подразумевается по умолчанию). Затем следует список перечисленных через запятую столбцов, которые выбираются запросом из таблиц, или символ '*' (звездочка) для выбора всей строки. Любые столбцы, не перечисленные здесь, не будут включены в результирующее отношение, соответствующее выполнению команды. Это, конечно, не значит, что они будут удалены или их информация будет стерта из таблиц, потому что запрос не воздействует на информацию в таблицах — он только показывает данные.
- FROM — ключевое слово, подобно SELECT, которое должно быть представлено в каждом запросе. Оно сопровождается пробелом и затем именами таблиц,

используемых в качестве источника информации. В случае если указано более одного имени таблицы, неявно подразумевается, что над перечисленными таблицами осуществляется операция декартова произведения. Таблицам можно присвоить имена-псевдонимы, что бывает полезно для осуществления операции соединения таблицы с самой собою или для доступа из вложенного подзапроса к текущей записи внешнего запроса (вложенные подзапросы здесь не рассматриваются).

Все последующие разделы оператора SELECT являются необязательными.

Самый простой запрос SELECT без необязательных частей соответствует просто декартову произведению. Например, выражение

```
SELECT *
FROM R1, R2
```

соответствует декартову произведению таблиц R1 и R2.

Выражение

```
SELECT R1.A, R2.B
FROM R1, R2
```

соответствует проекции декартова произведения двух таблиц на два столбца A из таблицы R1 и B из таблицы R2, при этом дубликаты всех строк сохранены, в отличие от операции проектирования в реляционной алгебре, где при проектировании по умолчанию все дубликаты кортежей уничтожаются.

- WHERE — ключевое слово, за которым следует предикат — условие, налагаемое на запись в таблице, которому она должна удовлетворять, чтобы попасть в выборку, аналогично операции селекции в реляционной алгебре.

Рассмотрим базу данных, которая моделирует сдачу сессии в некотором учебном заведении. Пусть она состоит из трех отношений R_1, R_2, R_3. Будем считать, что они представлены таблицами R1, R2 и R3 соответственно.

R_1 = ⟨ФИО, Дисциплина, Оценка⟩; R_2 = ⟨ФИО, Группа⟩;
R_3 = ⟨Группы, Дисциплина ⟩

R1		
ФИО	**Дисциплина**	**Оценка**
Петров Ф. И.	Базы данных	5
Сидоров К. А.	Базы данных	4
Миронов А. В.	Базы данных	2
Степанова К. Е.	Базы данных	2
Крылова Т. С.	Базы данных	5
Сидоров К. А.	Теория информации	4
Степанова К. Е.	Теория информации	2
Крылова Т. С.	Теория информации	5

R1		
ФИО	**Дисциплина**	**Оценка**
Миронов А. В.	Теория информации	Null
Владимиров В. А.	Базы данных	5
Трофимов П. А.	Сети и телекоммуникации	4
Иванова Е. А.	Сети и телекоммуникации	5
Уткина Н. В.	Сети и телекоммуникации	5
Владимиров В. А.	Английский язык	4
Трофимов П. А.	Английский язык	5
Иванова Е. А.	Английский язык	3
Петров Ф. И.	Английский язык	5

R2	
ФИО	**Группа**
Петров Ф. И.	4906
Сидоров К. А.	4906
Миронов А. В.	4906
Крылова Т. С.	4906
Владимиров В. А.	4906
Трофимов П. А.	4807
Иванова Е. А.	4807
Уткина Н. В.	4807

R3	
Группа	**Дисциплина**
4906	Базы данных
4906	Теория информации
4906	Английский язык
4807	Английский язык
4807	Сети и телекоммуникации

Приведем несколько примеров использования оператора SELECT.

- Вывести список всех групп (без повторений), где должны пройти экзамены.

```
SELECT DISTINCT Группы
FROM R3
```


Результат:

Группа
4906
4807

- Вывести список студентов, которые сдали экзамен по дисциплине «Базы данных» на «отлично».

```
SELECT ФИО
FROM R1
WHERE Дисциплина = "Базы данных" AND Оценка = 5
```

Результат:

ФИО
Петров Ф. И.
Крылова Т. С.

- Вывести список всех студентов, которым надо сдавать экзамены с указанием названий дисциплин, по которым должны проводиться эти экзамены.

```
SELECT ФИО.Дисциплина
FROM R2.R3
WHERE R2.Группа = R2.Группа;
```

Здесь часть WHERE задает условия соединения отношений R2 и R3, при отсутствии условий соединения в части WHERE результат будет эквивалентен расширенному декартову произведению, и в этом случае каждому студенту были бы приписаны все дисциплины из отношения R3, а не те, которые должна сдавать его группа.

Результат:

ФИО	Дисциплина
Петров Ф. И.	Базы данных
Сидоров К. А.	Базы данных
Миронов А. В.	Базы данных
Степанова К. Е.	Базы данных
Крылова Т. С.	Базы данных
Владимиров В. А.	Базы данных
Петров Ф. И.	Теория информации
Сидоров К. А.	Теория информации
Миронов А. В.	Теория информации
Степанова К. Е.	Теория информации

ФИО	Дисциплина
Крылова Т. С.	Теория информации
Владимиров В. А.	Теория информации
Петров Ф. И.	Английский язык
Сидоров К. А.	Английский язык
Миронов А. В.	Английский язык
Степанова К. Е.	Английский язык
Крылова Т. С.	Английский язык
Владимиров В. А.	Английский язык
Трофимов П. А.	Сети и телекоммуникации
Иванова Е. А.	Сети и телекоммуникации
Уткина Н. В.	Сети и телекоммуникации
Трофимов П. А.	Английский язык
Иванова Е. А.	Английский язык
Уткина Н. В.	Английский язык

- Вывести список лентяев, имеющих несколько двоек.

```
SELECT DISTINCT R1.ФИО
FROM R1 a, R1 b
WHERE a.ФИО = b.ФИО AND
    a.Дисциплина <> b.Дисциплина AND
    a.Оценка <= 2 AND b.Оценка <= 2;
```

Здесь мы использовали псевдонимы для именования отношения R_1 a и b, так как для записи условий поиска нам необходимо работать сразу с двумя экземплярами данного отношения.

Результат:

ФИО
Степанова К. Е.

Из этих примеров хорошо видно, что логика работы оператора выбора (декартово произведение—селекция—проекция) не совпадает с порядком описания в нем данных (сначала список полей для проекции, потом список таблиц для декартова произведения, потом условие соединения). Дело в том, что SQL изначально разрабатывался для применения конечными пользователями, и его стремились сделать возможно ближе к языку естественному, а не к языку алгоритмическому. По этой причине SQL на первых порах вызывает путаницу и раздражение у начинающих его изучать профессиональных программистов, которые привыкли разговаривать с машиной именно на алгоритмических языках.

Наличие неопределенных (Null) значений повышает гибкость обработки информации, хранящейся в БД. В наших примерах мы можем предположить ситуацию, когда студент пришел на экзамен, но не сдавал его по некоторой причине, в этом случае оценка по некоторой дисциплине для данного студента имеет неопределенное значение. В данной ситуации можно поставить вопрос: «Найти студентов, пришедших на экзамен, но не сдававших его с указанием названия дисциплины». Оператор SELECT будет выглядеть следующим образом:

```
SELECT ФИО, Дисциплина
FROM R1
WHERE Оценка IS NULL
```

Результат:

ФИО	Дисциплина
Миронов А. В.	Теория информации

Применение агрегатных функций и вложенных запросов в операторе выбора

В SQL добавлены дополнительные функции, которые позволяют вычислять обобщенные групповые значения. Для применения агрегатных функций предполагается предварительная операция группировки. В чем состоит суть операции группировки? При группировке все множество кортежей отношения разбивается на группы, в которых собираются кортежи, имеющие одинаковые значения атрибутов, которые заданы в списке группировки.

Например, сгруппируем отношение R1 по значению столбца Дисциплина. Мы получим 4 группы, для которых можем вычислить некоторые групповые значения, например количество кортежей в группе, максимальное или минимальное значение столбца Оценка.

Это делается с помощью агрегатных функций. Агрегатные функции вычисляют одиночное значение для всей группы таблицы. Список этих функций представлен в таблице 5.7.

Таблица 5.7. Агрегатные функции

Функция	Результат
COUNT	Количество строк или непустых значений полей, которые выбрал запрос
SUM	Сумма всех выбранных значений данного поля
AVG	Среднеарифметическое значение всех выбранных значений данного поля
MIN	Наименьшее из всех выбранных значений данного поля
MAX	Наибольшее из всех выбранных значений данного поля

R1			
	ФИО	**Дисциплина**	**Оценка**
Группа 1	Петров Ф. И.	Базы данных	5
	Сидоров К. А.	Базы данных	4
	Миронов А. В.	Базы данных	2
	Степанова К. Е.	Базы данных	2
	Крылова Т. С.	Базы данных	5
	Владимиров В. А.	Базы данных	5
Группа 2	Сидоров К. А.	Теория информации	4
	Степанова К. Е.	Теория информации	2
	Крылова Т. С.	Теория информации	5
	Миронов А. В.	Теория информации	Null
Группа 3	Трофимов П. А.	Сети и телекоммуникации	4
	Иванова Е. А.	Сети и телекоммуникации	5
	Уткина Н. В.	Сети и телекоммуникации	5
Группа 4	Владимиров В. А.	Английский язык	4
	Трофимов П. А.	Английский язык	5
	Иванова Е. А.	Английский язык	3
	Петров Ф. И.	Английский язык	5

Агрегатные функции используются подобно именам полей в операторе SELECT, но с одним исключением: они берут имя поля как аргумент. С функциями SUM и AVG могут использоваться только числовые поля. С функциями COUNT, MAX и MIN могут использоваться как числовые, так и символьные поля. При использовании с символьными полями MAX и MIN будут транслировать их в эквивалент ASCII кода и обрабатывать в алфавитном порядке. Некоторые СУБД позволяют использовать вложенные агрегаты, но это является отклонением от стандарта ANSI со всеми вытекающими отсюда последствиями.

Например, можно вычислить количество студентов, сдававших экзамены по каждой дисциплине. Для этого надо выполнить запрос с группировкой по полю «Дисциплина» и вывести в качестве результата название дисциплины и количество строк в группе по данной дисциплине. Применение символа * в качестве аргумента функции COUNT означает подсчет всех строк в группе.

```
SELECT R1.Дисциплина, COUNT(*)
FROM R1
GROUP BY R1.Дисциплина
```

Результат:

Дисциплина	**COUNT(*)**
Базы данных	6
Теория информации	4
Сети и телекоммуникации	3
Английский язык	4

Если же мы хотим сосчитать количество сдавших экзамен по какой-либо дисциплине, то нам необходимо исключить неопределенные значения из исходного отношения перед группировкой. В этом случае запрос будет выглядеть следующим образом:

```
SELECT R1.Дисциплина, COUNT(*)
FROM R1
WHERE R1.Оценка IS NOT NULL
GROUP BY R1.Дисциплина
```

Получим результат:

Дисциплина	**COUNT(*)**
Базы данных	6
Теория информации	3
Сети и телекоммуникации	3
Английский язык	4

В этом случае строка со студентом

Миронов А. В.	Теория информации	Null

не попадет в набор кортежей перед группировкой, поэтому количество кортежей в группе для дисциплины «Теория информации» будет на 1 меньше.

Можно применять агрегатные функции также и без операции предварительной группировки, в этом случае все отношение рассматривается как одна группа и для этой группы можно вычислить одно значение на группу.

Обратившись снова к базе данных «Сессия» (таблицы R1, R2, R3), найдем количество успешно сданных экзаменов:

```
SELECT COUNT(*)
FROM R1
WHERE Оценка > 2;
```

Это, конечно, отличается от выбора поля, поскольку всегда возвращается одиночное значение, независимо от того, сколько строк находится в таблице. Аргументом агрегатных функций могут быть отдельные столбцы таблиц. Но для

того, чтобы вычислить, например, количество различных значений некоторого столбца в группе, необходимо применить ключевое слово DISTINCT совместно с именем столбца. Вычислим количество различных оценок, полученных по каждой дисциплине:

```
SELECT R1.Дисциплина, COUNT(DISTINCT R1.Оценка)
FROM R1
WHERE R1.Оценка IS NOT NULL
GROUP BY R1.Дисциплина
```

Результат:

Дисциплина	COUNT(DISTINCT R1.Оценка)
Базы данных	3
Теория информации	3
Сети и телекоммуникации	2
Английский язык	3

В результат можно включить значение поля группировки и несколько агрегатных функций, а в условиях группировки можно использовать несколько полей. При этом группы образуются по набору заданных полей группировки. Операции с агрегатными функциями могут быть применены к объединению множества исходных таблиц. Например, поставим вопрос: определить для каждой группы и каждой дисциплины количество успешно сдавших экзамен и средний балл по дисциплине.

```
SELECT R2.Группа, R1.Дисциплина, COUNT(*), AVR(Оценка)
FROM R1,R2
WHERE R1.ФИО = R2.ФИО AND
R1.Оценка IS NOT NULL AND
R1.Оценка > 2
GROUP BY R2.Группа, R1.Дисциплина
```

Результат:

Дисциплина	COUNT(*)	AVR(Оценка)
Базы данных	6	3.83
Теория информации	3	3.67
Сети и телекоммуникации	3	4.66
Английский язык	4	4.25

Мы не можем использовать агрегатные функции в предложении WHERE, потому что предикаты оцениваются в терминах одиночной строки, а агрегатные функции — в терминах групп строк.

Предложение GROUP BY позволяет определять подмножество значений в особом поле в терминах другого поля и применять функцию агрегата к подмножеству. Это дает возможность объединять поля и агрегатные функции в едином предложении SELECT. Агрегатные функции могут применяться как в выражении вывода результатов строки SELECT, так и в выражении условия обработки сформированных групп HAVING. В этом случае каждая агрегатная функция вычисляется для каждой выделенной группы. Значения, полученные при вычислении агрегатных функций, могут быть использованы для вывода соответствующих результатов или для условия отбора групп.

Построим запрос, который выводит группы, в которых по одной дисциплине на экзаменах получено больше одной двойки:

```
SELECT R2.Группа
FROM R1,R2
WHERE R1.ФИО = R2.ФИО AND
R1.Оценка = 2
GROUP BY R2.Группа , R1.Дисциплина
HAVING count(*)> 1
```

В дальнейшем в качестве примера будем работать не с БД «Сессия», а с БД «Банк», состоящей из одной таблицы F, в которой хранится отношение F, содержащее информацию о счетах в филиалах некоторого банка:

F = (N, ФИО, Филиал, ДатаОткрытия, ДатаЗакрытия, Остаток);

Q = (Филиал, Город);

поскольку на этой базе можно ярче проиллюстрировать работу с агрегатными функциями и группировкой.

Например, предположим, что мы хотим найти суммарный остаток на счетах в филиалах. Можно сделать раздельный запрос для каждого из них, выбрав SUM(Остаток) из таблицы для каждого филиала. GROUP BY, однако, позволит поместить их все в одну команду:

```
SELECT Филиал, SUM(Остаток)
FROM F
GROUP BY Филиал;
```

GROUP BY применяет агрегатные функции независимо для каждой группы, определяемой с помощью значения поля Филиал. Группа состоит из строк с одинаковым значением поля Филиал, и функция SUM применяется отдельно для каждой такой группы, то есть суммарный остаток на счетах подсчитывается отдельно для каждого филиала. Значение поля, к которому применяется GROUP BY, имеет, по определению, только одно значение на группу вывода, как и результат работы агрегатной функции. Поэтому мы можем совместить в одном запросе агрегат и поле. Вы можете также использовать GROUP BY с несколькими полями.

Предположим, что мы хотели бы увидеть только те суммарные значения остатков на счетах, которые превышают $5000. Чтобы увидеть суммарные остатки

свыше $5000, необходимо использовать предложение HAVING. Предложение HAVING определяет критерии, используемые, чтобы удалять определенные группы из вывода, точно так же как предложение WHERE делает это для индивидуальных строк.

Правильной командой будет следующая:

```
SELECT Филиал, SUM(Остаток)
FROM F
GROUP BY Филиал
HAVING SUM(Остаток) > 5000;
```

Аргументы в предложении HAVING подчиняются тем же самым правилам, что и в предложении SELECT, где используется GROUP BY. Они должны иметь одно значение на группу вывода.

Следующая команда будет запрещена:

```
SELECT Филиал,SUM(Остаток)
FROM F
GROUP BY Филиал
HAVING ДатаОткрытия = 27/12/1999;
```

Поле ДатаОткрытия не может быть использовано в предложении HAVING, потому что оно может иметь больше чем одно значение на группу вывода. Чтобы избежать такой ситуации, предложение HAVING должно ссылаться только на агрегаты и поля, выбранные GROUP BY. Имеется правильный способ сделать вышеупомянутый запрос:

```
SELECT Филиал,SUM(Остаток)
FROM F
WHERE ДатаОткрытия = '27/12/1999'
GROUP BY Филиал;
```

Смысл данного запроса следующий: найти сумму остатков по каждому филиалу счетов, открытых 27 декабря 1999 года.

Как и говорилось ранее, HAVING может использовать только аргументы, которые имеют одно значение на группу вывода. Практически, ссылки на агрегатные функции — наиболее общие, но и поля, выбранные с помощью GROUP BY, также допустимы. Например, мы хотим увидеть суммарные остатки на счетах филиалов в Санкт-Петербурге, Пскове и Урюпинске:

```
SELECT Филиал, SUM(Остаток)
FROM  F,Q
WHERE  F.Филиал = Q.Филиал
GROUP BY Филиал
HAVING Филиал IN ("Санкт-Петербург", "Псков", "Урюпинск");
```

Поэтому в арифметических выражениях предикатов, входящих в условие выборки раздела HAVING, прямо можно использовать только спецификации столбцов, указанных в качестве столбцов группирования в разделе GROUP BY. Остальные столбцы можно специфицировать только внутри спецификаций агрегатных функций COUNT, SUM, AVG, MIN и MAX, вычисляющих в данном случае некоторое агрегатное значение для всей группы строк. Аналогично обстоит дело с подзапросами, входящими в предикаты условия выборки раздела HAVING: если в подзапросе используется характеристика текущей группы, то она может задаваться только путем ссылки на столбцы группирования.

Результатом выполнения раздела HAVING является сгруппированная таблица, содержащая только те группы строк, для которых результат вычисления условия поиска есть TRUE. В частности, если раздел HAVING присутствует в табличном выражении, не содержащем GROUP BY, то результатом его выполнения будет либо пустая таблица, либо результат выполнения предыдущих разделов табличного выражения, рассматриваемый как одна группа без столбцов группирования.

Вложенные запросы

Теперь вернемся к БД «Сессия» и рассмотрим на ее примере использование вложенных запросов.

С помощью SQL можно вкладывать запросы внутрь друг друга. Обычно внутренний запрос генерирует значение, которое проверяется в предикате внешнего запроса (в предложении WHERE или HAVING), определяющего, верно оно или нет. Совместно с подзапросом можно использовать предикат EXISTS, который возвращает истину, если вывод подзапроса не пуст.

В сочетании с другими возможностями оператора выбора, такими как группировка, подзапрос представляет собой мощное средство для достижения нужного результата. В части FROM оператора SELECT допустимо применять синонимы к именам таблицы, если при формировании запроса нам требуется более чем один экземпляр некоторого отношения. Синонимы задаются с использованием ключевого слова AS, которое может быть вообще опущено. Поэтому часть FROM может выглядеть следующим образом:

```
FROM R1 AS A, R1 AS B
```

или

```
FROM R1 A, R1 B;
```

оба выражения эквивалентны и рассматриваются как применения оператора SELECT к двум экземплярам таблицы R1.

Например, покажем, как выглядят на SQL некоторые запросы к БД «Сессия»:

- Список тех, кто сдал все положенные экзамены.

```
SELECT ФИО
FROM R1 as a
WHERE Оценка > 2
GROUP BY ФИО
HAVING COUNT(*) = (SELECT COUNT(*)
```

```
    FROM R2,R3
    WHERE R2.Группа=R3.Группа
AND ФИО=a.ФИО)
```

Здесь во встроенном запросе определяется общее число экзаменов, которые должен сдавать каждый студент, обучающийся в группе, в которой учится данный студент, и это число сравнивается с числом экзаменов, которые сдал данный студент.

- Список тех, кто должен был сдавать экзамен по БД, но пока еще не сдавал.

```
SELECT ФИО
FROM R2 a, R3
WHERE R2.Группа=R3.Группа AND Дисциплина = "БД" AND NOT EXISTS (SELECT ФИО
    FROM R1
    WHERE ФИО=a.ФИО AND Дисциплина = "БД")
```

Предикат EXISTS (SubQuery) истинен, когда подзапрос SubQuery не пуст, то есть содержит хотя бы один кортеж, в противном случае предикат EXISTS ложен.

Предикат NOT EXISTS обратно — истинен только тогда, когда подзапрос SubQuery пуст.

Обратите внимание, каким образом NOT EXISTS с вложенным запросом позволяет обойтись без операции разности отношений. Например, формулировка запроса со словом «все» может быть выполнена как бы с двойным отрицанием. Рассмотрим пример базы, которая моделирует поставку отдельных деталей отдельными поставщиками, она представлена одним отношением SP «Поставщики—детали» со схемой

```
SP (Номер_поставщика, номер_детали)
    P (номер_детали, наименование)
```

Вот каким образом формулируется ответ на запрос: «Найти поставщиков, которые поставляют все детали».

```
SELECT DISTINCT НОМЕР_ПОСТАВЩИКА
FROM SP SP1
WHERE NOT EXISTS
    (SELECT номер_детали
        FROM P
        WHERE NOT EXISTS
            (SELECT * FROM SP SP2
            WHERE SP2.номер_поставщика=SP1.номер_поставщика AND
            sp2.номер_детали = P.номер_детали));
```

Фактически мы переформулировали этот запрос так: «Найти поставщиков таких, что не существует детали, которую бы они не поставляли». Следует отметить, что этот запрос может быть реализован и через агрегатные функции с подзапросом:


```
SELECT DISTINCT Номер_поставщика
FROM SP
GROUP BY Номер_поставщика
HAVING Count(DISTINCT номер_детали) =
    (SELECT Count( номер_детали)
    FROM P)
```

В стандарте SQL92 операторы сравнения расширены до многократных сравнений с использованием ключевых слов ANY и ALL. Это расширение используется при сравнении значения определенного столбца со столбцом данных, возвращаемым вложенным запросом.

Ключевое слово ANY, поставленное в любом предикате сравнения, означает, что предикат будет истинен, если хотя бы для одного значения из подзапроса предикат сравнения истинен. Ключевое слово ALL требует, чтобы предикат сравнения был бы истинен при сравнении со всеми строками подзапроса.

Например, найдем студентов, которые сдали все экзамены на оценку не ниже чем «хорошо». Работаем с той же базой «Сессия», но добавим к ней еще одно отношение R4, которое характеризует сдачу лабораторных работ в течение семестра:

R_1 = (ФИО, Дисциплина, Оценка);

R_2 = (ФИО, Группа);

R_3 = (Группы, Дисциплина)

R_4 = (ФИО, Дисциплина, Номер_лаб_раб, Оценка);

```
Select R1.Фио
    From R1
    Where 4 > = All (Select R1.Оценка
        From R1 as R11
        Where R1.Фио = R11.Фио)
```

Рассмотрим еще один пример:

Выбрать студентов, у которых оценка по экзамену не меньше, чем хотя бы одна оценка по сданным им лабораторным работам по данной дисциплины:

```
Select R1.Фио
    From R1
    Where R1.Оценка >= ANY (Select R4.Оценка
        From R4
        Where R1.Дисциплина = R4. Дисциплина  AND  R1.Фио = R4.Фио)
```

Внешние объединения

Стандарт SQL2 расширил понятие условного объединения. В стандарте SQL1 при объединении отношений использовались только условия, задаваемые в час-

ти WHERE оператора SELECT, и в этом случае в результирующее отношение попадали только сцепленные по заданным условиям кортежи исходных отношений, для которых эти условия были определены и истинны. Однако в действительности часто необходимо объединять таблицы таким образом, чтобы в результат попали все строки из первой таблицы, а вместо тех строк второй таблицы, для которых не выполнено условие соединения, в результат попадали бы неопределенные значения. Или наоборот, включаются все строки из правой (второй) таблицы, а отсутствующие части строк из первой таблицы дополняются неопределенными значениями. Такие объединения были названы внешними в противоположность объединениям, определенным стандартом SQL1, которые стали называться внутренними.

В общем случае синтаксис части FROM в стандарте SQL2 выглядит следующим образом:

```
FROM <список исходных таблиц> |
   < выражение естественного объединения > |
   < выражение объединения > |
   < выражение перекрестного объединения > |
   < выражение запроса на объединение >
<список исходных таблиц>::= <имя_таблицы_1> [ имя синонима таблицы_1] [ …]
   [,<имя_таблицы_n>[ <имя синонима таблицы_n> ] ]

<выражение естественного объединения>:: =
   <имя_таблицы_1> NATURAL { INNER | FULL [OUTER] |
   LEFT [OUTER] | RIGHT [OUTER]} JOIN <имя_таблицы_2>

<выражение перекрестного объединения>:: = <имя_таблицы_1>
   CROSS JOIN <имя_таблицы_2>

<выражение запроса на объединение>::=
   <имя_таблицы_1> UNION JOIN <имя_таблицы_2>

<выражение объединения>::= <имя_таблицы_1> { INNER |
   FULL [OUTER] | LEFT [OUTER] | RIGHT [OUTER]}
JOIN {ON условие | [USING (список столбцов)]} <имя_таблицы_2>
```

В этих определениях INNER — означает внутреннее объединение, LEFT — левое объединение, то есть в результат входят все строки таблицы 1, а части результирующих кортежей, для которых не было соответствующих значений в таблице 2, дополняются значениями NULL (неопределено). Ключевое слово RIGHT означает правое внешнее объединение, и в отличие от левого объединения в этом случае в результирующее отношение включаются все строки таблицы 2, а недостающие части из таблицы 1 дополняются неопределенными значениями, Ключевое слово FULL определяет полное внешнее объединение: и левое и правое. При полном внешнем объединении выполняются и правое и левое внешние объединения и в результирующее отношение включаются все строки из таблицы 1, дополненные неопределенными значениями, и все строки из таблицы 2, также дополненные неопределенными значениями.

Ключевое слово OUTER означает внешнее, но если заданы ключевые слова FULL, LEFT, RIGHT, то объединение всегда считается внешним.

Рассмотрим примеры выполнения внешних объединений. Снова вернемся к БД «Сессия». Создадим отношение, в котором будут стоять все оценки, полученные всеми студентами по всем экзаменам, которые они должны были сдавать. Если студент не сдавал данного экзамена, то вместо оценки у него будет стоять неопределенное значение. Для этого выполним последовательно естественное внутреннее объединение таблиц R2 и R3 по атрибуту Группа, а полученное отношение соединим левым внешним естественным объединением с таблицей R1, используя столбцы ФИО и Дисциплина. При этом в стандарте разрешено использовать скобочную структуру, так как результат объединения может быть одним из аргументов в части FROM оператора SELECT.

```
SELECT R1.ФИО, R1.Дисциплина, R1.Оценка
FROM (R2 NATURAL INNER JOIN R3 ) LEFT JOIN R1 USING ( ФИО, Дисциплина)
```

Результат:

ФИО	Дисциплина	Оценка
Петров Ф. И.	Базы данных	5
Сидоров К. А.	Базы данных	4
Миронов А. В.	Базы данных	2
Степанова К. Е.	Базы данных	2
Крылова Т. С.	Базы данных	5
Владимиров В. А.	Базы данных	5
Петров Ф. И.	Теория информации	Null
Сидоров К. А.	Теория информации	4
Миронов А. В.	Теория информации	Null
Степанова К. Е.	Теория информации	2
Крылова Т. С.	Теория информации	5
Владимиров В. А.	Теория информации	Null
Петров Ф. И.	Английский язык	5
Сидоров К. А.	Английский язык	Null
Миронов А. В.	Английский язык	Null
Степанова К. Е.	Английский язык	Null
Крылова Т. С.	Английский язык	Null
Владимиров В. А.	Английский язык	4
Трофимов П. А.	Сети и телекоммуникации	4
Иванова Е. А.	Сети и телекоммуникации	5

ФИО	Дисциплина	Оценка
Уткина Н. В.	Сети и телекоммуникации	5
Трофимов П. А.	Английский язык	5
Иванова Е. А.	Английский язык	3
Уткина Н. В.	Английский язык	Null

Рассмотрим еще один пример, для этого возьмем БД «Библиотека». Она состоит из трех отношений, имена атрибутов здесь набраны латинскими буквами, что является необходимым в большинстве коммерческих СУБД.

```
BOOKS(ISBN, TITL, AUTOR, COAUTOR, YEARIZD, PAGES)
READER(NUM_READER, NAME_READER, ADRESS, HOOM_PHONE, WORK_PHONE, BIRTH_DAY)
EXEMPLARE(INV, ISBN, YES_NO, NUM_READER, DATE_IN, DATE_OUT)
```

Здесь таблица BOOKS описывает все книги, присутствующие в библиотеке, она имеет следующие атрибуты:

- ISBN — уникальный шифр книги;
- TITL — название книги;
- AUTOR — фамилия автора;
- COAUTOR — фамилия соавтора;
- YEARIZD — год издания;
- PAGES — число страниц.

Таблица READER хранит сведения обо всех читателях библиотеки, и она содержит следующие атрибуты:

- NUM_READER — уникальный номер читательского билета;
- NAME_READER — фамилию и инициалы читателя;
- ADRESS — адрес читателя;
- HOOM_PHONE — номер домашнего телефона;
- WORK_PHONE — номер рабочего телефона;
- BIRTH_DAY — дату рождения читателя.

Таблица EXEMPLARE содержит сведения о текущем состоянии всех экземпляров всех книг. Она включает в себя следующие столбцы:

- INV — уникальный инвентарный номер экземпляра книги;
- ISBN — шифр книги, который определяет, какая это книга, и ссылается на сведения из первой таблицы;
- YES_NO — признак наличия или отсутствия в библиотеке данного экземпляра в текущий момент;
- NUM_READER — номер читательского билета, если книга выдана читателю, и Null в противном случае;

- DATE_IN — если книга у читателя, то это дата, когда она выдана читателю;
- DATE_OUT — дата, когда читатель должен вернуть книгу в библиотеку.

Определим перечень книг у каждого читателя; если у читателя нет книг, то номер экземпляра книги равен NULL. Для выполнения этого поиска нам надо использовать левое внешнее объединение, то есть мы берем все строки из таблицы READER и соединяем со строками из таблицы EXEMPLARE, если во второй таблице нет строки с соответствующим номером читательского билета, то в строке результирующего отношения атрибут EXEMPLARE.INV будет иметь неопределенное значение NULL:

```
SELECT READER.NAME_READER, EXEMPLARE.INV
FROM READER RIGHT JOIN EXEMPLARE ON READER.NUM_READER=EXEMPLARE.NUM_READER
```

Операция внешнего объединения, как мы уже упоминали, может использоваться для формирования источников в предложении FROM, поэтому допустимым будет, например, следующий текст запроса:

```
SELECT *
   FROM ( BOOKS LEFT JOIN EXEMPLARE)
      LEFT JOIN
   (READER NATURAL JOIN EXEMPLARE)
      USING (ISBN)
```

При этом для книг, ни один экземпляр которых не находится на руках у читателей, значения номера читательского билета и дат взятия и возврата книги будут неопределенными.

Перекрестное объединение в трактовке стандарта SQL2 соответствует операции расширенного декартова произведения, то есть операции соединения двух таблиц, при которой каждая строка первой таблицы соединяется с каждой строкой второй таблицы.

Операция *запроса на объединение* эквивалентна операции теоретико-множественного объединения в алгебре. При этом требование эквивалентности схем исходных отношений сохраняется. Запрос на объединение выполняется по следующей схеме:

```
SELECT - запрос
   UNION
SELECT - запрос
   UNION
SELECT - запрос
```

Все запросы, участвующие в операции объединения, не должны содержать выражений, то есть вычисляемых полей.

Например, нужно вывести список читателей, которые держат на руках книгу «Идиот» или книгу «Преступление и наказание». Вот как будет выглядеть запрос:

```
SELECT READER.NAME_READER
FROM  READER, EXEMPLARE.BOOKS
WHERE EXEMPLARE.NUM_READER= READER.NUM_READER AND
EXEMPLRE.ISBN = BOOKS.ISBN AND
BOOKS.TITLE = "Идиот"
UNION
SELECT READER.NAME_READER
FROM  READER, EXEMPLARE,BOOKS
WHERE EXEMPLARE.NUM_READER= READER.NUM_READER AND
EXEMPLRE.ISBN = BOOKS.ISBN AND
BOOKS.TITLE = "Преступление и наказание"
```

По умолчанию при выполнении запроса на объединение дубликаты кортежей всегда исключаются. Поэтому, если найдутся читатели, у которых находятся на руках обе книги, то они все равно в результирующий список попадут только один раз.

Запрос на объединение может объединять любое число исходных запросов.

Так, к предыдущему запросу можно добавить еще читателей, которые держат на руках книгу «Замок»:

```
UNION
SELECT READER.NAME_READER
FROM  READER, EXEMPLARE.BOOKS
WHERE EXEMPLARE.NUM_READER= READER.NUM_READER AND
EXEMPLRE.ISBN = BOOKS.ISBN AND
BOOKS.TITLE = "Замок"
```

В том случае, когда вам необходимо сохранить все строки из исходных отношений, необходимо использовать ключевое слово ALL в операции объединения. В случае сохранения дубликатов кортежей схема выполнения запроса на объединение будет выглядеть следующим образом:

```
SELECT – запрос
   UNION ALL
SELECT – запрос
   UNION ALL
SELECT – запрос
```

Однако тот же результат можно получить простым изменением фразы WHERE первой части исходного запроса, соединив локальные условия логической операцией ИЛИ и исключив дубликаты кортежей.

```
SELECT DISTINCT READER.NAME_READER
FROM  READER, EXEMPLARE.BOOKS
```



```
WHERE EXEMPLARE.NUM_READER= READER.NUM_READER AND
EXEMPLRE.ISBN = BOOKS.ISBN AND
BOOKS.TITLE = "Идиот" OR
BOOKS.TITLE = "Преступление и наказание" OR
BOOKS.TITLE = "Замок"
```

Ни один из исходных запросов в операции UNION не должен содержать предложения упорядочения результата ORDER BY, однако результат объединения может быть упорядочен, для этого предложение ORDER BY с указанием списка столбцов упорядочения записывается после текста последнего исходного SELECT-запроса.

Операторы манипулирования данными

В операции манипулирования данными входят три операции: операция удаления записей — ей соответствует оператор DELETE, операция добавления или ввода новых записей — ей соответствует оператор INSERT и операция изменения (обновления записей) — ей соответствует оператор UPDATE. Рассмотрим каждый из операторов подробнее.

Все операторы манипулирования данными позволяют изменить данные только в одной таблице.

Оператор ввода данных INSERT имеет следующий синтаксис:

```
INSERT INTO имя_таблицы [(<список столбцов>) ] VALUES (<список значений>)
```

Подобный синтаксис позволяет ввести только одну строку в таблицу. Задание списка столбцов необязательно тогда, когда мы вводим строку с заданием значений всех столбцов. Например, введем новую книгу в таблицу BOOKS

```
INSERT INTO BOOKS ( ISBN,TITL,AUTOR,COAUTOR,YEARIZD,PAGES)
VALUES ("5-88782-290-2","Аппаратные средства IBM PC. Энциклопедия",
   "Гук М. ","",2000,816)
```

В этой книге только один автор, нет соавторов, но мы в списке столбцов задали столбец COAUTOR, поэтому мы должны были ввести соответствующее значение в разделе VALUES. Мы ввели пустую строку, потому что мы знаем точно, что нет соавтора. Мы могли бы ввести *неопределенное* значение NULL.

Так как мы вводим полную строку, то мы можем не задавать список столбцов, ограничиться только заданием перечня значений, в этом случае оператор ввода будет выглядеть следующим образом:

```
INSERT INTO BOOKS VALUES ("5-88782-290-2",
   "Аппаратные средства IBM PC. Энциклопедия","Гук М.","",2000,816)
```

Результаты работы обоих операторов одинаковые.

Наконец, мы можем ввести неполный перечень значений, то есть не вводить соавтора, так как он отсутствует для данного издания. Но в этом случае мы должны

задать список вводимых столбцов, тогда оператор ввода будет выглядеть следующим образом:

```
INSERT INTO BOOKS ( ISBN,TITL,AUTOR,YEARIZD,PAGES)
VALUES ("5-88782-290-2","Аппаратные средства IBM PC. Энциклопедия",
   "Гук М.",2000,816)
```

Столбцу COAUTOR будет присвоено в этом случае значение NULL.

Какие столбцы должны быть заданы при вводе данных? Это определяется тем, как описаны эти столбцы при описании соответствующей таблицы, и будет рассмотрено более подробно при описании языка DDL (Data Definition Language) в главе 8. Здесь мы пока отметим, что если столбец или атрибут имеет признак обязательный (NOT NULL) при описании таблицы, то оператор INSERT должен обязательно содержать данные для ввода в каждую строку данного столбца. Поэтому если в таблице все столбцы обязательные, то каждая вводимая строка должна содержать полный перечень вводимых значений, а указание имен столбцов в этом случае необязательно. В противном случае, если имеется хотя бы один необязательный столбец и вы не вводите в него значений, задание списка имен столбцов — обязательно.

В набор значений могут быть включены специальные функции и выражения. Ограничением здесь является то, что значения этих функций должны быть определены на момент ввода данных. Поэтому, например, мы можем сформировать оператор ввода данных в таблицу EXEMPLAR следующим образом:

```
INSERT INTO EXEMPLAR (INV,ISBN,YES_NO,NUM_READER,DATE_IN, DATE_OUT)
VALUES (1872, "5-88782-290-2",NO,344,GetDate(),DateAdd(d,GetDate(),14))
```

И это означает, что мы выдали экземпляр книги с инвентарным номером 1872 читателю с номером читательского билете 344, отметив, что этот экземпляр не присутствует с этого момента в библиотеке, и определили дату выдачи книги как текущую дату (функция GetDate()), а дату возврата задали двумя неделями позднее, использовав при этом функцию DateAdd (), которая позволяет к одной дате добавить заданное количество интервалов даты и тем самым получить новое значение типа «дата». Мы добавили 14 дней к текущей дате.

Оператор ввода данных позволяет ввести сразу множество строк, если их можно выбрать из некоторой другой таблицы. Допустим, что у нас есть таблица со студентами и в ней указаны основные данные о студентах: их фамилии, адреса, домашние телефоны и даты рождения. Тогда мы можем сделать всех студентов читателями нашей библиотеки одним оператором:

```
INSERT INTO READER (NAME_READER, ADRESS, HOOM_PHONE, BIRTH_DAY)
   SELECT (NAME_STUDENT, ADRESS, HOOM_PHONE, BIRTH_DAY)
   FROM STUDENT
```

При этом номер читательского билета может назначаться автоматически, поэтому мы не вводим значения этого столбца в таблицу. Кроме того, мы предполагаем, что у студентов дневного отделения еще нет работы и поэтому нет рабочего телефона, и мы его не вводим.

Оператор удаления данных позволяет удалить одну или несколько строк из таблицы в соответствии с условиями, которые задаются для удаляемых строк.

Синтаксис оператора DELETE следующий:

```
DELETE FROM имя_таблицы [WHERE условия_отбора]
```

Если условия отбора не задаются, то из таблицы удаляются все строки, однако это не означает, что удаляется вся таблица. Исходная таблица остается, но она остается пустой, незаполненной.

Например, если нам надо удалить результаты прошедшей сессии, то мы можем удалить все строки из отношения R1 командой

```
DELETE FROM R1
```

Условия отбора в части WHERE имеют тот же вид, что и условия фильтрации в операторе SELECT. Эти условия определяют, какие строки из исходного отношения будут удалены. Например, если мы исключим студента Миронова А. В., то мы должны написать следующую команду:

```
DELETE FROM R2
   WHERE ФИО = 'Миронов А.В.'
```

В части WHERE может находиться встроенный запрос. Например, если нам надо исключить неуспевающих студентов, то по закону о высшем образовании неуспевающим считается студент, имеющий две и более задолженности по последней сессии. Тогда нам в условиях отбора надо найти студентов, имеющих либо две или более двоек, либо два и более несданных экзамена из числа тех, которые студент сдавал. Для поиска таких горе-студентов нам надо выбрать из отношения R1 все строки с оценкой 2 или с неопределенным значением, потом надо сгруппировать полученный результат по атрибуту ФИО и, подсчитав количество строк в каждой группе, которое соответствует количеству несданных экзаменов каждым студентом, отобрать те группы, у которых количество строк не менее двух. Теперь попробуем просто записать эту сложную конструкцию на SQL и убедимся, что этот сложный запрос записывается достаточно компактно.

```
DELETE FROM R2
   WHERE R2.ФИО IN
      (SELECT R1.ФИО
      FROM R1
      WHERE Оценка = 2 OR Оценка IS NULL
      GROOP BY R1.ФИО
      HAVING COUNT(*) >= 2
```

Однако при выполнении операции DELETE, включающей сложный подзапрос, в подзапросе нельзя упоминать таблицу, из которой удаляются строки, поэтому СУБД отвергнет такой красивый подзапрос, который попытается удалить всех не только сдававших, но и несдававших студентов, которые имеют более двух задолженностей.

```
DELETE FROM R2
WHERE R2.ФИО IN
```

```
(SELECT R1.ФИО
   FROM (R2 NATURAL INNER JOIN R3 ) LEFT JOIN R1 USING ( ФИО. Дисциплина)
   WHERE Оценка = 2 OR Оценка IS NULL
   GROOP BY R1.ФИО
   HAVING COUNT(*) >= 2
```

Все операции манипулирования данными связаны с понятием целостности базы данных, которое будет рассматриваться далее в главе 9. В настоящий момент мне бы хотелось отметить только то, что операции манипулирования данными не всегда выполнимы, даже если синтаксически они написаны правильно. Действительно, если мы бы захотели удалить какую-нибудь группу из отношения R3, то СУБД не позволила бы нам это сделать, так как в отношениях R1 и R2 есть строки, связанные с удаляемой строкой в отношении R3. Почему так делается, мы узнаем позднее, а пока просто примем к сведению, что не все операторы манипулирования выполнимы.

Операция обновления данных UPDATE требуется тогда, когда происходят изменения во внешнем мире и их надо адекватно отразить в базе данных, так как надо всегда помнить, что база данных отражает некоторую предметную область. Например, в нашем учебном заведении произошло счастливое событие, которое связано с тем, что госпожа Степанова К. Е. пересдала экзамен по дисциплине «Базы данных» с двойки сразу на четверку. В этом случае нам надо срочно выполнить соответствующую корректировку таблицы R1. Операция обновления имеет следующий формат:

```
UPDATE имя_таблицы
   SET имя_столбца = новое_значение
   [WHERE условие_отбора]
```

Часть WHERE является необязательной, так же как и в операторе DELETE. Она играет здесь ту же роль, что и в операторе DELETE, — позволяет отобрать строки, к которым будет применена операция модификации. Если условие отбора не задается, то операция модификации будет применена ко всем строкам таблицы.

Для решения ранее поставленной задачи нам необходимо выполнить следующую операцию

```
UPDATE R1
   SET R1.Оценка = 4
   WHERE R1.ФИО = "Степанова К.Е." AND R1.Дисциплина = "Базы данных"
```

В каких случаях требуется провести изменение в нескольких строках? Это не такая уж редкая задача. Например, если мы расширим нашу учебную базу данных еще одним отношением, которое содержит перечень курсов, на которых учатся наши студенты, то можно с помощью операции обновления промоделировать операцию перевода групп на следующий курс. Пусть новое отношение R4 имеет следующую схему:

R_4 = <Группа, Курс>

R4	
Группа	**Курс**
4906	3
4807	4

В этом случае перевод на следующий курс можно выполнить следующей операцией обновления:

```
UPDATE R4
   SET R4.Курс = R4.Курс + 1
```

И результат будет выглядеть следующим образом:

Группа	Курс
4906	4
4807	5

Операция модификации, так же как и операция удаления, может использовать сложные подзапросы. Расширим нашу базу еще одним отношением, которое будет содержать перечень студентов, получающих стипендию с указанием надбавки, которую они получают за отличную учебу. Исходно там могут находиться все студенты с указанием неопределенного размера стипендии. По мере анализа отношения R1 мы можем постепенно заменять неопределенные значения на конкретные размеры стипендии. Отношение R5 имеет вид:

R5		
ФИО	**Группа**	**Стипендия**
Петров Ф. И.	4906	<Null>
Сидоров К. А.	4906	<Null>
Миронов А. В.	4906	<Null>
Крылова Т. С.	4906	<Null>
Владимиров В. А.	4906	<Null>
Трофимов П. А.	4807	<Null>
Иванова Е. А.	4807	<Null>
Уткина Н. В.	4807	<Null>

Будем считать наличие трех пятерок по сессии признаком повышенной стипендии, + 50% к основной, наличие двух пятерок из сданных экзаменов и отсутствие двоек и троек на сданных экзаменах — признаком повышения стипендии на 25%, наличие хотя бы одной двойки среди сданных экзаменов — признаком снятия или отсутствия стипендии вообще, то есть -100% надбавки. При отсутствии троек на сданных экзаменах назначим обычную стипендию с надбавкой 0%. Од-

нако все эти изменения мы должны будем сделать отдельными операциями обновления.

Назначение повышенной стипендии:

```
UPDATE R5
   SET R5.Стипендия = 50%
   WHERE R5.ФИО IN
      (SELECT R1.ФИО
      FROM R1
      WHERE R1.Оценка = 5
      GROOP BY R1.ФИО
      HAVING COUNT(*) =3 )
```

Назначение стипендии с надбавкой 25%:

```
UPDATE R5
   SET R5.Стипендия = 25%
   WHERE R5.ФИО IN
      (SELECT R1.ФИО
      FROM R1
      WHERE R1.ФИО NOT IN
         (SELECT A.ФИО
         FROM R1 A
         WHERE A.Оценка <=3 OR A.Оценка IS NULL)
         GROOP BY R1.ФИО
         HAVING COUNT(*)>=2 )
```

Назначение обычной стипендии:

```
UPDATE R5
   SET R5.Стипендия = 0%
   WHERE R5.ФИО IN
      (SELECT R1.ФИО
      FROM R1
      WHERE R1.Оценка >=4 AND R1.ФИО NOT IN
         (SELECT A.ФИО
         FROM R1 A
         WHERE A.Оценка <= 3 OR A.Оценка IS NULL) )
```

Снятие стипендии:

```
UPDATE R5
   SET R5.Стипендия = -100%
   WHERE R5.ФИО IN
```



```
      (SELECT R1.ФИО
      FROM R1
      WHERE R1.Оценка <= 2 OR R1.Оценка IS NULL)
```

Почему мы в первом запросе на обновление не использовали дополнительную проверку на отсутствие двоек, троек и несданных экзаменов, как мы сделали это при назначении следующих видов стипендии? Просто мы учли особенности нашей предметной области: у нас в соответствии с исходными данными не только 3 экзамена. Но если мы можем предположить, что число экзаменов может быть произвольным и изменяться от семестра к семестру, то нам надо изменить наш запрос. Запрос — это некоторый алгоритм решения конкретной задачи, которую мы формулируем заранее на естественном языке. И оттого, что наша задача решается всего одним оператором языка SQL, она не становится примитивной. Мощность языка SQL и состоит в том, что он позволяет одним предложением сформулировать ответы на достаточно сложные запросы, для реализации которых на традиционных языках понадобилось бы писать большую программу. Итак, подумаем, как нам надо изменить текст нашего запроса на обновление для назначения повышенной стипендии при любом количестве сданных экзаменов. Прежде всего, каждая группа может иметь свое число экзаменов в сессию, это зависит от специальности и учебного плана, по которому учится данная группа. Поэтому для каждого студента нам надо знать, сколько экзаменов он должен был сдавать и сколько экзаменов он сдал на пять, и в том случае, когда эти два числа равны, мы можем назначить ему повышенную стипендию.

Будем решать нашу задачу по шагам. В конечном счете нам все равно надо знать, сколько экзаменов должен сдавать каждый конкретный студент, поэтому сначала сосчитаем количество экзаменов, которые должна сдавать группа, в которой учится этот студент.

Это мы делать умеем, для этого надо сделать запрос SELECT над отношением R3, сгруппировав его по атрибуту Группа, и вывести для каждой группы количество дисциплин, по которым должны сдаваться экзамены. Если мы учтем, что в одной сессии по одной дисциплине не бывает более одного экзамена, то можно просто подсчитывать количество строк в каждой группе.

```
SELECT R3.Группа, Число_экзаменов = COUNT(*)
   FROM R3
   GROOP BY R3.Группа
```

Однако нам нужен не этот запрос, нам нужен запрос, в котором мы определяем для каждого студента количество экзаменов. Этот запрос мы должны строить по схеме встроенного запроса:

```
SELECT COUNT(*)
   FROM R3
   WHERE R2.Группа = R3.Группа
   GROOP BY R3.Группа
```

А почему мы здесь в части FROM не написали имя второго отношения R2? Мы имя этого отношения укажем для связи с вышестоящим запросом, когда будем формировать запрос полностью. Теперь попробуем сформулировать полностью запрос. Нам надо объединить отношения R1 и R2 по атрибуту ФИО, нам надо знать группу, в которой учится каждый студент, далее надо выбрать все строки с оценкой 5 и сгруппировать их по фамилии студента, сосчитав количество строк в каждой группе, а выбирать мы будем те группы, в которых число строк в группе равно числу строк во встроенном запросе, рассмотренном ранее, при условии равенства количества строк в группе результату подзапроса, который выводит только одно число.

```
SELECT R1.ФИО
   FROM R1.R2
   WHERE R1. ФИО = R2.ФИО AND R1.Оценка = 5
   GROOP BY R1.ФИО
   HAVING COUNT(*) = (SELECT COUNT(*)
      FROM R3
      WHERE R2.Группа = R3.Группа
      GROOP BY R3.Группа)
```

Ну а теперь нам осталась последняя простейшая операция: надо заменить старый вложенный запрос, определявший отличников, получивших три пятерки на сессии, на новый универсальный запрос:

```
UPDATE R5
   SET R5.Стипендия = 50%
   WHERE R5.ФИО IN (SELECT R1.ФИО
   FROM R1.R2
      WHERE R1. ФИО = R2.ФИО AND R1.Оценка = 5
      GROOP BY R1.ФИО
      HAVING COUNT(*) = (SELECT COUNT(*)
         FROM R3
         WHERE R2.Группа = R3.Группа
         GROOP BY R3.Группа))
```

Вот какой сложный запрос мы построили. Это ведь практически один оператор, а какую сложную задачу он решает. Действительно, мощность языка SQL иногда удивляет даже профессионалов, кажется невозможно построить один запрос для решения конкретной задачи, но когда начинаешь поэтапно его конструировать — все получается. Самое сложное — это сделать переход от словесной формулировки задачи к представлению ее в терминах нашего SQL, но этот процесс сродни процессу алгоритмизации при решении задач традиционного программирования, а он всегда был самым трудным, творческим и неформализуемым процессом. Недаром на заре развития программирования известный американский специалист по программированию Дональд Е. Кнут озаглавил свой многотомный капитальный труд по теории и практике программирования «Искусство программирования для ЭВМ» («The art of computer programming»).

Задания для самостоятельной работы

Задание 1. В отношении R5 отметить студентов — претендентов на отчисление. Считаем, что в отношении R1 находятся окончательные результаты сессии, и поэтому отчислению подлежат все студенты, которые не сдали или не сдавали два и более из положенных экзаменов в сессию. Для того чтобы зафиксировать этот факт, нам потребуется добавить еще один столбец в отношение R5, назовем его результат_сессии, и там могут быть два допустимых значения: переведен на следующий курс или отчислен. Запрос писать по универсальному алгоритму.

Задание 2. В отношении R5 отметить студентов, переведенных на следующий курс.

Задание 3. Провести отчисление студентов по результатам текущей сессии. Обратите внимание, что это уже другая операция по сравнению с заданиями 1 и 2.

ГЛАВА 6 Проектирование реляционных БД на основе принципов нормализации

Что такое проект? Это схема — эскиз некоторого устройства, который в дальнейшем будет воплощен в реальность. Что такое проект реляционной базы данных? Это набор взаимосвязанных отношений, в которых определены все атрибуты, заданы первичные ключи отношений и заданы еще некоторые дополнительные свойства отношений, которые относятся к принципам поддержки целостности и будут более подробно рассмотрены в главе 9. Почему именно взаимосвязанных отношений? Потому что при выполнении запросов мы производим объединение отношений и одни и те же значения должны в разных отношениях-таблицах обозначаться одинаково. Действительно, если мы в одной таблице оценки будем обозначать цифрами, а в другой словами «отлично», «хорошо» и т. д., то мы не сможем объединить эти таблицы по столбцу Оценка, хотя по смыслу это для нас одно и то же, но то, что интуитивно понятно человеку, совсем не понятно «умному» компьютеру. Это проблема систем с искусственным интеллектом, которые могут решать весьма сложные интеллектуальные задачи, трудные для рядового инженера, но иногда пасуют перед простейшими интуитивными ассоциациями, понятными любому школьнику. И это необходимо учитывать. Поэтому проект базы данных должен быть очень точен и выверен. Фактически проект базы данных — это фундамент будущего программного комплекса, который будет использоваться достаточно долго и многими пользователями. И как в любом здании, можно достраивать мансарды, переделывать крышу, можно даже менять окна, но заменить фундамент, не разрушив всего здания, невозможно. Этапы жизненного цикла базы данных изображены на рис. 6.1. Они аналогичны, в основном, развитию любой программной системы, однако в них есть определенная специфика, касающаяся только баз данных. Более подробно мы будем рассматривать этапы жизненного цикла БД в следующих разделах учебного пособия, потому что термины, которые мы вынуждены применять при этом описании, пока еще неизвестны нашим читателям.

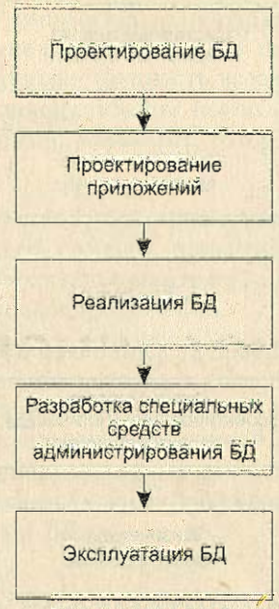


Рис. 6.1. Этапы жизненного цикла БД

Процесс проектирования БД представляет собой последовательность переходов от неформального словесного описания информационной структуры предметной области к формализованному описанию объектов предметной области в терминах некоторой модели. В общем случае можно выделить следующие этапы проектирования:

1. Системный анализ и словесное описание информационных объектов предметной области.
2. Проектирование инфологической модели предметной области — частично формализованное описание объектов предметной области в терминах некоторой семантической модели, например, в терминах Е-модели.
3. Даталогическое или логическое проектирование БД, то есть описание БД в терминах принятой даталогической модели данных.
4. Физическое проектирование БД, то есть выбор эффективного размещения БД на внешних носителях для обеспечения наиболее эффективной работы приложения.

Если мы учтем, что между вторым и третьим этапами необходимо принять решение, с использованием какой стандартной СУБД будет реализовываться наш проект, то условно процесс проектирования БД можно представить последовательностью выполнения пяти соответствующих этапов (см. рис. 6.2). Рассмотрим более подробно этапы проектирования БД.

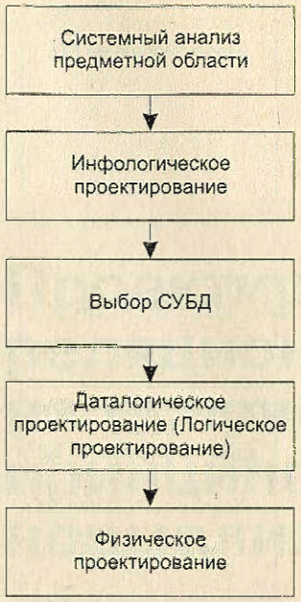


Рис. 6.2. Этапы проектирования БД

Системный анализ предметной области

С точки зрения проектирования БД в рамках системного анализа, необходимо осуществить первый этап, то есть провести подробное словесное описание объектов предметной области и реальных связей, которые присутствуют между описываемыми объектами. Желательно, чтобы данное описание позволяло корректно определить все взаимосвязи между объектами предметной области.

В общем случае существуют два подхода к выбору состава и структуры предметной области:

- *Функциональный подход* — он реализует принцип движения «от задач» и применяется тогда, когда заранее известны функции некоторой группы лиц и комплексов задач, для обслуживания информационных потребностей которых создается рассматриваемая БД. В этом случае мы можем четко выделить минимальный необходимый набор объектов предметной области, которые должны быть описаны.
- *Предметный подход* — когда информационные потребности будущих пользователей БД жестко не фиксируются. Они могут быть многоаспектными и весьма динамичными. Мы не можем точно выделить минимальный набор объектов предметной области, которые необходимо описывать. В описание предметной области в этом случае включаются такие объекты и взаимосвязи, которые наиболее характерны и наиболее существенны для нее. БД, конструируемая при этом, называется предметной, то есть она может быть использована при решении множества разнообразных, заранее не определенных задач. Конструирование предметной БД в некотором смысле кажется гораздо более заманчивым, однако трудность всеобщего охвата предметной области с невозможностью конкретизации потребностей пользователей может привести к избыточно сложной схеме БД, которая для конкретных задач будет неэффективной.

Чаще всего на практике рекомендуется использовать некоторый компромиссный вариант, который, с одной стороны, ориентирован на конкретные задачи или функциональные потребности пользователей, а с другой стороны, учитывает возможность наращивания новых приложений.

Системный анализ должен заканчиваться подробным описанием информации об объектах предметной области, которая требуется для решения конкретных задач и которая должна храниться в БД, формулировкой конкретных задач, которые будут решаться с использованием данной БД с кратким описанием алгоритмов их решения, описанием выходных документов, которые должны генерироваться в системе, описанием входных документов, которые служат основанием для заполнения данными БД.

Пример описания предметной области

Пусть требуется разработать информационную систему для автоматизации учета получения и выдачи книг в библиотеке. Система должна предусматривать режимы ведения системного каталога, отражающего перечень областей знаний, по которым имеются книги в библиотеке. Внутри библиотеки области знаний в систематическом каталоге могут иметь уникальный внутренний номер и полное наименование. Каждая книга может содержать сведения из нескольких областей знаний. Каждая книга в библиотеке может присутствовать в нескольких экземплярах. Каждая книга, хранящаяся в библиотеке, характеризуется следующими параметрами:

- уникальный шифр;
- название;
- фамилии авторов (могут отсутствовать);
- место издания (город);
- издательство;
- год издания;
- количество страниц;
- стоимость книги;
- количество экземпляров книги в библиотеке.

Книги могут иметь одинаковые названия, но они различаются по своему уникальному шифру (ISBN).

В библиотеке ведется картотека читателей.

На каждого читателя в картотеку заносятся следующие сведения:

- фамилия, имя, отчество;
- домашний адрес;
- телефон (будем считать, что у нас два телефона — рабочий и домашний);
- дата рождения.

Каждому читателю присваивается уникальный номер читательского билета.

Каждый читатель может одновременно держать на руках не более 5 книг. Читатель не должен одновременно держать более одного экземпляра книги одного названия.

Каждая книга в библиотеке может присутствовать в нескольких экземплярах. Каждый экземпляр имеет следующие характеристики:

- уникальный инвентарный номер;
- шифр книги, который совпадает с уникальным шифром из описания книг;
- место размещения в библиотеке.

В случае выдачи экземпляра книги читателю в библиотеке хранится специальный вкладыш, в котором должны быть записаны следующие сведения:

- номер билета читателя, который взял книгу;
- дата выдачи книги;
- дата возврата.

Предусмотреть следующие ограничения на информацию в системе:

1. Книга может не иметь ни одного автора.
2. В библиотеке должны быть записаны читатели не моложе 17 лет.
3. В библиотеке присутствуют книги, изданные начиная с 1960 по текущий год.
4. Каждый читатель может держать на руках не более 5 книг.
5. Каждый читатель при регистрации в библиотеке должен дать телефон для связи: он может быть рабочим или домашним.
6. Каждая область знаний может содержать ссылки на множество книг, но каждая книга может относиться к различным областям знаний.

С данной информационной системой должны работать следующие группы пользователей:

- библиотекари;
- читатели;
- администрация библиотеки.

При работе с системой библиотекарь должен иметь возможность решать следующие задачи:

1. Принимать новые книги и регистрировать их в библиотеке.
2. Относить книги к одной или к нескольким областям знаний.

3. Проводить каталогизацию книг, то есть назначение новых инвентарных номеров вновь принятым книгам, и, помещая их на полки библиотеки, запоминать место размещения каждого экземпляра.
4. Проводить дополнительную каталогизацию, если поступило несколько экземпляров книги, которая уже есть в библиотеке, при этом информация о книге в предметный каталог не вносится, а каждому новому экземпляру присваивается новый инвентарный номер и для него определяется место на полке библиотеки.
5. Проводить списание старых и не пользующихся спросом книг. Списывать можно только книги, ни один экземпляр которых не находится у читателей. Списание проводится по специальному акту списания, который утверждается администрацией библиотеки.
6. Вести учет выданных книг читателям, при этом предполагается два режима работы: выдача книг читателю и прием от него возвращаемых им книг обратно в библиотеку. При выдаче книг фиксируется, когда и какой экземпляр книги был выдан данному читателю и к какому сроку читатель должен вернуть этот экземпляр книги. При выдаче книг наличие свободного экземпляра и его конкретный номер могут определяться по заданному уникальному шифру книги или инвентарный номер может быть известен заранее. Не требуется вести «историю» чтения книг, то есть требуется отражать только текущее состояние библиотеки. При приеме книги, возвращаемой читателем, проверяется соответствие возвращаемого инвентарного номера книги выданному инвентарному номеру, и она ставится на свое старое место на полку библиотеки.
7. Проводить списание утерянных читателем книг по специальному акту списания или замены, подписанному администрацией библиотеки.
8. Проводить закрытие абонемента читателя, то есть уничтожение данных о нем, если читатель хочет выписаться из библиотеки и не является ее должником, то есть за ним не числится ни одной библиотечной книги.

Читатель должен иметь возможность решать следующие задачи:

1. Просматривать системный каталог, то есть перечень всех областей знаний, книги по которым есть в библиотеке.
2. По выбранной области знаний получить полный перечень книг, которые числятся в библиотеке.
3. Для выбранной книги получить инвентарный номер свободного экземпляра книги или сообщение о том, что свободных экземпляров книги нет. В случае отсутствия свободных экземпляров книги читатель должен иметь возможность узнать дату ближайшего предполагаемого возврата экземпляра данной книги. Читатель не может узнать данные о том, у кого в настоящий момент экземпляры данной книги находятся на руках (в целях обеспечения личной безопасности держателей требуемой книги).
4. Для выбранного автора получить список книг, которые числятся в библиотеке.

Администрация библиотеки должна иметь возможность получать сведения о должниках—читателях библиотеки, которые не вернули вовремя взятые книги; сведения о книгах, которые не являются популярными, т. е. ни один экземпляр

которых не находится на руках у читателей; сведения о стоимости конкретной книги, для того чтобы установить возможность возмещения стоимости утерянной книги или возможность замены ее другой книгой; сведения о наиболее популярных книгах, то есть таких, все экземпляры которых находятся на руках у читателей.

