Основы современных баз данных

Введение

§1. Базы данных и файловые системы.

На первой лекции мы рассмотрим общий смысл понятий БД и СУБД. Начнем с того, что с самого начала развития вычислительной техники образовались два основных направления ее использования. Первое направление - применение вычислительной техники для выполнения численных расчетов, которые слишком долго или вообще невозможно производить вручную. Становление этого направления способствовало интенсификации методов численного решения сложных математических задач, развитию класса языков программирования, ориентированных на удобную запись численных алгоритмов, становлению обратной связи с разработчиками новых архитектур ЭВМ.

Второе направление, которое непосредственно касается темы нашего курса, это использование средств вычислительной техники в автоматических или автоматизированных информационных системах. В самом широком смысле информационная система представляет собой программный комплекс, функции которого состоят в поддержке надежного хранения информации в памяти компьютера, выполнении специфических для данного приложения преобразований информации и/или вычислений, предоставлении пользователям удобного и легко осваиваемого интерфейса. Обычно объемы информации, с которыми приходится иметь дело таким системам, достаточно велики, а сама информация имеет достаточно сложную структуру. Классическими примерами информационных систем являются банковские системы, системы резервирования авиационных или железнодорожных билетов, мест в гостиницах и т.д.

На самом деле, второе направление возникло несколько позже первого. Это связано с тем, что на заре вычислительной техники компьютеры обладали ограниченными возможностями в части памяти. Понятно, что можно говорить о надежном и долговременном хранении информации только при наличии запоминающих устройств, сохраняющих информацию после выключения электрического питания. Оперативная память этим свойством обычно не обладает. В начале использовались два вида устройств внешней памяти: магнитные ленты и барабаны. При этом емкость магнитных лент была достаточно велика, но по своей физической природе они обеспечивали последовательный доступ к данным. Магнитные же барабаны (они больше всего похожи на современные магнитные диски с фиксированными головками) давали возможность произвольного доступа к данными, но были ограниченного размера.

Легко видеть, что указанные ограничения не очень существенны для чисто численных расчетов. Даже если программа должна обработать (или произвести) большой объем информации, при программировании можно продумать расположение этой информации во внешней памяти, чтобы программа работала как можно быстрее.

С другой стороны, для информационных систем, в которых потребность в текущих данных определяется пользователем, наличие только магнитных лент и барабанов неудовлетворительно. Представьте себе покупателя билета, который стоя у кассы должен дождаться полной перемотки магнитной ленты. Одним из естественных требований к таким системам является средняя быстрота выполнения операций.

Как кажется, именно требования к вычислительной технике со стороны нечисленных 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 внеш-

ней памяти, сама определяла расположение каждой порции данных на магнитной ленте или барабане и выполняла обмены между оперативной и внешней памятью 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.

1.1. Файловые системы.

Историческим шагом 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, возможно, 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.

Первая развитая файловая система была разработана фирмой IBM для ее серии 360. К настоящему времени она очень устарела, и мы не будем рассматривать ее подробно. Заметим лишь, что в этой системе поддерживались как чисто последовательные, так и индексно-последовательные файлы, а реализация во многом опиралась на возможности только появившихся к этому времени контроллеров управления дисковыми устройствами. Если учесть к тому же, что понятие файла в OS/360 было выбрано как основное абстрактное понятие, которому соответствовал любой внешний объект, включая внешние устройства, то работать с файлами на уровне пользователя было очень неудобно. Требовался целый ряд громоздких и перегруженных деталями конструкций. Все это хорошо знакомо программистам среднего и старшего поколения, которые прошли через использование отечественных аналогов компьютеров IBM.

1.1.1. Структуры файлов.

Дальше мы будем говорить о более современных организациях файловых систем. Начнем со структур файлов. Прежде всего, практически во всех современных компьютерах основными устройствами внешней памяти являются магнитные диски с подвижными головками, и именно они служат для хранения файлов. Такие магнитные диски представляют собой пакеты магнитных пластин (поверхностей), между которыми на одном рычаге двигается пакет магнитных головок. Шаг движения пакета головок является дискретным, и каждому положению пакета головок логически соответствует цилиндр магнитного диска. На каждой поверхности цилиндр "высекает" дорожку, так что каждая поверхность содержит число дорожек, равное числу цилиндров. При разметке магнитного диска (специальном действии, предшествующем использованию диска) каждая дорожка размечается на одно и то же количество блоков таким образом, что в каждый блок можно записать по максимуму одно и то же число байтов. Таким образом, для произведения обмена с магнитным диском на уровне аппаратуры нужно указать номер цилиндра, номер поверхности, номер блока на соответствующей дорожке и число байтов, которое нужно записать или прочитать от начала этого блока.

Однако эта возможность обмениваться с магнитными дисками порциями меньше объема блока в настоящее время не используется в файловых системах. Это связано с двумя обстоятельствами. Во-первых, при выполнении обмена с диском аппаратура выполняет три основных действия: подвод головок к нужному цилиндру, поиск на дорожке нужного блока и собственно обмен с этим блоком. Из всех этих действий в среднем наибольшее время занимает первое. Поэтому существенный выигрыш в суммарном времени обмена за счет считывания или записывания только части блока получить практически невозможно. Во-вторых, для того, чтобы работать с частями блоков, файловая система должна обеспечить соответствующего размера буфера оперативной памяти, что существенно усложняет распределение оперативной памяти.

Поэтому во всех файловых системах явно или неявно выделяется некоторый базовый уровень, обеспечивающий работу с файлами, представляющими набор прямо адресуемых в адресном пространстве файла блоков. Размер этих логических блоков файла совпадает или кратен

размеру физического блока диска и обычно выбирается равным размеру страницы виртуальной памяти, поддерживаемой аппаратурой компьютера совместно с операционной системой.

В некоторых файловых системах базовый уровень доступен пользователю, но более часто прикрывается некоторым более высоким уровнем, стандартным для пользователей. Распространены два основных подхода. При первом подходе, свойственном, например, файловым системам операционных систем фирмы DEC RSX и VMS, пользователи представляют файл как последовательность записей. Каждая запись - это последовательность байтов постоянного или переменного размера. Записи можно читать или записывать последовательно или позиционировать файл на запись с указанным номером. Некоторые файловые системы позволяют структурировать записи на поля и объявлять некоторые поля ключами записи. В таких файловых системах можно потребовать выборку записи из файла по ее заданному ключу. Естественно, что в этом случае файловая система поддерживает в том же (или другом, служебном) базовом файле дополнительные, невидимые пользователю, служебные структуры данных. Распространенные способы организации ключевых файлов основываются на технике хэширования и B -деревьев (мы будем говорить об этих приемах более подробно в следующих лекциях). Существуют и многоключевые способы организации файлов.

Второй подход, ставший распространенным вместе с операционной системой UNIX, состоит в том, что любой файл представляется как последовательность байтов. Из файла можно прочитать указанное число байтов либо начиная с его начала, либо предварительно произведя его позиционирование на байт с указанным номером. Аналогично, можно записать указанное число байтов в конец файла, либо предварительно произведя позиционирование файла. Заметим, что тем не менее скрытым от пользователя, но существующим во всех разновидностях файловых систем ОС UNIX, является базовое блочное представление файла.

Конечно, для обоих подходов можно обеспечить набор преобразующих функций, приводящих представление файла к некоторому другому виду. Примером тому служит поддержание стандартной файловой среды системы программирования на языке Си в среде операционных систем фирмы DEC.

1.1.2. Именование файлов.

Остановимся коротко на способах именования файлов. Все современные файловые системы поддерживают многоуровневое именование файлов за счет поддержания во внешней памяти дополнительных файлов со специальной структурой - каталогов. Каждый каталог содержит имена каталогов и/или файлов, содержащихся в данном каталоге. Таким образом, полное имя файла состоит из списка имен каталогов плюс имя файла в каталоге, непосредственно содержащем данный файл. Разница между способами именования файлов в разных файловых системах состоит в том, с чего начинается эта цепочка имен.

В этом отношении имеются два крайних варианта. Во многих системах управления файлами требуется, чтобы каждый архив файлов (полное дерево справочников) целиком располагался на одном дисковом пакете (или логическом диске, разделе физического дискового пакета, представляемом с помощью средств операционной системы как отдельный диск). В этом случае полное имя файла начинается с имени дискового устройства, на котором установлен соответствующий диск. Такой способ именования используется в файловых системах фирмы DEC, очень близко к этому находятся и файловые системы персональных компьютеров. Можно назвать эту организацию поддержанием изолированных файловых систем.

Другой крайний вариант был реализован в файловых системах операционной системы Multics. Эта система заслуживает отдельного большого разговора, в ней был реализован целый ряд оригинальных идей, но мы остановимся только на особенностях организации архива файлов. В файловой системе Multics пользователи представляли всю совокупность каталогов и файлов как единое дерево. Полное имя файла начиналось с имени корневого каталога, и пользователь не обязан был заботиться об установке на дисковое устройство каких-либо конкретных дисков. Сама система, выполняя поиск файла по его имени, запрашивала установку необходимых дисков. Такую файловую систему можно назвать полностью централизованной.

Конечно, во многом централизованные файловые системы удобнее изолированных: система управления файлами принимает на себя больше рутинной работы. Но в таких системах возникают существенные проблемы, если кому-то требуется перенести поддерево файловой системы на другую вычислительную установку. Компромиссное решение применено в файловых системах ОС UNIX. На базовом уровне в этих файловых системах поддерживаются изолированные архивы файлов. Один из этих архивов объявляется корневой файловой системой. После запуска системы можно "смонтировать" корневую файловую систему и ряд изолированных файловых систем в одну общую файловую систему. Технически это производится с помощью заведения в корневой файловой системе специальных пустых каталогов. Специальный системный вызов курьер ОС UNIX позволяет подключить к одному из этих пустых каталогов корневой каталог указанного архива файлов. После монтирования общей файловой системы именование файлов производится так же, как если бы она с самого начала была централизованной. Если учесть, что обычно монтирование файловой системы производится при раскрутке системы, то пользователи ОС UNIX обычно и не задумываются об исходном происхождении

1.1.3. Защита файлов.

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. Существовали попытки реализовать этот подход в полном объеме. Но это вызывало слишком большие накладные расходы как по хранению избыточной информации, так и по использованию этой информации для контроля правомочности доступа.

Поэтому в большинстве современных систем управления файлами применяется подход к защите файлов, впервые реализованный в ОС UNIX. В этой системе каждому зарегистрированному пользователю соответствует пара целочисленных идентификаторов: идентификатор группы, к которой относится этот пользователь, и его собственный идентификатор в группе. Соответственно, при каждом файле хранится полный идентификатор пользователя, который создал этот файл, и отмечается, какие действия с файлом может производить он сам, какие действия с файлом доступны для других пользователей той же группы, и что могут делать с файлом пользователи других групп. Эта информация очень компактна, при проверке требуется небольшое количество действий, и этот способ контроля доступа удовлетворителен в большинстве случаев.

1.1.4. Режим многопользовательского доступа.

Последнее, на чем мы остановимся в связи с файлами, - это способы их использования в многопользовательской среде. Если операционная система поддерживает многопользовательский режим, вполне реальна ситуация, когда два или более пользователей одновременно пытаются работать с одним и тем же файлом. Если все эти пользователи собираются только читать файл, ничего страшного не произойдет. Но если хотя бы один из них будет изменять файл, для корректной работы этой группы требуется взаимная синхронизация.

Исторически в файловых системах применялся следующий подход. В операции открытия файла (первой и обязательной операции, с которой должен начинаться сеанс работы с файлом) помимо прочих параметров указывался режим работы (чтение или изменение). Если к моменту выполнения этой операции от имени некоторой программы A файл уже находился в открытом состоянии от имени некоторой другой программы B (правильнее говорить "процесса", но мы не будем вдаваться в терминологические тонкости), причем существующий режим открытия был несовместимым с желаемым режимом (совместимы только режимы чтения), то в зависимости от особенностей системы программе A либо сообщалось о невозможности открытия файла в желаемом режиме, либо она блокировалась до тех пор, пока программа B не выполнит операцию закрытия файла.

Заметим, что в ранних версиях файловой системы ОС UNIX вообще не были реализованы какие бы то ни было средства синхронизации параллельного доступа к файлам. Операция открытия файла выполнялась всегда для любого существующего файла, если данный пользователь

имел соответствующие права доступа. При совместной работе синхронизацию следовало производить вне файловой системы (и особых средств для этого ОС UNIX не предоставляла). В современных реализациях файловых систем ОС UNIX по желанию пользователя поддерживается синхронизация при открытии файлов. Кроме того, существует возможность синхронизации нескольких процессов, параллельно модифицирующих один и тот же файл. Для этого введен специальный механизм синхронизационных захватов диапазонов адресов открытого файла.

1.2. Области применения файлов.

После этого краткого экскурса в историю файловых систем рассмотрим возможные области их применения. Прежде всего, конечно, файлы применяются для хранения текстовых данных: документов, текстов программ и т.д. Такие файлы обычно образуются и модифицируются с помощью различных текстовых редакторов. Структура текстовых файлов обычно очень проста: это либо последовательность записей, содержащих строки текста, либо последовательность байтов, среди которых встречаются специальные символы (например, символы конца строки).

Файлы с текстами программ используются как входные тексты компиляторов, которые в свою очередь формируют файлы, содержащие объектные модули. С точки зрения файловой системы, объектные файлы также обладают очень простой структурой - последовательность записей или байтов. Система программирования накладывает на эту структуру более сложную и специфичную для этой системы структуру объектного модуля. Подчеркнем, что логическая структура объектного модуля неизвестна файловой системе, эта структура поддерживается программами системы программирования.

Аналогично обстоит дело с файлами, формируемыми редакторами связей и содержащими образы выполняемых программ. Логическая структура таких файлов остается известной только редактору связей и загрузчику - программе операционной системы. Примерно такая же ситуация с файлами, содержащими графическую и звуковую информацию.

Одним словом, файловые системы обычно обеспечивают хранение слабо структурированной информации, оставляя дальнейшую структуризацию прикладным программам. В перечисленных выше случаях использования файлов это даже хорошо, потому что при разработке любой новой прикладной системы опираясь на простые, стандартные и сравнительно дешевые средства файловой системы можно реализовать те структуры хранения, которые наиболее естественно соответствуют специфике данной прикладной области.

1.3. Потребности информационных систем.

Однако ситуация коренным образом отличается для упоминавшихся в начале лекции информационных систем. Эти системы главным образом ориентированы на хранение, выбор и модификацию постоянно существующей информации. Структура информации зачастую очень сложна, и хотя структуры данных различны в разных информационных системах, между ними часто бывает много общего. На начальном этапе использования вычислительной техники для управления информацией проблемы структуризации данных решались индивидуально в каждой информационной системе. Производились необходимые надстройки над файловыми системами (библиотеки программ), подобно тому, как это делается в компиляторах, редакторах и т.д.

Но поскольку информационные системы требуют сложных структур данных, эти дополнительные индивидуальные средства управления данными являлись существенной частью информационных систем и практически повторялись от одной системы к другой. Стремление выделить и обобщить общую часть информационных систем, ответственную за управление сложно структурированными данными, явилось, на наш взгляд, первой побудительной причиной создания СУБД. Очень скоро стало понятно, что невозможно обойтись общей библиотекой программ, реализующей над стандартной базовой файловой системой более сложные методы хранения данных.

Покажем это на примере. Предположим, что мы хотим реализовать простую информационную систему, поддерживающую учет сотрудников некоторой организации. Система должна выполнять следующие действия: выдавать списки сотрудников по отделам, поддерживать возможность перевода сотрудника из одного отдела в другой, приема на работу новых сотрудников и увольнения работающих. Для каждого отдела должна поддерживаться возможность получения

имени руководителя этого отдела, общей численности отдела, общей суммы выплаченной в последний раз зарплаты и т.д. Для каждого сотрудника должна поддерживаться возможность выдачи номера удостоверения по полному имени сотрудника, выдачи полного имени по номеру удостоверения, получения информации о текущем соответствии занимаемой должности сотрудника и о размере его зарплаты.

Предположим, что мы решили основывать эту информационную систему на файловой системе и пользоваться при этом одним файлом, расширив базовые возможности файловой системы за счет специальной библиотеки функций. Поскольку минимальной информационной единицей в нашем случае является сотрудник, естественно потребовать, чтобы в этом файле содержалась одна запись для каждого сотрудника. Какие поля должна содержать такая запись? Полное имя сотрудника (*СОТР _ ИМЯ*), номер его удостоверения (*СОТР _ НОМЕР*), информацию о его соответствии занимаемой должности (для простоты, "да" или "нет") (*СОТР _ СТАТ*), размер зарплаты (*СОТР _ ЗАРП*), номер отдела (*СОТР _ ОТД _ НОМЕР*). Поскольку мы хотим ограничиться одним файлом, та же запись должна содержать имя руководителя отдела (*СОТР _ ОТД _ РУК*).

Функции нашей информационной системы требуют, чтобы обеспечивалась возможность многоключевого доступа к этому файлу по уникальным ключам (недублируемым в разных записях) *СОТР _ ИМЯ* и *СОТР _ НОМЕР* . Кроме того, должна обеспечиваться возможность выбора всех записей с общем значением *СОТР _ ОТД _ НОМЕР* , то есть доступ по неуникальному ключу. Для того, чтобы получить численность отдела или общий размер зарплаты, каждый раз при выполнении такой функции информационная система должна будет выбрать все записи о сотрудниках отдела и посчитать соответствующие общие значения.

Таким образом мы видим, что даже для такой простой системы ее реализация на базе файловой системы, во-первых, требует создания достаточно сложной надстройки для многоключевого доступа к файлам, и, во-вторых, вызывает требование существенной избыточности хранения (для каждого сотрудника одного отдела повторяется имя руководителя) и выполнение массовой выборки и вычислений для получения суммарной информации об отделах. Кроме того, если в ходе эксплуатации системы нам захочется, например, выдавать списки сотрудников, получающих заданную зарплату, то придется либо полностью просматривать файл, либо реструктуризовывать его, объявляя ключевым поле *СОТР _ ЗАРП* .

Первое, что приходит на ум, - это поддерживать два многоключевых файла: *СОТРУДНИКИ* и *ОТДЕЛЫ* . Первый файл должен содержать поля *СОТР _ ИМЯ* , *СОТР _ НОМЕР* , *СОТР _ СТАТ* , *СОТР _ ЗАРП* и *СОТР _ ОТД _ НОМЕР* , а второй - *ОТД _ НОМЕР* , *ОТД _ РУК* , *ОТД _ СОТР _ ЗАРП* (общий размер зарплаты) и *ОТД _ РАЗМЕР* (общее число сотрудников в отделе). Большинство неудобств, перечисленных в предыдущем абзаце, будут преодолены. Каждый из файлов будет содержать только недублируемую информацию, необходимости в динамических вычислениях суммарной информации не возникает. Но заметим, что при таком переходе наша информационная система должна обладать некоторыми новыми особенностями, сближающими ее с СУБД.

Прежде всего, система должна теперь знать, что она работает с двумя информационно связанными файлами (это шаг в сторону схемы базы данных), должна знать структуру и смысл каждого поля (например, что *СОТР _ ОТД _ НОМЕР* в файле *СОТРУДНИКИ* и *ОТД _ НОМЕР* в файле *ОТДЕЛЫ* означают одно и то же), а также понимать, что в ряде случаев изменение информации в одном файле должно автоматически вызывать модификацию во втором файле, чтобы их общее содержимое было согласованным. Например, если на работу принимается новый сотрудник, то необходимо добавить запись в файл *СОТРУДНИКИ* , а также соответствующим образом изменить поля *ОТД _ ЗАРП* и *ОТД _ РАЗМЕР* в записи файла *ОТДЕЛЫ* , описывающей отдел этого сотрудника.

Понятие ***согласованности данных*** является ключевым понятием баз данных. Фактически, если информационная система (даже такая простая, как в нашем примере) поддерживает согла-

сованное хранение информации в нескольких файлах, можно говорить о том, что она поддерживает базу данных. Если же некоторая вспомогательная система управления данными позволяет работать с несколькими файлами, обеспечивая их согласованность, можно назвать ее системой управления базами данных. Уже только требование поддержания согласованности данных в нескольких файлах не позволяет обойтись библиотекой функций: такая система должна иметь некоторые собственные данные (метаданные) и даже знания, определяющие целостность данных.

Но это еще не все, что обычно требуют от СУБД. Во-первых, даже в нашем примере неудобно реализовывать такие запросы как "выдать общую численность отдела, в котором работает Петр Иванович Сидоров". Было бы гораздо проще, если бы СУБД позволяла сформулировать такой запрос на близком пользователям языке. Такие языки называются ***языками запросов к базам данных***. Например, на языке SQL наш запрос можно было бы выразить в форме:

```
SELECT ОТД_РАЗМЕР
FROM СОТРУДНИКИ, ОТДЕЛЫ
WHERE СОТР_ИМЯ = "ПЕТР ИВАНОВИЧ СИДОРОВ"
AND СОТР_ОТД_НОМЕР = ОТД_НОМЕР
```

Таким образом, при формулировании запроса СУБД позволит не задумываться о том, как будет выполняться этот запрос. Среди ее метаданных будет содержаться информация о том, что поле *СОТР _ ИМЯ* является ключевым для файла *СОТРУДНИКИ* , а *ОТД _ НОМЕР* - для файла *ОТДЕЛЫ* , и система сама воспользуется этим. Если же возникнет потребность в получении списка сотрудников, не соответствующих занимаемой должности, то достаточно предъявить системе запрос:

```
SELECT СОТР_ИМЯ, СОТР_НОМЕР
FROM СОТРУДНИКИ
WHERE СОТР_СТАТ = "НЕТ"
```

и система сама выполнит необходимый полный просмотр файла *СОТРУДНИКИ* , поскольку поле *СОТР _ СТАТ* не является ключевым.

Далее, представьте себе, что в нашей первоначальной реализации информационной системы, основанной на использовании библиотек расширенных методов доступа к файлам, обрабатывается операция регистрации нового сотрудника. Следуя требованиям согласованного изменения файлов, информационная система вставила новую запись в файл *СОТРУДНИКИ* и собиралась модифицировать запись файла *ОТДЕЛЫ* , но именно в этот момент произошло аварийное выключение питания. Очевидно, что после перезапуска системы ее база данных будет находиться в рассогласованном состоянии. Потребуется выяснить это (а для этого нужно явно проверить соответствие информации в файлах *СОТРУДНИКИ* и *ОТДЕЛЫ*) и привести информацию в согласованное состояние. Настоящие СУБД берут такую работу на себя. Прикладная система не обязана заботиться о корректности состояния базы данных.

Наконец, представим себе, что мы хотим обеспечить параллельную (например, многотерминальную) работу с базой данных сотрудников. Если опираться только на использование файлов, то для обеспечения корректности на все время модификации любого из двух файлов доступ других пользователей к этому файлу будет блокирован (вспомните возможности файловых систем для синхронизации параллельного доступа). Таким образом, зачисление на работу Петра Ивановича Сидорова существенно затормозит получение информации о сотруднике Иване Сидоровиче Петрове, даже если они будут работать в разных отделах. Настоящие СУБД обеспечивают гораздо более тонкую синхронизацию параллельного доступа к данным.

Таким образом, СУБД решают множество проблем, которые затруднительно или вообще невозможно решить при использовании файловых систем. При этом существуют приложения, для которых вполне достаточно файлов; приложения, для которых необходимо решать, какой уровень работы с данными во внешней памяти для них требуется, и приложения, для которых безусловно нужны базы данных.

§2. Функции СУБД. Типовая организация СУБД. Примеры.

Как было показано в первой лекции, традиционных возможностей файловых систем оказывается недостаточно для построения даже простых информационных систем. Мы выявили несколько потребностей, которые не покрываются возможностями систем управления файлами: поддержание логически согласованного набора файлов; обеспечение языка манипулирования данными; восстановление информации после разного рода сбоев; реально параллельная работа нескольких пользователей. Можно считать, что если прикладная информационная система опирается на некоторую систему управления данными, обладающую этими свойствами, то эта система управления данными является ***системой управления базами данных (СУБД)***.

2.1. Основные функции СУБД.

Более точно, к числу функций СУБД принято относить следующие.

2.1.1. Непосредственное управление данными во внешней памяти.

Эта функция включает обеспечение необходимых структур внешней памяти как для хранения данных, непосредственно входящих в БД, так и для служебных целей, например, для убыстрения доступа к данным в некоторых случаях (обычно для этого используются индексы). 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.

2.1.2. Управление буферами оперативной памяти.

СУБД обычно работают с БД значительного размера, по крайней мере, этот размер обычно существенно больше доступного объема оперативной памяти. Понятно, что если при обращении к любому элементу данных будет производиться обмен с внешней памятью, то вся система будет работать со скоростью устройства внешней памяти. Практически единственным способом реального увеличения этой скорости является буферизация данных в оперативной памяти. При этом, даже если операционная система производит общесистемную буферизацию (как в случае ОС UNIX), этого недостаточно для целей СУБД, которая располагает гораздо большей информацией о полезности буферизации той или иной части БД. Поэтому в развитых СУБД поддерживается собственный набор буферов оперативной памяти с собственной дисциплиной замены буферов.

Заметим, что существует отдельное направление СУБД, которое ориентировано на постоянное присутствие в оперативной памяти всей БД. Это направление основывается на предположении, что в будущем объем оперативной памяти компьютеров будет настолько велик, что позволит не беспокоиться о буферизации. Пока эти работы находятся в стадии исследований.

2.1.3. Управление транзакциями.

Транзакция - это последовательность операций над БД, рассматриваемых СУБД как единое целое. Либо транзакция успешно выполняется, и СУБД фиксирует (*COMMIT*) изменения БД, произведенные этой транзакцией, во внешней памяти, либо ни одно из этих изменений никак не отражается на состоянии БД. Понятие транзакции необходимо для поддержания логической целостности БД. Если вспомнить наш пример информационной системы с файлами *СОТРУДНИКИ* и *ОТДЕЛЫ*, то единственным способом не нарушить целостность БД при выполнении операции приема на работу нового сотрудника является объединение элементарных операций над файлами *СОТРУДНИКИ* и *ОТДЕЛЫ* в одну транзакцию. Таким образом, поддержание механизма транзакций является обязательным условием даже однопользовательских СУБД (если, конечно, такая система заслуживает названия СУБД). Но понятие транзакции гораздо более важно в многопользовательских СУБД.

То свойство, что каждая транзакция начинается при целостном состоянии БД и оставляет это состояние целостным после своего завершения, делает очень удобным использование понятия транзакции как единицы активности пользователя по отношению к БД. При соответствующем

управлении параллельно выполняющимися транзакциями со стороны СУБД каждый из пользователей может в принципе ощущать себя единственным пользователем СУБД (на самом деле, это несколько идеализированное представление, поскольку в некоторых случаях пользователи многопользовательских СУБД могут ощутить присутствие своих коллег).

С управлением транзакциями в многопользовательской СУБД связаны важные понятия ***сериализации транзакций*** и ***сериального плана выполнения смеси транзакций***. Под сериализаций параллельно выполняющихся транзакций понимается такой порядок планирования их работы, при котором суммарный эффект смеси транзакций эквивалентен эффекту их некоторого последовательного выполнения. Сериальный план выполнения смеси транзакций - это такой план, который приводит к сериализации транзакций. Понятно, что если удается добиться действительно сериального выполнения смеси транзакций, то для каждого пользователя, по инициативе которого образована транзакция, присутствие других транзакций будет незаметно (если не считать некоторого замедления работы по сравнению с однопользовательским режимом).

Существует несколько базовых алгоритмов сериализации транзакций. В централизованных СУБД наиболее распространены алгоритмы, основанные на синхронизационных захватах объектов БД. При использовании любого алгоритма сериализации возможны ситуации конфликтов между двумя или более транзакциями по доступу к объектам БД. В этом случае для поддержания сериализации необходимо выполнить откат (ликвидировать все изменения, произведенные в БД) одной или более транзакций. Это один из случаев, когда пользователь многопользовательской СУБД может реально (и достаточно неприятно) ощутить присутствие в системе транзакций других пользователей.

2.1.4. Журнализация.

Одним из основных требований к СУБД является надежность хранения данных во внешней памяти. Под ***надежностью хранения*** понимается то, что СУБД должна быть в состоянии восстановить последнее согласованное состояние БД после любого аппаратного или программного сбоя. Обычно рассматриваются два возможных вида аппаратных сбоев: так называемые мягкие сбои, которые можно трактовать как внезапную остановку работы компьютера (например, аварийное выключение питания), и жесткие сбои, характеризуемые потерей информации на носителях внешней памяти. Примерами программных сбоев могут быть: аварийное завершение работы СУБД (по причине ошибки в программе или в результате некоторого аппаратного сбоя) или аварийное завершение пользовательской программы, в результате чего некоторая транзакция остается незавершенной. Первую ситуацию можно рассматривать как особый вид мягкого аппаратного сбоя; при возникновении последней требуется ликвидировать последствия только одной транзакции.

Понятно, что в любом случае для восстановления БД нужно располагать некоторой дополнительной информацией. Другими словами, поддержание надежности хранения данных в БД требует избыточности хранения данных, причем та часть данных, которая используется для восстановления, должна храниться особо надежно. Наиболее распространенным методом поддержания такой избыточной информации является ведение журнала изменений БД.

Журнал - это особая часть БД, недоступная пользователям СУБД и поддерживаемая с особой тщательностью (иногда поддерживаются две копии журнала, располагаемые на разных физических дисках), в которую поступают записи обо всех изменениях основной части БД. В разных СУБД изменения БД журнализуются на разных уровнях: иногда запись в журнале соответствует некоторой логической операции изменения БД (например, операции удаления строки из таблицы реляционной БД), иногда - минимальной внутренней операции модификации страницы внешней памяти; в некоторых системах одновременно используются оба подхода.

Во всех случаях придерживаются стратегии "упреждающей" записи в журнал (так называемого протокола Write Ahead Log - WAL). Грубо говоря, эта стратегия заключается в том, что запись об изменении любого объекта БД должна попасть во внешнюю память журнала раньше, чем измененный объект попадет во внешнюю память основной части БД. Известно, что если в СУБД корректно соблюдается протокол WAL, то с помощью журнала можно решить все проблемы восстановления БД после любого сбоя.

Самая простая ситуация восстановления - индивидуальный откат транзакции. Строго говоря, для этого не требуется общесистемный журнал изменений БД. Достаточно для каждой транзакции поддерживать локальный журнал операций модификации БД, выполненных в этой транзакции, и производить откат транзакции путем выполнения обратных операций, следуя от конца локального журнала. В некоторых СУБД так и делают, но в большинстве систем локальные журналы не поддерживают, а индивидуальный откат транзакции выполняют по общесистемному журналу, для чего все записи от одной транзакции связывают обратным списком (от конца к началу).

При мягком сбое во внешней памяти основной части БД могут находиться объекты, модифицированные транзакциями, не закончившимися к моменту сбоя, и могут отсутствовать объекты, модифицированные транзакциями, которые к моменту сбоя успешно завершились (по причине использования буферов оперативной памяти, содержимое которых при мягком сбое пропадает). При соблюдении протокола WAL во внешней памяти журнала должны гарантированно находиться записи, относящиеся к операциям модификации обоих видов объектов. Целью процесса восстановления после мягкого сбоя является состояние внешней памяти основной части БД, которое возникло бы при фиксации во внешней памяти изменений всех завершившихся транзакций и которое не содержало бы никаких следов незаконченных транзакций. Для того, чтобы этого добиться, сначала производят откат незавершенных транзакций (undo), а потом повторно воспроизводят (redo) те операции завершенных транзакций, результаты которых не отображены во внешней памяти. Этот процесс содержит много тонкостей, связанных с общей организацией управления буферами и журналом. Более подробно мы рассмотрим это в соответствующей лекции.

Для восстановления БД после жесткого сбоя используют журнал и архивную копию БД. Грубо говоря, архивная копия - это полная копия БД к моменту начала заполнения журнала (имеется много вариантов более гибкой трактовки смысла архивной копии). Конечно, для нормального восстановления БД после жесткого сбоя необходимо, чтобы журнал не пропал. Как уже отмечалось, к сохранности журнала во внешней памяти в СУБД предъявляются особо повышенные требования. Тогда восстановление БД состоит в том, что исходя из архивной копии по журналу воспроизводится работа всех транзакций, которые закончились к моменту сбоя. В принципе, можно даже воспроизвести работу незавершенных транзакций и продолжить их работу после завершения восстановления. Однако в реальных системах это обычно не делается, поскольку процесс восстановления после жесткого сбоя является достаточно длительным.

2.1.5. Поддержка языков БД.

Для работы с базами данных используются специальные языки, в целом называемые ***языками баз данных***. В ранних СУБД поддерживалось несколько специализированных по своим функциям языков. Чаще всего выделялись два языка - ***язык определения схемы БД (SDL - Schema Definition Language)*** и ***язык манипулирования данными (DML - Data Manipulation Language)***. SDL служил главным образом для определения логической структуры БД, т.е. той структуры БД, какой она представляется пользователям. DML содержал набор операторов манипулирования данными, т.е. операторов, позволяющих заносить данные в БД, удалять, модифицировать или выбирать существующие данные. Мы рассмотрим более подробно языки ранних СУБД в следующей лекции.

В современных СУБД обычно поддерживается единый интегрированный язык, содержащий все необходимые средства для работы с БД, начиная от ее создания, и обеспечивающий базовый пользовательский интерфейс с базами данных. Стандартным языком наиболее распространенных в настоящее время реляционных СУБД является язык SQL (Structured Query Language). В нескольких лекциях этого курса язык SQL будет рассматриваться достаточно подробно, а пока мы перечислим основные функции реляционной СУБД, поддерживаемые на "языковом" уровне (т.е. функции, поддерживаемые при реализации интерфейса SQL).

Прежде всего, язык SQL сочетает средства SDL и DML, т.е. позволяет определять схему реляционной БД и манипулировать данными. При этом именование объектов БД (для реляционной БД - именование таблиц и их столбцов) поддерживается на языковом уровне в том смысле,

что компилятор языка SQL производит преобразование имен объектов в их внутренние идентификаторы на основании специально поддерживаемых служебных таблиц-каталогов. Внутренняя часть СУБД (ядро) вообще не работает с именами таблиц и их столбцов.

Язык SQL содержит специальные средства определения ограничений целостности БД. Опять же, ограничения целостности хранятся в специальных таблицах-каталогах, и обеспечение контроля целостности БД производится на языковом уровне, т.е. при компиляции операторов модификации БД компилятор SQL на основании имеющихся в БД ограничений целостности генерирует соответствующий программный код.

Специальные операторы языка SQL позволяют определять так называемые представления БД, фактически являющиеся хранимыми в БД запросами (результатом любого запроса к реляционной БД является таблица) с именованными столбцами. Для пользователя представление является такой же таблицей, как любая базовая таблица, хранимая в БД, но с помощью представлений можно ограничить или наоборот расширить видимость БД для конкретного пользователя. Поддержание представлений производится также на языковом уровне.

Наконец, авторизация доступа к объектам БД производится также на основе специального набора операторов SQL. Идея состоит в том, что для выполнения операторов SQL разного вида пользователь должен обладать различными полномочиями. Пользователь, создавший таблицу БД, обладает полным набором полномочий для работы с этой таблицей. В число этих полномочий входит полномочие на передачу всех или части полномочий другим пользователям, включая полномочие на передачу полномочий. Полномочия пользователей описываются в специальных таблицах-каталогах, контроль полномочий поддерживается на языковом уровне.

Более точное описание возможных реализаций этих функций на основе языка SQL будет приведено в лекциях, посвященных языку SQL и его реализации.

2.2. Типовая организация современной СУБД.

Естественно, организация типичной СУБД и состав ее компонентов соответствует рассмотренному нами набору функций. Напомним, что мы выделили следующие основные функции СУБД:

- управление данными во внешней памяти;
- управление буферами оперативной памяти;
- управление транзакциями;
- журнализация и восстановление БД после сбоев;
- поддержание языков БД.

Логически в современной реляционной СУБД можно выделить наиболее внутреннюю часть - ядро СУБД (часто его называют Data Base Engine), компилятор языка БД (обычно SQL), подсистему поддержки времени выполнения, набор утилит. В некоторых системах эти части выделяются явно, в других - нет, но логически такое разделение можно провести во всех СУБД.

Ядро СУБД отвечает за управление данными во внешней памяти, управление буферами оперативной памяти, управление транзакциями и журнализацию. Соответственно, можно выделить такие компоненты ядра (по крайней мере, логически, хотя в некоторых системах эти компоненты выделяются явно), как менеджер данных, менеджер буферов, менеджер транзакций и менеджер журнала. Как можно было понять из первой части этой лекции, функции этих компонентов взаимосвязаны, и для обеспечения корректной работы СУБД все эти компоненты должны взаимодействовать по тщательно продуманным и проверенным протоколам. Ядро СУБД обладает собственным интерфейсом, не доступным пользователям напрямую и используемым в программах, производимых компилятором SQL (или в подсистеме поддержки выполнения таких программ) и утилитах БД. Ядро СУБД является основной резидентной частью СУБД. При использовании архитектуры "клиент-сервер" ядро является основной составляющей серверной части системы.

Основной функцией компилятора языка БД является компиляция операторов языка БД в некоторую выполняемую программу. Основной проблемой реляционных СУБД является то, что языки этих систем (а это, как правило, SQL) являются непроцедурными, т.е. в операторе такого языка специфицируется некоторое действие над БД, но эта спецификация не является процеду-

рой, а лишь описывает в некоторой форме условия совершения желаемого действия (вспомните примеры из первой лекции). Поэтому компилятор должен решить, каким образом выполнять оператор языка прежде, чем произвести программу. Применяются достаточно сложные методы оптимизации операторов, которые мы подробно рассмотрим в следующих лекциях. Результатом компиляции является выполняемая программа, представляемая в некоторых системах в машинных кодах, но более часто в выполняемом внутреннем машинно-независимом коде. В последнем случае реальное выполнение оператора производится с привлечением подсистемы поддержки времени выполнения, представляющей собой, по сути дела, интерпретатор этого внутреннего языка.

Наконец, в отдельные утилиты БД обычно выделяют такие процедуры, которые слишком накладно выполнять с использованием языка БД, например, загрузка и выгрузка БД, сбор статистики, глобальная проверка целостности БД и т.д. Утилиты программируются с использованием интерфейса ядра СУБД, а иногда даже с проникновением внутрь ядра.

2.3. Пример: System R.

Основными целями разработчиков System R являлись следующие:

- обеспечить ненавигационный интерфейс высокого уровня пользователя с системой, позволяющий достичь независимости данных и дать возможность пользователям работать максимально эффективно;
- обеспечить многообразие допустимых способов использования СУБД, включая программируемые транзакции, диалоговые транзакции и генерацию отчетов;
- поддерживать динамически изменяемую среду баз данных, в которой отношения, индексы, представления, транзакции и другие объекты могут легко добавляться и уничтожаться без приостановки нормального функционирования системы;
- обеспечить возможность параллельной работы с одной базой данных многих пользователей с допущением параллельной модификации объектов базы данных при наличии необходимых средств защиты целостности базы данных;
- обеспечить средства восстановления согласованного состояния баз данных после разного рода сбоев аппаратуры или программного обеспечения;
- обеспечить гибкий механизм, позволяющий определять различные представления хранимых данных, и ограничивать этими представлениями доступ пользователей к базе данных по выборке и модификации на основе механизма авторизации;
- обеспечить производительность системы при выполнении упомянутых функций, сопоставимую с производительностью существующих СУБД низкого уровня.

Структурная организация System R вполне согласуется с поставленными при ее разработке целями. Основными структурными компонентами System R являются система управления реляционной памятью (Relational Storage System - RSS) и компилятор запросов языка SQL. RSS обеспечивает интерфейс довольно низкого, но достаточного для реализации SQL уровня для доступа к хранимым в базе данным. Синхронизация транзакций, журнализация изменений и восстановление баз данных после сбоев также относятся к числу функций RSS. Компилятор запросов использует интерфейс RSS для доступа к разнообразной справочной информации (каталогам отношений, индексов, прав доступа, условий целостности, условных воздействий и т.д.) и производит рабочие программы, выполняемые в дальнейшем также с использованием интерфейса RSS. Таким образом, система естественно разделяется на два уровня - уровень управления памятью и синхронизацией, фактически, не зависящий от базового языка запросов системы, и языковой уровень (уровень SQL), на котором решается большинство проблем System R. Заметим, что эта независимость скорее условная, чем абсолютная: язык SQL можно заменить на другой язык, но он должен обладать примерно такой же семантикой.

§3. Ранние подходы к организации БД. Системы, основанные на инвертированных списках, иерархические и сетевые СУБД. Примеры. Сильные места и недостатки ранних систем.

Прежде, чем перейти к детальному и последовательному изучению реляционных систем БД, остановимся коротко на ранних (дореляционных) СУБД. В этом есть смысл по трем причинам: во-первых, эти системы исторически предшествовали реляционным, и для правильного понимания причин повсеместного перехода к реляционным системам нужно знать хотя бы что-нибудь про их предшественников; во-вторых, внутренняя организация реляционных систем во многом основана на использовании методов ранних систем; в-третьих, некоторое знание в области ранних систем будет полезно для понимания путей развития постреляционных СУБД.

Заметим, что в этой лекции мы ограничиваемся рассмотрением только общих подходов к организации трех типов ранних систем, а именно, систем, основанных на инвертированных списках, иерархических и сетевых систем управления базами данных. Мы не будем касаться особенностей каких-либо конкретных систем; это привело бы к изложению многих технических деталей, которые, хотя и интересны, находятся несколько в стороне от основной цели нашего курса. Детали можно найти в рекомендованной литературе.

Начнем с некоторых наиболее общих характеристик ранних систем:

- Эти системы активно использовались в течение многих лет, дольше, чем используется какая-либо из реляционных СУБД. На самом деле некоторые из ранних систем используются даже в наше время, накоплены громадные базы данных, и одной из актуальных проблем информационных систем является использование этих систем совместно с современными системами.
- Все ранние системы не основывались на каких-либо абстрактных моделях. Как мы упоминали, понятие модели данных фактически вошло в обиход специалистов в области БД только вместе с реляционным подходом. Абстрактные представления ранних систем появились позже на основе анализа и выявления общих признаков у различных конкретных систем.
- В ранних системах доступ к БД производился на уровне записей. Пользователи этих систем осуществляли явную навигацию в БД, используя языки программирования, расширенные функциями СУБД. Интерактивный доступ к БД поддерживался только путем создания соответствующих прикладных программ с собственным интерфейсом.
- Можно считать, что уровень средств ранних СУБД соотносится с уровнем файловых систем примерно так же, как уровень языка Кобол соотносится с уровнем языка Ассемблера. Заметим, что при таком взгляде уровень реляционных систем соответствует уровню языков Ада или APL.
- Навигационная природа ранних систем и доступ к данным на уровне записей заставляли пользователя самого производить всю оптимизацию доступа к БД, без какой-либо поддержки системы.
- После появления реляционных систем большинство ранних систем было оснащено "реляционными" интерфейсами. Однако в большинстве случаев это не сделало их по-настоящему реляционными системами, поскольку оставалась возможность манипулировать данными в естественном для них режиме.

3.1. Основные особенности систем, основанных на инвертированных списках.

К числу наиболее известных и типичных представителей таких систем относятся Datacom/DB компании Applied Data Research, Inc. (ADR), ориентированная на использование на машинах основного класса фирмы IBM, и Adabas компании Software AG.

Организация доступа к данным на основе инвертированных списков используется практически во всех современных реляционных СУБД, но в этих системах пользователи не имеют непосредственного доступа к инвертированным спискам (индексам). Кстати, когда мы будем рассматривать внутренние интерфейсы реляционных СУБД, вы увидите, что они очень близки к пользовательским интерфейсам систем, основанных на инвертированных списках.

3.1.1. Структуры данных.

База данных, организованная с помощью инвертированных списков, похожа на реляционную БД, но с тем отличием, что хранимые таблицы и пути доступа к ним видны пользователям. При этом:

- Строки таблиц упорядочены системой в некоторой физической последовательности.
- Физическая упорядоченность строк всех таблиц может определяться и для всей БД (так делается, например, в Datacom/DB).
- Для каждой таблицы можно определить произвольное число ключей поиска, для которых строятся индексы. Эти индексы автоматически поддерживаются системой, но явно видны пользователям.

3.1.2. Манипулирование данными.

Поддерживаются два класса операторов:

- Операторы, устанавливающие адрес записи, среди которых:
  - прямые поисковые операторы (например, найти первую запись таблицы по некоторому пути доступа);
  - операторы, находящие запись в терминах относительной позиции от предыдущей записи по некоторому пути доступа.
- Операторы над адресуемыми записями

Типичный набор операторов:

- *LOCATE FIRST* - найти первую запись таблицы T в физическом порядке; возвращает адрес записи;
- *LOCATE FIRST WITH SEARCH KEY EQUAL* - найти первую запись таблицы T с заданным значением ключа поиска K; возвращает адрес записи;
- *LOCATE NEXT* - найти первую запись, следующую за записью с заданным адресом в заданном пути доступа; возвращает адрес записи;
- *LOCATE NEXT WITH SEARCH KEY EQUAL* - найти следующую запись таблицы T в порядке пути поиска с заданным значением K; должно быть соответствие между используемым способом сканирования и ключом K; возвращает адрес записи;
- *LOCATE FIRST WITH SEARCH KEY GREATHER* - найти первую запись таблицы T в порядке ключа поиска K со значением ключевого поля, большим заданного значения K; возвращает адрес записи;
- *RETRIVE* - выбрать запись с указанным адресом;
- *UPDATE* - обновить запись с указанным адресом;
- *DELETE* - удалить запись с указанным адресом;
- *STORE* - включить запись в указанную таблицу; операция генерирует адрес записи.

3.1.3. Ограничения целостности.

Общие правила определения целостности БД отсутствуют. В некоторых системах поддерживаются ограничения уникальности значений некоторых полей, но в основном все возлагается на прикладную программу.

3.2. Иерархические системы.

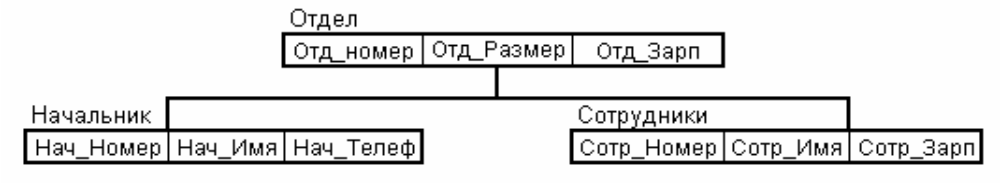
Типичным представителем (наиболее известным и распространенным) является Information Management System (IMS) фирмы IBM. Первая версия появилась в 1968 г. До сих пор поддерживается много баз данных, что создает существенные проблемы с переходом как на новую технологию БД, так и на новую технику.

3.2.1. Иерархические структуры данных.

Иерархическая БД состоит из упорядоченного набора деревьев; более точно, из упорядоченного набора нескольких экземпляров одного типа дерева.

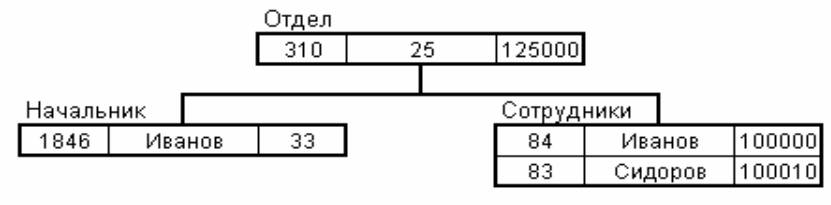
Тип дерева состоит из одного "корневого" типа записи и упорядоченного набора из нуля или более типов поддеревьев (каждое из которых является некоторым типом дерева). Тип дерева в целом представляет собой иерархически организованный набор типов записи.

Пример типа дерева (схемы иерархической БД):



Здесь Отдел является предком для Начальник и Сотрудники, а Начальник и Сотрудники - потомки Отдел. Между типами записи поддерживаются связи.

База данных с такой схемой могла бы выглядеть следующим образом (мы показываем один экземпляр дерева):



Все экземпляры данного типа потомка с общим экземпляром типа предка называются близнецами. Для БД определен полный порядок обхода - сверху-вниз, слева-направо.

В IMS использовалась оригинальная и нестандартная терминология: "сегмент" вместо "запись", а под "записью БД" понималось все дерево сегментов.

3.2.2. Манипулирование данными.

Примерами типичных операторов манипулирования иерархически организованными данными могут быть следующие:

- Найти указанное дерево БД (например, отдел 310);
- Перейти от одного дерева к другому;
- Перейти от одной записи к другой внутри дерева (например, от отдела - к первому сотруднику);
- Перейти от одной записи к другой в порядке обхода иерархии;
- Вставить новую запись в указанную позицию;
- Удалить текущую запись.

3.2.3. Ограничения целостности.

Автоматически поддерживается целостность ссылок между предками и потомками. ***Основное правило: никакой потомок не может существовать без своего родителя***. Заметим, что аналогичное поддержание целостности по ссылкам между записями, не входящими в одну иерархию, не поддерживается (примером такой "внешней" ссылки может быть содержимое поля *Каф _ Номер* в экземпляре типа записи *Куратор*).

В иерархических системах поддерживалась некоторая форма представлений БД на основе ограничения иерархии. Примером представления приведенной выше БД может быть иерархия

3.3. Сетевые системы.

Типичным представителем является Integrated Database Management System (IDMS) компании Cullinet Software, Inc., предназначенная для использования на машинах основного класса

фирмы IBM под управлением большинства операционных систем. Архитектура системы основана на предложениях Data Base Task Group (DBTG) Комитета по языкам программирования Conference on Data Systems Languages (CODASYL), организации, ответственной за определение языка программирования Кобол. Отчет DBTG был опубликован в 1971 г., а в 70-х годах появилось несколько систем, среди которых IDMS.

3.3.1. Сетевые структуры данных.

Сетевой подход к организации данных является расширением иерархического. В иерархических структурах запись-потомок должна иметь в точности одного предка; в сетевой структуре данных потомок может иметь любое число предков.

Сетевая БД состоит из набора записей и набора связей между этими записями, а если говорить более точно, из набора экземпляров каждого типа из заданного в схеме БД набора типов записи и набора экземпляров каждого типа из заданного набора типов связи.

Тип связи определяется для двух типов записи: предка и потомка. Экземпляр типа связи состоит из одного экземпляра типа записи предка и упорядоченного набора экземпляров типа записи потомка. Для данного типа связи L с типом записи предка P и типом записи потомка C должны выполняться следующие два условия:

- Каждый экземпляр типа P является предком только в одном экземпляре L;
- Каждый экземпляр C является потомком не более, чем в одном экземпляре L.

На формирование типов связи не накладываются особые ограничения; возможны, например, следующие ситуации:

- Тип записи потомка в одном типе связи L_1 может быть типом записи предка в другом типе связи L_2 (как в иерархии).
- Данный тип записи P может быть типом записи предка в любом числе типов связи.
- Данный тип записи P может быть типом записи потомка в любом числе типов связи.
- Может существовать любое число типов связи с одним и тем же типом записи предка и одним и тем же типом записи потомка; и если L_1 и L_2 - два типа связи с одним и тем же типом записи предка P и одним и тем же типом записи потомка C, то правила, по которым образуется родство, в разных связях могут различаться.
- Типы записи X и Y могут быть предком и потомком в одной связи и потомком и предком - в другой.
- Предок и потомок могут быть одного типа записи.

Простой пример сетевой схемы БД:

3.3.2. Манипулирование данными.

Примерный набор операций может быть следующим:

- Найти конкретную запись в наборе однотипных записей (инженера Сидорова);
- Перейти от предка к первому потомку по некоторой связи (к первому сотруднику отдела 310);
- Перейти к следующему потомку в некоторой связи (от Сидорова к Иванову);
- Перейти от потомка к предку по некоторой связи (найти отдел Сидорова);
- Создать новую запись;
- Уничтожить запись;
- Модифицировать запись;
- Включить в связь;

- Исключить из связи;
- Переставить в другую связь и т.д.

3.3.3. Ограничения целостности.

В принципе их поддержание не требуется, но иногда требуют целостности по ссылкам (как в иерархической модели).

3.4. Достоинства и недостатки.

Сильные места ранних СУБД:

- Развитые средства управления данными во внешней памяти на низком уровне;
- Возможность построения вручную эффективных прикладных систем;
- Возможность экономии памяти за счет разделения подобъектов (в сетевых системах).

Недостатки:

- Слишком сложно пользоваться;
- Фактически необходимы знания о физической организации;
- Прикладные системы зависят от этой организации;
- Их логика перегружена деталями организации доступа к БД.

Теоретические основы

Мы приступаем к изучению реляционных баз данных и систем управления реляционными базами данных. Этот подход является наиболее распространенным в настоящее время, хотя наряду с общепризнанными достоинствами обладает и рядом недостатков. К числу достоинств реляционного подхода можно отнести:

- наличие небольшого набора абстракций, которые позволяют сравнительно просто моделировать большую часть распространенных предметных областей и допускают точные формальные определения, оставаясь интуитивно понятными;
- наличие простого и в то же время мощного математического аппарата, опирающегося главным образом на теорию множеств и математическую логику и обеспечивающего теоретический базис реляционного подхода к организации баз данных;
- возможность ненавигационного манипулирования данными без необходимости знания конкретной физической организации баз данных во внешней памяти.

Реляционные системы далеко не сразу получили широкое распространение. В то время, как основные теоретические результаты в этой области были получены еще в 70-х, и тогда же появились первые прототипы реляционных СУБД, долгое время считалось невозможным добиться эффективной реализации таких систем. Однако отмеченные выше преимущества и постепенное накопление методов и алгоритмов организации реляционных баз данных и управления ими привели к тому, что уже в середине 80-х годов реляционные системы практически вытеснили с мирового рынка ранние СУБД.

В настоящее время основным предметом критики реляционных СУБД является не их недостаточная эффективность, а присущая этим системам некоторая ограниченность (прямое следствие простоты) при использование в так называемых нетрадиционных областях (наиболее распространенными примерами являются системы автоматизации проектирования), в которых требуются предельно сложные структуры данных. Еще одним часто отмечаемым недостатком реляционных баз данных является невозможность адекватного отражения семантики предметной области. Другими словами, возможности представления знаний о семантической специфике предметной области в реляционных системах очень ограничены. Современные исследования в области постреляционных систем главным образом посвящены именно устранению этих недостатков.