Этот совсем небольшой пример показывает, что перед началом разработки необходимо иметь точное представление о том, что же должно выполняться в нашей системе, какие пользователи в ней будут работать, какие задачи будет решать каждый пользователь. И это правильно, ведь когда мы строим здание, мы тоже заранее предполагаем: для каких целей оно предназначено, в каком климате оно будет стоять, на какой почве, и в зависимости от этого проектировщики могут предложить нам тот или иной проект. Но, к сожалению, очень часто по отношению к базам данных считается, что все можно определить потом, когда проект системы уже создан. Отсутствие четких целей создания БД может свести на нет все усилия разработчиков, и проект БД получится «плохим», неудобным, не соответствующим ни реально моделируемому объекту, ни задачам, которые должны решаться с использованием данной БД.

Отложим на время рассмотрение этапа инфологического моделирования предметной области — этому серьезному вопросу будет посвящена следующая, седьмая глава, а мы пойдем классическим путем и рассмотрим сначала этап даталогического проектирования. Напомним, что этап даталогического проектирования происходит уже после выбора конкретной модели данных. И мы рассматриваем даталогическое проектирование для реляционной модели данных.

Даталогическое проектирование

В реляционных БД даталогическое или логическое проектирование приводит к разработке схемы БД, то есть совокупности схем отношений, которые адекватно моделируют абстрактные объекты предметной области и семантические связи между этими объектами. Основой анализа корректности схемы являются так называемые функциональные зависимости между атрибутами БД. Некоторые зависимости между атрибутами отношений являются нежелательными из-за побочных эффектов и аномалий, которые они вызывают при модификации БД. При этом под процессом модификации БД мы понимаем внесение новых данных в БД или удаление некоторых данных из БД, а также обновление значений некоторых атрибутов.

Однако этап логического или даталогического проектирования не заканчивается проектированием схемы отношений. В общем случае в результате выполнения этого этапа должны быть получены следующие результирующие документы:

- Описание концептуальной схемы БД в терминах выбранной СУБД.
- Описание внешних моделей в терминах выбранной СУБД.
- Описание декларативных правил поддержки целостности базы данных.
- Разработка процедур поддержки семантической целостности базы данных.

- Однако перед тем как описывать построенную схему в терминах выбранной СУБД, нам надо выстроить эту схему. Именно этому процессу и посвящен данный раздел.
- Мы должны построить корректную схему БД, ориентируясь на реляционную модель данных.

ОПРЕДЕЛЕНИЕ

Корректной назовем схему БД, в которой отсутствуют нежелательные зависимости между атрибутами отношений.

Процесс разработки корректной схемы реляционной БД называется *логическим проектированием БД*.

Проектирование схемы БД может быть выполнено двумя путями:

- *путем декомпозиции* (*разбиения*), когда исходное множество отношений, входящих в схему БД заменяется другим множеством отношений (число их при этом возрастает), являющихся проекциями исходных отношений;
- *путем синтеза*, то есть путем компоновки из заданных исходных элементарных зависимостей между объектами предметной области схемы БД.

Классическая технология проектирования реляционных баз данных связана с теорией нормализации, основанной на анализе функциональных зависимостей между атрибутами отношений. Понятие функциональной зависимости является фундаментальным в теории нормализации реляционных баз данных. Мы определим его далее, а пока коснемся смысла этого понятия. Функциональные зависимости определяют устойчивые отношения между объектами и их свойствами в рассматриваемой предметной области. Именно поэтому процесс поддержки функциональных зависимостей, характерных для данной предметной области, является базовым для процесса проектирования.

Процесс проектирования с использованием декомпозиции представляет собой процесс последовательной нормализации схем отношений, при этом каждая последующая итерация соответствует нормальной форме более высокого уровня и обладает лучшими свойствами по сравнению с предыдущей.

Каждой нормальной форме соответствует некоторый определенный набор ограничений, и отношение находится в некоторой нормальной форме, если удовлетворяет свойственному ей набору ограничений.

В теории реляционных БД обычно выделяется следующая последовательность нормальных форм:

- первая нормальная форма (1NF);
- вторая нормальная форма (2NF);
- третья нормальная форма (3NF);
- нормальная форма Бойса—Кодда (BCNF);
- четвертая нормальная форма (4NF);
- пятая нормальная форма, или форма проекции-соединения (5NF или PJNF).

Основные свойства нормальных форм:

- каждая следующая нормальная форма в некотором смысле улучшает свойства предыдущей;
- при переходе к следующей нормальной форме свойства предыдущих нормальных форм сохраняются.

В основе классического процесса проектирования лежит последовательность переходов от предыдущей нормальной формы к последующей. Однако в процессе декомпозиции мы сталкиваемся с проблемой *обратимости*, то есть возможности восстановления исходной схемы. Таким образом, декомпозиция должна сохранять *эквивалентность* схем БД при замене одной схемы на другую.

ОПРЕДЕЛЕНИЕ

Схемы БД называются *эквивалентными*, если содержание исходной БД может быть получено путем естественного соединения отношений, входящих в результирующую схему, и при этом не появляется новых кортежей в исходной БД.

При выполнении эквивалентных преобразований сохраняется множество исходных фундаментальных функциональных зависимостей между атрибутами отношений.

Функциональные зависимости определяют не текущее состояние БД, а все возможные ее состояния, то есть они отражают те связи между атрибутами, которые присущи реальному объекту, который моделируется с помощью БД.

Поэтому определить функциональные зависимости по текущему состоянию БД можно только в том случае, если экземпляр БД содержит абсолютно полную информацию (то есть никаких добавлений и модификации БД не предполагается). В реальной жизни это требование невыполнимо, поэтому набор функциональных зависимостей задает разработчик, системный аналитик, исходя из глубокого системного анализа предметной области.

Приведем ряд основных определений.

Функциональной зависимостью набора атрибутов B отношения R от набора атрибутов A того же отношения, обозначаемой как

R.A -> R.B или A -> B

называется такое соотношение проекций R[A] и R[B], при котором в каждый момент времени любому элементу проекции R[A] соответствует только один элемент проекции R[B] , входящий вместе с ним в какой-либо кортеж отношения R.

Функциональная зависимость R.A -> R.B называется *полной*, если набор атрибутов B функционально зависит от A и не зависит функционально от любого подмножества A, то есть

R.A -> R.B называется полной, если:

$\forall A1 \subseteq A \Rightarrow R.A \not\rightarrow R.B$,

что читается следующим образом:

для любого A1, являющегося подмножеством A, R.B функционально не зависит от R.A, в противном случае зависимость R.A -> R.B называется неполной.

Функциональная зависимость R.A -> R.B называется *транзитивной*, если существует набор атрибутов C такой, что:

1. C не является подмножеством A.
2. C не включает в себя B.
3. Существует функциональная зависимость R.A -> R.C.
4. Не существует функциональной зависимости R.C -> R.A.
5. Существует функциональная зависимость R.C -> R.B.

Возможным ключом отношения называется набор атрибутов отношения, который полностью и однозначно (функционально полно) определяет значения всех остальных атрибутов отношения, то есть возможный ключ — это набор атрибутов, однозначно определяющий кортеж отношения, и при этом при удалении любого атрибута из этого набора его свойство однозначной идентификации кортежа теряется.

А может ли быть ситуация, когда отношение не имеет возможного ключа? Давайте вспомним определение отношения: *отношение — это подмножество декартова произведения множества доменов*. И в полном декартовом произведении все наборы значений различны, тем более в его подмножестве. Значит, обязательно для каждого отношения всегда существует набор атрибутов, по которому можно однозначно определить кортеж отношения. В вырожденном случае это просто полный набор атрибутов отношения, потому что если мы зададим для всех атрибутов конкретные значения, то, по определению отношения, мы получим только один кортеж.

В общем случае в отношении может быть несколько возможных ключей.

Среди всех возможных ключей отношения обычно выбирают один, который считается главным и который называют *первичным ключом отношения*.

Неключевым атрибутом называется любой атрибут отношения, не входящий в состав ни одного возможного ключа отношения.

Взаимно-независимые атрибуты — это такие атрибуты, которые не зависят функционально один от другого.

Если в отношении существует несколько функциональных зависимостей, то каждый атрибут или набор атрибутов, от которого зависит другой атрибут, называется *детерминантом* отношения.

Для функциональных зависимостей как фундаментальной основы проекта БД были проведены исследования, позволяющие избежать избыточного их представления. Ряд зависимостей могут быть выведены из других путем применения правил, названных аксиомами Армстронга, по имени исследователя, впервые сформулировавшего их. Это три основных аксиомы:

1. *Рефлексивность*: если B является подмножеством A, то A->B
2. *Дополнение*: если A->B , то AC->BC
3. *Транзитивность*: если A->B и B->C , то A->C.

Доказано, что данные правила являются полными и исчерпывающими, то есть, применяя их, из заданного множества функциональных зависимостей можно вывести все возможные функциональные зависимости.

Множество всех возможных функциональных зависимостей, выводимое из заданного набора исходных функциональных зависимостей, называется его замыканием.

ОПРЕДЕЛЕНИЕ

Отношение находится в первой нормальной форме тогда и только тогда, когда на пересечении каждого столбца и каждой строки находятся только элементарные значения атрибутов.

В некотором смысле это определение избыточно, потому что собственно оно определяет само отношение в теории реляционных баз данных. Однако в силу исторически сложившихся обстоятельств и для преемственности такое определение первой нормальной формы существует и мы должны с ним согласиться. Отношения, находящиеся в первой нормальной форме, часто называют просто нормализованными отношениями. Соответственно, ненормализованные отношения могут интерпретироваться как таблицы с неравномерным заполнением, например таблица «Расписание», которая имеет вид:

Преподаватель	День недели	Номер пары	Название дисциплины	Тип занятий	Группа
Петров В. И.	Понед.	1	Теор. выч. проц.	Лекция	4906
	Вторник	1	Комп. графика	Лаб. раб.	4907
	Вторник	2	Комп. графика	Лаб. раб.	4906
Киров В. А.	Понед.	2	Теор. информ.	Лекция	4906
	Вторник	3	Пр-е на C++	Лаб. раб.	4907
	Вторник	4	Пр-е на C++	Лаб. раб.	4906
Серов А. А.	Понед.	3	Защита инф.	Лекция	4944
	Среда	3	Пр-е на VB	Лаб. раб.	4942
	Четверг	4	Пр-е на VB	Лаб. раб.	4922

Здесь на пересечении одной строки и одного столбца находится целый набор элементарных значений, соответствующих набору дней, перечню пар, набору дисциплин, по которым проводит занятия один преподаватель.

Для приведения отношения «Расписание» к первой нормальной форме необходимо дополнить каждую строку фамилией преподавателя.

ОПРЕДЕЛЕНИЕ

Отношение находится во второй нормальной форме тогда и только тогда, когда оно находится в первой нормальной форме и не содержит неполных функциональных зависимостей неключевых атрибутов от атрибутов первичного ключа.

Преподаватель	День недели	Номер пары	Название дисциплины	Тип занятий	Группа
Петров В. И	Понед.	1	Теор. выч. проц.	Лекция	4906
Петров В. И	Вторник	1	Комп. графика	Лаб. раб.	4907
Петров В. И	Вторник	2	Комп. графика	Лаб. раб.	4906
Киров В. А.	Понед.	2	Теор. информ.	Лекция	4906
Киров В. А.	Вторник	3	Пр-е на C++	Лаб. раб.	4907
Киров В. А.	Вторник	4	Пр-е на C++	Лаб. раб.	4906
Серов А. А.	Понед.	3	Защита инф.	Лекция	4944
Серов А. А.	Среда	3	Пр-е на VB	Лаб. раб.	4942
Серов А. А.	Четверг	4	Пр-е на VB	Лаб. раб.	4922

Рассмотрим отношение, моделирующее сдачу студентами текущей сессии. Структура этого отношения определяется следующим набором атрибутов:

```
(ФИО, Номер зач.кн., Группа, Дисциплина, Оценка)
```

Так как каждый студент сдает целый набор дисциплин в процессе сессии, то первичным ключом отношения может быть (Номер. зач.кн., Дисциплина), который однозначно определяет каждую стоку отношения. С другой стороны, атрибуты ФИО и Группа зависят только от части первичного ключа — от значения атрибута Номер зач. кн., поэтому мы должны констатировать наличие неполных функциональных зависимостей в данном отношении. Для приведения данного отношения ко второй нормальной форме следует разбить его на проекции, при этом должно быть соблюдено условие восстановления исходного отношения без потерь. Такими проекциями могут быть два отношения:

```
(ФИО, Номер.зач.кн., Группа)

(Номер зач.кн., Дисциплина, Оценка)
```

Этот набор отношений не содержит неполных функциональных зависимостей, и поэтому эти отношения находятся во второй нормальной форме.

А почему надо приводить отношения ко второй нормальной форме? Иначе говоря, какие аномалии или неудобства могут возникнуть, если мы оставим исходное отношение и не будем его разбивать на два? Давайте рассмотрим ситуацию, когда студент переведен из одной группы в другую. Тогда в первом случае (если мы не разбивали исходное отношение на два) мы должны найти все записи с данным студентом и в них изменить значение атрибута Группа на новое. Во втором же случае меняется только один кортеж в первом отношении. И конечно, опасность нарушения корректности (непротиворечивости содержания) БД в первом случае выше. Может получиться так, что часть кортежей поменяет значения атрибута Группа, а часть по причине сбоя в работе аппаратуры останется в старом состоянии. И тогда наша БД будет содержать записи, которые относят одного студента одновременно к разным группам. Чтобы этого не произошло, мы должны принимать дополнительные непростые меры, например организовывать процесс согласованного изменения с использованием сложного механизма

транзакций, который мы будем рассматривать в главах, посвященных вопросам распределенного доступа к БД. Если же мы перешли ко второй нормальной форме, то мы меняем только один кортеж. Кроме того, если у нас есть студенты, которые еще не сдавали экзамены, то в исходном отношении мы вообще не можем хранить о них информацию, а во второй схеме информация о студентах и их принадлежности к конкретной группе хранится отдельно от информации, которая связана со сдачей экзаменов, и поэтому мы можем в этом случае отдельно работать со студентами и отдельно хранить и обрабатывать информацию об успеваемости и сдаче экзаменов, что в действительности и происходит.

ОПРЕДЕЛЕНИЕ

Отношение находится в третьей нормальной форме тогда и только тогда, когда оно находится во второй нормальной форме и не содержит транзитивных зависимостей.

Рассмотрим отношение, связывающее студентов с группами, факультетами и специальностями, на которых он учится.

(ФИО, Номер зач.кн., Группа, Факультет, Специальность, Выпускающая кафедра)

Первичным ключом отношения является Номер зач.кн., однако рассмотрим остальные функциональные зависимости. Группа, в которой учится студент, однозначно определяет факультет, на котором он учится, а также специальность и выпускающую кафедру. Кроме того, выпускающая кафедра однозначно определяет факультет, на котором обучаются студенты, выпускаемые по данной кафедре. Но если мы предположим, что одну специальность могут выпускать несколько кафедр, то специальность не определяет выпускающую кафедру. В этом случае у нас есть следующие функциональные зависимости:

Номер зач.кн. -> ФИО
Номер зач.кн. -> Группа
Номер зач.кн. -> Факультет
Номер зач.кн. -> Специальность
Номер зач.кн. -> Выпускающая кафедра
Группа -> Факультет
Группа -> Специальность
Группа -> Выпускающая кафедра
Выпускающая кафедра -> Факультет

И эти зависимости образуют транзитивные группы. Для того чтобы избежать этого, мы можем предложить следующий набор отношений:

(Номер зач.кн., ФИО, Специальность, Группа)
(Группа, Выпускающая кафедра)
(Выпускающая кафедра, Факультет)

Первичные ключи отношений выделены.

Теперь необходимо удостовериться, что при естественном соединении мы не потеряем ни одной строки и не получим лишних кортежей. И это упражнение я предлагаю выполнить вам самостоятельно.

Полученный набор отношений находится в третьей нормальной форме.

ОПРЕДЕЛЕНИЕ

Отношение находится в нормальной форме Бойса—Кодда, если оно находится в третьей нормальной форме и каждый детерминант отношения является возможным ключом отношения.

Рассмотрим отношение, моделирующее сдачу студентом текущих экзаменов. Предположим, что студент может сдавать экзамен по одной дисциплине несколько раз, если он получил неудовлетворительную оценку. Допустим, что во избежание возможных полных однофамильцев мы можем однозначно идентифицировать студента номером его зачетной книги, но, с другой стороны, у нас ведется электронный учет текущей успеваемости студентов, поэтому каждому студенту присваивается в период его обучения в вузе уникальный номер-идентификатор. Отношение, которое моделирует сдачу текущей сессии, имеет следующую структуру:

(Номер зач.кн., Идентификатор_студента, Дисциплина, Дата, Оценка)

Возможными ключами отношения являются Номер_зач.кн, Дисциплина, Дата и Идентификатор_студента, Дисциплина, Дата.

Какие функциональные зависимости у нас имеются?

Номер_зач.кн. Дисциплина. Дата -> Оценка;
Идентификатор_студента. Дисциплина. Дата -> Оценка;
Номер зач.кн. -> Идентификатор_студента;
Идентификатор_студента -> Номер зач.кн.

- Откуда взялись две последние функциональные зависимости? Но ведь мы предварительно описали, что каждому студенту ставится в соответствие один номер зачетной книжки и один Идентификатор_студента, поэтому по значению Номер зач.кн. можно однозначно определить Идентификатор_студента (это третья зависимость) и обратно (и это четвертая зависимость). Оценим это отношение.
- Это отношение находится в третьей нормальной форме, потому что неполных функциональных зависимостей непервичных атрибутов от атрибутов возможного ключа здесь не присутствует и нет транзитивных зависимостей. А как же третья и четвертая зависимости, разве они не являются неполными? Нет, потому что зависимым не является непервичный атрибут, то есть атрибут, не входящий ни в один возможный ключ. Поэтому придраться к этому мы не можем. Но вот под четвертую нормальную форму наше отношение не подходит, потому что у нас есть два детерминанта Номер зач.кн. и Идентификатор_студента, которые не являются возможными ключами отношения. Для приведения отношения к нормальной форме Бойса—Кодда надо разделить отношение, например, на два со следующими схемами:

(Идентификатор_студента, Дисциплина, Дата, Оценка)

(Номер зач.кн., Идентификатор_студента)

или наоборот:

(Номер зач.кн., Дисциплина, Дата, Оценка)

(Номер зач.кн., Идентификатор_студента)

Эти схемы равнозначны с точки зрения теории нормализации, поэтому выбирать проектировщикам следует исходя из некоторых дополнительных рассуждений. Ну, например, если учесть, что зачетные книжки могут теряться, то как они будут восстанавливаться: если с тем же самым номером, то нет разницы, но если с новым номером, то тогда первая схема предпочтительней.

В большинстве случаев достижение третьей нормальной формы или даже формы Бойса—Кодда считается достаточным для реальных проектов баз данных, однако в теории нормализации существуют нормальные формы высших порядков, которые уже связаны не с функциональными зависимостями между атрибутами отношений, а отражают более тонкие вопросы семантики предметной области и связаны с другими видами зависимостей. Прежде чем перейти к рассмотрению нормальных форм высших порядков, дадим еще несколько определений.

ОПРЕДЕЛЕНИЕ

В отношении R (A, B, C) существует *многозначная зависимость* (*multi valid dependence, MVD*), R.A ->> R.B в том и только в том случае, если множество значений B, соответствующее паре значений A и C, зависит только от A и не зависит от C.

Когда мы рассматривали функциональные зависимости, то каждому значению детерминанта соответствовало только одно значение зависимого от него атрибута. При рассмотрении многозначных зависимостей мы выделяем случаи, когда одному значению некоторого атрибута соответствует устойчиво постоянное множество значений другого атрибута. Когда это может быть? Рассмотрим конкретную ситуацию, понятную всем студентам. Пусть дано отношение, которое моделирует предстоящую сдачу экзаменов на сессии. Допустим, оно имеет вид:

(Номер зач.кн., Группа, Дисциплина)

Перечень дисциплин, которые должен сдавать студент, однозначно определяется не его фамилией, а номером группы (то есть специальностью, на которой он учится).

В данном отношении существуют следующие две многозначные зависимости:

Группа ->> Дисциплина

Группа ->> Номер зач.кн.

Это означает, что каждой группе однозначно соответствует перечень дисциплин по учебному плану и номер группы определяет список студентов, которые в этой группе учатся.

Если мы будем работать с исходным отношением, то мы не сможем хранить информацию о новой группе и ее учебном плане — перечне дисциплин, которые

должна пройти группа до тех пор, пока в нее не будут зачислены студенты. При изменении перечня дисциплин по учебному плану, например при добавлении новой дисциплины, внести эти изменения в отношение для всех студентов, занимающихся в данной группе, весьма затруднительно. С другой стороны, если мы добавляем студента в уже существующую группу, то мы должны добавить множество кортежей, соответствующих перечню дисциплин для данной группы. Эти аномалии модификации отношения как раз и связаны с наличием двух многозначных зависимостей.

В теории реляционных баз данных доказывается, что в общем случае в отношении R (A, B, C) существует многозначная зависимость R.A ->> R.B в том и только в том случае, когда существует многозначная зависимость R.A ->> R.C.

Дальнейшая нормализация отношений, подобных нашему, основывается на теореме Фейджина.

ТЕОРЕМА ФЕЙДЖИНА

Отношение R (A, B, C) можно спроецировать без потерь в отношения R1 (A, B) и R2 (A, C) в том и только в том случае, когда существует MVD A ->> B | C (что равнозначно наличию двух зависимостей A ->> B и A ->> C).

Под проецированием без потерь понимается такой способ декомпозиции отношения путем применения операции проекции, при котором исходное отношение полностью и без избыточности восстанавливается путем естественного соединения полученных отношений. Практически теорема доказывает наличие эквивалентной схемы для отношения, в котором существует несколько многозначных зависимостей.

ОПРЕДЕЛЕНИЕ

Отношение R находится в четвертой нормальной форме (4NF) в том и только в том случае, если в случае существования многозначной зависимости A ->> B все остальные атрибуты R функционально зависят от A.

В нашем примере можно произвести декомпозицию исходного отношения в два отношения:

(Номер зач.кн., Группа)

(Группа, Дисциплина)

Оба эти отношения находятся в 4NF и свободны от отмеченных аномалий. Действительно, обе операции модификации теперь упрощаются: добавление нового студента связано с добавлением всего одного кортежа в первое отношение, а добавление новой дисциплины выливается в добавление одного кортежа во второе отношение, кроме того, во втором отношении мы можем хранить любое количество групп с определенным перечнем дисциплин, в которые пока еще не зачислены студенты.

Последней нормальной формой является пятая нормальная форма 5NF, которая связана с анализом нового вида зависимостей, *зависимостей «проекции соединения»* (*project-join зависимости*, обозначаемые как *PJ-зависимости*). Этот вид

зависимостей является в некотором роде обобщением многозначных зависимостей.

ОПРЕДЕЛЕНИЕ

Отношение R (X, Y, ..., Z) удовлетворяет зависимости соединения (X, Y, ..., Z) в том и только в том случае, когда R восстанавливается без потерь путем соединения своих проекций на X, Y, ..., Z. Здесь X, Y, ..., Z — наборы атрибутов отношения R.

Наличие PJ-зависимости в отношении делает его в некотором роде избыточным и затрудняет операции модификации.

ОПРЕДЕЛЕНИЕ

Отношение R находится *в пятой нормальной форме* (нормальной форме проекции-соединения — PJ/NF) в том и только в том случае, когда любая зависимость соединения в R следует из существования некоторого возможного ключа в R.

Рассмотрим отношение R1:

```
R1(Преподаватель, Кафедра, Дисциплина)
```

Предположим, что каждый преподаватель может работать на нескольких кафедрах и на каждой кафедре может вести несколько дисциплин. В этом случае ключом отношения является полный набор из трех атрибутов. В отношении отсутствуют многозначные зависимости, и поэтому отношение находится в 4NF.

Введем следующие обозначения наборов атрибутов:

```
ПК (Преподаватель, Кафедра)
ПД (Преподаватель, Дисциплина)
КД (Кафедра, Дисциплина)
```

Допустим, что отношение R1 удовлетворяет зависимости проекции соединения (ПК, ПД, КД). Тогда отношение R1 не находится в NF/PJ, потому что единственным ключом его является полный набор атрибутов, а наличие зависимости PJ связано с наборами атрибутов, которые не составляют возможные ключи отношения R1. Для того чтобы привести это отношение к NF/PJ, его надо представить в виде трех отношений:

```
R2 (Преподаватель, Кафедра)
R3 (Преподаватель, Дисциплина)
R4 (Кафедра, Дисциплина)
```

Пятая нормальная форма редко используется на практике. В большей степени она является теоретическим исследованием. Очень тяжело определить само наличие зависимостей «проекции—соединения», потому что утверждение о наличии такой зависимости делается для всех возможных состояний БД, а не только для текущего экземпляра отношения R1. Однако знание о возможном наличии подобных зависимостей, даже теоретическое, нам все же необходимо.

ГЛАВА 7 Инфологическое моделирование

Инфологическая модель применяется на втором этапе проектирования БД, то есть после словесного описания предметной области. Зачем нужна инфологическая модель и какую пользу она дает проектировщикам? Еще раз хотим напомнить, что процесс проектирования длительный, он требует обсуждений с заказчиком, со специалистами в предметной области. Наконец, при разработке серьезных корпоративных информационных систем проект базы данных является тем фундаментом, на котором строится вся система в целом, и вопрос о возможном кредитовании часто решается экспертами банка на основании именно грамотно сделанного инфологического проекта БД. Следовательно, инфологическая модель должна включать такое формализованное описание предметной области, которое легко будет «читаться» не только специалистами по базам данных. И это описание должно быть настолько емким, чтобы можно было оценить глубину и корректность проработки проекта БД, и конечно, как говорилось раньше, оно не должно быть привязано к конкретной СУБД. Выбор СУБД — это отдельная задача, для корректного ее решения необходимо иметь проект, который не привязан ни к какой конкретной СУБД.

Инфологическое проектирование прежде всего связано с попыткой представления семантики предметной области в модели БД. Реляционная модель данных в силу своей простоты и лаконичности не позволяет отобразить семантику, то есть смысл предметной области. Ранние теоретико-графовые модели в большей степени отображали семантику предметной области. Они в явном виде определяли иерархические связи между объектами предметной области.

Проблема представления семантики давно интересовала разработчиков, и в семидесятых годах было предложено несколько моделей данных, названных семантическими моделями. К ним можно отнести семантическую модель данных, предложенную *Хаммером* (*Hammer*) и *Мак-Леодом* (*McLeon*) в 1981 году, функциональную модель данных *Шипмана* (*Shipman*), также созданную в 1981 году, модель «сущность—связь», предложенную *Ченом* (*Chen*) в 1976 году, и ряд других моделей. У всех моделей были свои положительные и отрицательные стороны, но испытание временем выдержала только последняя. И в настоящий момент именно модель Чена «сущность—связь», или «Entity Relationship», стала

фактическим стандартом при инфологическом моделировании баз данных. Общепринятым стало сокращенное название ER-модель, большинство современных CASE-средств содержат инструментальные средства для описания данных в формализме этой модели. Кроме того, разработаны методы автоматического преобразования проекта БД из ER-модели в реляционную, при этом преобразование выполняется в даталогическую модель, соответствующую конкретной СУБД. Все CASE-системы имеют развитые средства документирования процесса разработки БД, автоматические генераторы отчетов позволяют подготовить отчет о текущем состоянии проекта БД с подробным описанием объектов БД и их отношений как в графическом виде, так и в виде готовых стандартных печатных отчетов, что существенно облегчает ведение проекта.

В настоящий момент не существует единой общепринятой системы обозначений для ER-модели и разные CASE-системы используют разные графические нотации, но разобравшись в одной, можно легко понять и другие нотации.

Модель «сущность—связь»

Как любая модель, модель «сущность—связь» имеет несколько базовых понятий, которые образуют исходные кирпичики, из которых строятся уже более сложные объекты по заранее определенным правилам.

Эта модель в наибольшей степени согласуется с концепцией объектно-ориентированного проектирования, которая в настоящий момент несомненно является базовой для разработки сложных программных систем, поэтому многие понятия вам могут показаться знакомыми, и если это действительно так, то тем проще вам будет освоить технологию проектирования баз данных, основанную на ER-модели.

В основе ER-модели лежат следующие базовые понятия:

- *Сущность*, с помощью которой моделируется класс однотипных объектов. Сущность имеет имя, уникальное в пределах моделируемой системы. Так как сущность соответствует некоторому классу однотипных объектов, то предполагается, что в системе существует множество экземпляров данной сущности. Объект, которому соответствует понятие сущности, имеет свой набор *атрибутов* — характеристик, определяющих свойства данного представителя класса. При этом набор атрибутов должен быть таким, чтобы можно было различать конкретные экземпляры сущности. Например, у сущности Сотрудник может быть следующий набор атрибутов: Табельный номер, Фамилия, Имя, Отчество, Дата рождения, Количество детей, Наличие родственников за границей. Набор атрибутов, однозначно идентифицирующий конкретный экземпляр сущности, называют *ключевым*. Для сущности Сотрудник ключевым будет атрибут Табельный номер, поскольку для всех сотрудников данного предприятия табельные номера будут различны. Экземпляром сущности Сотрудник будет описание конкретного сотрудника предприятия. Одно из общепринятых графических обозначений сущности — прямоугольник, в верхней части которого записано имя сущности, а ниже перечисляются атрибуты,

причем ключевые атрибуты помечаются, например, подчеркиванием или специальным шрифтом (рис. 7.1):

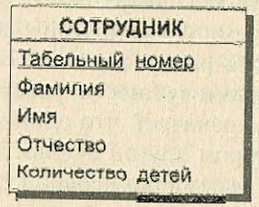


Рис. 7.1. Пример определения сущности в модели ER

Между сущностями могут быть установлены *связи* — бинарные ассоциации, показывающие, каким образом сущности соотносятся или взаимодействуют между собой. Связь может существовать между двумя разными сущностями или между сущностью и ей же самой (*рекурсивная связь*). Она показывает, как связаны экземпляры сущностей между собой. Если связь устанавливается между двумя сущностями, то она определяет взаимосвязь между экземплярами одной и другой сущности. Например, если у нас есть связь между сущностью «Студент» и сущностью «Преподаватель» и эта связь — руководство дипломными проектами, то каждый студент имеет только одного руководителя, но один и тот же преподаватель может руководить множеством студентов-дипломников. Поэтому это будет связь «один-ко-многим» (1:M), один со стороны «Преподаватель» и многие со стороны «Студент» (см. рис. 7.2).

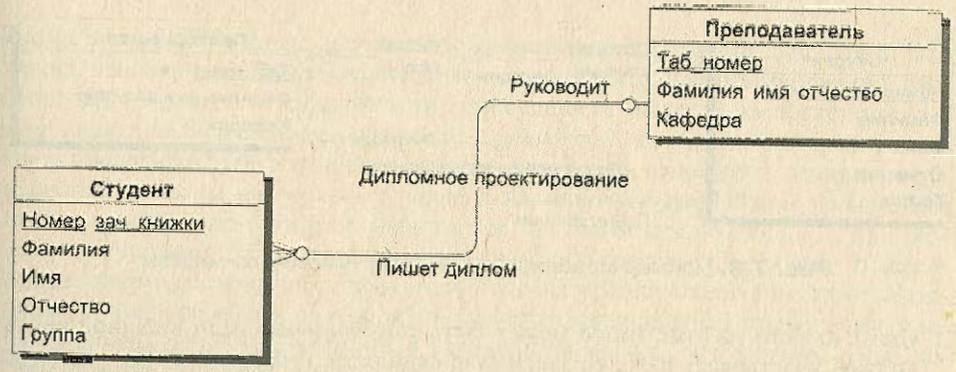


Рис. 7.2. Пример отношения «один-ко-многим» при связывании сущностей «Студент» и «Преподаватель»

В разных нотациях мощность связи изображается по-разному. В нашем примере мы используем нотацию CASE системы POWER DESIGNER, здесь множественность изображается путем разделения линии связи на 3. Связь имеет общее имя «Дипломное проектирование» и имеет имена ролей со стороны обеих сущностей. Со стороны студента эта роль называется «Пишет диплом под руководством», со стороны преподавателя эта связь называется «Руководит». Гра-

фическая интерпретация связи позволяет сразу прочитать смысл взаимосвязи между сущностями, она наглядна и легко интерпретируема. Связи делятся на три типа по множественности: *один-к-одному* (1:1), *один-ко-многим* (1:M), *многие-ко-многим* (M:M). Связь один-к-одному означает, что экземпляр одной сущности связан только с одним экземпляром другой сущности. Связь 1: M означает, что один экземпляр сущности, расположенный слева по связи, может быть связан с несколькими экземплярами сущности, расположенными справа по связи. Связь «один-к-одному» (1:1) означает, что один экземпляр одной сущности связан только с одним экземпляром другой сущности, а связь «многие-ко-многим» (M:M) означает, что один экземпляр первой сущности может быть связан с несколькими экземплярами второй сущности, и наоборот, один экземпляр второй сущности может быть связан с несколькими экземплярами первой сущности. Например, если мы рассмотрим связь типа «Изучает» между сущностями «Студент» и «Дисциплина», то это связь типа «многие-ко-многим» (M:M), потому что каждый студент может изучать несколько дисциплин, но и каждая дисциплина изучается множеством студентов. Такая связь изображена на рис. 7.3.

- Между двумя сущностями может быть задано сколько угодно связей с разными смысловыми нагрузками. Например, между двумя сущностями «Студент» и «Преподаватель» можно установить две смысловые связи, одна — рассмотренная уже ранее «Дипломное проектирование», а вторая может быть условно названа «Лекции», и она определяет, лекции каких преподавателей слушает данный студент и каким студентам данный преподаватель читает лекции. Ясно, что это связь типа *многие-ко-многим*. Пример этих связей приведен на рис. 7.3.

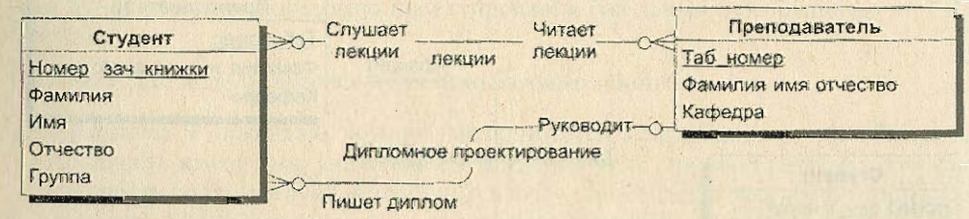


Рис. 7.3. Пример моделирования связи «многие-ко-многим»

- Связь любого из этих типов может быть *обязательной*, если в данной связи должен участвовать каждый экземпляр сущности, *необязательной* — если не каждый экземпляр сущности должен участвовать в данной связи. При этом связь может быть *обязательной с одной стороны* и *необязательной с другой стороны*. Обязательность связи тоже по-разному обозначается в разных нотациях. Мы снова используем нотацию POWER DESIGNER. Здесь необязательность связи обозначается пустым кружочком на конце связи, а обязательность перпендикулярной линией, перечеркивающей связь. И эта нотация имеет простую интерпретацию. Кружочек означает, что ни один экземпляр не может участвовать в этой связи. А перпендикуляр интерпретируется как то, что по крайней мере один экземпляр сущности участвует в этой связи.

Рассмотрим для этого ранее приведенный пример связи «Дипломное проектирование». На нашем рисунке эта связь интерпретируется как необязательная с двух сторон. Но ведь на самом деле каждый студент, который пишет диплом, должен иметь своего руководителя дипломного проектирования, но, с другой стороны, не каждый преподаватель должен вести дипломное проектирование. Поэтому в данной смысловой постановке изображение этой связи изменится и будет выглядеть таким, как представлено на рис. 7.4.

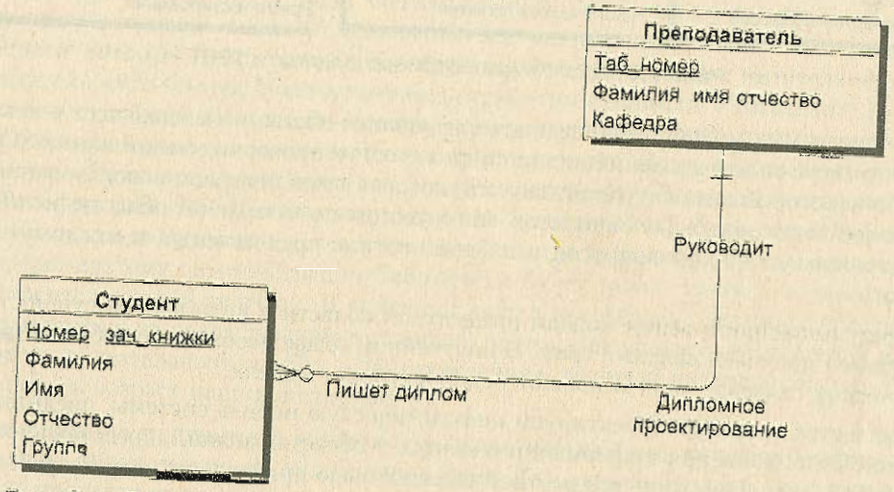


Рис. 7.4. Пример обязательной и необязательной связи между сущностями

Кроме того, в ER-модели допускается принцип категоризации сущностей. Это значит, что, как и в объектно-ориентированных языках программирования, вводится понятие подтипа сущности, то есть сущность может быть представлена в виде двух или более своих подтипов — *сущностей*, каждая из которых может иметь общие атрибуты и отношения и/или атрибуты и отношения, которые определяются однажды на верхнем уровне и наследуются на нижнем уровне. Все подтипы одной сущности рассматриваются как взаимоисключающие, и при разделении сущности на подтипы она должна быть представлена в виде полного набора взаимоисключающих подтипов. Если на уровне анализа не удается выявить полный перечень подтипов, то вводится специальный подтип, называемый условно ПРОЧИЕ, который в дальнейшем может быть уточнен. В реальных системах бывает достаточно ввести подтипизацию на двух-трех уровнях.

Сущность, на основе которой строятся подтипы, называется *супертипом*. Любой экземпляр супертипа должен относиться к конкретному подтипу. Для графического изображения принципа категоризации или типизации сущности вводится специальный графический элемент, называемый *узел-дискриминатор*, в нотации POWER DESIGNER он изображается в виде полукруга, выпуклой стороной обращенного к суперсущности. Эта сторона соединяется направленной стрелкой с суперсущностью, а к диаметру этого круга стрелками подсоединяются подтипы данной сущности (см. рис. 7.5).

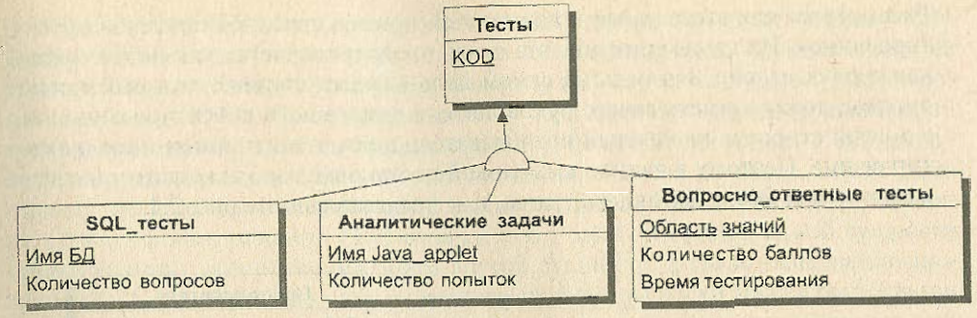


Рис. 7.5. Диаграмма подтипов сущности ТЕСТ

Эту диаграмму можно расшифровать следующим образом. Каждый тест в некоторой системе тестирования является либо тестом проверки знаний языка SQL, либо некоторой аналитической задачей, которая выполняется с использованием заранее написанных Java-апплетов, либо тестом по некоторой области знаний, состоящим из набора вопросов и набора ответов, предлагаемых к каждому вопросу.

В результате построения модели предметной области в виде набора сущностей и связей получаем связный граф. В полученном графе необходимо избегать циклических связей — они выявляют некорректность модели.

В качестве примера спроектируем инфологическую модель системы, предназначенной для хранения информации о книгах и областях знаний, представленных в библиотеке. Описание предметной области было приведено ранее. Разработку модели начнем с выделения основных сущностей.

Прежде всего, существует сущность «Книги», каждая книга имеет уникальный шифр, который является ее ключом, и ряд атрибутов, которые взяты из описания предметной области. Множество экземпляров сущности определяет множество книг, которые хранятся в библиотеке. Каждый экземпляр сущности «Книги» соответствует не конкретной книге, стоящей на полке, а описанию некоторой книги, которое дается обычно в предметном каталоге библиотеке. Каждая книга может присутствовать в нескольких экземплярах, и это как раз те конкретные книги, которые стоят на полках библиотеки. Для того чтобы отразить это, мы должны ввести сущность «Экземпляры», которая будет содержать описания всех экземпляров книг, которые хранятся в библиотеке. Каждый экземпляр сущности «Экземпляры» соответствует конкретной книге на полке. Каждый экземпляр имеет уникальный инвентарный номер, однозначно определяющий конкретную книгу. Кроме того, каждый экземпляр книги может находиться либо в библиотеке, либо на руках у некоторого читателя, и в последнем случае для данного экземпляра указываются дополнительно дата взятия книги читателем и дата предполагаемого возврата книги.

Между сущностями «Книги» и «Экземпляры» существует связь «один-ко-многим» (1:M), обязательная с двух сторон. Чем определяется данный тип связи? Мы можем предположить, что каждая книга может присутствовать в библиотеке в нескольких экземплярах, поэтому связь «один-ко-многим». При этом если в библиотеке нет ни одного экземпляра данной книги, то мы не будем хранить

ее описание, поэтому если книга описана в сущности «Книги», то по крайней мере один экземпляр этой книги присутствует в библиотеке. Это означает, что со стороны книги связь обязательная. Что касается сущности «Экземпляры», то не может существовать в библиотеке ни одного экземпляра, который бы не относился к конкретной книге, поэтому и со стороны «Экземпляры» связь тоже обязательная.

Теперь нам необходимо определить, как в нашей системе будет представлен читатель. Естественно предложить ввести для этого сущность «Читатели», каждый экземпляр которой будет соответствовать конкретному читателю. В библиотеке каждому читателю присваивается уникальный номер читательского билета, который будет однозначно идентифицировать нашего читателя. Номер читательского билета будет ключевым атрибутом сущности «Читатели». Кроме того, в сущности «Читатели» должны присутствовать дополнительные атрибуты, которые требуются для решения поставленных задач, этими атрибутами будут: «Фамилия Имя Отчество», «Адрес читателя», «Телефон домашний» и «Телефон рабочий». Почему мы ввели два отдельных атрибута под телефоны? Потому что надо в разное время звонить по этим телефонам, чтобы застать читателя, поэтому администрации библиотеки будет важно знать, к какому типу относится данный телефон. В описании нашей предметной области существует ограничение на возраст наших читателей, поэтому в сущности «Читатели» надо ввести обязательный атрибут «Дата рождения», который позволит нам контролировать возраст наших читателей.

Из описания предметной области мы знаем, что каждый читатель может держать на руках несколько экземпляров книг. Для отражения этой ситуации нам надо провести связь между сущностями «Читатели» и «Экземпляры». А почему не между сущностями «Читатели» и «Книги»? Потому что читатель берет из библиотеки конкретный экземпляр конкретной книги, а не просто книгу. А как же узнать, какая книга у данного читателя? А это можно будет узнать по дополнительной связи между сущностями «Экземпляры» и «Книги», и эта связь каждому экземпляру ставит в соответствие одну книгу, поэтому мы в любой момент можем однозначно определить, какие книги находятся на руках у читателя, хотя связываем с читателем только инвентарные номера взятых книг. Между сущностями «Читатели» и «Экземпляры» установлена связь «один-ко-многим», и при этом она не обязательная с двух сторон. Читатель в данный момент может не держать ни одной книги на руках, а с другой стороны, данный экземпляр книги может не находиться ни у одного читателя, а просто стоять на полке в библиотеке.

Теперь нам надо отразить последнюю сущность, которая связана с системным каталогом. Системный каталог содержит перечень всех областей знаний, сведения по которым содержатся в библиотечных книгах. Мы можем вспомнить системный каталог в библиотеке, с которого мы обычно начинаем поиск нужных нам книг, если мы не знаем их авторов и названий. Название области знаний может быть длинным и состоять из нескольких слов, поэтому для моделирования системного каталога мы введем сущность «Системный каталог» с двумя атрибутами: «Код области знаний» и «Название области знаний». Атрибут «Код области знаний» будет ключевым атрибутом сущности.

Из описания предметной области нам известно, что каждая книга может содержать сведения из нескольких областей знаний, а с другой стороны, из практики известно, что в библиотеке может присутствовать множество книг, относящихся к одной и той же области знаний, поэтому нам необходимо установить между сущностями «Системный каталог» и «Книги» связь «многие-ко-многим», обязательную с двух сторон. Действительно, в системном каталоге не должно присутствовать такой области знаний, сведения по которой не представлены ни в одной книге нашей библиотеки, противное было бы бессмысленно. И обратно, каждая книга должна быть отнесена к одной или нескольким областям знаний для того, чтобы читатель мог ее быстрее найти.

Инфологическая модель предметной области «Библиотека» представлена на рис. 7.6.

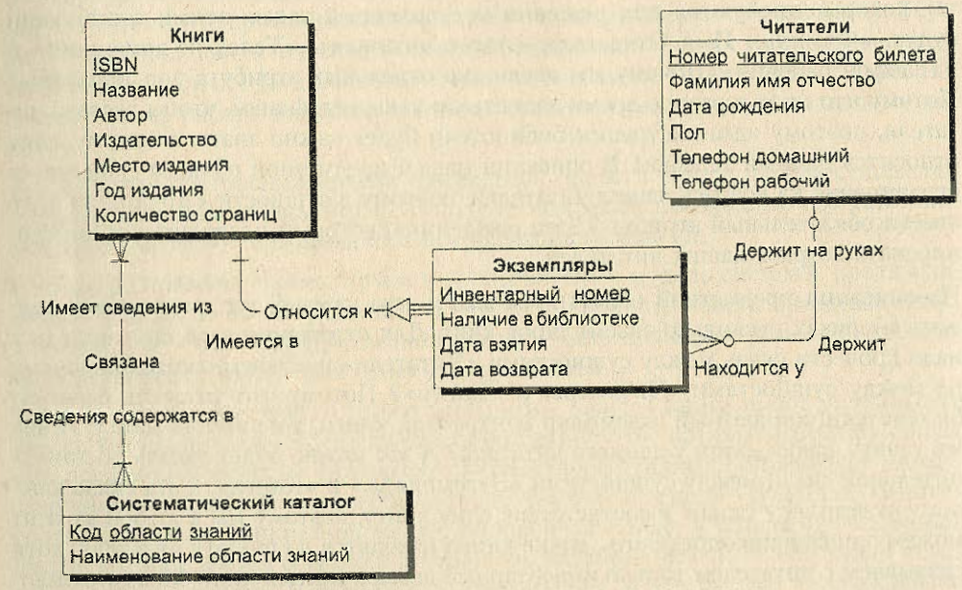


Рис. 7.6. Инфологическая модель «Библиотека»

Инфологическая модель «Библиотека» разработана нами под те задачи, которые были перечислены ранее. В этих задачах мы не ставили условие хранения истории чтения книги, например, с целью поиска того, кто раньше держал книгу и мог нанести ей вред или забыть в ней случайно большую сумму денег. Если бы мы ставили перед собой задачу хранения и этой информации, то наша инфологическая модель была бы другой. Я оставлю эту задачу для вашего самостоятельного творчества.

Переход к реляционной модели данных

Инфологическая модель используется на ранних стадиях разработки проекта. Если понимать язык условных обозначений, которые соответствуют категориям ER-модели, то ее можно легко «читать», следовательно, она доступна для анализа программистам-разработчикам, которые будут разрабатывать отдельные приложения. Она имеет однозначную интерпретацию, в отличие от некоторых предложений естественного языка, и поэтому здесь не может быть никакого недопонимания со стороны разработчиков.

Все специалисты всегда предпочитают выражать свои мысли на некотором формальном языке, который обеспечивает однозначную их трактовку. Таким языком для программистов раньше был язык алгоритмов. Любой алгоритм имел однозначную интерпретацию. Он мог быть реализован на разных языках программирования, но сам алгоритм был и оставался одним и тем же. В первые годы развития вычислительной техники широко издавались сборники алгоритмов для широко распространенных математических задач. Эти сборники программистами прочитывались как увлекательные детективные романы, и они все настоящим программистам были понятны, хотя специалисты других профилей смотрели на эти сборники как на издания на иностранных, неведомых им, языках. Для описания алгоритмов могли использоваться разные формализмы. Одним из таких формализмов был метаязык, в котором использовались слова на естественном языке и каждый мог прочесть эти слова, но смысл самого алгоритма мог понять только тот, кто владел знаниями трактовки алгоритмов.

Вот таким условным общепринятым языком описания базы данных и стал язык ER-модели. Для ER-модели существует алгоритм однозначного преобразования ее в реляционную модель данных, что позволило в дальнейшем разработать множество инструментальных систем, поддерживающих процесс разработки информационных систем, базирующихся на технологии баз данных. И во всех этих системах существуют средства описания инфологической модели разрабатываемой БД с возможностью автоматической генерации той даталогической модели, на которой будет реализовываться проект в дальнейшем.

Рассмотрим правила преобразования ER-модели в реляционную.

1. Каждой сущности ставится в соответствие отношение реляционной модели данных. При этом имена сущности и отношения могут быть различными, потому что на имена сущностей могут не накладываться дополнительные синтаксические ограничения, кроме уникальности имени в рамках модели. Имена отношений могут быть ограничены требованиями конкретной СУБД, чаще всего эти имена являются идентификаторами в некотором базовом языке, они ограничены по длине и не должны содержать пробелов и некоторых специальных символов. Например, сущность может быть названа «Книжный каталог», а соответствующее ей отношение желательно назвать, например, BOOKS (без пробелов и латинскими буквами).
2. Каждый атрибут сущности становится атрибутом соответствующего отношения. Переименование атрибутов должно происходить в соответствии с теми

же правилами, что и переименование отношений в п.1. Для каждого атрибута задается конкретный допустимый в СУБД тип данных и обязательность или необязательность данного атрибута (то есть допустимость или недопустимость NULL значений для него).

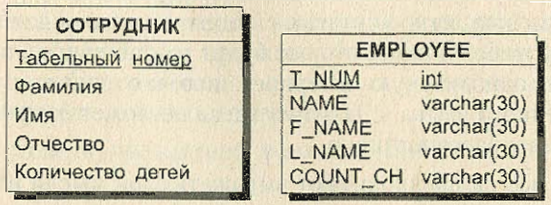


Рис. 7.7. Преобразование сущности СОТРУДНИК к отношению EMPLOYEE

3. Первичный ключ сущности становится PRIMARY KEY соответствующего отношения. Атрибуты, входящие в первичный ключ отношения, автоматически получают свойство обязательности (NOT NULL).

Column Code	Type
COUNT_CH	Varchar(30) Null
F_NAME	Varchar(30) Not Null
L_NAME	Varchar(30) Null
NAME	Varchar(30) Null
T_NUM	Int Not Null

Рис. 7.8. Свойства атрибутов отношения EMPLOYEE

4. В каждое отношение, соответствующее подчиненной сущности, добавляется набор атрибутов основной сущности, являющейся первичным ключом основной сущности. В отношении, соответствующем подчиненной сущности, этот набор атрибутов становится внешним ключом (FOREING KEY).
5. Для моделирования необязательного типа связи на физическом уровне у атрибутов, соответствующих внешнему ключу, устанавливается свойство допустимости неопределенных значений (признак NULL). При обязательном типе связи атрибуты получают свойство отсутствия неопределенных значений (признак NOT NULL).
6. Для отражения категоризации сущностей при переходе к реляционной модели возможны несколько вариантов представления. Возможно создать только одно отношение для всех подтипов одного супертипа. В него включают все атрибуты всех подтипов. Однако тогда для ряда экземпляров ряд атрибутов не будет иметь смысла. И даже если они будут иметь неопределенные значения, то потребуются дополнительные правила различения одних подтипов от других. Достоинством такого представления является то, что создается всего одно отношение.

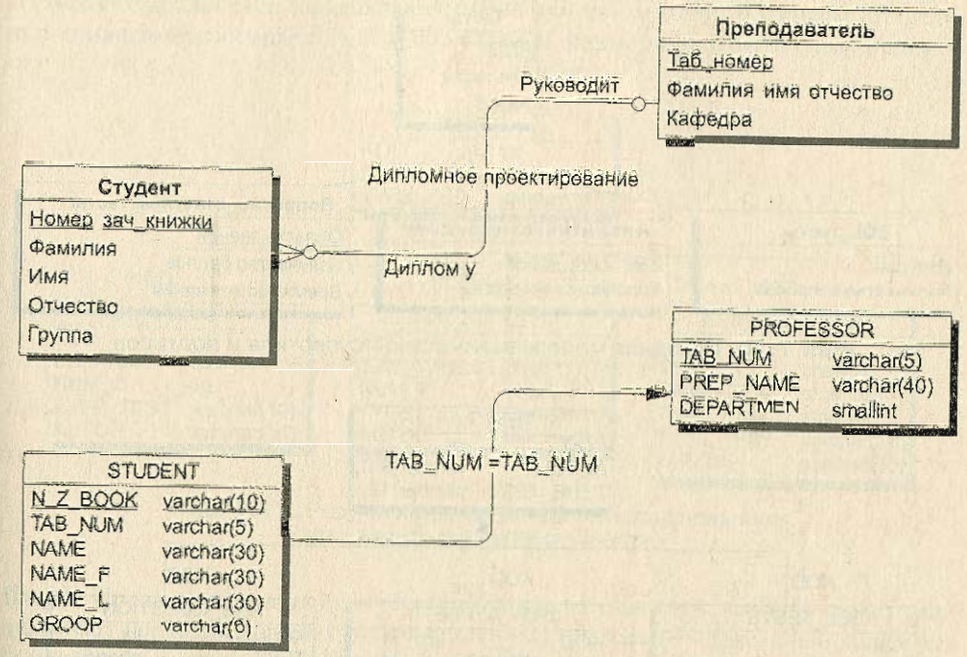


Рис. 7.9. Преобразование взаимосвязанных сущностей СТУДЕНТ и ПРЕПОДАВАТЕЛЬ к взаимосвязанным отношениям STUDENT и PROFESSOR

7. При втором способе для каждого подтипа и для супертипа создаются свои отдельные отношения. Недостатком такого способа представления является то, что создается много отношений, однако достоинств у такого способа больше, так как вы работаете только со значимыми атрибутами подтипа. Кроме того, для возможности переходов к подтипам от супертипа необходимо в супертип включить идентификатор связи.
8. Дополнительно при описании отношения между типом и подтипами необходимо указать тип дискриминатора. Дискриминатор может быть взаимоисключающим (M/E, mutually exclusive) или нет. Если установлен данный тип дискриминатора, то это значит, что один экземпляр сущности супертипа связан только с одним экземпляром сущности подтипа и для каждого экземпляра сущности супертипа существует потомок. Кроме того, необходимо указать для второго способа, наследуется ли только идентификатор супертипа в подтипы или наследуются все атрибуты супертипа.
9. Если мы зададим наследование только идентификатора, то мы получим следующее преобразование (см. рис. 7.10 и 7.11).

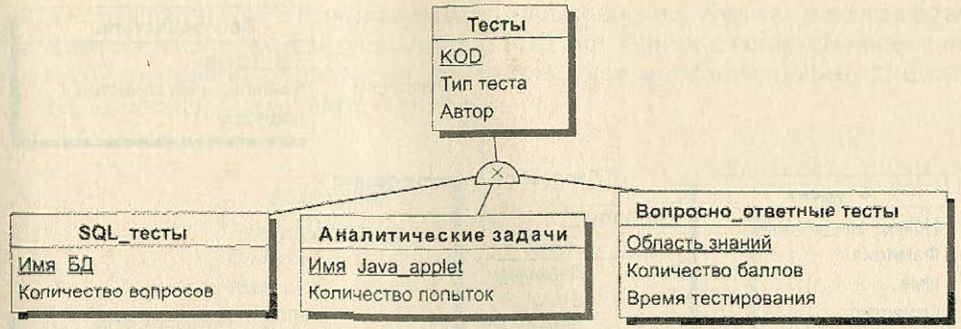


Рис. 7.10. Исходная модель взаимосвязи супертипа и подтипов

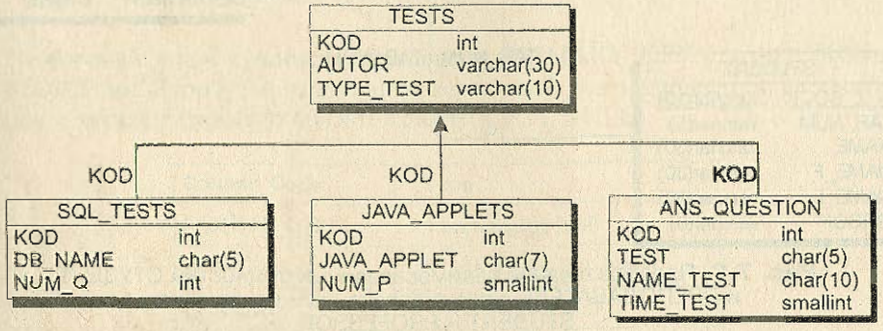


Рис.7.11. Результирующая модель с наследованием только идентификатора суперсущности

Разрешение связей типа «многие-ко-многим». Так как в реляционной модели данных поддерживаются между отношениями только связи типа «один-ко-многим», а в ER-модели допустимы связи «многие-ко-многим», то необходим специальный механизм преобразования, который позволит отразить множественные связи, неспецифические для реляционной модели, с помощью допустимых для нее категорий. Это делается введением специального дополнительного связующего отношения, которое связано с каждым исходным связью «один-ко-многим», атрибутами этого отношения являются первичные ключи связываемых отношений. Так, например, в схеме «Библиотека» присутствует связь такого типа между сущностью «Книги» и «Системный каталог». Для разрешения этой неспецифической связи при переходе к реляционной модели должно быть введено специальное дополнительное отношение, которое имеет всего два атрибута: ISBN (шифр книги) и KOD (код области знаний). При этом каждый из атрибутов нового отношения является внешним ключом (FOREING KEY), а вместе они образуют первичный ключ (PRIMARY KEY) новой связующей сущности. На рис. 7.12 представлена реляционная модель, соответствующая представленной ранее на рис. 7.6 инфологической модели «Библиотека».

Теория нормализации, которую мы рассматривали ранее применительно к реляционной модели, применима и к модели «сущность—связь». Поэтому нормализацию можно проводить и на уровне инфологической (семантической) модели

и смысл ее аналогичен нормализации реляционной модели. Алгоритм приведения семантической модели к 5-й нормальной форме может быть следующим:

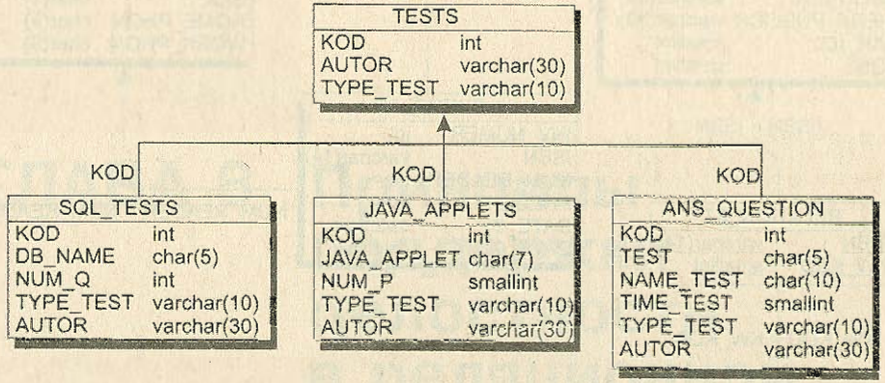


Рис. 7.12. Результирующая модель с наследованием всех атрибутов суперсущности

Шаг 1. Проанализировать схему на присутствие сущностей, которые скрыто моделируют несколько разных взаимосвязанных классов объектов реального мира (именно это соответствует ненормализованным отношениям). Если такое выявлено, то разделить каждую из этих сущностей на несколько новых сущностей и установить между ними соответствующие связи, полученная схема будет находиться в первой нормальной форме. Перейти к шагу 2.

Шаг 2. Проанализировать все сущности, имеющие составные первичные ключи, на наличие неполных функциональных зависимостей непервичных атрибутов от атрибутов возможного ключа. Если такие зависимости обнаружены, то разделить данные сущности на 2, определить для каждой сущности первичные ключи и установить между ними соответствующие связи. Полученная схема будет находиться во второй нормальной форме. Перейти к шагу 3.

Шаг 3. Проанализировать неключевые атрибуты всех сущностей на наличие транзитивных функциональных зависимостей. При обнаружении таковых расщепить каждую сущность на несколько таким образом, чтобы ликвидировать транзитивные зависимости. Схема находится в третьей нормальной форме. Перейти к шагу 4.

Шаг 4. Проанализировать все сущности на наличие детерминантов, которые не являются возможными ключами. При обнаружении подобных расщепить сущность на две, установив между ними соответствующие связи. Полученная схема соответствует нормальной форме Бойса—Кодда. Перейти к шагу 5.

Шаг 5. Проанализировать все сущности на наличие многозначных зависимостей. Если обнаружатся сущности, у которых имеется более одной многозначной зависимости, то расщепить такие сущности на две, установив между ними соответствующие связи. Полученная схема будет находиться в четвертой нормальной форме. Перейти к шагу 6.

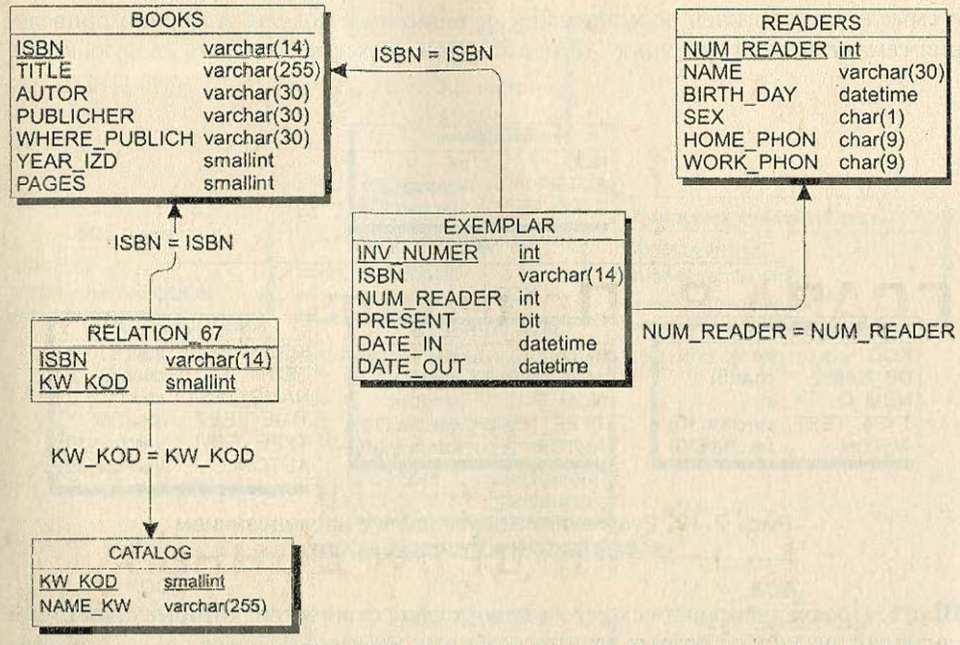


Рис. 7.13. Реляционная схема «Библиотека»

Шаг 6. Проанализировать сущности на наличие в них зависимостей проекции-соединения. При обнаружении таковых расщепить сущность на требуемое число взаимосвязанных сущностей и установить между ними требуемые связи. Полученная таким образом схема будет находиться в пятой нормальной форме и, будучи формально преобразованной к реляционной схеме по указанным выше принципам, даст реляционную схему также в пятой нормальной форме.