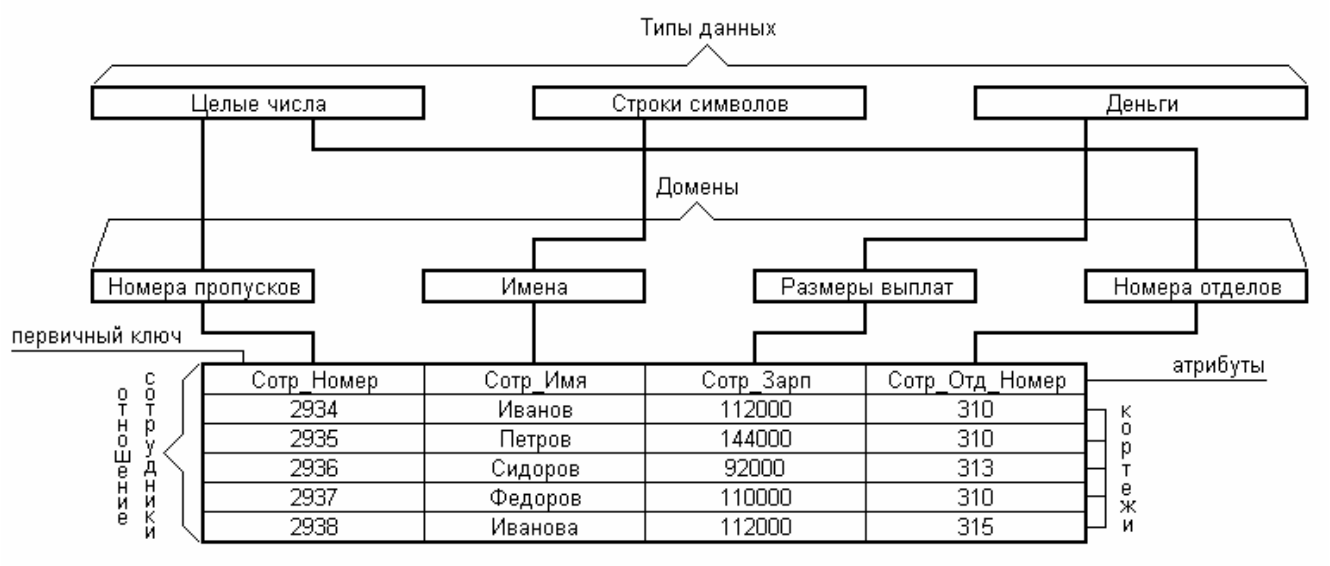
§4. Общие понятия реляционного подхода к организации БД. Основные концепции и термины.

На этой лекции мы введем на сравнительно неформальном уровне основные понятия реляционных баз данных, а также определим существо реляционной модели данных. Основной целью лекции является демонстрация простоты и возможности интуитивной интерпретации этих понятий. В дальнейших лекциях будут приводиться более формальные определения, на которых основывается математическая теория реляционных баз данных.

4.1. Базовые понятия реляционных баз данных.

Основными понятиями реляционных баз данных являются тип данных, домен, атрибут, кортеж, первичный ключ и отношение.

Для начала покажем смысл этих понятий на примере отношения *СОТРУДНИКИ* , содержащего информацию о сотрудниках некоторой организации:



4.1.1. Тип данных.

Понятие ***тип данных*** в реляционной модели данных полностью адекватно понятию типа данных в языках программирования. Обычно в современных реляционных БД допускается хранение символьных, числовых данных, битовых строк, специализированных числовых данных (таких как "деньги"), а также специальных "темпоральных" данных (дата, время, временной интервал). Достаточно активно развивается подход к расширению возможностей реляционных систем абстрактными типами данных (соответствующими возможностями обладают, например, системы семейства Ingres/Postgres). В нашем примере мы имеем дело с данными трех типов: строки символов, целые числа и "деньги".

4.1.2. Домен.

Понятие ***домена*** более специфично для баз данных, хотя и имеет некоторые аналогии с подтипами в некоторых языках программирования. В самом общем виде домен определяется заданием некоторого базового типа данных, к которому относятся элементы домена, и произвольного логического выражения, применяемого к элементу типа данных. Если вычисление этого логического выражения дает результат "истина", то элемент данных является элементом домена.

Наиболее правильной интуитивной трактовкой понятия домена является понимание домена как допустимого потенциального множества значений данного типа. Например, домен "Имена" в нашем примере определен на базовом типе строк символов, но в число его значений могут входить только те строки, которые могут изображать имя (в частности, такие строки не могут начинаться с мягкого знака).

Следует отметить также семантическую нагрузку понятия домена: данные считаются сравнимыми только в том случае, когда они относятся к одному домену. В нашем примере значения

доменов "Номера пропусков" и "Номера групп" относятся к типу целых чисел, но не являются сравнимыми. Заметим, что в большинстве реляционных СУБД понятие домена не используется, хотя в Oracle V.7 оно уже поддерживается.

4.1.3. Схема отношения, схема базы данных.

Схема отношения - это именованное множество пар {имя атрибута, имя домена (или типа, если понятие домена не поддерживается)}. Степень или "арность" схемы отношения - мощность этого множества. Степень отношения *СОТРУДНИКИ* равна четырем, то есть оно является 4-арным. Если все атрибуты одного отношения определены на разных доменах, осмысленно использовать для именования атрибутов имена соответствующих доменов (не забывая, конечно, о том, что это является всего лишь удобным способом именования и не устраняет различия между понятиями домена и атрибута).

Схема БД (в структурном смысле) - это набор именованных схем отношений.

4.1.4. Кортеж, отношение.

Кортеж, соответствующий данной схеме отношения, - это множество пар {имя атрибута, значение}, которое содержит одно вхождение каждого имени атрибута, принадлежащего схеме отношения. "Значение" является допустимым значением домена данного атрибута (или типа данных, если понятие домена не поддерживается). Тем самым, степень или "арность" кортежа, т.е. число элементов в нем, совпадает с "арностью" соответствующей схемы отношения. Попросту говоря, кортеж - это набор именованных значений заданного типа.

Отношение - это множество кортежей, соответствующих одной схеме отношения. Иногда, чтобы не путаться, говорят "отношение-схема" и "отношение-экземпляр", иногда схему отношения называют заголовком отношения, а отношение как набор кортежей - телом отношения. На самом деле, понятие схемы отношения ближе всего к понятию структурного типа данных в языках программирования. Было бы вполне логично разрешать отдельно определять схему отношения, а затем одно или несколько отношений с данной схемой.

Однако в реляционных базах данных это не принято. Имя схемы отношения в таких базах данных всегда совпадает с именем соответствующего отношения-экземпляра. В классических реляционных базах данных после определения схемы базы данных изменяются только отношения-экземпляры. В них могут появляться новые и удаляться или модифицироваться существующие кортежи. Однако во многих реализациях допускается и изменение схемы базы данных: определение новых и изменение существующих схем отношения. Это принято называть ***эволюцией схемы базы данных***.

Обычным житейским представлением отношения является таблица, заголовком которой является схема отношения, а строками - кортежи отношения-экземпляра; в этом случае имена атрибутов именуют столбцы этой таблицы. Поэтому иногда говорят "столбец таблицы", имея в виду "атрибут отношения". Когда мы перейдем к рассмотрению практических вопросов организации реляционных баз данных и средств управления, мы будем использовать эту житейскую терминологию. Этой терминологии придерживаются в большинстве коммерческих реляционных СУБД.

Реляционная база данных - это набор отношений, имена которых совпадают с именами схем отношений в схеме БД.

Как видно, основные структурные понятия реляционной модели данных (если не считать понятия домена) имеют очень простую интуитивную интерпретацию, хотя в теории реляционных БД все они определяются абсолютно формально и точно.

4.2. Фундаментальные свойства отношений.

Остановимся теперь на некоторых важных свойствах отношений, которые следуют из приведенных ранее определений:

4.2.1. Отсутствие кортежей-дубликатов.

То свойство, что отношения не содержат кортежей-дубликатов, следует из определения отношения как множества кортежей. В классической теории множеств по определению каждое множество состоит из различных элементов.

Из этого свойства вытекает наличие у каждого отношения так называемого первичного ключа - набора атрибутов, значения которых однозначно определяют кортеж отношения. Для каждого отношения по крайней мере полный набор его атрибутов обладает этим свойством. Однако при формальном определении первичного ключа требуется обеспечение его "минимальности", т.е. в набор атрибутов первичного ключа не должны входить такие атрибуты, которые можно отбросить без ущерба для основного свойства - однозначно определять кортеж. Понятие ***первичного ключа*** является исключительно важным в связи с понятием целостности баз данных.

Забегая вперед, заметим, что во многих практических реализациях РСУБД допускается нарушение свойства уникальности кортежей для промежуточных отношений, порождаемых неявно при выполнении запросов. Такие отношения являются не множествами, а мультимножествами, что в ряде случаев позволяет добиться определенных преимуществ, но иногда приводит к серьезным проблемам.

4.2.2. Отсутствие упорядоченности кортежей.

Свойство отсутствия упорядоченности кортежей отношения также является следствием определения отношения-экземпляра как множества кортежей. Отсутствие требования к поддержанию порядка на множестве кортежей отношения дает дополнительную гибкость СУБД при хранении баз данных во внешней памяти и при выполнении запросов к базе данных. Это не противоречит тому, что при формулировании запроса к БД, например, на языке SQL можно потребовать сортировки результирующей таблицы в соответствии со значениями некоторых столбцов. Такой результат, вообще говоря, не отношение, а некоторый упорядоченный список кортежей.

4.2.3. Отсутствие упорядоченности атрибутов.

Атрибуты отношений не упорядочены, поскольку по определению схема отношения есть множество пар {имя атрибута, имя домена}. Для ссылки на значение атрибута в кортеже отношения всегда используется имя атрибута. Это свойство теоретически позволяет, например, модифицировать схемы существующих отношений не только путем добавления новых атрибутов, но и путем удаления существующих атрибутов. Однако в большинстве существующих систем такая возможность не допускается, и хотя упорядоченность набора атрибутов отношения явно не требуется, часто в качестве неявного порядка атрибутов используется их порядок в линейной форме определения схемы отношения.

4.2.4. Атомарность значений атрибутов.

Значения всех атрибутов являются атомарными. Это следует из определения домена как потенциального множества значений простого типа данных, т.е. среди значений домена не могут содержаться множества значений (отношения). Принято говорить, что в реляционных базах данных допускаются только нормализованные отношения или отношения, представленные в первой нормальной форме. Потенциальным примером ненормализованного отношения является следующее:

НОМЕР_ОТДЕЛА	ОТДЕЛ		
	СОТР_НОМЕР	СОТР_ИМЯ	СОТР_ЗАРП
310	2934	Иванов	112,000
	2935	Петров	144,000
	2937	Федоров	110,000
313	2936	Сидоров	92,000
315	2938	Иванова	112,000

Можно сказать, что здесь мы имеем бинарное отношение, значениями атрибута *ОТДЕЛЫ* которого являются отношения. Заметим, что исходное отношение *СОТРУДНИКИ* является нормализованным вариантом отношения *ОТДЕЛЫ* :

СОТР_НОМЕР	СОТР_ИМЯ	СОТР_ЗАПР	СОТР_ОТД_НОМЕР
2934	Иванов	112,000	310
2935	Петров	144,000	310

2936	Сидоров	92,000	313
2937	Федоров	110,000	310
2938	Иванова	112,000	315

Нормализованные отношения составляют основу классического реляционного подхода к организации баз данных. Они обладают некоторыми ограничениями (не любую информацию удобно представлять в виде плоских таблиц), но существенно упрощают манипулирование данными. Рассмотрим, например, два идентичных оператора занесения кортежа:

- Зачислить сотрудника Кузнецова (пропуск номер 3000, зарплата 115,000) в отдел номер 320.
- Зачислить сотрудника Кузнецова (пропуск номер 3000, зарплата 115,000) в отдел номер 310.

Если информация о сотрудниках представлена в виде отношения *СОТРУДНИКИ* , оба оператора будут выполняться одинаково (вставить кортеж в отношение *СОТРУДНИКИ*). Если же работать с ненормализованным отношением *ОТДЕЛЫ* , то первый оператор выразится в занесение кортежа, а второй - в добавление информации о Кузнецове в множественное значение атрибута *ОТДЕЛ* кортежа с первичным ключом 310.

4.3. Реляционная модель данных.

Когда в предыдущих разделах мы говорили об основных понятиях реляционных баз данных, мы не опирались на какую-либо конкретную реализацию. Эти рассуждения в равной степени относились к любой системе, при построении которой использовался реляционный подход.

Другими словами, мы использовали понятия так называемой реляционной модели данных. Модель данных описывает некоторый набор родовых понятий и признаков, которыми должны обладать все конкретные СУБД и управляемые ими базы данных, если они основываются на этой модели. Наличие модели данных позволяет сравнивать конкретные реализации, используя один общий язык.

Хотя понятие модели данных является общим, и можно говорить о иерархической, сетевой, некоторой семантической и т.д. моделях данных, нужно отметить, что это понятие было введено в обиход применительно к реляционным системам и наиболее эффективно используется именно в этом контексте. Попытки прямолинейного применения аналогичных моделей к дореляционным организациям показывают, что реляционная модель слишком "велика" для них, а для постреляционных организаций она оказывается "мала".

4.3.1. Общая характеристика.

Наиболее распространенная трактовка реляционной модели данных, по-видимому, принадлежит Дейту, который воспроизводит ее (с различными уточнениями) практически во всех своих книгах. Согласно Дейту реляционная модель состоит из трех частей, описывающих разные аспекты реляционного подхода: структурной части, манипуляционной части и целостной части.

В структурной части модели фиксируется, что единственной структурой данных, используемой в реляционных БД, является нормализованное n-арное отношение. По сути дела, в предыдущих двух разделах этой лекции мы рассматривали именно понятия и свойства структурной составляющей реляционной модели.

В манипуляционной части модели утверждаются два фундаментальных механизма манипулирования реляционными БД - реляционная алгебра и реляционное исчисление. Первый механизм базируется в основном на классической теории множеств (с некоторыми уточнениями), а второй - на классическом логическом аппарате исчисления предикатов первого порядка. Мы рассмотрим эти механизмы более подробно на следующей лекции, а пока лишь заметим, что основной функцией манипуляционной части реляционной модели является обеспечение меры реляционности любого конкретного языка реляционных БД: язык называется реляционным, если он обладает не меньшей выразительностью и мощностью, чем реляционная алгебра или реляционное исчисление.

4.3.2. Целостность сущности и ссылок.

Наконец, в целостной части реляционной модели данных фиксируются два базовых требования целостности, которые должны поддерживаться в любой реляционной СУБД. Первое требование называется ***требованием целостности сущностей***. Объекту или сущности реального мира в реляционных БД соответствуют кортежи отношений. Конкретно требование состоит в том, что любой кортеж любого отношения отличим от любого другого кортежа этого отношения, т.е. другими словами, любое отношение должно обладать первичным ключом. Как мы видели в предыдущем разделе, это требование автоматически удовлетворяется, если в системе не нарушаются базовые свойства отношений.

Второе требование называется ***требованием целостности по ссылкам*** и является несколько более сложным. Очевидно, что при соблюдении нормализованности отношений сложные сущности реального мира представляются в реляционной БД в виде нескольких кортежей нескольких отношений. Например, представим, что нам требуется представить в реляционной базе данных сущность *ОТДЕЛ* с атрибутами *ОТД _ НОМ* (номер отдела), *ОТД _ КОЛ* (количество сотрудников) и *ОТД _ СОТР* (набор сотрудников отдела). Для каждого сотрудника нужно хранить *СОТР _ НОМ* (номер сотрудника), *СОТР _ ИМЯ* (имя сотрудника) и *СОТР _ ЗАРП* (заработная плата сотрудника). Как мы вскоре увидим, при правильном проектировании соответствующей БД в ней появятся два отношения:

```
ОТДЕЛЫ (ОТД_НОМ, ОТД_КОЛ)
```

(первичный ключ - *ОТД _ НОМ*) и

```
СОТРУДНИКИ (СОТР_НОМ, СОТР_ИМЯ, СОТР_ЗАРП, СОТР_ОТД_НОМ)
```

(первичный ключ - *СОТР _ НОМ*).

Как видно, атрибут *СОТР _ ОТД _ НОМ* появляется в отношении *СОТРУДНИКИ* не потому, что номер отдела является собственным свойством сотрудника, а лишь для того, чтобы иметь возможность восстановить при необходимости полную сущность *ОТДЕЛ* . Значение атрибута *СОТР _ ОТД _ НОМ* в любом кортеже отношения *СОТРУДНИКИ* должно соответствовать значению атрибута *ОТД _ НОМ* в некотором кортеже отношения *ОТДЕЛЫ* . Атрибут такого рода называется ***внешним ключом***, поскольку его значения однозначно характеризуют сущности, представленные кортежами некоторого другого отношения (т.е. задают значения их первичного ключа). Говорят, что отношение, в котором определен внешний ключ, ссылается на соответствующее отношение, в котором такой же атрибут является первичным ключом.

Требование целостности по ссылкам, или требование внешнего ключа состоит в том, что для каждого значения внешнего ключа, появляющегося в ссылающемся отношении, в отношении, на которое ведет ссылка, должен найтись кортеж с таким же значением первичного ключа, либо значение внешнего ключа должно быть неопределенным (т.е. ни на что не указывать). Для нашего примера это означает, что если для сотрудника указан номер отдела, то этот отдел должен существовать.

Ограничения целостности сущности и по ссылкам должны поддерживаться СУБД. Для соблюдения целостности сущности достаточно гарантировать отсутствие в любом отношении кортежей с одним и тем же значением первичного ключа. С целостностью по ссылкам дела обстоят несколько более сложно.

Понятно, что при обновлении ссылающегося отношения (вставке новых кортежей или модификации значения внешнего ключа в существующих кортежах) достаточно следить за тем, чтобы не появлялись некорректные значения внешнего ключа. Но как быть при удалении кортежа из отношения, на которое ведет ссылка?

Здесь существуют три подхода, каждый из которых поддерживает целостность по ссылкам. Первый подход заключается в том, что запрещается производить удаление кортежа, на который существуют ссылки (т.е. сначала нужно либо удалить ссылающиеся кортежи, либо соответствующим образом изменить значения их внешнего ключа). При втором подходе при удалении

кортежа, на который имеются ссылки, во всех ссылающихся кортежах значение внешнего ключа автоматически становится неопределенным. Наконец, третий подход (каскадное удаление) состоит в том, что при удалении кортежа из отношения, на которое ведет ссылка, из ссылающегося отношения автоматически удаляются все ссылающиеся кортежи.

В развитых реляционных СУБД обычно можно выбрать способ поддержания целостности по ссылкам для каждой отдельной ситуации определения внешнего ключа. Конечно, для принятия такого решения необходимо анализировать требования конкретной прикладной области.

§5. Базисные средства манипулирования реляционными данными.

В предыдущей лекции мы говорили про три составляющих реляционной модели данных. Две из них - структурную и целостную составляющие - мы рассмотрели более или менее подробно, а манипуляционной части реляционной модели данных посвящается эта лекция.

Как мы отмечали в предыдущей лекции, в манипуляционной составляющей определяются два базовых механизма манипулирования реляционными данными - основанная на теории множеств реляционная алгебра и базирующееся на математической логике (точнее, на исчислении предикатов первого порядка) реляционное исчисление. В свою очередь, обычно рассматриваются два вида реляционного исчисления - исчисление доменов и исчисление предикатов.

Все эти механизмы обладают одним важным свойством: они замкнуты относительно понятия отношения. Это означает, что выражения реляционной алгебры и формулы реляционного исчисления определяются над отношениями реляционных БД и результатом вычисления также являются отношения. В результате любое выражение или формула могут интерпретироваться как отношения, что позволяет использовать их в других выражениях или формулах.

Как мы увидим, алгебра и исчисление обладают большой выразительной мощностью: очень сложные запросы к базе данных могут быть выражены с помощью одного выражения реляционной алгебры или одной формулы реляционного исчисления. Именно по этой причине именно эти механизмы включены в реляционную модель данных. Конкретный язык манипулирования реляционными БД называется ***реляционно полным***, если любой запрос, выражаемый с помощью одного выражения реляционной алгебры или одной формулы реляционного исчисления, может быть выражен с помощью одного оператора этого языка.

Известно (и мы не будем это доказывать), что механизмы реляционной алгебры и реляционного исчисления эквивалентны, т.е. для любого допустимого выражения реляционной алгебры можно построить эквивалентную (т.е. производящую такой же результат) формулу реляционного исчисления и наоборот. Почему же в реляционной модели данных присутствуют оба эти механизма?

Дело в том, что они различаются уровнем процедурности. Выражения реляционной алгебры строятся на основе алгебраических операций (высокого уровня), и подобно тому, как интерпретируются арифметические и логические выражения, выражение реляционной алгебры также имеет процедурную интерпретацию. Другими словами, запрос, представленный на языке реляционной алгебры, может быть вычислен на основе вычисления элементарных алгебраических операций с учетом их старшинства и возможного наличия скобок. Для формулы реляционного исчисления однозначная интерпретация, вообще говоря, отсутствует. Формула только устанавливает условия, которым должны удовлетворять кортежи результирующего отношения. Поэтому языки реляционного исчисления являются более непроцедурными или декларативными.

Поскольку механизмы реляционной алгебры и реляционного исчисления эквивалентны, то в конкретной ситуации для проверки степени реляционности некоторого языка БД можно пользоваться любым из этих механизмов.

Заметим, что крайне редко алгебра или исчисление принимаются в качестве полной основы какого-либо языка БД. Обычно (как, например, в случае языка SQL) язык основывается на некоторой смеси алгебраических и логических конструкций. Тем не менее, знание алгебраических и логических основ языков баз данных часто бывает полезно на практике.

В нашем изложении мы в основном следуем подходу Дейта, примененному (хотя и не изобретенному) им в последнем издании книги "Введение в системы баз данных". Для экономии времени и места мы не будем вводить каких-либо строгих синтаксических конструкций, а в основном ограничимся рассмотрением материала на содержательном уровне.

5.1. Реляционная алгебра.

Основная идея реляционной алгебры состоит в том, что коль скоро отношения являются множествами, то средства манипулирования отношениями могут базироваться на традиционных теоретико-множественных операциях, дополненных некоторыми специальными операциями, специфичными для баз данных.

Существует много подходов к определению реляционной алгебры, которые различаются набором операций и способами их интерпретации, но в принципе, более или менее равносильны. Мы опишем немного расширенный начальный вариант алгебры, который был предложен Коддом. В этом варианте набор основных алгебраических операций состоит из восьми операций, которые делятся на два класса - теоретико-множественные операции и специальные реляционные операции. В состав теоретико-множественных операций входят операции:

- объединения отношений;
- пересечения отношений;
- взятия разности отношений;
- прямого произведения отношений.

Специальные реляционные операции включают:

- ограничение отношения;
- проекцию отношения;
- соединение отношений;
- деление отношений.

Кроме того, в состав алгебры включается операция присваивания, позволяющая сохранить в базе данных результаты вычисления алгебраических выражений, и операция переименования атрибутов, дающая возможность корректно сформировать заголовок (схему) результирующего отношения.

5.1.1. Общая интерпретация реляционных операций.

Если не вдаваться в некоторые тонкости, которые мы рассмотрим в следующих подразделах, то почти все операции предложенного выше набора обладают очевидной и простой интерпретацией.

- При выполнении операции объединения двух отношений производится отношение, включающее все кортежи, входящие хотя бы в одно из отношений-операндов.
- Операция пересечения двух отношений производит отношение, включающее все кортежи, входящие в оба отношения-операнда.
- Отношение, являющееся разностью двух отношений включает все кортежи, входящие в отношение - первый операнд, такие, что ни один из них не входит в отношение, являющееся вторым операндом.
- При выполнении прямого произведения двух отношений производится отношение, кортежи которого являются конкатенацией (сцеплением) кортежей первого и второго операндов.
- Результатом ограничения отношения по некоторому условию является отношение, включающее кортежи отношения-операнда, удовлетворяющее этому условию.
- При выполнении проекции отношения на заданный набор его атрибутов производится отношение, кортежи которого производятся путем взятия соответствующих значений из кортежей отношения-операнда.
- При соединении двух отношений по некоторому условию образуется результирующее отношение, кортежи которого являются конкатенацией кортежей первого и второго отношений и удовлетворяют этому условию.

- У операции реляционного деления два операнда - бинарное и унарное отношения. Результирующее отношение состоит из одноатрибутных кортежей, включающих значения первого атрибута кортежей первого операнда таких, что множество значений второго атрибута (при фиксированном значении первого атрибута) совпадает со множеством значений второго операнда.
- Операция переименования производит отношение, тело которого совпадает с телом операнда, но имена атрибутов изменены.
- Операция присваивания позволяет сохранить результат вычисления реляционного выражения в существующем отношении БД.

Поскольку результатом любой реляционной операции (кроме операции присваивания) является некоторое отношение, можно образовывать реляционные выражения, в которых вместо отношения-операнда некоторой реляционной операции находится вложенное реляционное выражение.

5.1.2. Замкнутость реляционной алгебры и операция переименования.

Как мы говорили в предыдущей лекции, каждое отношение характеризуется схемой (или заголовком) и набором кортежей (или телом). Поэтому, если действительно желать иметь алгебру, операции которой замкнуты относительно понятия отношения, то каждая операция должна производить отношение в полном смысле, т.е. оно должно обладать и телом, и заголовком. Только в этом случае будет действительно возможно строить вложенные выражения.

Заголовок отношения представляет собой множество пар

```
<имя-атрибута, имя-домена>
```

Если посмотреть на общий обзор реляционных операций, приведенный в предыдущем подразделе, то видно, что домены атрибутов результирующего отношения однозначно определяются доменами отношений-операндов. Однако с именами атрибутов результата не всегда все так просто.

Например, представим себе, что у отношений-операндов операции прямого произведения имеются одноименные атрибуты с одинаковыми доменами. Каким был бы заголовок результирующего отношения? Поскольку это множество, в нем не должны содержаться одинаковые элементы. Но и потерять атрибут в результате недопустимо. А это значит, что в этом случае вообще невозможно корректно выполнить операцию прямого произведения.

Аналогичные проблемы могут возникать и в случаях других двуместных операций. Для их разрешения в состав операций реляционной алгебры вводится операция переименования. Ее следует применять в любом случае, когда возникает конфликт именования атрибутов в отношениях - операндах одной реляционной операции. Тогда к одному из операндов сначала применяется операция переименования, а затем основная операция выполняется уже безо всяких проблем.

В дальнейшем изложении мы будем предполагать применение операции переименования во всех конфликтных случаях.

5.1.3. Особенности теоретико-множественных операций реляционной алгебры.

Хотя в основе теоретико-множественной части реляционной алгебры лежит классическая теория множеств, соответствующие операции реляционной алгебры обладают некоторыми особенностями.

Начнем с операции объединения (все, что будет говориться по поводу объединения, переносится на операции пересечения и взятия разности). Смысл операции объединения в реляционной алгебре в целом остается теоретико-множественным. Но если в теории множеств операция объединения осмысленна для любых двух множеств-операндов, то в случае реляционной алгебры результатом операции объединения должно являться отношение. Если допустить в реляционной алгебре возможность теоретико-множественного объединения произвольных двух отношений (с разными схемами), то, конечно, результатом операции будет множество, но множество разнотипных кортежей, т.е. не отношение. Если исходить из требования замкнутости реляционной

алгебры относительно понятия отношения, то такая операция объединения является бессмысленной.

Все эти соображения приводят к появлению понятия ***совместимости отношений по объединению***: два отношения совместимы по объединению в том и только в том случае, когда обладают одинаковыми заголовками. Более точно, это означает, что в заголовках обоих отношений содержится один и тот же набор имен атрибутов, и одноименные атрибуты определены на одном и том же домене.

Если два отношения совместимы по объединению, то при обычном выполнении над ними операций объединения, пересечения и взятия разности результатом операции является отношение с корректно определенным заголовком, совпадающим с заголовком каждого из отношений-операндов. Напомним, что если два отношения "почти" совместимы по объединению, т.е. совместимы во всем, кроме имен атрибутов, то до выполнения операции типа соединения эти отношения можно сделать полностью совместимыми по объединению путем применения операции переименования.