ГЛАВА 8 Принципы поддержки целостности в реляционной модели данных

Одним из основополагающих понятий в технологии баз данных является понятие *целостности*. В общем случае это понятие прежде всего связано с тем, что база данных отражает в информационном виде некоторый объект реального мира или совокупность взаимосвязанных объектов реального мира. В реляционной модели объекты реального мира представлены в виде совокупности взаимосвязанных отношений. Под целостностью будем понимать соответствие информационной модели предметной области, хранимой в базе данных, объектам реального мира и их взаимосвязям в каждый момент времени. Любое изменение в предметной области, значимое для построенной модели, должно отражаться в базе данных, и при этом должна сохраняться однозначная интерпретация информационной модели в терминах предметной области.

Мы отметили, что только существенные или значимые изменения предметной области должны отслеживаться в информационной модели. Действительно, модель всегда представляет собой некоторое упрощение реального объекта, в модели мы отражаем только то, что нам важно для решения конкретного набора задач. Именно поэтому в информационной системе «Библиотека» мы, например, не отразили место хранения конкретных экземпляров книг, потому что мы не ставили задачу автоматической адресации библиотечных стеллажей. И в этом случае любое перемещение книг с одного места на другое не будет отражено в модели, это перемещение несущественно для наших задач. С другой стороны, процесс взятия книги читателем или возврат любой книги в библиотеку для нас важен, и мы должны его отслеживать в соответствии с изменениями в реальной предметной области. И с этой точки зрения наличие у экземпляра книги указателя на его отсутствие в библиотеке и одновременное отсутствие записи о конкретном номере читательского билета, за которым числится этот экземпляр книги, является противоречием, такого быть не должно. И в модели данных должны

быть предусмотрены средства и методы, которые позволят нам обеспечивать динамическое отслеживание в базе данных согласованных действий, связанных с согласованным изменением информации. Именно этим вопросам и посвящена данная глава.

Общие понятия и определения целостности

Поддержка целостности в реляционной модели данных в ее классическом понимании включает в себя 3 аспекта.

Во-первых, это поддержка *структурной целостности*, которая трактуется как то, что реляционная СУБД должна допускать работу только с однородными структурами данных типа «реляционное отношение». При этом понятие «реляционного отношения» должно удовлетворять всем ограничениям, накладываемым на него в классической теории реляционной БД (отсутствие дубликатов кортежей, соответственно обязательное наличие первичного ключа, отсутствие понятия упорядоченности кортежей).

В дополнение к структурной целостности необходимо рассмотреть проблему неопределенных Null значений. Как уже указывалось раньше, неопределенное значение интерпретируется в реляционной модели как значение, неизвестное на данный момент времени. Это значение при появлении дополнительной информации в любой момент времени может быть заменено на некоторое конкретное значение. При сравнении неопределенных значений не действуют стандартные правила сравнения: одно неопределенное значение никогда не считается равным другому неопределенному значению. Для выявления равенства значения некоторого атрибута неопределенному применяют специальные стандартные предикаты:

```
<имя атрибута>IS NULL и <имя атрибута> IS NOT NULL.
```

Если в данном кортеже (в данной строке) указанный атрибут имеет неопределенное значение, то предикат IS NULL принимает значение TRUE (Истина), а предикат IS NOT NULL — FALSE (Ложь), в противном случае предикат IS NULL принимает значение FALSE, а предикат IS NOT NULL принимает значение TRUE.

Ведение Null значений вызвало необходимость модификации классической двузначной логики и превращения ее в трехзначную. Все логические операции, производимые с неопределенными значениями, подчиняются этой логике в соответствии с заданной таблицей истинности.

Таблица 8.1. Таблица истинности для логических операций с неопределенными значениями

A	B	Not A	A&B	A V B
TRUE	TRUE	FA	TRUE	TRUE
TRUE	FALSE	FALSE	FALSE	TRUE

A	B	Not A	A&B	A V B
TRUE	Null	FALSE	Null	TRUE
FALSE	TRUE	TRUE	FALSE	TRUE
FALSE	FALSE	TRUE	FALSE	FALSE
FALSE	Null	TRUE	FALSE	Null
Null	TRUE	Null	Null	TRUE
Null	FALSE	Null	FALSE	Null
Null	Null	Null	Null	Null

В стандарте SQL2 появилась возможность сравнивать не только конкретные значения атрибутов с неопределенным значением, но и результаты логических выражений сравнивать с неопределенным значением, для этого введена специальная логическая константа UNKNOWN. В этом случае операция сравнения выглядит как:

```
<Логическое выражение> IS {TRUE | FALSE | UNKNOWN}
```

Во-вторых, это поддержка *языковой целостности*, которая состоит в том, что реляционная СУБД должна обеспечивать языки описания и манипулирования данными не ниже стандарта SQL. Не должны быть доступны иные низкоуровневые средства манипулирования данными, не соответствующие стандарту.

Именно поэтому доступ к информации, хранимой в базе данных, и любые изменения этой информации могут быть выполнены только с использованием операторов языка SQL.

В-третьих, это поддержка *ссылочной целостности* (Declarative Referential Integrity, DRI), означает обеспечение одного из заданных принципов взаимосвязи между экземплярами кортежей взаимосвязанных отношений:

- кортежи подчиненного отношения уничтожаются при удалении кортежа основного отношения, связанного с ними.
- кортежи основного отношения модифицируются при удалении кортежа основного отношения, связанного с ними, при этом на месте ключа родительского отношения ставится неопределенное Null значение.

Ссылочная целостность обеспечивает поддержку непротиворечивого состояния БД в процессе модификации данных при выполнении операций добавления или удаления.

Кроме указанных ограничений целостности, которые в общем виде не определяют семантику БД, вводится понятие *семантической поддержки целостности*.

Структурная, языковая и ссылочная целостность определяют правила работы СУБД с реляционными структурами данных. Требования поддержки этих трех видов целостности говорят о том, что каждая СУБД должна уметь это делать, а разработчики должны это учитывать при построении баз данных с использованием реляционной модели. И эти требования поддержки целостности достаточно абстрактны, они определяют допустимую форму представления и обработки информации в реляционных базах данных. Но с другой стороны, эти аспекты

никак не касаются содержания базы данных. Для определения некоторых ограничений, которые связаны с содержанием базы данных, требуются другие методы. Именно эти методы и сведены в поддержку семантической целостности.

Давайте рассмотрим конкретный пример. То, что мы можем построить схему базы данных или ее концептуальную модель только из совокупности нормализованных таблиц, определяет структурную целостность. И мы построили нашу схему библиотеки из пяти взаимосвязанных отношений. Но мы не можем с помощью перечисленных трех методов поддержки целостности обеспечить ряд правил, которые определены в нашей предметной области и должны в ней соблюдаться. К таким правилам могут быть отнесены следующие:

1. В библиотеке должны быть записаны читатели не моложе 17 лет.
2. В библиотеке присутствуют книги, изданные начиная с 1960 по текущий год.
3. Каждый читатель может держать на руках не более 5 книг.
4. Каждый читатель при регистрации в библиотеке должен дать телефон для связи: он может быть рабочим или домашним.

Принципы семантической поддержки целостности как раз и позволяют обеспечить автоматическое выполнение тех условий, которые перечислены ранее.

Семантическая поддержка может быть обеспечена двумя путями: декларативным и процедурным путем. Декларативный путь связан с наличием механизмов в рамках СУБД, обеспечивающих проверку и выполнение ряда декларативно заданных правил-ограничений, называемых чаще всего «бизнес-правилами» (Business Rules) или декларативными ограничениями целостности.

Выделяются следующие виды декларативных ограничений целостности:

- Ограничения целостности атрибута: значение по умолчанию, задание обязательности или необязательности значений (Null), задание условий на значения атрибутов.

  Задание значения по умолчанию означает, что каждый раз при вводе новой строки в отношение, при отсутствии данных в указанном столбце этому атрибуту присваивается именно значение по умолчанию. Например, при вводе новых книг разумно в качестве значения по умолчанию для года издания задать значение текущего года. Например, для MS Access 97 это выражение будет иметь вид:

  ```
  YEAR(NOW())
  ```

  Здесь `NOW()` — функция, возвращающая значение текущей даты, `YEAR(data)` — функция, возвращающая значение года указанной в качестве параметра даты.

  В качестве условия на значение для года издания надо задать выражение, которое будет истинным только тогда, когда год издания будет лежать в пределах от 1960 года до текущего года. В конкретных СУБД это значение будет формироваться с использованием специальных встроенных функций СУБД.

  Для MS Access 97 это выражение будет выглядеть следующим образом:

  ```
  Between 1960 AND YEAR(NOW())
  ```


  В СУБД MS SQL Server7.0 значение по умолчанию записывается в качестве «бизнес-правила». В этом случае будет использоваться выражение, в котором явным образом должно быть указано имя соответствующего столбца, например:

  ```
  YEAR_PUBL >= 1960 AND YEAR_PUBL <= YEAR(GETDATE())
  ```

  Здесь `GETDATE()` — функция MS SQL Server7.0, возвращающая значение текущей даты, `YEAR_PUBL` — имя столбца, соответствующего году издания.

- Ограничения целостности, задаваемые на уровне доменов, при поддержке доменной структуры. Эти ограничения удобны, если в базе данных присутствуют несколько столбцов разных отношений, которые принимают значения из одного и того же множества допустимых значений. Некоторые СУБД поддерживают подобную доменную структуру, то есть разрешают определять отдельно домены, задавать тип данных для каждого домена и задавать соответственно ограничения в виде бизнес-правил для доменов. А для атрибутов задается не примитивный первичный тип данных, а их принадлежность тому или другому домену. Иногда доменная структура выражена неявно и в ряде СУБД применяется специальная терминология для этого. Так, например, в MS SQL Server 7.0 вместо понятия домена вводится понятие *типа данных, определенных пользователем*, но смысл этого типа данных фактически эквивалентен смыслу домена. В этом случае действительно удобно задать ограничение на значение прямо на уровне домена, тогда оно автоматически будет выполняться для всех атрибутов, которые принимают значения из этого домена. А почему удобно задать это ограничение на уровне домена? А если мы зададим это ограничение для каждого атрибута, входящего в домен, разве наша система будет работать неправильно? Нет, конечно, она будет работать правильно, но представьте себе, что у вас в организации изменились правила работы, которые выражены в виде декларативных ограничений на значения. В нашем случае, например, мы будем комплектовать библиотеку более новыми книгами и теперь будем принимать в библиотеку книги, изданные не позднее 1980 года. А если это ограничение у нас задано не на один столбец, то нам надо просматривать все отношения и во всех отношениях менять старое правило на новое. Не легче ли заменить его один раз в домене, а все атрибуты, которые принимают значения из этого домена, будут автоматически работать по новому правилу.

  Да, это действительно легче, тем более что в процессе работы схема базы данных разрастается и начинает содержать более сотни отношений, и задача нетривиальная — найти все отношения, в которых ранее установлено это ограничение и исправить его.

  Одним из основных правил при разработке проекта базы данных, как мы уже упоминали раньше, является минимизация избыточности, а это означает, что если возможно информацию о чем-то, в том числе и об ограничениях, хранить в одном месте, то это надо делать обязательно.

- Ограничения целостности, задаваемые на уровне отношения. Некоторые семантические правила невозможно преобразовать в выражения, которые будут применимы только к одному столбцу. В нашем примере с библиотекой

мы не сможем выразить требование наличия по крайней мере одного телефонного номера для быстрой связи с читателем. У нас под телефоны отведены два столбца, это в некотором роде искусственно, но специально так сделано, чтобы показать вам другой тип ограничений. Каждый из атрибутов является в общем случае необязательным и может принимать неопределенные значения: не обязательно должен быть задан как рабочий, так и домашний телефон. Мы хотим потребовать, чтобы из двух по крайней мере один телефон был бы задан обязательно. Попробуем сформулировать это в терминологии неопределенных значений баз данных. Домашний телефон должен быть задан (NOT NULL) или рабочий телефон должен быть задан (NOT NULL). Для MS Access97 или для MS SQL Server97 соответствующее выражение будет выглядеть следующим образом:

```
HOME_PHON IS NOT NULL OR WORK_PHON IS NOT NULL
```

- Ограничения целостности, задаваемые на уровне связи между отношениями: задание обязательности связи, принципов каскадного удаления и каскадного изменения данных, задание поддержки ограничений по мощности связи. Эти виды ограничений могут быть выражены заданием обязательности или необязательности значений внешних ключей во взаимосвязанных отношениях.

Декларативные ограничения целостности относятся к ограничениям, которые являются немедленно проверяемыми. Есть ограничения целостности, которые являются откладываемыми. Эти ограничения целостности поддерживаются механизмом транзакций и триггеров. Мы их рассмотрим в следующих главах.

Операторы DDL в языке SQL с заданием ограничений целостности

Декларативные ограничения целостности задаются на уровне операторов создания таблиц. В стандарте SQL оператор создания таблиц имеет следующий синтаксис:

```
<определение таблицы>::=CREATE TABLE <имя таблицы>
(<описание элемента таблицы> [{,<описание элемента таблицы>}...])
<описание элемента таблицы>::=<определение столбца>|
<определение ограничений таблицы>
<определение столбца>::=<имя столбца> <тип данных>
   [<значение по умолчанию>][<дополнительные ограничения столбца>...]
   <значение по умолчанию>::=DEFAULT { <literal> | USER | NULL }
<дополнительные ограничения столбца>::=NOT NULL
   [<ограничение уникальности столбца>]|
   <ограничение по ссылкам столбца>|
   CHECK (<условия проверки на допустимость>)
<ограничение уникальности столбца>::= UNIQUE
```



```
<ограничение по ссылкам столбца>::=FOREIGN KEY <спецификация ссылки>
<спецификация ссылки>::= REFERENCES <имя основной таблицы>
   (<имя первичного ключа основной таблицы>)
```

Давайте кратко прокомментируем оператор определения таблицы, синтаксис которого мы задали с помощью традиционной формы Бэкуса—Наура.

При описании таблицы задается имя таблицы, которое является идентификатором в базовом языке СУБД и должно соответствовать требованиям именования объектов в данном языке.

Кроме имени таблицы в операторе указывается список элементов таблицы, каждый из которых служит либо для определения столбца, либо для определения ограничения целостности определяемой таблицы. Требуется наличие хотя бы одного определения столбца. То есть таблицу, которая не имеет ни одного столбца, определить нельзя. Количество столбцов в одной таблице не ограничено, но в конкретных СУБД обычно бывают ограничения на количество атрибутов. Так, например, в MS SQL Server 6.5 максимальное количество столбцов в таблице было 250, но уже в MS SQL Server 7.0 оно увеличено до 1024.

Оператор CREATE TABLE определяет так называемую базовую таблицу, то есть реальное хранилище данных.

Как видно, кроме обязательной части, в которой задается имя столбца и его тип данных, определение столбца может содержать два необязательных раздела: значение столбца по умолчанию и раздел дополнительных ограничений целостности столбца.

В разделе значения по умолчанию указывается значение, которое должно быть помещено в строку, заносимую в данную таблицу, если значение данного столбца явно не указано. В соответствии со стандартом языка SQL значение по умолчанию может быть указано в виде литеральной константы с типом, соответствующим типу столбца; путем задания ключевого слова USER, которому при выполнении оператора занесения строки соответствует символьная строка, содержащая имя текущего пользователя (в этом случае столбец должен иметь тип символьных строк); или путем задания ключевого слова NULL, означающего, что значением по умолчанию является неопределенное значение. Если значение столбца по умолчанию не специфицировано и в разделе ограничений целостности столбца указано NOT NULL (то есть наличие неопределенных значений запрещено), то попытка занести в таблицу строку с незаданным значением данного столбца приведет к ошибке.

Задание в разделе ограничений целостности столбца выражения NOT NULL приводит к неявному порождению проверочного ограничения целостности для всей таблицы "CHECK (C IS NOT NULL)" (где C — имя данного столбца). Если ограничение NOT NULL не указано и раздел умолчаний отсутствует, то неявно порождается раздел умолчаний DEFAULT NULL. Если указана спецификация уникальности, то порождается соответствующая спецификация уникальности для таблицы.

При задании ограничений уникальности данный столбец определяется как возможный ключ, что предполагает уникальность каждого вводимого значения в данный столбец. И если это ограничение задано, то СУБД будет автоматиче-

ски осуществлять проверку на отсутствие дубликатов значений данного столбца во всей таблице.

Если в разделе ограничений целостности указано ограничение по ссылкам данного столбца, то порождается соответствующее определение ограничения по ссылкам для таблицы: FOREIGN KEY(<имя столбца>) <спецификация ссылки>, что означает, что значения данного столбца должны быть взяты из соответствующего столбца родительской таблицы. Родительской таблицей в данном случае называется таблица, которая связана с данной таблицей связью «один-ко-многим» (1:M). При этом каждая строка родительской таблицы может быть связана с несколькими строками определяемой таблицы. Трансляция операторов SQL проводится в режиме интерпретации, поэтому важно, чтобы сначала была бы описана родительская таблица, а потом уже все подчиненные (дочерние) таблицы, связанные с ней. Иначе транслятор определит ссылку на неопределенный объект.

Наконец, если указано проверочное ограничение столбца, то условие поиска этого ограничения должно ссылаться только на данный столбец, и неявно порождается соответствующее проверочное ограничение для всей таблицы. В проверочных ограничениях, накладываемых на столбец, нельзя задавать сравнение со значениями других столбцов данной таблицы.

В главе 5 определены типы данных, которые допустимы по стандартам SQL. Попробуем написать простейший оператор создания таблицы BOOKS из базы данных «Библиотека».

При этом будем предполагать наличие следующих ограничений целостности:

- Шифр книги — последовательность символов длиной не более 14, однозначно определяющая книгу, значит, это — фактически первичный ключ таблицы BOOKS.
- Название книги — последовательность символов, не более 120. Обязательно должно быть задано.
- Автор — последовательность символов, не более 30, может быть не задан.
- Соавтор — последовательность символов, не более 30, может быть не задан.
- Год издания — целое число, не менее 1960 и не более текущего года. По умолчанию ставится текущий год.
- Издательство — последовательность символов, не более 20, может отсутствовать.
- Количество страниц — целое число не менее 5 и не более 1000.

```
CREATE TABLE BOOKS
(
   ISBN      varchar(14)  NOT NULL PRIMARY KEY,
   TITLE     varchar(120) NOT NULL,
   AUTOR     varchar (30) NULL,
   COAUTOR   varchar(30)  NULL,
   YEAR_PUBL smallint DEFAULT Year(GetDate()) CHECK(YEAR_PUBL >= 1960 AND
      YEAR_PUBL <= YEAR(GetDate())),
```



```
   PUBLICH   varchar(20)  NULL,
   PAGES     smallint CHECK(PAGES > = 5 AND PAGES <= 1000)
);
```

Почему мы не задали обязательность значения для количества страниц в книге? Потому что это является следствием проверочного ограничения, заданного на количество страниц, количество страниц всегда должно лежать в пределах от 5 до 1000, значит, оно не может быть незаданным и система это контролирует автоматически.

Теперь зададим описание таблицы «Читатели», которой соответствует отношение READERS:

- Номер читательского билета — это целое число в пределах 32 000 и он уникально определяет читателя.
- Имя, фамилия читателя — это последовательность символов, не более 30.
- Адрес — это последовательность символов, не более 50.
- Номера телефонов рабочего и домашнего — последовательность символов, не более 12.
- Дата рождения — календарная дата. В библиотеку принимаются читатели не младше 17 лет.

```
CREATE TABLE READERS
(
READER_ID      Smallint(4)  PRIMARY KEY,
   FIRST_NAME  char(30)     NOT NULL,
   LAST_NAME   char(30)     NOT NULL,
   ADRES       char(50),
   HOME_PHON   char(12),
   WORK_PHON   char(12),
   BIRTH_DAY date CHECK(DateDiff(year, GetDate(),BIRTH_DAY) >=17)
);
```

Здесь DateDiff (часть даты, начальная дата, конечная дата) — функция MS SQL Server 7.0, которая определяет разность между начальной и конечной датами, заданную в единицах, определенных первым параметром — часть даты. Мы задали в качестве параметра Year, что значит, что мы разность определяем в годах.

Теперь зададим операцию создания таблицы EXEMPLAR (экземпляры книги). В этой таблице первичным ключом является атрибут, задающий инвентарный номер экземпляра книги. В такой постановке мы полагаем, что при поступлении книг в библиотеку им просто присваиваются соответствующие порядковые номера. Для того чтобы не утруждать библиотекаря все время помнить, какой номер был последним, мы можем воспользоваться тем, что некоторые СУБД допускают специальный инкрементный тип данных, то есть такой, значения которого автоматически увеличиваются или уменьшаются на заданную величину при каждом новом вводе данных. В СУБД MS Access такой тип данных называется

«счетчик» (counter) и он всегда имеет начальное значение 1 и шаг, равный тоже 1, то есть при вводе каждого нового значения счетчик увеличивается на 1, значит, практически считает вновь введенные значения. В СУБД MS SQL Server 7.0 это свойство IDENTITY, которое может быть присвоено ряду целочисленных типов данных. В отличие от «счетчика» свойство IDENTITY позволяет считать с любым шагом, положительным или отрицательным, но обязательно целым. Если мы не задаем дополнительных параметров этому свойству, то оно начинает работать как счетчик в MS Access, начиная с единицы и добавляя при каждом вводе тоже единицу.

Кроме того, таблица EXEMPLAR является подчиненной двум другим ранее определенным таблицам: BOOKS и READERS. При этом с таблицей BOOKS таблица EXEMPLAR связана обязательной связью, потому что не может быть ни одного экземпляра книги, который бы не был приписан конкретной книге. С таблицей READERS таблица EXEMPLAR связана необязательной связью, потому что не каждый экземпляр в данный момент находится на руках у читателя. Для моделирования этих связей при создании таблицы EXEMPLAR должны быть определены два внешних ключа (FOREIGN KEY). При этом атрибут, соответствующий шифру книги (мы его назовем так же, как и в родительской таблице — ISBN), является обязательным, то есть не может принимать неопределенных значений, а атрибут, который является внешним ключом для связи с таблице READERS, является необязательным и может принимать неопределенные значения.

Необязательными там являются два остальных атрибута: дата взятия и дата возврата книги, оба они имеют тип данных, соответствующей календарной дате. Атрибут, который содержит информацию о присутствии или отсутствии книги, имеет логический тип. Напишем оператор создания таблицы EXEMPLAR в синтаксисе MS SQL Server 7.0:

```
CREATE TABLE EXEMPLAR
(
   EXEMPLAR_ID  INT          IDENTITY PRIMARY KEY,
   ISBN         varchar(14)  NOT NULL FOREIGN KEY references BOOKS(ISBN),
   READER_ID    Smallint(4)  NULL FOREIGN KEY references READERS (READER_ID),
   DATA_IN      date,
   DATA_OUT     date,
   EXIST        Logical,
);
```

Как мы уже знаем, не все декларативные ограничения могут быть заданы на уровне столбцов таблицы, часть ограничений может быть задана только на уровне всей таблицы. Например, если мы имеем в качестве первичного ключа не один атрибут, а последовательность атрибутов, то мы не можем определить ограничение типа PRIMARY KEY (первичный ключ) только на уровне всей таблицы.

Допустим, что мы считаем экземпляры книги не подряд, а отдельно для каждого издания, тогда таблица EXEMPLAR в качестве первичного ключа будет иметь набор из двух атрибутов: это шифр книги (ISBN) и порядковый номер экземпляра

данной книги (ID_EXEMPL), в этом случае оператор создания таблицы EXEMPLAR будет выглядеть следующим образом:

```
CREATE TABLE EXEMPLAR
(
   ID_EXEMPLAR  int          NOT NULL,
   ISBN         varchar(14)  NOT NULL FOREIGN KEY references BOOKS(ISBN),
   READER_ID    Smallint(4)  NULL FOREIGN KEY references READERS (READER_ID),
   DATA_IN      date,
   DATA_OUT     date,
   EXIST        Logical,
   PRIMARY KEY (ID_EXEMPLAR, ISBN)
);
```

Мы видим, что один и тот же атрибут ISBN, с одной стороны, является внешним ключом (FORIGN KEY), а с другой стороны, является частью первичного ключа (PRIMARY KEY). И ограничение типа первичный ключ (PRIMARY KEY) задается не на уровне одного атрибута, а на уровне всей таблицы, потому что оно содержит набор атрибутов.

То же самое можно сказать и о проверочных (CHECK) ограничениях, если условия проверки предполагают сравнения значений нескольких столбцов таблицы. Введем дополнительное ограничение для таблицы BOOKS, которое может быть сформулировано следующим образом: соавтор не может быть задан, если не задан автор. При описании книги допустимо не задавать ни автора, ни соавтора, или задать и автора и соавтора, или задать только автора. Однако задание соавтора в отсутствие задания автора считается ошибочным. В этом случае оператор создания таблицы BOOKS будет выглядеть следующим образом:

```
CREATE TABLE BOOKS
(
   ISBN      varchar(14)     NOT NULL PRIMARY KEY,
   TITLE     varchar(120)    NOT NULL,
   AUTOR     varchar (30)    NULL,
   COAUTOR   varchar(30)     NULL,
   YEAR_PUBL smallint DEFAULT Year(GetDate()) CHECK(YEAR_PUBL >= 1960 AND
      YEAR_PUBL <= YEAR(GetDate())),
   PUBLICH   varchar(20)     NULL,
   PAGES     smallint        CHECK(PAGES > = 5 AND PAGES <= 1000),
   CHECK (NOT (AUTOR IS NULL AND COAUTOR IS NOT NULL))
);
```

Для анализа ошибок целесообразно именовать все ограничения, особенно если таблица содержит несколько ограничений одного типа. Для именования ограничений используется ключевое слово CONSTRAINT, после которого следует уникаль-

ное имя ограничения, затем тип ограничения и его выражения. Для идентификации ограничений рекомендуют использовать систему именования, которая легко позволит определить при получении сообщения об ошибке, которое вырабатывает СУБД, какое ограничение нарушено. Обычно имя ограничения состоит из краткого названия типа ограничения, далее через символ подчеркивания идет имя атрибута или таблицы, в зависимости от того, к какому уровню относится ограничение, и, наконец, порядковый номер ограничения данного типа, если к одному объекту задается несколько ограничений одного типа.

Сокращенные обозначения ограничений состоят из одной или двух букв и могут быть следующими:

- PK — для первичного ключа;
- FK — для внешнего ключа;
- CK — для проверочного ограничения;
- U — для ограничения уникальности;
- DF — для ограничения типа значение по умолчанию.

Приведем пример оператора создания таблицы BOOKS с именованными ограничениями:

```
CREATE TABLE BOOKS
(
ISBN       varchar(14)     NOT NULL,
TITLE      varchar(120)    NOT NULL,
AUTOR      varchar (30)    NULL,
COAUTOR    varchar(30)     NULL,
YEAR_PUBL smallint         NOT NULL,
PUBLICH    varchar(20)     NULL,
PAGES      smallint        NOT NULL,
CONSTRAINT PK_BOOKS PRIMARY KEY (ISBN),
CONSTRAINT DF_ YEAR_PUBL DEFAULT (Year(GetDate()),
CONSTRAINT CK_ YEAR_PUBL CHECK (YEAR_PUBL >= 1960 AND
   YEAR_PUBL <= YEAR(GetDate())),
CONSTRANT CK_PAGES CHECK (PAGES > = 5 AND PAGES <= 1000),
CONSTRAINT CK_BOOKS CHECK (NOT (AUTOR IS NULL AND COAUTOR IS NOT NULL))
);
CREATE TABLE READERS
(
   READER_ID    Smallint     PRIMARY KEY,
   FIRST_NAME   char(30)     NOT NULL,
   LAST_NAME    char(30)     NOT NULL,
   ADRES        char(50),
```



```
   HOME_PHON    char(12),
   WORK_PHON    char(12),
   BIRTH_DAY    date      CHECK( DateDiff(year, GetDate(),BIRTH_DAY) >=17 ),
CONSTRAINT CK_READERS CHECK (HOME_PHON IS NOT NULL OR WORK_PHON IS NOT NULL)
);
CREATE TABLE CATALOG
(
   ID_CATALOG        Smallint     PRIMARY KEY,
   KNOWELEDGE_AREA   varchar(150)
);
CREATE TABLE EXEMPLAR
(
   ID_EXEMPLAR int          NOT NULL,
   ISBN        varchar(14)  NOT NULL FOREIGN KEY references BOOKS(ISBN),
   READER_ID   Smallint(4)  NULL FOREIGN KEY references READERS (READER_ID),
   DATA_IN     date,
   DATA_OUT    date,
   EXIST       Logical,
   PRIMARY KEY (ID_EXEMPLAR, ISBN)
);
CREATE TABLE RELATION_1
(
ISBN        varchar(14)  NOT NULL FOREIGN KEY references BOOKS(ISBN),
ID_CATALOG  smallint     NOT NULL FOREIGN KEY references CATALOG(ID_CATALOG),
CONSTRAINT PK_RELATION_1 PRIMARY KEY (ISBN,ID_CATALOG)
);
```

Операторы языка SQL, как указывалось ранее, транслируются в режиме интерпретации, в отличие от большинства алгоритмических языков, трансляторы для которых выполнены по принципу компиляции. В режиме интерпретации каждый оператор отдельно транслируется, то есть переводится в машинные коды, и тут же выполняется. В режиме компиляции вся программа, то есть совокупность операторов, сначала переводится в машинные коды, а затем может быть выполнена как единое целое. Такая особенность SQL накладывает ограничение на порядок описания создаваемых таблиц. Действительно, если при трансляции оператора описания подчиненной таблицы с указанным внешним ключом и соответствующей ссылкой на родительскую таблицу эта родительская таблица не будет обнаружена, то мы получим сообщение об ошибке с указанием ссылки на несуществующий объект. Сначала должны быть описаны все основные таблицы, а потом подчиненные таблицы.

В нашем примере с библиотекой порядок описания таблиц следующий:

1. Таблица BOOKS
2. Таблица READERS
3. Таблица CATALOG (системный каталог)
4. Таблица EXEMPLAR
5. Таблица RELATION_1 (дополнительная связующая таблица между книгами и системным каталогом).

Набор операторов языка SQL принято называть не программой, а скриптом. Тогда скрипт, который добавит набор из 5 взаимосвязанных таблиц базы данных «Библиотека» в существующую базу данных, будет выглядеть следующим образом:

```
CREATE TABLE BOOKS
(
   ISBN          varchar(14)     NOT NULL ,
   TITLE         varchar(120)    NOT NULL,
   AUTOR         varchar (30)    NULL,
   COAUTOR       varchar(30)     NULL,
   YEAR_PUBL     smallint        NOT NULL,
   PUBLICH       varchar(20)     NULL,
   PAGES         smallint        NOT NULL,
CONSTRAINT PK_BOOKS PRIMARY KEY (ISBN),
CONSTRAINT DF_ YEAR_PUBL DEFAULT (Year(GetDate()),
CONSTRAINT CK_ YEAR_PUBL CHECK (YEAR_PUBL >= 1960 AND
   YEAR_PUBL <= YEAR(GetDate())),
CONSTRANT CK_PAGES CHECK (PAGES >. = 5 AND PAGES <= 1000),
CONSTRAINT CK_BOOKS CHECK (NOT (AUTOR IS NULL AND COAUTOR IS NOT NULL))
CREATE TABLE READERS
(
   READER_ID       Smallint     PRIMARY KEY,
   FIRST_NAME      char(30)     NOT NULL,
   LAST_NAME       char(30)     NOT NULL,
   ADRES           char(50),
   HOME_PHON       char(12),
   WORK_PHON       char(12),
   BIRTH_DAY date  CHECK( DateDiff(year, GetDate(),BIRTH_DAY) >=17 ),
CONSTRAINT CK_READERS CHECK (HOME_PHON IS NOT NULL OR WORK_PHON IS NOT NULL)
);
CREATE TABLE CATALOG
(
```



```
   ID_CATALOG          Smallint     PRIMARY KEY,
   KNOWELEDGE_AREA     varchar(150)
);
CREATE TABLE EXEMPLAR
(
   ID_EXEMPLAR     int          NOT NULL,
   ISBN            varchar(14)  NOT NULL FOREIGN KEY references BOOKS(ISBN),
   READER_ID       Smallint(4)  NULL FOREIGN KEY references READERS (READER_ID),
   DATA_IN         date,
   DATA_OUT        date,
   EXIST           Logical,
   PRIMARY KEY (ID_EXEMPLAR, ISBN)
);
CREATE TABLE RELATION_1
(
ISBN         varchar(14)  NOT NULL FOREIGN KEY references BOOKS(ISBN),
ID_CATALOG   smallint     NOT NULL FOREIGN KEY references CATALOG(ID_CATALOG),
CONSTRAINT PK_RELATION_1 PRIMARY KEY (ISBN,ID_CATALOG)
).
```

При написании скрипта мы добавили в оператор создания таблицы «Читатели» ограничение на уровне таблицы, которое связано с обязательным наличием хотя бы одного из двух телефонов.

Средства определения схемы базы данных

В стандарте SQL1 задается спецификация оператора описания схемы базы данных, но не указывается способ создания собственно базы данных, поэтому в различных СУБД используются неодинаковые подходы к этому вопросу.

Например, в СУБД ORACLE база данных создается в ходе установки программного обеспечения собственно СУБД. Все таблицы пользователей помещаются в единую базу данных. Однако они могут быть разделены на группы, объединенные в подсхемы. Понятие подсхемы не стандартизировано в SQL и не используется в других СУБД.

В состав СУБД INGRES входит специальная системная утилита, имеющая имя CREATEDB, которая позволяет создавать новые базы данных. Права на использование этой утилиты имеет администратор сервера. Для удаления базы данных существует соответствующая утилита DESTROYDB.

В СУБД MS SQL Server существует специальный оператор CREATE DATABASE, который является частью языка определения данных, для удаления базы данных

в языке определен оператор DROP DATABASE. Правами на создание баз данных наделяются администраторы баз данных, которых в общем случае может быть несколько. Правами более высокого уровня обладает администратор сервера баз данных (SQL Server), который и может предоставить права администратора базы данных другим пользователям сервера. Администраторы баз данных могут удалить только свою базу данных. Приведем пример оператора создания схемы базы данных в MS SQL Server 7.0:

```
CREATE DATABASE database_name
[ON [PRIMARY][<спецификация файла>[....n]][,<группа файлов> [....n]]]
   [ LOG ON { <спецификация файла> [....n]} ][ FOR LOAD | FOR ATTACH ]
<спецификация файла> ::=
( [ NAME = логическое имя файла,]FILENAME = 'физическое имя файла'
   [, SIZE = размер][, MAXSIZE = { максимальный размер | UNLIMITED } ]
   [, FILEGROWTH = инкремент увеличения файла] ) [,...n]
<группа файлов>::= FILEGROUP имя группы файлов <спецификация файла> [....n]
```

Здесь

- database_name — имя базы данных, идентификатор в системе;
- ON — ключевое слово, которое означает, что далее будут заданы спецификации файлов, которые будут использованы для размещения базы данных;
- PRIMARY — ключевое слово, которое определяет первичное файловое пространство, в котором будет размещена собственно база данных;
- LOG ON — ключевое слово, которое задает спецификацию файлов, которые будут использованы для хранения журналов транзакций;
- FOR LOAD — ключевое слово, которое определяет, что после создания базы данных будет произведена загрузка базы данных данными;
- FOR ATTACH — предложение, которое определяет, что база данных для управления будет подсоединена к другому серверу.

Почти все параметры, кроме имени базы данных, являются необязательными, поэтому оператор создания простой базы данных «Библиотека» может выглядеть следующим образом:

```
CREATE DATABASE Library
```

Для изменения схемы базы данных в MS SQL Server 7.0 может быть использована команда:

```
ALTER DATABASE database
{ ADD FILE <спецификация файлов> [....n] [TO FILEGROUP filegroup_name]
| ADD LOG FILE <спецификация файлов> [....n]
| REMOVE FILE имя_файла
| ADD FILEGROUP имя_группы файлов
|REMOVE FILEGROUP имя группы_файлов
```



```
|MODIFY FILE <спецификация файлов>
|MODIFY FILEGROUP имя_группы_файлов имя_свойства_группы файлов}
```

Здесь свойства группы файлов определяет одно из допустимых ключевых слов:

- READONLY — только для чтения;
- READWRITE — для чтения и записи;
- DEFAULT — назначает данную группу файлов в качестве группы по умолчанию, в которой размещаются данные, если не задано дополнительных условий размещения информации.

Как видно, при изменении схемы базы данных в нее могут быть добавлены (ADD) дополнительные файлы и файловые группы или удалены (REMOVE) ранее определенные файлы или файловые группы. Назначение этих файлов нам будет более понятно после того, как мы познакомимся с физическими моделями и файловыми структурами, используемыми для хранения данных в базах данных.

Сейчас мы познакомимся с последней командой, которая предназначена для удаления базы данных. В MS SQL Server 7.0 это команда имеет следующий синтаксис:

```
DROP DATABASE database_name
```

После выполнения этой команды уничтожается вся база данных вместе с содержащимися в ней данными.

Средства изменения описания таблиц и средства удаления таблиц

В язык SQL добавлены средства изменения схемы таблиц. Что можно и что нельзя изменять в описании таблицы? В стандарте SQL2 добавлены достаточно широкие возможности по модификации уже существующих схем таблиц. Для модификации таблиц используется оператор ALTER TABLE, который позволяет выполнить следующие операции изменения для схемы таблицы:

- добавить новый столбец в уже существующую и заполненную таблицу;
- изменить значение по умолчанию для какого-либо столбца;
- удалить столбец из существующей таблицы;
- добавить или удалить первичный ключ таблицы;
- добавить или удалить новый внешний ключ таблицы;
- добавить или удалить условие уникальности;
- добавить или удалить условие проверки для любого столбца или для таблицы в целом.

Синтаксис оператора ALTER TABLE:

```
<Изменить описание таблицы>::= ALTER TABLE <имя таблицы>
   { ADD <определение столбца> |
```

```
ALTER <имя столбца> {SET DEFAULT <значение>
DROP DEFAULT } |
DROP <имя столбца> {CASCADE | RESTRICT} |
ADD { <определение первичного ключа>|
<определение внешнего ключа> |
<условие уникальности данных> |
<условие проверки>   } |
DROP CONSTRAINT имя условия { CASCADE |
RESTRICT}
}
```

Одним оператором ALTER TABLE можно провести только одно из перечисленных изменений, например, за один раз можно добавить один столбец. Если вам требуется добавить два столбца, то необходимо применить два оператора.

Давайте рассмотрим несколько примеров. Чаще всего применяется операция добавления столбца. Предложение определения нового столбца в операторе ALTER TABLE имеет точно такой же синтаксис, как и в операторе создания таблицы. Добавим столбец EDUCATION (образовние), содержащий символьный тип данных, с заданным перечнем значений («начальное», «среднее», «неоконченное высшее», «высшее»).

```
ALTER TABLE READERS
ADD EDUCATION varchar (30) DEFAULT NULL
CHECK (EDUCATION IS NULL OR
    EDUCATION= "начальное" OR
    EDUCATION= "среднее " OR EDUCATION= "неоконченное высшее" OR
    EDUCATION= "высшее" )
```

В таблицу READERS будет добавлен столбец EDUCATION, в который по умолчанию будут добавлены все кортежи неопределенного значения. В дальнейшем эти значения могут быть заменены на одно из допустимых символьных значений.

Добавим ограничение на соответствие между датами взятия и возврата книги в таблице EXEMPLAR. Действительно, если даты введены, то требуется, чтобы дата возврата книги была бы больше на срок выдачи книги. Считаем, что стандартным сроком являются 2 недели. Теперь сформулируем оператор изменения таблицы EXEMPLARE:

```
ALTER TABLE EXEMPLARE
ADD CONSTRAINT CK_ EXEMPLARE CHECK ((DATA_IN IS NULL AND DATA_OUT IS NULL) OR
    (DATA_OUT >= DATA_IN +14) )
```

Здесь мы применили операцию сложения к календарной дате, которая предполагает, что добавляют заданное число дней.

Операция удаления столбца связана с проверкой ссылочной целостности, и поэтому не разрешается удалять столбцы, связанные с поддержкой ссылочной целостности таблицы, то есть нельзя удалить столбцы родительской таблицы, входящие в первичный ключ таблицы, если на них есть ссылки в подчиненных таблицах.

При изменении первичного ключа таблицы следует быть внимательными. Во-первых, у исходной таблицы могут быть подчиненные, при этом первичный ключ исходной таблицы является внешним ключом для подчиненных таблиц, и просто его удалить невозможно, СУБД контролирует ссылочную целостность и не позволит выполнить операцию удаления первичного ключа таблицы, если на него имеются ссылки. Следовательно, в этом случае порядок изменения первичного ключа должен быть таким, как на рис. 8.1:

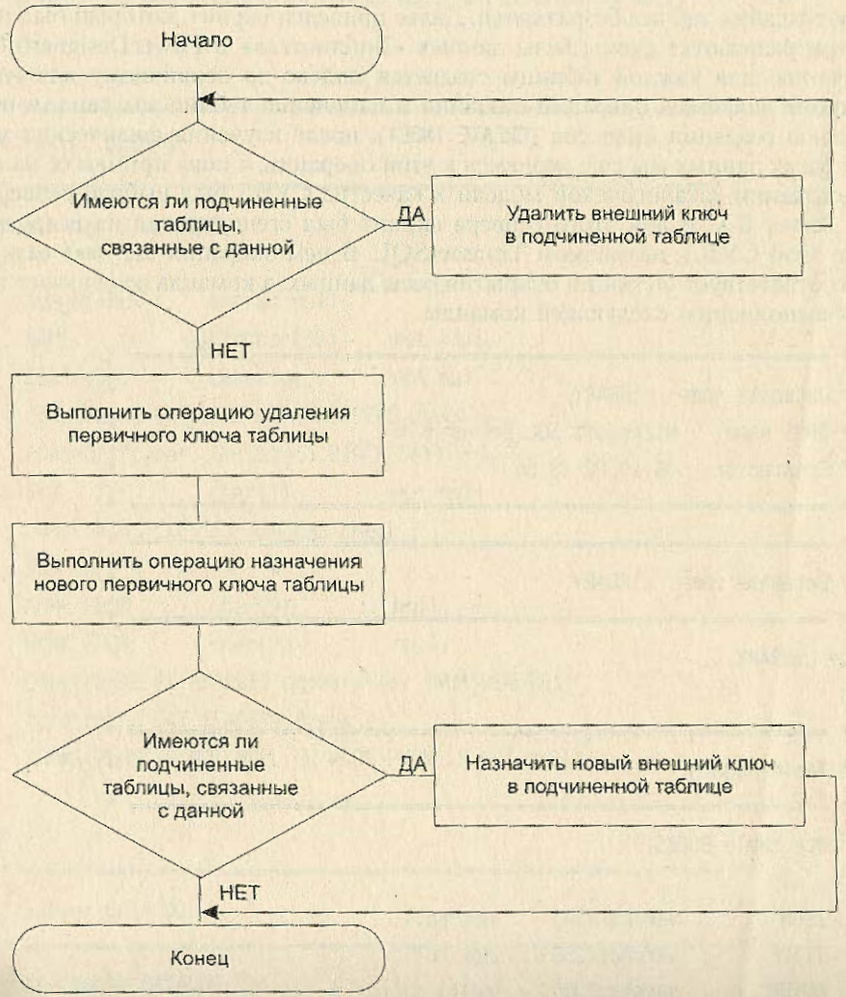


Рис. 8.1. Алгоритм изменения первичного ключа таблицы

Чаще всего операция ALTER TABLE применяется в CASE-системах при автоматической генерации скриптов создания таблиц в базе данных. В этих системах универсальный алгоритм предполагает сначала создание всех таблиц, которые заданы в даталогической модели, и только после этого добавляются соответствующие связи. И это понятно — в отличие от человеческого разума искусственный интеллект CASE-системы будет испытывать затруднения в определении иерархических взаимосвязей таблиц базы данных, поэтому он предпочитает использовать универсальный алгоритм, в котором сначала все объекты определяются, а затем добавляются соответствующие свойства для атрибутов, которые являются внешними ключами с указанием требуемых ссылок. В этом случае все операции назначения внешних ключей будут считаться корректными, потому что все объекты были описаны заранее, и для такого алгоритма порядок создания таблиц безразличен. Далее приведен скрипт, который был получен при разработке схемы базы данных «Библиотека» в PowerDesigner6.1. По умолчанию для каждой таблицы создается индекс по первичному ключу, так что кроме знакомых операций создания и изменения таблиц мы увидим еще и операцию создания индексов (CREATE INDEX), после изучения физических моделей в базах данных мы еще вернемся к этой операции, а пока примем ее на веру. При создании даталогичекой модели в качестве СУБД был выбран сервер MS SQL Server 6.X, и для этого сервера скрипт был сгенерирован на встроенном языке этой СУБД, называмом TransactSQL. В нем операция USE <имя базы данных> соответствует операции открытия базы данных, а команда go означает переход к выполнению следующей команды.

```
/* ============================================================ */
/* Database name: LIBRARY                                       */
/* DBMS name:   Microsoft SQL Server 6.x                        */
/* Created on:  06.10.00 18:56                                  */
/* ============================================================ */
/* ============================================================ */
/* Database name: LIBRARY                                       */
/* ============================================================ */
use LIBRARY
go
/* ============================================================ */
/* Table: BOOKS                                                 */
/* ============================================================ */
create table BOOKS
(
   ISBN          varchar(14)     not null,
   TITLE         varchar(255)    not null,
   AUTOR         varchar(30)     null,
   COAUTOR       varchar(30)     null,
   PUBLICHER     varchar(30)     null,
```



```
   WHERE_PUBLICH  varchar(30)     null,
   YEAR_IZD       smallint        not null
   constraint CKC_YEAR_IZD_BOOKS check (
   YEAR_PUBL >= 1969 AND YEAR_PUBL <= YEAR(GetDate())),
   PAGES smallint not null
   constraint CKC_PAGES_BOOKS check (
   PAGES between 5 and 1000),
   constraint PK_BOOKS primary key (ISBN),
   constraint CKT_BOOKS check (
   (AUTOR IS NOT NULL OR (AUTOR IS NULL AND COAUTOR IS NULL)))
)
go
/* ============================================================ */
/* Table: READERS                                               */
/* ============================================================ */
create table READERS
(
   NUM_READER     intnot null,
   NAME           varchar(30)     not null,
   BIRTH_DAY      datetime        not null
   constraint CKC_BIRTH_DAY_READERS check (
   (DateDiff(year, GetDate(),BIRTH_DAY) >=17 )),
   SEX            char(1)         not null
   constraint CKC_SEX_READERS check (
   SEX in ('M','Ж','м','ж')),
   HOME_PHON      char(9)         null,
   WORK_PHON      char(9)         null,
   constraint PK_READERS primary key (NUM_READER),
   constraint CKT_READERS check (
   (HOME_PHON IS NOT NULL OR WORK_PHON IS NOT NULL))
)
go
/* ============================================================ */
/* Table: CATALOG                                               */
/* ============================================================ */
create table CATALOG
(
   KW_KOD         smallint        not null,
```

```
    NAME_KW      varchar(255)    null,
    constraint PK_CATALOG primary key (KW_KOD)
)
go
/* ============================================================ */
/* Table: EXEMPLAR                                              */
/* ============================================================ */
create table EXEMPLAR
(
    INV_NUMER    intnot          null,
    ISBN         varchar(14)     not null,
    NUM_READER   int             null,
    PRESENT      bitnot          null,
    DATE_IN      datetime        null,
    DATE_OUT     datetime        null,
    constraint PK_EXEMPLAR primary key (INV_NUMER)
)
go
/* ============================================================ */
/* Index: RELATION_43_FK                                        */
/* ============================================================ */
create index RELATION_43_FK on EXEMPLAR (ISBN)
go
/* ============================================================ */
/* Index: RELATION_44_FK                                        */
/* ============================================================ */
create index RELATION_44_FK on EXEMPLAR (NUM_READER)
go
/* ============================================================ */
/* Table: RELATION_67                                           */
/* ============================================================ */
create table RELATION_67
(
    ISBN         varchar(14)     not null,
    KW_KOD       smallint        not null,
    constraint PK_RELATION_67 primary key (ISBN, KW_KOD)
)
go
```



```
/* ============================================================ */
/* Index: IOIINEONY_E_IAEANOE_CIAIEE_FK                         */
/* ============================================================ */
create index IOIINEONY_E_IAEANOE_CIAIEE_FK on RELATION_67 (ISBN)
go
/* ============================================================ */
/* Index: I_AANOAAEAIA_A_EIEAAO_FK                              */
/* ============================================================ */
create index I_AANOAAEAIA_A_EIEAAO_FK on RELATION_67 (KW_KOD)
go
alter table EXEMPLAR
   add constraint FK_EXEMPLAR_RELATION__BOOKS foreign key (ISBN)
   references BOOKS (ISBN)
go
alter table EXEMPLAR
   add constraint FK_EXEMPLAR_RELATION__READERS foreign key (NUM_READER)
   references READERS (NUM_READER)
go
alter table RELATION_67
   add constraint FK_RELATION_IOIINEONY_BOOKS foreign key (ISBN)
   references BOOKS (ISBN)
go
alter table RELATION_67
   add constraint FK_RELATION_I_AANOAAE_CATALOG foreign key (KW_KOD)
   references CATALOG (KW_KOD)
go
```

В языке SQL присутствует и операция удаления таблиц. Синтаксис этой операции предельно прост:

```
<Удалить таблицу>::= DROP TABLE <имя таблицы> [CASCADE | RESTRICT]
```

Параметр CASCADE означает, что при удалении таблицы одновременно удаляются и все объекты, связанные с ней. С таблицей, кроме рассмотренных ранее ограничений, могут быть связаны также объекты типа триггеров и представления. Понятие представления будет рассмотрено в следующем подразделе, а триггеров мы коснемся в разделах, связанных с архитектурой клиент-сервер. Однако операция удаления объектов определяется еще правами пользователей, что связано с концепцией безопасности в базах данных. Это значит, что если вы не являетесь владельцем объекта, то вы можете не иметь прав на его удаление. И в этом случае синтаксически правильный оператор DROP TABLE не может быть выполнен системой в силу отсутствия прав на удаление связанных с удаляемой

таблицей объектов. Кроме того, операция удаления таблицы не должна нарушать целостность базы данных, поэтому удалять таблицу, на которую имеются ссылки других таблиц, невозможно.

Например, в нашей схеме, связанной с библиотекой, мы не можем удалить ни таблицу BOOKS, ни таблицу READERS, ни таблицу CATALOG. У этих таблиц есть связь с подчиненными таблицами EXEMPLAR и RELATION_67. Поэтому если вы хотите удалить некоторый набор таблиц, то сначала необходимо грамотно построить последовательность их удаления, которая не нарушит базовых принципов поддержки целостности вашей схемы БД. В нашем примере последовательность операторов удаления таблиц может быть следующей:

```
DROP TABLE EXEMPLAR
DROP TABLE RELATION_67
DROP TABLE CATALOG
DROP TABLE READERS
DROP TABLE BOOKS
```

Понятие представления операции создания представлений

Для описания внешних моделей в реляционной модели могут использоваться представления. *Представление* (*View*) — это SQL-запрос на выборку, который пользователь воспринимает как некоторое виртуальное отношение. Задание представлений входит в описание схемы БД в реляционных СУБД. Представления позволяют скрыть ненужные несущественные детали для разных пользователей, модифицировать реальные структуры данных в удобном для приложений виде и, наконец, разграничить права доступа к данным и тем самым повысить защиту данных от несанкционированного доступа.

В отличие от реальной таблицы представление в том виде, как оно сконструировано, не существует в базе данных, это действительно только виртуальное отношение, хотя все данные, которые представлены в нем, действительно существуют в базе данных, но в разных отношениях. Они скомпонованы для пользователя в удобном виде из реальных таблиц с помощью некоторого запроса. Однако пользователь может этого не знать, он может обращаться с этим представлением как со стандартной таблицей. Представление при создании получает некоторое уникальное имя, его описание хранится в описании схемы базы данных, и СУБД в любой момент времени при обращении к этому представлению выполняет запрос, соответствующий его описанию, поэтому пользователь, работая с представлением, в каждый момент времени видит действительно реальные, актуальные на настоящий момент данные. Оно формируется как бы на лету, в момент обращения.

Оператор определения представления имеет следующий вид:

```
<создание представления>::= CREATE VIEW <имя представления>
[ (<список столбцов>)] AS <SQL-запрос>
```


При необходимости в представлении можно задать новое имя для каждого столбца виртуальной таблицы. При этом надо помнить, что если указывается список столбцов, то он должен содержать ровно столько столбцов, сколько содержит их SQL-запрос.

Если список имен столбцов в представлении не задан, то каждый столбец представления получает имя соответствующего столбца запроса.

Рассмотрим типичные виды представлений и их назначение.

Горизонтальное представление

Этот вид представления широко применяется для уменьшения объема реальных таблиц в обработке и ограничения доступа пользователей к закрытой для них информации. Так, например, правилом хорошего тона считается, что руководитель подразделения в некоторой фирме может видеть оклады и результаты работы только своих сотрудников, в этом случае для него создается горизонтальное представление, в которое загружены строки общей таблицы сотрудников, работающих в его подразделении.

Например, у нас есть таблица «Сотрудник» (EMPLOYEE) с полями «Табельный номер» (T_NUM), «ФИО» (NAME), «должность»(POSITION), «оклад»(SALARY), «надбавка»(PREMIUM), «отдел» (DEPARTMENT).

Для приложения, с которым работает начальник отдела продаж, будет создано представление

```
CREATE VIEW SAL_DEPT
   AS
   SELECT *
   FROM EMPLOYEE
   WHERE DEPARTMENT= "Отдел продаж"
```

Вертикальное представление

Этот вид представления практически соответствует выполнению операции проектирования некоторого отношения на ряд столбцов. Он используется в основном для скрытия информации, которая не должна быть доступна в конкретной внешней модели.

Например, для работника табельной службы, который учитывает присутствие сотрудников на работе, информация об окладе и надбавке должна быть закрыта. Для него можно создать следующее вертикальное представление:

```
CREATE VIEW TABEL
   AS
   SELECT T_NUM,NAME, POSITION, DEPARTMENT
   FROM EMPLOYEE
```

Сгруппированные представления

Эти представления содержат запросы, которые имеют группировку. Сгруппированные представления всегда должны содержать список столбцов. Они могут использовать агрегированные функции в качестве результирующих столбцов, а в дальнейшем это представление может использоваться как виртуальная таблица, например, в других запросах.

Создадим представление, которое определяет суммарный фон заработной платы и надбавок по каждому подразделению с указанием количества сотрудников, минимальной, максимальной и средней зарплаты и надбавки по подразделению. Такой запрос позволяет сравнить заработную плату и надбавки прямо по всем подразделениям, и он может быть очень эффективно использован администрацией при проведении сравнительного анализа подразделений фирмы.

```
CREATE VIEW RATE
DEPARTMENT, COUNT(*), SUM(SALARY), SUM(PREMIUM), MAX(SALARY), MIN(SALARY),
   AVERAGE (SALARY), MAX(PREMIUM), MIN(PREMIUM), AVERAGE (PREMIUM)
AS
SELECT DEPARTMENT, COUNT(*), SUM(SALARY), SUM(PREMIUM), MAX(SALARY),
   MIN(SALARY), AVERAGE (SALARY), MAX(PREMIUM), MIN(PREMIUM),
   AVERAGE (PREMIUM)
FROM EMPLOYEE
GROUP BY DEPARTMENT
```

Объединенные представления

Часто представления базируются на многотабличных запросах. Такое использование позволяет упростить разработку пользовательского интерфейса, сохранив при этом корректность схемы базы данных. Для примера снова обратимся к базе данных «Библиотека» и создадим представление, которое содержит список читателей-должников с указанием книг, которые у них на руках, и указанных в базе сроков сдачи этих книг. Такое представление может понадобиться для административного приложения, которое разрабатывается для директора библиотеки или его заместителя, они должны принимать административные меры для наказания нарушителей и возврата книг в библиотеку.

```
CREATE VIEW DEBTORS
ISBN,TITLE, NUM_READER,NAME,ADRES,HOME_PHON, WORK_PHON,DATA_OUT
AS
SELECT ISBN,TITLE,NUM_READER,NAME,ADRES,HOME_PHON, WORK_PHON,DATA_OUT
FROM BOOKS,EXEMPLAR,READERS
WHERE BOOKS.ISBN = EXEMPLAR.ISBN AND
EXEMPLAR.NUM_READER = READERS.NUM_READER AND
EXEMPLAR.PRESENT = FALSE AND
EXEMPLAR.DATA_OUT < GetDate()
```


Ограничение стандарта SQL1 на обновление представлений

Несмотря на то, что для пользователей представления выглядят как реальные отношения, существует ряд ограничений на операции модификации данных, связанные с представлениями.

СУБД может обновлять данные через представления только в том случае, если она может однозначно сопоставить каждой строке представления строку из реальной таблицы базы данных, а для каждого обновляемого столбца представления однозначно определить исходный столбец исходной таблицы базы данных. Далеко не для всех запросов это возможно сделать. Действительно, запросы с группировкой, сложные запросы с подзапросами возвращают результат, который СУБД не сможет однозначно интерпретировать в терминах реальных таблиц БД.

Согласно стандарту, представление можно обновлять только в том случае, когда его запрос соответствует следующим требованиям:

- В запросе должен отсутствовать предикат DISTINCT, то есть повторяющиеся строки не должны исключаться из таблицы результатов запроса.
- В предложении FROM должна быть задана только одна таблица, которую можно обновлять, то есть у представления должна быть только одна исходная таблица (это горизонтальное или вертикальное представление), а пользователь должен иметь соответствующие права доступа к ней. Если таблица сама является представлением, то она тоже должна удовлетворять данным условиям.
- Каждое имя в списке возвращаемых столбцов должно быть ссылкой на простой столбец: в списке не должны содержаться выражения, вычисляемые столбцы или агрегатные функции.
- В предложении WHERE не должен стоять вложенный запрос; в нем могут присутствовать только простые условия поиска.
- В запросе не должно присутствовать выражение группировки GROUP BY или HAVING.
- Однако в ряде коммерческих СУБД эти требования смягчены и операции модификации разрешены для более широкого класса представлений.

ГЛАВА 9 Физические модели баз данных

Физические модели баз данных определяют способы размещения данных в среде хранения и способы доступа к этим данным, которые поддерживаются на физическом уровне. Исторически первыми системами хранения и доступа были файловые структуры и системы управления файлами (СУФ), которые фактически являлись частью операционных систем. СУБД создавала над этими файловыми моделями свою надстройку, которая позволяла организовать всю совокупность файлов таким образом, чтобы она работала как единое целое и получала централизованное управление от СУБД. Однако непосредственный доступ осуществлялся на уровне файловых команд, которые СУБД использовала при манипулировании всеми файлами, составляющими хранимые данные одной или нескольких баз данных.

Однако механизмы буферизации и управления файловыми структурами не приспособлены для решения задач собственно СУБД, эти механизмы разрабатывались просто для традиционной обработки файлов, и с ростом объемов хранимых данных они стали неэффективными для использования СУБД. Тогда постепенно произошел переход от базовых файловых структур к непосредственному управлению размещением данных на внешних носителях самой СУБД. И пространство внешней памяти уже выходило из-под владения СУФ и управлялось непосредственно СУБД. При этом механизмы, применяемые в файловых системах, перешли во многом и в новые системы организации данных во внешней памяти, называемые чаще страничными системами хранения информации. Поэтому наш раздел, посвященный физическим моделям данных, мы начнем с обзора файлов и файловых структур, используемых для организации физических моделей, применяемых в базах данных, а в конце ознакомимся с механизмами организации данных во внешней памяти, использующими страничный принцип организации.

Файловые структуры, используемые для хранения информации в базах данных

В каждой СУБД по-разному организованы хранение и доступ к данным, однако существуют некоторые файловые структуры, которые имеют общепринятые способы организации и широко применяются практически во всех СУБД.

В системах баз данных файлы и файловые структуры, которые используются для хранения информации во внешней памяти, можно классифицировать следующим образом (см. рис. 9.1).

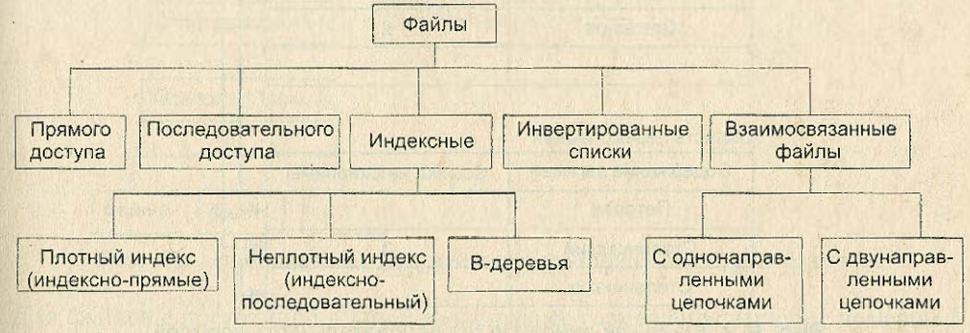


Рис. 9.1. Классификация файлов, используемых в системах баз данных

С точки зрения пользователя, *файлом* называется поименованная линейная последовательность записей, расположенных на внешних носителях. На рис. 9.2 представлена такая условная последовательность записей.

Так как файл — это линейная последовательность записей, то всегда в файле можно определить текущую запись, предшествующую ей и следующую за ней. Всегда существует понятие первой и последней записи файла. Не будем вдаваться в особенности физической организации внешней памяти, выделим в ней те черты, которые существенны для рассмотрения нашей темы.

В соответствии с методами управления доступом различают устройства внешней памяти *с произвольной адресацией* (магнитные и оптические диски) и устройства *с последовательной адресацией* (магнитофоны, стримеры).

На устройствах с произвольной адресацией теоретически возможна установка головок чтения-записи в произвольное место мгновенно. Практически существует время позиционирования головки, которое весьма мало по сравнению со временем считывания-записи.

В устройствах с последовательным доступом для получения доступа к некоторому элементу требуется «перемотать (пройти)» все предшествующие ему элементы информации. На устройствах с последовательным доступом вся память рассматривается как линейная последовательность информационных элементов (см. рис. 9.3).

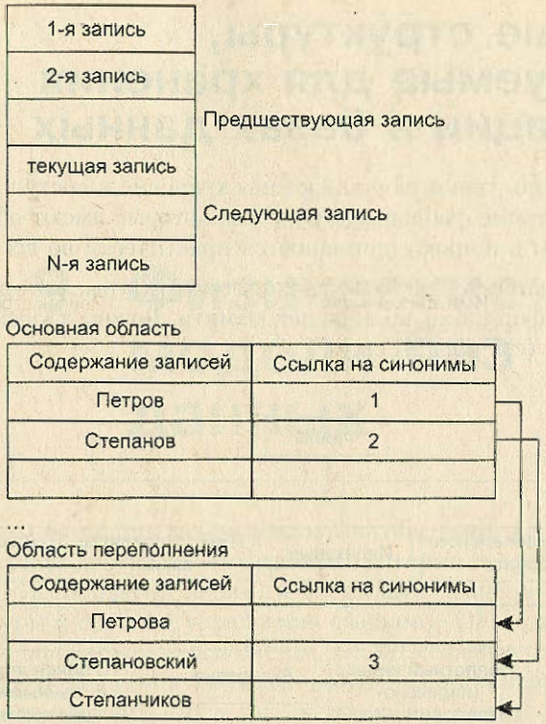


Рис. 9.2. Файл как линейная последовательность записей

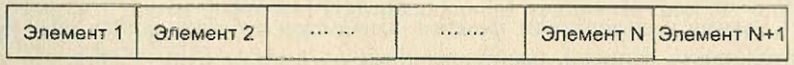


Рис. 9.3. Модель хранения информации на устройстве последовательного доступа

Файлы с постоянной длиной записи, расположенные на устройствах прямого доступа (УПД), являются *файлами прямого доступа*.

В этих файлах физический адрес расположения нужной записи может быть вычислен по номеру записи (NZ).

Каждая файловая система СУФ — система управления файлами поддерживает некоторую иерархическую файловую структуру, включающую чаще всего неограниченное количество уровней иерархии в представлении внешней памяти (см. рис. 9.4).

Для каждого файла в системе хранится следующая информация:

- имя файла;
- тип файла (например, расширение или другие характеристики);
- размер записи;
- количество занятых физических блоков;
- базовый начальный адрес;

- ссылка на сегмент расширения;
- способ доступа (код защиты).

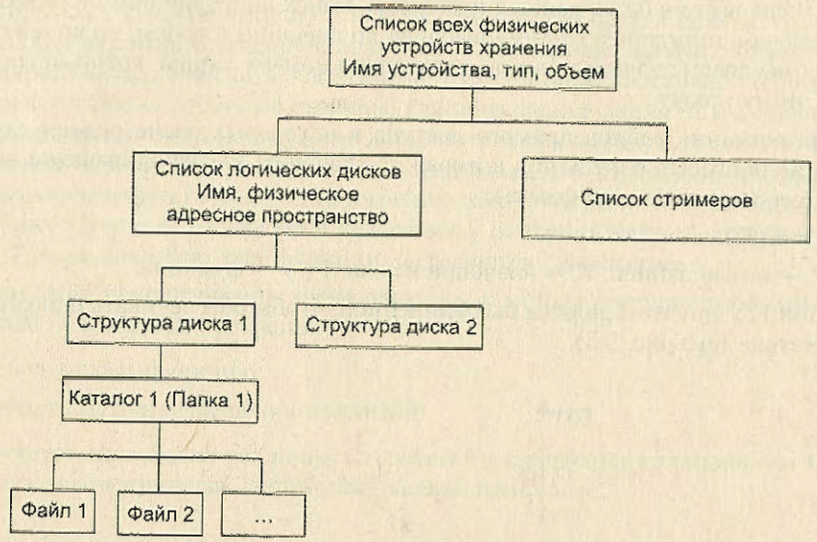


Рис. 9.4. Иерархическая организация файловой структуры хранения

Для файлов с постоянной длиной записи адрес размещения записи с номером К может быть вычислен по формуле:

$$BA + (K - 1) * LZ + 1,$$

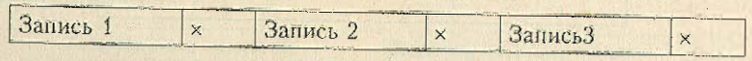
где BA — базовый адрес, LZ — длина записи.

И как мы уже говорили ранее, если можно всегда определить адрес, на который необходимо позиционировать механизм считывания-записи, то устройства прямого доступа делают это практически мгновенно, поэтому для таких файлов чтение произвольной записи практически не зависит от ее номера. Файлы прямого доступа обеспечивают наиболее быстрый доступ к произвольным записям, и их использование считается наиболее перспективным в системах баз данных.

На устройствах последовательного доступа могут быть организованы файлы только последовательного доступа.

Файлы с переменной длиной записи всегда являются файлами последовательного доступа. Они могут быть организованы двумя способами:

1. Конец записи отличается специальным маркером.



2. В начале каждой записи записывается ее длина.

Здесь LZN — длина N-й записи.

Файлы с прямым доступом обеспечивают наиболее быстрый способ доступа. Мы не всегда можем хранить информацию в виде файлов прямого доступа, но главное — это то, что доступ по номеру записи в базах данных весьма неэффективен. Чаще всего в базах данных необходим поиск по первичному или возможному ключам, иногда необходима выборка по внешним ключам, но во всех этих случаях мы знаем значение ключа, но не знаем номера записи, который соответствует этому ключу.

При организации файлов прямого доступа в некоторых очень редких случаях возможно построение функции, которая по значению ключа однозначно вычисляет адрес (номер записи файла).

NZ = F(K),

где NZ — номер записи, K — значение ключа, F( ) — функция.

Функция F() при этом должна быть линейной, чтобы обеспечивать однозначное соответствие (см. рис. 9.5).

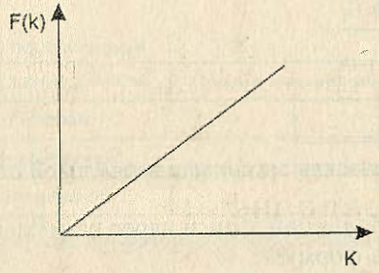


Рис. 9.5. Пример линейной функции пересчета значения ключа в номер записи

Однако далеко не всегда удается построить взаимно-однозначное соответствие между значениями ключа и номерами записей.

Часто бывает, что значения ключей разбросаны по нескольким диапазонам (см. рис. 9.6).

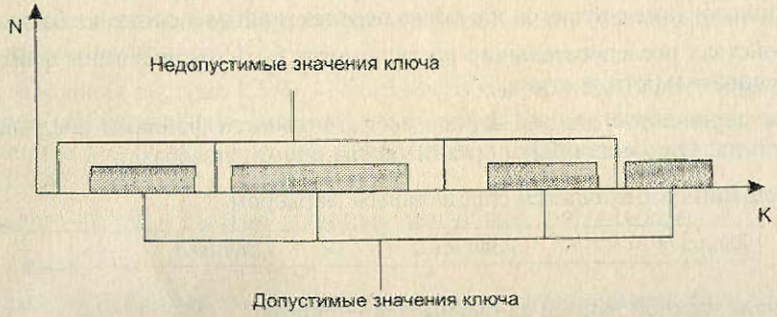


Рис. 9.6. Допустимые значения ключа

В этом случае не удается построить взаимно-однозначную функцию, либо эта функция будет иметь множество незадействованных значений, которые соответствуют недопустимым значениям ключа. В подобных случаях применяют различные методы хэширования (рандомизации) и создают специальные хэш- функции.

Суть методов хэширования состоит в том, что мы берем значения ключа (или некоторые его характеристики) и используем его для начала поиска, то есть мы вычисляем некоторую хэш-функцию h(k) и полученное значение берем в качестве адреса начала поиска. То есть мы не требуем полного взаимно-однозначного соответствия, но, с другой стороны, для повышения скорости мы ограничиваем время этого поиска (количество дополнительных шагов) для окончательного получения адреса. Таким образом, мы допускаем, что нескольким разным ключам может соответствовать одно значение хэш-функции (то есть один адрес). Подобные ситуации называются *коллизиями*. Значения ключей, которые имеют одно и то же значение хэш-функции, называются *синонимами*.

Поэтому при использовании хэширования как метода доступа необходимо принять два независимых решения:

- выбрать хэш-функцию;
- выбрать метод разрешения коллизий.

Существует множество различных стратегий разрешения коллизий, но мы для примера рассмотрим две достаточно распространенные.

Стратегия разрешения коллизий с областью переполнения

Первая стратегия условно может быть названа стратегией с областью переполнения.

При выборе этой стратегии область хранения разбивается на 2 части:

- основную область;
- область переполнения.

Для каждой новой записи вычисляется значение хэш-функции, которое определяет адрес ее расположения, и запись заносится в основную область в соответствии с полученным значением хэш-функции.

Основная область:

Содержание записей	Ссылка на синонимы
Петров	1
Сахаров	3

Область переполнения:

Содержание записей	Ссылка на синонимы
Петрова	2
Петровский	
Сахарочкин	

Если вновь заносимая запись имеет значение функции хэширования такое же, которое использовала другая запись, уже имеющаяся в БД, то новая запись заносится в область переполнения на первое свободное место, а в записи-синониме, которая находится в основной области, делается ссылка на адрес вновь размещенной записи в области переполнения. Если же уже существует ссылка в записи-синониме, которая расположена в основной области, то тогда новая запись получает дополнительную информацию в виде ссылки и уже в таком виде заносится в область переполнения.

При этом цепочка синонимов не разрывается, но мы не просматриваем ее до конца, чтобы расположить новую запись в конце цепочки синонимов, а располагаем всегда новую запись на второе место в цепочке синонимов, что существенно сокращает время размещения новой записи. При таком алгоритме время размещения любой новой записи составляет не более двух обращений к диску, с учетом того, что номер первой свободной записи в области переполнения хранится в виде системной переменной.

Рассмотрим теперь механизмы поиска произвольной записи и удаления записи для этой стратегии хэширования.

При поиске записи также сначала вычисляется значение ее хэш-функции и считывается первая запись в цепочке синонимов, которая расположена в основной области. Если искомая запись не соответствует первой в цепочке синонимов, то далее поиск происходит перемещением по цепочке синонимов, пока не будет обнаружена требуемая запись. Скорость поиска зависит от длины цепочки синонимов, поэтому качество хэш-функции определяется максимальной длиной цепочки синонимов. Хорошим результатом может считаться наличие не более 10 синонимов в цепочке.

При удалении произвольной записи сначала определяется ее место расположения. Если удаляемой является первая запись в цепочке синонимов, то после удаления на ее место в основной области заносится вторая (следующая) запись в цепочке синонимов, при этом все указатели (ссылки на синонимы) сохраняются.

Если же удаляемая запись находится в середине цепочки синонимов, то необходимо провести корректировку указателей: в записи, предшествующей удаляемой, в цепочке ставится указатель из удаляемой записи. Если это последняя запись в цепочке, то все равно механизм изменения указателей такой же, то есть в предшествующую запись заносится признак отсутствия следующей записи в цепочке, который ранее хранился в последней записи.

Организация стратегии свободного замещения

При этой стратегии файловое пространство не разделяется на области, но для каждой записи добавляется 2 указателя: указатель на предыдущую запись в цепочке синонимов и указатель на следующую запись в цепочке синонимов. Отсутствие соответствующей ссылки обозначается специальным символом, например нулем. Для каждой новой записи вычисляется значение хэш-функции, и если данный адрес свободен, то запись попадает на заданное место и становится первой в цепочке синонимов. Если адрес, соответствующий полученному значению

хэш-функции, занят, то по наличию ссылок определяется, является ли запись, расположенная по указанному адресу, первой в цепочке синонимов. Если да, то новая запись располагается на первом свободном месте и для нее устанавливаются соответствующие ссылки: она становится второй в цепочке синонимов, на нее ссылается первая запись, а она ссылается на следующую, если таковая есть.

Если запись, которая занимает требуемое место, не является первой записью в цепочке синонимов, значит, она занимает данное место «незаконно» и при появлении «законного владельца» должна быть «выселена», то есть перемещена на новое место. Механизм перемещения аналогичен занесению новой записи, которая уже имеет синоним, занесенный в файл. Для этой записи ищется первое свободное место и корректируются соответствующие ссылки: в записи, которая является предыдущей в цепочке синонимов для перемещаемой записи, заносится указатель на новое место перемещаемой записи, указатели же в самой перемещаемой записи остаются прежние.

После перемещения «незаконной» записи вновь вносимая запись занимает свое законное место и становится первой записью в новой цепочке синонимов.

Механизмы удаления записей во многом аналогичны механизмам удаления в стратегии с областью переполнения. Однако еще раз кратко опишем их.

Если удаляемая запись является первой записью в цепочке синонимов, то после удаления на ее место перемещается следующая (вторая) запись из цепочки синонимов и проводится соответствующая корректировка указателя третьей записи в цепочке синонимов, если таковая существует.

Если же удаляется запись, которая находится в середине цепочки синонимов, то производится только корректировка указателей: в предшествующей записи указатель на удаляемую запись заменяется указателем на следующую за удаляемой запись, а в записи, следующей за удаляемой, указатель на предыдущую запись заменяется на указатель на запись, предшествующую удаляемой.

Вопросы для самостоятельной работы

1. Сравнить обе стратегии и определить, какая из них будет наиболее перспективной и в каких случаях.
2. Разработать алгоритмы удаления записей для первой и второй стратегий. Показать, как определяются ссылки.

Индексные файлы

Несмотря на высокую эффективность хэш-адресации, в файловых структурах далеко не всегда удается найти соответствующую функцию, поэтому при организации доступа по первичному ключу широко используются индексные файлы. В некоторых коммерческих системах индексными файлами называются также и файлы, организованные в виде инвертированных списков, которые используются для доступа по вторичному ключу. Мы будем придерживаться классической интерпретации индексных файлов и надеемся, что если вы столкнетесь с иной

интерпретацией, то сумеете разобраться в сути, несмотря на некоторую путаницу в терминологии. Наверное, это отчасти связано с тем, что область баз данных является достаточно молодой областью знаний, и несмотря на то, что здесь уже выработалась определенная терминология, многие поставщики коммерческих СУБД предпочитают свой упрощенный сленг при описании собственных продуктов. Иногда это связано с тем, что в целях рекламы они не хотят ссылаться на старые, хорошо известные модели и методы организации информации в системе, а изобретают новые названия при описании своих моделей, тем самым пытаясь разрекламировать эффективность своих продуктов. Хорошее знание принципов организации данных поможет вам объективно оценивать решения, предлагаемые поставщиками современных СУБД, и не попадаться на рекламные крючки.

Индексные файлы можно представить как файлы, состоящие из двух частей. Это не обязательно физическое совмещение этих двух частей в одном файле, в большинстве случаев индексная область образует отдельный индексный файл, а основная область образует файл, для которого создается индекс. Но нам удобнее рассматривать эти две части совместно, так как именно взаимодействие этих частей и определяет использование механизма индексации для ускорения доступа к записям.

Мы предполагаем, что сначала идет индексная область, которая занимает некоторое целое число блоков, а затем идет основная область, в которой последовательно расположены все записи файла.

В зависимости от организации индексной и основной областей различают 2 типа файлов: с *плотным индексом* и с *неплотным индексом*. Эти файлы имеют еще дополнительные названия, которые напрямую связаны с методами доступа к произвольной записи, которые поддерживаются данными файловыми структурами.

Файлы с плотным индексом называются также индексно-прямыми файлами, а файлы с неплотным индексом называются также индексно-последовательными файлами. Смысл этих названий нам будет ясен после того, как мы более подробно рассмотрим механизмы организации данных файлов.

Файлы с плотным индексом, или индексно-прямые файлы

Рассмотрим *файлы с плотным индексом*. В этих файлах основная область содержит последовательность записей одинаковой длины, расположенных в произвольном порядке, а структура индексной записи в них имеет следующий вид:

Значение ключа	Номер записи

Здесь *значение ключа* — это значение первичного ключа, а *номер записи* — это порядковый номер записи в основной области, которая имеет данное значение первичного ключа.

Так как индексные файлы строятся для первичных ключей, однозначно определяющих запись, то в них не может быть двух записей, имеющих одинаковые значения первичного ключа. В индексных файлах с плотным индексом для каждой записи в основной области существует одна запись из индексной области. Все записи в индексной области упорядочены по значению ключа, поэтому можно применить более эффективные способы поиска в упорядоченном пространстве.

Длина доступа к произвольной записи оценивается не в абсолютных значениях, а в количестве обращений к устройству внешней памяти, которым обычно является диск. Именно обращение к диску является наиболее длительной операцией по сравнению со всеми обработками в оперативной памяти.

Наиболее эффективным алгоритмом поиска на упорядоченном массиве является логарифмический, или бинарный, поиск. Очень хорошо изложил этот алгоритм барон Мюнхгаузен, когда он объяснял, как поймать льва в пустыне. При этом все пространство поиска разбивается пополам, и так как оно строго упорядочено, то определяется сначала, не является ли элемент искомым, а если нет, то в какой половине его надо искать. Следующим шагом мы определенную половину также делим пополам и производим аналогичные сравнения, и т. д., пока не обнаружим искомый элемент. Максимальное количество шагов поиска определяется двоичным логарифмом от общего числа элементов в искомом пространстве поиска:

$$T_n = \log_2 N,$$

где N — число элементов.

Однако в нашем случае является существенным только число обращений к диску при поиске записи по заданному значению первичного ключа. Поиск происходит в индексной области, где применяется двоичный алгоритм поиска индексной записи, а потом путем прямой адресации мы обращаемся к основной области уже по конкретному номеру записи. Для того чтобы оценить максимальное время доступа, нам надо определить количество обращений к диску для поиска произвольной записи.

На диске записи файлов хранятся в блоках. Размер блока определяется физическими особенностями дискового контроллера и операционной системой. В одном блоке могут размещаться несколько записей. Поэтому нам надо определить количество индексных блоков, которое потребуется для размещения всех требуемых индексных записей, а потому максимальное число обращений к диску будет равно двоичному логарифму от заданного числа блоков плюс единица. Зачем нужна единица? После поиска номера записи в индексной области мы должны еще обратиться к основной области файла. Поэтому формула для вычисления максимального времени доступа в количестве обращений к диску выглядит следующим образом:

$$T_n = \log_2 N_{бл.инд.} + 1.$$

Давайте рассмотрим конкретный пример и сравним время доступа при последовательном просмотре и при организации плотного индекса.

Допустим, что мы имеем следующие исходные данные:

Длина записи файла (LZ) — 128 байт. Длина первичного ключа (LK) — 12 байт. Количество записей в файле (KZ) — 100000. Размер блока (LB) — 1024 байт.

Рассчитаем размер индексной записи. Для представления целого числа в пределах 100000 нам потребуется 3 байта, можем считать, что у нас допустима только четная адресация, поэтому нам надо отвести 4 байта для хранения номера записи, тогда длина индексной записи будет равна сумме размера ключа и ссылки на номер записи, то есть:

LI = LK + 4 = I4 + 4 = 16 байт.

Определим количество индексных блоков, которое требуется для обеспечения ссылок на заданное количество записей. Для этого сначала определим, сколько индексных записей может храниться в одном блоке:

KIZB = LB/LI = 1024/16 = 64 индексных записи в одном блоке.

Теперь определим необходимое количество индексных блоков:

KIB = KZ/KZIB = 100000/64 = 1563 блока.

Мы округлили в большую сторону, потому что пространство выделяется целыми блоками, и последний блок у нас будет заполнен не полностью.

А теперь мы уже можем вычислить максимальное количество обращений к диску при поиске произвольной записи:

$T_{поиска} = \log_2 KIB + 1 = \log_2 1563 + 1 = 11 + 1 = 12$ обращений к диску.

Логарифм мы тоже округляем, так как считаем количество обращений, а оно должно быть целым числом.

Следовательно, для поиска произвольной записи по первичному ключу при организации плотного индекса потребуется не более 12 обращений к диску. А теперь оценим, какой выигрыш мы получаем, ведь организация индекса связана с дополнительными накладными расходами на его поддержку, поэтому такая организация может быть оправдана только в том случае, когда она действительно дает значительный выигрыш. Если бы мы не создавали индексное пространство, то при произвольном хранении записей в основной области нам бы в худшем случае было необходимо просмотреть все блоки, в которых хранится файл, временем просмотра записей внутри блока мы пренебрегаем, так как этот процесс происходит в оперативной памяти.

Количество блоков, которое необходимо для хранения всех 100 000 записей, мы определим по следующей формуле:

KBO = KZ/(LB/LZ) = 100000/(1024/128) = 12500 блоков.

И это означает, что максимальное время доступа равно 12500 обращений к диску. Да, действительно, выигрыш существенный.

Рассмотрим, как осуществляются операции добавления и удаления новых записей.

При операции добавления осуществляется запись в конец основной области. В индексной области необходимо произвести занесение информации в конкретное место, чтобы не нарушать упорядоченности. Поэтому вся индексная область файла разбивается на блоки и при начальном заполнении в каждом блоке остается свободная область (процент расширения) (рис. 9.7):

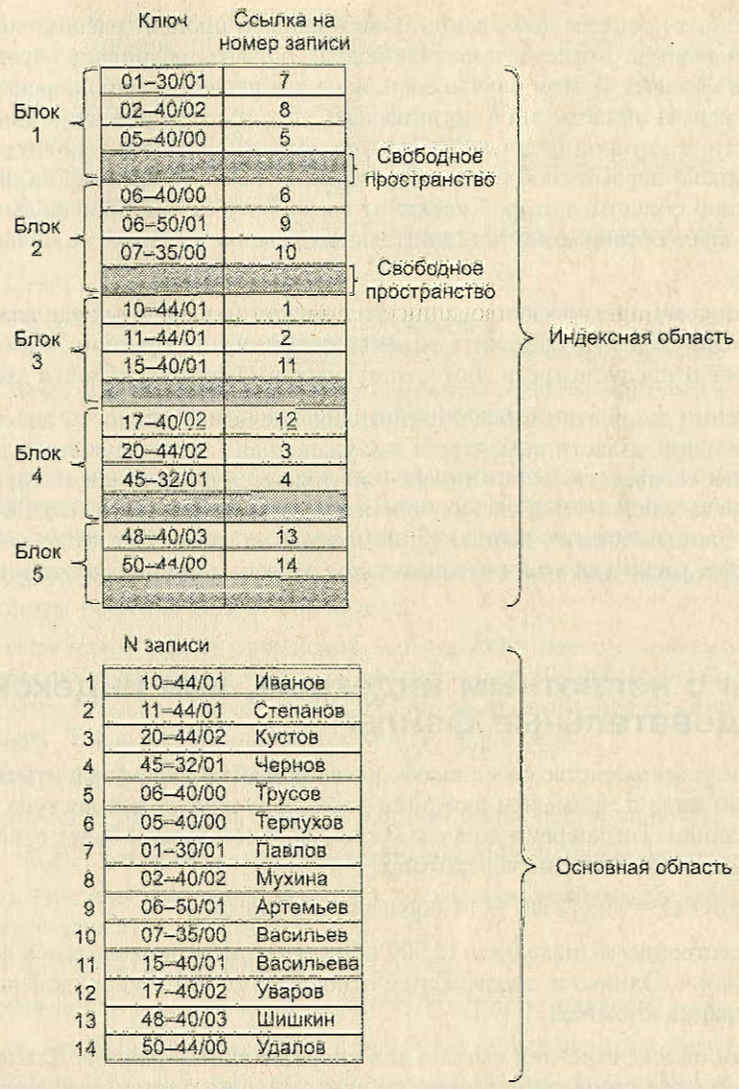


Рис. 9.7. Пример организации файла с плотным индексом

После определения блока, в который должен быть занесен индекс, этот блок копируется в оперативную память, там он модифицируется путем вставки в нужное место новой записи (благо в оперативной памяти это делается на несколько порядков быстрее, чем на диске) и, измененный, записывается обратно на диск.

Определим максимальное количество обращений к диску, которое требуется при добавлении записи, — это количество обращений, необходимое для поиска записи плюс одно обращение для занесения измененного индексного блока и плюс одно обращение для занесения записи в основную область.

$T_{добавления} = \log_2 N + 1 + 1 + 1.$

Естественно, в процессе добавления новых записей процент расширения постоянно уменьшается. Когда исчезает свободная область, возникает переполнение индексной области. В этом случае возможны два решения: либо перестроить заново индексную область, либо организовать область переполнения для индексной области, в которой будут храниться не поместившиеся в основную область записи. Однако первый способ потребует дополнительного времени на перестройку индексной области, а второй увеличит время на доступ к произвольной записи и потребует организации дополнительных ссылок в блоках на область переполнения.

Именно поэтому при проектировании физической базы данных так важно заранее как можно точнее определить объемы хранимой информации, спрогнозировать ее рост и предусмотреть соответствующее расширение области хранения.

При удалении записи возникает следующая последовательность действий: запись в основной области помечается как удаленная (отсутствующая), в индексной области соответствующий индекс уничтожается физически, то есть записи, следующие за удаленной записью, перемещаются на ее место и блок, в котором хранился данный индекс, заново записывается на диск. При этом количество обращений к диску для этой операции такое же, как и при добавлении новой записи.

Файлы с неплотным индексом, или индексно-последовательные файлы

Попробуем усовершенствовать способ хранения файла: будем хранить его в упорядоченном виде и применим алгоритм двоичного поиска для доступа к произвольной записи. Тогда время доступа к произвольной записи будет существенно меньше. Для нашего примера это будет:

$$T = \log_2 KBO = \log_2 12500 = 14 \text{ обращений к диску.}$$

И это существенно меньше, чем 12 500 обращений при произвольном хранении записей файла. Однако и поддержание основного файла в упорядоченном виде также операция сложная.

Неплотный индекс строится именно для упорядоченных файлов. Для этих файлов используется принцип внутреннего упорядочения для уменьшения количества хранимых индексов. Структура записи индекса для таких файлов имеет следующий вид:

Значение ключа первой записи блока	Номер блока с этой записью

В индексной области мы теперь ищем нужный блок по заданному значению первичного ключа. Так как все записи упорядочены, то значение первой записи блока позволяет нам быстро определить, в каком блоке находится искомая запись. Все остальные действия происходят в основной области. На рис. 9.8 представлен пример заполнения основной и индексной областей, если первичным ключом являются целые числа.

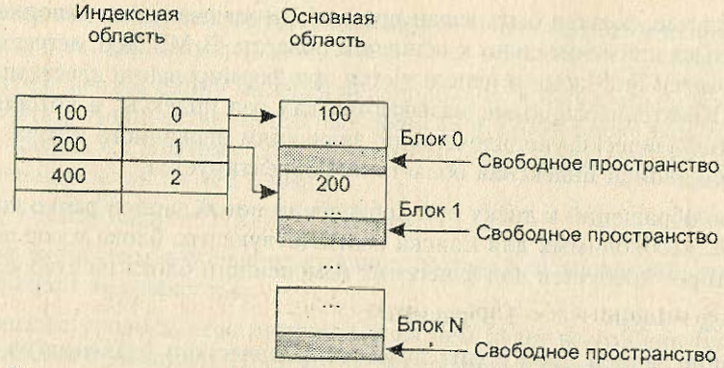


Рис. 9.8. Пример заполнения индексной и основной области при организации неплотного индекса

Время сортировки больших файлов весьма значительно, но поскольку файлы поддерживаются сортированными с момента их создания, накладные расходы в процессе добавления новой информации будут гораздо меньше.

Оценим время доступа к произвольной записи для файлов с неплотным индексом. Алгоритм решения задачи аналогичен.

Сначала определим размер индексной записи. Если ранее ссылка рассчитывалась исходя из того, что требовалось ссылаться на 100 000 записей, то теперь нам требуется ссылаться всего на 12 500 блоков, поэтому для ссылки достаточно двух байт. Тогда длина индексной записи будет равна:

$$LI = LK + 2 = 14 + 2 = 14 \text{ байт.}$$

Тогда количество индексных записей в одном блоке будет равно:

$$KIZB = LB/LI = 1024/14 = 73 \text{ индексные записи в одном блоке.}$$

Определим количество индексных блоков, которое необходимо для хранения требуемых индексных записей:

$$KIB = KBO/KZIB = 12500/73 = 172 \text{ блока.}$$

Тогда время доступа по прежней формуле будет определяться:

$$T_{\text{поиска}} = \log_2 KIB + 1 = \log_2 172 + 1 = 8 + 1 = 9 \text{ обращений к диску.}$$

Мы видим, что при переходе к неплотному индексу время доступа уменьшилось практически в полтора раза. Поэтому можно признать, что организация неплотного индекса дает выигрыш в скорости доступа.

Рассмотрим процедуры добавления и удаления новой записи при подобном индексе.

Здесь механизм включения новой записи принципиально отличен от ранее рассмотренного. Здесь новая запись должна заноситься сразу в требуемый блок на требуемое место, которое определяется заданным принципом упорядоченности на множестве значений первичного ключа. Поэтому сначала ищется требуемый блок основной памяти, в который надо поместить новую запись, а потом этот блок считывается, затем в оперативной памяти корректируется содержимое блока и он снова записывается на диск на старое место. Здесь, так же как и

в первом случае, должен быть задан процент первоначального заполнения блоков, но только применительно к основной области. В MS SQL server этот процент называется Full-factor и используется при формировании кластеризованных индексов. Кластеризованными называются как раз индексы, в которых исходные записи физически упорядочены по значениям первичного ключа. При внесении новой записи индексная область не корректируется.

Количество обращений к диску при добавлении новой записи равно количеству обращений, необходимых для поиска соответствующего блока плюс одно обращение, которое требуется для занесения измененного блока на старое место.

$T_{добавления} = \log_2 N + 1 + 1$ обращений.

Уничтожение записи происходит путем ее физического удаления из основной области, при этом индексная область обычно не корректируется, даже если удаляется первая запись блока. Поэтому количество обращений к диску при удалении записи такое же, как и при добавлении новой записи.

Организация индексов в виде B-tree (B-деревьев)

Калькированный термин «В-дерево», в котором смешивается английский символ «B» и добавочное слово на русском языке, настолько устоялся в литературе, посвященной организации физического хранения данных, что я не решусь его корректировать.

Встретив как-то термин «Б-дерево», я долго его трактовала, потому что привыкла уже к устоявшемуся обозначению. Поэтому будем работать с этим термином.

Построение B-деревьев связано с простой идеей построения индекса над уже построенным индексом. Действительно, если мы построим неплотный индекс, то сама индексная область может быть рассмотрена нами как основной файл, над которым надо снова построить неплотный индекс, а потом снова над новым индексом строим следующий и так до того момента, пока не останется всего один индексный блок.

Мы в общем случае получим некоторое дерево, каждый родительский блок которого связан с одинаковым количеством подчиненных блоков, число которых равно числу индексных записей, размещаемых в одном блоке. Количество обращений к диску при этом для поиска любой записи одинаково и равно количеству уровней в построенном дереве. Такие деревья называются сбалансированными (balanced) именно потому, что путь от корня до любого листа в этом дереве одинаков. Именно термин «сбалансированное» от английского «balanced» — «сбалансированный, взвешенный» и дал название данному методу организации индекса.

Построим подобное дерево для нашего примера и рассчитаем для него количество уровней и, соответственно, количество обращений к диску.

На первом уровне число блоков равно числу блоков основной области, это нам известно, — оно равно 12 500 блоков. Второй уровень образуется из неплотного индекса, мы его тоже уже строили и вычислили, что количество блоков индексной области в этом случае равно 172 блокам. А теперь над этим вторым уровнем снова построим неплотный индекс.

Мы не будем менять длину индексной записи, а будем считать ее прежней, равной 14 байтам. Количество индексных записей в одном блоке нам тоже известно, и оно равно 73. Поэтому сразу определим, сколько блоков нам необходимо для хранения ссылок на 172 блока.

$KIB_3 = KIB_2/KZIB = 172/73 = 3$ блока

Мы снова округляем в большую сторону, потому что последний, третий, блок будет заполнен не полностью.

И над третьим уровнем строим новый, и на нем будет всего один блок, в котором будет всего три записи. Поэтому число уровней в построенном дереве равно четырем, и соответственно количество обращений к диску для доступа к произвольной записи равно четырем (рис. 9.9). Это не максимально возможное число обращений, а всегда одно и то же, одинаковое для доступа к любой записи.

$T_д = R_{уровн.} = 4$

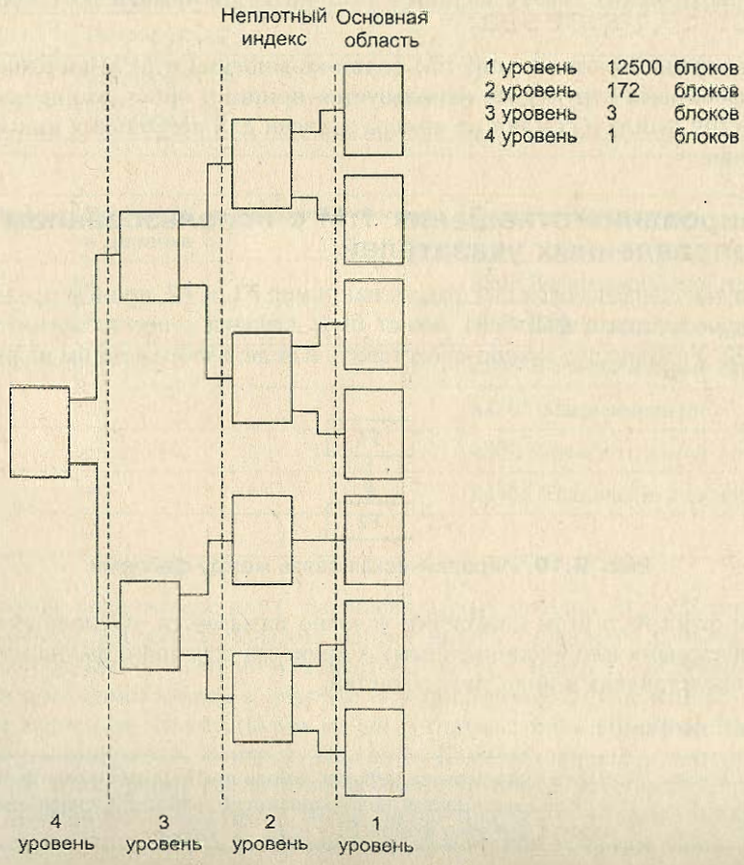


Рис. 9.9. Построенное B-дерево

Механизм добавления и удаления записи при организации индекса в виде B-дерева аналогичен механизму, применяемому в случае с неплотным индексом.

И наконец, последнее, что хотелось бы прояснить, — это наличие вторых названий для плотного и неплотного индексов.

В случае плотного индекса после определения местонахождения искомой записи доступ к ней осуществляется прямым способом по номеру записи, поэтому этот способ организации индекса и называется индексно-прямым.

В случае неплотного индекса после нахождения блока, в котором расположена искомая запись, поиск внутри блока требуемой записи происходит последовательным просмотром и сравнением всех записей блока. Поэтому способ индексации с неплотным индексом называется еще и индексно-последовательным.

Моделирование отношений «один-ко-многим» на файловых структурах

Отношение иерархии является типичным для баз данных, поэтому моделирование иерархических связей является типичным для физических моделей баз данных.

Для моделирования отношений 1:M (один-ко-многим) и M:M (многие-ко-многим) на файловых структурах используется принцип организации цепочек записей внутри файла и ссылки на номера записей для нескольких взаимосвязанных файлов.

Моделирование отношения 1:M с использованием однонаправленных указателей

В этом случае связываются два файла, например F1 и F2, причем предполагается, что одна запись в файле F1 может быть связана с несколькими записями в файле F2. Условно это можно представить в виде, изображенном на рис. 9.10.

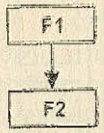


Рис. 9.10. Иерархическая связь между файлами

При этом файл F1 в этом комплексе условно называется «Основным», а файл F2 — «зависимым» или «подчиненным». Структура основного файла может быть условно представлена в виде трех областей.

«Основной файл» F1.

Ключ	Запись	Ссылка-указатель на первую запись в «Подчиненном» файле, с которой начинается цепочка записей файла F2, связанных с данной записью файла F1

В подчиненном файле также к каждой записи добавляется специальный указатель, в нем хранится номер записи, которая является следующей в цепочке записей «подчиненного» файла, связанной с одной записью «основного» файла.

Таким образом, каждая запись «подчиненного файла» делится на две области: область указателя и область, содержащую собственно запись.

Структура записи «подчиненного» файла.

Указатель на следующую запись в цепочке	Содержимое записи

В качестве примера рассмотрим связь между преподавателями и занятиями, которые эти преподаватели проводят. В файле F1 приведен список преподавателей, а в файле F2 — список занятий, которые они ведут.

F1		
Номер записи	**Ключ и остальная запись**	**Указатель**
1	Иванов И. Н. ...	1
2	Петров А. А.	3
3	Сидоров П. А.	2
4	Яковлев В. В.	

F2		
Номер записи	**Указатель на следующую запись в цепочке**	**Содержимое записи**
1	4	4306 Вычислительные сети
2	–	4307 Контроль и диагностика
3	6	4308 Вычислительные сети
4	5	84305 Моделирование
5	–	4309 Вычислительные сети
6	–	84405 Техническая диагностика
7	–	

В этом случае содержимое двух взаимосвязанных файлов F1 и F2 может быть расшифровано следующим образом: первая запись в файле F1 связана с цепочкой записей файла F2, которая начинается с записи номер 1, следующая запись номер 4 и последняя запись в цепочке — запись номер 5. Последняя — потому что пятая запись не имеет ссылки на следующую запись в цепочке. Аналогично можно расшифровать и остальные связи. Если мы проведем интерпретацию данных связей на уровне предметной области, то можно утверждать, что преподаватель Иванов ведет предмет « Вычислительные сети» в группе 4306, «Моделирование» в группе 84305 и «Вычислительные сети» в группе 4309.

Аналогично могут быть расшифрованы и остальные взаимосвязанные записи.

Алгоритм нахождения нужных записей «подчиненного» файла

- **Шаг 1.** Ищется запись в «основном» файле в соответствии с его организацией (с помощью функции хэширования, или с использованием индексов, или другим образом). Если требуемая запись найдена, то переходим к шагу 2, в противном случае выводим сообщение об отсутствии записи основного файла.
- **Шаг 2.** Анализируем указатель в основном файле если он пустой, то есть стоит прочерк, значит, для этой записи нет ни одной связанной с ней записи в «подчиненном файле», и выводим соответствующее сообщение, в противном случае переходим к шагу 3.
- **Шаг 3.** По ссылке-указателю в найденной записи основного файла переходим прямым методом доступа по номеру записи на первую запись в цепочке «Подчиненного» файла. Переходим к шагу 4.
- **Шаг 4.** Анализируем текущую запись на содержание если это искомая запись, то мы заканчиваем поиск, в противном случае переходим к шагу 5.
- **Шаг 5.** Анализируем указатель на следующую запись в цепочке если он пуст, то выводим сообщение, что искомая запись отсутствует, и прекращаем поиск, в противном случае по ссылке-указателю переходим на следующую запись в «подчиненном файле» и снова переходим к шагу 4.

Использование цепочек записей позволяет эффективно организовывать модификацию взаимосвязанных файлов.

Алгоритм удаления записи из цепочки «подчиненного» файла

- **Шаг 1.** Ищется удаляемая запись в соответствии с ранее рассмотренным алгоритмом. Единственным отличием при этом является обязательное сохранение в специальной переменной номера предыдущей записи в цепочке, допустим, это переменная NP.
- **Шаг 2.** Запоминаем в специальной переменной указатель на следующую запись в найденной записи, например, заносим его в переменную NS. Переходим к шагу 3.
- **Шаг 3.** Помечаем специальным символом, например символом звездочка (*), найденную запись, то есть в позиции указателя на следующую запись в цепочке ставим символ «*» — это означает, что данная запись отсутствует, а место в файле свободно и может быть занято любой другой записью.
- **Шаг 4.** Переходим к записи с номером, который хранится в NP, и заменяем в ней указатель на содержимое переменной NS.

Для того чтобы эффективно использовать дисковое пространство при включении новой записи в «подчиненный файл», ищется первое свободное место, т. е. запись, помеченная символом «*», и на ее место заносится новая запись, после этого производится модификация соответствующих указателей. При этом необходимо различать 3 случая:

1. Добавление записи на первое место в цепочке.
2. Добавление записи в конец цепочки.
3. Добавление записи на заданное место в цепочке.

Задание для самостоятельной работы

Разработать алгоритмы добавления записи для всех трех случаев

Однако часто бывает необходимо просматривать цепочку подчиненных записей в двух направлениях: прямом и обратном. В этом случае применяют двойные указатели.

В «основном файле» один указатель равен номеру первой записи в цепочке записей «подчиненного файла», а второй — номеру последней записи.

В «подчиненном файле» один указатель равен номеру следующей записи в цепочке, а другой — номеру предыдущей записи в цепочке. Для первой и последней записей в цепочке один из указателей пуст, то есть равен пробелу.

Для нашего примера это выглядит следующим образом:

F1			
Номер записи	**Ключ и остальная запись**	**Указатель на первую запись**	**Указатель на последнюю запись**
1	Иванов И. Н.	1	5
2	Петров А. А.	3	6
3	Сидоров П. А.	2	2
4	Яковлев В. В.		

F2			
Номер записи	**Указатель на предыдущую запись в цепочке**	**Указатель на следующую запись в цепочке**	**Содержимое записи**
1	–	4	4306 Вычислительные сети
2	–	–	4307 Контроль и диагностика
3	–	6	4308 Вычислительные сети
4	1	5	84305 Моделирование
5	4	–	4309 Вычислительные сети
6	3	–	84405 Техническая диагностика
7		–	

Один файл («подчиненный» или «основной») может быть связан с несколькими другими файлами, при этом для каждой связи моделируются свои указатели. Связь двух основных файлов F1 и F2 с одним связующим файлом F3 моделируется на

F1		
Ключ	Содержимое записи	Указатель на файл А

F2		
Ключ	Содержимое	Указатель на файл А

F3		
Цепочки для файла F1	Содержимое записи	Цепочки для файла F2

Инвертированные списки

До сих пор мы рассматривали структуры данных, которые использовались для ускорения доступа по первичному ключу. Однако достаточно часто в базах данных требуется проводить операции доступа по вторичным ключам. Напомним, что вторичным ключом является набор атрибутов, которому соответствует набор искомых записей. Это означает, что существует множество записей, имеющих одинаковые значения вторичного ключа. Например, в случае нашей БД «Библиотека» вторичным ключом может служить место издания, год издания. Множество книг могут быть изданы в одном месте, и множество книг могут быть изданы в один год.

Для обеспечения ускорения доступа по вторичным ключам используются структуры, называемые инвертированными списками, которые послужили основой организации индексных файлов для доступа по вторичным ключам.

Инвертированный список в общем случае — это двухуровневая индексная структура. Здесь на первом уровне находится файл или часть файла, в которой упорядоченно расположены значения вторичных ключей. Каждая запись с вторичным ключом имеет ссылку на номер первого блока в цепочке блоков, содержащих номера записей с данным значением вторичного ключа. На втором уровне находится цепочка блоков, содержащих номера записей, содержащих одно и то же значение вторичного ключа. При этом блоки второго уровня упорядочены по значениям вторичного ключа.

И наконец, на третьем уровне находится собственно основной файл.

Механизм доступа к записям по вторичному ключу при подобной организации записей весьма прост. На первом шаге мы ищем в области первого уровня заданное значение вторичного ключа, а затем по ссылке считываем блоки второго уровня, содержащие номера записей с заданным значением вторичного ключа, а далее уже прямым доступом загружаем в рабочую область пользователя содержимое всех записей, содержащих заданное значение вторичного ключа.

На рис. 9.11 представлен пример инвертированного списка, составленного для вторичного ключа «Номер группы» в списке студентов некоторого учебного заведения. Для более наглядного представления мы ограничили размер блока пятью записями (целыми числами).

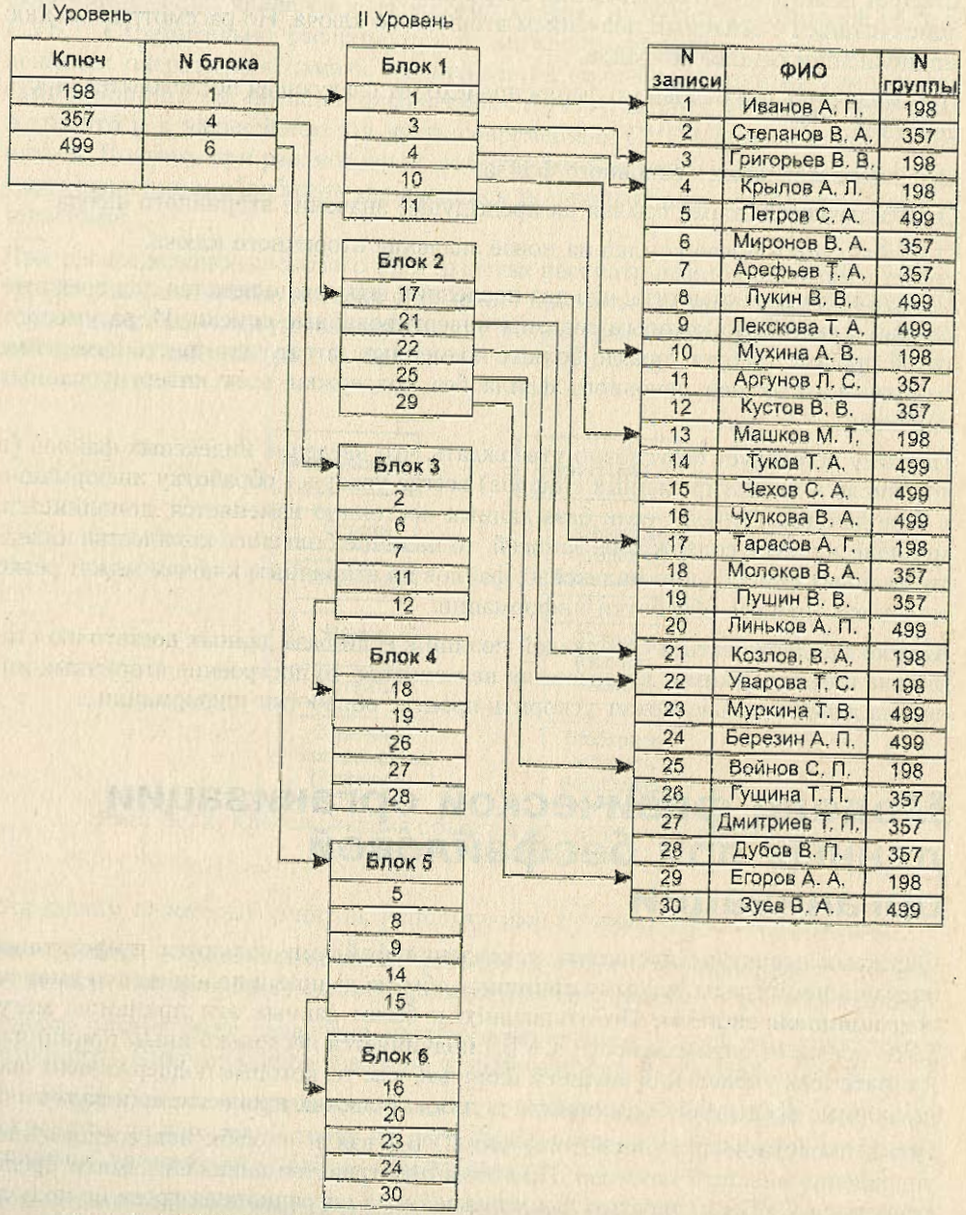


Рис. 9.11. Построение инвертированного списка по номеру группы для списка студентов

Для одного основного файла может быть создано несколько инвертированных списков по разным вторичным ключам.

Следует отметить, что организация вторичных списков действительно ускоряет поиск записей с заданным значением вторичного ключа. Но рассмотрим вопрос модификации основного файла.

При модификации основного файла происходит следующая последовательность действий:

- Изменяется запись основного файла.
- Исключается старая ссылка на предыдущее значение вторичного ключа.
- Добавляется новая ссылка на новое значение вторичного ключа.

При этом следует отметить, что два последних шага выполняются для всех вторичных ключей, по которым созданы инвертированные списки. И, разумеется, такой процесс требует гораздо больше временных затрат, чем просто изменение содержимого записи основного файла без поддержки всех инвертированных списков.

Поэтому не следует безусловно утверждать, что введение индексных файлов (в том числе и инвертированных списков) всегда ускоряет обработку информации в базе данных. Отнюдь, если база данных постоянно изменяется, дополняется, модифицируется содержимое записей, то наличие большого количества инвертированных списков или индексных файлов по вторичным ключам может резко замедлить процесс обработки информации.

Можно придерживаться следующей позиции: если база данных достаточно стабильна и ее содержимое практически не меняется, то построение вторичных индексов действительно может ускорить процесс обработки информации.

Модели физической организации данных при бесфайловой организации

Файловая структура и система управления файлами являются прерогативой операционной среды, поэтому принципы обмена данными подчиняются законам операционной системы. По отношению к базам данных эти принципы могут быть далеки от оптимальности. СУБД подчиняется несколько иным принципам и стратегиям управления внешней памятью, чем те, которые поддерживают операционные среды для большинства пользовательских процессов или задач.

Это и послужило причиной того, что СУБД взяли на себя непосредственное управление внешней памятью. При этом пространство внешней памяти предоставляется СУБД полностью для управления, а операционная среда не получает непосредственного доступа к этому пространству.

Физическая организация современных баз данных является наиболее закрытой, она определяется как коммерческая тайна для большинства поставщиков коммерческих СУБД. И здесь не существует никаких стандартов, поэтому в общем

случае каждый поставщик создает свою уникальную структуру и пытается обосновать ее наилучшие качества по сравнению со своими конкурентами. Физическая организация является в настоящий момент наиболее динамичной частью СУБД. Стремительно расширяются возможность устройств внешней памяти, дешевеет оперативная память, увеличивается ее объем и поэтому изменяются сами принципы организации физических структур данных. И можно предположить, что и в дальнейшем эта часть современных СУБД будет постоянно меняться. Поэтому при рассмотрении моделей данных, используемых для физического хранения и обработки, мы коснемся только наиболее общих принципов и тенденций.

При распределении дискового пространства рассматриваются две схемы структуризации: физическая, которая определяет хранимые данные, и логическая, которая определяет некоторые логические структуры, связанные с концептуальной моделью данных (рис. 9.12).

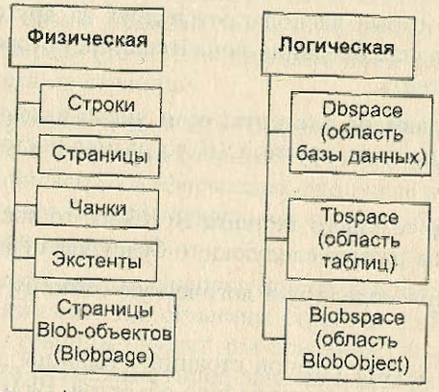


Рис. 9.12. Классификация объектов при статичной организации физической модели данных

Определим некоторые понятия, используемые в указанной классификации.

Чанк (chank) — представляет собой часть диска, физическое пространство на диске, которое ассоциировано одному процессу (on line процессу обработки данных).

Чанком может быть назначено неструктурированное устройство, часть этого устройства, блочно-ориентированное устройство или просто файл UNIX.

Чанк характеризуется маршрутным именем, смещением (от физического начала устройства до начальной точки на устройстве, которая используется как чанк), размером, заданным в Кбайтах или Мбайтах.

При использовании блочных устройств и файлов величина смещения считается равной нулю.

Логические единицы образуются совокупностью экстентов, то есть таблица моделируется совокупностью экстентов.

Экстент — это непрерывная область дисковой памяти.

Для моделирования каждой таблицы используется 2 типа экстентов: первый и последующие.

Первый экстент задается при создании нового объекта типа таблица, его размер задается при создании. EXTENTSIZE — размер первого экстента, NEXT SIZE — размер каждого следующего экстента.

Минимальный размер экстента в каждой системе свой, но в большинстве случаев он равен 4 страницам, максимальный — 2 Гбайтам.

Новый экстент создается после заполнения предыдущего и связывается с ним специальной ссылкой, которая располагается на последней странице экстента. В ряде систем экстенты называются сегментами, но фактически эти понятия эквиваленты.

При динамическом заполнении БД данными применяется специальный механизм адаптивного определения размера экстентов.

Внутри экстента идет учет свободных станиц.

Между экстентами, которые располагаются друг за другом без промежутков, производится своеобразная операция конкатенации, которая просто увеличивает размер первого экстента.

Механизм удвоения размера экстента: если число выделяемых экстентов для процесса растет в пропорции, кратной 16, то размер экстента удваивается каждые 16 экстентов.

Например, если размер текущего экстента 16 Кбайт, то после заполнения 16 экстентов данного размера размер следующего будет увеличен до 32 Кбайт.

Совокупность экстентов моделирует логическую единицу — таблицу-отношение (tblspace).

Экстенты состоят из четырех типов страниц: страницы данных, страницы индексов, битовые страницы и страницы blob-объектов. Blob — это сокращение Binary Larg Object, и соответствует оно неструктурированным данным. В ранних СУБД такие данные относились к типу Memo. В современных СУБД к этому типу относятся неструктурированные большие текстовые данные, картинки, просто наборы машинных кодов. Для СУБД важно знать, что этот объект надо хранить целиком, что размеры этих объектов от записи к записи могут резко отличаться и этот размер в общем случае неограничен.

Основной единицей осуществления операций обмена (ввода-вывода) является страница данных. Все данные хранятся постранично. При табличном хранении данные на одной странице являются однородными, то есть станица может хранить только данные или только индексы.

Все страницы данных имеют одинаковую структуру, представленную на рис. 9.13.

Слот — это 4-байтовое слово, 2 байта соответствуют смещению строки на странице и 2 байта — длина строки. Слоты характеризуют размещение строк данных на странице. На одной странице хранится не более 255 строк. В базе данных каждая строка имеет уникальный идентификатор в рамках всей базы данных, часто называемый RowID — номер строки, он имеет размер 4 байта и состоит из номера страницы и номера строки на странице. Под номер страницы отводится

3 байта, поэтому при такой идентификации возможна адресация к 16 777 215 страницам.

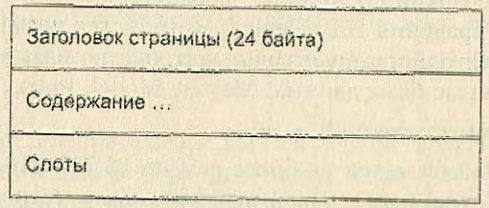


Рис. 9.13. Обобщенная структура страницы данных

При упорядочении строк на страницах не происходит физического перемещения строк, все манипуляции происходят со слотами.

При переполнении страниц создается специальный вид страниц, называемых страницами остатка. Строки, не уместившиеся на основной странице, связываются (линкуются) со своим продолжением на страницах остатка с помощью ссылок-указателей «вперед» (то есть на продолжение), которые содержат номер страницы и номер слота на странице.

Страницы индексов организованы в виде B-деревьев. Страницы blob предназначены для хранения слабоструктурированной информации, содержащей тексты большого объема, графическую информацию, двоичные коды. Эти данные рассматриваются как потоки байтов произвольного размера, в страницах данных делаются ссылки на эти страницы.

Битовые страницы служат для трассировки других типов страниц. В зависимости от трассируемых страниц битовые страницы строятся по 2-битовой или 4-битовой схеме. 4-битовые страницы служат для хранения сведений о столбцах типа Varchar, Byte, Text, для остальных типов данных используются 2-битовые страницы.

Битовая структура трассирует 32 страницы. Каждая битовая структура представлена двумя 4-байтными словами. Каждая i-я позиция описывает одну i-ю страницу. Сочетание разрядов в i-х позициях двух слов обозначает состояние данной страницы: ее тип и занятость.

При обработке данных СУБД организует специальные структуры в оперативной памяти, называемые разделяемой памятью, и специальные структуры во внешней памяти, называемые журналами транзакций. Разделяемая память служит для кэширования данных при работе с внешней памятью с целью сокращения времени доступа, кроме того, разделяемая память служит для эффективной поддержки режимов одновременной параллельной работы пользователей с базой данных.

Журнал транзакций служит для управления корректным выполнением транзакций.

Однако тема параллельной обработки данных выходит за рамки данного раздела, и поэтому архитектура разделяемой памяти будет освещена в разделах, посвященных распределенной обработке данных.

Структура хранения данных для MS SQL 6.5

В версии 6.0 и 6.5 MS SQL Server логическая структура хранения рассматривается в следующей иерархии [1]. Файлы операционной системы представляются как устройства для хранения БД (Device), устройства нумеруются. Сервер может управлять 256 устройствами. Главное устройство называется MASTER: на нем хранятся системные базы данных: Master, Model, Pubs, TempDb.

Устройство Master имеет номер 0 — ноль.

Каждое устройство разбивается не более чем на 16 777 216 виртуальных страниц по 2 Кбайта (Virtual page), максимальный размер устройства 32 Гбайт.

Первые 4 страницы устройства Master заняты под блок конфигурации (Configuration block) — там хранятся все параметры конфигурации сервера. На устройствах размещаются конкретные базы. На каждом устройстве может быть размещено несколько баз, но и одна база может быть размещена на нескольких устройствах.

Каждая страница БД имеет свой уникальный номер.

Физически используются 3 единицы хранения данных:

- страница;
- блок (extent) — 16Кб из 8 следующих друг за другом страниц;
- единица размещения (Allocation Unit) — 512 Кб из 32 последовательных блоков (256 страниц).

При создании новой базы данных пространство для нее отводится единицами размещения. Минимальный объем базы данных для данной версии сервера равен 1 Мбайт, то есть составляет 2 единицы размещения.

Страницы бывают пяти типов:

- страницы размещения (Allocation page);
- страницы данных (Data page);
- индексные страницы (Index page);
- текстовые страницы (Text/image page);
- статистические страницы (Distribution page).

Любая страница имеет заголовок, занимающий 32 байта. Заголовок содержит номер страницы, номера предыдущей и следующей страниц, идентификатор объекта — владельца страницы и сведения о свободном пространстве на странице. Как видно из заголовка, страницы связаны в двунаправленный список.

Первая страница каждой единицы размещения является страницей размещения. Таким образом, все страницы, кратные 256, начиная с 0 являются страницами размещения. Они хранят информацию, необходимую для управления размещением страниц внутри единицы размещения. Страница размещения содержит 32 16-байтовых структуры, по одной на каждый блок. Каждая структура содержит следующую информацию:

- идентификатор объекта—владельца блока;

- номер следующего блока в цепи;
- номер предыдущего блока в цепи;
- битовую карту распределения блока (Allocation bitmap);
- битовую карту перераспределения блока (Deallocation bitmap);
- идентификатор индекса (если таковой есть), размещенного на блоке;
- статус.

Битовая карта распределения блока хранится в единственном байте, каждый бит которого соответствует одной странице блока. Если бит равен 1, то страница в данный момент содержит данные, если 0 — то страница свободна.

Карта перераспределения применяется для отслеживания страниц, которые освобождаются в течение транзакций. Реально страница помечается как пустая только после успешной фиксации (завершения) транзакции. Это делается, чтобы другие транзакции не обращались к странице до подтверждения того факта, что она освобождена.

Все страницы в блоке могут использоваться только одной таблицей или ее индексом. Это означает, что таблица может занимать минимально 1 блок — 16 Кбайт, даже если она содержит всего несколько строк.

Страницы данных используются для хранения собственно данных. Структурно страницу данных можно подразделить на три зоны: заголовок, строки данных и таблицу смещения (см. рис. 9.14).

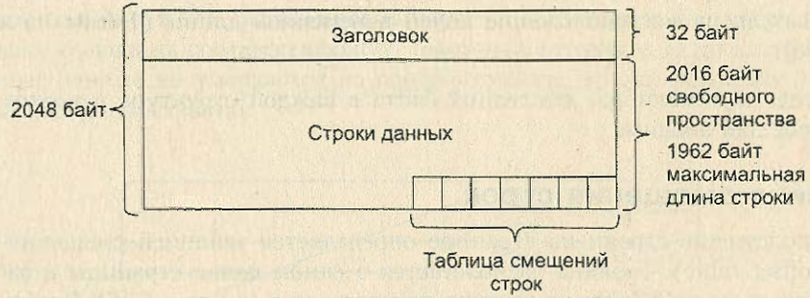


Рис. 9.14. Структура страницы данных для MS SQL Server 6.5

Строка данных должна полностью умещаться на странице, поэтому существуют ограничения на длину строки. Размер страницы 2048 байт, 32 байта занимает заголовок. Кроме того, в таблице смещения отводится по 2 байта на каждую строку на странице.

Страницы данных, относящиеся к одной таблице, объединяются в двунаправленный список и организуют цепочки.

Данные хранятся на страницах в виде строк (кортежей). Каждая строка данных кроме собственно данных хранит дополнительную форматирующую информацию. Длина строки зависит от определения полей таблицы и конкретных данных в ней. Независимо от объявления, каждая строка имеет номер и поле с количеством полей переменной длины (к ним относятся также поля, допускающие

неопределенные значения NULL). Оба эти поля имеют размер по одному байту, следовательно, количество строк на странице не превышает 256, а на количество полей также существует внешнее ограничение 250 полей в одной таблице. Структура строки таблицы приведена на рис. 9.15.

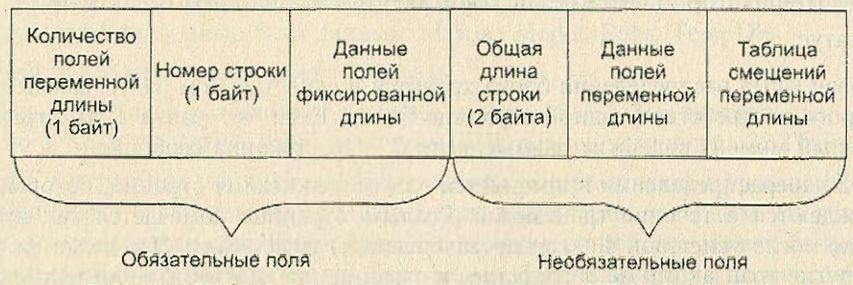


Рис. 9.15. Структура строки данных для MS SQL Server 6.5

Вторая часть — это необязательная область, она существует только тогда, когда имеются в записи поля переменной длины.

Таблица смещений (Column offset table) состоит из:

- таблицы подстройки смещений (Offset table adjust bytes) — по 1 дополнительному байту на каждое поле, смещение которого превышает 256 плюс 1 байт;
- указателя на местоположение таблицы смещений;
- указателя на местоположение полей переменной длины (1 байт на каждое поле).

Указатели занимают два последних байта в каждой структуре и поэтому они доступны для анализа.

Таблица смещения строк

Местоположение строки на странице определяется таблицей смещения строк (Row offset table). Таблица располагается в самом конце страницы и забирает дополнительно по 2 байта на каждую строку данных (см. рис. 9.16). Чтобы найти строку с заданным номером, SQL Server считывает из соответствующей ячейки смещение, которое и является адресом требуемой строки. Ячейка таблицы однозначно связана с определенным номером строки.

Удаленные строки имеют нулевое смещение. Поэтому из примера на рис. 9.16 видно, что строки 1 и 4 удалены.

У этой модели есть недостаток. После удаления строки в таблице смещения все равно остается ссылка на нее, которая занимает 1 байт. Однако при добавлении новой строки SQL Server проверяет таблицу смещений и ищет нулевое смещение, и новой строке присваивается номер удаленной, а в соответствующую ячейку таблицы смещений заносится адрес новой строки.