Заметим, что включение в состав операций реляционной алгебры трех операций объединения, пересечения и взятия разности является очевидно избыточным, поскольку известно, что любая из этих операций выражается через две других. Тем не менее, Кодд в свое время решил включить все три операции, исходя из интуитивных потребностей потенциального пользователя системы реляционных БД, далекого от математики.

Другие проблемы связаны с операцией взятия прямого произведения двух отношений. В теории множеств прямое произведение может быть получено для любых двух множеств, и элементами результирующего множества являются пары, составленные из элементов первого и второго множеств. Поскольку отношения являются множествами, то и для любых двух отношений возможно получение прямого произведения. Но результат не будет отношением! Элементами результата будут являться не кортежи, а пары кортежей.

Поэтому в реляционной алгебре используется специализированная форма операции взятия прямого произведения - расширенное прямое произведение отношений. При взятии расширенного прямого произведения двух отношений элементом результирующего отношения является кортеж, являющийся конкатенацией (или слиянием) одного кортежа первого отношения и одного кортежа второго отношения.

Но теперь возникает второй вопрос - как получить корректно сформированный заголовок отношения-результата? Очевидно, что проблемой может быть именование атрибутов результирующего отношения, если отношения-операнды обладают одноименными атрибутами.

Эти соображения приводят к появлению понятия ***совместимости по взятию расширенного прямого произведения***. Два отношения совместимы по взятию прямого произведения в том и только в том случае, если множества имен атрибутов этих отношений не пересекаются. Любые два отношения могут быть сделаны совместимыми по взятию прямого произведения путем применения операции переименования к одному из этих отношений.

Следует заметить, что операция взятия прямого произведения не является слишком осмысленной на практике. Во-первых, мощность ее результата очень велика даже при допустимых мощностях операндов, а во-вторых, результат операции не более информативен, чем взятые в совокупности операнды. Как мы увидим немного ниже, основной смысл включения операции расширенного прямого произведения в состав реляционной алгебры состоит в том, что на ее основе определяется действительно полезная операция соединения.

По поводу теоретико-множественных операций реляционной алгебры следует еще заметить, что все четыре операции являются ассоциативными. Т. е., если обозначить через OP любую из четырех операций, то $(A\ OP\ B)\ OP\ C = A\ (B\ OP\ C)$, и, следовательно, без введения двусмысленности можно писать $A\ OP\ B\ OP\ C$ (A, B и C - отношения, обладающие свойствами, требуемыми для корректного выполнения соответствующей операции). Все операции, кроме взятия разности, являются коммутативными, т.е. $A\ OP\ B = B\ OP\ A$.

5.1.4. Специальные реляционные операции.

В этом подразделе мы несколько подробнее рассмотрим специальные реляционные операции реляционной алгебры: ограничение, проекция, соединение и деление.

Операция ограничения.

Операция ограничения требует наличия двух операндов: ограничиваемого отношения и простого условия ограничения. Простое условие ограничения может иметь либо вид ($a\ comp\_op\ b$), где a и b - имена атрибутов ограничиваемого отношения, для которых осмысленна операция сравнения $comp\_op$, либо вид ($a\ comp\_op\ const$), где a - имя атрибута ограничиваемого отношения, а $const$ - литерально заданная константа.

В результате выполнения операции ограничения производится отношение, заголовок которого совпадает с заголовком отношения-операнда, а в тело входят те кортежи отношения-операнда, для которых значением условия ограничения является $true$.

Пусть $UNION$ обозначает операцию объединения, $INTERSECT$ - операцию пересечения, а $MINUS$ - операцию взятия разности. Для обозначения операции ограничения будем использовать конструкцию $A\ WHERE\ comp$, где A - ограничиваемое отношение, а $comp$ - простое условие сравнения. Пусть $comp_1$ и $comp_2$ - два простых условия ограничения. Тогда по определению:

- $A\ WHERE\ comp_1\ AND\ comp_2$ обозначает то же самое, что и ($A\ WHERE\ comp_1$) $INTERSECT$ ($A\ WHERE\ comp_2$)
- $A\ WHERE\ comp_1\ OR\ comp_2$ обозначает то же самое, что и ($A\ WHERE\ comp_1$) $UNION$ ($A\ WHERE\ comp_2$)
- $A\ WHERE\ NOT\ comp_1$ обозначает то же самое, что и $A\ MINUS$ ($A\ WHERE\ comp_1$)

С использованием этих определений можно использовать операции ограничения, в которых условием ограничения является произвольное булевское выражение, составленное из простых условий с использованием логических связок AND, OR, NOT и скобок.

На интуитивном уровне операцию ограничения лучше всего представлять как взятие некоторой "горизонтальной" вырезки из отношения-операнда.

Операция взятия проекции.

Операция взятия проекции также требует наличия двух операндов - проецируемого отношения А и списка имен атрибутов, входящих в заголовок отношения A.

Результатом проекции отношения A по списку атрибутов $a_1, a_2, ..., a_n$ является отношение, с заголовком, определяемым множеством атрибутов $a_1, a_2, ..., a_n$, и с телом, состоящим из кортежей вида $< a_1 : v_1, a_2 : v_2, ..., a_n : v_n >$ таких, что в отношении A имеется кортеж, атрибут a_1 которого имеет значение v_1, атрибут a_2 имеет значение $v_2, ...$, атрибут a_n имеет значение v_n. Тем самым, при выполнении операции проекции выделяется "вертикальная" вырезка отношения-операнда с естественным уничтожением потенциально возникающих кортежей-дубликатов.

Операция соединения отношений.

Общая операция соединения (называемая также соединением по условию) требует наличия двух операндов - соединяемых отношений и третьего операнда - простого условия. Пусть соединяются отношения A и B. Как и в случае операции ограничения, условие соединения $comp$ имеет вид либо ($a\ comp\_op\ b$), либо ($a\ comp\_op\ const$), где a и b - имена атрибутов отношений A и B, $const$ - литерально заданная константа, а $comp\_op$ - допустимая в данном контексте операция сравнения.

Тогда по определению результатом операции сравнения является отношение, получаемое путем выполнения операции ограничения по условию $comp$ прямого произведения отношений A и B.

Если внимательно осмыслить это определение, то станет ясно, что в общем случае применение условия соединения существенно уменьшит мощность результата промежуточного прямого произведения отношений-операндов только в том случае, когда условие соединения имеет вид

($a\ comp\_op\ b$), где a и b - имена атрибутов разных отношений-операндов. Поэтому на практике обычно считают реальными операциями соединения именно те операции, которые основываются на условии соединения приведенного вида.

Хотя операция соединение в нашей интерпретации не является примитивной (поскольку она определяется с использованием прямого произведения и проекции), в силу особой практической важности она включается в базовый набор операций реляционной алгебры. Заметим также, что в практических реализациях соединение обычно не выполняется именно как ограничение прямого произведения. Имеются более эффективные алгоритмы, гарантирующие получение такого же результата.

Имеется важный частный случай соединения - эквисоединение и простое, но важное расширение операции эквисоединения - естественное соединение. Операция соединения называется ***операцией эквисоединения***, если условие соединения имеет вид ($a = b$), где a и b - атрибуты разных операндов соединения. Этот случай важен потому, что (a) он часто встречается на практике, и (b) для него существуют эффективные алгоритмы реализации.

Операция естественного соединения применяется к паре отношений A и B, обладающих (возможно составным) общим атрибутом c (т.е. атрибутом с одним и тем же именем и определенным на одном и том же домене). Пусть ab обозначает объединение заголовков отношений A и B. Тогда естественное соединение A и B - это спроектированный на ab результат эквисоединения A и B по A/c и BBC. Если вспомнить введенное нами в конце предыдущей главы определение внешнего ключа отношения, то должно стать понятно, что основной смысл операции естественного соединения - возможность восстановления сложной сущности, декомпозированной по причине требования первой нормальной формы. Операция естественного соединения не включается прямо в состав набора операций реляционной алгебры, но она имеет очень важное практическое значение.

Операция деления отношений.

Эта операция наименее очевидна из всех операций реляционной алгебры и поэтому нуждается в более подробном объяснении. Пусть заданы два отношения - A с заголовком $\{a_1, a_2, ..., a_n, b_1, b_2, ..., b_m\}$ и B с заголовком $\{b_1, b_2, ..., b_m\}$. Будем считать, что атрибут b_i отношения A и атрибут b_i отношения B не только обладают одним и тем же именем, но и определены на одном и том же домене. Назовем множество атрибутов $\{a_i\}$ составным атрибутом a, а множество атрибутов $\{b_i\}$ - составным атрибутом b. После этого будем говорить о реляционном делении бинарного отношения $A(a,b)$ на унарное отношение $B(b)$.

Результатом деления A на B является унарное отношение $C(a)$, состоящее из кортежей v таких, что в отношении A имеются кортежи $< v, w >$ такие, что множество значений $\{w\}$ включает множество значений атрибута b в отношении B.

Предположим, что в базе данных сотрудников поддерживаются два отношения:

```
СОТРУДНИКИ (ИМЯ, ОТД_НОМЕР)
ИМЕНА (ИМЯ)
```

причем унарное отношение *ИМЕНА* содержит все фамилии, которыми обладают сотрудники организации. Тогда после выполнения операции реляционного деления отношения *СОТРУДНИКИ* на отношение *ИМЕНА* будет получено унарное отношение, содержащее номера отделов, сотрудники которых обладают всеми возможными в этой организации именами.

5.2. Реляционное исчисление.

Предположим, что мы работаем с базой данных, обладающей схемой

```
СОТРУДНИКИ (СОТР_НОМ, СОТР_ИМЯ, СОТР_ЗАРП, ОТД_НОМ)
ОТДЕЛЫ (ОТД_НОМ, ОТД_КОЛ, ОТД_НАЧ)
```

и хотим узнать имена и номера сотрудников, являющихся начальниками отделов с количеством сотрудников больше 50.

Если бы для формулировки такого запроса использовалась реляционная алгебра, то мы получили бы алгебраическое выражение, которое читалось бы, например, следующим образом:

- выполнить соединение отношений *СОТРУДНИКИ* и *ОТДЕЛЫ* по условию $СОТР\_НОМ = ОТД\_НАЧ$;
- ограничить полученное отношение по условию $ОТД\_КОЛ > 50$;
- спроецировать результат предыдущей операции на атрибут $СОТР\_ИМЯ$, $СОТР\_НОМ$.

Мы четко сформулировали последовательность шагов выполнения запроса, каждый из которых соответствует одной реляционной операции. Если же сформулировать тот же запрос с использованием реляционного исчисления, которому посвящается этот раздел, то мы получили бы формулу, которую можно было бы прочитать, например, следующим образом: Выдать $СОТР\_ИМЯ$ и $СОТР\_НОМ$ для сотрудников таких, что существует отдел с таким же значением $ОТД\_НАЧ$ и значением $ОТД\_КОЛ$ большим 50.

Во второй формулировке мы указали лишь характеристики результирующего отношения, но ничего не сказали о способе его формирования. В этом случае система должна сама решить, какие операции и в каком порядке нужно выполнить над отношениями *СОТРУДНИКИ* и *ОТДЕЛЫ*. Обычно говорят, что алгебраическая формулировка является процедурной, т.е. задающей правила выполнения запроса, а логическая - описательной (или декларативной), поскольку она всего лишь описывает свойства желаемого результата. Как мы указывали в начале лекции, на самом деле эти два механизма эквивалентны и существуют не очень сложные правила преобразования одного формализма в другой.

5.2.1. Кортежные переменные и правильно построенные формулы.

Реляционное исчисление является прикладной ветвью формального механизма исчисления предикатов первого порядка. Базисными понятиями исчисления являются понятие переменной с определенной для нее областью допустимых значений и понятие правильно построенной формулы, опирающейся на переменные, предикаты и кванторы.

В зависимости от того, что является областью определения переменной, различаются исчисление кортежей и исчисление доменов. В исчислении кортежей областями определения переменных являются отношения базы данных, т.е. допустимым значением каждой переменной является кортеж некоторого отношения. В исчислении доменов областями определения переменных являются домены, на которых определены атрибуты отношений базы данных, т.е. допустимым значением каждой переменной является значение некоторого домена. Мы рассмотрим более подробно исчисление кортежей, а в конце лекции коротко опишем особенности исчисления доменов.

В отличие от раздела, посвященного реляционной алгебре, в этом разделе нам не удастся избежать использования некоторого конкретного синтаксиса, который мы, тем не менее, формально определять не будем. Необходимые синтаксические конструкции будут вводиться по мере необходимости. В совокупности, используемый синтаксис близок, но не полностью совпадает с синтаксисом языка баз данных QUEL, который долгое время являлся основным языком СУБД Ingres.

Для определения кортежной переменной используется оператор *RANGE*. Например, для того, чтобы определить переменную *СОТРУДНИК*, областью определения которой является отношение *СОТРУДНИКИ*, нужно употребить конструкцию

```
RANGE СОТРУДНИК IS СОТРУДНИКИ
```

Как мы уже говорили, из этого определения следует, что в любой момент времени переменная *СОТРУДНИК* представляет некоторый кортеж отношения *СОТРУДНИКИ*. При использовании кортежных переменных в формулах можно ссылаться на значение атрибута переменной

(это аналогично тому, как, например, при программировании на языке Си можно сослаться на значение поля структурной переменной). Например, для того, чтобы сослаться на значение атрибута *СОТР_ИМЯ* переменной *СОТРУДНИК*, нужно употребить конструкцию *СОТРУДНИК.СОТР_ИМЯ*.

Правильно построенные формулы (*WFF* - Well-Formed Formula) служат для выражения условий, накладываемых на кортежные переменные. Основой *WFF* являются простые сравнения (*comparison*), представляющие собой операции сравнения скалярных значений (значений атрибутов переменных или литерально заданных констант). Например, конструкция *СОТРУДНИК.СОТР_НОМ* = 140 является простым сравнением. По определению, простое сравнение является *WFF*, а *WFF*, заключенная в круглые скобки, является простым сравнением.

Более сложные варианты *WFF* строятся с помощью логических связок *NOT*, *AND*, *OR* и *IF...THEN*. Так, если *form* - *WFF*, а *comp* - простое сравнение, то *NOT form*, *comp AND form*, *comp OR form* и *IF comp THEN form* являются *WFF*.

Наконец, допускается построение WFF с помощью кванторов. Если *form* - это *WFF*, в которой участвует переменная *var*, то конструкции *EXISTS var* (*form*) и *FORALL var* (*form*) представляют *WFF*.

Переменные, входящие в *WFF*, могут быть свободными или связанными. Все переменные, входящие в *WFF*, при построении которой не использовались кванторы, являются свободными. Фактически, это означает, что если для какого-то набора значений свободных кортежных переменных при вычислении *WFF* получено значение *true*, то эти значения кортежных переменных могут входить в результирующее отношение. Если же имя переменной использовано сразу после квантора при построении *WFF* вида *EXISTS var* (*form*) или *FORALL var* (*form*), то в этой *WFF* и во всех *WFF*, построенных с ее участием, *var* - это связанная переменная. Это означает, что такая переменная не видна за пределами минимальной *WFF*, связавшей эту переменную. При вычислении значения такой *WFF* используется не одно значение связанной переменной, а вся ее область определения.

Пусть *СОТР*1 и *СОТР*2 - две кортежные переменные, определенные на отношении *СОТРУДНИКИ*. Тогда

```
WFF EXISTS СОТР2 (СОТР1.СОТР_ЗАРП > СОТР2.СОТР_ЗАРП)
```

для текущего кортежа переменной *СОТР*1 принимает значение *true* в том и только в том случае, если во всем отношении *СОТРУДНИКИ* найдется кортеж (связанный с переменной *СОТР*2) такой, что значение его атрибута *СОТР_ЗАРП* удовлетворяет внутреннему условию сравнения.

```
WFF FORALL СОТР2 (СОТР1.СОТР_ЗАРП > СОТР2.СОТР_ЗАРП)
```

для текущего кортежа переменной *СОТР*1 принимает значение *true* в том и только в том случае, если для всех кортежей отношения СОТРУДНИКИ (связанных с переменной *СОТР*2) значения атрибута *СОТР_ЗАРП* удовлетворяют условию сравнения.

На самом деле, правильнее говорить не о свободных и связанных переменных, а о свободных и связанных вхождениях переменных. Легко видеть, что если переменная *var* является связанной в *WFF form*, то во всех *WFF*, включающих данную, может использоваться имя переменной *var*, которая может быть свободной или связанной, но в любом случае не имеет никакого отношения к вхождению переменной *var* в *WFF form*. Вот пример:

```
EXISTS СОТР2 (СОТР1.СОТР_ОТД_НОМ = СОТР2.СОТР_ОТД_НОМ) AND
   FORALL СОТР2 (СОТР1.СОТР_ЗАРП > СОТР2.СОТР_ЗАРП)
```

Здесь мы имеем два связанных вхождения переменной $COTP2$ с совершенно разным смыслом.

5.2.2. Целевые списки и выражения реляционного исчисления.

Итак, WFF обеспечивают средства формулировки условия выборки из отношений БД. Чтобы можно было использовать исчисление для реальной работы с БД, требуется еще один компонент, который определяет набор и имена столбцов результирующего отношения. Этот компонент называется целевым списком ($target\_list$).

Целевой список строится из целевых элементов, каждый из которых может иметь следующий вид:

- $var.attr$, где var - имя свободной переменной соответствующей WFF, а $attr$ - имя атрибута отношения, на котором определена переменная var;
- var, что эквивалентно наличию подсписка $var.attr_1, var.attr_2, ..., var.attr_n$, где $attr_1, attr_2, ..., attr_n$ включает имена всех атрибутов определяющего отношения;
- $new\_name = var.attr$; new_name - новое имя соответствующего атрибута результирующего отношения.

Последний вариант требуется в тех случаях, когда в WFF используются несколько свободных переменных с одинаковой областью определения.

Выражением реляционного исчисления кортежей называется конструкция вида $target\_list\ WHERE\ WFF$. Значением выражения является отношение, тело которого определяется WFF, а набор атрибутов и их имена - целевым списком.

5.2.3. Реляционное исчисление доменов.

В исчислении доменов областью определения переменных являются не отношения, а домены. Применительно к базе данных $СОТРУДНИКИ - ОТДЕЛЫ$ можно говорить, например, о доменных переменных $ИМЯ$ (значения - допустимые имена) или $СОТР\_НОМ$ (значения - допустимые номера сотрудников).

Основным формальным отличием исчисления доменов от исчисления кортежей является наличие дополнительного набора предикатов, позволяющих выражать так называемые условия членства. Если R - это n-арное отношение с атрибутами $a_1, a_2, ..., a_n$, то условие членства имеет вид

$$R(a_{i1} : v_{i1}, a_{i2} : v_{i2}, ..., a_{im} : v_{im}) \quad (m \le n)$$

где v_{ij} - это либо литерально задаваемая константа, либо имя кортежной переменной. Условие членства принимает значение *true* в том и только в том случае, если в отношении R существует кортеж, содержащий указанные значения указанных атрибутов. Если v_{ij} - константа, то на атрибут a_{ij} задается жесткое условие, не зависящее от текущих значений доменных переменных; если же v_{ij} - имя доменной переменной, то условие членства может принимать разные значения при разных значениях этой переменной.

Во всех остальных отношениях формулы и выражения исчисления доменов выглядят похожими на формулы и выражения исчисления кортежей. В частности, конечно, различаются свободные и связанные вхождения доменных переменных.

Для примера сформулируем с использованием исчисления доменов запрос "Выдать номера и имена сотрудников, не получающих минимальную заработную плату" (будем считать для простоты, что мы определили доменные переменные, имена которых совпадают с именами атрибутов отношения $СОТРУДНИКИ$, а в случае, когда требуется несколько доменных переменных, определенных на одном домене, мы будем добавлять в конце имени цифры):

```
СОТР_НОМ, СОТР_ИМЯ WHERE EXISTS СОТР_ЗАРП1
   (СОТРУДНИКИ (СОРТ_ЗАРП1) AND
   СОТРУДНИКИ (СОТР_НОМ, СОТР_ИМЯ, СОТР_ЗАРП) AND
```

```
        СОРТ_ЗАРП > СОТР_ЗАРП1)
```

Реляционное исчисление доменов является основой большинства языков запросов, основанных на использовании форм. В частности, на этом исчислении базировался известный язык Query-by-Example, который был первым (и наиболее интересным) языком в семействе языков, основанных на табличных формах.

§6. Проектирование реляционных БД.

При проектировании базы данных решаются две основных проблемы:

- Каким образом отобразить объекты предметной области в абстрактные объекты модели данных, чтобы это отображение не противоречило семантике предметной области и было по возможности лучшим (эффективным, удобным и т.д.)? Часто эту проблему называют проблемой логического проектирования баз данных.
- Как обеспечить эффективность выполнения запросов к базе данных, т.е. каким образом, имея в виду особенности конкретной СУБД, расположить данные во внешней памяти, создание каких дополнительных структур (например, индексов) потребовать и т.д.? Эту проблему называют проблемой физического проектирования баз данных.

В случае реляционных баз данных трудно представить какие-либо общие рецепты по части физического проектирования. Здесь слишком много зависит от используемой СУБД. Например, при работе с СУБД Ingres можно выбирать один из предлагаемых способов физической организации отношений, при работе с System R следовало бы прежде всего подумать о кластеризации отношений и требуемом наборе индексов и т.д. Поэтому мы ограничимся вопросами логического проектирования реляционных баз данных, которые существенны при использовании любой реляционной СУБД.

Более того, мы не будем касаться очень важного аспекта проектирования - определения ограничений целостности (за исключением ограничения первичного ключа). Дело в том, что при использовании СУБД с развитыми механизмами ограничений целостности (например, SQL-ориентированных систем) трудно предложить какой-либо общий подход к определению ограничений целостности. Эти ограничения могут иметь очень общий вид, и их формулировка пока относится скорее к области искусства, чем инженерного мастерства. Самое большее, что предлагается по этому поводу в литературе, это автоматическая проверка непротиворечивости набора ограничений целостности.

Так что будем считать, что проблема проектирования реляционной базы данных состоит в обоснованном принятии решений о том,

- из каких отношений должна состоять БД и
- какие атрибуты должны быть у этих отношений.

6.1. Проектирование реляционных баз данных с использованием нормализации.

Сначала будет рассмотрен классический подход, при котором весь процесс проектирования производится в терминах реляционной модели данных методом последовательных приближений к удовлетворительному набору схем отношений. Исходной точкой является представление предметной области в виде одного или нескольких отношений, и на каждом шаге проектирования производится некоторый набор схем отношений, обладающих лучшими свойствами. Процесс проектирования представляет собой процесс нормализации схем отношений, причем каждая следующая нормальная форма обладает свойствами лучшими, чем предыдущая.

Каждой нормальной форме соответствует некоторый определенный набор ограничений, и отношение находится в некоторой нормальной форме, если удовлетворяет свойственному ей набору ограничений. Примером набора ограничений является ограничение первой нормальной формы - значения всех атрибутов отношения атомарны. Поскольку требование первой нормальной формы является базовым требованием классической реляционной модели данных, мы будем считать, что исходный набор отношений уже соответствует этому требованию.

В теории реляционных баз данных 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 нормальная форма проекции-соединения (5NF или PJ/NF).

Основные свойства нормальных форм:

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

В основе процесса проектирования лежит метод нормализации, декомпозиция отношения, находящегося в предыдущей нормальной форме, в два или более отношения, удовлетворяющих требованиям следующей нормальной формы.

Наиболее важные на практике нормальные формы отношений основываются на фундаментальном в теории реляционных баз данных понятии ***функциональной зависимости***. Для дальнейшего изложения нам потребуются несколько определений.

Определение 1. Функциональная зависимость. В отношении R атрибут Y функционально зависит от атрибута X (X и Y могут быть составными) в том и только в том случае, если каждому значению X соответствует в точности одно значение Y : $R.X$ (r) $R.Y$.

Определение 2. Полная функциональная зависимость. Функциональная зависимость $R.X$ (r) $R.Y$ называется полной, если атрибут Y не зависит функционально от любого точного подмножества X .

Определение 3. Транзитивная функциональная зависимость. Функциональная зависимость $R.X \rightarrow R.Y$ называется транзитивной, если существует такой атрибут Z , что имеются функциональные зависимости $R.X \rightarrow R.Z$ и $R.Z \rightarrow R.Y$ и отсутствует функциональная зависимость $R.Z \rightarrow R.X$. (При отсутствии последнего требования мы имели бы "неинтересные" транзитивные зависимости в любом отношении, обладающем несколькими ключами.)

Определение 4. Неключевой атрибут. Неключевым атрибутом называется любой атрибут отношения, не входящий в состав первичного ключа (в частности, первичного).

Определение 5. Взаимно независимые атрибуты. Два или более атрибута взаимно независимы, если ни один из этих атрибутов не является функционально зависимым от других.

6.1.1. Вторая нормальная форма.

Рассмотрим следующий пример схемы отношения:

```
СОТРУДНИКИ-ОТДЕЛЫ-ПРОЕКТЫ
(СОТР_НОМ, СОТР_ЗАРП, ОТД_НОМ, ПРО_НОМ, СОТР_ЗАДАН)
```

Первичный ключ:

```
СОТР_НОМ, ПРО_НОМ
```

Функциональные зависимости:

```
СОТР_НОМ -> СОТР_ЗАРП
СОТР_НОМ -> ОТД_НОМ
ОТД_НОМ -> СОТР_ЗАРП
СОТР_НОМ, ПРО_НОМ -> СОТР_ЗАДАН
```

Как видно, хотя первичным ключом является составной атрибут $СОТР\_НОМ$, $ПРО\_НОМ$, атрибуты $СОТР\_ЗАРП$ и $ОТД\_НОМ$ функционально зависят от части первичного ключа, атрибута $СОТР\_НОМ$. В результате мы не сможем вставить в отношение

СОТРУДНИКИ – ОТДЕЛЫ – ПРОЕКТЫ кортеж, описывающий сотрудника, который еще не выполняет никакого проекта (первичный ключ не может содержать неопределенное значение). При удалении кортежа мы не только разрушаем связь данного сотрудника с данным проектом, но утрачиваем информацию о том, что он работает в некотором отделе. При переводе сотрудника в другой отдел мы будем вынуждены модифицировать все кортежи, описывающие этого сотрудника, или получим несогласованный результат. Такие неприятные явления называются аномалиями схемы отношения. Они устраняются путем нормализации.

Определение 6. Вторая нормальная форма (в этом определении предполагается, что единственным ключом отношения является первичный ключ). Отношение R находится во второй нормальной форме (2NF) в том и только в том случае, когда находится в 1NF, и каждый неключевой атрибут полностью зависит от первичного ключа.

Можно произвести следующую декомпозицию отношения *СОТРУДНИКИ – ОТДЕЛЫ – ПРОЕКТЫ* в два отношения *СОТРУДНИКИ – ОТДЕЛЫ* и *СОТРУДНИКИ – ПРОЕКТЫ* :

```
СОТРУДНИКИ-ОТДЕЛЫ (СОТР_НОМ, СОТР_ЗАРП, ОТД_НОМ)
```

Первичный ключ:

```
СОТР_НОМ
```

Функциональные зависимости:

```
СОТР_НОМ -> СОТР_ЗАРП
СОТР_НОМ -> ОТД_НОМ
ОТД_НОМ -> СОТР_ЗАРП

СОТРУДНИКИ-ПРОЕКТЫ (СОТР_НОМ, ПРО_НОМ, СОТР_ЗАДАН)
```

Первичный ключ:

```
СОТР_НОМ, ПРО_НОМ
```

Функциональные зависимости:

```
СОТР_НОМ, ПРО_НОМ -> СОТР_ЗАДАН
```

Каждое из этих двух отношений находится в 2NF, и в них устранены отмеченные выше аномалии (легко проверить, что все указанные операции выполняются без проблем).