В SQL Server 6.5 используется понятие кластерного индекса. В таблице, для которой создается кластерный индекс, данные хранятся строго упорядоченно по

полю, для которого создан этот кластерный индекс. Это поле (или набор полей) является первичным ключом таблицы или обладает свойством уникальности. При заполнении таблиц с кластерным индексом вводится параметр, соответствующий проценту заполнения страницы (fill-factor). Если страница заполнена, то данные заносятся на последнюю страницу в цепочке страниц, занятых этой таблицей.

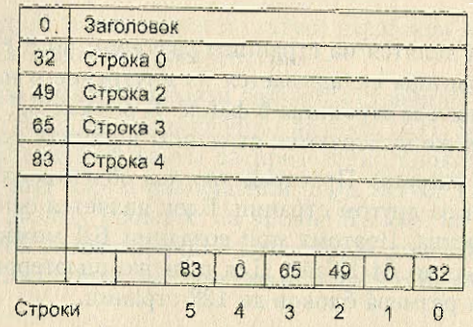


Рис. 9.16. Пример заполнения таблицы смещения строк

Некластерный индекс хранится отдельно от данных.

Текстовые страницы предназначены для хранения данных типа Next или Image.

На одной текстовой странице хранятся только данные одной строки основной таблицы (см. рис. 9.17). В основной таблице в соответствующем месте хранится только ссылка на соответствующую текстовую страницу. Если неструктурированные данные не умещаются на одной странице, то они образуют цепочку взаимосвязанных страниц.

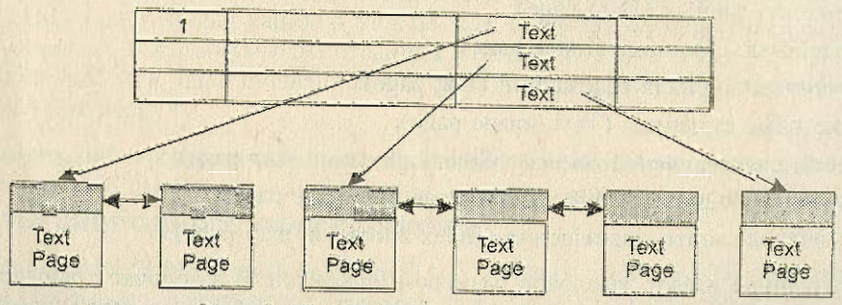


Рис. 9.17. Хранение текстовых данных

Структуры хранения данных в SQL Server 7.0

В новой версии сервера баз данных фирма Microsoft реализовала абсолютно новый механизм хранения.

SQL Server 7.0 организует следующую иерархию хранения:

- База данных — некоторый объем физического пространства, на котором размещаются данные, принадлежащие одной логической базе данных.
- Файл. Каждая база данных содержит не менее двух файлов. Один из них отводится под журнал транзакций. И в отличие от версии 6.5 в новой версии журнал транзакций не может располагаться в одном файле с данными. И еще одно принципиальное отличие — в новой версии каждый файл может принадлежать только одной базе данных, у нас не может быть разделяемых файлов.
- Страница. Файлы делятся на страницы размером по 8 Кбайт каждая. Логический номер страницы складывается из внутреннего номера базы данных, номера файла и номера страницы в файле. В рамках БД файлы нумеруются, начиная с 1, и так же нумеруются страницы в рамках файла.
- Блоки (экстенты, extents). Пространство под объекты отводится блоками по 8 следующих друг за другом страниц. Блок является основной единицей отведения пространства. Поэтому при создании БД можно указывать размер файла с точностью до 64 Кбайт. Для суперкомпьютеров заложена возможность увеличения размера блоков до 128 страниц.

В отличие от версии 6.5 объекты БД не обязательно занимают целый блок. На начальном этапе заполнения объект может занимать внутри блока несколько страниц. Поэтому существуют два типа блоков:

- Однородные (Uniform). Все страницы однородного блока принадлежат одному объекту БД.
- Смешанные (Mixed). Разные страницы в блоке принадлежат разным объектам.

Когда объект создается, то обычно его первые страницы отводятся в смешанном блоке, по мере роста объекта он уже размещается в однородных блоках.

В SQL 7.0 существуют уже 7 типов страниц:

- страница данных (Data page);
- индексные страницы (Index page);
- страницы журнала транзакций (Log page);
- текстовые страницы (Text/image page);
- карты распределения блоков (Global allocation map page);
- карты свободного пространства (Page free space page);
- индексные карты размещения (Index allocation map page).

Все страницы имеют заголовок размером 96 байтов. В заголовке хранится общая информация, используемая ядром СУБД для работы со страницами. На странице в отличие от блока хранится однородная информация. Поэтому среди параметров страницы задаются:

- номер страницы в формате <номер файла, номер страницы>;
- идентификатор объекта, которому принадлежит страница;
- номер индекса, которому принадлежит страница;
- уровень внутри индексного дерева, которому принадлежит страница;

- количество отведенных строк на странице, количество заполненных слотов;
- общий объем свободного пространства на странице;
- указатель на расположение свободного пространства после последней строки на странице;
- минимальная длина строки на странице;
- объем зарезервированного пространства.

После заголовка следует информация о статусе страницы в картах распределения блоков и карте свободного пространства.

Новыми в архитектуре дисковой памяти являются страницы размещения. В этих страницах хранятся сведения о размещении данных. SQL Server 7.0 использует три типа страниц размещения: карты распределения блоков, карты свободного пространства, индексные карты размещения. SQL Server 7.0 хранит информацию размещения на разных уровнях: на уровне блоков, на уровне страниц, на уровне объектов. Такой разносторонний мониторинг помогает СУБД оптимизировать работу в соответствии с требованиями конкретного запроса.

Карты распределения блоков

В данных картах хранится информация о распределении блоков. Карта распределения блоков состоит из стандартного заголовка и одного битового массива в 64 000 битов. Каждый бит характеризует один блок. Поэтому одна страница карты распределения описывает пространство в 64 000 блоков или 4 Гбайт данных.

Карты распределения блоков делятся на два типа:

- Глобальная карта распределения (Global allocation map, GAM) хранит информацию об использовании блоков. Если бит установлен в 0, то блок занят данными, если в 1 — то блок свободен.
- Вторичная глобальная карта распределения (Secondary global allocation map, SGAM) хранит информацию о типе блоков. Если бит установлен в 1, то блок смешанный и минимум одна страница в нем свободна, в остальных случаях (блок свободен, блок смешанный, но свободных страниц нет, блок однородный) бит равен 0.

При отведении пространства сервер использует обе карты распределения.

Карты свободного пространства

Степень заполнения страниц в SQL 7.0 отслеживает специальный механизм — карты свободного пространства (Page free space page, PFS). Каждая PFS-страница хранит информацию о 8000 страниц, по 1 байту на страницу. Каждый байт представляет собой битовую карту, которая сообщает о степени занятости страницы и о том, принадлежит ли она объекту.

Первые страницы файла БД всегда используются под карты распределения. Страница № 1 состоит из двух частей. После стандартного заголовка страницы следует заголовок файла, содержащий его описание, затем размещается блок PFS. Страницы PFS повторяются через каждые 8000 страниц, если размер файла

превосходит один блок. Страница № 2 — это GAM, страница № 3 — это SGAM. Карты распределения блоков повторяются через каждые 512 000 страниц. Кроме того, каждая девятая страница первичного файла — это загрузочная страница БД (database boot page), содержащая описание БД и параметры конфигурации.

Карты размещения

Для организации связи между блоками и расположенными на них объектами используются индексные карты размещения (Index Allocation Map, IAM). Каждая таблица или индекс имеют одну или более страниц IAM. В каждом файле, в котором размещаются таблица или индекс, существует минимум одна карта размещения для этой таблицы или индекса. Страницы IAM размещаются произвольно внутри файла и отводятся по мере необходимости. IAM объединены друг с другом в цепочку двунаправленными ссылками. Указатель на первую карту размещения содержится в поле FirstIAM системной таблицы Sysindex.

Каждая IAM описывает некоторый диапазон блоков и представляет собой битовую карту: если бит установлен в 1, то в данном блоке есть страницы, принадлежащие данному объекту, если в 0 — то нет.

Все страницы размещения не связаны напрямую с некоторым объектом БД, они соответствуют некоторой системной информации, поэтому параметр «идентификатор объекта» для всех этих страниц одинаков и равен 99.

Страницы данных

На самом общем уровне здесь, так же как и в предыдущей версии, страница данных делится на 3 зоны: заголовок, область данных и таблицу смещений, но размер страницы увеличен, размер заголовка также и некоторые отличия существуют и в структуре остальных зон.

Прежде всего стоит отметить, что в отличие от SQL Server 6.5 в новой версии страницы данных не связаны друг с другом в цепочки. За связь между страницами и объектами отвечает новая специальная структура — карты размещения.

Кроме того, ранее данные на странице хранились непрерывно. При удалении строки данные внутри страницы перемещались так, чтобы не оставалось пустот. Однако такой подход затруднял строчные блокировки. И в новой версии данные на странице не обязательно хранятся непрерывно. Здесь допустимы пропуски. При удалении строки пустое пространство помечается и потом его может занять новая строка, но перемещения строк не происходит.

В SQL Server 7.0 теперь поддерживается классический термин слот (Slot), и это место размещения строки на странице. Если таблица не имеет кластеризованного индекса, то номер слота является идентификатором строки и не меняется, пока не будет удалена соответствующая строка. Если же таблица имеет кластеризованный индекс, то слоты располагаются в порядке, задаваемом индексом.

Строки данных

Строки данных претерпели существенное изменение. Отметим наиболее важные моменты.

- Номера строки больше нет — строка идентифицируется номером слота, который ее определяет, либо значением кластерного ключа.
- В версии 6.5 поля, допускающие NULL, хранятся точно так же, как поля переменной длины. В версии 7.0 поля фиксированной длины всегда занимают свою полную длину, значение NULL задается специальным флагом. Это облегчает замену неопределенного значения на некоторое конкретное без перемещения строк на странице.
- Фиксированные поля вместе с описателями хранятся до полей переменной длины, так же как и в 6.5.
- В каждой строке хранится общая длина строки и текущие длины полей переменной длины. Отсутствуют таблицы смещений и подстройки смещений. Данные считываются последовательно с начального адреса.
- Максимальное количество полей в строке 1024, в версии 6.5 только 256.

Текстовые страницы

В версии 7.0 изменены принципы хранения текстовых полей. Строки данных по-прежнему содержат 16-байтные указатели на текстовые данные. Однако хранение самих текстовых данных производится иначе.

Текстовая страница теперь может содержать несколько текстовых полей. Собственно данные хранятся в виде сбалансированного дерева (B-tree). Строка данных содержит указатель на корневую структуру (Root structure) размером 84 байта.

Данные длиной менее 64 байт хранятся в корневой структуре. Для данных до 32 Кбайт корневая структура (Root structure) может адресовать 4 блока данных (это не блоки страниц) до 8 Кбайт каждый. Блоки наращиваются до 8 Кбайт (реально на одной текстовой странице может быть размещено 8080 байт). Например, если первая порция данных составляет 4 Кбайта, то отводится один блок. Если в дальнейшем данные увеличиваются до 6 Кбайт, то первый блок увеличивается до 6 Кбайт, а второй блок имеет размер всего 2 Кбайта.

Если же длина текстового поля более 32 Кбайт, то строятся промежуточные узлы.

В версии 7.0 текстовая страница может содержать данные нескольких текстовых полей (рис. 9.18).

Страницы журнала транзакций

В отличие от версии 6.5 в новой версии под журнал транзакций отводится всегда отдельный файл. В предыдущей версии журнал транзакций являлся системной таблицей и хранился в системном сегменте LOG. Несмотря на настоятельные рекомендации располагать журнал транзакций и файлы базы данных на разных физических устройствах, СУБД автоматически не контролировала этот факт. В новой версии журнал транзакций всегда хранится в отдельном файле.

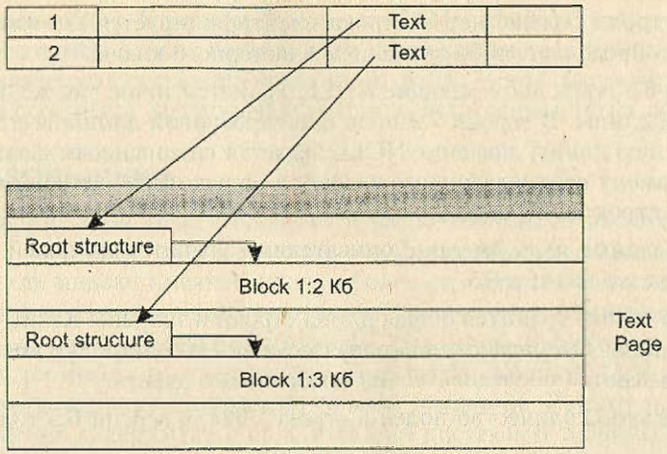


Рис. 9.18. Пример хранения текстовых данных на одной странице

Архитектура разделяемой памяти

По причинам объективно существующей разницы в скорости работы процессоров, оперативной памяти и устройств внешней памяти (эта разница в скорости существовала, существует и будет существовать всегда) буферизация страниц базы данных в оперативной памяти — единственный реальный способ достижения удовлетворительной эффективности СУБД.

Операционные системы создают специальные системные буферы, которые служат для кэширования пользовательских процессов. Однако стратегия буферизации, применяемая в операционных средах, не соответствует целям и задачам СУБД, поэтому для оптимизации обработки данных одной из главных задач СУБД является создание эффективной системы управления процессом буферизации.

Разделяемая память, управляемая СУБД, состоит из нескольких типов буферов:

- Буферы страниц данных, которые содержат копии страниц данных, с которыми работает СУБД.
- Буферы страниц журнала транзакций, которые отражают процесс выполнения транзакции — последовательности операций над БД, переводящей БД из одного непротиворечивого состояния в другое непротиворечивое состояние.
- Системные буферы, которые содержат общую информацию о БД, о пользователях, о физической структуре БД, о базе метаданных.

Информация в буферах взаимосвязана, и требуется эффективная система поддержки единой работы всех частей разделяемой памяти.

Если бы запись об изменении базы данных реально немедленно записывалась во внешнюю память, это привело бы к существенному замедлению работы системы.

Поэтому записи в журнал тоже буферизуются: при нормальной работе очередная страница выталкивается во внешнюю память журнала только при полном заполнении записями.

Но реальная ситуация является более сложной. Имеются два вида буферов — буфер журнала и буфер страниц оперативной памяти, которые содержат связанную информацию. И те и другие буферы могут выталкиваться во внешнюю память. Проблема состоит в выработке некоторой общей политики выталкивания, которая обеспечивала бы возможности восстановления состояния базы данных после сбоев.

Буфера не выделяются для каждого пользовательского процесса, они выделяются для всех процессов сервера БД. Это позволяет увеличить степень параллелизма при исполнении клиентских процессов.

Разделяемая память наиболее эффективно используется вспомогательными процессами сервера, иногда называемыми демонами, которые используются для синхронизации взаимодействующих процессов на сервере.

На этом мы закончили рассмотрение физических моделей, применяемых в базах данных. Физические модели большей частью скрыты от пользователей. Однако в SQL существует команда создания индексных файлов. При этом по умолчанию стандартно создаются индексные файлы для первичных ключей, для вторичных ключей индексные файлы создаются дополнительной командой CREATE INDEX, которая имеет следующий формат:

```
CREATE [UNIQUE] INDEX <имя_индекса> ON <имя_таблицы>
( <имя_столбца>[<признак упорядочения>] |
   [,<имя_столбца>[<признак упорядочения>]…])
<имя_индекса > – уникальный идентификатор в системе.
<Признак упорядочения>::={ASC | DESC}
```

Здесь ASC — признак упорядочения по возрастанию, DESC — признак упорядочения по убыванию значений соответствующего столбца в индексе.

Индекс может быть удален командой DROP, которая имеет следующий формат:

```
DROP INDEX <имя_индекса>
```

ГЛАВА 10 Распределенная обработка данных

При размещении БД на персональном компьютере, который не находится в сети, БД всегда используется в монопольном режиме. Даже если БД используют несколько пользователей, они могут работать с ней только последовательно, и поэтому вопросов о поддержании корректной модификации БД в этом случае здесь не стоит, они решаются организационными мерами — то есть определением требуемой последовательности работы конкретных пользователей с соответствующей БД. Однако даже в некоторых настольных БД требуется учитывать последовательность изменения данных при обработке, чтобы получить корректный результат: так, например, при запуске программы балансного бухгалтерского отчета все бухгалтерские проводки — финансовые операции должны быть решены заранее до запуска конечного приложения.

Однако работа на изолированном компьютере с небольшой базой данных в настоящий момент становится уже нехарактерной для большинства приложений. БД отражает информационную модель реальной предметной области, она растет по объему и резко увеличивается количество задач, решаемых с ее использованием, и в соответствии с этим увеличивается количество приложений, работающих с единой базой данных. Компьютеры объединяются в локальные сети, и необходимость распределения приложений, работающих с единой базой данных по сети, является несомненной.

Действительно, даже когда вы строите БД для небольшой торговой фирмы, у вас появляется ряд специфических пользователей БД, которые имеют свои бизнес-функции и территориально могут находиться в разных помещениях, но все они должны работать с единой информационной моделью организации, то есть с единой базой данных.

Параллельный доступ к одной БД нескольких пользователей, в том случае если БД расположена на одной машине, соответствует режиму распределенного доступа к централизованной БД. (Такие системы называются *системами распределенной обработки данных*.)

Если же БД распределена по нескольким компьютерам, расположенным в сети, и к ней возможен параллельный доступ нескольких пользователей, то мы имеем дело с параллельным доступом к распределенной БД. Подобные системы называются *системами распределенных баз данных*. В общем случае режимы использования БД можно представить в следующем виде (см. рис. 10.1).

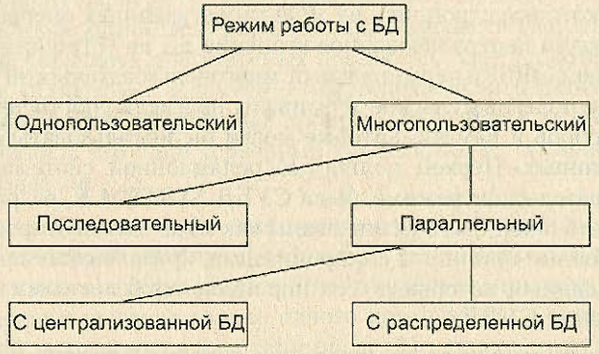


Рис. 10.1. Режимы работы с базой данных

Определим терминологию, которая нам потребуется для дальнейшей работы. Часть терминов нам уже известна, но повторим здесь их дополнительно.

Терминология

Пользователь БД — программа или человек, обращающийся к БД на ЯМД.

Запрос — процесс обращения пользователя к БД с целью ввода, получения или изменения информации в БД.

Транзакция — последовательность операций модификации данных в БД, переводящая БД из одного непротиворечивого состояния в другое непротиворечивое состояние.

Логическая структура БД — определение БД на физически независимом уровне, ближе всего соответствует концептуальной модели БД.

Топология БД = Структура распределенной БД — схема распределения физической БД по сети.

Локальная автономность — означает, что информация локальной БД и связанные с ней определения данных принадлежат локальному владельцу и им управляются.

Удаленный запрос — запрос, который выполняется с использованием модемной связи.

Возможность реализации удаленной транзакции — обработка одной транзакции, состоящей из множества SQL-запросов на одном удаленном узле.

Поддержка распределенной транзакции — допускает обработку транзакции, состоящей из нескольких запросов SQL, которые выполняются на нескольких узлах сети (удаленных или локальных), но каждый запрос в этом случае обраба-

тывается только на одном узле, то есть запросы не являются распределенными. При обработке одной распределенной транзакции разные локальные запросы могут обрабатываться в разных узлах сети.

Распределенный запрос — запрос, при обработке которого используются данные из БД, расположенные в разных узлах сети.

Системы распределенной обработки данных в основном связаны с первым поколением БД, которые строились на мультипрограммных операционных системах и использовали централизованное хранение БД на устройствах внешней памяти центральной ЭВМ и терминальный многопользовательский режим доступа к ней. При этом пользовательские терминалы не имели собственных ресурсов — то есть процессоров и памяти, которые могли бы использоваться для хранения и обработки данных. Первой полностью реляционной системой, работающей в многопользовательском режиме, была СУБД SYSTEM R, разработанная фирмой IBM, именно в ней были реализованы как язык манипулирования данными SQL, так и основные принципы синхронизации, применяемые при распределенной обработке данных, которые до сих пор являются базисными практически во всех коммерческих СУБД.

Общая тенденция движения от отдельных mainframe-систем к открытым распределенным системам, объединяющим компьютеры среднего класса, получила название DownSizing. Этот процесс оказал огромное влияние на развитие архитектур СУБД и поставил перед их разработчиками ряд сложных задач. Главная проблема состояла в технологической сложности перехода от централизованного управления данными на одном компьютере и СУБД, использовавшей собственные модели, форматы представления данных и языки доступа к данным и т. д., к распределенной обработке данных в неоднородной вычислительной среде, состоящей из соединенных в глобальную сеть компьютеров различных моделей и производителей.

В то же время происходил встречный процесс — UpSizing. Бурное развитие персональных компьютеров, появление локальных сетей также оказали серьезное влияние на эволюцию СУБД. Высокие темпы роста производительности и функциональных возможностей PC привлекли внимание разработчиков профессиональных СУБД, что привело к их активному распространению на платформе настольных систем.

Сегодня возобладала тенденция создания информационных систем на такой платформе, которая точно соответствовала бы ее масштабам и задачам. Она получила название RightSizing (помещение ровно в тот размер, который необходим).

Однако и в настоящее время большие ЭВМ сохраняются и сосуществуют с современными открытыми системами. Причина этого проста — в свое время в аппаратное и программное обеспечение больших ЭВМ были вложены огромные средства: в результате многие продолжают их использовать, несмотря на морально устаревшую архитектуру. В то же время перенос данных и программ с больших ЭВМ на компьютеры нового поколения сам по себе представляет сложную техническую проблему и требует значительных затрат.

Модели «клиент—сервер» в технологии баз данных

Вычислительная модель «клиент—сервер» исходно связана с парадигмой открытых систем, которая появилась в 90-х годах и быстро эволюционировала. Сам термин «клиент-сервер» исходно применялся к архитектуре программного обеспечения, которое описывало распределение процесса выполнения по принципу взаимодействия двух программных процессов, один из которых в этой модели назывался «клиентом», а другой — «сервером». Клиентский процесс запрашивал некоторые услуги, а серверный процесс обеспечивал их выполнение. При этом предполагалось, что один серверный процесс может обслужить множество клиентских процессов.

Ранее приложение (пользовательская программа) не разделялась на части, оно выполнялось некоторым монолитным блоком. Но возникла идея более рационального использования ресурсов сети. Действительно, при монолитном исполнении используются ресурсы только одного компьютера, а остальные компьютеры в сети рассматриваются как терминалы. Но теперь, в отличие от эпохи main-фреймов, все компьютеры в сети обладают собственными ресурсами, и разумно так распределить нагрузку на них, чтобы максимальным образом использовать их ресурсы.

И как в промышленности, здесь возникает древняя как мир идея распределения обязанностей, разделения труда. Конвейеры Форда сделали в свое время прорыв в автомобильной промышленности, показав наивысшую производительность труда именно из-за того, что весь процесс сборки был разбит на мелкие и максимально простые операции и каждый рабочий специализировался на выполнении только одной операции, но эту операцию он выполнял максимально быстро и качественно.

Конечно, в вычислительной технике нельзя было напрямую использовать технологию автомобильного или любого другого механического производства, но идею использовать было можно. Однако для воплощения идеи необходимо было разработать модель разбиения единого монолитного приложения на отдельные части и определить принципы взаимосвязи между этими частями.

Основной принцип технологии «клиент—сервер» применительно к технологии баз данных заключается в разделении функций стандартного интерактивного приложения на 5 групп, имеющих различную природу:

- функции ввода и отображения данных (Presentation Logic);
- прикладные функции, определяющие основные алгоритмы решения задач приложения (Business Logic);
- функции обработки данных внутри приложения (Database Logic);
- функции управления информационными ресурсами (Database Manager System);
- служебные функции, играющие роль связок между функциями первых четырех групп.

Структура типового приложения, работающего с базой данных приведена на рис. 10.2.

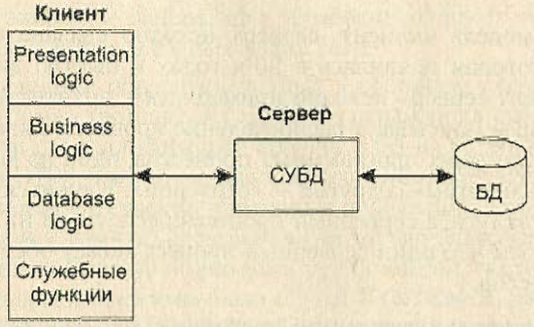


Рис. 10.2. Структура типового интерактивного приложения, работающего с базой данных

Презентационная логика (Presentation Logic) как часть приложения определяется тем, что пользователь видит на своем экране, когда работает приложение. Сюда относятся все интерфейсные экранные формы, которые пользователь видит или заполняет в ходе работы приложения, к этой же части относится все то, что выводится пользователю на экран как результаты решения некоторых промежуточных задач либо как справочная информация. Поэтому основными задачами презентационной логики являются:

- формирование экранных изображений;
- чтение и запись в экранные формы информации;
- управление экраном;
- обработка движений мыши и нажатие клавиш клавиатуры.

Некоторые возможности для организации презентационной логики приложений предоставляет знако-ориентированный пользовательский интерфейс, задаваемый моделями CICS (Customer Control Information System) и IMS/DC фирмы IBM и моделью TSO (Time Sharing Option) для централизованной main-фреймовой архитектуры. Модель GUI — графического пользовательского интерфейса, поддерживается в операционных средах Microsoft's Windows, Windows NT, в OS/2 Presentation Manager, X-Windows и OSF/Motif.

Бизнес-логика, или логика собственно приложений (Business processing Logic), — это часть кода приложения, которая определяет собственно алгоритмы решения конкретных задач приложения. Обычно этот код пишется с использованием различных языков программирования, таких как C, C++, Cobol, SmallTalk, Visual-Basic.

Логика обработки данных (Data manipulation Logic) — это часть кода приложения, которая связана с обработкой данных внутри приложения. Данными управляет собственно СУБД (DBMS). Для обеспечения доступа к данным используются язык запросов и средства манипулирования данными стандартного языка SQL.

Обычно операторы языка SQL встраиваются в языки 3-го или 4-го поколения (3GL, 4GL), которые используются для написания кода приложения.

Процессор управления данными (Database Manager System Processing) — это собственно СУБД, которая обеспечивает хранение и управление базами данных. В идеале функции СУБД должны быть скрыты от бизнес-логики приложения, однако для рассмотрения архитектуры приложения нам надо их выделить в отдельную часть приложения.

В централизованной архитектуре (Host-based processing) эти части приложения располагаются в единой среде и комбинируются внутри одной исполняемой программы.

В децентрализованной архитектуре эти задачи могут быть по-разному распределены между серверным и клиентским процессами. В зависимости от характера распределения можно выделить следующие модели распределений (см. рис. 10.3):

- распределенная презентация (Distribution presentation, DP);
- удаленная презентация (Remote Presentation, RP);
- распределенная бизнес-логика (Remote business logic, RBL);
- распределенное управление данными (Distributed data management, DDM);
- удаленное управление данными (Remote data management, RDA).

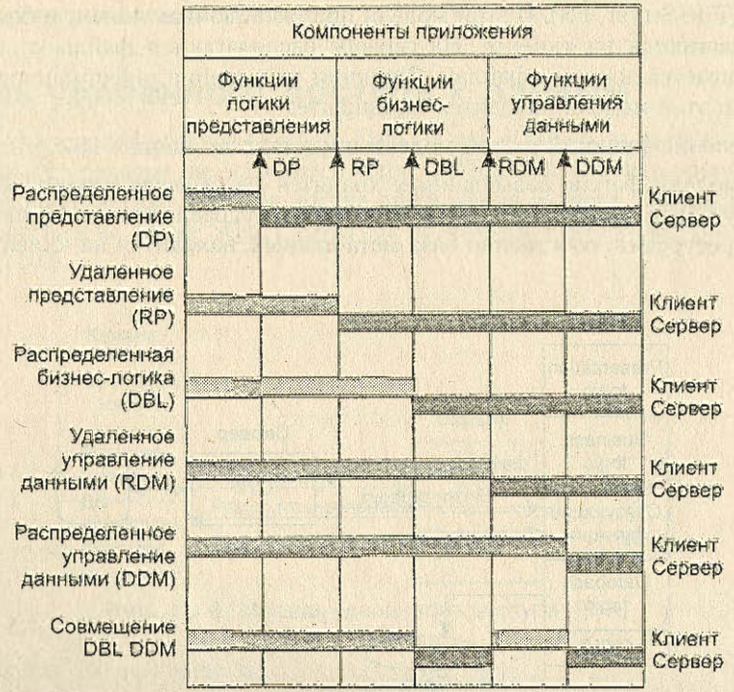


Рис. 10.3. Распределение функций приложения в моделях «клиент—сервер»

Эта условная классификация показывет, как могут быть распределены отдельные задачи между серверным и клиенскими процессами. В этой классификации отсутствует реализация удаленной бизнес-логики. Действительно, считается, что она не может быть удалена сама по себе полностью. Считается, что она может быть распределена между разными процессами, которые в общем-то могут выполняться на разных платформах, но должны корректно кооперироваться (взаимодействовать) друг с другом.

Двухуровневые модели

Двухуровневая модель фактически является результатом распределения пяти указанных функций между двумя процессами, которые выполняются на двух платформах: на клиенте и на сервере. В чистом виде почти никакая модель не существует, однако рассмотрим наиболее характерные особенности каждой двухуровневой модели.

Модель удаленного управления данными. Модель файлового сервера

Модель удаленного управления данными также называется моделью файлового сервера (File Server, FS). В этой модели презентационная логика и бизнес-логика располагаются на клиенте. На сервере располагаются файлы с данными и поддерживается доступ к файлам. Функции управления информационными ресурсами в этой модели находятся на клиенте.

Распределение функций в этой модели представлено на рис. 10.4.

В этой модели файлы базы данных хранятся на сервере, клиент обращается к серверу с файловыми командами, а механизм управления всеми информационными ресурсами, собственно база мета-данных, находится на клиенте.

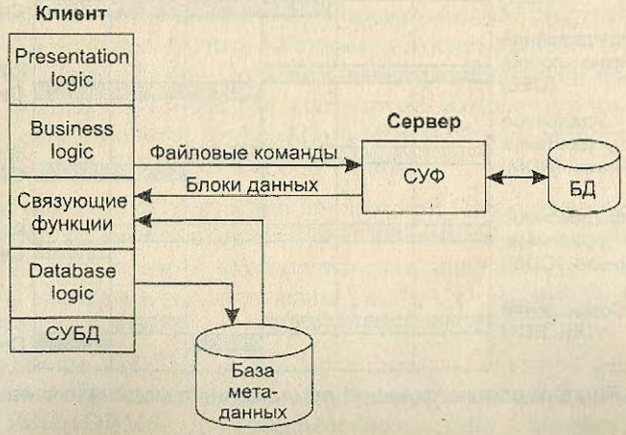


Рис. 10.4. Модель файлового сервера

Достоинства этой модели в том, что мы уже имеем разделение монопольного приложения на два взаимодействующих процесса. При этом сервер (серверный процесс) может обслуживать множество клиентов, которые обращаются к нему с запросами. Собственно СУБД должна находиться в этой модели на клиенте.

Каков алгоритм выполнения запроса клиента?

Запрос клиента формулируется в командах ЯМД. СУБД переводит этот запрос в последовательность файловых команд. Каждая файловая команда вызывает перекачку блока информации на клиента, далее на клиенте СУБД анализирует полученную информацию, и если в полученном блоке не содержится ответ на запрос, то принимается решение о перекачке следующего блока информации и т. д.

Перекачка информации с сервера на клиент производится до тех пор, пока не будет получен ответ на запрос клиента.

Недостатки:

- высокий сетевой трафик, который связан с передачей по сети множества блоков и файлов, необходимых приложению;
- узкий спектр операций манипулирования с данными, который определяется только файловыми командами;
- отсутствие адекватных средств безопасности доступа к данным (защита только на уровне файловой системы).

Модель удаленного доступа к данным

В модели удаленного доступа (Remote Data Access, RDA) база данных хранится на сервере. На сервере же находится ядро СУБД. На клиенте располагается презентационная логика и бизнес-логика приложения. Клиент обращается к серверу с запросами на языке SQL. Структура модели удаленного доступа приведена на рис. 10.5.

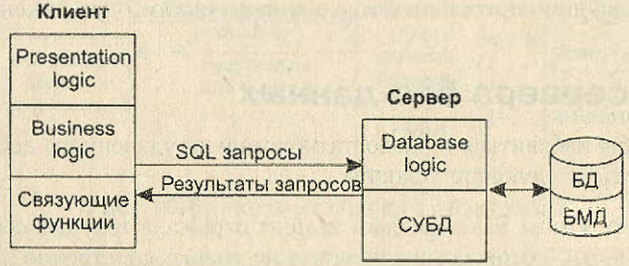


Рис. 10.5. Модель удаленного доступа (RDA)

Преимущества данной модели:

- перенос компонента представления и прикладного компонента на клиентский компьютер существенно разгрузил сервер БД, сводя к минимуму общее число процессов в операционной системе;

- сервер БД освобождается от несвойственных ему функций; процессор или процессоры сервера целиком загружаются операциями обработки данных, запросов и транзакций. (Это становится возможным, если отказаться от терминалов, не располагающих ресурсами, и заменить их компьютерами, выполняющими роль клиентских станций, которые обладают собственными локальными вычислительными ресурсами);
- резко уменьшается загрузка сети, так как по ней от клиентов к серверу передаются не запросы на ввод-вывод в файловой терминологии, а запросы на SQL, и их объем существенно меньше. В ответ на запросы клиент получает только данные, релевантные запросу, а не блоки файлов, как в FS-модели.

Основное достоинство RDA-модели — унификация интерфейса «клиент-сервер», стандартом при общении приложения-клиента и сервера становится язык SQL.

Недостатки:

- все-таки запросы на языке SQL при интенсивной работе клиентских приложений могут существенно загрузить сеть;
- так как в этой модели на клиенте располагается и презентационная логика, и бизнес-логика приложения, то при повторении аналогичных функций в разных приложениях код соответствующей бизнес-логики должен быть повторен для каждого клиентского приложения. Это вызывает излишнее дублирование кода приложений;
- сервер в этой модели играет пассивную роль, поэтому функции управления информационными ресурсами должны выполняться на клиенте. Действительно, например, если нам необходимо выполнять контроль страховых запасов товаров на складе, то каждое приложение, которое связано с изменением состояния склада, после выполнения операций модификации данных, имитирующих продажу или удаление товара со склада, должно выполнять проверку на объем остатка, и в случае, если он меньше страхового запаса, формировать соответствующую заявку на поставку требуемого товара. Это усложняет клиентское приложение, с одной стороны, а с другой — может вызвать необоснованный заказ дополнительных товаров несколькими приложениями.

Модель сервера баз данных

Для того чтобы избавиться от недостатков модели удаленного доступа, должны быть соблюдены следующие условия:

1. Необходимо, чтобы БД в каждый момент отражала текущее состояние предметной области, которое определяется не только собственно данными, но и связями между объектами данных. То есть данные, которые хранятся в БД, в каждый момент времени должны быть непротиворечивыми.
2. БД должна отражать некоторые правила предметной области, законы, по которым она функционирует (business rules). Например, завод может нормально работать только в том случае, если на складе имеется некоторый достаточный запас (страховой запас) деталей определенной номенклатуры, деталь может быть запущена в производство только в том случае, если на складе имеется в наличии достаточно материала для ее изготовления, и т. д.
3. Необходим постоянный контроль за состоянием БД, отслеживание всех изменений и адекватная реакция на них: например, при достижении некоторым измеряемым параметром критического значения должно произойти отключение определенной аппаратуры, при уменьшении товарного запаса ниже допустимой нормы должна быть сформирована заявка конкретному поставщику на поставку соответствующего товара.
4. Необходимо, чтобы возникновение некоторой ситуации в БД четко и оперативно влияло на ход выполнения прикладной задачи.
5. Одной из важнейших проблем СУБД является контроль типов данных. В настоящий момент СУБД контролирует синтаксически только стандартно-допустимые типы данных, то есть такие, которые определены в DDL (data definition language) — языке описания данных, который является частью SQL. Однако в реальных предметных областях у нас действуют данные, которые несут в себе еще и семантическую составляющую, например, это координаты объектов или единицы различных метрик, например рабочая неделя в отличие от реальной имеет сразу после пятницы понедельник.

Данную модель поддерживают большинство современных СУБД: Informix, Ingres, Sybase, Oracle, MS SQL Server. Основу данной модели составляет механизм хранимых процедур как средство программирования SQL-сервера, механизм триггеров как механизм отслеживания текущего состояния информационного хранилища и механизм ограничений на пользовательские типы данных, который иногда называется механизмом поддержки доменной структуры. Модель сервера баз данных представлена на рис. 10.6.

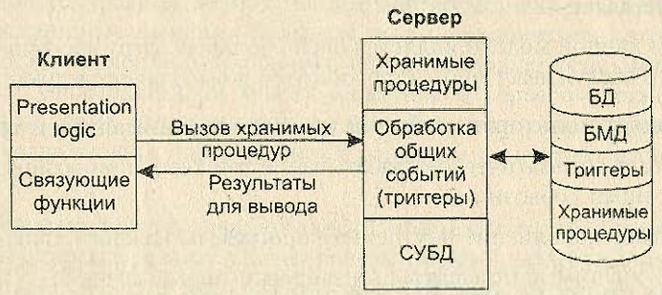


Рис. 10.6. Модель активного сервера БД

В этой модели бизнес-логика разделена между клиентом и сервером. На сервере бизнес-логика реализована в виде хранимых процедур — специальных программных модулей, которые хранятся в БД и управляются непосредственно СУБД. Клиентское приложение обращается к серверу с командой запуска хранимой процедуры, а сервер выполняет эту процедуру и регистрирует все изменения в БД, которые в ней предусмотрены. Сервер возвращает клиенту данные, релевантные его запросу, которые требуются клиенту либо для вывода на

экран, либо для выполнения части бизнес-логики, которая расположена на клиенте. Трафик обмена информацией между клиентом и сервером резко уменьшается.

Централизованный контроль в модели сервера баз данных выполняется с использованием механизма триггеров. Триггеры также являются частью БД.

Термин «триггер» взят из электроники и семантически очень точно характеризует механизм отслеживания специальных событий, которые связаны с состоянием БД. Триггер в БД является как бы некоторым тумблером, который срабатывает при возникновении определенного события в БД. Ядро СУБД проводит мониторинг всех событий, которые вызывают созданные и описанные триггеры в БД, и при возникновении соответствующего события сервер запускает соответствующий триггер. Каждый триггер представляет собой также некоторую программу, которая выполняется над базой данных. Триггеры могут вызывать хранимые процедуры.

Механизм использования триггеров предполагает, что при срабатывании одного триггера могут возникнуть события, которые вызовут срабатывание других триггеров. Этот мощный инструмент требует тонкого и согласованного применения, чтобы не получился бесконечный цикл срабатывания триггеров.

В данной модели сервер является активным, потому что не только клиент, но и сам сервер, используя механизм триггеров, может быть инициатором обработки данных в БД.

И хранимые процедуры, и триггеры хранятся в словаре БД, они могут быть использованы несколькими клиентами, что существенно уменьшает дублирование алгоритмов обработки данных в разных клиентских приложениях.

Для написания хранимых процедур и триггеров используется расширение стандартного языка SQL, так называемый встроенный SQL. Встроенный SQL мы рассмотрим в главе 12.

Недостатком данной модели является очень большая загрузка сервера. Действительно, сервер обслуживает множество клиентов и выполняет следующие функции:

- осуществляет мониторинг событий, связанных с описанными триггерами;
- обеспечивает автоматическое срабатывание триггеров при возникновении связанных с ними событий;
- обеспечивает исполнение внутренней программы каждого триггера;
- запускает хранимые процедуры по запросам пользователей;
- запускает хранимые процедуры из триггеров;
- возвращает требуемые данные клиенту;
- обеспечивает все функции СУБД: доступ к данным, контроль и поддержку целостности данных в БД, контроль доступа, обеспечение корректной параллельной работы всех пользователей с единой БД.

Если мы переложили на сервер большую часть бизнес-логики приложений, то требования к клиентам в этой модели резко уменьшаются. Иногда такую модель называют моделью с «тонким клиентом», в отличие от предыдущих моделей, где на клиента возлагались гораздо более серьезные задачи. Эти модели называются моделями с «толстым клиентом».

Для разгрузки сервера была предложена трехуровневая модель.

Модель сервера приложений

Эта модель является расширением двухуровневой модели и в ней вводится дополнительный промежуточный уровень между клиентом и сервером. Архитектура трехуровневой модели приведена на рис. 10.7. Этот промежуточный уровень содержит один или несколько серверов приложений.

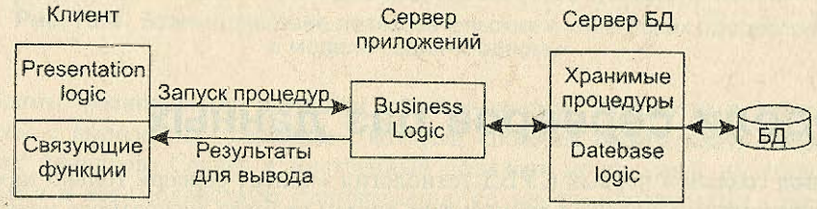


Рис. 10.7. Модель сервера приложений

В этой модели компоненты приложения делятся между тремя исполнителями:

- *Клиент* обеспечивает логику представления, включая графический пользовательский интерфейс, локальные редакторы; клиент может запускать локальный код приложения клиента, который может содержать обращения к локальной БД, расположенной на компьютере-клиенте. Клиент исполняет коммуникационные функции front-end части приложения, которые обеспечивают доступ клиенту в локальную или глобальную сеть. Дополнительно реализация взаимодействия между клиентом и сервером может включать в себя управление распределенными транзакциями, что соответствует тем случаям, когда клиент также является клиентом менеджера распределенных транзакций.
- *Серверы приложений* составляют новый промежуточный уровень архитектуры. Они спроектированы как исполнения общих незагружаемых функций для клиентов. Серверы приложений поддерживают функции клиентов как частей взаимодействующих рабочих групп, поддерживают сетевую доменную операционную среду, хранят и исполняют наиболее общие правила бизнес-логики, поддерживают каталоги с данными, обеспечивают обмен сообщениями и поддержку запросов, особенно в распределенных транзакциях.
- *Серверы баз данных* в этой модели занимаются исключительно функциями СУБД: обеспечивают функции создания и ведения БД, поддерживают целостность реляционной БД, обеспечивают функции хранилищ данных (warehouse services). Кроме того, на них возлагаются функции создания резервных копий БД и восстановления БД после сбоев, управления выполнением транз-

акций и поддержки устаревших (унаследованных) приложений (legacy application).

Отметим, что эта модель обладает большей гибкостью, чем двухуровневые модели. Наиболее заметны преимущества модели сервера приложений в тех случаях, когда клиенты выполняют сложные аналитические расчеты над базой данных, которые относятся к области OLAP-приложений. (On-line analytical processing.) В этой модели большая часть бизнес-логики клиента изолирована от возможностей встроенного SQL, реализованного в конкретной СУБД, и может быть выполнена на стандартных языках программирования, таких как C, C++, SmallTalk, Cobol. Это повышает переносимость системы, ее масштабируемость.

Функции промежуточных серверов могут быть в этой модели распределены в рамках глобальных транзакций путем поддержки XA-протокола (X/Open transaction interface protocol), который поддерживается большинством поставщиков СУБД.

Модели серверов баз данных

В период создания первых СУБД технология «клиент-сервер» только зарождалась. Поэтому изначально в архитектуре систем не было адекватного механизма организации взаимодействия процессов типа «клиент» и процессов типа «сервер». В современных же СУБД он является фактически основополагающим и от эффективности его реализации зависит эффективность работы системы в целом.

Рассмотрим эволюцию типов организации подобных механизмов. В основном этот механизм определяется структурой реализации серверных процессов, и часто он называется архитектурой сервера баз данных.

Первоначально, как мы уже отмечали, существовала модель, когда управление данными (функция сервера) и взаимодействие с пользователем были совмещены в одной программе. Это можно назвать нулевым этапом развития серверов БД.

Затем функции управления данными были выделены в самостоятельную группу — сервер, однако модель взаимодействия пользователя с сервером соответствовала парадигме «один-к-одному» (рис. 10.8), то есть сервер обслуживал запросы только одного пользователя (клиента), и для обслуживания нескольких клиентов нужно было запустить эквивалентное число серверов.

Выделение сервера в отдельную программу было революционным шагом, который позволил, в частности, поместить сервер на одну машину, а программный интерфейс с пользователем — на другую, осуществляя взаимодействие между ними по сети. Однако необходимость запуска большого числа серверов для обслуживания множества пользователей сильно ограничивала возможности такой системы.

Для обслуживания большого числа клиентов на сервере должно быть запущено большое количество одновременно работающих серверных процессов, а это резко повышало требования к ресурсам ЭВМ, на которой запускались все серверные процессы. Кроме того, каждый серверный процесс в этой модели запускался как независимый, поэтому если один клиент сформировал запрос, который был только что выполнен другим серверным процессом для другого клиента, то запрос тем не менее выполнялся повторно. В такой модели весьма сложно обеспечить взаимодействие серверных процессов. Эта модель самая простая, и исторически она появилась первой.

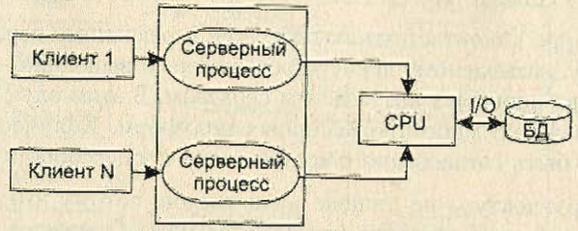


Рис.10.8. Взаимодействие пользовательских и клиентских процессов в модели «один-к-одному»

Проблемы, возникающие в модели «один-к-одному», решаются в архитектуре «систем с выделенным сервером», который способен обрабатывать запросы от многих клиентов. Сервер единственный обладает монополией на управление данными и взаимодействует одновременно со многими клиентами (рис. 10.9). Логически каждый клиент связан с сервером отдельной нитью («thread»), или потоком, по которому пересылаются запросы. Такая архитектура получила название многопотоковой односерверной («multi-threaded»).

Она позволяет значительно уменьшить нагрузку на операционную систему, возникающую при работе большого числа пользователей («trashing»).

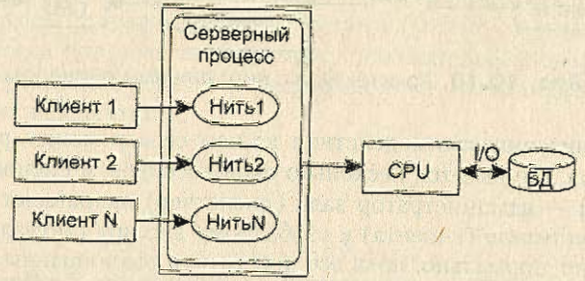


Рис.10.9. Многопотоковая односерверная архитектура

Кроме того, возможность взаимодействия с одним сервером многих клиентов позволяет в полной мере использовать разделяемые объекты (начиная с открытых файлов и кончая данными из системных каталогов), что значительно уменьшает потребности в памяти и общее число процессов операционной системы. Например, системой с архитектурой «один-к-одному» будет создано 100 копий процессов СУБД для 100 пользователей, тогда как системе с многопотоковой архитектурой для этого понадобится только один серверный процесс.

Однако такое решение имеет свои недостатки. Так как сервер может выполняться только на одном процессоре, возникает естественное ограничение на примене-

ние СУБД для мультипроцессорных платформ. Если компьютер имеет, например, четыре процессора, то СУБД с одним сервером используют только один из них, не загружая оставшиеся три.

В некоторых системах эта проблема решается вводом промежуточного диспетчера. Подобная архитектура называется архитектурой виртуального сервера («virtual server») (рис. 10.10).

В этой архитектуре клиенты подключаются не к реальному серверу, а к промежуточному звену, называемому диспетчером, который выполняет только функции диспетчеризации запросов к актуальным серверам. В этом случае нет ограничений на использование многопроцессорных платформ. Количество актуальных серверов может быть согласовано с количеством процессоров в системе.

Однако и эта архитектура не лишена недостатков, потому что здесь в систему добавляется новый слой, который размещается между клиентом и сервером, что увеличивает трату ресурсов на поддержку баланса загрузки актуальных серверов («load balancing») и ограничивает возможности управления взаимодействием «клиент—сервер». Во-первых, становится невозможным направить запрос от конкретного клиента конкретному серверу, во-вторых, серверы становятся равноправными — нет возможности устанавливать приоритеты для обслуживания запросов.

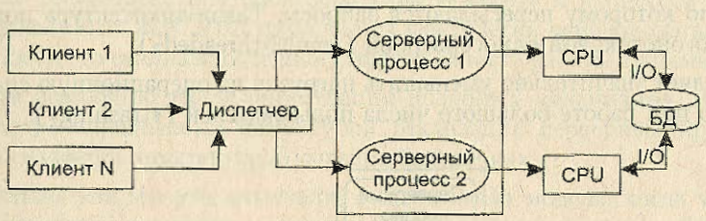


Рис. 10.10. Архитектура с виртуальным сервером

Подобная организация взаимодействия клиент-сервер может рассматриваться как аналог банка, где имеется несколько окон кассиров, и специальный банковский служащий — администратор зала (диспетчер) направляет каждого вновь пришедшего посетителя (клиента) к свободному кассиру (актуальному серверу). Система работает нормально, пока все посетители равноправны (имеют равные приоритеты), однако стоит лишь появиться посетителям с высшим приоритетом, которые должны обслуживаться в специальном окне, как возникают проблемы. Учет приоритета клиентов особенно важен в системах оперативной обработки транзакций, однако именно эту возможность не может предоставить архитектура систем с диспетчеризацией.

Современное решение проблемы СУБД для мультипроцессорных платформ заключается в возможности запуска нескольких серверов базы данных, в том числе и на различных процессорах. При этом каждый из серверов должен быть многопотоковым. Если эти два условия выполнены, то есть основания говорить о многопотоковой архитектуре с несколькими серверами, представленной на рис. 10.11.

Она также может быть названа многонитевой мультисерверной архитектурой. Эта архитектура связана с вопросами распараллеливания выполнения одного пользовательского запроса несколькими серверными процессами.

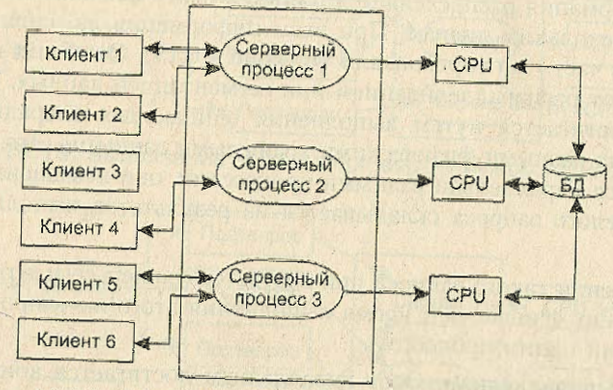


Рис. 10.11. Многопотоковая мультисерверная архитектура

Существует несколько возможностей распараллеливания выполнения запроса. В этом случае пользовательский запрос разбивается на ряд подзапросов, которые могут выполняться параллельно, а результаты их выполнения потом объединяются в общий результат выполнения запроса. Тогда для обеспечения оперативности выполнения запросов их подзапросы могут быть направлены отдельным серверным процессам, а потом полученные результаты объединены в общий результат (см. рис 10.12). В данном случае серверные процессы не являются независимыми процессами, такими, как рассматривались ранее. Эти серверные процессы принято называть нитями (treads), и управление нитями множества запросов пользователей требует дополнительных расходов от СУБД, однако при оперативной обработке информации в хранилищах данных такой подход наиболее перспективен.

Рис. 10.12. Многонитевая мультисерверная архитектура

Типы параллелизма

Рассматривают несколько путей распараллеливания запросов.

Горизонтальный параллелизм. Этот параллелизм возникает тогда, когда хранимая в БД информация распределяется по нескольким физическим устройствам хранения — нескольким дискам. При этом информация из одного отношения разбивается на части по горизонтали (см. рис. 10.13). Этот вид параллелизма иногда называют распараллеливанием или сегментацией данных. И параллельность здесь достигается путем выполнения одинаковых операций, например фильтрации, над разными физическими хранимыми данными. Эти операции могут выполняться параллельно разными процессами, они независимы. Результат выполнения целого запроса складывается из результатов выполнения отдельных операций.

Время выполнения такого запроса при соответствующем сегментировании данных существенно меньше, чем время выполнения этого же запроса традиционными способами одним процессом.

Вертикальный параллелизм. Этот параллелизм достигается конвейерным выполнением операций, составляющих запрос пользователя. Этот подход требует серьезного усложнения в модели выполнения реляционных операций ядром СУБД. Он предполагает, что ядро СУБД может произвести декомпозицию запроса, базируясь на его функциональных компонентах, и при этом ряд подзапросов может выполняться параллельно, с минимальной связью между отдельными шагами выполнения запроса.

Действительно, если мы рассмотрим, например, последовательность операций реляционной алгебры:

```
R5=R1 [ A,C]
R6=R2 [A,B,D]
R7 = R5[A > 128]
R8 = R5[A]R6.
```

то операции первую и третью можно объединить и выполнить параллельно с операцией два, а затем выполнить над результатами последнюю четвертую операцию.

Общее время выполнения подобного запроса, конечно, будет существенно меньше, чем при традиционном способе выполнения последовательности из четырех операций (см. рис. 10.13).

И третий вид параллелизма является гибридом двух ранее рассмотренных (см. рис. 10.14).

Наиболее активно применяются все виды параллелизма в OLAP-приложениях, где эти методы позволяют существенно сократить время выполнения сложных запросов над очень большими объемами данных.

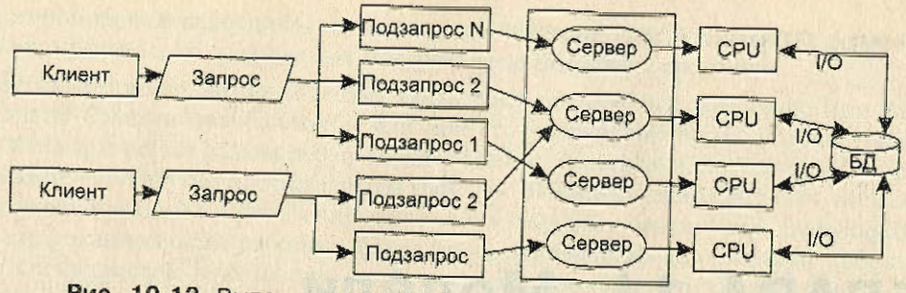


Рис. 10.13. Выполнение запроса при вертикальном параллелизме

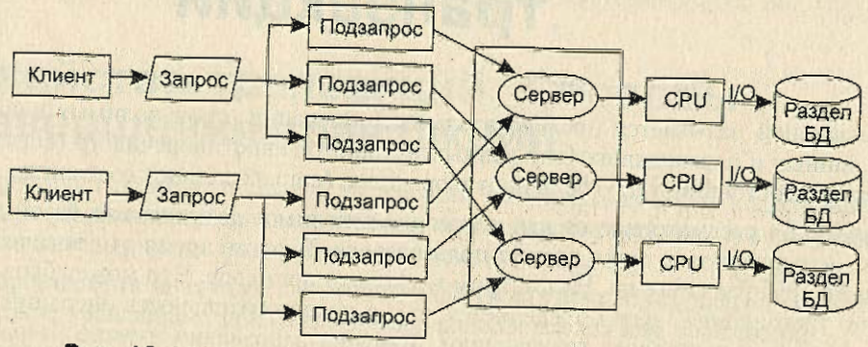


Рис. 10.14. Выполнение запроса при гибридном параллелизме

ГЛАВА 11 Модели транзакций

Транзакцией называется последовательность операций, производимых над базой данных и переводящих базу данных из одного непротиворечивого (согласованного) состояния в другое непротиворечивое (согласованное) состояние.

Транзакция рассматривается как некоторое неделимое действие над базой данных, осмысленное с точки зрения пользователя. В то же время это логическая единица работы системы. Рассмотрим несколько примеров. Что может быть названо транзакцией? Кем определяется, какая последовательность операций над базой данных составляет транзакцию? Конечно, однозначно именно разработчик определяет, какая последовательность операций составляет единое целое, то есть транзакцию. Разработчик приложений или хранимых процедур определяет это исходя из смысла обработки данных, именно семантика совокупности операций над базой данных, которая моделирует с точки зрения разработчика некоторую одну неразрывную работу, и составляет транзакцию. Допустим, выделим работу по вводу данных о поступивших книгах, новых книгах, которых не было раньше в библиотеке. Тогда эту операцию можно разбить на две последовательные: сначала ввод данных о книге — это новая строка в таблице BOOKS, а потом ввод данных обо всех экземплярах новой книги — это ввод набора новых строк в таблицу EXEMPLAR в количестве, равном количеству поступивших экземпляров книги. Если эта последовательность работ будет прервана, то наша база данных не будет соответствовать реальному объекту, поэтому желательно выполнять ее как единую работу над базой данных.

Следующий пример, который связан с принятием заказа в фирме на изготовление компьютера. Компьютер состоит из комплектующих, которые сразу резервируются за данным заказом в момент его формирования. Тогда транзакцией будет вся последовательность операций, включающая следующие операции:

- ввод нового заказа со всеми реквизитами заказчика;
- изменения состояния для всех выбранных комплектующих на складе на «занято» с привязкой их к определенному заказу;
- подсчет стоимости заказа с формированием платежного документа типа выставляемого счета к оплате;
- включение нового заказа в производство.

С точки зрения работника, это единая последовательность операций; если она будет прервана, то база данных потеряет свое целостное состояние.

Еще один пример, который весьма характерен для учебных заведений. При длительной болезни преподавателя или при его увольнении перед администрацией кафедры встает задача перераспределения всей нагрузки, которую ведет преподаватель, по другим преподавателям кафедры. Видов нагрузки может быть несколько: чтение лекций и проведение занятий по текущему расписанию, руководство квалификационными работами бакалавров, руководство дипломными проектами специалистов, руководство магистерскими диссертациями, индивидуальная научно-исследовательская работа со студентами. И для каждого вида нагрузки необходимо найти исполнителей и назначить им дополнительную нагрузку.

Свойства транзакций. Способы завершения транзакций

Существуют различные модели транзакций, которые могут быть классифицированы на основании различных свойств, включающих структуру транзакции, параллельность внутри транзакции, продолжительность и т. д.

В настоящий момент выделяют следующие типы транзакций: плоские или классические транзакции, цепочечные транзакции и вложенные транзакции.

Плоские, или традиционные, транзакции, характеризуются четырьмя классическими свойствами: атомарности, согласованности, изолированности, долговечности (прочности) — ACID (Atomicity, Consistency, Isolation, Durability). Иногда традиционные транзакции называют ACID-транзакциями. Упомянутые выше свойства означают следующее:

- *Свойство атомарности* (Atomicity) выражается в том, что транзакция должна быть выполнена в целом или не выполнена вовсе.
- *Свойство согласованности* (Consistency) гарантирует, что по мере выполнения транзакций данные переходят из одного согласованного состояния в другое — транзакция не разрушает взаимной согласованности данных.
- *Свойство изолированности* (Isolation) означает, что конкурирующие за доступ к базе данных транзакции физически обрабатываются последовательно, изолированно друг от друга, но для пользователей это выглядит так, как будто они выполняются параллельно.
- *Свойство долговечности* (Durability) трактуется следующим образом: если транзакция завершена успешно, то те изменения в данных, которые были ею произведены, не могут быть потеряны ни при каких обстоятельствах (даже в случае последующих ошибок).

Возможны два варианта завершения транзакции. Если все операторы выполнены успешно и в процессе выполнения транзакции не произошло никаких сбоев программного или аппаратного обеспечения, транзакция фиксируется.

Фиксация транзакции — это действие, обеспечивающее запись на диск изменений в базе данных, которые были сделаны в процессе выполнения транзакции.

До тех пор пока транзакция не зафиксирована, допустимо аннулирование этих изменений, восстановление базы данных в то состояние, в котором она была на момент начала транзакции. Фиксация транзакции означает, что все результаты выполнения транзакции становятся постоянными. Они станут видимыми другим транзакциям только после того, как текущая транзакция будет зафиксирована. До этого момента все данные, затрагиваемые транзакцией, будут «видны» пользователю в состоянии на начало текущей транзакции.

Если в процессе выполнения транзакции случилось нечто такое, что делает невозможным ее нормальное завершение, база данных должна быть возвращена в исходное состояние. Откат транзакции — это действие, обеспечивающее аннулирование всех изменений данных, которые были сделаны операторами SQL в теле текущей незавершенной транзакции.

Каждый оператор в транзакции выполняет свою часть работы, но для успешного завершения всей работы в целом требуется безусловное завершение всех их операторов. Группирование операторов в транзакции сообщает СУБД, что вся эта группа должна быть выполнена как единое целое, причем такое выполнение должно поддерживаться автоматически.

В стандарте ANSI/ISO SQL определены модель транзакций и функции операторов COMMIT и ROLLBACK. Стандарт определяет, что транзакция начинается с первого SQL-оператора, инициируемого пользователем или содержащегося в программе, изменяющего текущее состояние базы данных. Все последующие SQL-операторы составляют тело транзакции. Транзакция завершается одним из четырех возможных путей (рис. 11.1):

1) оператор COMMIT означает успешное завершение транзакции; его использование делает постоянными изменения, внесенные в базу данных в рамках текущей транзакции;
2) оператор ROLLBACK прерывает транзакцию, отменяя изменения, сделанные в базе данных в рамках этой транзакции; новая транзакция начинается непосредственно после использования ROLLBACK;
3) успешное завершение программы, в которой была инициирована текущая транзакция, означает успешное завершение транзакции (как будто был использован оператор COMMIT);
4) ошибочное завершение программы прерывает транзакцию (как будто был использован оператор ROLLBACK).

В этой модели каждый оператор, который изменяет состояние БД, рассматривается как транзакция, поэтому при успешном завершении этого оператора БД переходит в новое устойчивое состояние.

В первых версиях коммерческих СУБД была реализована модель транзакций ANSI/ISO. В дальнейшем в СУБД SYBASE была реализована расширенная модель транзакций, которая включает еще ряд дополнительных операций. В модели SYBASE используются следующие четыре оператора:

- Оператор BEGIN TRANSACTION сообщает о начале транзакции. В отличие от модели в стандарте ANSI/ISO, где начало транзакции неявно задается первым

оператором модификации данных, в модели SYBASE начало транзакции задается явно с помощью оператора начала транзакции.

- Оператор COMMIT TRANSACTION сообщает об успешном завершении транзакции. Он эквивалентен оператору COMMIT в модели стандарта ANSI/ISO. Этот оператор, как и оператор COMMIT, фиксирует все изменения, которые производились в БД в процессе выполнения транзакции.
- Оператор SAVE TRANSACTION создает внутри транзакции точку сохранения, которая соответствует промежуточному состоянию БД, сохраненному на момент выполнения этого оператора. В операторе SAVE TRANSACTION может стоять имя точки сохранения. Поэтому в ходе выполнения транзакции может быть запомнено несколько точек сохранения, соответствующих нескольким промежуточным состояниям.
- Оператор ROLLBACK имеет две модификации. Если этот оператор используется без дополнительного параметра, то он интерпретируется как оператор отката всей транзакции, то есть в этом случае он эквивалентен оператору отката ROLLBACK в модели ANSI/ISO. Если же оператор отката имеет параметр и записан в виде ROLLBACK B, то он интерпретируется как оператор частичного отката транзакции в точку сохранения B.

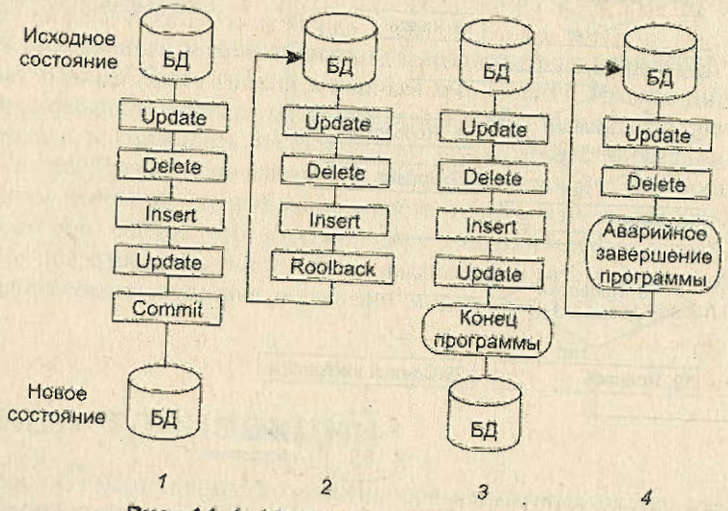


Рис. 11.1. Модель транзакций ANSI/ISO

Принципы выполнения транзакций в расширенной модели транзакций представлены на рис. 11.2. На рисунке операторы помечены номерами, чтобы нам удобнее было проследить ход выполнения транзакции во всех допустимых случаях.

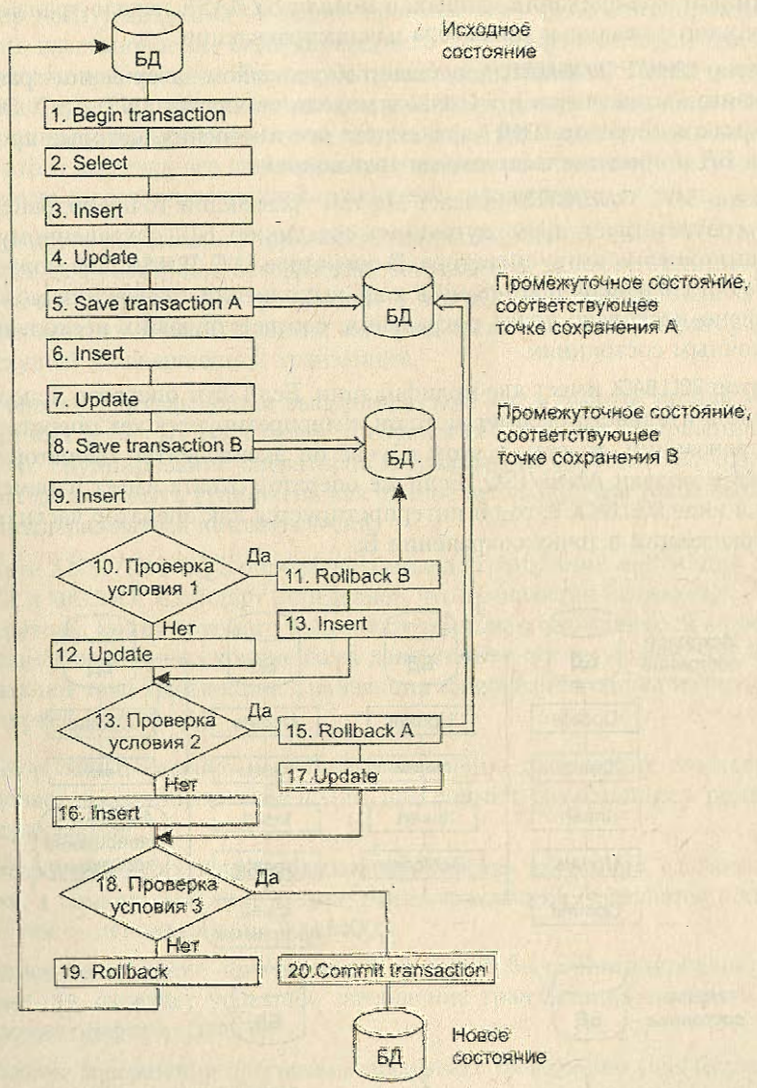


Рис. 11.2. Примеры выполнения транзакций в расширенной модели

Транзакция начинается явным оператором начала транзакции, который имеет в нашей схеме номер 1. Далее идет оператор 2, который является оператором поиска и не меняет текущее состояние БД, а следующие за ним операторы 3 и 4 переводят базу данных уже в новое состояние. Оператор 5 сохраняет это новое промежуточное состояние БД и помечает его как промежуточное состояние в точке A. Далее следуют операторы 6 и 7, которые переводят базу данных в новое состояние. А оператор 8 сохраняет это состояние как промежуточное состояние в точке B. Оператор 9 выполняет ввод новых данных, а оператор 10 проводит некоторую проверку условия 1; если условие 1 выполнено, то выполняется оператор 11, который проводит откат транзакции в промежуточное состояние B. Это означает, что последствия действий оператора 9 как бы стираются и база данных снова возвращается в промежуточное состояние B, хотя после выполнения оператора 9 она уже находилась в новом состоянии. И после отката транзакции вместо оператора 9, который выполнялся раньше из состояния B БД, выполняется оператор 13 ввода новых данных, и далее управление передается оператору 14. Оператор 14 снова проверяет условие, но уже некоторое новое условие 2; если условие выполнено, то управление передается оператору 15, который выполняет откат транзакции в промежуточное состояние A, то есть все операторы, которые изменяли БД, начиная с 6 и заканчивая 13, считаются невыполненными, то есть результаты их выполнения исчезли и мы снова находимся в состоянии A, как после выполнения оператора 4. Далее управление передается оператору 17, который обновляет содержимое БД, после этого управление передается оператору 18, который связан с проверкой условия 3. Проверка заканчивается либо передачей управления оператору 20, который фиксирует транзакцию, и БД переходит в новое устойчивое состояние, и изменить его в рамках текущей транзакции невозможно. Либо, если управление передано оператору 19, то транзакция откатывается к началу и БД возвращается в свое начальное состояние, а все промежуточные состояния здесь уже проверены, и выполнить операцию отката в эти промежуточные состояния после выполнения оператора 19 невозможно.

Конечно, расширенная модель транзакции, предложенная фирмой SYBASE, поддерживает гораздо более гибкий механизм выполнения транзакций. Точки сохранения позволяют устанавливать маркеры внутри транзакции таким образом, чтобы имелась возможность отмены только части работы, проделанной в транзакции. Целесообразно использовать точки сохранения в длинных и сложных транзакциях, чтобы обеспечить возможность отмены изменения для определенных операторов. Однако это обусловливает дополнительные затраты ресурсов системы — оператор выполняет работу, а изменения затем отменяются; обычно усовершенствования в логике обработки могут оказаться более оптимальным решением.

Журнал транзакций

Реализация в СУБД принципа сохранения промежуточных состояний, подтверждения или отката транзакции обеспечивается специальным механизмом, для поддержки которого создается некоторая системная структура, называемая *Журналом транзакций*.

Однако назначение журнала транзакций гораздо шире. Он предназначен для обеспечения надежного хранения данных в БД.

А это требование предполагает, в частности, возможность восстановления согласованного состояния базы данных после любого рода аппаратных и программных сбоев. Очевидно, что для выполнения восстановлений необходима некоторая дополнительная информация. В подавляющем большинстве современных реляционных СУБД такая избыточная дополнительная информация поддержи-

вается в виде журнала изменений базы данных, чаще всего называемого *Журналом транзакций*.

Итак, общей целью журнализации изменений баз данных является обеспечение возможности восстановления согласованного состояния базы данных после любого сбоя. Поскольку основой поддержания целостного состояния базы данных является механизм транзакций, журнализация и восстановление тесно связаны с понятием транзакции. Общими принципами восстановления являются следующие:

- результаты зафиксированных транзакций должны быть сохранены в восстановленном состоянии базы данных;
- результаты незафиксированных транзакций должны отсутствовать в восстановленном состоянии базы данных.

Это, собственно, и означает, что восстанавливается последнее по времени согласованное состояние базы данных.

Возможны следующие ситуации, при которых требуется производить восстановление состояния базы данных.

- Индивидуальный откат транзакции. Этот откат должен быть применен в следующих случаях:
  - стандартной ситуацией отката транзакции является ее явное завершение оператором ROLLBACK;
  - аварийное завершение работы прикладной программы, которое логически эквивалентно выполнению оператора ROLLBACK, но физически имеет иной механизм выполнения;
  - принудительный откат транзакции в случае взаимной блокировки при параллельном выполнении транзакций. В подобном случае для выхода из тупика данная транзакция может быть выбрана в качестве «жертвы» и принудительно прекращено ее выполнение ядром СУБД.
- Восстановление после внезапной потери содержимого оперативной памяти (мягкий сбой). Такая ситуация может возникнуть в следующих случаях:
  - при аварийном выключении электрического питания;
  - при возникновении неустранимого сбоя процессора (например, срабатывании контроля оперативной памяти) и т. д. Ситуация характеризуется потерей той части базы данных, которая к моменту сбоя содержалась в буферах оперативной памяти.
- Восстановление после поломки основного внешнего носителя базы данных (жесткий сбой). Эта ситуация при достаточно высокой надежности современных устройств внешней памяти может возникать сравнительно редко, но тем не менее СУБД должна быть в состоянии восстановить базу данных даже и в этом случае. Основой восстановления является архивная копия и журнал изменений базы данных.

Для восстановления согласованного состояния базы данных при индивидуальном откате транзакции нужно устранить последствия операторов модификации

базы данных, которые выполнялись в этой транзакции. Для восстановления непротиворечивого состояния БД при мягком сбое необходимо восстановить содержимое БД по содержимому журналов транзакций, хранящихся на дисках. Для восстановления согласованного состояния БД при жестком сбое надо восстановить содержимое БД по архивным копиям и журналам транзакций, которые хранятся на неповрежденных внешних носителях.

Во всех трех случаях основой восстановления является избыточное хранение данных. Эти избыточные данные хранятся в журнале, содержащем последовательность записей об изменении базы данных.

Возможны два основных варианта ведения журнальной информации. В первом варианте для каждой транзакции поддерживается отдельный локальный журнал изменений базы данных этой транзакцией. Такие журналы называются локальными журналами. Они используются для индивидуальных откатов транзакций и могут поддерживаться в оперативной (правильнее сказать, в виртуальной) памяти. Кроме того, поддерживается общий журнал изменений базы данных, используемый для восстановления состояния базы данных после мягких и жестких сбоев.

Этот подход позволяет быстро выполнять индивидуальные откаты транзакций, но приводит к дублированию информации в локальных и общем журналах. Поэтому чаще используется второй вариант — поддержание только общего журнала изменений базы данных, который используется и при выполнении индивидуальных откатов. Далее мы рассматриваем именно этот вариант.

Общая структура журнала условно может быть представлена в виде некоторого последовательного файла, в котором фиксируется каждое изменение БД, которое происходит в ходе выполнения транзакции. Все транзакции имеют свои внутренние номера, поэтому в едином журнале транзакций фиксируются все изменения, проводимые всеми транзакциями.

Каждая запись в журнале транзакций помечается номером транзакции, к которой она относится, и значениями атрибутов, которые она меняет. Кроме того, для каждой транзакции в журнале фиксируется команда начала и завершения транзакции (см. рис. 11.3).

Для большей надежности журнал транзакций часто дублируется системными средствами коммерческих СУБД, именно поэтому объем внешней памяти во много раз превышает реальный объем данных, которые хранятся в хранилище.

Имеются два альтернативных варианта ведения журнала транзакций: протокол с отложенными обновлениями и протокол с немедленными обновлениями.

Ведение журнала по принципу отложенных изменений предполагает следующий механизм выполнения транзакций:

1. Когда транзакция T1 начинается, в протокол заносится запись
   <T1 Begin transaction>
2. На протяжении выполнения транзакции в протоколе для каждой изменяемой записи записывается новое значение: <T1, ID_RECORD, атрибут, новое значение … >. Здесь ID_RECORD — уникальный номер записи.

3. Если все действия, из которых состоит транзакция T1, успешно выполнены, то транзакция частично фиксируется и в протокол заносится <T1 COMMIT>.
4. После того как транзакция фиксирована, записи протокола, относящиеся к T1, используются для внесения соответствующих изменений в БД.
5. Если происходит сбой, то СУБД просматривает протокол и выясняет, какие транзакции необходимо переделать. Транзакцию T1 необходимо переделать, если протокол содержит обе записи <T1 BEGIN TRANSACTION> и <T1 COMMIT>. БД может находиться в несогласованном состоянии, однако все новые значения измененных элементов данных содержатся в протоколе, и это требует повторного выполнения транзакции. Для этого используется некоторая системная процедура REDO(), которая заменяет все значения элементов данных на новые, просматривая протокол в прямом порядке.
6. Если в протоколе не содержится команда фиксации транзакции COMMIT, то никаких действий проводить не требуется, а транзакция запускается заново.

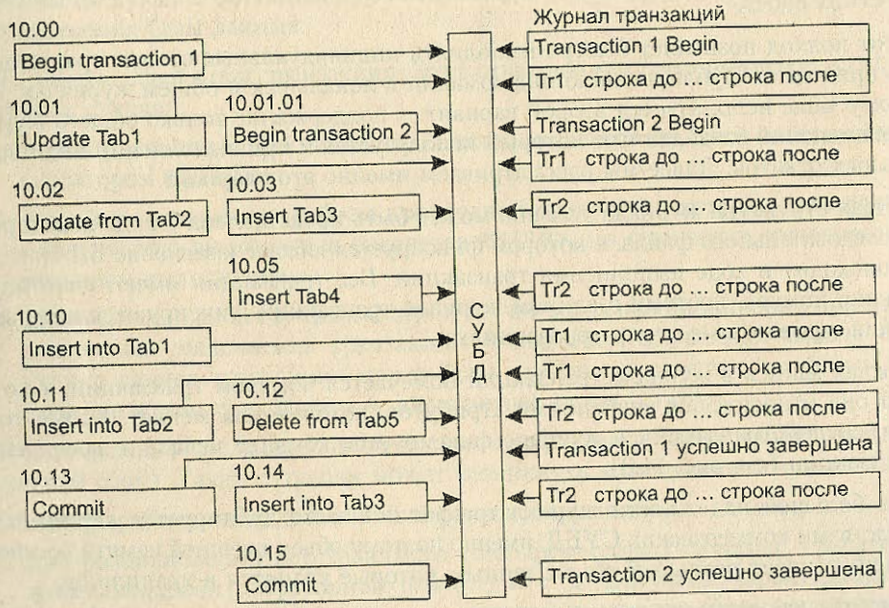


Рис. 11.3. Журнал транзакций

Альтернативный механизм с немедленным выполнением предусматривает внесение изменений сразу в БД, а в протокол заносятся не только новые, но и все старые значения изменяемых атрибутов, поэтому каждая запись выглядит <T1, ID_RECORD, атрибут новое значение старое значение ...>. При этом запись в журнал предшествует непосредственному выполнению операции над БД. Когда транзакция фиксируется, то есть встречается команда <T1 COMMIT> и она выполняется, то все изменения оказываются уже внесенными в БД и не требуется никаких дальнейших действий по отношению к этой транзакции.

При откате транзакции выполняется системная процедура UNDO(), которая возвращает все старые значения в отмененной транзакции, последовательно проходя по протоколу начиная с команды BEGIN TRANSACTION.

Для восстановления при сбое используется следующий механизм:

- Если транзакция содержит команду начала транзакции, но не содержит команды фиксации с подтверждением ее выполнения, то выполняется последовательность действий как при откате транзакции, то есть восстанавливаются старые значения.
- Если сбой произошел после выполнения последней команды изменения БД, но до выполнения команды фиксации, то команда фиксации выполняется, а с БД никаких изменений не происходит. Работа происходит только на уровне протокола.
- Однако следует отметить, что проблемы восстановления выглядят гораздо сложнее приведенных ранее алгоритмов, с учетом того, что изменения как в журнал, так и в БД заносятся не сразу, а буферируются. Этому посвящен следующий раздел.

Журнализация и буферизация

Журнализация изменений тесно связана не только с управлением транзакциями, но и с буферизацией страниц базы данных в оперативной памяти.

Если бы запись об изменении базы данных, которая должна поступить в журнал при выполнении любой операции модификации базы данных, реально немедленно записывалась бы во внешнюю память, это привело бы к существенному замедлению работы системы. Поэтому записи в журнале тоже буферизуются: при нормальной работе очередная страница выталкивается во внешнюю память журнала только при полном заполнении записями.

Проблема состоит в выработке некоторой общей политики выталкивания, которая обеспечивала бы возможность восстановления состояния базы данных после сбоев.

Проблема не возникает при индивидуальных откатах транзакций, поскольку в этих случаях содержимое оперативной памяти не утрачено и можно пользоваться содержимым как буфера журнала, так и буферов страниц базы данных. Но если произошел мягкий сбой и содержимое буферов утрачено, для проведения восстановления базы данных необходимо иметь некоторое согласованное состояние журнала и базы данных во внешней памяти.

Основным принципом согласованной политики выталкивания буфера журнала и буферов страниц базы данных является то, что запись об изменении объекта базы данных должна попадать во внешнюю память журнала раньше, чем измененный объект оказывается во внешней памяти базы данных. Соответствующий протокол журнализации (и управления буферизацией) называется Write Ahead Log (WAL) — «пиши сначала в журнал» и состоит в том, что если требуется записать во внешнюю память измененный объект базы данных, то перед этим нужно гарантировать запись во внешнюю память журнала транзакций записи о его изменении.

Другими словами, если во внешней памяти базы данных находится некоторый объект базы данных, по отношению к которому выполнена операция модификации, то во внешней памяти журнала обязательно находится запись, соответствующая этой операции. Обратное неверно, то есть если во внешней памяти журнале содержится запись о некоторой операции изменения объекта базы данных, то сам измененный объект может отсутствовать во внешней памяти базы данных.

Дополнительное условие на выталкивание буферов накладывается тем требованием, что каждая успешно завершившаяся транзакция должна быть реально зафиксирована во внешней памяти. Какой бы сбой не произошел, система должна быть в состоянии восстановить состояние базы данных, содержащее результаты всех зафиксированных к моменту сбоя транзакций.

Простым решением было бы выталкивание буфера журнала, за которым следует массовое выталкивание буферов страниц базы данных, изменявшихся данной транзакцией. Довольно часто так и делают, но это вызывает существенные накладные расходы при выполнении операции фиксации транзакции.

Оказывается, что минимальным требованием, гарантирующим возможность восстановления последнего согласованного состояния базы данных, является выталкивание при фиксации транзакции во внешнюю память журнала всех записей об изменении базы данных этой транзакцией. При этом последней записью в журнал, производимой от имени данной транзакции, является специальная запись о конце транзакции.

Рассмотрим теперь, как можно выполнять операции восстановления базы данных в различных ситуациях, если в системе поддерживается общий для всех транзакций журнал с общей буферизацией записей, поддерживаемый в соответствии с протоколом WAL.

Индивидуальный откат транзакции

Для того чтобы можно было выполнить по общему журналу индивидуальный откат транзакции, все записи в журнале по данной транзакции связываются в обратный список. Началом списка для незакончившихся транзакций является запись о последнем изменении базы данных, произведенном данной транзакцией. Для закончившихся транзакций (индивидуальные откаты которых уже невозможны) началом списка является запись о конце транзакции, которая обязательно вытолкнута во внешнюю память журнала. Концом списка всегда служит первая запись об изменении базы данных, произведенном данной транзакцией. Обычно в каждой записи проставляется уникальный идентификатор транзакции, чтобы можно было восстановить прямой список записей об изменениях базы данных данной транзакцией.

Итак, индивидуальный откат транзакции (еще раз подчеркнем, что это возможно только для незакончившихся транзакций) выполняется следующим образом:

- Выбирается очередная запись из списка данной транзакции.
- Выполняется противоположная по смыслу операция: вместо операции INSERT выполняется соответствующая операция DELETE, вместо операции DELETE выполняется INSERT и вместо прямой операции UPDATE обратная операция UPDATE, восстанавливающая предыдущее состояние объекта базы данных.
- Любая из этих обратных операций также заносится в журнал. Собственно, для индивидуального отката это не нужно, но при выполнении индивидуального отката транзакции может произойти мягкий сбой, при восстановлении после которого потребуется откатить такую транзакцию, для которой не полностью выполнен индивидуальный откат.
- При успешном завершении отката в журнал заносится запись о конце транзакции. С точки зрения журнала такая транзакция является зафиксированной.

Восстановление после мягкого сбоя

К числу основных проблем восстановления после мягкого сбоя относится то, что одна логическая операция изменения базы данных может изменять несколько физических блоков базы данных, например, страницу данных и несколько страниц индексов. Страницы базы данных буферизуются в оперативной памяти и выталкиваются независимо. Несмотря на применение протокола WAL, после мягкого сбоя набор страниц внешней памяти базы данных может оказаться несогласованным, то есть часть страниц внешней памяти соответствует объекту до изменения, часть — после изменения. К такому состоянию объекта неприменимы операции логического уровня.

Состояние внешней памяти базы данных называется физически согласованным, если наборы страниц всех объектов согласованы, то есть соответствуют состоянию объекта либо до его изменения, либо после изменения.

Будем считать, что в журнале отмечаются точки физической согласованности базы данных — моменты времени, в которые во внешней памяти содержатся согласованные результаты операций, завершившихся до соответствующего момента времени, и отсутствуют результаты операций, которые не завершились, а буфер журнала вытолкнут во внешнюю память. Немного позже мы рассмотрим, как можно достичь физической согласованности. Назовем такие точки tpc (time of physical consistency) — точками физического согласования.

Тогда к моменту мягкого сбоя возможны следующие состояния транзакций:

- транзакция успешно завершена, то есть выполнена операция подтверждения транзакции COMMIT и для всех операций транзакции получено подтверждение ее выполнения во внешней памяти;
- транзакция успешно завершена, но для некоторых операций не получено подтверждение их выполнения во внешней памяти;
- транзакция получила и выполнила команду отката ROLLBACK;
- транзакция не завершена.

Физическая согласованность базы данных

Каким же образом можно обеспечить наличие точек физической согласованности базы данных, то есть как восстановить состояние базы данных в момент tpc? Для этого используются два основных подхода: подход, основанный на исполь-

зовании теневого механизма, и подход, в котором применяется журнализация постраничных изменений базы данных.

При открытии файла таблица отображения номеров его логических блоков в адреса физических блоков внешней памяти считывается в оперативную память. При модификации любого блока файла во внешней памяти выделяется новый блок. При этом текущая таблица отображения (в оперативной памяти) изменяется, а теневая — сохраняется неизменной. Если во время работы с открытым файлом происходит сбой, во внешней памяти автоматически сохраняется состояние файла до его открытия. Для явного восстановления файла достаточно повторно считать в оперативную память теневую таблицу отображения.

Общая идея теневого механизма показана на рис. 11.4.

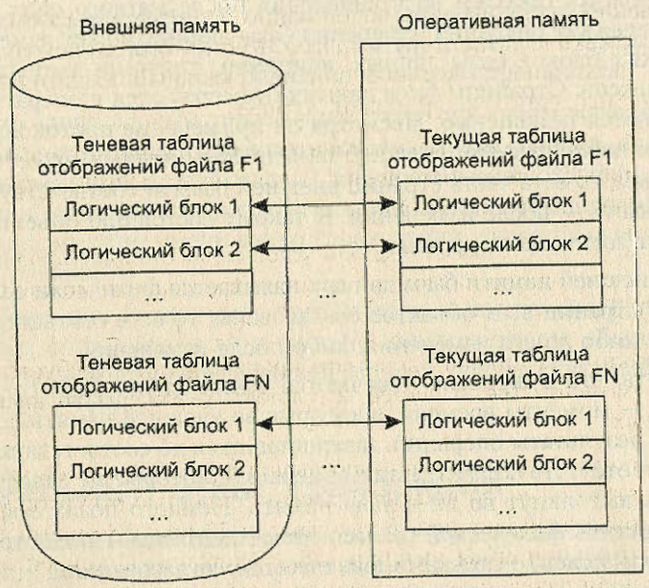


Рис. 11.4. Использование теневых таблиц отображения информации

В контексте базы данных теневой механизм используется следующим образом. Периодически выполняются операции установления точки физической согласованности базы данных (checkpoints). Для этого все логические операции завершаются, все буферы оперативной памяти, содержимое которых не соответствует содержимому соответствующих страниц внешней памяти, выталкиваются. Теневая таблица отображения файлов базы данных заменяется на текущую (правильнее сказать, текущая таблица отображения записывается на место теневой).

Восстановление к tpc происходит мгновенно: текущая таблица отображения заменяется на теневую (при восстановлении просто считывается теневая таблица отображения). Все проблемы восстановления решаются, но за счет слишком большого перерасхода внешней памяти. В пределе может потребоваться вдвое больше внешней памяти, чем реально нужно для хранения базы данных. Теневой механизм — это надежное, но слишком грубое средство. Обеспечивается

согласованное состояние внешней памяти в один общий для всех объектов момент времени. На самом деле достаточно иметь совокупность согласованных наборов страниц, каждому из которых может соответствовать свои временные отсчеты.

Для выполнения такого более слабого требования наряду с логической журнализацией операций изменения базы данных производится журнализация постраничных изменений. Первый этап восстановления после мягкого сбоя состоит в постраничном откате незакончившихся логических операций. Подобно тому как это делается с логическими записями по отношению к транзакциям, последней записью о постраничных изменениях от одной логической операции является запись о конце операции.

В этом подходе имеются два метода решения проблемы. При использовании первого метода поддерживается общий журнал логических и страничных операций. Естественно, наличие двух видов записей, интерпретируемых абсолютно по-разному, усложняет структуру журнала. Кроме того, записи о постраничных изменениях, актуальность которых носит локальный характер, существенно (и не очень осмысленно) увеличивают журнал.

Поэтому все более популярным становится поддержание отдельного (короткого) журнала постраничных изменений. Такая техника применяется, например, в известном продукте Informix Online.

Предположим, что некоторым способом удалось восстановить внешнюю память базы данных к состоянию на момент времени tpc (как это можно сделать — немного позже). Тогда:

- Для транзакции T1 никаких действий производить не требуется. Она закончилась до момента tpc, и все ее результаты отражены во внешней памяти базы данных.
- Для транзакции T2 нужно повторно выполнить оставшуюся часть операций (redo). Действительно, во внешней памяти полностью отсутствуют следы операций, которые выполнялись в транзакции T2 после момента tpc. Следовательно, повторная прямая интерпретация операций T2 корректна и приведет к логически согласованному состоянию базы данных (поскольку транзакция T2 успешно завершилась до момента мягкого сбоя, в журнале содержатся записи обо всех изменениях, произведенных этой транзакцией).
- Для транзакции T3 нужно выполнить в обратном направлении первую часть операций (undo). Действительно, во внешней памяти базы данных полностью отсутствуют результаты операций T3, которые были выполнены после момента tpc. С другой стороны, во внешней памяти гарантированно присутствуют результаты операций T3, которые были выполнены до момента tpc. Следовательно, обратная интерпретация операций T3 корректна и приведет к согласованному состоянию базы данных (поскольку транзакция T3 не завершилась к моменту мягкого сбоя, при восстановлении необходимо устранить все последствия ее выполнения).
- Для транзакции T4, которая успела начаться после момента tpc и закончиться до момента мягкого сбоя, нужно выполнить полную повторную прямую интерпретацию операций (redo).

- Наконец, для начавшейся после момента tpc и не успевшей завершиться к моменту мягкого сбоя транзакции T5 никаких действий предпринимать не требуется. Результаты операций этой транзакции полностью отсутствуют во внешней памяти базы данных.

Восстановление после жесткого сбоя

Понятно, что для восстановления последнего согласованного состояния базы данных после жесткого сбоя журнала изменений базы данных явно недостаточно. Основой восстановления в этом случае являются журнал и архивная копия базы данных.

Восстановление начинается с обратного копирования базы данных из архивной копии. Затем для всех закончившихся транзакций выполняется redo, то есть операции повторно выполняются в прямом порядке.

Более точно, происходит следующее:

- по журналу в прямом направлении выполняются все операции;
- для транзакций, которые не закончились к моменту сбоя, выполняется откат.

На самом деле, поскольку жесткий сбой не сопровождается утратой буферов оперативной памяти, можно восстановить базу данных до такого уровня, чтобы можно было продолжить даже выполнение незакончившихся транзакций. Но обычно это не делается, потому что восстановление после жесткого сбоя — это достаточно длительный процесс.

Хотя к ведению журнала предъявляются особые требования по части надежности, в принципе возможна и его утрата. Тогда единственным способом восстановления базы данных является возврат к архивной копии. Конечно, в этом случае не удастся получить последнее согласованное состояние базы данных, но это лучше, чем ничего.

Последний вопрос, который мы коротко рассмотрим, относится к производству архивных копий базы данных. Самый простой способ — архивировать базу данных при переполнении журнала. В журнале вводится так называемая «желтая зона», при достижении которой образование новых транзакций временно блокируется. Когда все транзакции закончатся и, следовательно, база данных придет в согласованное состояние, можно производить ее архивацию, после чего начинать заполнять журнал заново.

Можно выполнять архивацию базы данных реже, чем переполняется журнал. При переполнении журнала и окончании всех начатых транзакций можно архивировать сам журнал. Поскольку такой архивированный журнал, по сути дела, требуется только для воссоздания архивной копии базы данных, журнальная информация при архивации может быть существенно сжата.

Параллельное выполнение транзакций

Если с БД работают одновременно несколько пользователей, то обработка транзакций должна рассматриваться с новой точки зрения. В этом случае СУБД должна не только корректно выполнять индивидуальные транзакции и восстанавливать согласованное состояние БД после сбоев, но она призвана обеспечить корректную параллельную работу всех пользователей над одними и теми же данными. По теории каждый пользователь и каждая транзакция должны обладать свойством изолированности, то есть они должны выполняться так, как если бы только один пользователь работал с БД. И средства современных СУБД позволяют изолировать пользователей друг от друга именно таким образом. Однако в этом случае возникают проблемы замедления работы пользователей. Рассмотрим более подробно проблемы, которые возникают при параллельной обработке транзакций.

Основные проблемы, которые возникают при параллельном выполнении транзакций, делятся условно на 4 типа:

- Пропавшие изменения. Эта ситуация может возникать, если две транзакции одновременно изменяют одну и ту же запись в БД. Например, работают два оператора на приеме заказов, первый оператор принял заказ на 30 мониторов. Когда он запрашивал склад, то там числилось 40 мониторов, и он, получив подтверждение от клиента, выставил счет и оформил продажу 30 мониторов из 40. Параллельно с ним работает второй оператор, который принимает заказ на 20 таких же мониторов (ну уж очень хорошая модель и дешево) и, в свою очередь запросив состояние склада и получив исходно ту же цифру 40, он успешно оформляет заказ для своего клиента. Заканчивая работу с данным заказом, он выполняет команду Обновить (UPDATE), которая заносит 20 как остаток любимых мониторов на складе. Но после этого, наконец, любезно попрощавшись со своим клиентом и заверив его в скорейшей доставке заказанных мониторов, заканчивает работу со своим заказом первый оператор и также выполняет команду Обновить и заносит 10 как остаток тех же мониторов на складе. Каждый из них доволен своей работой, но мы-то знаем, что произошло. Прежде всего, они продали 50 мониторов из наличествующих 40 штук, и далее на складе еще числится 10 подобных мониторов. БД теперь находится в несогласованном состоянии, а у фирмы возникли серьезные проблемы. Изменения, сделанные вторым оператором, были проигнорированы программой выполнения заказа, с которой работал первый оператор. Подобная ситуация представлена на рис. 11.5.
- Проблемы промежуточных данных. Рассмотрим ту же проблему одновременной работы двух операторов. Допустим, первый оператор, ведя переговоры со своим заказчиком, ввел заказанные 30 мониторов, но перед окончательным оформлением заказа клиент захотел выяснить еще некоторые характеристики товара. Приложение, с которым работает первый оператор, уже изменило остаток мониторов на складе, и там сейчас находится информация о 10 оставшихся мониторах. В это время второй оператор пытается принять

заказ от своего клиента на 20 мониторов, но его приложение показывает, что на складе осталось всего 10 мониторов, и оператор вынужден отказать выгодному клиенту, который идет в другую фирму, весьма неудовлетворенный работой нашей компании. А в этот момент клиент оператора 1 заканчивает обсуждение дополнительных характеристик наших мониторов и принимает весьма невыгодное решение не покупать у нас мониторы, и приложение оператора 1 выполняет откат транзакции, и на складе снова оказывается 40 мониторов. Мы потеряли выгодного заказчика, но еще хуже было бы, если бы клиент второго оператора согласился на 10 оставшихся мониторов, и приложение, с которым работает оператор два, отработав свой алгоритм, занесло 0 (ноль) оставшихся мониторов на складе, а после этого приложение оператора один снова бы записало исходные 40 мониторов на складе, хотя 10 их них уже проданы. Такая ситуация оказалась возможной потому, что приложение второго оператора имело доступ к промежуточным данным, которые сформировало первое приложение.

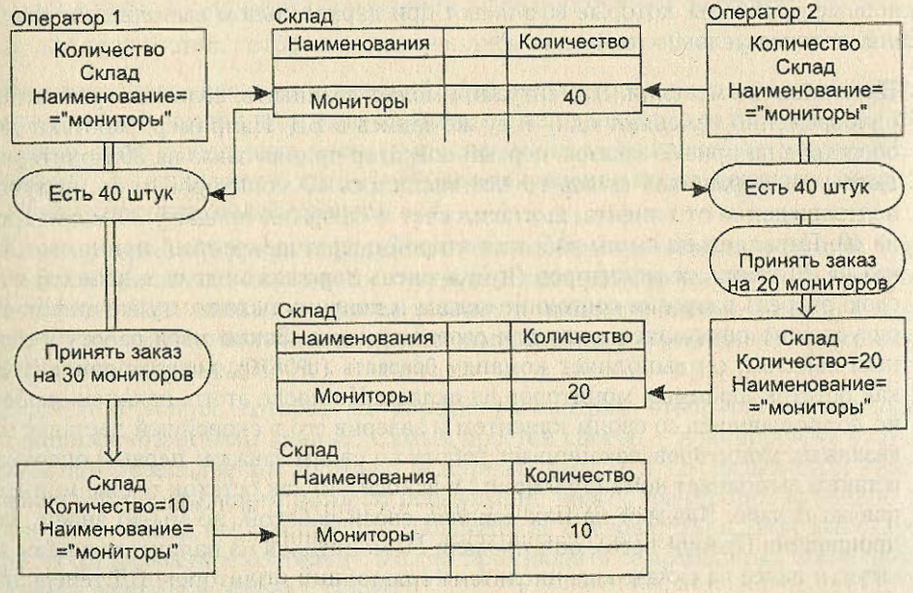


Рис. 11.5. Проблема пропавших обновлений

- Проблемы несогласованных данных. Рассмотрим ту же самую ситуацию с заказом мониторов. Предположим, что ситуация несколько изменилась. И оба оператора начинают работать практически одновременно. Они оба получают начальное состояние склада 40 мониторов, а далее первый оператор успешно завершает переговоры со своим клиентом и продает ему 30 мониторов. Он завершает работу своего приложения, и оно выполняет команду фиксации транзакции `COMMIT`. Состояние базы данных непротиворечивое. В этот момент, выяснив все тонкости и характеристики наших мониторов, клиент второго оператора также решает сделать заказ, и второй оператор, повторно получая состояние склада, видит, что оно изменилось. База данных находится в непротиворечивом состоянии, но второй оператор считает, что нарушена целостность его транзакции, в течение выполнения одной работы он получил два различных состояния склада. Эта ситуация возникла потому, что приложение первого оператора смогло изменить кортеж с данными, который уже прочитало приложение второго оператора.

- Проблемы строк-призраков (строк-фантомов). Предположим, что администратор нашей фирмы поручил секретарю напечатать итоговый отчет по результатам работы за текущий месяц. И допустим, что приложение печатает отчет в двух видах: в подробном и в укрупненном. В момент, когда приложение печати начало формировать свой первый вид отчета, один из операторов принимает еще один заказ, поэтому к моменту формирования укрупненного отчета в БД появились новые сведения о продажах, которые и были внесены в укрупненный отчет. Мы получили два отчета в одном приложении, которые содержат разные цифры и не совпадают друг с другом. Такое стало возможно потому, что приложение печати выполнило два одинаковых запроса и получило два разных результата. БД находится в согласованном состоянии, но приложение печати работает некорректно.

Для того чтобы избежать подобных проблем, требуется выработать некоторую процедуру согласованного выполнения параллельных транзакций. Эта процедура должна удовлетворять следующим правилам:

- В ходе выполнения транзакции пользователь видит только согласованные данные. Пользователь не должен видеть несогласованных промежуточных данных.
- Когда в БД две транзакции выполняются параллельно, то СУБД гарантированно поддерживает принцип независимого выполнения транзакций, который гласит, что результаты выполнения транзакций будут такими же, как если бы вначале выполнялась транзакция 1, а потом транзакция 2, или наоборот, сначала транзакция 2, а потом транзакция 1.

Такая процедура называется сериализацией транзакций. Фактически она гарантирует, что каждый пользователь (программа), обращающаяся к базе данных, работает с ней так, как будто не существует других пользователей (программ), одновременно с ним обращающихся к тем же данным.

Для поддержки параллельной работы транзакций строится специальный план.

План (способ) выполнения набора транзакций называется сериальным, если результат совместного выполнения транзакций эквивалентен результату некоторого последовательного выполнения этих же транзакций.

Самым простым было бы последовательное выполнение транзакций, но такой план не оптимален по времени, существуют более гибкие методы управления параллельным доступом к БД. Наиболее распространенным механизмом, который используется коммерческими СУБД для реализации на практике сериализации транзакций является механизм блокировок. Самый простой вариант — это блокировка объекта на все время действия транзакции. Подобный пример рассмотрен на рис. 11.6. Здесь две транзакции, названные условно A и B, рабо-

тают с тремя таблицами: T1, T2 и T3. В момент начала работы с любым объектом этот объект блокируется транзакцией, которая с ним начала работу, и он становится недоступным всем другим транзакциям до окончания транзакции, заблокировавшей («захватившей») данный объект. После окончания транзакции все заблокированные ею объекты разблокируются и становятся доступными другим транзакциям. Если транзакция обращается к заблокированному объекту, то она остается в состоянии ожидания до момента разблокировки этого объекта, после чего она может продолжать обработку данного объекта. Поэтому транзакция B ожидает разблокировки таблицы T2 транзакцией A. Над прямоугольниками стоит условное время выполнения операций.

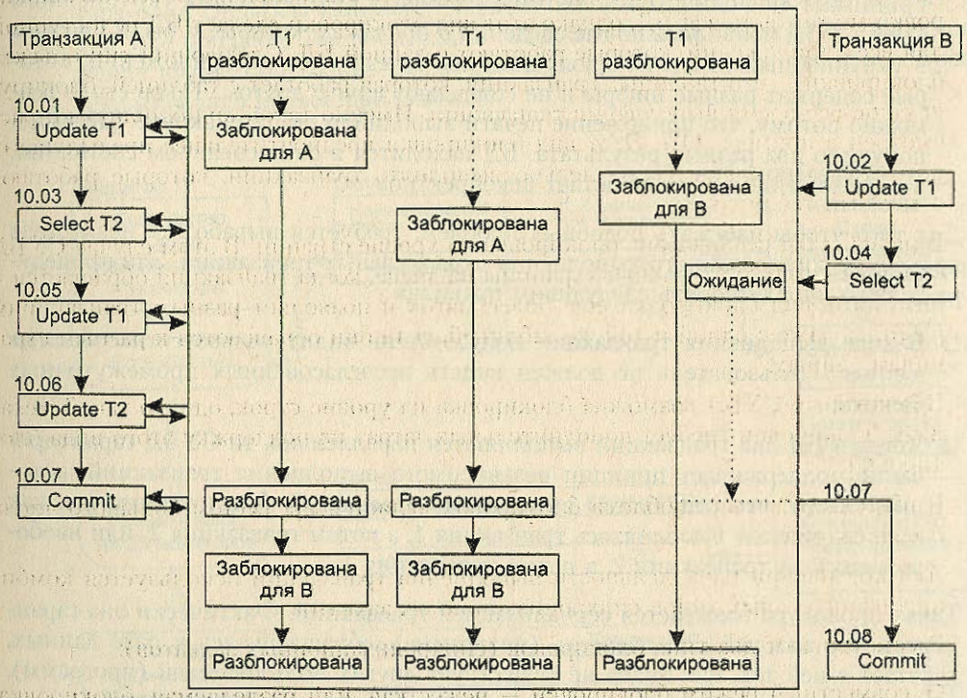


Рис. 11.6. Блокировки при одновременном выполнении двух транзакций

В общем случае на момент выполнения транзакция получает как бы монопольный доступ к объектам БД, с которыми она работает. В этом случае другие транзакции не получают доступа к объектам БД до момента окончания транзакции. Такой механизм действительно ликвидирует все перечисленные ранее проблемы: пропавшие изменения, неподтвержденные данные, несогласованные данные, строки-фантомы. Однако такая блокировка создает новые проблемы — задержку выполнения транзакций из-за блокировок.

Рассмотрим существующие типы конфликтов между двумя параллельными транзакциями. Можно выделить следующие типы:

- W-W — транзакция 2 пытается изменять объект, измененный незакончившейся транзакцией 1;
- R-W — транзакция 2 пытается изменять объект, прочитанный незакончившейся транзакцией 1;
- W-R — транзакция 2 пытается читать объект, измененный незакончившейся транзакцией 1.

Практические методы сериализации транзакций основываются на учете этих конфликтов.

Блокировки, называемые также синхронизационными захватами объектов, могут быть применены к разному типу объектов. Наибольшим объектом блокировки может быть вся БД, однако этот вид блокировки сделает БД недоступной для всех приложений, которые работают с данной БД. Следующий тип объекта блокировки — это таблицы. Транзакция, которая работает с таблицей, блокирует ее на все время выполнения транзакции. Именно такой вид блокировки рассмотрен в примере 11.7. Этот вид блокировки предпочтительнее предыдущего, потому что позволяет параллельно выполнять транзакции, которые работают с другими таблицами.

В ряде СУБД реализована блокировка на уровне страниц. В этом случае СУБД блокирует только отдельные страницы на диске, когда транзакция обращается к ним. Этот вид блокировки еще более мягок и позволяет разным транзакциям работать даже с одной и той же таблицей, если они обращаются к разным страницам данных.

В некоторых СУБД возможна блокировка на уровне строк, однако такой механизм блокировки требует дополнительных затрат на поддержку этого вида блокировки.

В настоящее время проблема блокировок является предметом большого числа исследований.

Для повышения параллельности выполнения транзакций используется комбинирование разных типов синхронизационных захватов.

Рассматривают два типа блокировок (синхронизационных захватов):

- совместный режим блокировки — нежесткая, или разделяемая, блокировка, обозначаемая как S (Shared). Этот режим обозначает разделяемый захват объекта и требуется для выполнения операции чтения объекта. Объекты, заблокированные таким образом, не изменяются в ходе выполнения транзакции и доступны другим транзакциям также, но только в режиме чтения;
- монопольный режим блокировки — жесткая, или эксклюзивная, блокировка, обозначаемая как X (eXclusive). Данный режим блокировки предполагает монопольный захват объекта и требуется для выполнения операций занесения, удаления и модификации. Объекты, заблокированные данным типом блокировки, фактически остаются в монопольном режиме обработки и недоступны для других транзакций до момента окончания работы данной транзакции.

Захваты объектов несколькими транзакциями по чтению совместимы, то есть нескольким транзакциям допускается читать один и тот же объект, захват объ-

екта одной транзакцией по чтению не совместим с захватом другой транзакцией того же объекта по записи, и захваты одного объекта разными транзакциями по записи не совместимы. Правила совместимости захватов одного объекта разными транзакциями изображены на рис. 11.7:

		Транзакция B		
		Разблокирована	Нежесткая блокировка	Жесткая блокировка
Транзакция A	Разблокирована	Да	Да	Да
	Нежесткая блокировка	Да	Да	Нет
	Жесткая блокировка	Да	Нет	Нет

Рис.11.7. Правила применения жесткой и нежесткой блокировок транзакций

В примере, представленном на рис. 11.7 считается, что первой блокирует объект транзакция A, а потом пытается получить к нему доступ транзакция B.

На рис. 11.8 приведен ранее рассмотренный пример с выполнением транзакций 1 и 2, но с учетом разных типов блокировки. На рисунке видно, что, применив нежесткую блокировку к таблице 2 со стороны транзакции 1, мы обеспечили существенное уменьшение времени выполнения транзакции 2. Теперь транзакция 2 не ждет окончания транзакции 1, и поэтому завершает свою работу намного раньше.

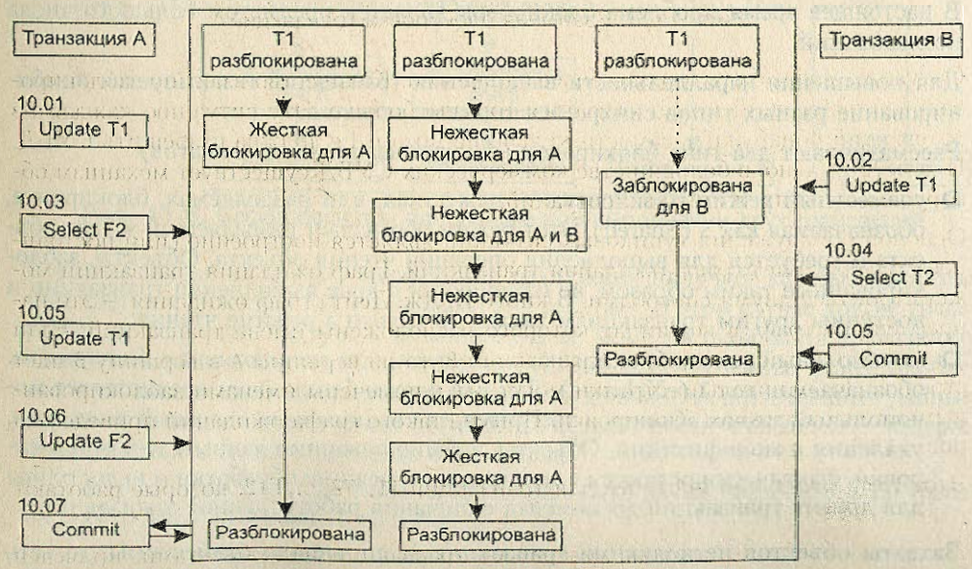


Рис. 11.8. Использование жесткой и нежесткой блокировки

К сожалению, применения разных типов блокировок приводит к проблеме тупиков. Эта проблема не нова. Проблема тупиков возникла при рассмотрении выполнения параллельных процессов в операционных средах и также была связана с управлением разделяемыми (совместно используемыми) ресурсами.

Действительно, рассмотрим пример. Пусть транзакция A сначала жестко блокирует таблицу 1, а потом жестко блокирует таблицу 2. Транзакция B, наоборот, сначала жестко блокирует таблицу 2, а потом жестко блокирует таблицу 1. Если обе эти транзакции начали работу одновременно, то после выполнения операций модификации первыми объектами каждой транзакции они обе окажутся в бесконечном ожидании: транзакция A будет ждать завершения работы транзакции B и разблокировки таблицы 2, а транзакция B также безрезультатно будет ждать окончания работы транзакции A и разблокировки таблицы 1 (см. рис. 11.9).

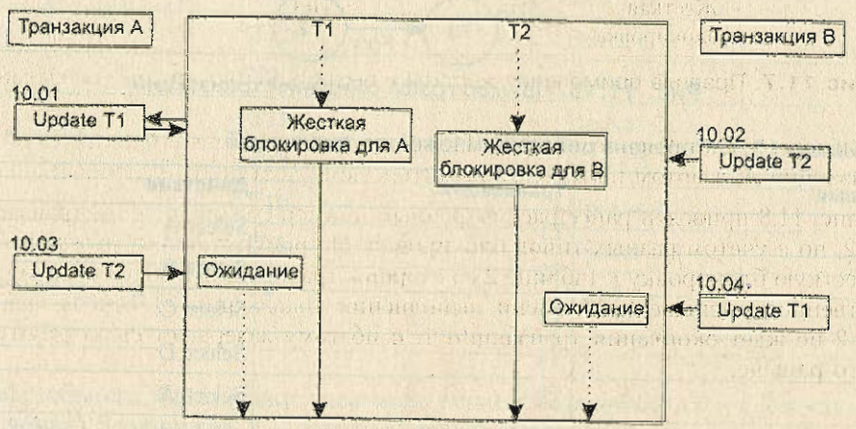


Рис. 11.9. Взаимная блокировка транзакций

Ситуации могут быть гораздо более сложными. Количество взаимно заблокированных транзакций может оказаться гораздо больше. Эту ситуацию каждая из транзакций обнаружить самостоятельно не может. Ее должна разрешить СУБД. И действительно, в большинстве коммерческих СУБД существует механизм обнаружения таких тупиковых ситуаций.

Основой обнаружения тупиковых ситуаций является построение (или постоянное поддержание) графа ожидания транзакций. Граф ожидания транзакций может строиться двумя способами. В книге К. Дж. Дейта граф ожидания — это направленный граф, в вершинах которого расположены имена транзакций. Если транзакция A ждет окончания транзакции B, то из вершины A в вершину B идет стрелка. Дополнительно стрелки могут быть помечены именами заблокированных объектов и типом блокировки. Пример такого графа ожиданий приведен на рис. 11.10.

Этот граф ожиданий построен для транзакций T1, T2,...,T12, которые работают с объектами БД A,B,...,H.

Перечень действий, которые совершают транзакции над объектами, приведен в табл. 11.1.

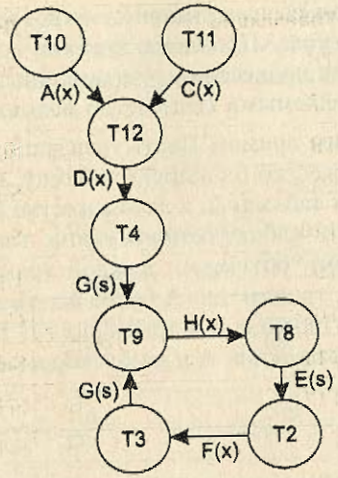


Рис. 11.10. Пример графа ожиданий транзакций

Таблица 11.1. Перечень действий множества транзакций

Время	Транзакция	Действие
0	T1	Select A
1	T2	Select B
2	T1	Select C
3	T4	Select D
4	T5	Select A
5	T2	Select E
6	T2	Update E
7	T3	Select F
8	T2	Select F
9	T5	Update A
10	T1	Commit
11	T6	Select A
12	T5	Commit
13	T6	Select C
14	T6	Update C
15	T7	Select G
16	T8	Select H
17	T9	Select G
18	T9	Update G

Время	Транзакция	Действие
19	T8	Select E
20	T7	Commit
21	T9	Select H
22	T3	Select G
23	T10	Select A
24	T9	Update H
25	T6	Commit
26	T11	Select C
27	T12	Select D
28	T12	Select C
29	T2	Update F
30	T11	Update C
31	T12	Select A
32	T10	Update A
33	T12	Update D
34	T2	Select G
35	–	–

На графе объекты блокировки помечены типами блокировок, `S` — нежесткая (разделяемая) блокировка, `X` — жесткая (эксклюзивная) блокировка.

На диаграмме состояний ожидания видно, что транзакции `T9`, `T8`, `T2` и `T3` образуют цикл. Именно наличие цикла и является признаком возникновения тупиковой ситуации. Поэтому в момент 3 перечисленные транзакции будут заблокированы.

Разрушение тупика начинается с выбора в цикле транзакций так называемой транзакции-жертвы, то есть транзакции, которой решено пожертвовать, чтобы обеспечить возможность продолжения работы других транзакций.

Критерием выбора является стоимость транзакции; жертвой выбирается самая дешевая транзакция. Стоимость транзакции определяется на основе многофакторной оценки, в которую с разными весами входят время выполнения, число накопленных захватов, приоритет.

После выбора транзакции-жертвы выполняется откат этой транзакции, который может носить полный или частичный характер. При этом, естественно, освобождаются захваты и может быть продолжено выполнение других транзакций.

В лекциях профессора С. Д. Кузнецова приводится несколько иной принцип построения графа ожидания. В этом случае граф ожидания транзакций строится в виде ориентированного двудольного графа, в котором существует два типа вершин — вершины, соответствующие транзакциям, и вершины, соответствую-

щие объектам захвата. В этом графе существует дуга, ведущая из вершины-транзакции к вершине-объекту, если для этой транзакции существует удовлетворенный захват объекта. В графе существует дуга из вершины-объекта к вершине-транзакции, если транзакция ожидает удовлетворения захвата объекта.

Для распознавания тупика здесь, так же как и в первом методе, производится построение графа ожидания транзакций и в этом графе ищутся циклы. Традиционной техникой (для которой существует множество разновидностей) нахождения циклов в ориентированном графе является редукция графа.

Не вдаваясь в детали, редукция состоит в том, что прежде всего из графа ожидания удаляются все дуги, исходящие из вершин-транзакций, в которые не входят дуги из вершин-объектов. (Это как бы соответствует той ситуации, что транзакции, не ожидающие удовлетворения захватов, успешно завершились и освободили захваты.) Для тех вершин-объектов, для которых не осталось входящих дуг, но существуют исходящие, ориентация исходящих дуг изменяется на противоположную (это моделирует удовлетворение захватов). После этого снова срабатывает первый шаг, и так до тех пор, пока на первом шаге не прекратится удаление дуг. Если в графе остались дуги, то они обязательно образуют цикл.

Естественно, такое насильственное устранение тупиковых ситуаций является нарушением принципа изолированности пользователей.

Заметим, что в централизованных системах стоимость построения графа ожидания сравнительно невелика, но она становится слишком большой в по-настоящему распределенных СУБД, в которых транзакции могут выполняться в разных узлах сети. Поэтому в таких системах обычно используются другие методы сериализации транзакций.

Для обеспечения сериализации транзакций синхронизационные захваты объектов, произведенные по инициативе транзакции, можно снимать только при ее завершении. Это требование порождает двухфазный протокол синхронизационных захватов — 2PL(two phase lock) или 2PC (two phase commit). В соответствии с этим протоколом выполнение транзакции разбивается на две фазы:

- первая фаза транзакции — накопление захватов;
- вторая фаза (фиксация или откат) — освобождение захватов.

В языке SQL введен оператор явной блокировки таблицы, который позволяет точно задать тип блокировки для всей таблицы. Синтаксис операции блокировки имеет вид:

```
LOCK TABLE имя_таблицы IN {SHARED | EXCLUSIVE} MODE
```

Имеет смысл блокировать таблицу полностью, когда выполняется операция множественной модификации одной таблицы, то есть когда в ней изменяется большое количество строк. Эта операция иногда называется пакетным обновлением.

Конечно, у блокировки таблицы есть тот недостаток, что все остальные транзакции должны ждать окончания обновления таблицы. Но режим пакетного обновления одной таблицы работает достаточно быстро, и общая производительность выполнения множества транзакций может даже повыситься в этом случае.

Уровни изолированности пользователей

Достаточно легко убедиться, что при соблюдении двухфазного протокола синхронизационных захватов действительно обеспечивается полная сериализация транзакций. Однако иногда приложению, которое выполняет транзакцию, не сколько важны точные данные, сколько скорость выполнения запросов. Например, в системах поддержки принятия решений по электронным торгам важно просто иметь представление об общей картине торгов, на основании которого принимается решение об повышении или снижении ставок и т. д. Для смягчения требований сериализации транзакций вводится понятие уровня изолированности пользователя.

Уровни изолированности пользователей связаны с проблемами, которые возникают при параллельном выполнении транзакций и которые были рассмотрены нами ранее.

Всего введено 4 уровня изолированности пользователей. Самый высокий уровень изолированности соответствует протоколу сериализации транзакций, это уровень SERIALIZABLE. Этот уровень обеспечивает полную изоляцию транзакций и полную корректную обработку параллельных транзакций.

Следующий уровень изолированности называется уровнем подтвержденного чтения — REPEATABLE READ. На этом уровне транзакция не имеет доступа к промежуточным или окончательным результатам других транзакций, поэтому такие проблемы, как пропавшие обновления, промежуточные или несогласованные данные, возникнуть не могут. Однако во время выполнения своей транзакции вы можете увидеть строку, добавленную в БД другой транзакцией. Поэтому один и тот же запрос, выполненный в течение одной транзакции, может дать разные результаты, то есть проблема строк-призраков остается. Однако если такая проблема критична, лучше ее разрешать алгоритмически, изменяя алгоритм обработки, исключая повторное выполнение запроса в одной транзакции.

Второй уровень изолированности связан с подтвержденным чтением, он называется READ COMMITED. На этом уровне изолированности транзакция не имеет доступа к промежуточным результатам других транзакций, поэтому проблемы пропавших обновлений и промежуточных данных возникнуть не могут. Однако окончательные данные, полученные в ходе выполнения других транзакций, могут быть доступны нашей транзакции. При этом уровне изолированности транзакция не может обновлять строку, уже обновленную другой транзакцией. При попытке выполнить подобное обновление транзакция будет отменена автоматически, во избежание возникновения проблемы пропавшего обновления.

И наконец, самый низкий уровень изолированности называется уровнем неподтвержденного, или грязного, чтения. Он обозначается как READ UNCOMMITED. При этом уровне изолированности текущая транзакция видит промежуточные и несогласованные данные, и также ей доступны строки-призраки. Однако даже при этом уровне изолированности СУБД предотвращает пропавшие обновления.

В стандарте SQL2 существует оператор задания уровня изолированности выполнения транзакции. Он имеет следующий синтаксис:

```
SET TRANSACTION IZOLATION LEVEL [{SERIALIZABLE |
    REPEATABLE READ |
    READ COMMITED |
    READ UNCOMMITED}] [{READ WRITE |
    READ ONLY }]
```

Дополнительно в этом операторе может быть указано, операции какого типа выполняются в транзакции. По умолчанию предполагается уровень SERIALIZABLE. Если задан уровень READ UNCOMMITED, то допустимы только операции чтения в транзакции, поэтому в этом случае нельзя установить операции READ WRITE. На рис. 11.11 приведено соответствие уровней изолированности транзакций и проблем, возникающих при параллельном выполнении транзакций.

Уровень изоляции	Появление пропавших обновлений	Появление промежуточных данных	Появление несогласованных данных	Появление строк-призраков
SERIALIZABLE	СУБД предотвращает	СУБД предотвращает	СУБД предотвращает	СУБД предотвращает
REPEATABLE READ	СУБД предотвращает	СУБД предотвращает	СУБД предотвращает	Может произойти
READ COMMITED	СУБД предотвращает	СУБД предотвращает	Может произойти	Может произойти
READ UNCOMMITED	СУБД предотвращает	Может произойти	Может произойти	Может произойти

Рис.11.11. Уровни изолированности транзакций и проблемы многопользовательской работы

В разных коммерческих СУБД могут быть реализованы не все уровни изолированности, это необходимо выяснить в технической документации.

Гранулированные синхронизационные захваты

Мы уже говорили, что объектами блокирования могут быть объекты разного уровня, начиная с целой БД и заканчивая кортежем.

Понятно, что чем крупнее объект синхронизационного захвата (неважно, какой природы этот объект — логический или физический), тем меньше синхронизационных захватов будет поддерживаться в системе, и при этом, соответственно, будут меньшие накладные расходы. Более того, если выбрать в качестве уровня объектов для захватов файл или отношение, то будет решена даже проблема фантомов (если это не ясно сразу, посмотрите еще раз на формулировку проблемы фантомов и определение двухфазного протокола захватов).

Но вся беда в том, что при использовании для захватов крупных объектов возрастает вероятность конфликтов транзакций и тем самым уменьшается допускаемая степень их параллельного выполнения. Фактически при укрупнении

объекта синхронизационного захвата мы умышленно огрубляем ситуацию и видим конфликты в тех ситуациях, когда на самом деле конфликтов нет. Действительно, если транзакция T1 обрабатывает первую, пятую и двадцатую строку в таблице R1, но блокирует всю таблицу, то транзакция T2, которая обрабатывает шестую и восьмую строки той же таблицы не сможет получить к ним доступ, хотя на уровне строк никаких конфликтов нет.

В большинстве современных систем используются покортежные, то есть построковые синхронизационные захваты.

Однако нелепо было бы применять покортежную блокировку в случае выполнения, например, операции удаления всего отношения или удаления всех строк в отношении.

Подобные рассуждения привели к понятию гранулированных синхронизационных захватов и разработке соответствующего механизма.

При применении этого подхода синхронизационные захваты могут запрашиваться по отношению к объектам разного уровня: файлам, отношениям и кортежам. Требуемый уровень объекта определяется тем, какая операция выполняется (например, для выполнения операции уничтожения отношения объектом синхронизационного захвата должно быть все отношение, а для выполнения операции удаления кортежа — этот кортеж). Объект любого уровня может быть захвачен в режиме S (разделяемом) или X (монопольном). Вводится специальный протокол гранулированных захватов и определены новые типы захватов: перед захватом объекта в режиме S или X соответствующий объект более высокого уровня должен быть захвачен в режиме IS, IX или SIX.

IS (Intented for Shared lock, предваряющий разделяемую блокировку) по отношению к некоторому составному объекту O означает намерение захватить некоторый входящий в O объект в совместном режиме. Например, при намерении читать кортежи из отношения R это отношение должно быть захвачено в режиме IS (а до этого в таком же режиме должен быть захвачен файл).

IX (Intented for eXclusive lock, предваряющий жесткую блокировку) по отношению к некоторому составному объекту O означает намерение захватить некоторый входящий в O объект в монопольном режиме. Например, при намерении удалять кортежи из отношения R это отношение должно быть захвачено в режиме IX (а до этого в таком же режиме должен быть захвачен файл).

SIX (Shared, Intented for eXclusive lock, разделяемая блокировка объекта, предваряющая дальнейшие жесткие блокировки его составляющих) по отношению к некоторому составному объекту O означает совместный захват всего этого объекта с намерением впоследствии захватывать какие-либо входящие в него объекты в монопольном режиме. Например, если выполняется длинная операция просмотра отношения с возможностью удаления некоторых просматриваемых кортежей, то экономичнее всего захватить это отношение в режиме SIX (а до этого захватить файл в режиме IS).

Весьма трудно описать словами все возможные ситуации. Приведем полную таблицу совместимости захватов, анализируя которую можно выявить все случаи (см. табл. 11.2).

Таблица 11.2. Матрица совместимости блокировок.