Если допустить наличие нескольких ключей, то определение 6 примет следующий вид:

Определение 6~. Отношение R находится во второй нормальной форме (2NF) в том и только в том случае, когда оно находится в 1NF, и каждый неключевой атрибут полностью зависит от каждого ключа R.

Здесь и далее мы не будем приводить примеры для отношений с несколькими ключами. Они слишком громоздки и относятся к ситуациям, редко встречающимся на практике.

6.1.2. Третья нормальная форма.

Рассмотрим еще раз отношение *СОТРУДНИКИ – ОТДЕЛЫ* , находящееся в 2NF. Заметим, что функциональная зависимость $СОТР\_НОМ -> СОТР\_ЗАРП$ является транзитивной; она является следствием функциональных зависимостей $СОТР\_НОМ -> ОТД\_НОМ$ и $ОТД\_НОМ -> СОТР\_ЗАРП$. Другими словами, заработная плата сотрудника на самом деле является характеристикой не сотрудника, а отдела, в котором он работает (это не очень естественное предположение, но достаточное для примера).

В результате мы не сможем занести в базу данных информацию, характеризующую заработную плату отдела, до тех пор, пока в этом отделе не появится хотя бы один сотрудник (первич-

ный ключ не может содержать неопределенное значение). При удалении кортежа, описывающего последнего сотрудника данного отдела, мы лишимся информации о заработной плате отдела. Чтобы согласованным образом изменить заработную плату отдела, мы будем вынуждены предварительно найти все кортежи, описывающие сотрудников этого отдела. Т.е. в отношении *СОТРУДНИКИ – ОТДЕЛЫ* по-прежнему существуют аномалии. Их можно устранить путем дальнейшей нормализации.

Определение 7. Третья нормальная форма (снова определение дается в предположении существования единственного ключа). Отношение R находится в третьей нормальной форме (3NF) в том и только в том случае, если находится в 2NF и каждый неключевой атрибут нетранзитивно зависит от первичного ключа.

Можно произвести декомпозицию отношения *СОТРУДНИКИ – ОТДЕЛЫ* в два отношения *СОТРУДНИКИ* и *ОТДЕЛЫ* :

```
СОТРУДНИКИ (СОТР_НОМ, ОТД_НОМ)
```

Первичный ключ:

```
СОТР_НОМ
```

Функциональные зависимости:

```
СОТР_НОМ -> ОТД_НОМ

ОТДЕЛЫ (ОТД_НОМ, СОТР_ЗАРП)
```

Первичный ключ:

```
ОТД_НОМ
```

Функциональные зависимости:

```
ОТД_НОМ -> СОТР_ЗАРП
```

Каждое из этих двух отношений находится в 3NF и свободно от отмеченных аномалий.

Если отказаться от того ограничения, что отношение обладает единственным ключом, то определение 3NF примет следующую форму:

Определение 7~. Отношение R находится в третьей нормальной форме (3NF) в том и только в том случае, если находится в 1NF, и каждый неключевой атрибут не является транзитивно зависимым от какого-либо ключа R .

На практике третья нормальная форма схем отношений достаточна в большинстве случаев, и приведением к третьей нормальной форме процесс проектирования реляционной базы данных обычно заканчивается. Однако иногда полезно продолжить процесс нормализации.

6.1.3. Нормальная форма Бойса-Кодда.

Рассмотрим следующий пример схемы отношения:

```
СОТРУДНИКИ-ПРОЕКТЫ (СОТР_НОМ, СОТР_ИМЯ, ПРО_НОМ, СОТР_ЗАДАН)
```

Возможные ключи:

```
СОТР_НОМ, ПРО_НОМ
СОТР_ИМЯ, ПРО_НОМ
```

Функциональные зависимости:

```
СОТР_НОМ -> СОТР_ИМЯ
СОТР_НОМ -> ПРО_НОМ
СОТР_ИМЯ -> СОТР_НОМ
```

```
СОТР_ИМЯ -> ПРО_НОМ
СОТР_НОМ, ПРО_НОМ -> СОТР_ЗАДАН
СОТР_ИМЯ, ПРО_НОМ -> СОТР_ЗАДАН
```

В этом примере мы предполагаем, что личность сотрудника полностью определяется как его номером, так и именем (это снова не очень жизненное предположение, но достаточное для примера).

В соответствии с определением 7~ отношение *СОТРУДНИКИ – ПРОЕКТЫ* находится в 3NF. Однако тот факт, что имеются функциональные зависимости атрибутов отношения от атрибута, являющегося частью первичного ключа, приводит к аномалиям. Например, для того, чтобы изменить имя сотрудника с данным номером согласованным образом, нам потребуется модифицировать все кортежи, включающие его номер.

Определение 8. Детерминант. Детерминант - любой атрибут, от которого полностью функционально зависит некоторый другой атрибут.

Определение 9. Нормальная форма Бойса-Кодда. Отношение *R* находится в нормальной форме Бойса-Кодда (BCNF) в том и только в том случае, если каждый детерминант является возможным ключом.

Очевидно, что это требование не выполнено для отношения *СОТРУДНИКИ – ПРОЕКТЫ* . Можно произвести его декомпозицию к отношениям *СОТРУДНИКИ* и *СОТРУДНИКИ – ПРОЕКТЫ* :

```
СОТРУДНИКИ (СОТР_НОМ, СОТР_ИМЯ)
```

Возможные ключи:

```
СОТР_НОМ
СОТР_ИМЯ
```

Функциональные зависимости:

```
СОТР_НОМ -> СОТР_ИМЯ
СОТР_ИМЯ -> СОТР_НОМ

СОТРУДНИКИ-ПРОЕКТЫ (СОТР_НОМ, ПРО_НОМ, СОТР_ЗАДАН)
```

Возможный ключ:

```
СОТР_НОМ, ПРО_НОМ
```

Функциональные зависимости:

```
СОТР_НОМ, ПРО_НОМ -> СОТР_ЗАДАН
```

Возможна альтернативная декомпозиция, если выбрать за основу *СОТР _ ИМЯ* . В обоих случаях получаемые отношения *СОТРУДНИКИ* и *СОТРУДНИКИ – ПРОЕКТЫ* находятся в BCNF, и им не свойственны отмеченные аномалии.

6.1.4. Четвертая нормальная форма.

Рассмотрим пример следующей схемы отношения:

```
ПРОЕКТЫ (ПРО_НОМ,ПРО_СОТР, ПРО_ЗАДАН)
```

Отношение *ПРОЕКТЫ* содержит номера проектов, для каждого проекта список сотрудников, которые могут выполнять проект, и список заданий, предусматриваемых проектом. Сотрудники могут участвовать в нескольких проектах, и разные проекты могут включать одинаковые задания.

Каждый кортеж отношения связывает некоторый проект с сотрудником, участвующим в этом проекте, и заданием, который сотрудник выполняет в рамках данного проекта (мы предполагаем, что любой сотрудник, участвующий в проекте, выполняет все задания, предусмотренные этим проектом). По причине сформулированных выше условий единственным возможным ключем отношения является составной атрибут $ПРО\_НОМ$, $ПРО\_СОТР$, $ПРО\_ЗАДАН$, и нет никаких других детерминантов. Следовательно, отношение $ПРОЕКТЫ$ находится в BCNF. Но при этом оно обладает недостатками: если, например, некоторый сотрудник присоединяется к данному проекту, необходимо вставить в отношение $ПРОЕКТЫ$ столько кортежей, сколько заданий в нем предусмотрено.

Определение 10. ***Многозначные зависимости.***

В отношении $R(A,B,C)$ существует многозначная зависимость $R.A->->R.B$ 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.

В отношении $ПРОЕКТЫ$ существуют следующие две многозначные зависимости:

```
ПРО_НОМ -> -> ПРО_СОТР
ПРО_НОМ -> -> ПРО_ЗАДАН
```

Легко показать, 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 отношению $ПРОЕКТЫ$, основывается на следующей теореме:

Теорема. ***Фейджина.***

Отношение $R(A,B,C)$ можно спроецировать без потерь в отношения $R_1(A,B)$ и $R_2(A,C)$ в том и только в том случае, когда существует MVD $A->->B \mid C$.

Под проецированием без потерь понимается такой способ декомпозиции отношения, при котором исходное отношение полностью и без избыточности восстанавливается путем естественного соединения полученных отношений.

Определение 11. ***Четвертая нормальная форма.***

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 отношения $ПРОЕКТЫ$ в два отношения $ПРОЕКТЫ-СОТРУДНИКИ$ и $ПРОЕКТЫ-ЗАДАНИЯ$:

```
ПРОЕКТЫ-СОТРУДНИКИ (ПРО_НОМ, ПРО_СОТР)
ПРОЕКТЫ-ЗАДАНИЯ (ПРО_НОМ, ПРО_ЗАДАН)
```

Оба эти отношения находятся в 4NF и свободны от отмеченных аномалий.

6.1.5. Пятая нормальная форма.

Во всех рассмотренных до этого момента нормализациях производилась декомпозиция одного отношения в два. Иногда это сделать не удается, но возможна декомпозиция в большее число отношений, каждое из которых обладает лучшими свойствами.

Рассмотрим, например, отношение

```
СОТРУДНИКИ-ОТДЕЛЫ-ПРОЕКТЫ (СОТР_НОМ, ОТД_НОМ, ПРО_НОМ)
```

Предположим, что один и тот же сотрудник может работать в нескольких отделах и работать в каждом отделе над несколькими проектами. Первичным ключем этого отношения является полная совокупность его атрибутов, отсутствуют функциональные и многозначные зависимости.

Поэтому отношение находится в 4NF. Однако в нем могут существовать аномалии, которые можно устранить путем декомпозиции в три отношения.

Определение 12. ***Зависимость соединения*****.**

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 $X,Y,...,Z$.

Определение 13. ***Пятая нормальная форма*****.**

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.

Введем следующие имена составных атрибутов:

```
СО = {СОТР_НОМ, ОТД_НОМ}
СП = {СОТР_НОМ, ПРО_НОМ}
ОП = {ОТД_НОМ, ПРО_НОМ}
```

Предположим, что в отношении *СОТРУДНИКИ – ОТДЕЛЫ – ПРОЕКТЫ* существует зависимость соединения:

```
* (СО, СП, ОП)
```

На примерах легко показать, что при вставках и удалениях кортежей могут возникнуть проблемы. Их можно устранить путем декомпозиции исходного отношения в три новых отношения:

```
СОТРУДНИКИ-ОТДЕЛЫ (СОТР_НОМ, ОТД_НОМ)
СОТРУДНИКИ-ПРОЕКТЫ (СОТР_НОМ, ПРО_НОМ)
ОТДЕЛЫ-ПРОЕКТЫ (ОТД_НОМ, ПРО_НОМ)
```

Пятая нормальная форма - это последняя нормальная форма, которую можно получить путем декомпозиции. Ее условия достаточно нетривиальны, и на практике 5NF не используется. Заметим, что зависимость соединения является обобщением как многозначной зависимости, так и функциональной зависимости.

6.2. Семантическое моделирование данных, ER-диаграммы.

Широкое распространение реляционных СУБД и их использование в самых разнообразных приложениях показывает, что реляционная модель данных достаточна для моделирования предметных областей. Однако проектирование реляционной базы данных в терминах отношений на основе кратко рассмотренного нами механизма нормализации часто представляет собой очень сложный и неудобный для проектировщика процесс.

При этом проявляется ограниченность реляционной модели данных в следующих аспектах:

- Модель не предоставляет достаточных средств для представления смысла данных. Семантика реальной предметной области должна независимым от модели способом представляться в голове проектировщика. В частности, это относится к упоминавшейся нами проблеме представления ограничений целостности.
- Для многих приложений трудно моделировать предметную область на основе плоских таблиц. В ряде случаев на самой начальной стадии проектирования проектировщику приходится производить насилие над собой, чтобы описать предметную область в виде одной (возможно, даже ненормализованной) таблицы.
- Хотя весь процесс проектирования происходит на основе учета зависимостей, реляционная модель не предоставляет каких-либо средств для представления этих зависимостей.
- Несмотря на то, что процесс проектирования начинается с выделения некоторых существенных для приложения объектов предметной области ("сущностей") и выявления связей между этими сущностями, реляционная модель данных не предлагает какого-либо аппарата для разделения сущностей и связей.

6.2.1. Семантические модели данных.

Потребности проектировщиков баз данных в более удобных и мощных средствах моделирования предметной области вызвали к жизни направление семантических моделей данных. При том, что любая развитая семантическая модель данных, как и реляционная модель, включает структурную, манипуляционную и целостную части, главным назначением семантических моделей является обеспечение возможности выражения семантики данных.

Прежде, чем мы коротко рассмотрим особенности одной из распространенных семантических моделей, остановимся на их возможных применениях.

Наиболее часто на практике семантическое моделирование используется на первой стадии проектирования базы данных. При этом в терминах семантической модели производится концептуальная схема базы данных, которая затем вручную преобразуется к реляционной (или какой-либо другой) схеме. Этот процесс выполняется под управлением методик, в которых достаточно четко оговорены все этапы такого преобразования.

Менее часто реализуется автоматизированная компиляция концептуальной схемы в реляционную. При этом известны два подхода: на основе явного представления концептуальной схемы как исходной информации для компилятора и построения интегрированных систем проектирования с автоматизированным созданием концептуальной схемы на основе интервью с экспертами предметной области. И в том, и в другом случае в результате производится реляционная схема базы данных в третьей нормальной форме (более точно следовало бы сказать, что автору неизвестны системы, обеспечивающие более высокий уровень нормализации).

Наконец, третья возможность, которая еще не вышла (или только выходит) за пределы исследовательских и экспериментальных проектов, - это работа с базой данных в семантической модели, т.е. СУБД, основанные на семантических моделях данных. При этом снова рассматриваются два варианта: обеспечение пользовательского интерфейса на основе семантической модели данных с автоматическим отображением конструкций в реляционную модель данных (это задача примерно такого же уровня сложности, как автоматическая компиляция концептуальной схемы базы данных в реляционную схему) и прямая реализация СУБД, основанная на какой-либо семантической модели данных. Наиболее близко ко второму подходу находятся современные объектно-ориентированные СУБД, модели данных которых по многим параметрам близки к семантическим моделям (хотя в некоторых аспектах они более мощны, а в некоторых - более слабы).

6.2.2. Основные понятия модели Entity-Relationship (Сущность-Связи).

Далее мы кратко рассмотрим некоторые черты одной из наиболее популярных семантических моделей данных - модель "Сущность-Связи" (часто ее называют кратко ER-моделью).

На использовании разновидностей ER-модели основано большинство современных подходов к проектированию баз данных (главным образом, реляционных). Модель была предложена Ченом (Chen) в 1976 г. Моделирование предметной области базируется на использовании графических диаграмм, включающих небольшое число разнородных компонентов. В связи с наглядностью представления концептуальных схем баз данных ER-модели получили широкое распространение в системах CASE, поддерживающих автоматизированное проектирование реляционных баз данных. Среди множества разновидностей ER-моделей одна из наиболее развитых применяется в системе CASE фирмы ORACLE. Ее мы и рассмотрим. Более точно, мы сосредоточимся на структурной части этой модели.

Основными понятиями ER-модели являются сущность, связь и атрибут.

Сущность - это реальный или представляемый объект, информация о котором должна сохраняться и быть доступна. В диаграммах ER-модели сущность представляется в виде прямоугольника, содержащего имя сущности. При этом имя сущности - это имя типа, а не некоторого конкретного экземпляра этого типа. Для большей выразительности и лучшего понимания имя сущности может сопровождаться примерами конкретных объектов этого типа.

Ниже изображена сущность *АЭРОПОРТ* с примерными объектами Шереметьево и Хитроу:

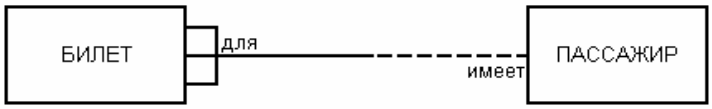
Каждый экземпляр сущности должен быть отличим от любого другого экземпляра той же сущности (это требование в некотором роде аналогично требованию отсутствия кортежей-дубликатов в реляционных таблицах).

Связь - это графически изображаемая ассоциация, устанавливаемая между двумя сущностями. Эта ассоциация всегда является бинарной и может существовать между двумя разными сущностями или между сущностью и ей же самой (рекурсивная связь). В любой связи выделяются два конца (в соответствии с существующей парой связываемых сущностей), на каждом из которых указывается имя конца связи, степень конца связи (сколько экземпляров данной сущности связывается), обязательность связи (т.е. любой ли экземпляр данной сущности должен участвовать в данной связи).

Связь представляется в виде линии, связывающей две сущности или ведущей от сущности к ней же самой. При это в месте "стыковки" связи с сущностью используются трехточечный вход в прямоугольник сущности, если для этой сущности в связи могут использоваться много (many) экземпляров сущности, и одноточечный вход, если в связи может участвовать только один экземпляр сущности. Обязательный конец связи изображается сплошной линией, а необязательный - прерывистой линией.

Как и сущность, связь - это типовое понятие, все экземпляры обеих пар связываемых сущностей подчиняются правилам связывания.

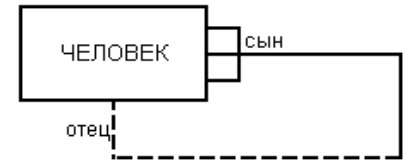
В изображенном ниже примере связь между сущностями *БИЛЕТ* и *ПАССАЖИР* связывает билеты и пассажиров. При том конец сущности с именем "для" позволяет связывать с одним пассажиром более одного билета, причем каждый билет должен быть связан с каким-либо пассажиром. Конец сущности с именем "имеет" означает, что каждый билет может принадлежать только одному пассажиру, причем пассажир не обязан иметь хотя бы один билет.



Лаконичной устной трактовкой изображенной диаграммы является следующая:

- Каждый *БИЛЕТ* предназначен для одного и только одного *ПАССАЖИРА*;
- Каждый *ПАССАЖИР* может иметь один или более *БИЛЕТОВ*.

На следующем примере изображена рекурсивная связь, связывающая сущность *ЧЕЛОВЕК* с ней же самой. Конец связи с именем "сын" определяет тот факт, что у одного отца может быть более чем один сын. Конец связи с именем "отец" означает, что не у каждого человека могут быть сыновья.



Лаконичной устной трактовкой изображенной диаграммы является следующая:

- Каждый *ЧЕЛОВЕК* является сыном одного и только одного *ЧЕЛОВЕКА*;
- Каждый *ЧЕЛОВЕК* может являться отцом для одного или более *ЛЮДЕЙ* ("*ЧЕЛОВЕКОВ*").

Атрибутом сущности является любая деталь, которая служит для уточнения, идентификации, классификации, числовой характеристики или выражения состояния сущности. Имена атрибутов заносятся в прямоугольник, изображающий сущность, под именем сущности и изображаются малыми буквами, возможно, с примерами.

Пример:

Уникальным идентификатором сущности является атрибут, комбинация атрибутов, комбинация связей или комбинация связей и атрибутов, уникально отличающая любой экземпляр сущности от других экземпляров сущности того же типа.

6.2.3. Нормальные формы ER-схем.

Как и в реляционных схемах баз данных, в ER-схемах вводится понятие нормальных форм, причем их смысл очень близко соответствует смыслу реляционных нормальных форм. Заметим, что формулировки нормальных форм ER-схем делают более понятным смысл нормализации реляционных схем. Мы приведем только очень краткие и неформальные определения трех первых нормальных форм.

В первой нормальной форме ER-схемы устраняются повторяющиеся атрибуты или группы атрибутов, т.е. производится выявление неявных сущностей, "замаскированных" под атрибуты.

Во второй нормальной форме устраняются атрибуты, зависящие только от части уникального идентификатора. Эта часть уникального идентификатора определяет отдельную сущность.

В третьей нормальной форме устраняются атрибуты, зависящие от атрибутов, не входящих в уникальный идентификатор. Эти атрибуты являются основой отдельной сущности.

6.2.4. Более сложные элементы ER-модели.

Мы остановились только на самых основных и наиболее очевидных понятиях ER-модели данных. К числу более сложных элементов модели относятся следующие:

- Подтипы и супертипы сущностей. Как в языках программирования с развитыми типовыми системами (например, в языках объектно-ориентированного программирования), вводится возможность наследования типа сущности, исходя из одного или нескольких супертипов. Интересные нюансы связаны с необходимостью графического изображения этого механизма.
- Связи "many-to-many". Иногда бывает необходимо связывать сущности таким образом, что с обоих концов связи могут присутствовать несколько экземпляров сущности (например, все члены кооператива сообща владеют имуществом кооператива). Для этого вводится разновидность связи "многие-со-многими".
- Уточняемые степени связи. Иногда бывает полезно определить возможное количество экземпляров сущности, участвующих в данной связи (например, служащему разрешается участвовать не более, чем в трех проектах одновременно). Для выражения этого семантического ограничения разрешается указывать на конце связи ее максимальную или обязательную степень.
- Каскадные удаления экземпляров сущностей. Некоторые связи бывают настолько сильными (конечно, в случае связи "один-ко-многим"), что при удалении опорного экземпляра сущности (соответствующего концу связи "один") нужно удалить и все экземпляры сущности, соответствующие концу связи "многие". Соответствующее требование "каскадного удаления" можно сформулировать при определении сущности.
- Домены. Как и в случае реляционной модели данных бывает полезна возможность определения потенциально допустимого множества значений атрибута сущности (домена).

Эти и другие более сложные элементы модели данных "Сущность-Связи" делают ее существенно более мощной, но одновременно несколько усложняют ее использование. Конечно, при реальном использовании ER-диаграмм для проектирования баз данных необходимо ознакомиться со всеми возможностями.

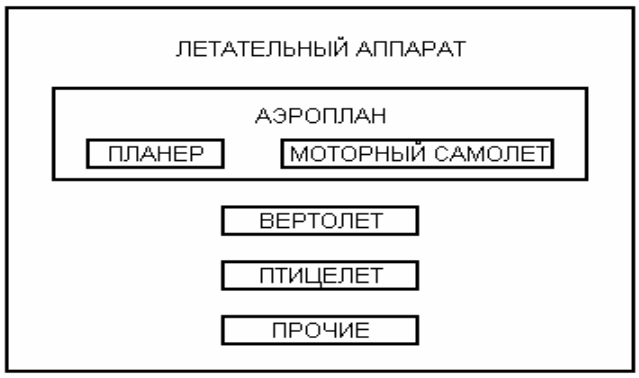
В нашей лекции мы немного подробнее разберем только один из упомянутых элементов - подтип сущности.

Сущность может быть расщеплена на два или более взаимно исключающих подтипа, каждый из которых включает общие атрибуты и/или связи. Эти общие атрибуты и/или связи явно

определяются один раз на более высоком уровне. В подтипах могут определяться собственные атрибуты и/или связи. В принципе подтипизация может продолжаться на более низких уровнях, но опыт показывает, что в большинстве случаев оказывается достаточно двух-трех уровней.

Сущность, на основе которой определяются подтипы, называется супертипом. Подтипы должны образовывать полное множество, т.е. любой экземпляр супертипа должен относиться к некоторому подтипу. Иногда для полноты приходится определять дополнительный подтип *ПРОЧИЕ*.

Пример: Супертип *ЛЕТАТЕЛЬНЫЙ АППАРАТ*.



Как полагается это читать? От супертипа: *ЛЕТАТЕЛЬНЫЙ АППАРАТ*, который должен быть *АЭРОПЛАНОМ*, *ВЕРТОЛЕТОМ*, *ПТИЦЕЛЕТОМ* или *ДРУГИМ ЛЕТАТЕЛЬНЫМ АППАРАТОМ*. От подтипа: *ВЕРТОЛЕТ*, который относится к типу *ЛЕТАТЕЛЬНОГО АППАРАТА*. От подтипа, который является одновременно супертипа: *АЭРОПЛАН*, который относится к типу *ЛЕТАТЕЛЬНОГО АППАРАТА* и должен быть *ПЛАНЕРОМ* или *МОТОРНЫМ САМОЛЕТОМ*.

Иногда удобно иметь два или более разных разбиения сущности на подтипы. Например, сущность *ЧЕЛОВЕК* может быть разбита на подтипы по профессиональному признаку (*ПРОГРАММИСТ*, *ДОЯРКА* и т.д.), а может - по половому признаку (*МУЖЧИНА*, *ЖЕНЩИНА*).

6.2.5. Получение реляционной схемы из ER-схемы.

Шаг 1. Каждая простая сущность превращается в таблицу. Простая сущность - сущность, не являющаяся подтипом и не имеющая подтипов. Имя сущности становится именем таблицы.

Шаг 2. Каждый атрибут становится возможным столбцом с тем же именем; может выбираться более точный формат. Столбцы, соответствующие необязательным атрибутам, могут содержать неопределенные значения; столбцы, соответствующие обязательным атрибутам, - не могут.

Шаг 3. Компоненты уникального идентификатора сущности превращаются в первичный ключ таблицы. Если имеется несколько возможных уникальных идентификатора, выбирается наиболее используемый. Если в состав уникального идентификатора входят связи, к числу столбцов первичного ключа добавляется копия уникального идентификатора сущности, находящейся на дальнем конце связи (этот процесс может продолжаться рекурсивно). Для именования этих столбцов используются имена концов связей и/или имена сущностей.

Шаг 4. Связи многие-к-одному (и один-к-одному) становятся внешними ключами. Т.е. делается копия уникального идентификатора с конца связи "один", и соответствующие столбцы составляют внешний ключ. Необязательные связи соответствуют столбцам, допускающим неопределенные значения; обязательные связи - столбцам, не допускающим неопределенные значения.

Шаг 5. Индексы создаются для первичного ключа (уникальный индекс), внешних ключей и тех атрибутов, на которых предполагается в основном базировать запросы.

Шаг 6. Если в концептуальной схеме присутствовали подтипы, то возможны два способа:

- все подтипы в одной таблице (а)
- для каждого подтипа - отдельная таблица (б)

При применении способа (а) таблица создается для наиболее внешнего супертипа, а для подтипов могут создаваться представления. В таблицу добавляется по крайней мере один столбец, содержащий код ТИПА; он становится частью первичного ключа.

При использовании метода (б) для каждого подтипа первого уровня (для более нижних - представления) супертип воссоздается с помощью представления UNION (из всех таблиц подтипов выбираются общие столбцы - столбцы супертипа).

Все в одной таблице	**Таблица - на подтип**
Преимущества	
1. Все хранится вместе 2. Легкий доступ к супертипам и подтипу 3. Требуется меньше таблиц	1. Более ясны правила подтипов 2. Программы работают только с нужными таблицами
Недостатки	
1. Слишком общее решение 2. Требуется дополнительная логика работы с разными наборами столбцов и разными ограничениями 3. Потенциальное узкое место (в связи с блокировками) 4.Столбцы подтипов должны быть необязательными 5. В некоторых СУБД для хранения неопределенных значений требуется дополнительная память	1. Слишком много таблиц 2. Смущающие столбцы UNION 3. Потенциальная потеря производительности при работе через UNION 4. Над супертипом невозможны модификации

Шаг 7. Имеется два способа работы при наличии исключающих связей:

- общий домен (а)
- явные внешние ключи (б)

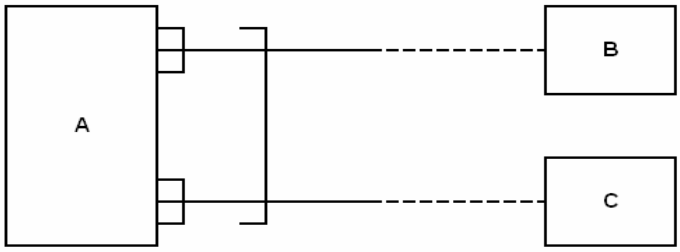
Если остающиеся внешние ключи все в одном домене, т.е. имеют общий формат (способ (а)), то создаются два столбца: идентификатор связи и идентификатор сущности. Столбец идентификатора связи используется для различения связей, покрываемых дугой исключения. Столбец идентификатора сущности используется для хранения значений уникального идентификатора сущности на дальнем конце соответствующей связи.