L1\L2	X	S	IX	IS	SIX
Нет блокировки	Да	Да	Да	Да	Да
X	Нет	Нет	Нет	Нет	Нет
S	Нет	Да	Нет	Да	Нет
IX	Нет	Нет	Да	Да	Нет
IS	Нет	Да	Да	Да	Да
SIX	Нет	Нет	Нет	Да	Нет

Протокол гранулированных захватов требует соблюдения следующих правил:

1. Прежде чем транзакция установит S-блокировку на данный кортеж, она должна установить блокировку IS или другую, более сильную блокировку на отношение, в котором содержится данный кортеж.
2. Прежде чем транзакция установит X-блокировку на данный кортеж, она должна установить IX-блокировку или другую более сильную блокировку на отношение, в которое входит кортеж.

Блокировка L1 называется более сильной по отношению к блокировке L2 тогда и только тогда, когда для любой конфликтной ситуации (Нет — недопустимо) в столбце блокировки L2 в некоторой строке матрицы совместимости блокировок (см. табл. 11.2) существует также конфликт в столбце блокировки L1 в той же строке.

Диаграмма приоритетов блокировок приведена на рис. 11.12.

Рис. 11.12. Диаграмма приоритета блокировок различных типов

Предикатные синхронизационные захваты

Несмотря на привлекательность метода гранулированных синхронизационных захватов, следует отметить, что он не решает проблему фантомов (если, конечно, не ограничиться использованием захватов отношений в режимах S и X).

Известно, что проблема фантомов не возникает, если объектом блокировки является целое отношение. Именно это свойство и послужило основой разработки

метода предикатных синхронизационных захватов. В этом случае мы рассматриваем захват отношения — простой и частный случай предикатного захвата.

Суть этого метода — оценить множество кортежей, которое связано с той или иной транзакций, и если эти два множества, относящиеся к одному отношению, не пересекаются, то две транзакции могут оперировать ими параллельно без взаимной блокировки, а результаты выполнения обеих транзакций будут корректными.

Поскольку любая операция над реляционной базой данных задается некоторым условием (то есть в ней указывается не конкретный набор объектов базы данных, над которыми нужно выполнить операцию, а условие, которому должны удовлетворять объекты этого набора), идеальным выбором было бы требовать синхронизационный захват в режиме S или X именно этого условия. Но если посмотреть на общий вид условий, допускаемых, например, в языке SQL, то становится абсолютно непонятно, как определить совместимость двух предикатных захватов. Ясно, что без этого использовать предикатные захваты для синхронизации транзакций невозможно, а в общей форме проблема неразрешима.

К счастью, эта проблема сравнительно легко решается для случая простых условий. Будем называть простым условием конъюнкцию простых предикатов, имеющих вид:

```
имя-атрибута { операция сравнения } значение
```

Здесь операция сравнения: =, >, <

В типичных СУБД, поддерживающих двухуровневую организацию (языковой уровень и уровень управления внешней памяти), в интерфейсе подсистем управления памятью (которая обычно заведует и сериализацией транзакций) допускаются только простые условия. Подсистема языкового уровня производит компиляцию исходного оператора со сложным условием в последовательность обращений к ядру СУБД, в каждом из которых содержатся только простые условия. Следовательно, в случае типовой организации реляционной СУБД простые условия можно использовать как основу предикатных захватов.

Для простых условий совместимость предикатных захватов легко определяется на основе следующей геометрической интерпретации. Пусть R — отношение с атрибутами a_1, a_2, ..., a_n, а m_1, m_2, ..., m_n — множества допустимых значений a_1, a_2, ..., a_n соответственно (все эти множества — конечные). Тогда можно сопоставить R конечное n-мерное пространство возможных значений кортежей R. Любое простое условие «вырезает» m-мерный прямоугольник в этом пространстве (m <= n).

Тогда S-X, X-S, X-X предикатные захваты от разных транзакций совместимы, если соответствующие прямоугольники не пересекаются.

Это иллюстрируется следующим примером, показывающим, что в каких бы режимах не требовала транзакция 1 захвата условия (1<=a<=4) & (b=5), а транзакция 2 — условия (1<=a<=5) & (1<=b<=3), эти захваты всегда совместимы.

Пример: (n = 2)

Заметим, что предикатные захваты простых условий описываются таблицами, немногим отличающимися от таблиц традиционных синхронизаторов.

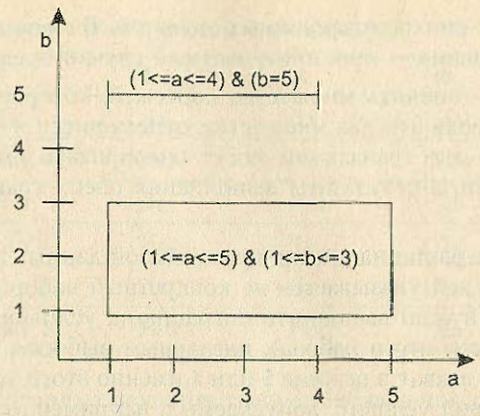


Рис. 11.13. Области действия предикатных захватов

Метод временных меток

Альтернативный метод сериализации транзакций, хорошо работающий в условиях редких конфликтов транзакций и не требующий построения графа ожидания транзакций, основан на использовании временных меток.

Основная идея метода (у которого существует множество разновидностей) состоит в следующем: если транзакция T1 началась раньше транзакции T2, то система обеспечивает такой режим выполнения, как если бы T1 была целиком выполнена до начала T2.

Для этого каждой транзакции T предписывается временная метка t, соответствующая времени начала T. При выполнении операции над объектом r транзакция T помечает его своей временной меткой и типом операции (чтение или изменение).

Перед выполнением операции над объектом r транзакция T1 выполняет следующие действия:

- Проверяет, не закончилась ли транзакция T, пометившая этот объект. Если T закончилась, T1 помечает объект r и выполняет свою операцию.
- Если транзакция T не завершилась, то T1 проверяет конфликтность операций. Если операции неконфликтны, при объекте r остается или проставляется временная метка с меньшим значением, и транзакция T1 выполняет свою операцию.
- Если операции T1 и T конфликтуют, то если t(T) > t(T1) (то есть транзакция T является более «молодой», чем T1), производится откат T и T1 продолжает работу.
- Если же t(T) < t(T1) (T «старше» T1), то T1 получает новую временную метку и начинается заново.

К недостаткам метода временных меток относятся потенциально более частые откаты транзакций, чем в случае использования синхронизационных захватов. Это связано с тем, что конфликтность транзакций определяется более грубо.

Кроме того, в распределенных системах не очень просто вырабатывать глобальные временные метки с отношением полного порядка (это отдельная большая наука).

Но в распределенных системах эти недостатки окупаются тем, что не нужно распознавать тупики, а как мы уже отмечали, построение графа ожидания в распределенных системах стоит очень дорого.

ГЛАВА 12 Встроенный SQL

Язык SQL, как мы уже видели в главе 5, предназначен для организации доступа к базам данных. При этом предполагается, что доступ к БД может быть осуществлен в двух режимах: в интерактивном режиме и в режиме выполнения прикладных программ (приложений).

Эта двойственность SQL создает ряд преимуществ:

- Все возможности интерактивного языка запросов доступны и в прикладном программировании.
- Можно в интерактивном режиме отладить основные алгоритмы обработки информации, которые в дальнейшем могут быть готовыми вставлены в работающие приложения.

SQL действительно является языком по работе с базами данных, но в явном виде он не является языком программирования. В нем отсутствуют традиционные операторы, организующие циклы, позволяющие объявить и использовать внутренние переменные, организовать анализ некоторых условий и возможность изменения хода программы в зависимости от выполненного условия. В общем случае можно назвать SQL подъязыком, который служит исключительно для управления базами данных. Для создания приложений, настоящих программ необходимо использовать другие, базовые языки программирования, в которые операторы языка SQL будут встраиваться.

Базовыми языками программирования могут быть языки C, COBOL, PL/1, Pascal.

Существуют два способа применения SQL в прикладных программах:

- *Встроенный SQL*. При таком подходе операторы SQL встраиваются непосредственно в исходный тегст программы на базовом языке. При компиляции программы со встроенными операторами SQL используется специальный препроцессор SQL, который преобразует исходный текст в исполняемую программу.
- *Интерфейс программирования приложений* (*API application program interface*). При использовании данного метода прикладная программа взаимодействует с СУБД путем применения специальных функций. Вызывая эти функции, программа передает СУБД операторы SQL и получает обратно результаты запросов. В этом случае не требуется специализированный препроцессор.

Процесс выполнения операторов SQL может быть условно разделен на 5 этапов (см. рис. 12.1).

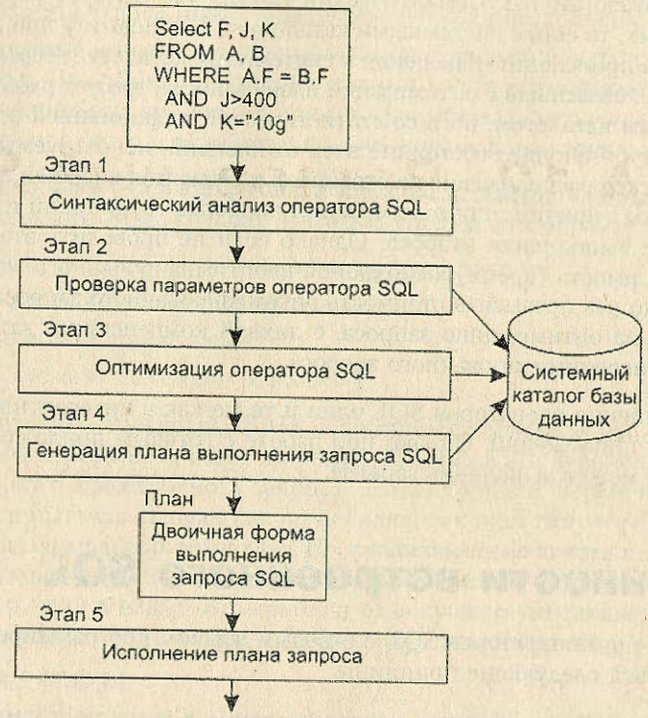


Рис. 12.1. Процесс выполнения операторов SQL

1. На первом этапе выполняется синтаксический анализ оператора SQL. На этом этапе проверяется корректность записи SQL-оператора в соответствии с правилами синтаксиса.
2. На этом этапе проверяется корректность параметров оператора SQL: имен отношений, имен полей данных, привилегий пользователя по работе с указанными объектами. Здесь обнаруживаются семантические ошибки.
3. На этом этапе проводится оптимизация запроса. СУБД проводит разделение целостного запроса на ряд минимальных операций и оптимизирует последовательность их выполнения с точки зрения стоимости выполнения запроса. На этом этапе строится несколько планов выполнения запроса и выбирается из них один — оптимальный для данного состояния БД.
4. На четвертом этапе СУБД генерирует двоичную версию оптимального плана запроса, подготовленного на этапе 3. Двоичный план выполнения запроса в СУБД фактически является эквивалентом объектного кода программы.
5. И наконец, только на пятом этапе СУБД реализует (выполняет) разработанный план, тем самым выполняя оператор SQL.

6. Следует отметить, что перечисленные этапы отличаются по числу обращений к БД и по процессорному времени, требуемому для их выполнения. Синтаксический анализ проводится очень быстро, он не требует обращения к системным каталогам БД. Семантический анализ уже требует работы с базой мета-данных, то есть с системными каталогами БД, поэтому при выполнении этого этапа происходит обращение к системному каталогу и серьезная работа с ним. Этап, связанный с оптимизаций плана запроса, требует работы не только с системным каталогом, но и со статистической информацией о БД, которая характеризует текущее состояние всех отношений, используемых в запросе, их физическое расположение на страницах и сегментах внешней памяти. В силу указанных причин этап оптимизации наиболее трудоемкий и длительный в процессе выполнения запроса. Однако если не проводить этап оптимизации, то стоимость (время) выполнения неоптимизированного запроса может в несколько раз превысить стоимость оптимизированного запроса. Время, потраченное на оптимизацию запроса, с лихвой компенсирует затраты на выполнение неоптимизированного запроса.

Этапы выполнения операторов SQL одни и те же как в интерактивном режиме, так и внутри приложений. Однако при работе с готовым приложением многие этапы СУБД может выполнить заранее.

Особенности встроенного SQL

При объединении операторов SQL с базовым языком программирования должны соблюдаться следующие принципы:

- Операторы SQL включаются непосредственно в текст программы на исходном языке программирования. Исходная программа поступает на вход препроцессора SQL, который компилирует операторы SQL.
- Встроенные операторы SQL могут ссылаться на переменные базового языка программирования.
- Встроенные операторы SQL получают результаты SQL-запросов с помощью переменных базового языка программирования.
- Для присвоения неопределенных значений (NULL) атрибутам отношений БД используются специальные функции.
- Для обеспечения построчной обработки результатов запросов во встроенный SQL добавляются несколько новых операторов, которые отсутствуют в интерактивном SQL.

Операторы манипулирования данными не требуют изменения для их встраивания в программный SQL. Однако оператор поиска (SELECT) потребовал изменений.

Стандартный оператор SELECT возвращает набор данных, релевантный сформированным условиям запроса. В интерактивном SQL этот полученный набор данных просто выводится на консоль пользователя и он может просмотреть полученные результаты. Встроенный оператор SELECT должен создавать структуры

данных, которые согласуются с базовыми языками программирования. Во встроенном SQL запросы делятся на 2 типа:

- Однострочные запросы, где ожидаемые результаты соответствуют одной строке данных. Эта строка может содержать значения нескольких столбцов.
- Многострочные запросы, результатом которых является получение целого набора строк. При этом приложение должно иметь возможность проработать все полученные строки. Значит, должен существовать механизм, который поддерживает просмотр и обработку полученного набора строк.

Первый тип запроса — однострочный запрос во встроенном SQL вызвал модификацию оператора SQL, которая выглядит следующим образом:

```
SELECT [{ALL | DISTINCT}] <список возвращаемых столбцов>
   INTO <список переменных базового языка>
   FROM <список исходных таблиц>
   [WHERE <условия соединения и поиска>]
```

Мы видим, что во встроенный SELECT добавился новый для нас раздел, содержащий список переменных базового языка. Именно в эти переменные будет помещен результат однострочного запроса, поэтому список переменных базового языка должен быть согласован как по порядку, так и по типу и размеру данных со списком возвращаемых столбцов. По правилам любого языка программирования все базовые переменные предварительно описаны в прикладной программе. Например, если в нашей БД «Библиотека» существует таблица READERS (Читатели), мы можем получить сведения о конкретном читателе.

```
CREATE TABLE READERS
(
   READER_ID Smallint(4) PRIMARY KEY,
   FIRST_NAME char(30) NOT NULL,
   LAST_NAME char(30) NOT NULL,
   ADRES char(50) ,
   HOME_PHON char(12) ,
   WORK_PHON char(12) ,
   BIRTH_DAY date CHECK( DateDiff(year, GetDate(),BIRTH_DAY) >=17 )
);
```

Для этого опишем базовые переменные. Рассмотрим пример для MS SQL SERVER 7.0, используя язык Transact SQL. При описании локальных переменных в языке Transact SQL используется специальный символ @. Комментарии в Transact SQL заключены в парные символы /* комментарий */.

```
DECLARE @READER_ID int
DECLARE @FIRS_NAME Char(30), @LAST_NAME Char(30), @ADRES Char(50)
DECLARE @HOME_PHON Char(12),@WORK_PHON Char(12)
/* зададим уникальный номер читательского билета */
```

```
SET @READER_ID = 4
/* теперь выполним запрос и поместим полученные сведения в определенные
   ранее переменные */
SELECT READERS.FIRST_NAME, READERS.LAST_NAME, READERS.ADRES,
   READERS.HOME_PHON, READERS.WORK_PHON
INTO @FIRS_NAME, @LAST_NAME, @ADRES, @HOME_PHON,@WORK_PHON
FROM READERS
WHERE READERS.READER_ID = @READER_ID
```

В этом простом примере мы имена переменных сделали такими же, как и имена столбцов таблицы READERS, но это необязательно. Однако транслятор различает эти объекты, именно поэтому в диалекте Transact SQL принято локальные переменные предварять специальным символом @. В примере мы использовали квалифицированные имена полей, имена полей, предваряемые именем таблицы. В нашем случае это тоже необязательно, потому что запрос выбирает данные только из одной таблицы.

В нашем примере базовые переменные играют разную роль. Локальная переменная @READER_ID является входной по отношению к запросу. Ей присвоено значение 4, и в запросе это значение используется для фильтрации данных, поэтому эта переменная используется в условии WHERE.

Остальные базовые переменные играют роль выходных переменных, в них СУБД помещает результат выполнения запроса, помещая в них значения соответствующих полей отношения READERS, извлеченные из БД.

Операторы, связанные с многострочными запросами

Рассмотрим более сложные многострочные запросы.

Для реализации многострочных запросов вводится новое понятие — понятие курсора или указателя набора записей. Для работы с курсором добавляется несколько новых операторов SQL:

1. Оператор DECLARE CURSOR — определяет выполняемый запрос, задаст имя курсора и связывает результаты запроса с заданным курсором. Этот оператор не является исполняемым для запроса, он только определяет структуру будущего множества записей и связывает ее с уникальным именем курсора. Этот оператор подобен операторам описания данных в языках программирования.
2. Оператор OPEN дает команду СУБД выполнить описанный запрос, создать виртуальный набор строк, который соответствует заданному запросу. Оператор OPEN устанавливает указатель записей (курсор) перед первой строкой виртуального набора строк результата.
3. Оператор FETCH продвигает указатель записей на следующую позицию в виртуальном наборе записей. В большинстве коммерческих СУБД оператор перемещения FETCH реализует более широкие функции перемещения, он позволяет перемещать указатель на произвольную запись, вперед и назад, допускает как абсолютную адресацию, так и относительную адресацию, позволяет установить курсор на первую или последнюю запись виртуального набора.
4. Оператор CLOSE закрывает курсор и прекращает доступ к виртуальному набору записей. Он фактически ликвидирует связь между курсором и результатом выполнения базового запроса. Однако в коммерческих СУБД оператор CLOSE не всегда означает уничтожение виртуального набора записей. Мы коснемся этого далее, когда будем рассматривать работу с курсором в MS SQL Server 7.0.

Оператор определения курсора

Стандарт определяет следующий синтаксис оператора определения курсора:

```
DECLARE <имя_курсора> CURSOR FOR <спецификация_курсора>
<спецификация курсора>::= <выражение_запроса SELECT>
```

Имя курсора — это допустимый идентификатор в базовом языке программирования.

В объявлении курсора могут быть использованы базовые переменные. Однако необходимо помнить, что на момент выполнения оператора OPEN значения всех базовых переменных, используемых в качестве входных переменных, связанных с условиями фильтрации значений в базовом запросе, должны быть уже заданы.

Определим курсор, который содержит список всех должников нашей библиотеки. Должниками назовем читателей, которые имеют на руках хотя бы одну книгу, срок сдачи которой уже прошел.

```
DECLARE Debtor_reader_cursor CURSOR FOR
SELECT READERS.FIRST_NAME, READERS.LAST_NAME, READERS.ADRES,
   READERS.HOME_PHON, READERS.WORK_PHON, BOOKS.TITLE
FROM READERS,BOOKS.EXEMPLAR
WHERE READERS.READER_ID = EXEMPLAR.READER_ID AND
   BOOKS.ISBN = EXEMPLARE.ISBN AND
   EXEMPLAR.DATA_OUT > Getdate()
ORDER BY READERS.FIRST_NAME
```

При определении курсора мы снова использовали функцию Transact SQL Getdate(), которая возвращает значение текущей даты. Таким образом, определенный курсор будет создавать набор строк, содержащих перечень должников, с указанием названий книг, которые они не вернули вовремя в библиотеку.

В соответствии со стандартом SQL2 Transact SQL содержит расширенное определение курсора

```
DECLARE <имя_курсора> [INSENSITIVE] [SCROLL] CURSOR
FOR <оператор выбора SELECT>
[FOR {READ ONLY | UPDATE [OF <имя столбца 1> [,...n]]}]
```

Параметр INSENSITIVE (нечувствительный) определяет режим создания набора строк, соответствующего определяемому курсору, при котором все изменения в исходных таблицах, произведенные после открытия курсора другими пользователями, не видны в нем. Такой набор данных нечувствителен ко всем изменениям, которые могут проводиться другими пользователями в исходных таблицах, этот тип курсора соответствует некоторому мгновенному слепку с БД.

СУБД более быстро и экономно может обрабатывать такой курсор, поэтому если для вас действительно важно рассмотреть и обработать состояние БД на некоторый конкретный момент времени, то имеет смысл создать «нечувствительный курсор».

Ключевое слово SCROLL определяет, что допустимы любые режимы перемещения по курсору (FIRST, LAST, PRIOR, NEXT, RELATIVE, ABSOLUTE) в операторе FETCH.

Если не указано ключевое слово SCROLL, то считается доступной только стандартное перемещение вперед: спецификация NEXT в операторе FETCH.

Если указана спецификация READ ONLY (только для чтения), то изменения и обновления исходных таблиц не будут выполняться с использованием данного курсора. Курсор с данной спецификацией может быть самым быстрым в обработке, однако если вы не укажите специально спецификацию READ ONLY, то СУБД будет считать, что вы допускаете операции модификации с базовыми таблицами, и в этом случае для обеспечения целостности БД СУБД будет гораздо медленнее обрабатывать ваши операции с курсором.

При использовании параметра UPDATE [OF <имя столбца 1> [,...<имя столбца n>]] мы задаем перечень столбцов, в которых допустимы изменения в процессе нашей работы с курсором. Такое ограничение упростит и ускорит работу СУБД. Если этот параметр не указан, то предполагается, что допустимы изменения всех столбцов курсора.

Вернемся к нашему примеру. Если мы преследуем цель мгновенного слепка БД, дающего сведения о должниках, то применим все параметры, позволяющие ускорить работу с нашим курсором. Тогда оператор описания курсора будет выглядеть следующим образом:

```
DECLARE Debtor_reader_cursor INSENSITIVE CURSOR
FOR
SELECT READERS.FIRST_NAME, READERS.LAST_NAME, READERS.ADRES,
   READERS.HOME_PHON, READERS.WORK_PHON, BOOKS.TITLE
FROM READERS,BOOKS,EXEMPLAR
WHERE READERS.READER_ID = EXEMPLAR.READER_ID AND
   BOOKS.ISBN = EXEMPLARE.ISBN AND
   EXEMPLAR.DATA_OUT > Getdate()
ORDER BY READERS.FIRST_NAME
FOR READ ONLY
```

При описании курсора нет ограничений на вид оператора SELECT, который используется для создания базового набора строк, связанного с курсором. В операторе SELECT могут использоваться группировки и встроенные подзапросы и вычисляемые поля.

Оператор открытия курсора

Оператор открытия курсора имеет следующий синтаксис:

```
OPEN <имя_курсора> [USING <список базовых переменных>]
```

Именно оператор открытия курсора инициирует выполнение базового запроса, соответствующего описанию курсора, заданному в операторе DECLARE ... CURSOR. При выполнении оператора OPEN СУБД производит семантическую проверку курсора, то есть выполняет этапы со 2 по 5 в алгоритме выполнения запросов (рис. 12.1), поэтому именно здесь СУБД возвращает коды ошибок прикладной программе, сообщающие ей о результатах выполнения базового запроса. Ошибки могут возникнуть в результате неправильного задания имен полей или имен исходных таблиц или при попытке извлечь данные из таблиц, к которым данный пользователь не имеет доступа.

По стандарту СУБД возвращает код завершения операции в специальной системной переменной SQLCODE. В прикладной программе пользователь может анализировать эту переменную, что необходимо делать после выполнения каждого оператора SQL. При неудачном выполнении операции открытия курсора СУБД возвращает отрицательное значение SQLCODE.

В случае удачного завершения выполнения оператора открытия курсора набор данных, сформированный в результате базового запроса, остается доступным пользователю до момента выполнения оператора закрытия курсора.

Однако надо помнить, что СУБД автоматически закрывает все курсоры в случае завершения транзакции (COMMIT) или отката транзакции (ROLLBACK). После того как курсор закрыт его можно открыть снова, но при этом соответствующий запрос выполнится заново. Поэтому допустимо, что содержимое первого курсора будет не соответствовать его содержимому при повторном открытии, потому что за это время изменилось состояние БД.

Оператор чтения очередной строки курсора

После открытия указатель текущей строки установлен перед первой строкой курсора. Стандартно оператор FETCH перемещает указатель текущей строки на следующую строку и присваивает базовым переменным значение столбцов, соответствующее текущей строке.

Простой оператор FETCH имеет следующий синтаксис:

```
FETCH <имя_курсора> INTO <список переменных базового языка >
```

Оператор извлечения очередной строки из курсора будет выглядеть следующим образом:

```
FETCH Debtor_reader_cursor into @FIRST_NAME, @LAST_NAME, @ADRES,  @HOME_PHON,
   @WORK_PHON, @TITLE
```

Расширенный оператор FETCH имеет следующий синтаксис:

```
FETCH
[NEXT | PRIOR | FIRST | LAST
   | ABSOLUTE {n | <имя_переменной>}
   |RELATIVE{n|<имя_переменной>}]
FROM
   <имя_курсора> INTO <список базовых переменных>
```

Здесь параметр NEXT задает выбор следующей строки после текущей из базового набора строк, связанного с курсором. Параметр PRIOR задает перемещение на предыдущую строку по отношению к текущей. Параметр FIRST задает перемещение на первую строку набора, а параметр LAST задает перемещение на последнюю строку набора.

Кроме того, в расширенном операторе перемещения допустимо переместиться сразу на заданную строку, при этом допустима как абсолютная адресация, заданием параметра ABSOLUTE, так и относительная адресация, заданием параметра RELATIVE. При относительной адресации положительное число сдвигает указатель вниз от текущей записи, отрицательное число сдвигает вверх от текущей записи.

Однако для применения расширенного оператора FETCH в соответствии со стандартом SQL2 описание курсора обязательно должно содержать ключевое слово SCROLL. Иногда такие курсоры называют в литературе прокручиваемыми курсорами. В стандарт эти курсоры вошли сравнительно недавно, поэтому в коммерческих СУБД очень часто операторы по работе с подобными курсорами серьезно отличаются. Правда, реалии сегодняшнего дня заставляют поставщиков коммерческих СУБД более строго соблюдать последний стандарт SQL. В технической документации можно встретить две версии синтаксиса оператора FETCH: одну, которая соответствует стандарту, и другую, которая расширяет стандарт дополнительными возможностями, предоставляемыми только данной СУБД для работы с курсором.

Если вы предполагаете, что ваша БД может быть перенесена на другую платформу, а это надо всегда предусматривать, то лучше пользоваться стандартными возможностями. В этом случае ваше приложение будет более платформенно-независимым и легче будет его перенести на другую СУБД.

Оператор закрытия курсора

Оператор закрытия курсора имеет простой синтаксис, он выглядит следующим образом:

```
CLOSE <имя_курсора>
```

Оператор закрытия курсора закрывает временную таблицу, созданную оператором открытия курсора, и прекращает доступ прикладной программы к этому объекту. Единственным параметром оператора закрытия является имя курсора.

Оператор закрытия может быть выполнен в любой момент после оператора открытия курсора.

В некоторых коммерческих СУБД кроме оператора закрытия курсора используется еще оператор деактивации (уничтожения) курсора. Например, в MS SQL Server 7.0 наряду с оператором закрытия курсора используется оператор

```
DEALLOCATE <имя_курсора>
```

Здесь оператор закрытия курсора не уничтожает набор данных, связанный с курсором, он только закрывает к нему доступ и освобождает все блокировки, которые ранее были связаны с данным курсором.

При выполнении оператора DEALLOCATE SQL Server освобождает разделяемую память, используемую командой описания курсора DECLARE. После выполнения этой команды невозможно выполнение команды OPEN для данного курсора.

Удаление и обновление данных с использованием курсора

Курсоры в прикладных программах часто используются для последовательного просмотра данных. Если курсор не связан с операцией группировки, то фактически каждая строка курсора соответствует строго только одной строке исходной таблицы, и в этом случае курсор удобно использовать для оперативной корректировки данных. В стандарте определены операции модификации данных, связанные с курсором. Операция удаления строки, связанной с текущим указателем курсора, имеет следующий синтаксис:

```
DELETE FROM <имя_таблицы> WHERE CURRENT OF <имя курсора>
```

Если указанный в операторе курсор открыт и установлен на некоторую строку, и курсор определяет изменяемую таблицу, то текущая строка курсора удаляется, а он позиционируется перед следующей строкой. Таблица, указанная в разделе FROM оператора DELETE, должна быть таблицей, указанной в самом внешнем разделе FROM спецификации курсора.

Если нам необходимо прочитать следующую строку курсора, то надо снова выполнить оператор FETCH NEXT.

Аналогично курсор может быть использован для модификации данных. Синтаксис операции позиционной модификации следующий:

```
UPDATE <имя_таблицы> SET <имя_столбца1>= {<значение> | NULL}
[{,<имя_столбца_N>= {<значение> | NULL}}...]
WHERE CURRENT OF <имя_курсора>
```

Одним оператором позиционного обновления могут быть заменены несколько значений столбцов строки таблицы, соответствующей текущей позиции курсора. После выполнения операции модификации позиция курсора не изменяется.

Для того чтобы можно было применять позиционные операторы удаления (DELETE) и модификации (UPDATE), курсор должен удовлетворять определенным требованиям. Согласно стандарту SQL1, это следующие требования:

- Запрос, связанный с курсором, должен считывать данные из одной исходной таблицы, то есть в предложении FROM запроса SELECT, связанного с определением курсора (DECLARE CURSOR), должна быть задана только одна таблица.
- В запросе не может присутствовать параметр упорядочения ORDER BY. Для того чтобы сохранялось взаимно однозначное соответствие строк курсора и исходной таблицы, курсор не должен идентифицировать упорядоченный набор данных.
- В запросе не должно присутствовать ключевое слово DISTINCT.
- Запрос не должен содержать операций группировки, то есть в нем не должно присутствовать предложение GROUP BY или HAVING.
- Пользователь, который хочет применить операции позиционного удаления или обновления, должен иметь соответствующие права на выполнение данных операций над базовой таблицей. (О правах и привилегиях пользователя мы поговорим в главе 13.)

Использование курсора для операций обновления значительно усложняет работу с подобным курсором со стороны СУБД, поэтому операции, связанные с позиционной модификацией, выполняются гораздо медленнее, чем операции с курсорами, которые используются только для чтения. Именно поэтому рекомендуется обязательно указывать в операторе определения курсора предложение READ ONLY, если вы не собираетесь использовать данный курсор для операций модификации. По умолчанию, если нет дополнительных указаний, СУБД создает курсор с возможностью модификации.

Курсоры — удобное средство для формирования бизнес-логики приложений, но следует помнить, что если вы открываете курсор с возможностью модификации, то СУБД блокирует все строки базовой таблицы, вошедшие в ваш курсор, и тем самым блокируется работа других пользователей с данной таблицей.

Чтобы свести к минимуму количество требуемых блокировок, при работе интерактивных программ следует придерживаться следующих правил:

- Необходимо делать транзакции как можно короче.
- Необходимо выполнять оператор завершения COMMIT после каждого запроса и как можно скорее после изменений, сделанных программой.
- Необходимо избегать программ, в которых осуществляется интенсивное взаимодействие с пользователем или осуществляется просмотр очень большого количества строк данных.
- Если возможно, то лучше не применять прокручиваемые курсоры (SCROLL), потому что они требуют блокирования всех строк выборки, связанных с открытым курсором.
- Использование простого последовательного курсора позволит системе разблокировать текущую строку, как только будет выполнена операция FETCH, что минимизирует блокировки других пользователей, работающих параллельно с вами и использующих те же таблицы.
- Если возможно, определяйте курсор как READ ONLY.

Однако когда мы рассматривали модели «клиент—сервер», применяемые в БД, то определили, что в развитых моделях серверов баз данных большая часть бизнес-логики клиентского приложения выполняется именно на сервере, а не на клиенте. Для этого используются специальные объекты, которые называются хранимыми процедурами и хранятся в БД, как таблицы и другие базовые объекты.

В связи с этим фактом курсоры, которые могут быть использованы в приложениях, обычно делятся на курсоры сервера и курсоры клиента. Курсор сервера создается и выполняется на сервере, данные, связанные с ним, не пересылаются на компьютер клиента. Курсоры сервера определяются обычно в хранимых процедурах или триггерах.

Курсоры клиента — это те курсоры, которые определяются в прикладных программах, выполняемых на клиенте. Набор строк, связанный с данным курсором, пересылается на клиент и там обрабатывается. Если с курсором связан большой набор данных, то операция пересылки набора строк, связанных с курсором, может занять значительное время и значительные ресурсы сети и клиентского компьютера.

Конечно, курсоры сервера более экономичны и выполняются быстрее. Поэтому последней рекомендацией, связанной с использованием курсоров, будет рекомендация трансформировать логику работы вашего приложения, чтобы как можно чаще вместо курсоров клиента использовать курсоры сервера.

Хранимые процедуры

С точки зрения приложений, работающих с БД, хранимые процедуры (Stored Procedure) — это подпрограммы, которые выполняются на сервере. По отношению к БД — это объекты, которые создаются и хранятся в БД. Они могут быть вызваны из клиентских приложений. При этом одна процедура может быть использована в любом количестве клиентских приложений, что позволяет существенно сэкономить трудозатраты на создание прикладного программного обеспечения и эффективно применять стратегию повторного использования кода. Так же как и любые процедуры в стандартных языках программирования, хранимые процедуры могут иметь входные и выходные параметры или не иметь их вовсе.

Хранимые процедуры могут быть активизированы не только пользовательскими приложениями, но и триггерами.

Хранимые процедуры пишутся на специальном встроенном языке программирования, они могут включать любые операторы SQL, а также включают некоторый набор операторов, управляющих ходом выполнения программ, которые во многом схожи с подобными операторами процедурно ориентированных языков программирования. В коммерческих СУБД для написания текстов хранимых процедур используются собственные языки программирования, так, в СУБД Oracle для этого используется язык PL /SQL, а в MS SQL Server и System11 фирмы Sybase используется язык Transact SQL. В последних версиях Oracle объявлено использование языка Java для написания хранимых процедур.

Хранимые процедуры являются объектами БД. Каждая хранимая процедура компилируется при первом выполнении, в процессе компиляции строится оптимальный план выполнения процедуры. Описание процедуры совместно с планом ее выполнения хранится в системных таблицах БД.

Для создания хранимой процедуры применяется оператор SQL CREATE PROCEDURE.

По умолчанию выполнить хранимую процедуру может только ее владелец, которым является владелец БД, и создатель хранимой процедуры. Однако владелец хранимой процедуры может делегировать права на ее запуск другим пользователям.

Имя хранимой процедуры является идентификатором в языке программирования, на котором она пишется, и должно удовлетворять всем требованиям, которые предъявляются к идентификаторам в данном языке.

В MS SQL Server хранимая процедура создается оператором:

```
CREATE PROC[EDURE] <имя_процедуры> [:<версия>]
[{@параметр1 тип_данных}
[VARYING] [= <значение_по_умолчанию>] [OUTPUT]]
[,.параметрN…]
[ WITH
   { RECOMPILE
   | ENCRYPTION
   | RECOMPILE, ENCRYPTION}]
[FOR REPLICATION]
AS
Тело процедуры
```

Здесь необязательное ключевое слово VARYING определяет заданное значение по умолчанию для определенного ранее параметра.

Ключевое слово RECOMPILE определяет режим компиляции создаваемой хранимой процедуры. Если задано ключевое слово RECOMPILE, то процедура будет перекомпилироваться каждый раз, когда она будет вызываться на исполнение. Это может резко замедлить исполнение процедуры. Но, с другой стороны, если данные, обрабатываемые данной хранимой процедурой, настолько динамичны, что предыдущий план исполнения, составленный при ее первом вызове, может быть абсолютно неэффективен при последующих вызовах, то стоит применять данный параметр при создании этой процедуры.

Ключевое слово ENCRYPTION определяет режим, при котором исходный текст хранимой процедуры не сохраняется в БД. Такой режим применяется для того, чтобы сохранить авторское право на интеллектуальную продукцию, которой и являются хранимые процедуры. Часто такой режим применяется, когда вы ставите готовую базу заказчику и не хотите, чтобы исходные тексты разработанных вами хранимых процедур были бы доступны администратору БД, работающему у заказчика. Однако надо помнить, что если вы захотите отредактировать текст хранимой процедуры сами, то вы его не сможете извлечь из БД тоже, его

надо будет хранить отдельно в некотором текстовом файле. И это не самое плохое, но вот в случае восстановления БД после серьезной аварии для перекомпиляции потребуются первоначальные исходные тексты всех хранимых процедур. Поэтому защита вещь хорошая, но она усложняет сопровождение и модификацию хранимых процедур.

Однако кроме имени хранимой процедуры все остальные параметры являются необязательными. Процедуры могут быть процедурами или процедурами-функциями. И эти понятия здесь трактуются традиционно, как в языках программирования высокого уровня. Хранимая процедура-функция возвращает значение, которое присваивается переменной, определяющей имя процедуры. Процедура в явном виде не возвращает значение, но в ней может быть использовано ключевое слово OUTPUT, которое определяет, что данный параметр является выходным.

Рассмотрим несколько примеров простейших хранимых процедур.

```
/* процедура проверки наличия экземпляров данной книги
   параметры:
   @ISBN шифр книги
   процедура возвращает параметр, равный количеству экземпляров
   Если возвращается ноль, то это значит, что нет свободных экземпляров данной
   книги в библиотеке.
*/
CREATE PROCEDURE COUNT_EX (@ISBN varchar(12))
AS
/* определим внутреннюю переменную */
DECLARE @TEK_COUNT int
/* выполним соответствующий оператор SELECT
Будем считать только экземпляры, которые в настоящий момент находятся
не на руках у читателей, а в библиотеке */
select @TEK_COUNT = select count(*) FROM EXEMPLAR WHERE ISBN = @ISBN
   AND READER_ID Is NULL AND EXIST = True
/* 0 - ноль означает, что нет ни одного свободного экземпляра данной книги
в библиотеке */
RETURN @TEK_COUNT
```

Хранимая процедура может быть вызвана несколькими способами. Простейший способ — это использование оператора:

```
EXEC <имя процедуры> <значение_входного_параметра1>…
<имя_переменной_для_выходного параметра1>…
```

При этом все входные и выходные параметры должны быть заданы обязательно и в том порядке, в котором они определены в процедуре.

Например, если мне надо найти число экземпляров книги «Oracle8. Энциклопедия пользователя», которая имеет ISBN 966-7393-08-09, то текст вызова ранее созданной хранимой процедуры может быть следующим:

```
/*определили две переменные
@Ntek - количество экземпляров данной книги в наличие в библиотеке
@ISBN - международный шифр книги */
declare @Ntek int
DECLARE @ISBN VARCHAR(14)
/* Присвоим значение переменной @ISBN */
Select @ISBN = '966-7393-08-09'
/* Присвоим переменной @Ntek результаты выполнения хранимой процедуры
COUNT_EX */
EXEC @Ntek = COUNT_EX @ISBN
```

Если у вас определено несколько версий хранимой процедуры, то при вызове вы можете указать номер конкретной версии для исполнения. Так, например, в версии 2 процедуры COUNT_EX последний оператор исполнения этой процедуры имеет вид:

```
EXEC @Ntek = COUNT_EX;2 @ISBN
```

Однако если в процедуре определены значения входных параметров по умолчанию, то при запуске процедуры могут быть указаны значения не всех параметров. В этом случае оператор вызова процедуры может быть записан в следующем виде:

```
EXEC <имя процедуры> <имя_параметра1>=<значение параметра1>…
   <имя_параметраN>=<значение параметраN>…
```

Например, создадим процедуру, которая считает количество книг, изданных конкретным издательством в конкретном году. При создании процедуры зададим для года издания по умолчанию значение текущего года.

```
CREATE PROCEDURE COUNT_BOOKS (@YEARIZD int = Year(GetDate()),
   @PUBLICH varchar(20))
/* процедура подсчета количества книг конкретного издательства, изданных
   в конкретом году
параметры:
   @YEARIZD int год издания
   @PUBLICH название издательства
*/
AS
DECLARE @TEK_Count int
Select @TEK_count = Select COUNT(ISBN)
```



```
From BOOKS
Where YEARIZD = @YEARIZD AND PUBLICH =@PUBLICH
/* одновременно с исполнением оператора Select мы присваиваем результаты его
работы определенной ранее переменной @TEK_Count */
/* при формировании результата работы нашей процедуры мы должны учесть,
что в нашей библиотеке, возможно, нет ни одной книги некоторого издательства
для заданного года. Результат выполнения запроса SELECT в этом случае будет
иметь неопределенное значение, но анализировать все-таки лучше числовые
значения. Поэтому в качестве возвращаемого значения мы используем результаты
работы специальной встроенной функции Transact SQL COALESCE (n1,n2,…,nm),
которая возвращает первое конкретное, то есть не равное NULL, значение из
списка значений n1,n2,…,nm. */
Return COALESCE (@TEK_Count,0)
```

Теперь вызовем эту процедуру, для этого подготовим переменную, куда можно поместить результаты выполнения процедуры.

```
declare @N int
Exec @N = COUNT_BOOKS @PUBLICH = 'Питер'
```

В переменной @N мы получим количество книг в нашей библиотеке, изданных издательством «Питер» в текущем году. Мы можем обратиться к этой процедуре и задав все параметры:

```
Exec @N = COUNT_BOOKS @PUBLICH = 'BHW', @YEARIZD = 1999
```

Тогда получим количество книг, изданных издательством «BHW» в 1999 году и присутствующих в нашей библиотеке.

Если мы задаем параметры по именам, то нам необязательно задавать их в том порядке, в котором они описаны при создании процедуры.

Каждая хранимая процедура является объектом БД. Она имеет уникальное имя и уникальный внутренний номер в системном каталоге. При изменении текста хранимой процедуры мы должны сначала уничтожить данную процедуру как объект, хранимый в БД, и только после этого записать на ее место новую. Следует отметить, что при удалении хранимой процедуры удаляются одновременно все ее версии, нельзя удалить только одну версию хранимой процедуры.

Для того чтобы автоматизировать процесс уничтожения старой процедуры и замены ее на новую, в начале текста хранимой процедуры можно выполнить проверку наличия объекта типа «хранимая процедура» с данным именем в системном каталоге и при наличии описания данного объекта удалить его из системного каталога. В этом случае текст хранимой процедуры предваряется специальным оператором проверки и может иметь, например, следующий вид:

```
/* проверка существования в системном каталоге объекта с данным именем
   и типом, созданного владельцем БД */
if exists (select * from sysobjects where id = object_id('dbo.NEW_BOOKS') and
sysstat & 0xf = 4)
```

```
/* если объект существует, то сначала его удалим из системного каталога */
   drop procedure dbo.NEW_BOOKS
GO

CREATE PROCEDURE NEW_BOOKS (@ISBN varchar(12),@TITL varchar(255),@AUTOR
varchar(30),@COAUTOR varchar(30),@YEARIZD int,@PAGES INT,@NUM_EXEMPL INT)
/* процедура ввода новой книги с указанием количества экземпляров данной книги
параметры
   @ISBN        varchar(12)    шифр книги
   @TITL        varchar(255)   название
   @AUTOR       varchar(30)    автор
   @COAUTOR     varchar(30)    соавтор
   @YEARIZD     int            год издания
   @PAGES       INT            количество страниц
   @NUM_EXEMPL  INT            количество экземпляров
*/
AS
/*опишем переменную, в которой будет храниться количество оставшихся
не оприходованных экземпляров книги, т.е. таких, которым еще не заданы
инвентарные номера */
DECLARE @TEK int
/ *вводим данные о книге в таблицу BOOKS */
INSERT INTO BOOKS VALUES(@ISBN,@TITL,@AUTOR,@COAUTOR,@YEARIZD,@PAGES)
/* назначение значения текущего счетчика осташихся к вводу экземпляров*/
SELECT @TEK = @NUM_EXEMPL
/* организуем цикл для ввода новых экземпляров данной книги */
WHILE @TEK>0 /* пока количество оставшихся экземпляров больше нуля */
BEGIN
/* так как для инвентарного номера экземпляра книги мы задали свойство
IDENTITY, то нам не надо вводить инвентарный номер, СУБД сама автоматически
вычислит его, добавив единицу к предыдущему, введет при выполнении оператора
ввода INSERT.
Поле, определяющее присутствие экземпляра в библиотеке (EXIST) - логическое
поле, мы введем туда значение TRUE,которое соответствует присутствию
экземпляра книги в библиотеке.
Даты взятия и возврата мы можем не заполнять, тогда по умолчанию СУБД
подставит туда значение, соответствующее 1 января 1900 года, если мы не хотим
хранить такие бессмысленные данные, то можем ввести для обоих полей дата
время, значения текущей даты. */
```



```
insert into EXEMPLAR (ISBN,DATA_IN,DATA_OUT,EXIST)
VALUES (@ISBN,GetDate(),GetDate()),TRUE)
/* изменение текущего значения счетчика количества оставшихся экземпляров */
   SELECT @TEK = @TEK - 1
End /* конец цикла ввода данных о экземпляре книги*/
GO
```

Если мы не использовали инкрементное поле в качестве инвентарного номера экземпляра, то мы могли бы сами назначать инвентарный номер, увеличивая на единицу номер последнего хранимого в библиотеке экземпляра книги. Можно было бы попробовать просто сосчитать количество существующих экземпляров в библиотеке, но мы могли удалить некоторые, и тогда номер нового экземпляра может быть уже использован, и мы не сможем ввести данные, система не позволит нам нарушить уникальность первичного ключа.

Текст процедуры в этом случае будет иметь вид:

```
/* проверка существования в системном каталоге объекта с данным именем
и типом, созданного владельцем БД */
if exists (select * from sysobjects where id = object_id('dbo.NEW_BOOKS') and
sysstat & 0xf = 4)
/* если объект существует, то сначала его удалим из системного каталога */
   drop procedure dbo.NEW_BOOKS
CREATE PROCEDURE NEW_BOOKS (@ISBN varchar(12),@TITL varchar(255),@AUTOR
varchar(30),@COAUTOR varchar(30),@YEARIZD int,@PAGES INT,@NUM_EXEMPL INT)
/* процедура ввода новой книги с указанием количества экземпляров данной книги
параметры
   @ISBN        varchar(12)    шифр книги
   @TITL        varchar(255)   название
   @AUTOR       varchar(30)    автор
   @COAUTOR     varchar(30)    соавтор
   @YEARIZD     int            год издания
   @PAGES       INT            количество страниц
   @NUM_EXEMPL  INT            количество экземпляров
*/
AS
DECLARE @TEK int
declare @INV int
INSERT INTO BOOKS VALUES(@ISBN,@TITL,@AUTOR,@COAUTOR,@YEARIZD,@PAGES)
/* назначение значения текущего счетчика осташихся к вводу экземпляров*/
SELECT @TEK = @NUM_EXEMPL
/* определение максимального значения инвентарного номера в библиотеке */
```

```
SELECT @INV = SELECT MAX( ID_EXEMPLAR) FROM EXEMPLAR
/* организуем цикл для ввода новых экземпляров данной книги */
WHILE @TEK>0 /* пока количество оставшихся экземпляров больше нуля */
BEGIN
insert into EXEMPLAR (ID_EXEMPLAR,ISBN,DATA_IN,DATA_OUT,EXIST)
VALUES (@INV,@ISBN,GETDATE(),GetDate(), TRUE)
/* изменение текущих значений счетчика и инвентарного номера */
   SELECT @TEK = @TEK - 1
   SELECT @INV = @INV + 1
End /* конец цикла ввода данных о экземпляре книги*/
GO
```

Хранимые процедуры могут вызывать одна другую. Создадим хранимую процедуру, которая возвращает номер читательского билета для конкретного читателя.

```
if exists (select * from sysobjects where id = object_id('dbo. CK_READER') and
sysstat & 0xf = 4)
/* если объект существует, то сначала его удалим из системного каталога */
   drop procedure dbo.CK_READER

/* Процедура возвращает номер читательского билета, если читатель есть и 0 в
противном случае. В качестве параметров передаем фамилию и дату рождения */

CREATE PROCEDURE CK_READER (@FIRST_NAME varchar(30),@BIRTH_DAY varchar(12))
AS
/*опишем переменную, в которой будет храниться номер читательского билета*/
DECLARE @NUM_READER INT
/* определение наличия читателя */
select @NUM_READER = select NUM_READER from READERS
   WHERE FIRST_NAME = @ FIRST_NAME AND
   AND convert(varchar(8),BIRTH_DAY,4)=@BIRTH_DAY
RETURN COALESCE(@NUM_READER,0)
```

Мы здесь использовали фукнцию преобразования типа данных dataTime в тип данных varchar(8). Это было необходимо сделать для согласования типов данных при выполнении операции сравнения. Действительно, входная переменная @BIRTH_DAY имеет символьный тип (varchar), а поле базы данных BIRTH_DAY имеет тип SmallDateTime.

Хранимые процедуры допускают наличие нескольких выходных параметров. Для этого каждый выходной параметр должен после задания своего типа данных иметь дополнительное ключевое слово OUTPUT. Рассмотрим пример хранимой процедуры с несколькими выходными параметрами.

Создадим процедуру ввода нового читателя, при этом внутри процедуры выполним проверку наличия в нашей картотеке данного читателя, чтобы не назначать ему новый номер читательского билета. При этом выходными параметрами процедуры будут номер читательского билета, признак того, был ли ранее записан читатель с данными характеристиками в нашей библиотеке, а если он был записан, то сколько книг за ним числится.

```
/* проверка наличия данной процедуры в нашей БД*/
if exists (select * from sysobjects where id =
object_id(N'[dbo].[NEW_READER]') and OBJECTPROPERTY(id, N'IsProcedure') = 1)
drop procedure [dbo].[NEW_READER]
GO
/* процедура проверки существования читателя с заданными значениями вводимых
параметров
Процедура возвращает новый номер читательского билета, если такого читателя не
было сообщает старый номер и количество книг, которое должен читатель в
противном случае */
CREATE PROCEDURE NEW_READER (@NAME_READER varchar(30),@ADRES
varchar(40),@HOOM_PHONE char(9),@WORK_PHONE char(9),
@BIRTH_DAY varchar(8),
   @NUM_READER int OUTPUT,
   /* выходной параметр, определяющий номер читательского билета*/
   @Y_N  int OUTPUT,
   /* выходной параметр, определяющий был ли читатель ранее записан
      в библиотеку*/
   @COUNT_BOOKS  int OUTPUT
   /* выходной параметр, определяющий количество книг, которое числится
      за читателем*/)
AS
/* переменная, в которой будет храниться номер читательского билета,
   если читатель уже был записан в библиотеку */
DECLARE @N_R int
/* определение наличия читателя */
EXEC @N_R = CK_READER @NAME_READER,@BIRTH_DAY
   IF @N_R= 0  Or @N_R Is Null
/* если читатель с заданными характеристиками не найден, т. е. переменной
   @N_R присвоено значение нуль или ее значение неопределено, перейдем
   к назначению для нового читателя нового номера читательского билета */
   BEGIN
/* так как мы номер читательского билета определили как инкрементное поле,
   то в операторе ввода мы его не указываем система сама назначит новому
   читателю очередной номер  */
```

```
    INSERT INTO READER(NAME_READER,ADRES,HOOM_PHONE,WORK_PHONE,BIRTH_DAY)
    VALUES (@NAME_READER,@ADRES,@HOOM_PHONE,@WORK_PHONE,Convert(smalldatetime,
       @BIRTH_DAY,4) )
/* в операторе INSERT мы должны преобразовать символьную переменную @BIRTH_DAY
   в тип данных smalldatetime, который определен для  поля дата рождения
   BIRTH_DAY. Это преобразование мы сделаем с помощью встроенной функции
   Transact SQL Convert */
/* теперь определим назначенный номер  читальского билета */
    select @NUM_READER = NUM_READER FROM READER
       WHERE NAME_READER = @NAME_READER
       AND convert(varchar(8),BIRTH_DAY,4)=@BIRTH_DAY
    /* здесь мы снова используем функцию преобразования типа, но в этом случае
       нам необходимо преобразовать поле BIRTH_DAY из типа  smalldatetime
       к типу varchar(8), в котором задан входной параметр @BIRTH_DAY */
       Select @Y_N =0
    /* присваиваем выходному параметру @Y_N значение 0 (ноль), что
       соответствует тому,что данный читатель ранее в нашей библиотеке
       не был записан */
       Select @COUNT_BOOKS = 0
    /* присваиваем выходному параметру, хранящему количество книг, числящихся
       за читателем значение ноль */
       Return 1
    END
else
/* если значение переменной @N_R не равно нулю, то читатель с заданными
   характеристиками был ранее записан в нашей библиотеке */
    BEGIN
/* определение количества книг у читателя  с найденным номером читательского
   билета */
    select @COUNT_BOOKS = COUNT(INV_NUMBER) FROM EXEMPLAR WHERE
       NUM_READER = @N_R
    select @Count_books = COALESCE( @COUNT_BOOKS,0)
/* присваиваем выходному параметру @COUNT_BOOKS значение, равное количеству
   книг, которые числятся за нашим читателем, если в предыдущем запросе
   @COUNT_BOOKS было присвоено неопределенное значение, то мы заменим его
   на ноль, используя для этого встроенную функцию COALESCE(@COUNT_BOOKS,0),
   которая возвращает первое определенное значение из списка значений,
   заданных в качестве ее параметров */
    Select @Y_N = 1
    /* присваиваем выходному параметру @Y_N значение 1, что соответствует тому,
       что данный читатель ранее в нашей библиотеке  был записан */
```



```
    Select @NUM_READER = @N_R
    /* присваиваем выходному параметру @NUM_READER определенный ранее номер
       читательского билета */
    return 0
end
```

Теперь посмотрим, как работает наша новая процедура, для этого в режиме интерактивного выполнения запросов (то есть в QueryAnalyzer MS SQL Server 7.0) запишем следующую последовательность команд:

```
-- пример использования выходных параметров при вызове процедуры
-- new reader
-- зададим необходимые нам переменные
Declare @K int, @N int, @B int
exec NEW_READER 'Пушкин В.В.','Лиговский 22-90',
'333-55-99','444-66-88','01.06.83',@NUM_READER =@K OUTPUT,
   @Y_N = @N OUTPUT,@COUNT_BOOKS = @B OUTPUT
-- теперь выведем результаты работы нашей процедуры используя ранее
-- определенные нами переменные
Select 'номер билета',@K,'да-нет',@N,'кол-во книг',@B
```

Мы получим результат:

Номер билета	да-нет	кол-во книг
18	0	0

Если же мы снова запустим нашу процедуру с теми же параметрами, то есть повторим выполнение подготовленных выше операторов, то получим уже иной ответ:

Номер билета	да-нет	кол-во книг
18	1	0

и это означает, что господин Пушкин В. В. уже записан в нашей библиотеке, но он не успел взять ни одной книги, поэтому за ним числится 0 (ноль) книг.

Теперь обратимся к оценке эффективности применения хранимых процедур.

Если рассмотреть этапы выполнения одинакового текста части приложения, содержащего SQL-операторы, самостоятельно на клиенте и в качестве хранимой процедуры, то можно отметить, что на клиенте выполняются все 5 этапов выполнения SQL-операторов, а хранимая процедура может храниться в БД в уже скомпилированном виде, и ее исполнение займет гораздо меньше времени (см. рис. 12.2).

Кроме того, хранимые процедуры, как уже упоминалось, могут быть использованы несколькими приложениями, а встроенные операторы SQL должны быть включены в каждое приложение повторно.

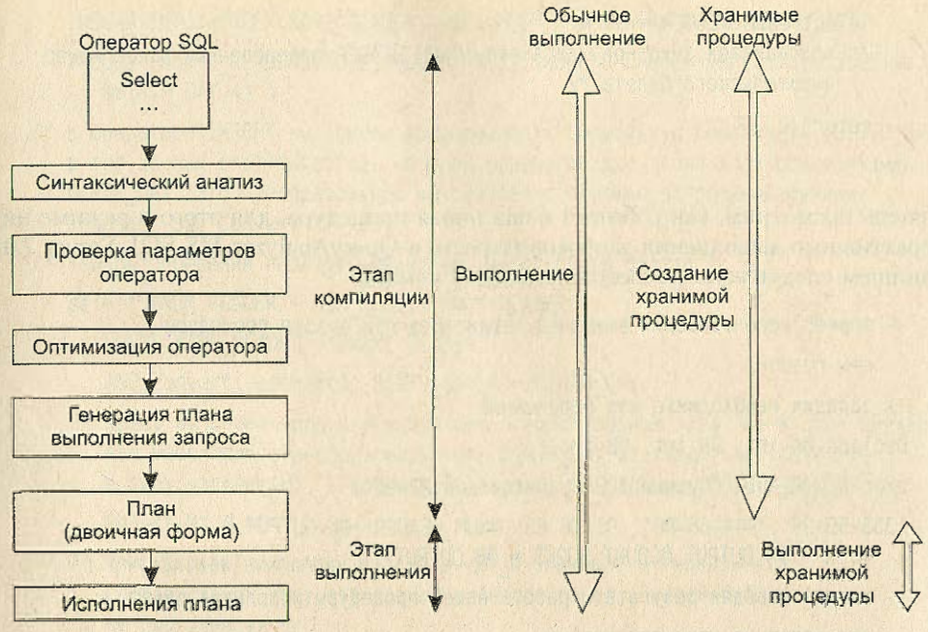


Рис. 12.2. Процесс выполнения операторов SQL на клиенте и процесс выполнения хранимой процедуры

Хранимые процедуры также играют ключевую роль в повышении быстродействия при работе в сети с архитектурой «клиент—сервер».

На рис. 12.3 показан пример выполнения последовательности операторов SQL на клиенте, а на рис. 12.4 показан пример выполнения той же последовательности операторов SQL, оформленных в виде хранимой процедуры. В этом случае клиент обращается к серверу только для выполнения команды запуска хранимой процедуры. Сама хранимая процедура выполняется на сервере. Объем пересылаемой по сети информации резко сокращается во втором случае.

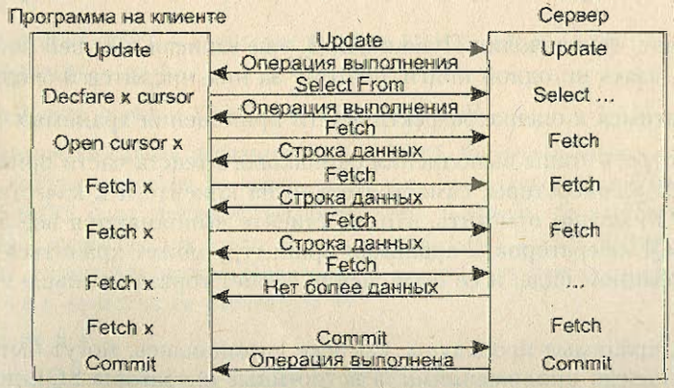


Рис. 12.3. Сетевой трафик при выполнении встроенных SQL-операторов

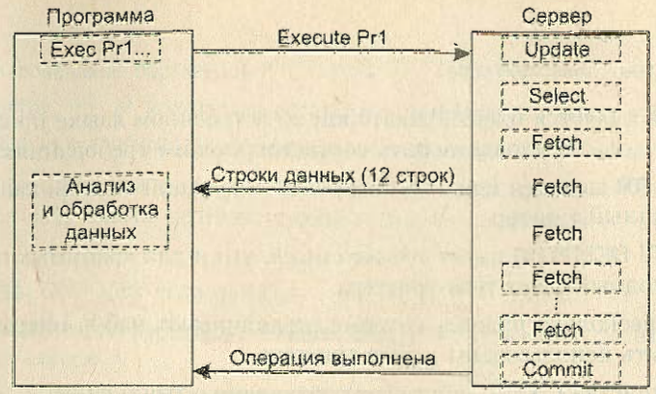


Рис. 12.4. Сетевой трафик при выполнении хранимой процедуры на сервере

Триггеры

Фактически триггер — это специальный вид хранимой процедуры, которую SQL Server вызывает при выполнении операций модификации соответствующих таблиц. Триггер автоматически активизируется при выполнении операции, с которой он связан. Триггеры связываются с одной или несколькими операциями модификации над одной таблицей.

В разных коммерческих СУБД рассматриваются разные триггеры. Так, в MS SQL Server триггеры определены только как постфильтры, то есть такие триггеры, которые выполняются после свершения события.

В СУБД Oracle определены два типа триггеров: триггеры, которые могут быть запущены перед реализацией операции модификации, они называются BEFORE-триггерами, и триггеры, которые активизируются после выполнения соответствующей модификации, аналогично триггерам MS SQL Server, — они называются AFTER-триггерами.

Триггеры могут быть эффективно использованы для поддержки семантической целостности БД, однако приоритет их ниже, чем приоритет правил-ограничений (constraints), задаваемых на уровне описания таблиц и на уровне связей между таблицами. При написании триггеров всегда надо помнить об этом, при нарушении правил целостности по связям (DRI declarative Referential Integrity) триггер просто может никогда не сработать.

В стандарте SQL1 ни хранимые процедуры, ни триггеры были не определены. Но в добавлении к стандарту SQL2, выпущенному в 1996 году, те и другие объекты были стандартизированы и определены.

Для создания триггеров используется специальная команда:

```
CREATE TRIGGER <имя_триггера>
ON <имя_таблицы>
FOR {[INSERT][, UPDATE] [, DELETE] }
[WITH ENCRIPTING]
```

```
AS
SQL-операторы (Тело триггера)
```

Имя триггера является идентификатором во встроенном языке программирования СУБД и должно удовлетворять соответствующим требованиям.

В параметре FOR задается одна или несколько операций модификации, которые запускают данный триггер.

Параметр WITH ENCRIPTING имеет тот же смысл, что и для хранимых процедур, он скрывает исходный текст тела триггера.

Существует несколько правил, которые ограничивают набор операторов, которые могут быть использованы в теле триггера.

Так, в большинстве СУБД действуют следующие ограничения:

- Нельзя использовать в теле триггера операции создания объектов БД (новой БД, новой таблицы, нового индекса, новой хранимой процедуры, нового триггера, новых индексов, новых представлений).
- Нельзя использовать в триггере команду удаления объектов DROP для всех типов базовых объектов БД.
- Нельзя использовать в теле триггера команды изменения базовых объектов ALTER TABLE, ALTER DATABASE.
- Нельзя изменять права доступа к объектам БД, то есть выполнять команду GRAND или REVOKE.
- Нельзя создать триггер для представления (VIEW) .
- В отличие от хранимых процедур, триггер не может возвращать никаких значений, он запускается автоматически сервером и не может связаться самостоятельно ни с одним клиентом.

Рассмотрим пример триггера, который срабатывает при удалении экземпляра некоторой книги, например, в случае утери этой книги читателем. Что же может делать этот триггер? А он может выполнять следующую проверку: проверять, остался ли еще хоть один экземпляр данной книги в библиотеке, и если это был последний экземпляр книги в библиотеке, то резонно удалить описание книги из предметного каталога, чтобы наши читатели зря не пытались заказать эту книгу.