Если результирующие внешние ключи не относятся к одному домену, то для каждой связи, покрываемой дугой исключения, создаются явные столбцы внешних ключей; все эти столбцы могут содержать неопределенные значения.

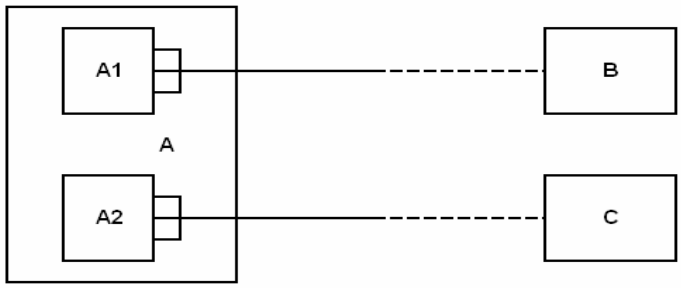
Общий домен	**Явные внешние ключи**
Преимущества	
Нужно только два столбца	Условия соединения - явные
Недостатки	
Оба дополнительных атрибута должны использоваться в соединениях	Слишком много столбцов

Альтернативные модели сущностей:

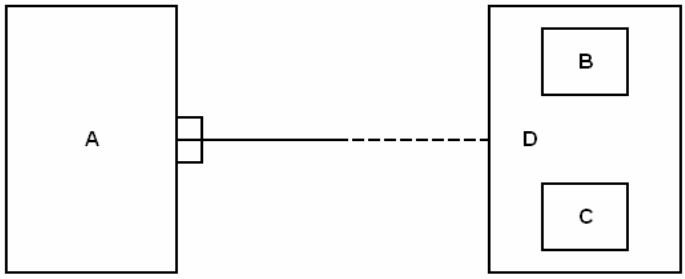
Вариант 1 (плохой):



Вариант 2 (существенно лучше, если подтипы действительно существуют):



Вариант 3 (годится при наличии осмысленного супертипа D):



Две классические экспериментальные системы

§7. System R: общая организация системы, основы языка SQL.

Система управления реляционными базами данных System R разрабатывалась в исследовательской лаборатории фирмы IBM в 1975-1979 г.г. Эта работа оказала революционизирующее влияние на развитие теории и практики реляционных систем во всем мире. Именно System R практически доказала жизнеспособность реляционного подхода к управлению базами данных.

После успешного завершения работ по созданию этой системы и получения экспериментальных результатов ее использования был разработан целый ряд коммерчески доступных реляционных систем, в том числе и на основе непосредственного развития System R (возможности одной из коммерчески доступных реляционных систем - DB2 - описываются в переведенной на русский язык книге К. Дейта "Руководство по реляционной СУБД DB2). Исключительно важен опыт, приобретенный при разработке этой системы. Практически во всех более поздних реляционных СУБД в той или иной степени используются методы, примененные в System R.

После завершения разработки System R фирма IBM активно продолжала работы по реляционным СУБД, причем в нескольких направлениях. Первое направление мы уже отмечали - разработка коммерческих реляционных СУБД. Второе направление - построение распределенной реляционной СУБД на основе идей System R. Экспериментальный вариант такой системы, System R*, был успешно разработан в IBM. Эта работа также существенно обогатила опыт ис-

следователей и разработчиков распределенных СУБД. Наконец, третье направление – исследование и разработка реляционных систем, предназначенных для нетрадиционных приложений.

Организации СУБД System R посвящена обширная библиография. Для информации мы приводим ее в конце этой лекции. Хотя официально разработка этой системы началась в 1975 г., первые публикации, связанные с этой системой, появились еще в 1974 г. В частности, в одной из первых публикаций была предложена основа базового языка System R SQL (тогда этот язык назывался SEQUEL, и до сих пор многие называют его именно так; кстати, разработчики System R (а теперь и компания Oracle) рекомендуют произносить название SQL именно как SEQUEL). Поскольку публикации появлялись по ходу практической реализации системы, каждая из них отражает состояние дел (идейное и практическое) именно на том этапе работы, когда была написана соответствующая статья. Некоторые идеи и представления, естественно, изменялись по ходу работы. Сравнительно законченное представление о системе в целом дают только заключительные публикации. С другой стороны, многие интересные моменты совершенно не отражены в этих последних статьях, и мы постараемся привести более полный обзор идей и методов, примененных в System R. При этом мы будем останавливаться и на некоторых возможных альтернативных решениях, которые были найдены разработчиками System R, но практически не были использованы.

7.1. Используемая терминология.

Что касается общей терминологии реляционного подхода, мы будем активно пользоваться соответствующими терминами. К таким терминам относятся названия реляционных операций – селекция, проекция, соединение; названия теоретико-множественных операций - объединение, пересечение, разность и т.д.

В тех случаях, когда традиционная терминология System R расходится с общепринятой, мы будем отдавать предпочтение терминологии System R. В частности, это касается использования термина "поле отношения" вместо "атрибут отношения".

В самой System R при переходе к коммерческим системам также произошла некоторая смена терминологии. В частности, в некоторых последних публикациях появилась тенденция к употреблению более привычных в среде пользователей IBM терминов: файл, запись и т.д. Мы будем использовать термины System R, более близкие реляционным системам. Далее мы опишем некоторые основные термины System R, исходя при этом в основном не из теоретических соображений, а стремясь отразить практические аспекты соответствующих понятий.

Базовым понятием System R является понятие таблицы (приближенный к реализации эквивалент основного понятия реляционного подхода отношение; иногда, в зависимости от контекста, мы будем использовать и этот термин). Таблица - это некоторая регулярная структура, состоящая из конечного набора однотипных записей - кортежей. Каждый кортеж одного отношения состоит из конечного (и одинакового) числа полей кортежа, причем i-тое поле каждого кортежа одного отношения может содержать данные только одного типа, и набор допустимых типов данных в System R предопределен и фиксирован. В силу регулярности структуры отношения понятие поля кортежа расширяется до понятия поля таблицы. I-тое поле таблицы можно трактовать как набор одноместных кортежей, полученных выборкой i-тых полей из каждого кортежа этой таблицы, т.е. в общепринятой терминологии как проекцию отношения на i-тый атрибут. В терминологию System R не входит понятие домена, оно заменяется здесь понятием типа поля, т.е. типом данных, хранение которых в данном поле допускается (это не вполне эквивалентная замена, но такова реальность System R).

Таблицы, составляющие базу данных System R, могут физически храниться в одном или нескольких сегментах, которые проще всего понимать как файлы внешней памяти (и это вполне соответствует действительности). Сегменты разбиваются на страницы, в которых располагаются кортежи отношений и вспомогательные служебные структуры данных индексы. Соответственно, каждый сегмент содержит две группы страниц - страницы данных и страницы индексной информации. Страницы каждой группы имеют фиксированный размер, но страницы с индексной информацией меньше по размеру, чем страницы данных. В страницах данных могут распола-

гаться кортежи более, чем одного отношения (это очень важное свойство физической организации баз данных System R; следующие из этой организации преимущества разъясним позже).

Этим, конечно, не исчерпывается набор понятий System R, но остальные термины мы будем пояснять по ходу изложения, поскольку для этого требуется соответствующий понятийный контекст.

7.2. Основные цели System R и их связь с архитектурой системы.

Основными целями разработчиков System R являлись следующие:

- обеспечить ненавигационный интерфейс высокого уровня пользователя с системой, позволяющий достичь независимости данных и дать возможность пользователям работать максимально эффективно;
- обеспечить многообразие допустимых способов использования СУБД, включая программируемые транзакции, диалоговые транзакции и генерацию отчетов;
- поддерживать динамически изменяемую среду баз данных, в которой отношения, индексы, представления, транзакции и другие объекты могут легко добавляться и уничтожаться без приостановки нормального функционирования системы;
- обеспечить возможность параллельной работы с одной базой данных многих пользователей с допущением параллельной модификации объектов базы данных при наличии необходимых средств защиты целостности базы данных;
- обеспечить средства восстановления согласованного состояния баз данных после разного рода сбоев аппаратуры или программного обеспечения;
- обеспечить гибкий механизм, позволяющий определять различные представления хранимых данных и ограничивать этими представлениями доступ пользователей к базе данных по выборке и модификации на основе механизма авторизации;
- обеспечить производительность системы при выполнении упомянутых функций, сопоставимую с производительностью существующих СУБД низкого уровня.

Прежде всего отметим, что в основном поставленные цели при разработке System R были достигнуты. Рассмотрим теперь, какими средствами были достигнуты эти цели, и как более точно можно интерпретировать их в контексте System R.

Основой System R является реляционный язык SQL. Иногда его называют языком запросов или языком манипулирования данными, но на самом деле его возможности гораздо шире. Средствами SQL (с соответствующей системной поддержкой) решаются многие из поставлен-ных целей. Язык SQL включает средства динамической компиляции запросов, на основе чего возможно построение диалоговых систем обработки запросов. Допускается динамическая параметризация статически откомпилированных запросов, в результате чего возможно построение эффективных (не требующих динамической компиляции) диалоговых систем со стандартными наборами (параметризуемых) запросов.

Средствами SQL определяются все доступные пользователю объекты баз данных: таблицы, индексы, представления. Имеются средства уничтожения любого такого объекта. Соответствующие операторы языка могут выполняться в любой момент, и возможность выполнения операции данным пользователем зависит от ранее предоставленных ему прав.

Что касается целостности баз данных, то в System R под целостным состоянием базы данных понимается состояние, удовлетворяющее набору сохраняемых при базе данных предикатов целостности. Эти предикаты, называемые в System R условиями целостности (assertions), задаются также средствами языка SQL. Любое предложение языка выполняется в пределах некоторой транзакции - неделимой в смысле состояния базы данных последовательности предложений языка. Неделимость означает, что все изменения, произведенные в пределах одной транзакции либо целиком отображаются в состоянии базы данных, либо полностью в нем отсутствуют. Последняя возможность возникает при откате транзакции, который может произойти по инициативе пользователя (при выполнении соответствующего оператора SQL) или по инициативе системы.

Одной из причин отката транзакции по инициативе системы является как раз нарушение целостности базы данных в результате действий данной транзакции (другие возможные условия отката транзакции по инициативе системы мы рассмотрим позже). Язык SQL содержит средство

установки так называемых точек сохранения (savepoint). При инициируемом пользователем откате транзакции можно указать номер точки сохранения, выше которого откат не распространяется. Инициируемый системой откат транзакции производится до ближайшей точки сохранения, в которой условие, вызвавшее откат, уже отсутствует. В частности, откат инициированный по причине нарушения условия целостности, производится до ближайшей точки сохранения, в которой условия целостности соблюдены. (Заметим, что средства установки точек сохранения отсутствуют в коммерческих расширениях System R).

Естественно, что для реального выполнения отката транзакции необходимо запоминание некоторой информации о выполнении транзакции. В System R для этих и других целей используется специальный набор данных - журнал, в который помещаются записи обо всех меняющих состояние базы данных операциях всех транзакций. При откате транзакции происходит процесс обратного выполнения транзакции (undo), в ходе которого в обратном порядке выполняются все изменения, запомненные в журнале.

В языке SQL имеется средство определения так называемых условных воздействий (triggers), позволяющих автоматически поддерживать целостность базы данных при модификациях ее объектов. Условное воздействие - это каталогизированная операция модификации, для которой задано условие ее автоматического выполнения. Особенно существенно наличие такого аппарата в связи с наличием рассматриваемых ниже представлений базы данных, которыми может быть ограничен доступ к базе данных для ряда пользователей. Возможна ситуация, когда такие пользователи просто не могут соблюдать целостность базы данных без автоматического выполнения условных воздействий, поскольку они просто "не видят" всей базы данных и, в частности, не могут представить всех ограничений ее целостности. Заметим, что, за исключением ранних публикаций по System R, реализация механизма условных воздействий нигде не описывалась, хотя в принципе подходы к реализации достаточно понятны. Этот механизм не реализован в коммерческих системах, возникших на базе System R. Видимо, это связано с возникающими дополнительными непредсказуемыми для пользователей накладными расходами при выполнении транзакций.

Язык SQL содержит средства определения представлений. Представление - это запомненный именованный запрос на выборку данных (из одной или нескольких таблиц). Поскольку SQL - это реляционный язык, то результатом выполнения любого запроса на выборку является таблица, и поэтому концептуально можно относиться к любому представлению как к таблице (при определении представления можно, в частности, присвоить имена полям этой таблицы). В языке допускается использование ранее определенных представлений практически везде, где допускается использование таблиц (с некоторыми ограничениями по поводу возможностей модификации через представления). Наличие возможности определять представления в совокупности с развитой системой авторизации позволяет ограничить доступ некоторых пользователей к базе данных выделенным набором представлений.

Авторизация доступа к базе данных основана также на средствах SQL. При создании любого объекта базы данных выполняющий эту операцию пользователь становится полновластным владельцем этого объекта, т.е. может выполнять по отношению к этому объекту любую функцию из предопределенного набора. Далее этот пользователь может выполнить оператор SQL, означающий передачу всех его прав на этот объект (или их подмножества) любому другому пользователю. В частности, этому пользователю может быть передано право на передачу всех переданных ему прав (или их части) третьему пользователю и т.д. Одним из прав пользователя по отношению к объекту является право на изъятие у других пользователей всех или некоторых прав, которые ранее им были переданы. Эта операция распространяется транзитивно на всех дальнейших наследников этих прав.

Наличие в языке средств определения представлений и авторизации в принципе позволяет обойтись при эксплуатации System R без традиционного администратора баз данных, поскольку практически все системные действия производятся на основе средств SQL. Тем не менее, если организационно администратор баз данных требуется, то его работа достаточно упрощается за счет унифицированного набора средств управления. Кроме того, в System R каталоги баз дан-

ных поддерживаются также в виде таблиц, и к ним применены все запросы языка SQL. Заметим, что в коммерческих СУБД появился ряд дополнительных утилит, не связанных с языком SQL (например, утилиты сбора статистики или массовой загрузки базы данных), и в этих системах, видимо, без администратора базы данных не обойтись.

По части обеспечения параллельной работы многих пользователей с одной базой данных, основной подход System R состоит в том, что пользователь не обязан знать о наличии других, конкурирующих с ним за доступ к базе данных, пользователей, т.е. система ответственна за обеспечение изолированности пользователей с гарантией отсутствия их взаимного влияния в пределах транзакций. Из этого следует, во-первых, что в интерфейсе пользователя с системой (т.е. в языке SQL) не должно быть средств регулирования взаимодействий с другими пользователями и, во-вторых, что система должна обеспечить автоматическую сериализацию набора транзакций, т.е. обеспечить режим выполнения этого набора транзакций, эквивалентный по конечному результату некоторому последовательному выполнению этих транзакций. Эта проблема решается в System R за счет автоматического выполнения синхронизационных захватов по отношению ко всем изменяемым объектам базы данных. Имеется ряд тонкостей, связанных с такой синхронизацией, на которых мы остановимся ниже.

Одним из основных требований к СУБД вообще и к System R в частности является обеспечение надежности баз данных по отношению к различного рода сбоям. К таким сбоям могут относиться программные ошибки прикладного и системного уровня, сбои процессора, поломки внешних носителей и т.д. В частности, к одному из видов сбоев можно отнести упоминавшиеся выше нарушения целостности базы данных, и автоматический инициируемый системой откат транзакции - это системное средство восстановления базы данных после сбоев такого рода. Как мы отмечали, такое восстановление происходит путем обратного выполнения транзакции на основе информации о внесенных ею изменениях, запомненной в журнале. На информации журнала основано восстановление базы данных и после сбоев другого рода. Управление журнализацией и восстановлением в System R весьма интересно, применяемые методы в ряде случаев отличаются от методов, используемых в других СУБД.

Что касается естественных требований к эффективности системы, то здесь основные решения связаны со спецификой физической организации баз данных на внешней памяти, буферизацией используемых страниц базы данных в оперативной памяти и развитой техникой оптимизации запросов, сформулированных на SQL, производимой на стадии их компиляции.

Структурная организация System R вполне согласуется с поставленными при ее разработке целями и выбранными решениями. Основными структурными компонентами System R являются система управления реляционной памятью (Relational Storage System - *RSS*) и компилятор запросов языка SQL. *RSS* обеспечивает интерфейс довольно низкого, но достаточного для реализации SQL, уровня для доступа к хранимым в базе данным. Синхронизация транзакций, журнализация изменений и восстановление баз данных после сбоев также относятся к числу функций *RSS*. Компилятор запросов использует интерфейс *RSS* для доступа к разнообразной справочной информации (каталоги отношений, индексов, прав доступа, условий целостности, условных воздействий и т.д.) и производит рабочие программы, выполняемые в дальнейшем также с использованием интерфейса *RSS*. Таким образом, система естественно разделяется на два уровня - уровень управления памятью и синхронизацией, фактически, не зависящий от базового языка запросов системы, и языковый уровень (уровень SQL), на котором решается большинство проблем System R. Заметим, что эта независимость скорее условная, чем абсолютная: язык SQL можно заменить на другой язык, но он должен обладать примерно такой же семантикой.

Далее мы последовательно рассмотрим особенности организации *RSS*, процесс компиляции и оптимизации запросов и технику выполнения откомпилированных транзакций (включая отмеченную выше возможность динамической компиляции запросов).

7.3. Организация внешней памяти в базах данных System R.

Как мы отмечали, база данных System R располагается в одном или нескольких сегментах внешней памяти. Каждый сегмент состоит из страниц данных и страниц индексной информации. Размер страницы данных в сегменте может быть выбран равным либо 4, либо 32 килобай-

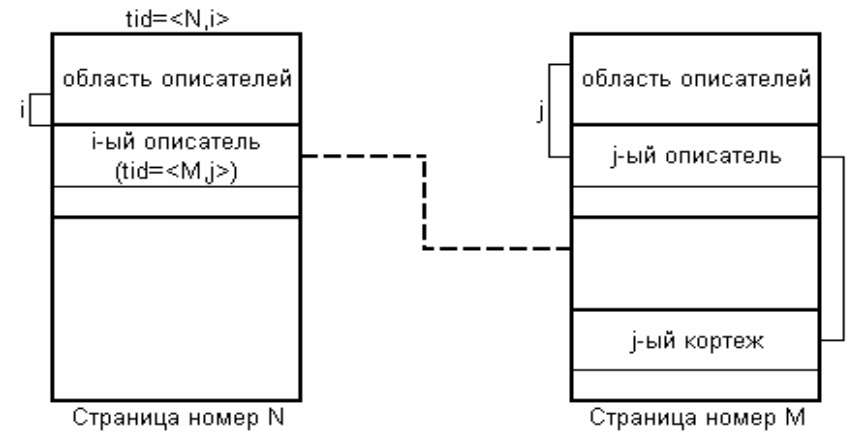
там; размер страницы индексной информации равен 512 байтам. Кроме того, при работе *RSS* поддерживается дополнительный набор данных для ведения журнала. Для повышения надежности журнала (а это наиболее критичная информация; при ее потере восстановление базы данных после сбоев невозможно) этот набор данных дублируется на двух внешних носителях.

В каждой странице данных хранятся кортежи одного или нескольких отношений. Фундаментальным понятием *RSS* является идентификатор кортежа (tuple identifier - *tid*). Гарантируется неизменяемость *tid* 'а во все время существования кортежа в базе данных независимо от перемещений кортежа внутри страницы и даже при перемещении кортежа в другую страницу. Реально *tid* представляет собой пару

```
<номер страницы, индекс описателя кортежа в странице>
```

При этом кортеж может реально располагаться в данной странице:

или в другой странице:



Во втором случае описатель кортежа содержит не координаты кортежа в данной странице, а *tid* , указывающий на реальное положение кортежа в другой странице. Легко видеть, что применение такого подхода позволяет ограничиться максимум одним уровнем косвенности.

Поскольку допускается нахождение в одной странице данных кортежей разных отношений, каждый кортеж должен, кроме содержательной части, включать служебную информацию, идентифицирующую отношение, которому принадлежит данный кортеж. Кроме того, в System R (точнее, в языке SQL) допускается динамическое добавление полей к существующим отношениям. При этом реально происходит лишь модификация описателя отношения в отношении-каталоге отношений. В существующем кортеже отношения новое поле возникает только при модификации этого кортежа, затрагивающей новое поле. Это позволяет избежать массовой перестройки хранимого отношения при добавлении к нему новых полей, но, естественно, требует хранения при кортеже дополнительной служебной информации, определяющей реальное число

полей в данном кортеже. (Заметим, что удалять существующие поля существующего отношения в SQL System R не разрешается).

На основе наличия неизменяемых во время существования кортежей *tid* 'ов в System R поддерживаются дополнительные управляющие структуры - индексы. Каждый индекс определен на одном или нескольких полях отношения, значения которых составляют его ключ, и позволяет производить прямой поиск по ключу кортежей (их *tid* 'ов) и последовательное сканирование отношения по индексу, начиная с указанного ключа, в порядке возрастания или убывания значений ключа. Некоторые индексы при их создании могут обладать атрибутом уникальности. В таком индексе не допускаются дубликаты ключа. Это единственное средство SQL указания системе первичного ключа отношения (фактически, набора первичного и всех альтернативных ключей отношения).

Для организации индексов в System R применяется техника B-деревьев. Каждый индекс занимает отдельный набор страниц, номер корневой страницы запоминается в описателе индекса. Использование B-деревьев позволяет достичь эффективности при прямом поиске, поскольку они в силу своей сильной ветвистости обладают небольшой глубиной. Кроме того, B-деревья сохраняют порядок ключей в листовых блоках иерархии, что позволяет производить последовательное сканирование отношения в порядке возрастания или убывания значений полей, на которых определен индекс. Фундаментальное свойство B-деревьев - автоматическая балансировка дерева - допускает произведение лишь локальных модификаций индекса при переполнениях и опустошениях страниц индекса. (Мы достаточно вольно используем здесь термин B-дерево. На самом деле в System R используется модифицированный по сравнению с исходным вариант B-деревьев, который называют B^*-, а иногда B^+-деревьями). В самих B-деревьях System R ничего особенного нет; более подробно мы на этом останавливаться не будем. Отметим только, что System R, насколько нам известно, была первой системой, в которой для организации индексов использовались B-деревья. Эту традицию соблюдает большинство реляционных систем, возникших после System R.

Видимо, наиболее важной особенностью физической организации баз данных в System R является возможность обеспечения кластеризации связанных кортежей одного или нескольких отношений. Под кластеризацией кортежей понимается физически близкое расположение (в пределах одной страницы данных) логически связанных кортежей. Обеспечение соответствующей кластеризации позволяет добиться высокой эффективности системы при выполнении выделенного класса запросов. В силу большой важности понятия кластеризации в System R и ее развитиях рассмотрим историю вопроса более подробно.

В окончательном варианте System R существует только одно средство определения условий кластеризации отношения - объявить до заполнения отношения один (и только один) индекс, определенный на полях этого отношения, кластеризованным. Тогда, если заполнение отношения кортежами производится в порядке возрастания или убывания значений полей кластеризации (в зависимости от атрибутики индекса), система физически располагает кортежи в страницах данных в том же порядке. Кроме того, в каждой странице данных кластеризованного отношения оставляется некоторое резервное свободное пространство. При последующих вставках кортежей в такое отношение система стремится поместить каждый кортеж в одну из страниц данных, в которых уже находятся кортежи этого отношения с такими же (или близкими) значениями полей кластеризации. Естественно, что поддерживать идеальную кластеризацию отношения можно только до определенного предела, пока не исчерпается резервная память в страницах. Далее этого предела степень кластеризации отношения начинает уменьшаться, и для восстановления идеальной кластеризации отношения требуется физическая реорганизация отношения (ее можно произвести средствами SQL).

Очевидным преимуществом кластеризации отношения является то, что при последовательном сканировании кластеризованного отношения с использованием кластеризованного индекса потребуется ровно столько чтений страниц данных с внешней памяти, сколько страниц занимают кортежи этого отношения. Следовательно, при правильно выбранных критериях кластериза-

ции запросы, связанные с заданием условий на полях кластеризации можно выполнить почти оптимально.

В ранних версиях System R существовал еще один способ физического доступа к кортежам отношения и, соответственно, еще один способ указания условия кластеризации с использованием так называемых связей (*links*). На уровне физического представления связь - это физическая ссылка (*tid*) из одного кортежа на другой (не обязательно одного отношения). В языке SEQUEL (до того момента, когда его стали называть SQL) существовали средства определения связей в иерархической манере: можно было объявить некоторое отношение родительским по отношению к тому же или другому отношению-потомку. При этом указывались поля родительского отношения и отношения-потомка, в соответствии со значениями которых образовывалась иерархия. Правила построения были очень простыми - проводились связи между кортежем родительского отношения ко всем кортежам отношения-потомка с теми же значениями полей связывания. На самом деле, все кортежи отношения-потомка с общим значением полей связывания образовывали кольцевой список, на который проводилась одна связь из соответствующего кортежа родительского отношения. Естественно, от отношения-родителя требовалась уникальность по отношению к значениям полей связывания.

Следует заметить, что мы описали способ использования механизма связей, который поддерживался в ранних версиях SEQUEL. В интерфейсе *RSS* System R этого периода допускалась возможность произвольного проведения связей без учета совпадения значений полей связывания. Тем самым, в системе в целом не использовались все возможности *RSS*, которые с избытком превосходили потребности организации иерархических бинарных связей по совпадению полей связывания.

Для одного отношения допускалось создание многих связей: кортеж отношения мог быть родителем нескольких иерархий и входить в несколько других иерархий в качестве потомка. При этом одна связь могла быть объявлена кластеризованной. Тогда система стремилась поместить в одну страницу данных все кортежи одной иерархии. При этом, естественно, использовалась возможность размещения в одной странице данных кортежей нескольких отношений. Основной смысл такой кластеризации заключался в возможности оптимизации выполнения некоторых запросов, включающих (экви)соединение двух связанных отношений в соответствии со значениями полей связывания.

В более поздних публикациях по System R упоминания о механизме связей исчезли, из чего можно заключить, что разработчики отказались от его использования. Думается, что основными причинами отказа от использования связей были следующие. Во-первых, средства построения связей, обеспечиваемые *RSS*, были очень низкого уровня, гораздо более низкого, чем средства поддержания индексов. Если при занесении кортежа *RSS* обеспечивала автоматическую коррекцию всех индексов, то для коррекции связей требовалось выполнить ряд дополнительных обращений к *RSS*, из-за чего время выполнения операции занесения кортежа, конечно, увеличивалось (то же касается операций удаления и модификации кортежа). Во-вторых, при реализации этого механизма, возникают дополнительные синхронизационные проблемы нижнего уровня (уровня совместного доступа к страницам данных). В частности, наличие прямых ссылок между страницами данных увеличивает вероятность возникновения синхронизационных тупиков. Наконец, в-третьих, все эти дополнительные накладные расходы не окупались предоставляемыми механизмом связей преимуществами. Действительно, максимального эффекта от использования связей можно достичь только при выполнении операции соединения двух кластеризованных по этой связи отношений, если поле соединения совпадает с полем связывания, и условия, накладываемые на родительское отношение, выделяют в нем ровно один кортеж. Очевидно, что такие запросы на практике редки. (Отметим, что приведенные соображения принадлежат автору и не излагались в публикациях по System R, так что на самом деле причины могли быть и другими.)

Кроме отношений и индексов при работе System R на внешней памяти могут располагаться еще и временные объекты - списки (*lists*). Список - это мгновенный снимок некоторой выборки с проекцией кортежей одного отношения, возможно, упорядоченный в соответствии со значени-

ями некоторых полей. Средства работы со списками имеются в интерфейсе *RSS*, но их, естественно, нет в SQL. Соответственно, эти средства используются только внутри системы при выполнении запросов (в частности, один из наиболее эффективных алгоритмов выполнения соединений основан на использовании отсортированных списков кортежей). Публикации по System R не дают точного представления о структурах данных, используемых при организации списков, но исходя из здравого смысла можно предположить, что они устроены не так, как отношения (например, для кортежа, входящего в список, не требуется адресация через *tid*), и что располагаются они во временных файлах (в случае сбоя системы все временные объекты пропадают).

7.4. Интерфейс RSS.

Мы опишем свое представление об интерфейсе *RSS*, которое не соответствует в точности ни одной из публикаций из приведенного списка литературы, а является скорее некоторой компиляцией, согласующейся с завершающими публикациями.

Как мы уже отмечали, на уровне *RSS* отсутствует именование объектов базы данных, употребляемое на уровне SQL. Вместо имен объектов используются их уникальные идентификаторы, являющиеся прямыми или косвенными адресами внутренних описателей объектов на внешней памяти для постоянных объектов или в оперативной памяти для временных объектов.

Можно выделить следующие группы операций:

- операции сканирования отношений и списков;
- операции создания и уничтожения постоянных и временных объектов базы данных;
- операции модификации отношений и списков;
- операция добавления поля к отношению;
- операции управления прохождением транзакции;
- операция явной синхронизации.

Операции группы сканирования позволяют последовательно в порядке, определяемом типом сканирования, прочитать кортежи отношения или списка, удовлетворяющие требуемым условиям. Группа включает операции *OPEN*, *NEXT* и *CLOSE*, означающие, соответственно, начало сканирования, требование следующего кортежа, удовлетворяющего условиям, и конец сканирования.

Для отношений возможны два режима сканирования: по отношению и по индексу. При сканировании по отношению единственным параметром операции *OPEN* является идентификатор отношения (включающий, кстати, и идентификатор сегмента, в котором это отношение хранится). По причине того, что в System R допускается размещение в одной странице данных кортежей нескольких отношений, сканирование по отношению предполагает последовательный просмотр всех страниц сегмента с выделением в них кортежей, входящих в данное отношение; это очень дорогой способ сканирования отношения. При этом порядок выборки кортежей определяется их физическим размещением в страницах сегмента, т.е. предопределен системой.

При открытии сканирования отношения по одному из его индексов в число параметров операции *OPEN* входит идентификатор индекса, определенного ранее на полях этого отношения. Кроме того, можно указать диапазон сканирования в терминах значений полей, составляющих ключ индекса. При открытии сканирования по индексу производится начальная установка указателя сканирования в позицию листа *B*-дерева индекса, соответствующую левой границе заданного диапазона. Процесс сканирования состоит в последовательном продвижении по листовым вершинам индекса до достижения правой границы диапазона сканирования с выборкой *tid*'ов и чтением соответствующих кортежей. Легко видеть, что если сканирование производится не по кластеризованному индексу, то в худшем случае может потребоваться столько чтений страниц данных из внешней памяти, сколько *tid*'ов было встречено, т.е. эффективность сканирования по индексу определяется "широтой" заданного диапазона сканирования. При этом, конечно, имеется то преимущество, что порядок сканирования соответствует порядку возрастания или убывания значений ключа индекса.

Наконец, при сканировании списка, как и при сканировании по отношению, единственным параметром операции *OPEN* является идентификатор списка, но, в отличии от сканирования по отношению это сканирование максимально эффективно: читаются только страницы, содержа-

щие кортежи из данного списка, и порядок сканирования совпадает с порядком занесения кортежей в список или порядком списка, если он упорядочен.

В результате успешного выполнения операции открытия сканирования (если нет ошибок в параметрах) вырабатывается и возвращается идентификатор сканирования, который используется в качестве входного параметра других операций этой группы.

Операция *NEXT* выполняет чтение следующего кортежа указанного сканирования, удовлетворяющего условиям данной операции. Условие представляет собой дизъюнктивную нормальную форму простых условий, относящихся к значениям указанных полей отношения. Простое условие - это условие вида

```
<идентификатор-поля op константа>
```

где *op* - операция сравнения <, <=, >, >=, = или !=. Общее условие является параметром операции *NEXT*. Семантика операции *NEXT* следующая: начиная с текущей позиции сканирования выбираются кортежи отношения в порядке, определяемом типом сканирования, до тех пор, пока не встретится кортеж, значения полей которого удовлетворяют указанному условию; этот кортеж и является результатом операции; если при выборке следующего кортежа достигается правая граница диапазона сканирования (правая граница значения ключа при сканировании по индексу или последний кортеж отношения или списка при сканировании без индекса), вырабатывается особый признак результата. После этого единственным разумным действием является закрытие сканирования - операция *CLOSE*.

Операция *CLOSE* может быть выполнена в данной транзакции по отношению к любому ранее открытому сканированию независимо от его состояния (т.е. независимо от того, достигнута ли при сканировании правая граница диапазона сканирования). Параметром операции является идентификатор сканирования, и ее выполнение приводит к тому, что этот идентификатор становится недействительным (и, соответственно, уничтожаются служебные структуры памяти *RSS*, относящиеся к данному сканированию).

Группа операций создания и уничтожения постоянных и временных объектов базы данных включает операции создания таблиц (*CREATE TABLE*), списков (*CREATE LIST*), индексов (*CREATE IMAGE*) и уничтожения любого из подобных объектов (*DROP TABLE*, *DROP LIST* и *DROP IMAGE*). Входным параметром операций создания таблиц и списков является специфи-катор структуры объекта, т.е. число полей объекта и спецификаторы их типов. Кроме того, при спецификации полей отношения указывается допущение или недопущение неопределенных зна-чений полей в кортежах этого отношения или списка (неопределенные значения кодируются специальным образом; любая операция сравнения константы данного типа с неопределенным значением по определению вырабатывает значение *FALSE*, кроме операции сравнения на сов-падение со специальной литеральной константой *NULL*). В результате выполнения этих опера-ций заводится описатель в служебном отношении описателей отношений или оперативной па-мяти (в зависимости от того, создается ли постоянный объект или временный), и вырабатывает-ся идентификатор объекта, который служит входным параметром других операций, относящих-ся к соответствующему объекту (в частности, параметром операции *OPEN* при открытии ска-нирования объекта).

Входными параметрами операции *CREATE IMAGE* являются идентификатор таблицы, для которой создается индекс, список номеров полей, значения которых составляют ключ индекса, и признаки упорядочения по возрастанию или убыванию для всех составляющих ключ полей. Кроме того, может быть указан признак уникальности индекса, т.е. запрещения наличия в данном индексе ключей-дубликатов. Если операция выполняется по отношению к пустой в этот момент таблице, то выполнение операции такое же простое, как и для операций создания таблиц и списков: создается описатель в служебном отношении описателей индексов и возвращается идентификатор индекса (который, в частности, используется в качестве входного параметра операции открытия сканирования отношения по индексу).

Если же к моменту создания индекса соответствующая таблица не пуста (а это допускается), то операция становится существенно более дорогостоящей, поскольку при ее выполнении происходит реальное создание *B* -дерева индекса, что требует по меньшей мере одного последовательного просмотра отношения. При этом, если создаваемый индекс имеет признак уникальности, то это контролируется при создании B-дерева, и если уникальность нарушается, то операция не выполняется (т.е. индекс не создается). Из этого следует, что хотя создание индексов в динамике не запрещается, более эффективно создавать все индексы на данной таблице до ее заполнения. Заметим, что создание кластеризованного индекса для непустого отношения запрещено, поскольку соответствующую кластеризацию отношения без его реструктуризации получить невозможно.

Операции *DROP TABLE* , *DROP LIST* и *DROP IMAGE* могут быть выполнены в любой момент независимо от состояния объектов. Выполнение операции приводит к уничтожению соответствующего объекта и, вследствие этого, недействительности его идентификатора.

Следует отметить, что массовые операции над постоянными объектами (*CREATE IMAGE* , *DROP TABLE* и *DROP IMAGE*) требуют дополнительных накладных расходов в связи с необходимостью обеспечения возможности откатов транзакции, в результате чего требуется выполнение массовых обратных действий. Особенно сильно это затрагивает операцию уничтожения непустых таблиц, поскольку требует журнализации всех содержащихся в них к моменту уничтожения кортежей. Поэтому, хотя уничтожение непустых таблиц и не запрещено, нужно иметь в виду, что это очень дорогостоящая операция.

Группа операций модификации отношений и списков включает операции вставки кортежа в отношение или список (*INSERT*), удаления кортежа из отношения (*DELETE*) и модификации кортежа в отношении (*UPDATE*).

Параметрами операции вставки кортежа являются идентификатор отношения или списка и набор значений полей кортежа. Среди значений полей могут быть литеральные неопределенные значения *NULL* . Естественно, при выполнении операции контролируется допустимость неопределенных значений в соответствующих полях. При занесении кортежа в кластеризованное отношение поиск места в сегменте под кортеж производится с использованием кластеризованного индекса: система пытается вставить кортеж в страницу данных, уже содержащую кортежи с теми же или близкими значениями полей кластеризации. При занесении кортежа в некластеризованное отношение место под кортеж выделяется в первой подходящей странице данных. Наконец, при вставке кортежа в список он помещается в конец списка.

При занесении кортежа в постоянное отношение производится коррекция всех индексов, определенных на этом отношении. Реально это выражается во вставке новой записи во все B-деревья индексов. При этом могут произойти переполнения одной или нескольких страниц индекса, что вызовет переливание части записей в соседние страницы или расщепление страниц. Если индекс определен с атрибутом уникальности, то проверяется соблюдение этого условия, и если оно нарушено, операция вставки считается невыполненной. Из этого видно, что операция вставки кортежа тем более накладна, чем больше индексов определено для данной таблицы (это относится и к операциям удаления и модификации кортежей).

В результате успешного выполнения операции вставки кортежа в отношение вырабатывается *tid* нового кортежа, который сообщается в качестве ответного параметра и может быть в дальнейшем использован как прямой параметр операций удаления и модификации кортежей отношения. При занесении кортежа в список значение *tid* 'а не вырабатывается (списки допускают только последовательное сканирование и добавление новых кортежей в конец, над ними нельзя определить индексов, поэтому косвенная адресация кортежей списков через *tid* 'ы не требуется).

Операции удаления и модификации кортежей допускаются только для кортежей постоянных таблиц. Естественно, что для выполнения этих операций необходимо идентифицировать соответствующий кортеж. Интерфейс *RSS* допускает два способа такой идентификации: с помощью *tid* 'а кортежа (явная адресация) и с использованием идентификатора открытого к этому времени

сканирования. Первый вариант возможен, поскольку *tid* кортежа сообщается как ответный параметр операции занесения кортежа в постоянную таблицу. При идентификации кортежа с помощью идентификатора сканирования имеется в виду кортеж, прочитанный с помощью последней операции *NEXT*. Если при такой идентификации выполнена операция *DELETE* или *UPDATE*, задевающая порядок сканирования (т.е. сканирование ведется по индексу и операция модификации меняет поле кортежа, входящее в состав ключа этого индекса), то текущий кортеж скана теряется, и его идентификатор нельзя использовать для идентификации кортежа до выполнения следующей операции *NEXT*.

Единственным параметром операции *DELETE* является идентификатор кортежа (tid или идентификатор сканирования). Параметры операции *UPDATE* включают, кроме этого идентификатора, спецификацию изменяемых полей кортежа (список номеров полей и их новых значений). Среди значений могут находиться литеральные изображения неопределенных значений, если соответствующие поля отношения допускают хранение неопределенных значений.

При выполнении операции *DELETE* производится коррекция всех индексов, определенных на данном отношении. Операция *UPDATE* также может повлечь коррекцию индексов, если затрагивает поля, входящие в состав их ключей.

Кроме описанных "атомарных" операций сканирования и модификации таблиц и списков, интерфейс *RSS* включает одну "макрооперацию", позволяющую за одно обращение к *RSS* построить отсортированный по значением заданных полей список. Эта операция - *BUILDLIST* - включает сканирование заданного отношения или списка, создание нового списка, в который включаются указанные поля выбираемых кортежей, и сортировку построенного списка в соответствии со значениями указанных полей. Идентификатор заново построенного отсортированного списка является ответным параметром операции.

Соответственно, параметрами операции *BUILDLIST* являются набор параметров для открытия сканирования (допускается любой способ сканирования), список номеров полей, составляющих кортежи нового списка, и список номеров полей, по которым нужно производить сортировку (как и в случае создания нового индекса, можно отдельно для каждого из этих полей указать требование к сортировке по возрастанию или убыванию значений данного поля). Отдельным параметром операции *BUILDLIST* является признак, в соответствии со значением которого допускаются или не допускаются кортежи-дубликаты в новом списке. Забегая вперед, заметим, что допущение или недопущение дубликатов в отсортированном списке зависит от того, для каких целей он строится. Например, если список строится для выполнения операции соединения, то дубликатов в нем быть не должно. Если же список строится для вычисления агрегатных функций (*COUNT*, *AVG* и т.д.), то дубликаты из него убирать нельзя. Более подробно мы рассмотрим этот и близкие вопросы в связи с проблемами оптимизации запросов в System R.

Операция *RSS* добавления поля к существующему отношению позволяет в динамике изменять схему таблицы. Параметрами операции *CHANGE* являются идентификатор существующей таблицы и спецификация нового поля (его тип). При выполнении операции изменяется только описатель данного отношения в служебном отношении описателей отношений. Как мы уже отмечали в предыдущем подразделе, до выполнения первой операции *UPDATE*, затрагивающей новое поле таблицы, реально ни в одном кортеже таблицы память под новое поле выделяться не будет. По умолчанию значения нового поля во всех кортежах таблицы, в которые еще не производилось явное занесение значения, считаются неопределенными. Тем самым, ни для одного поля, динамически добавленного к существующей таблице, не может быть запрещено хранение неопределенных значений.

Каждая операция *RSS* выполняется в пределах некоторой транзакции. Интерфейс *RSS* включает набор операций управления прохождением транзакции: начать транзакцию (*BEGIN TRANSACTION*), закончить транзакцию (*END TRANSACTION*), установить точку сохранения (*SAVE*) и выполнить откат до указанной точки сохранения или до начала транзакции (*RESTORE*).

Мы не отмечали этого раньше, но на самом деле при обращении к любой функции *RSS*, кроме *BEGIN TRANSACTION*, должен указываться еще один параметр - идентификатор транзакции. Этот идентификатор и вырабатывается при выполнении операции *BEGIN TRANSACTION*, которая сама входных параметров не требует.

В любой точке транзакции до выполнения операции *END TRANSACTION* может быть выполнен откат данной транзакции, т.е. обратное выполнение всех изменений, произведенных в данной транзакции, и восстановление состояния сканов. Откат может быть произведен до начала транзакции (в этом случае о восстановлении состояния сканов говорить бессмысленно) или до установленной ранее в транзакции точке сохранения.

Точка сохранения устанавливается с помощью операции *SAVE*. При выполнении этой операции запоминается состояние сканов данной транзакции, открытых к моменту выполнения *SAVE*, и координаты последней записи об изменениях в базе данных в журнале, произведенной от имени данной транзакции. Ответным параметром операции *SAVE* (а прямых параметров, кроме идентификатора транзакции, она не требует) является идентификатор точки сохранения. Этот идентификатор в дальнейшем может быть использован как прямой параметр операции *RESTORE*, при выполнении которой производится восстановление базы данных по журналу, с использованием записей о ее изменениях от данной транзакции до того состояния, в котором находилась база данных к моменту установления указанной точки сохранения. Кроме того, по локальной информации в оперативной памяти, привязанной к транзакции, восстанавливается состояние ее сканов. Откат к началу транзакции инициируется также обращением к операции *RESTORE*, но с указанием некоторого предопределенного идентификатора точки сохранения.

При выполнении своих транзакций пользователи System R изолированы один от другого, т.е. не ощущают того, что система функционирует в многопользовательском режиме. Это достигается за счет наличия в *RSS* механизма неявной синхронизации (более полно это мы обсудим в следующем подразделе). Пока заметим лишь, что до конца транзакции никакие изменения базы данных, произведенные в пределах этой транзакции, не могут быть использованы в других транзакциях (попытка использования таких данных приводит к временным синхронизационным блокировкам этих транзакций). При выполнении операции *END TRANSACTION* происходит "фиксация" изменений, произведенных в данной транзакции, т.е. они становятся видимыми в других транзакциях. Реально это означает снятие синхронизационных захватов с объектов базы данных, изменявшихся в транзакции. Из этого следует, что после выполнения *END TRANSACTION* невозможны индивидуальные откаты данной транзакции. RSS просто делает недействительным идентификатор данной транзакции, и после выполнения операции окончания транзакции отвергает все операции с таким идентификатором.

Последняя операция интерфейса *RSS* - операция явной синхронизации *LOCK*. Эта операция позволяет установить явный синхронизационный захват на указанное отношение (параметром операции является идентификатор таблицы). Выполнение операции *LOCK* гарантирует, что никакая другая транзакция до конца данной не сможет изменить данное отношение (вставить в него новый кортеж, удалить или модифицировать существующий), если установлен захват отношения в режиме чтения, или даже прочитать любой кортеж этого отношения, если установлен захват в режиме изменения.

Из всего, что говорилось раньше по поводу подхода к синхронизации в System R и соответствующего разбиения системы на уровни, следует нелогичность наличия этой операции в интерфейсе RSS. На самом деле, логически эта операция избыточна, т.е. если бы ее не было, можно вполне реализовать SQL на оставшейся части операций. До изложения материала следующего подраздела об этом трудно говорить, но предварительно заметим, что операция *LOCK* введена в интерфейс *RSS* для возможности оптимизации выполнения запросов. Дело в том, что, как видно из описания интерфейса, он покортежный. Следовательно, и информация для синхронизации носит достаточно узкий характер. В то же время на уровне SQL имеется более полная информация. Например, если обрабатывается предложение *SQL DELETE FROM EMP*, то известно, что будут удалены все кортежи указанной таблицы. Понятно, что как бы не реализовывался

механизм синхронизации в *RSS*, в данном случае выгоднее сообщить сразу, что изменения касаются всего отношения. Но снова забегая вперед, заметим, что ситуации в компиляторе SQL, когда очевидна выгода от использования явной синхронизации, достаточно редки. Пользоваться этим средством можно только очень осмотрительно, потому что неоправданные захваты таких крупных объектов могут резко ограничить степень асинхронности выполнения транзакций.

7.5. Синхронизация в System R.

System R с самого начала задумывалась как многопользовательская система, обеспечивающая режим мультидоступа к базам данных. Поэтому вопросам синхронизации доступа всегда уделялось очень большое внимание. Разработчики System R сформулировали и частично решили много проблем синхронизации, соответствующие публикации давно стали классикой, и на них ссылаются практически во всех работах, связанных с синхронизацией в системах управления базами данных. Мы постараемся привести историческую ретроспективу решений в области синхронизации в System R. Предварительно заметим, что вопросы синхронизации находятся в тесной связи с вопросами журнализации изменений и восстановления состояния базы данных.

Начнем с рассмотрения целей, которыми руководствовались разработчики System R при выработке своего подхода к синхронизации. Дело в том, что начальной целью синхронизации операций было не обеспечение изолированности пользователей, а поддержка средств обеспечения логической целостности баз данных. Как мы отмечали во введении, логическая целостность баз данных System R поддерживается на основе наличия ранее сформулированных и запомненных в каталогах базы данных ограничений целостности. В конце каждой транзакции или при выполнении явного предложения SQL проверяется ненарушение ограничений целостности изменениями, произведенными в данной транзакции. Если обнаруживается нарушение ограничений целостности, то с помощью операции *RSS RESTORE* производится откат транзакции, нарушившей ограничения.

Для того, чтобы можно было корректно выполнить такой откат, необходимо, чтобы до конца транзакции объекты базы данных, изменявшиеся транзакцией, не могли изменяться другими транзакциями. В противном случае возникает так называемая проблема потерянных изменений. Действительно, пусть транзакция 1 изменяет некоторый объект базы данных *A*. Затем другая транзакция 2 также изменяет объект *A*, после чего производится откат транзакции 1 (по причине, например, нарушения ей ограничений целостности). Тогда при следующем чтении объекта *A* транзакция 2 увидит его состояние, отличное от того, в которое он перешел в результате его изменения транзакцией 2.

Исходным постулатом System R было то, что потерянные изменения допускать нельзя, и обеспечение этого являлось минимальным требованием к системе синхронизации. Соответствующий режим выполнения транзакций называется в System R первым уровнем совместимости транзакций. Это наиболее низкий уровень синхронизации, вызывающий минимальные накладные расходы в системе. Технически синхронизация на первом уровне совместимости предполагает долговременные (до конца транзакции) синхронизационные захваты изменяемых объектов базы данных и отсутствие каких-либо захватов для читаемых объектов.

С точки зрения обеспечения логической целостности баз данных первый уровень совместимости вполне удовлетворителен, но вызывает некоторые проблемы внутри транзакций. Основная проблема - возможность чтения грязных данных. Действительно, читающая транзакция абсолютно не синхронизуется с изменяющими транзакциями, и поэтому в ней может быть прочитано некоторое промежуточное с логической точки зрения состояние объекта (мы подчеркиваем, что "грязным" объект может быть только на логическом уровне; физическую согласованность в базе данных поддерживает другой, "физический" уровень синхронизации *RSS*, на котором мы остановимся позже). Следующий, второй уровень совместимости транзакций System R обеспечивает отсутствие грязных данных. Технически синхронизация на втором уровне совместимости предполагает долговременные синхронизационные захваты (до конца транзакции) изменяемых объектов и кратковременные (на время выполнения операции) захваты читаемых объектов.

Второй уровень совместимости транзакций System R гарантирует отсутствие грязных данных, но не свободен от еще одной проблемы - неповторяющихся чтений. Если транзакция два раза (подряд или с некоторым временным промежутком) читает один и тот же объект базы данных, то состояние этого объекта может быть разным при разных чтениях, поскольку в промежутке между ними (а он может возникнуть, даже если чтения идут в транзакции подряд) другая транзакция может модифицировать объект. Эту проблему решает третий уровень совместимости транзакций System R, являющийся тем самым уровнем максимальной изолированности пользователей, о котором мы говорили раньше. Технически синхронизация на третьем уровне совместимости предполагает долговременные (до конца транзакции) синхронизационные захваты всех объектов, читаемых и изменяемых в данной транзакции.

В начальных версиях System R обеспечивались все перечисленные уровни совместимости транзакций. Соответственно, существовали параметры операции *RSS BEGIN TRANSACTION*, определявшие уровень совместимости данной транзакции. Однако, уже первый опыт использования System R показал, что наиболее применительным является третий уровень совместимости. При этом существовали приложения, которые устраивал первый уровень (в основном, приложения, связанные со статистической обработкой, в которых ошибки "грязных" данных исправлялись за счет большого числа чтений). Второй уровень совместимости оказался практически неприменимым. В результате в последних версиях разработчики оставили только третий уровень совместимости, и далее мы будем иметь в виду только его.

Поскольку интерфейс *RSS* - покортежный, то логично производить синхронизацию в *RSS* именно на уровне кортежей или, вернее, их уникальных идентификаторов - *tid* 'ов. Заметим, однако, что если две или более транзакций читают один кортеж, то с точки зрения синхронизации это вполне допустимо, а если хотя бы одна транзакция изменяет кортеж, то она должна блокировать все остальные транзакции, выполняющие операции чтения или изменения данного кортежа. Из этого следует потребность в двух разных режимах захватов кортежей - совместном режиме (для чтения) и монопольном (для изменений). Следуя терминологии System R, далее мы будем называть эти режимы режимом S (совместным) и режимом X (монопольным). Естественно формулируются правила совместимости захватов одного объекта в режимах S и X: захват режима S совместим с захватом режима S и несовместим с захватом режима X; захват режима X несовместим с захватом любого режима.

Соответственно, при чтении кортежа *RSS* прежде всего устанавливает захват этого кортежа в режиме S; при вставке, удалении или модификации кортежа - в режиме X. Если к этому моменту кортеж захвачен от имени другой транзакции в режиме, несовместимом с требуемым, транзакция, обратившаяся к *RSS*, блокируется, и требование захвата кортежа ставится в очередь до тех пор, пока не возникнут условия удовлетворения требуемого захвата, т.е. не будут сняты конфликтующие захваты с кортежа.

В принципе режим покортежных захватов достаточен для удовлетворения требований, ранее сформулированных в этом подразделе. Следует отметить, что большинство реляционных систем баз данных и ограничиваются покортежными (или даже более грубыми постраничными) захватами. Тем не менее в контексте System R синхронизация такого типа недостаточна.

Основной проблемой является возможность появления кортежей-фантомов при повторных сканированиях отношений в одной транзакции (даже на третьем уровне совместимости). Действительно, после первого сканирования все прочитанные кортежи будут захвачены в режиме S, из-за чего никакая другая транзакция не сможет удалить или модифицировать такие кортежи. Но ничто не мешает вставить в другой транзакции в то же отношение новый кортеж, значения полей которого удовлетворяют условиям сканирования первой транзакции. Тогда при повторном сканировании того же отношения с теми же условиями сканирования будет прочитан кортеж, которого не было при первом сканировании, т.е. появится кортеж-фантом.

Был предложен подход к решению проблемы фантомов на основе так называемых предикатных захватов. Идея подхода состоит в том, что при сканировании следует на самом деле захватывать не индивидуальные прочитанные кортежи, а все виртуальное множество кортежей, которые могли бы быть прочитаны при данном сканировании. Для этого достаточно "захватить"

условие (предикат), которому должны удовлетворять все кортежи при данном сканировании. Например, если должны быть прочитаны кортежи отношения со значением поля $a > 5$, то захватывается предикат $a > 5$. Если некоторая другая транзакция выполняет операцию, например, вставки в то же отношение кортежа со значением поля $a > 5$, то предикатный захват, предшествующий реальному выполнению этой операции, должен конфликтовать с захватом первой транзакции и т.д.

Естественно, что чем более точно формулируется предикат, тем больше реальная параллельность выполнения транзакций в системе с предикатными захватами. С этой точки зрения наибольшего уровня асинхронности можно было бы достичь, если допустить использование в качестве синхронизационных предикатов условий выборки, удаления или модификации языка SQL (или другого языка запросов высокого уровня). Но с другой стороны, чем более общий вид предикатов допускается, тем сложнее становится проблема обнаружения совместимости соответствующих захватов. Достаточно легко реализуется система предикатных захватов, в которой предикаты представляют собой логические выражения, состоящие из простых предикатов сравнения полей отношения с константными значениями. Разработчики System R не пошли и на такую систему, но некоторые идеи предикатных захватов использовали.

При наличии только покортежных захватов неясным остается вопрос синхронизации в *RSS* покортежных операций и операций создания и уничтожения отношений и индексов, и добавления полей к существующему отношению. Непонятно, как сочетать покортежные захваты с возможность явного захвата отношения и т.д. Очевидно, что общая система предикатных захватов такие проблемы снимает, но как мы уже отмечали, разработчики System R не пошли на ее реализацию. Вместо этого был разработан протокол иерархических гранулированных захватов (с некоторыми элементами предикатных захватов).

Основная идея состоит в том, что имеется некоторая иерархия памяти хранения кортежей: сегмент-отношение-кортеж. Объекты каждого уровня иерархии могут быть захвачены. Кроме того, можно захватывать указанный диапазон значений любого индекса. Набор возможных режимов захватов расширяется так называемыми целевыми (*intented*) захватами. Семантически целевой захват "сложного" объекта (сегмента или отношения) означает намерение данной транзакции устанавливать целевые или обычные захваты на более низком уровне иерархии. Введены следующие типы целевых захватов: IX ($intented\ to\ X$), IS ($intented\ to\ S$) и SIX ($shared, intented\ to\ X$).