Текст этого триггера на языке Transact SQL приведен ниже:

```
/* Проверка существования данного триггера в системном каталоге */
if exists (select * from sysobjects where id = object_id('dbo.DEL_EXEMP') and
sysstat & 0xf = 8)
   drop trigger dbo.DEL_EXEMP
GO

CREATE TRIGGER DEL_EXEMP ON dbo.EXEMPLAR /* мы создаем триггер  для таблицы
EXEMPLAR */
FOR DELETE /* только для операции удаления */
```



```
AS
/* опишем локальные переменные */
DECLARE @Ntek int /* количество оставшихся экземпляров  удаленной книги */
DECLARE @DEL_EX VARCHAR(12) /* шифр удаленного экземпляра*/
Begin /* по временной системной таблице, содержащей удаленные записи,
определяем шифр книги, соответствующей последнему удаленномуэкземпляру */

SELECT @DEL_EX = ISBN From deleted
/* вызовем хранимую процедуру, которая определит количество экземпляров книги
с заданным шифром */
EXEC @Ntek = COUNT_EX @DEL_EX
/* Если больше нет экземпляров данной книги, то мы удаляем запись о книге из
таблицы BOOKS */
IF @Ntek = 0 DELETE from BOOKS WHERE BOOKS.ISBN = @DEL_EXENDGO
```

Динамический SQL

Возможности операторов встроенного SQL, описанные ранее, относятся к статическому SQL. В статическом SQL вся информация об операторе SQL известна на момент компиляции. Однако очень часто в диалоговых программах требуется более гибкая форма выполнения операторов SQL. Фактически, сам текст оператора SQL формируется уже во время выполнения программы.

Сформированный таким образом текст SQL-оператора поступает в СУБД, которая должна его скомпилировать и выполнить «на лету», в процессе работы приложения. Если мы снова вернемся к этапам выполнения SQL-операторов, то первые четыре действия, связанные с синтаксическим анализом, семантическим анализом, построением и оптимизацией плана выполнения запроса, выполняются на этапе компиляции. В момент исполнения этого оператора СУБД просто изымает хранимый план выполнения этого оператора и исполняет его.

В случае динамического SQL ситуация абсолютно иная. На момент компиляции мы не видим и не знаем текст оператора SQL и не можем выполнить ни одного из четырех обозначенных этапов. Все этапы СУБД должна будет выполнять с ходу, без предварительной подготовки в момент исполнения программы.

На рис. 12.5 представлены условные временные диаграммы выполнения SQL-операторов в статическом SQL и в динамическом SQL. Конечно, динамический SQL гораздо менее эффективен в смысле производительности, по сравнению со статическим SQL. Поэтому во всех случаях, когда это возможно, необходимо избегать динамического SQL. Но бывают случаи, когда отказ от динамического SQL серьезно усложняет приложение. Например, в случае с поиском по произвольному множеству параметров невозможно заранее предусмотреть все возможные комбинации запросов, даже если возможных параметров два десятка. А если их больше, то именно динамический SQL становится наиболее удобным методом решения необъятной проблемы.

Наиболее простой формой динамического SQL является оператор непосредственного выполнения EXECUTE IMMEDIATE. Этот оператор имеет следующий синтаксис:

```
EXECUTE IMMEDIATE <имя_базовой переменной>
```

Базовая переменная содержит текст SQL оператора.

Однако оператор непосредственного выполнения пригоден для выполнения операций, которые не возвращают результаты. Так же как в статическом SQL, для работы с множеством записей вводится понятие курсора и добавляются операторы по работе с курсором, и в динамическом SQL должны быть определены подобные структуры.

Прежде всего было предложено разделить выполнение SQL-оператора в динамическом SQL на два отдельных этапа. Первый этап называется подготовительным, он фактически включает 4 первых этапа выполнения SQL-операторов, рассмотренные нами ранее: синтаксический и семантический анализ, построение и оптимизация плана выполнения оператора.

Этот этап выполняется оператором PREPARE, синтаксис которого приведен ниже:

```
PREPARE <имя_оператора> FROM <имя_базовой переменной>
```

<имя_оператора> - это идентификатор базового языка.

Далее на втором этапе этот определенный на первом этапе оператор может быть выполнен операцией EXECUTE, которая имеет синтаксис:

```
EXECUTE <имя_оператора> USING {<список базовых переменных> |
DESCRIPTOR <имя_дескриптора>}
```

Здесь DESCRIPTOR — это некоторая структура, которая описывается на клиенте, но создается и управляется сервером. Дескриптор представляет совокупность элементов данных, принадлежащих СУБД. Программное обеспечение СУБД должно содержать и поддерживать набор операций над дескрипторами. Эта структура была введена в стандарт SQL2 для типизации динамического SQL.

В стандарт SQL2 введены следующие операции над дескрипторами:

- ALLOCATE DESCRIPTOR <имя_дескриптора> [WITH MAX <число_элементов>] — оператор связывает имя дескриптора с числом его базовых элементов и обеспечивает выделение памяти под данный дескриптор.
- DEALLOCATE DESCRIPTOR <имя_дескриптора> — оператор освобождает разделяемую память СУБД, занятую хранением описания данного дескриптора. После выполнения данного оператора невозможно обратиться к дескриптору ни с одной операцией.
- SET DESCRIPTOR {COUNT = <имя_базовой переменной> | VALUE <номер элемента> {<имя_элемента>= <имя_базовой переменной>[...]}} — оператор занесения в дескриптор описания передаваемых параметров. Описания передаются СУБД, которая их обрабатывает, внося соответствующие изменения в область данных, отведенную под дескриптор.

- GET DESCRIPTOR { <имя_базовой переменной> = COUNT | VALUE <номер элемента> {<имя_базовой переменной>=<имя_элемента>[...]} — оператор получения информации из дескрипторов после выполнения запроса.
- DESCRIBE [INPUT | OUTPUT] <имя_оператора> USING SQL DESCRIPTOR <имя_дескриптора> — оператор, позволяющий получить описания таблиц результатов запросов (DESCRIBE OUTPUT) или входных параметров (DESCRIBE INPUT).
- OPEN <имя_курсора> [USING <список базовых переменных> | USING SQL DESCRIPTOR <имя_дескриптора>] — динамический оператор открытия курсора.
- FETCH <имя_курсора> [USING <список базовых переменных> | USING SQL DESCRIPTOR <имя_дескриптора>] — динамический оператор перемещения по курсору.
- DEALLOCATE PREPARE <имя_оператора> — оператор уничтожает ранее подготовленный план выполнения оператора SQL и освобождает разделяемую память СУБД, связанную с хранением этого плана. Этот оператор имеет смысл применять, если вы не будете больше применять команду выполнения к подготовленному ранее оператору SQL.

Следует отметить, что в настоящий момент большинство СУБД реализуют динамический SQL несколько отличными от стандарта способами, однако в ближайшем будущем все поставщики вынуждены будут перейти к стандарту, так как именно это привлекает пользователей и делает переносимым разрабатываемое прикладное программное обеспечение.

ГЛАВА 13 Защита информации в базах данных

В современных СУБД поддерживается один из двух наиболее общих подходов к вопросу обеспечения безопасности данных: избирательный подход и обязательный подход. В обоих подходах единицей данных или «объектом данных», для которых должна быть создана система безопасности, может быть как вся база данных целиком, так и любой объект внутри базы данных.

Эти два подхода отличаются следующими свойствами:

- В случае избирательного управления некоторый пользователь обладает различными правами (привилегиями или полномочиями) при работе с данными объектами. Разные пользователи могут обладать разными правами доступа к одному и тому же объекту. Избирательные права характеризуются значительной гибкостью.
- В случае избирательного управления, наоборот, каждому объекту данных присваивается некоторый классификационный уровень, а каждый пользователь обладает некоторым уровнем допуска. При таком подходе доступом к определенному объекту данных обладают только пользователи с соответствующим уровнем допуска.
- Для реализации избирательного принципа предусмотрены следующие методы. В базу данных вводится новый тип объектов БД — это пользователи. Каждому пользователю в БД присваивается уникальный идентификатор. Для дополнительной защиты каждый пользователь кроме уникального идентификатора снабжается уникальным паролем, причем если идентификаторы пользователей в системе доступны системному администратору, то пароли пользователей хранятся чаще всего в специальном кодированном виде и известны только самим пользователям.
- Пользователи могут быть объединены в специальные группы пользователей. Один пользователь может входить в несколько групп. В стандарте вводится понятие группы PUBLIC, для которой должен быть определен минимальный стандартный набор прав. По умолчанию предполагается, что каждый вновь создаваемый пользователь, если специально не указано иное, относится к группе PUBLIC.

- Привилегии или полномочия пользователей или групп — это набор действий (операций), которые они могут выполнять над объектами БД.
- В последних версиях ряда коммерческих СУБД появилось понятие «роли». Роль — это поименованный набор полномочий. Существует ряд стандартных ролей, которые определены в момент установки сервера баз данных. И имеется возможность создавать новые роли, группируя в них произвольные полномочия. Введение ролей позволяет упростить управление привилегиями пользователей, структурировать этот процесс. Кроме того, введение ролей не связано с конкретными пользователями, поэтому роли могут быть определены и сконфигурированы до того, как определены пользователи системы.
- Пользователю может быть назначена одна или несколько ролей.
- Объектами БД, которые подлежат защите, являются все объекты, хранимые в БД: таблицы, представления, хранимые процедуры и триггеры. Для каждого типа объектов есть свои действия, поэтому для каждого типа объектов могут быть определены разные права доступа.

На самом элементарном уровне концепции обеспечения безопасности баз данных исключительно просты. Необходимо поддерживать два фундаментальных принципа: проверку полномочий и проверку подлинности (аутентификацию).

Проверка полномочий основана на том, что каждому пользователю или процессу информационной системы соответствует набор действий, которые он может выполнять по отношению к определенным объектам. Проверка подлинности означает достоверное подтверждение того, что пользователь или процесс, пытающийся выполнить санкционированное действие, действительно тот, за кого он себя выдает.

Система назначения полномочий имеет в некотором роде иерархический характер. Самыми высокими правами и полномочиями обладает системный администратор или администратор сервера БД. Традиционно только этот тип пользователей может создавать других пользователей и наделять их определенными полномочиями.

СУБД в своих системных каталогах хранит как описание самих пользователей, так и описание их привилегий по отношению ко всем объектам.

Далее схема предоставления полномочий строится по следующему принципу. Каждый объект в БД имеет владельца — пользователя, который создал данный объект. Владелец объекта обладает всеми правами-полномочиями на данный объект, в том числе он имеет право предоставлять другим пользователям полномочия по работе с данным объектом или забирать у пользователей ранее предоставленные полномочия.

В ряде СУБД вводится следующий уровень иерархии пользователей — это администратор БД. В этих СУБД один сервер может управлять множеством СУБД (например, MS SQL Server, Sybase).

В СУБД Oracle применяется однобазовая архитектура, поэтому там вводится понятие подсхемы — части общей схемы БД и вводится пользователь, имеющий доступ к подсхеме.

В стандарте SQL не определена команда создания пользователя, но практически во всех коммерческих СУБД создать пользователя можно не только в интерактивном режиме, но и программно с использованием специальных хранимых процедур. Однако для выполнения этой операции пользователь должен иметь право на запуск соответствующей системной процедуры.

В стандарте SQL определены два оператора: GRANT и REVOKE соответственно предоставления и отмены привилегий.

Оператор предоставления привилегий имеет следующий формат:

```
GRANT {<список действий> | ALL PRIVILEGES }
   ON <имя_объекта> TO {<имя_пользователя> | PUBLIC }
   [WITH GRANT OPTION ]
```

Здесь список действий определяет набор действий из общедопустимого перечня действий над объектом данного типа.

Параметр ALL PRIVILEGES указывает, что разрешены все действия из допустимых для объектов данного типа.

<имя_объекта> — задает имя конкретного объекта: таблицы, представления, хранимой процедуры, триггера.

<имя_пользователя> или PUBLIC определяет, кому предоставляются данные привилегии.

Параметр WITH GRANT OPTION является необязательным и определяет режим, при котором передаются не только права на указанные действия, но и право передавать эти права другим пользователям. Передавать права в этом случае пользователь может только в рамках разрешенных ему действий.

Рассмотрим пример, пусть у нас существуют три пользователя с абсолютно уникальными именами user1, user2 и user3. Все они являются пользователями одной БД.

User1 создал объект Tab1, он является владельцем этого объекта и может передать права на работу с эти объектом другим пользователям. Допустим, что пользователь user2 является оператором, который должен вводить данные в Tab1 (например, таблицу новых заказов), а пользователь user 3 является большим начальником (например, менеджером отдела), который должен регулярно просматривать введенные данные.

Для объекта типа таблица полным допустимым перечнем действий является набор из четырех операций: SELECT, INSERT, DELETE, UPDATE. При этом операция обновление может быть ограничена несколькими столбцами.

Общий формат оператора назначения привилегий для объекта типа таблица будет иметь следующий синтаксис:

```
GRANT {[SELECT][,INSERT][,DELETE][,UPDATE (<список столбцов>)]}
   ON <имя_таблицы>
   TO {<имя_пользователя> | PUBLIC }
   [WITH GRANT OPTION ]
```


Тогда резонно будет выполнить следующие назначения:

```
GRANT INSERT
   ON Tab1
   TO user2
GRANT SELECT
   ON Tab1
   TO user3
```

Эти назначения означают, что пользователь user2 имеет право только вводить новые строки в отношение Tab1, а пользователь user3 имеет право просматривать все строки в таблице Tab1.

При назначении прав доступа на операцию модификации можно уточнить, значение каких столбцов может изменять пользователь. Допустим, что менеджер отдела имеет право изменять цену на предоставляемые услуги. Предположим, что цена задается в столбце COST таблицы Tab1. Тогда операция назначения привилегий пользователю user3 может измениться и выглядеть следующим образом:

```
GRANT SELECT, UPDATE (COST)
   ON Tab1
   TO user3
```

Если наш пользователь user1 предполагает, что пользователь user4 может его заменять в случае его отсутствия, то он может предоставить этому пользователю все права по работе с созданной таблицей Tab1.

```
GRANT ALL PRIVILEGES
   ON Tab1
   TO user4
WITH GRANT OPTION
```

В этом случае пользователь user4 может сам назначать привилегии по работе с таблицей Tab1 в отсутствие владельца объекта пользователя user1. Поэтому в случае появления нового оператора пользователя user5 он может назначить ему права на ввод новых строк в таблицу командой

```
GRANT INSERT
   ON Tab1
   TO user5
```

Если при передаче полномочий набор операций над объектом ограничен, то пользователь, которому переданы эти полномочия, может передать другому пользователю только те полномочия, которые есть у него, или часть этих полномочий. Поэтому если пользователю user4 были делегированы следующие полномочия:

```
GRANT SELECT, UPDATE, DELETE
    ON Tab1
    TO user4
WITH GRANT OPTION.
```

то пользователь user4 не сможет передать полномочия на ввод данных пользователю user5, потому что эта операция не входит в список разрешенных для него самого.

Кроме непосредственного назначения прав по работе с таблицами эффективным методом защиты данных может быть создание представлений, которые будут содержать только необходимые столбцы для работы конкретного пользователя и предоставление прав на работу с данным представлением пользователю.

Так как представления могут соответствовать итоговым запросам, то для этих представлений недопустимы операции изменения, и, следовательно, для таких представлений набор допустимых действий ограничивается операцией SELECT. Если же представления соответствуют выборке из базовой таблицы, то для такого представления допустимыми будут все 4 операции: SELECT, INSERT, UPDATE и DELETE.

Для отмены ранее назначенных привилегий в стандарте SQL определен оператор REVOKE. Оператор отмены привилегий имеет следующий синтаксис:

```
REVOKE {<список операций> | ALL PRIVILEGES}
    ON <имя_объекта>
    FROM {<список пользователей> | PUBLIC } {CASCADE | RESTRICT }
```

Параметры CASCADE или RESTRICT определяют, каким образом должна производиться отмена привилегий. Параметр CASCADE отменяет привилегии не только пользователя, который непосредственно упоминался в операторе GRANT при предоставлении ему привилегий, но и всем пользователям, которым этот пользователь присвоил привилегии, воспользовавшись параметром WITH GRANT OPTION.

Например, при использовании операции:

```
REVOKE ALL PRIVILEGES
    ON Tab1
    TO user4 CASCADE
```

будут отменены привилегии и пользователя user5, которому пользователь user4 успел присвоить привилегии.

Параметр RESTRICKT ограничивает отмену привилегий только пользователю, непосредственно упомянутому в операторе REVOKE. Но при наличии делегированных привилегий этот оператор не будет выполнен. Так, например, операция:

```
REVOKE ALL PRIVILEGES
    ON Tab1
    TO user4 RESTRICT
```


не будет выполнена, потому что пользователь user4 передал часть своих полномочий пользователю user5.

Посредством оператора REVOKE можно отобрать все или только некоторые из ранее присвоенных привилегий по работе с конкретным объектом. При этом из описания синтаксиса оператора отмены привилегий видно, что можно отобрать привилегии одним оператором сразу у нескольких пользователей или у целой группы PUBLIC.

Поэтому корректным будет следующее использование оператора REVOKE:

```
REVOKE INSERT
    ON Tab1
    TO user2,user4 CASCADE
```

При работе с другими объектами изменяется список операций, которые используются в операторах GRANT и REVOKE.

По умолчанию действие, соответствующее запуску (исполнению) хранимой процедуры, назначается всем членам группы PUBLIC.

Если вы хотите изменить это условие, то после создания хранимой процедуры необходимо записать оператор REVOKE.

```
REVOKE EXECUTE
    ON COUNT_EX
    TO PUBLIC CASCADE
```

И теперь мы можем назначить новые права пользователю user4.

```
GRANT EXECUTE
    ON COUNT_EX
    TO user4
```

Системный администратор может разрешить некоторому пользователю создавать и изменять таблицы в некоторой БД. Тогда он может записать оператор предоставления прав следующим образом:

```
GRANT CREATE TABLE, ALTER TABLE, DROP TABLE
    ON DB_LIB
    TO user1
```

В этом случае пользователь user1 может создавать, изменять или удалять таблицы в БД DB_LIB, однако он не может разрешить создавать или изменять таблицы в этой БД другим пользователям, потому что ему дано разрешение без права делегирования своих возможностей.

В некоторых СУБД пользователь может получить права создавать БД. Например, в MS SQL Server системный администратор может предоставить пользователю main_user право на создание своей БД на данном сервере. Это может быть сделано следующей командой:

```
GRANT CREATE DATABASE
    ON SERVER_0
    TO main_user
```

По принципу иерархии пользователь main_user, создав свою БД, теперь может предоставить права на создание или изменение любых объектов в этой БД другим пользователям.

В СУБД, которые поддерживают однобазовую архитектуру, такие разрешения недопустимы. Например, в СУБД Oracle на сервере создается только одна БД, но пользователи могут работать на уровне подсхемы (части таблиц БД и связанных с ними объектов). Поэтому там вводится понятие системных привилегий. Их очень много, 80 различных привилегий.

Для того чтобы разрешить пользователю создавать объекты внутри этой БД, используется понятие системной привилегии, которая может быть назначена одному или нескольким пользователям. Они выдаются только на действия и конкретный тип объекта. Поэтому если вы, как системный администратор, предоставили пользователю право создания таблиц (CREATE TABLE), то для того чтобы он мог создать триггер для таблицы, ему необходимо предоставить еще одну системную привилегию CREATE TRIGGER. Система защиты в Oracle считается одной из самых мощных, но это имеет и обратную сторону — она весьма сложная. Поэтому задача администрирования в Oracle требует хорошего знания как семантики принципов поддержки прав доступа, так и физической реализации этих возможностей.

Реализация системы защиты в MS SQL Server

SQL server 6.5 поддерживает 3 режима проверки при определении прав пользователя:

1. Стандартный (standard).
2. Интегрированный (integrated security).
3. Смешанный (mixed).

Стандартный режим защиты предполагает, что каждый пользователь должен иметь учетную запись как пользователь домена NT Server. Учетная запись пользователя домена включает имя пользователя и его индивидуальный пароль. Пользователи доменов могут быть объединены в группы. Как пользователь домена пользователь получает доступ к определенным ресурсам домена. В качестве одного из ресурсов домена и рассматривается SQL Server. Но для доступа к SQL Server пользователь должен иметь учетную запись пользователя MS SQL Server. Эта учетная запись также должна включать уникальное имя пользователя сервера и его пароль. При подключении к операционной среде пользователь задает свое имя и пароль пользователя домена. При подключении к серверу баз данных пользователь задает свое уникальное имя пользователя SQL Server и свой пароль.

Интегрированный режим предполагает, что для пользователя задается только одна учетная запись в операционной системе, как пользователя домена, а SQL Server идентифицирует пользователя по его данным в этой учетной записи. В этом случае пользователь задает только одно свое имя и один пароль.

В случае смешанного режима часть пользователей может быть подключена к серверу с использованием стандартного режима, а часть с использованием интегрированного режима.

В MS SQL Server 7.0 оставлены только 2 режима: интегрированный, называемый Windows NT Authentication Mode (Windows NT Authentication), и смешанный, Mixed Mode (Windows NT Authentication and SQL Server Authentication). Алгоритм проверки аутентификации пользователя в MS SQL Server 7.0 приведен на рис. 13.1.

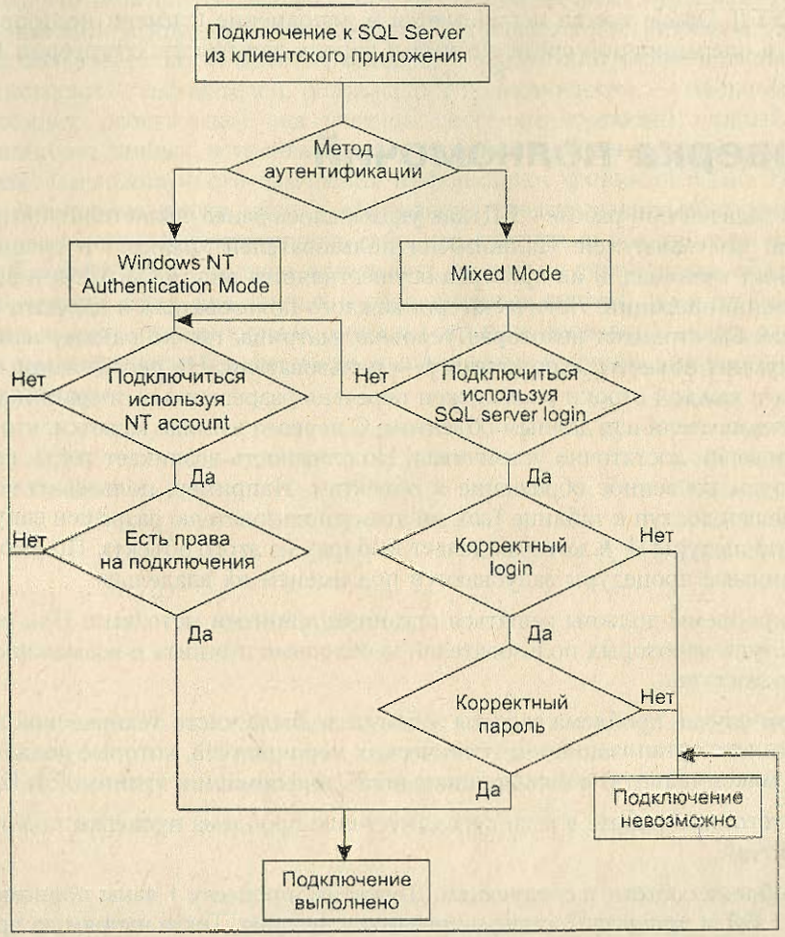


Рис. 13.1. Алгоритм проверки аутентификации пользователя в MS SQL Server 7.0

При попытке подключения к серверу БД сначала проверяется, какой метод аутентификации определен для данного пользователя. Если определен Windows NT Authentication Mode, то далее проверяется, имеет ли данный пользователь домена доступ к ресурсу SQL Server, если он имеет доступ, то выполняется по-

пытка подключения с использованием имени пользователя и пароля, определенных для пользователя домена; если данный пользователь имеет права подключения к SQL Server, то подключение выполняется успешно, в противном случае пользователь получает сообщение о том, что данному пользователю не разрешено подключение к SQL Server. При использовании смешанного режима аутентификации средствами SQL Server проводится последовательная проверка имени пользователя (login) и его пароля (password); если эти параметры заданы корректно, то подключение завершается успешно, в противном случае пользователь также получает сообщение о невозможности подключиться к SQL Server.

Для СУБД Oracle всегда используется в дополнение к имени пользователя и пароля в операционной среде его имя и пароль для работы с сервером БД.

Проверка полномочий

Второй задачей при работе с БД, как указывалось ранее, является проверка полномочий пользователей. Полномочия пользователей хранятся в специальных системных таблицах, и их проверка осуществляется ядром СУБД при выполнении каждой операции. Логически для каждого пользователя и каждого объекта в БД как бы строится некоторая условная матрица, где по одному измерению расположены объекты, а по другому — пользователи. На пересечении каждого столбца и каждой строки расположен перечень разрешенных операций для данного пользователя над данным объектом. С первого взгляда кажется, что эта модель проверки достаточно устойчивая. Но сложность возникает тогда, когда мы используем косвенное обращение к объектам. Например, пользователю `user_N` не разрешен доступ к таблице `Tab1`, но этому пользователю разрешен запуск хранимой процедуры `SP_N`, которая делает выборку из этого объекта. По умолчанию все хранимые процедуры запускаются под именем их владельца.

Такие проблемы должны решаться организационными методами. При разрешении доступа некоторых пользователей необходимо помнить о возможности косвенного доступа.

В любом случае проблема защиты никогда не была чисто технической задачей, это комплекс организационно-технических мероприятий, которые должны обеспечить максимальную конфиденциальность информации, хранимой в БД.

Кроме того, при работе в сети существует еще проблема проверки подлинности полномочий.

Эта проблема состоит в следующем. Допустим, процессу 1 даны полномочия по работе с БД, а процессу 2 такие полномочия не даны. Тогда напрямую процесс 2 не может обратиться к БД, но он может обратиться к процессу 1 и через него получить доступ к информации из БД.

Поэтому в безопасной среде должна присутствовать модель проверки подлинности, которая обеспечивает подтверждение заявленных пользователями или процессами идентификаторов. Проверка полномочий приобрела еще большее значение в условиях массового распространения распределенных вычислений. При

существующем высоком уровне связности вычислительных систем необходимо контролировать все обращения к системе.

Проблемы проверки подлинности обычно относят к сфере безопасности коммуникаций и сетей, поэтому мы не будем их здесь более обсуждать, за исключением следующего замечания. В целостной системе компьютерной безопасности, где четкое выполнение программы защиты информации обеспечивается за счет взаимодействия соответствующих средств в операционных системах, сетях, базах данных, проверка подлинности имеет прямое отношение к безопасности баз данных.

Заметим, что модель безопасности, основанная на базовых механизмах проверки полномочий и проверки подлинности, не решает таких проблем, как украденные пользовательские идентификаторы и пароли или злонамеренные действия некоторых пользователей, обладающих полномочиями, — например, когда программист, работающий над учетной системой, имеющей полный доступ к учетной базе данных, встраивает в код программы «Троянского коня» с целью хищения или намеренного изменения информации, хранимой в БД. Такие вопросы выходят за рамки нашего обсуждения средств защиты баз данных, но следует тем не менее представлять себе, что программа обеспечения информационной безопасности должна охватывать не только технические области (такие как защита сетей, баз данных и операционных систем), но и проблемы физической защиты, надежности персонала (скрытые проверки), аудит, различные процедуры поддержки безопасности, выполняемые вручную или частично автоматизированные.

ГЛАВА 14 Обобщенная архитектура СУБД

Мы рассмотрели отдельные аспекты работы СУБД. Теперь попробуем кратко обобщить все, что узнали, и построим некоторую условную обобщенную структуру СУБД. На рис. 14.1 изображена такая структура. Здесь условно показано, что СУБД должна управлять внешней памятью, в которой расположены файлы с данными, файлы журналов и файлы системного каталога.

С другой стороны, СУБД управляет и оперативной памятью, в которой располагаются буфера с данными, буфера журналов, данные системного каталога, которые необходимы для поддержки целостности и проверки привилегий пользователей. Кроме того, в оперативной памяти во время работы СУБД располагается информация, которая соответствует текущему состоянию обработки запросов, там хранятся планы выполнения скомпилированных запросов и т. д.

Модуль управления внешней памятью обеспечивает создание необходимых структур внешней памяти как для хранения данных, непосредственно входящих в БД, так и для служебных целей, например для ускорения доступа к данным в некоторых случаях (обычно для этого используются индексы). Как мы рассматривали ранее, в некоторых реализациях СУБД активно используются возможности существующих файловых систем, в других работа производится вплоть до уровня устройств внешней памяти. Но подчеркнем, что в развитых СУБД пользователи в любом случае не обязаны знать, использует ли СУБД файловую систему, и если использует, то как организованы файлы. В частности, СУБД поддерживает собственную систему именования объектов БД.

Модуль управления буферами оперативной памяти предназначен для решения задач эффективной буферизации, которая используется практически для выполнения всех остальных функций СУБД.

Условно оперативную память, которой управляет СУБД, можно представить как совокупность буферов, хранящих страницы данных, буферов, хранящих страницы журналов транзакций и область совместно используемого пула (см. рис. 14.2). Последняя область содержит фрагменты системного каталога, которые необходимо постоянно держать в оперативной памяти, чтобы ускорить обработку запросов пользователей, и область операторов SQL с курсорами. Фрагменты системного каталога в некоторых реализациях называются словарем данных. В стандарте SQL2 определены общие требования к системному каталогу.

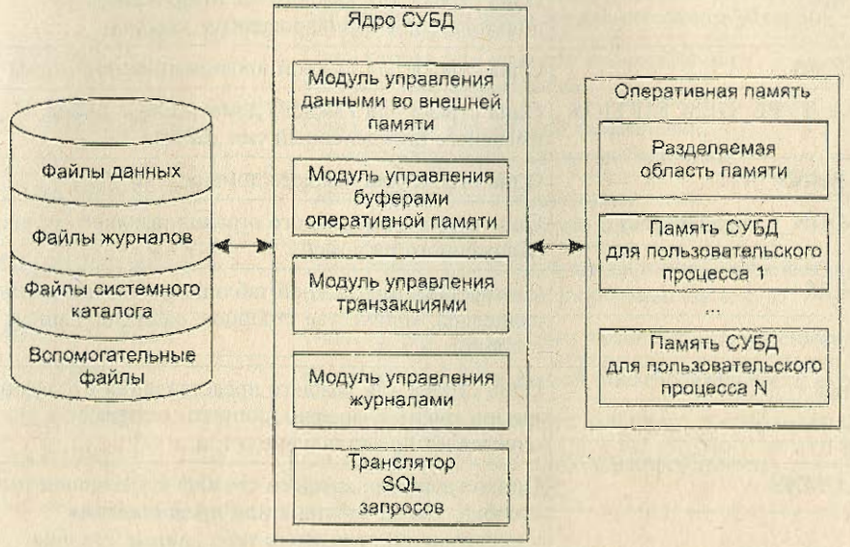


Рис. 14.1. Обобщенная структура СУБД

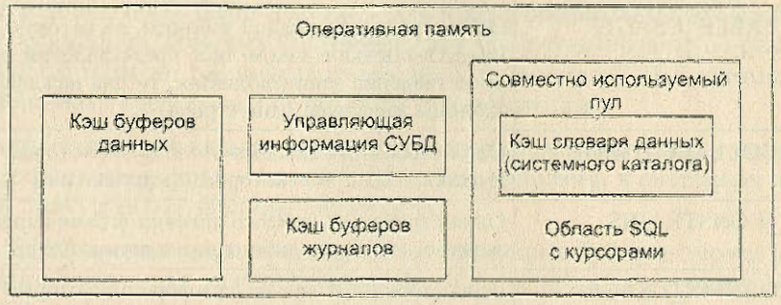


Рис. 14.2. Оперативная память, управляемая СУБД

Системный каталог в реляционных СУБД представляет собой совокупность специальных таблиц, которыми владеет сама СУБД. Таблицы системного каталога создаются автоматически при установке программного обеспечения сервера БД. Все системные таблицы обычно объединяются некоторым специальным «системным идентификатором пользователя». При обработке SQL-запросов СУБД постоянно обращается к этим таблицам. В некоторых СУБД разрешен ограниченный доступ пользователей к ряду системных таблиц, однако только в режиме чтения. Только системный администратор имеет некоторые права на модификацию данных в некоторых системных таблицах.

Каждая таблица системного каталога содержит информацию об отдельных структурных элементах БД. В стандарте SQL2 определены следующие системные таблицы:

Таблица 14.1. Содержание системного каталога по стандарту SQL2

Системная таблица	Содержание
USERS	Одна строка для каждого идентификатора пользователя с зашифрованным паролем
SCHEMA	Одна строка для каждой информационной схемы
DATA_TYPE_DESCRIPTION	Одна строка для каждого домена или столбца, имеющего определенный тип данных
DOMAINS	Одна строка для каждого домена
DOMAIN_CONSTRAINS	Одна строка для каждого ограничивающего условия, наложенного на домен
TABLES	Одна строка для каждой таблицы с указанием имени, владельца, количества столбцов, размеров данных столбцов, и т. д.
VIEWS	Одна строка для каждого представления с указанием имени, имени владельца, запроса, который определяет представление и т. д.
COLUMNS	Одна строка для каждого столбца с указанием имени столбца, имени таблицы или представления, к которому он относится, типа данных столбца, его размера, допустимости или недопустимости неопределенных значений (NULL) и т. д.
VIEW_TABLE_USAGE	Одна строка для каждой таблицы, на которую имеется ссылка в каком-либо представлении (если представление многотабличное, то для каждой таблицы заносится одна строка)
VIEW_COLUMN_USAGE	Одна строка для каждого столбца, на который имеется ссылка в некотором представлении
TABLE_CONSTRAINS	Одна строка для каждого условия ограничения, заданного в каком-либо определении таблицы
KEY_COLUMN_USAGE	Одна строка для каждого столбца, на который наложено условие уникальности и который присутствует в определении первичного или внешнего ключа (если первичный или внешний ключ заданы несколькими столбцами, то для каждого из них задается отдельная строка)
REFERENTIAL_CONSTRAINTS	Одна строка для каждого внешнего ключа, присутствующего в определении таблицы
CHECK_ CONSTRAINTS	Одна строка для каждого условия проверки, заданного в определении таблицы
CHECK_TABLE_USAGE	Одна строка для каждой таблицы, на которую имеется ссылка в условиях проверки, ограничительном условии для домена или всей таблицы

Системная таблица	Содержание
CHECK_COLUMN_USAGE	Одна строка для каждого столбца, на который имеется ссылка в условии проверки, ограничительном условии для домена или ином ограничительном условии
ASSERTIONS	Одна строка для каждого декларативного утверждения целостности
TABLE_PRIVILEGES	Одна строка для каждой привилегии, предоставленной на какую-либо таблицу
COLUMN_PRIVILEGES	Одна строка для каждой привилегии, предоставленной на какой-либо столбец
USAGE_PRIVILEGES	Одна строка для каждой привилегии, предоставленной на какой-либо домен, набор символов и т. д.
CHARACTER_SETS	Одна строка для каждого заданного набора символов
COLLATIONS	Одна строка для заданной последовательности
TRANSLATIONS	Одна строка для каждого заданного преобразования
SQL_LAGUAGES	Одна строка для каждого заданного языка, поддерживаемого СУБД

Стандарт SQL2 не требует, чтобы СУБД в точности поддерживала требуемый набор системных таблиц. Стандарт ограничивается требованием того, чтобы для рядовых пользователей были доступны некоторые специальные представления системного каталога. Поэтому системные таблицы организованы по-разному в разных СУБД и имеют различные имена, но большинство СУБД предо-ставляют ряд основных представлений рядовым пользователям.

Кроме того, системный каталог отражает некоторые дополнительные возможности, предоставляемые конкретными СУБД. Так, например, в системном каталоге Oracle присутствуют таблицы синонимов.

Область SQL содержит данные связывания, временные буферы, дерево разбора и план выполнения для каждого оператора SQL, переданного серверу БД. Область разделяемого пула ограничена в размере, поэтому, возможно, в ней не могут поместиться все операторы SQL, которые были выполнены с момента запуска сервера БД. Ядро СУБД удаляет старые, давно не используемые операторы, освобождая память под новые операторы SQL. Если пользователь выполняет запрос, план выполнения которого уже хранится в разделяемом пуле, то СУБД не производит его разбор и построение нового плана, она сразу запускает его на выполнение, возможно, с новыми параметрами.

Модуль управления транзакциями поддерживает механизмы фиксации и отката транзакций, он связан с модулем управления буферами оперативной памяти и обеспечивает сохранение всей информации, которая требуется после мягких или жестких сбоев в системе. Кроме того, модуль управления транзакциями содержит специальный механизм поиска тупиковых ситуаций или взаимоблокировок и реализует одну из принятых стратегий принудительного завершения транзакций для развязывания тупиковых ситуаций.

Особое внимание надо обратить на модуль поддержки SQL. Это практически транслятор с языка SQL и блок оптимизации запросов.

В общем, оптимизация запросов может быть разделена на синтаксическую и семантическую.

Методы синтаксической оптимизации запросов

Методы синтаксической оптимизации запросов связаны с построением некоторой эквивалентной формы, называемой иногда канонической формой, которая требует меньших затрат на выполнение запроса, но дает результат, полностью эквивалентный исходному запросу.

К методам, используемым при синтаксической оптимизации запросов, относятся следующие:

- **Логические преобразования запросов.** Прежде всего это относится к преобразованию предикатов, входящих в условие выборки. Предикаты, содержащие операции сравнения простых значений. Такой предикат имеет вид `арифметическое выражение ОС арифметическое выражение`, где `ОС` — операция сравнения, а арифметические выражения левой и правой частей в общем случае содержат имена полей отношений и константы (в языке SQL среди констант могут встречаться и имена переменных объемлющей программы, значения которых становятся известными только при реальном выполнении запроса).

  Канонические представления могут быть различными для предикатов разных типов. Если предикат включает только одно имя поля, то его каноническое представление может, например, иметь вид `имя поля ОС константное арифметическое выражение` (эта форма предиката — простой предикат селекции — очень полезна при выполнении следующего этапа оптимизации). Если начальное представление предиката имеет вид

  ```
  (n+12)*R.B ОС 100
  ```

  здесь `n` — переменная языка, `R.B` — имя столбца B отношения R, `ОС` – допустимая операция сравнения.

  Каноническим представлением такого предиката может быть

  ```
  R.B ОС 100/(n+12)
  ```

  В этом случае мы один раз для заданного значения переменной `n` вычисляем выражение в скобках и правую часть операции сравнения `100/(n +12)`, а потом каждую строку можем сравнивать с полученным значением.

  Если предикат включает в точности два имени поля разных отношений (или двух разных вхождений одного отношения), то его каноническое представление может иметь вид `имя поля ОС арифметическое выражение`, где арифметическое выражение в правой части включает только константы и второе имя поля (это тоже форма, полезная для выполнения следующего шага оптимизации, —

предикат соединения; особенно важен случай эквисоединения, когда `ОС` — это равенство). Если в начальном представлении предикат имеет вид:

```
12*(R1.A)-n*(R2.B) ОС m,
```

то его каноническое представление:

```
R1.A ОС (m+n*(R2.B)/12
```

В общем случае желательно приведение предиката к каноническому представлению вида `арифметическое выражение ОС константное арифметическое выражение`, где выражения правой и левой частей также приведены к каноническому представлению. В дальнейшем можно произвести поиск общих арифметических выражений в разных предикатах запроса. Это оправдано, поскольку при выполнении запроса вычисление арифметических выражений будет производиться при выборке каждого очередного кортежа, то есть потенциально большое число раз.

При приведении предикатов к каноническому представлению можно вычислять константные выражения и избавляться от логических отрицаний.

Еще один класс логических преобразований связан с приведением к каноническому виду логического выражения, задающего условие выборки запроса. Как правило, используются либо дизъюнктивная, либо конъюнктивная нормальные формы. Выбор канонической формы зависит от общей организации оптимизатора.

При приведении логического условия к каноническому представлению можно производить поиск общих предикатов (они могут существовать изначально, могут появиться после приведения предикатов к каноническому виду или в процессе нормализации логического условия) и упрощать логическое выражение за счет, например, выявления конъюнкции взаимно противоречащих предикатов.

- **Преобразования запросов с изменением порядка реляционных операций.** В традиционных оптимизаторах распространены логические преобразования, связанные с изменением порядка выполнения реляционных операций.

  Например, имеем следующий запрос:

  ```
  R1 NATURAL JOIN R2
     WHERE R1.A ОС a AND
     R2.B C b
  ```

  Здесь `a` и `b` некоторые константы, которые ограничивают значение атрибутов отношений `R1` и `R2`.

  Если мы его рассмотрим в терминах реляционной алгебры, то это естественное соединение отношений `R1` и `R2`, в которых заданы внутренние ограничения на кортежи каждого отношения.

  Для уменьшения числа соединяемых кортежей резонно сначала произвести операции выборки на каждом отношении и только после этого перейти к операции естественного соединения.

Поэтому данный запрос будет эквивалентен следующей последовательности операций реляционной алгебры:

```
R3 = R1[R1.A OC a]
R4 = R2[R2.B C b]
R5 = R3*[ ]*R4
```

Хотя немногие реляционные системы имеют языки запросов, основанные в чистом виде на реляционной алгебре, правила преобразований алгебраических выражений могут быть полезны и в других случаях. Довольно часто реляционная алгебра используется в качестве основы внутреннего представления запроса. Естественно, что после этого можно выполнять и алгебраические преобразования.

В частности, существуют подходы, связанные с преобразованием запросов на языке SQL к алгебраической форме. Особенно важно то, что реляционная алгебра более проста, чем язык SQL. Преобразование запроса к алгебраической форме упрощает дальнейшие действия оптимизатора по выборке оптимальных планов. Вообще говоря, развитый оптимизатор запросов системы, ориентированной на SQL, должен выявить все возможные планы выполнения любого запроса, но «пространство поиска» этих планов в общем случае очень велико; в каждом конкретном оптимизаторе используются свои эвристики для сокращения пространства поиска. Некоторые, возможно, наиболее оптимальные планы никогда не будут рассматриваться. Разумное преобразование запроса на SQL к алгебраическому представлению сокращает пространство поиска планов выполнения запроса с гарантией того, что оптимальные планы потеряны не будут.

- **Приведение запросов с вложенными подзапросами к запросам с соединениями.** Основным отличием языка SQL от языка реляционной алгебры является возможность использовать в логическом условии выборки предикаты, содержащие вложенные подзапросы. Глубина вложенности не ограничивается языком, то есть, вообще говоря, может быть произвольной. Предикаты с вложенными подзапросами при наличии общего синтаксиса могут обладать весьма различной семантикой. Единственным общим для всех возможных семантик вложенных подзапросов алгоритмом выполнения запроса является вычисление вложенного подзапроса всякий раз при вычислении значения предиката. Поэтому естественно стремиться к такому преобразованию запроса, содержащего предикаты со вложенными подзапросами, которое сделает семантику подзапроса более явной, предоставив тем самым в дальнейшем оптимизатору возможность выбрать способ выполнения запроса, наиболее точно соответствующий семантике подзапроса.

  Каноническим представлением запроса на n отношениях называется запрос, содержащий n−1 предикат соединения и не содержащий предикатов с вложенными подзапросами. Фактически каноническая форма — это алгебраическое представление запроса.

  Например, запрос с вложенным подзапросом:


```
(SELECT R1.A
   FROM R1
   WHERE R1.B IN
   (SELECT R2.B FROM R2 WHERE R1.C = R2.D)
)
```

эквивалентен

```
(SELECT R1.A
   FROM R1, R2
   WHERE R1.A = R2.B AND R1.C = R2.D)
```

Второй запрос:

```
(SELECT R1.A FROM R1 WHERE R1.K =
   (SELECT AVG (R2.B) FROM R2 WHERE R1.C = R2.D)
```

или

```
(SELECT R1.A
   FROM R1, R3
   WHERE R1.C = R3.D AND R1.K = R3.L)
   R3 = SELECT R2.D, L AVG (R2.B)
      FROM R2
      GROUP BY R2.D
```

При использовании подобного подхода в оптимизаторе запросов не обязательно производить формальные преобразования запросов. Оптимизатор должен в большей степени использовать семантику обрабатываемого запроса, а каким образом она будет распознаваться — это вопрос техники.

Заметим, что в кратко описанном нами подходе имеются некоторые тонкие семантические некорректности. Известны исправленные методы, но они слишком сложны технически, чтобы рассматривать их в данном пособии.

Методы семантической оптимизации запросов

Рассмотренные ранее методы никак не связаны с семантикой конкретной БД, они применимы к любой БД, вне зависимости от ее конкретного содержания. Семантические методы оптимизации основаны как раз на учете семантики конкретной БД. Таких методов в различных реализациях может быть множество, мы с вами коснемся лишь некоторых из них:

- **Преобразование запросов с учетом семантической информации.** Это прежде всего относится к запросам, которые выполняются над представлениями. Само представление представляет собой запрос. В БД представление хранится в виде скомпилированного плана выполнения запроса, то есть в нем в не-

которой канонической форме представлены уже все предикаты и сам план выполнения запроса. При преобразовании внешнего запроса производится объединение внешнего запроса с внутренней формой запроса, составляющего основу представления, и строится обобщенная каноническая форма, объединяющая оба запроса. Для этой новой формы проводится анализ и преобразование предикатов. Поэтому при выполнении запроса над представлением будет выполнено не два, а только один запрос, оптимизированный по обобщенным параметрам запроса.

- **Использование ограничений целостности при анализе запросов.** Ограничения целостности связаны с условиями, которые накладываются на значения столбцов таблицы. При выполнении запросов над таблицами условия запросов объединяются специальным образом с условиями ограничений таблицы и полученные обобщенные предикаты уже анализируются. Допустим, что мы ищем в нашей библиотеке читателей с возрастом более 100 лет, но если у нас есть ограничение, заданное для таблицы READERS, которое ограничивает дату рождения наших читателей, так чтобы читатель имел дату рождения в пределах от 17 до 100 лет включительно. Поэтому оптимизатор запроса, сопоставив два эти предиката, может сразу определить, что результатом запроса будет пустое множество.

После оптимизации запрос имеет непроцедурный вид, то есть в нем не определен жесткий порядок выполнения элементарных операций над исходными объектами. На следующем этапе строятся все возможные планы выполнения запросов и для каждого из них производятся стоимостные оценки. Оценка планов выполнения запроса основана на анализе текущих объемов данных, хранящихся в отношениях БД, и на статистическом анализе хранимой информации. В большинстве СУБД ведется учет диапазона значений отдельного столбца с указанием процентного содержания для каждого диапазона. Поэтому при построении плана запроса СУБД может оценить объем промежуточных отношений и построить план таким образом, чтобы на наиболее ранних этапах выполнения запроса минимизировать количество строк, включаемых в промежуточные отношения.

Кроме ядра СУБД каждый поставщик обеспечивает специальные инструментальные средства, облегчающие администрирование БД и разработку новых проектов БД и пользовательских приложений для данного сервера. В последнее время практически все утилиты и инструментальные средства имеют развитый графический интерфейс.

Для разработки приложений пользователи могут применять не только инструментальные средства, поставляемые вместе с сервером БД, но и средства сторонних поставщиков. Так, в нашей стране получила большую популярность инструментальная среда Delphi, которая позволяет разрабатывать приложения для различных серверов БД. За рубежом более популярными являются инструментальные системы быстрой разработки приложений (RAF Rapid Application Foundation) продукты компании Advanced Information System, инструментальной среды Power Builder фирмы Power Soft, системы SQL Windows фирмы Gupta (Taxedo).

ЗАКЛЮЧЕНИЕ Перспективы развития БД и СУБД

Современные базы данных являются основой многочисленных информационных систем. Информация, накопленная в них, является чрезвычайно ценным материалом, и в настоящий момент широко распространяются методы обработки баз данных с точки зрения извлечения из них дополнительных знаний, методов, которые связаны с обобщением и различными дополнительными способами обработки данных. Базы данных в данной концепции выступают как хранилища информации, это направление называется «Хранилища данных» (Data Warehouse).

Для работы с «Хранилищами данных» наиболее значимым становится так называемый интеллектуальный анализ данных (ИАД), или data mining, — это процесс выявления значимых корреляций, образцов и тенденций в больших объемах данных. Учитывая высокие темпы роста объемов накопленной в современных хранилищах данных информации, невозможно недооценить роль ИАД. По мнению специалистов Gartner Group, уже в 1998 г. ИАД вошел в десятку важнейших информационных технологий. В последние годы началось активное внедрение технологии ИАД. Ее активно используют как крупные корпорации, так и более мелкие фирмы, которые серьезно относятся к вопросам анализа и прогнозирования своей деятельности. Естественно, на рынке программных продуктов стали появляться соответствующие инструментальные средства.

Особенно широко методы ИАД применяются в бизнес-приложениях аналитиками и руководителями компаний. Для этих категорий пользователей разрабатываются инструментальные средства высокого уровня, позволяющие решать достаточно сложные практические задачи без специальной математической подготовки. Актуальность использования ИАД в бизнесе связана с жесткой конкуренцией, возникшей вследствие перехода от «рынка продавца» к «рынку покупателя». В этих условиях особенно важно качество и обоснованность принимаемых решений, что требует строгого количественного анализа имеющихся данных. При работе с большими объемами накапливаемой информации необходимо постоянно оперативно отслеживать динамику рынка, а это практически невозможно без автоматизации аналитической деятельности.

В бизнес-приложениях наибольший интерес представляет интеграция методов интеллектуального анализа данных с технологией оперативной аналитической обработки данных (On-Line Analytical Processing, OLAP). OLAP использует многомерное представление агрегированных данных для быстрого доступа к важной информации и дальнейшего ее анализа.

Системы OLAP обеспечивают аналитикам и руководителям быстрый последовательный интерактивный доступ к внутренней структуре данных и возможность преобразования исходных данных с тем, чтобы они позволяли отразить структуру системы нужным для пользователя способом. Кроме того, OLAP-системы позволяют просматривать данные и выявлять имеющиеся в них закономерности либо визуально, либо простейшими методами (такими как линейная регрессия), а включение в их арсенал нейросетевых методов обеспечивает существенное расширение аналитических возможностей.

В основе концепции оперативной аналитической обработки (OLAP) лежит многомерное представление данных. Термин OLAP ввел Кодд (E. F. Codd) в 1993 году. В своей статье он рассмотрел недостатки реляционной модели, в первую очередь невозможность «объединять, просматривать и анализировать данные с точки зрения множественности измерений, то есть самым понятным для корпоративных аналитиков способом», и определил общие требования к системам OLAP, расширяющим функциональность реляционных СУБД и включающим многомерный анализ как одну из своих характеристик.

Следует заметить, что Кодд обозначает термином OLAP многомерный способ представления данных исключительно на концептуальном уровне. Используемые им термины — «Многомерное концептуальное представление» («Multidimensional conceptual view»), «Множественные измерения данных» («Multiple data dimensions»), «Сервер OLAP» («OLAP server») — не определяют физического механизма хранения данных (термины «многомерная база данных» и «многомерная СУБД» не встречаются ни разу).

Часто в публикациях аббревиатурой OLAP обозначается не только многомерный взгляд на данные, но и хранение самих данных в многомерной БД, что в принципе неверно.

По Кодду, многомерное концептуальное представление (multi-dimensional conceptual view) является наиболее естественным взглядом управляющего персонала на объект управления. Оно представляет собой множественную перспективу, состоящую из нескольких независимых измерений, вдоль которых могут быть проанализированы определенные совокупности данных. Одновременный анализ по нескольким измерениям данных определяется как многомерный анализ. Каждое измерение включает направления консолидации данных, состоящие из серии последовательных уровней обобщения, где каждый вышестоящий уровень соответствует большей степени агрегации данных по соответствующему измерению. Так, измерение Исполнитель может определяться направлением консолидации, состоящим из уровней обобщения «предприятие—подразделение—отдел—служащий». Измерение Время может даже включать два направления консолидации — «год—квартал—месяц—день» и «неделя—день», поскольку счет времени по месяцам и по неделям несовместим. В этом случае становится возможным произвольный выбор желаемого уровня детализации информации по каждому

из измерений. Операция спуска (drilling down) соответствует движению от высших ступеней консолидации к низшим; напротив, операция подъема (rolling up) означает движение от низших уровней к высшим.

Следующим новым направлением в развитии систем управления базами данных является направление, связанное с отказом от нормализации отношений. Во многом нормализация отношений нарушает естественные иерархические связи между объектами, которые достаточно распространены в нашем мире. Возможность сохранять их на концептуальном (но не на физическом) уровне позволяет пользователям более естественно отражать семантику предметной области. В настоящий момент уже существует теоретическое обоснование работы с ненормализованными отношениями и практические реализации подобных систем.

Дальнейшим расширением в структурных преобразованиях являются объектно-ориентированные базы данных. В объектно-ориентированной парадигме предметная область моделируется как множество классов взаимодействующих объектов. Каждый объект характеризуется набором свойств, которые являются как бы его пассивными характеристиками и набором методов работы с этим объектом. Работать с объектом можно только с использованием его методов. Атрибуты объекта могут принимать определенное множество допустимых значений, набор конкретных значений атрибутов объекта определяет его состояние. Используя методы работы с объектом, можно изменять значение его атрибутов и тем самым как бы изменять состояние самого объекта. Множество объектов с одним и тем же набором атрибутов и методов образует класс объектов. Объект должен принадлежать только одному классу (если не учитывать возможности наследования). Допускается наличие примитивных предопределенных классов, объекты-экземпляры которых не имеют атрибутов: целые, строки и т. д. Класс, объекты которого могут служить значениями атрибута объектов другого класса, называется доменом этого атрибута.

Одной из наиболее перспективных черт объектно-ориентированной парадигмы является принцип наследования. Допускается порождение нового класса на основе уже существующего класса, и этот процесс называется наследованием. В этом случае новый класс, называемый подклассом существующего класса (суперкласса), наследует все атрибуты и методы суперкласса. В подклассе, кроме того, могут быть определены дополнительные атрибуты и методы. Различаются случаи простого и множественного наследования. В первом случае подкласс может определяться только на основе одного суперкласса, во втором случае суперклассов может быть несколько. Если в языке или системе поддерживается единичное наследование классов, набор классов образует древовидную иерархию. При поддержании множественного наследования классы связаны в ориентированный граф с корнем, называемый решеткой классов. Объект подкласса считается принадлежащим любому суперклассу этого класса.

Можно считать, что наиболее важным качеством ООБД (объектно-ориентированной базы данных), которое позволяет реализовать объектно-ориентированный подход, является учет поведенческого аспекта объектов.

В прикладных информационных системах, основывавшихся на БД с традиционной организацией (вплоть до тех, которые базировались на семантических моде-

лях данных), существовал принципиальный разрыв между структурной и поведенческой частями. Структурная часть системы поддерживалась всем аппаратом БД, ее можно было моделировать, верифицировать и т. д., а поведенческая часть создавалась изолированно. В частности, отсутствовали формальный аппарат и системная поддержка совместного моделирования и гарантий согласованности структурной (статической) и поведенческой (динамической) частей. В среде ООБД проектирование, разработка и сопровождение прикладной системы становятся процессом, в котором интегрируются структурный и поведенческий аспекты. Конечно, для этого нужны специальные языки, позволяющие определять объекты и создавать на их основе прикладную систему.

Специфика применения объектно-ориентированного подхода для организации и управления БД потребовала уточненного толкования классических концепций и некоторого их расширения.

Прежде всего, возникло направление, которое предполагает возможность хранения объектов внутри реляционной БД, тогда дополнительно необходимо предусмотреть хранение и использование специфических методов работы с этими объектами, а это в свою очередь требует расширения стандарта языка SQL. Частично это уже сделано в новом стандарте SQL3, однако там далеко не все вопросы получили однозначное разрешение.

Однако часть разработчиков придерживается мнения о необходимости полного отказа от реляционной парадигмы и перехода на объектно-ориентированную парадигму. Для перехода к объектно-ориентированным БД стандарт объектного проектирования был дополнен стандартизованными средствами доступа к базам данных (стандарт ODMG93).

Поставщики коммерческих СУБД немедленно отреагировали на эту потребность. Практически каждая уважающая себя фирма обратилась к объектным технологиям и продуктивно сотрудничает с разработчиками объектно-ориентированных СУБД. IBM и Oracle доработали свои СУБД (соответственно, DB2 и ORACLE), добавив объектную надстройку над реляционным ядром системы. Другой путь выбрал Informix, который приобрел серьезную объектно-реляционную СУБД Illustra и встроил ее в свою СУБД. В результате получился продукт, именующийся универсальным сервером. Другой лидер рынка СУБД — Computer Associates, поступил иначе. Он сделал ставку на чисто объектную базу Jasmine, активно пропагандируя ее достоинства. Кто прав — покажет будущее.

Следующим направлением развития баз данных является появление так называемых темпоральных баз данных, то есть баз данных, чувствительных ко времени. Фактически БД моделирует состояние объектов предметной области в некоторый текущий момент времени. Однако в ряде прикладных областей необходимо исследовать именно изменение состояний объектов во времени. Если использовать чисто реляционную модель, то требуется строить и хранить дополнительно множество отношений, имеющих одинаковые схемы, отличающиеся временем существования или снятия данных. Гораздо перспективнее и удобнее для этого использовать специальные механизмы снятия срезов по времени для определенных объектов БД. Основной тезис темпоральных систем состоит в том, что для любого объекта данных, созданного в момент времени t1 и уничтоженного в момент времени t2, в БД сохраняются (и доступны пользователям) все его состояния во временном интервале [t1, t2). При обозначении интервала квадратные скобки означают, что граница интервала включена в него, а круглые скобки означают, что точка на временной оси, соответствующая границе интервала, не включается в интервал. И действительно, если объект уничтожен в момент времени t2, то в этой точке временной оси он уже не существует, поэтому мы оставляем правую границу временного интервала открытой.

Еще одним из перспективных направлений развития баз данных является направление, связанное с объединением технологии экспертных систем и баз данных и развитие так называемых дедуктивных баз данных. Эти базы основаны на выявлении новых знаний из баз данных не путем запросов или аналитической обработки, а путем использования правил вывода и построения цепочек применения этих правил для вывода ответов на запросы. Для этих баз данных существуют языки запросов, отличные от классического SQL. В экспертных системах также знания экспертов хранятся в форме правил, чаще всего используются так называемые продукционные правила типа «если *описание ситуации*, то *описание действия*». Хранение подобных правил и организация вывода на основании имеющихся фактов под силу современным СУБД.

И наконец, последним, но, может быть, самым значительным направлением развития баз данных является перспектива взаимодействия Web-технологии и баз данных. Простота и доступность Web-технологии, возможность свободной публикации информации в Интернете, так чтобы она была доступна любому количеству пользователей, несомненно, сразу завоевали авторитет у большого числа пользователей. Однако процесс накопления слабоструктурированной информации быстро проходит и далее наступает момент обеспечения эффективного управления этой разнообразной информацией. И это уже серьезная проблема. Некоторые исследователи даже вывели определенную тенденцию, которая выражается в том, что наиболее популярные сайты со временем становятся неуправляемыми, в море информации невозможно отыскать то, что требуется. С одной стороны, Web представляет собой одну громадную базу данных. Однако до сих пор, вместо того чтобы превратиться в неотъемлемую часть инфраструктуры Web, базы данных остаются на вторых ролях. Во-первых, дизайнеры крупнейших Web-серверов с миллионами страниц содержимого постепенно перекладывают задачи управления страницами с файловых систем на системы баз данных. Во-вторых, системы баз данных используются в качестве серверов электронной коммерции, помогая отслеживать профили, транзакции, счета и инвентарные листы. В-третьих, ведущие Web-издатели примериваются к использованию систем баз данных для хранения информационного наполнения, имеющего сложную природу. Однако в подавляющей части Web-узлов, особенно в тех, которые принадлежат провайдерам и держателям поисковых машин, технология баз данных не применяется. В небольших Web-узлах, как правило, используются статические HTML-страницы, хранящиеся в обычных файловых системах.

В будущем статические HTML-страницы все чаще станут заменять системами управления динамически формируемым содержимым. Уже сейчас, например, торговцы по каталогам не просто преобразуют бумажные каталоги в наборы статических HTML-страниц. Фактически они представляют электронный каталог, позволяющий заказчикам оперативно узнать то, что их интересует, не пролис-

тывая ненужную информацию: например, продает ли поставщик серые джемперы большого размера. Продавцы предлагают клиентам персонализированные манекены, позволяющие увидеть, как будет сидеть на них одежда. Для персонализации требуются весьма сложные модели данных.

HTML расширяется до XML, языка расширяемой разметки, который лучше описывает структурированные данные. К сожалению, XML, похоже, способен породить хаос в системах баз данных. Развивающийся подъязык запросов XML напоминает процедурные языки обработки запросов, превалировавшие 25 лет тому назад. Кроме того, XML стимулирует использование кэшей (наборов) данных на стороне клиента с поддержкой обновлений, что заставляет разработчиков погружаться в трясину проблем распределенных транзакций. К несчастью, значительная часть работ по XML происходит без серьезного участия сообщества исследователей систем баз данных.

Авторы Web-публикаций нуждаются в инструментах для быстрого и экономичного построения хранилищ данных, рассчитанных на сложные приложения. Это, в свою очередь, формирует требования к технологии баз данных для создания, управления, поиска и обеспечения безопасности содержимого Web-узлов.

С другой стороны, универсальность Web-клиента становится весьма привлекательной для разработчиков несложных приложений, которые смогут работать с базами данных. В этом случае не требуется установка каждого клиента, достаточно выслать код доступа и клиент автоматически может уже работать с базой данных, при этом вам все равно, где находится клиент, он может работать как в локальной, так и в глобальной сети, если технология это позволяет. А это весьма удобно, если вы можете с любого рабочего места, имея соответствующий пароль, получить доступ к необходимым данным. Подобные системы называются системами, разработанными по интранет-технологии, то есть технологии, использующей принципы технологий Интернета, но реализованные во внутренней локальной сети.

Для разработки интернет-приложений, которые связаны с базами данных, широко используются новые средства программирования: это язык PERL, язык PHP (Personal Home Page Tools), язык Javascript и ряд других. Это действительно грандиозно и, главное, очень интересно, но это уже темы для других книг. Пробуйте и дерзайте, я думаю, познакомившись с базами данных, вы еще не раз с ними столкнетесь в жизни. Я желаю вам успехов и корректных запросов к базам данных. Вы ведь уже знаете: каков вопрос, таков и ответ. Любая база данных может стать вашим помощником или мучителем, это зависит от разработчиков, мне хочется, чтобы для вас они всегда играли только первую роль.

Алфавитный указатель

Карпова Татьяна Сергеевна

Базы данных: модели, разработка, реализация

Главный редактор *В. Усманов*
Ответственный редактор *Е. Строганова*
Руководитель проекта *И. Корнеев*
Литературный редактор *Н. Дубнова*
Художник *Н. Биржаков*
Верстка *Ю. Сергиенко*
Корректоры *Д. Благов, Т. Брылева*

Лицензия ИД № 01940 от 05.06.00.

Подписано в печать 29.06.01. Формат 70×100$^{1}/_{16}$.
Усл. п. л. 24,51. Доп. тираж 5000 экз. Заказ № 1028.

ЗАО «Питер Бук».
196105, Санкт-Петербург, ул. Благодатная, 67.

Налоговая льгота — общероссийский классификатор продукции ОК 005-93,
том 2; 953000 — книги и брошюры.

Отпечатано с фотоформ в ФГУП «Печатный двор» им. А. М. Горького Министерства РФ
по делам печати, телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.