Захват объекта в режиме IX соответствует намерению транзакции производить захваты объектов ниже по иерархии в режимах IX или X. Захват объекта в режиме IS соответствует намерению транзакции производить захваты объектов ниже по иерархии в режимах IS или S. Наконец, захват объекта в режиме SIX соответствует захвату объекта в режиме S и намерению транзакции захватывать объекты ниже по иерархии в режиме X. Из этого следуют следующие правила совместимости захватов разных режимов:

	X	S	IX	IS	SIX
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 синхронизации с использованием перечисленных режимов захватов следующий: чтобы захватить объект (например, кортеж отношения) в режиме S (X), нужно предварительно установить захваты в режиме IS (IX) соответствующих объектов выше по иерархии (в случае захвата кортежа - сегмента и отношения); при этом захваты должны устанавливаться, начиная от корня иерархии (в нашем случае - сначала для сегмента, затем для отношения и только потом для кортежа).

Очевидно, что протокол иерархических захватов решает проблему совместимости глобальных захватов сложного объекта (например, захватов отношения в режиме S) с захватами подобъектов этого объекта (кортежей). Но также очевидно, что протокол, вообще говоря, не решает проблему фантомов. Если отношение сканируется без использования индекса, то отсутствие фантомов можно гарантировать, если предварительно захватить все отношение в режиме S. Тогда, в соответствии с иерархическим протоколом, никакая другая транзакция не сможет занести в это отношение новый кортеж, потому что она будет блокирована при попытке захватить отношение в режиме IX (захваты отношения в режимах S и IX несовместимы). Можно, конечно, захватывать все отношение и при сканировании с использованием индекса. Таким образом можно решить проблему фантомов, но это очень неэффективное решение, потому что оно резко ограничивает возможности параллельного выполнения транзакций. Любая только читающая отношение транзакция конфликтует с любой транзакций, изменяющей это отношение. С другой стороны, в RSS при сканировании отношения по индексу имеется дополнительная информация (диапазон сканирования), которая ограничивает множество кортежей, среди которых не должны возникать фантомы.

Исходя из этих соображений, было предложено ввести в систему синхронизации элементы предикатных захватов. Заметим сначала, что технически захваты сегментов, отношений и кортежей трактуются единообразно, как захваты tid'ов. При захвате кортежа на самом деле захватывается его tid. При захвате сегмента или отношения на самом деле захватывается tid описателя соответствующего объекта во внутренних отношениях описателей таких объектов (сегментов и отношений). Предлагалось расширить систему синхронизации, разрешив применять захваты к паре идентификатор индекса - интервал значений ключа этого индекса. К такой паре разрешено применять захваты в любом из допустимых режимов, причем два захвата совместимы в том и только в том случае, если они совместимы в соответствии с приведенной выше таблицей, или указанные диапазоны значений ключей не пересекаются.

При наличии такой возможности, если открывается сканирование отношения по индексу, отношение захватывается в режиме IS, и в этом же режиме захватывается пара идентификатор индекса - диапазон сканирования. При занесении (удалении) кортежа отношение захватывается в режиме IX, и в этом же режиме для каждого индекса, определенного на данном отношении, захватывается пара идентификатор индекса - значение ключа из затрагиваемого операцией кортежа. Тогда читающие транзакции реально конфликтуют только с теми изменяющими транзакциями, которые затрагивают диапазон сканирования. При этом решается проблема фантомов, и асинхронность транзакций ограничивается "по существу", т.е. только тогда, когда их параллельное выполнение создает проблемы.

Заметим сразу, что описанное решение проблем синхронизации далеко от идеального. Во-первых, по-прежнему при сканировании отношения без использования индексов отсутствие фантомов можно гарантировать только при полном захвате всего отношения в режиме S. Во-вторых, даже при сканировании по индексу условие реальной выборки кортежа часто может быть строже простого указания диапазона сканирования, а это значит, что требуемый захват слишком сильный, т.е. охватывает более широкое множество кортежей, чем то, которое будет реальным результатом сканирования.

Видимо, по этим причинам, а также по причинам требуемого усложнения системы синхронизации, описанные средства борьбы с фантомами не были реализованы в System R (по крайней мере, это следует из заключительных публикаций). Более того, в силу половинчатости этого решения и слишком большого ограничения степени асинхронности разработчики отказались и от неявных захватов отношения в режиме S при сканировании без использования индексов. (Напомним, что возможность явного захвата целиком отношения осталась). Тем самым, System R не гарантирует отсутствие фантомов при повторном сканировании отношения.

Опыт System R в области синхронизации оказал очень большое влияние на разработчиков реляционных СУБД во всем мире. Особенно это касается предложений по части предикатных захватов. В ряде существующих или проектируемых СУБД предикатные захваты составляют

основу системы синхронизации, которая, конечно, при этом становится существенно более сложной, чем в System R.

В заключение данного подраздела кратко упомянем о еще одном уровне синхронизации, присутствующем в *RSS*, - уровне физической синхронизации. Мы уже отмечали, что после выполнения любой операции *RSS* оставляет базу данных в физически согласованном виде. Это означает, в частности, корректность всех межстраничных ссылок. Примерами таких ссылок могут быть ссылки между станицами *B* -деревьев индексов и т.д. Во время выполнения операций изменения (занесения, модификации или удаления кортежа) может возникать временная некорректность состояния страниц данных. Для того, чтобы каждая операция при начале своего выполнения имела корректную информацию, необходима дополнительная кратковременная синхронизация на уровне страниц. На время выполнения операции все необходимые страницы захватываются в режиме чтения или изменения. Захваты снимаются при окончании выполнения операции.

И последнее замечание. При синхронизации транзакций могут возникнуть тупиковые ситуации, когда две или более транзакции не могут продолжать свое выполнение по причине взаимных блокировок. *RSS* не предпринимает каких-либо действий по предотвращению тупиков. Вместо этого периодически проверяется состояние системы захватов на предмет обнаружения тупика, и если тупик обнаруживается, выбираются одна или несколько транзакций - жертв, для которых инициируется откат к началу или к ближайшей точке сохранения транзакции, гарантирующий разрушение тупика (при откате транзакции ее синхрозахваты снимаются). Выбор жертвы производится в соответствии с критериями минимальной стоимости проделанной транзакцией работы, которую придется повторить после отката. Мы не будем более подробно рассматривать схемы обнаружения и разрушения тупиков. Заметим лишь, что при обнаружении тупика применяется широко распространенная техника редукции графа ожидания с целью обнаружения в нем циклов, наличие которых и свидетельствует о наличии тупика.

7.6. Журнализация и восстановление в System R.

Одно из основных требований к любой системе управления базами данных состоит в том, что СУБД должна надежно хранить базы данных. Это означает, что СУБД должна поддерживать средства восстановления состояния баз данных после любых возможных сбоев. К таким сбоям относятся индивидуальные сбои транзакций (например, деление на ноль в прикладной программе, инициировавшей выполнение транзакции); сбой процессора при работе СУБД (так называемые мягкие сбои) и сбои (поломки) внешних носителей, на которых расположены базы данных (жесткие сбои).

Ситуации, возникающие при сбоях каждого из отмеченных классов, различны и, вообще говоря, требуют разных подходов к организации восстановления баз данных. Индивидуальные сбои транзакций означают, что все изменения, произведенные в базе данных некоторой транзакцией, незаконны, и их необходимо устранить. Для этого необходимо выполнить индивидуальный откат транзакции такого же типа, как при выполнении *RSS* явной операции *RESTORE*.

При возникновении мягкого сбоя системы утрачивается содержимое оперативной памяти. Восстановление состояния базы данных состоит в том, что после его завершения база данных должна содержать все изменения, произведенные транзакциями, закончившимися к моменту сбоя, и не должна содержать ни одного изменения, произведенного транзакциями, которые к моменту сбоя не закончились. Существенным аспектом ситуации является то, что состояние базы данных на внешней памяти не разрушено, что позволяет сделать процесс восстановления не слишком длительным.

Жесткие сбои приводят к полной или частичной потере содержимого баз данных на внешней памяти. Тем не менее цель процесса восстановления та же, что и в случае мягкого сбоя: после завершения этого процесса база данных должна содержать все изменения, произведенные транзакциями, закончившимися к моменту сбоя, и не должна содержать ни одного изменения, произведенного транзакциями, не закончившимися к моменту сбоя. В случае жесткого сбоя единственно возможный подход к восстановлению состояния базы данных может быть основан на

использовании ранее произведенной копии базы данных. В общем случае процесс восстановления после жесткого сбоя существенно более накладен, чем после мягкого сбоя.

Алгоритмы восстановления System R основаны на двух базовых средствах - ведении журнала и поддержке теневых состояний сегментов. Рассмотрим сначала механизм журнализации. Мы уже упоминали о наличии журнала в предыдущих подразделах. Журнал - это отдельный файл внешней памяти, для которого для надежности обычно поддерживаются две копии, и в который помещается информация обо всех операциях изменения состояния базы данных. В предыдущем подразделе мы упоминали об использовании журнала для отката транзакции по явной операции *RESTORE* или при неявных откатах при разрушении тупиков. Та же схема употребляется и при откатах индивидуальных транзакций при сбоях.

Механизм индивидуального отката основан на обратном выполнении всех изменений, произведенных данной транзакцией (undo). При этом из журнала в обратном хронологическому порядке выбираются все записи об изменении базы данных, произведенные от имени данной транзакции. Для этого необходима идентификация всех записей в журнале. В System R все записи одной транзакции связываются в один список в порядке, обратном хронологическому. Ссылка в списке представляет собой адрес записи в файле-журнале. Поскольку схема индивидуального отката едина для всех ситуаций индивидуальных сбоев, в частности для ситуации разрушения тупиков, то обратное выполнение операций сопровождается снятием установленных при прямой работе транзакции синхронизационных захватов с объектов базы данных. Следовательно, после выполнения индивидуального отката транзакции ситуация в системе такова, как если бы транзакция никогда и не начиналась.

Специфика мягкого сбоя системы состоит в утрате состояния оперативной памяти. В оперативной памяти находятся буфера базы данных. Поддерживаются буфера двух сортов: буфера журнала и буфера собственно базы данных. Буфера журнала содержат последние записи в журнал. Имеются два буфера журнала. Как только один буфер полностью заполняется, производится его запись в файл-журнал и продолжается заполнение второго буфера. Таким образом, при обычной работе системы обмены с файлом журнала не приводят к приостановке работы. Буфера базы данных содержат копии страниц базы данных, которые использовались в последнее время. В силу обычных в программировании принципов локализации ссылок достаточно вероятно, что после помещения копии страницы базы данных в буфер эта страница потребуется в ближайшем будущем. Поэтому наличие копии страницы в буфере позволит избежать обмена с устройством внешней памяти, когда эта страница понадобится в следующий раз.

Заметим, что размер буферного пула СУБД во многом определяет ее производительность. Обычная реляционная СУБД, такая, как System R, при наличии достаточного размера буферного пула вполне конкурентоспособна по отношению к системам, основанным на специализированной аппаратуре машин баз данных.

Задача System R по обеспечению надежного завершения транзакций, т.е. гарантированному наличию произведенных ими изменений в базе данных, требует наличия во внешней памяти информации об этих изменениях. Для этого при окончании любой транзакции поддерживается гарантированное присутствие в файле-журнале всех записей об изменениях, произведенных этой транзакцией. При использовании буферизации для записи в журнал для этого достаточно насильственно вытолкнуть во внешнюю память недозаполненный буфер журнала. Под насильственным выталкиванием понимается запись буфера во внешнюю память в соответствии не с логикой ведения журнала, а с логикой окончания транзакции. Только после произведения такого насильственного выталкивания буфера журнала транзакция считается закончившейся. Заметим, что последней записью в журнале от любой изменяющей базу данных транзакции является запись о конце транзакции. Эти записи используются при восстановлении. Рассмотрим теперь (пока не совсем точно) как осуществляется в System R восстановление базы данных после мягкого сбоя.

Основой алгоритма восстановления является то, что система придерживается правила упреждающей записи в журнал (*WAL* - Write Ahead Log). Это правило означает, что при выталкивании любой страницы из буфера страниц сначала гарантируется наличие в файле журнала за-

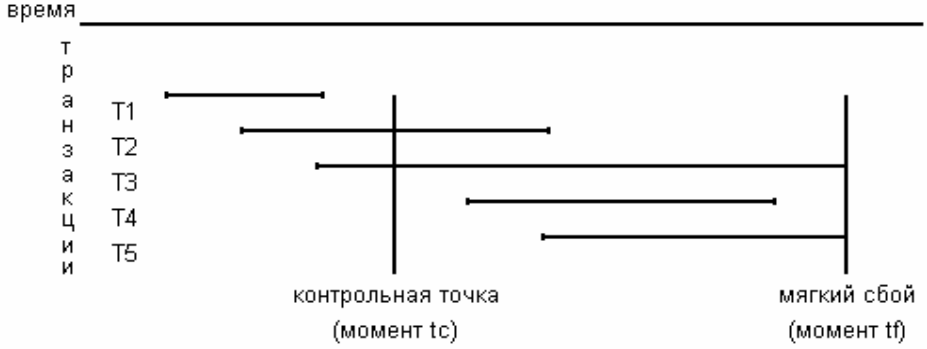
писи, относящейся к изменениям этой страницы после момента ее выталкивания в буфер. Поскольку записи в журнал блокируются, то для соблюдения правила *WAL* перед выталкиванием страницы данных необходимо вытолкнуть недозаполненный буфер журнала, если он содержит запись, относящуюся к изменению страницы. Применение правила *WAL* гарантирует, что если во внешней памяти находится страница базы данных, то в файле журнала находятся все записи об операциях, вызвавших изменение этой страницы. Обратное неверно: в файле журнала могут содержаться записи об изменении некоторых страниц базы данных, а сами эти изменения могут быть не отражены в состояниях страниц во внешней памяти.

При окончании любой транзакции (т.е. выполнении операции *RSS END TRANSACTION*) производится выталкивание недозаполненного буфера журнала и тем самым гарантируется наличие в журнале полной информации обо всех изменениях, произведенных данной транзакцией. Насильственное выталкивание страниц буфера базы данных не производится (слишком накладно было бы производить такие выталкивания при окончании любой транзакции). Тем самым после мягкого сбоя состояние базы данных во внешней памяти может не соответствовать тому, которое должно было бы быть после окончания транзакций. Следовательно, после мягкого сбоя некоторые страницы во внешней памяти могут не содержать информации, помещенной в них уже закончившимися транзакциями, а другие страницы могут содержать информацию, помещенную транзакциями, которые к моменту сбоя не закончились. При восстановлении необходимо добавить информацию в страницах первого типа и удалить информацию в страницах второго типа.

System R периодически устанавливает системные контрольные точки. Более подробно мы остановимся на этом ниже. Пока заметим лишь, что при установлении такой контрольной точки производится насильственное выталкивание во внешнюю память буфера журнала и всех буферов страниц. Это дорогостоящая операция, и выполняется она достаточно редко. При каждой системной контрольной точке в журнал помещается специальная запись.

Предположим, что последняя системная контрольная точка устанавливалась в момент времени tc, а мягкий сбой произошел в некоторый более поздний момент времени tf. Тогда все транзакции системы можно разбить на пять категорий.

Транзакции категории $T1$ начались и кончились до момента tc. Следовательно, все произведенные ими изменения базы данных надежно находятся во внешней памяти, и по отношению к ним никаких действий при восстановлении производить не нужно. Транзакции категории $T2$ начались до момента tc, но успели кончиться к моменту мягкого сбоя tf. Изменения, произведенные такими транзакциями после момента tc, могли не попасть во внешнюю память, и при восстановлении должны быть повторно произведены. Транзакции категории $T3$ начались до момента tc, но не кончились к моменту сбоя. Все их изменения, произведенные до момента tc, и, возможно, некоторые изменения, произведенные после момента tc, содержатся во внешней памяти. При восстановлении их необходимо удалить. Транзакции категории $T4$ начались после момента установки системной контрольной точки и успели закончиться до момента сбоя. Их изменения могли не отобразиться во внешнюю память; при восстановлении их необходимо выполнить повторно. Наконец, транзакции категории $T5$ начались после момента tc и не закончились к моменту сбоя. Их изменения должны быть удалены из страниц во внешней памяти.



В принципе можно было бы выполнить все необходимые восстановительные действия после мягкого сбоя, основываясь только на информации из журнала. Однако в System R ситуация несколько упрощается за счет применения техники теневых страниц. Принцип теневых страниц давно использовался в файловых системах, поддерживающих файлы со страничной организацией. В соответствии с этим принципом после открытия файла на изменение модифицированные страницы записываются на новое место внешней памяти (т.е. под них выделяются свободные блоки внешней памяти). При этом во внешней памяти сохраняется старая (теневая) таблица отображения страниц файла во внешнюю память, а в оперативной памяти по ходу изменения файла формируется новая таблица. При закрытии файла заново сформированная таблица записывается во внешнюю память, образуя новую теневую таблицу, а блоки внешней памяти, содержащие предыдущие образы страниц файла, освобождаются. При сбое процессора тем самым автоматически сохраняется состояние файла, в котором он находился перед последним открытием (конечно, с возможной потерей некоторых блоков внешней памяти, которые затем собираются с помощью специальной утилиты). Допускаются операции явной фиксации текущего состояния файла и явного отката состояния файла к точке последней фиксации.

В System R применяется развитие идей теневого механизма в контексте мультидоступных баз данных. Как мы уже отмечали, сегменты баз данных System R представляют собой файлы со страничной организацией. Соответственно, существуют и таблицы приписки этих файлов в блоки внешней памяти. При выполнении операции установки системной контрольной точки после выталкивания буферов страниц во внешнюю память таблицы отображения всех сегментов также фиксируются во внешней памяти, т.е. становятся теневыми. Далее до следующей контрольной точки доступ к страницам сегментов производится через таблицы отображения, располагаемые в оперативной памяти, и каждая изменяемая страница любого сегмента записывается на новое место внешней памяти с коррекцией соответствующей текущей таблицы отображения.

Тогда, если происходит мягкий сбой, все сегменты автоматически переходят в состояние, соответствующее последней системной контрольной точке, т.е. изменения, произведенные позже момента установления этой контрольной точки, в них просто не содержатся.

Это достаточно сильно упрощает процедуру восстановления после мягкого сбоя. Система вообще не должна предпринимать никаких действий по отношению к изменениям транзакций типа $T5$: этих изменений нет во внешней памяти. При восстановлении достаточно выполнить обратные изменения транзакций типа $T3$ (undo в терминологии System R), повторно выполнить изменения транзакций типа $T2$ (redo в терминологии System R; заметим, кстати, что эти изменения можно теперь выполнять безусловно, не заботясь о том, что они, возможно, и так содержатся во внешней памяти). Кроме того, нужно просто повторить изменения транзакций типа $T4$. Естественно, что начинать действия по журналу следует с записи о последней контрольной точке.

Справедливости ради отметим, что на самом деле теневой механизм используется в System R главным образом не для упрощения процедуры восстановления после мягкого сбоя. Как мы уже отмечали, без этого можно обойтись. Главная причина в другом, а именно, в том, что восстановление базы данных можно начинать только от ее физически согласованного состояния. Дело в том, что в журнал помещается информация об изменении объектов базы данных, а не страниц. Например, в журнале может находиться информация о модификации кортежа в виде триплета

```
<tid, старое состояние кортежа, новое состояние кортежа>
```

Реально же при выполнении операции модификации изменяются несколько страниц: исходная страница; возможно, страница замены, если кортеж не поместился в исходную страницу; страницы индексов. И так происходит при выполнении любой операции изменения базы данных. Поскольку буфера страниц выталкиваются во внешнюю память по отдельности, то к моменту мягкого сбоя во внешней памяти может возникнуть набор физически рассогласованных стра-

ниц, не соответствующий никакой журнализуемой операции. При таком состоянии внешней памяти восстановление по журналу невозможно.

Когда выполняется операция установки системной контрольной точки, то до насильственного выталкивания буферов страниц система дожидается завершения всех операций всех транзакции и до окончания выталкивания не допускает выполнения новых операций. Поэтому теневое состояние всех сегментов базы данных физически согласовано и может служить основой восстановления по журналу.

При жестких сбоях утрачивается содержимое всех или части сегментов базы данных. Для восстановления базы данных используются журнал и ранее произведенная копия базы данных. В System R допускается посегментное восстановление. Для этого копия сегмента переписывается с архивного носителя на заново выделенный рабочий носитель, а затем по журналу повторяются все изменения, производившиеся в объектах этого сегмента после момента копирования. Поскольку в момент жесткого сбоя содержимое оперативной памяти не утрачивается, то возможно продолжение выполнения транзакций после завершения восстановления. Более того, если авария коснулась только части сегментов базы данных, то транзакции на фоне процесса восстановления могут продолжать работу с объектами базы данных, расположенными в неповрежденных сегментах.

Единственным требованием к архивной копии сегмента является то, что она должна находиться в согласованном состоянии (поскольку восстановление ведется в терминах записей журнала). Поэтому для создания архивной копии сегмента достаточно лишь дождаться конца выполнения операций над объектами данного сегмента и запретить начало новых операций до конца копирования. Тем самым, выполнение архивной копии не требует перевода системы в какой-либо особый режим работы и только незначительно тормозит нормальную работу транзакций.

В заключение данного подраздела заметим, что в первых версиях System R в качестве архивного носителя использовались магнитные ленты. Однако со временем стало ясно, что во-первых, надежность магнитных лент существенно меньше надежности магнитных дисков, а во-вторых, они стали уступать и в емкости. Поэтому в последних версиях системы использовалась только дисковая память.

И последнее замечание. Журнал System R располагается в файле большого, но постоянного размера. Он используется в циклическом режиме. Когда записи журнала достигают конца файла, они начинают помещаться в его начало. Поскольку переход на начало файла можно считать утратой предыдущего журнала, этот переход сопровождается копированием сегментов базы данных. В некоторых других системах используется архивизация самого журнала.

§8. Ingres: общая организация системы, основы языка Quel.

8.1. История СУБД Ingres.

По своей значимости для развития и распространения реляционного подхода к управлению базами данных СУБД Ingres (Interactive Graphics and Retrieval System) находится близко к System R, хотя история и организация этой системы во многом отличается от System R. Для начала коротко рассмотрим историю Ingres.

Проект и экспериментальный вариант СУБД Ingres были разработаны в университете Беркли под руководством одного из наиболее известных в мире ученых и специалистов в области баз данных Майкла Стоунбрейкера (Michael Stonebraker). С самого начала СУБД Ingres разрабатывалась как мобильная система, функционирующая в среде ОС UNIX. Первая версия Ingres была рассчитана на 16-разрядные компьютеры и работала главным образом на машинах серии PDP. Это была первая СУБД, распространяемая бесплатно для использования в университетах. Впоследствии группа Стоунбрейкера перенесла Ingres в среду ОС UNIX BSD, которая также была разработана в университете Беркли. Семейство СУБД Ingres из университета Беркли принято называть "университетской Ingres".

В начале 80-х была образована компания RTI (Relational Technology Inc.) для доведения университетских прототипов до уровня коммерческих продуктов. С этого момента стали различать университетскую и коммерческую СУБД Ingres. В настоящее время коммерческая Ingres поддерживается, развивается и продается компанией Computer Associates. Сейчас это одна из развитых коммерческих реляционных СУБД.

Хотя во многих отношениях коммерческие варианты Ingres являются более развитыми, чем университетские, в учебных целях гораздо интереснее говорить про университетские разработки. Во-первых, как в случае любого коммерческого продукта, информация о внутренней организации коммерческой Ingres в основном носит закрытый характер. В то же время, по поводу университетской Ingres имеется много высококачественных публикаций. Во-вторых, университетскую Ingres можно опробовать на практике и даже посмотреть ее исходные тексты. Наконец, в-третьих, именно в университетской Ingres были опробованы многие оригинальные идеи, используемые в настоящее время во многих других системах. С использованием этой системы в университете Беркли (и других университетах) проводились многие учебные и исследовательские работы.

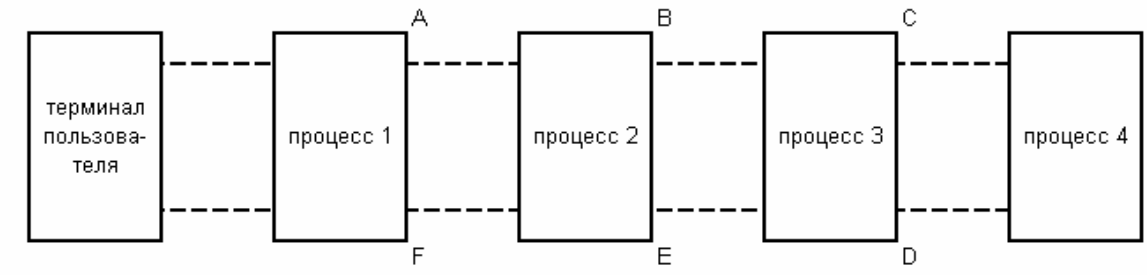
Поэтому в данной лекции мы будем рассматривать организацию университетской версии СУБД Ingres, которая тесно связана с особенностями языка QUEL (в такой же степени, в какой System R тесно связана с особенностями языка SQL). Далее, говоря о СУБД Ingres, мы будем в этой лекции иметь в виду университетскую Ingres.

8.2. Ingres как UNIX-ориентированная СУБД. Динамическая структура системы: набор процессов.

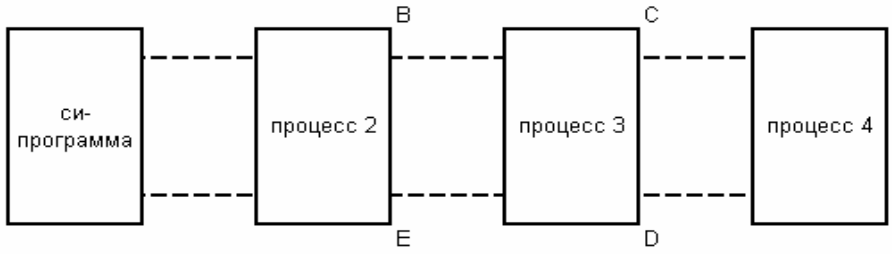
СУБД Ingres проектировалась в расчете на использование в среде ОС UNIX. Эта система играла роль своего рода виртуальной машины. Ориентация на использование UNIX наложила существенный отпечаток на общую организацию Ingres, на статическую и динамическую структуру СУБД.

Прежде всего, все базы данных, обслуживаемые СУБД Ingres на данном UNIX-компьютере, хранятся в одном поддереве файловой системы. Каждой базе данных соответствует отдельный справочник, каждое отношение базы данных (включая служебные отношения) хранится в отдельном файле ОС UNIX. Защита программных компонентов системы от несанкционированного выполнения и баз данных от несанкционированного доступа основывается главным образом на общем механизме защиты файлов ОС UNIX. При установке СУБД Ingres автоматически заводится специальный "пользователь" ОС UNIX с именем Ingres, от имени которого работают все системные процессы Ingres, и только ему разрешается запускать эти системные процессы и обращаться к файлам баз данных. Более точное управление доступом берет на себя Ingres.

Существуют две возможности вызова Ingres - в интерактивном режиме командой языка Shell или из прикладной программы, написанной на языке EQUEL и преобразованной прекомпилятором языка EQUEL к программе на языке Си. В первом случае создается следующая структура процессов:



Во втором случае структура процессов выглядит следующим образом:



Процесс 1 - это интерактивный терминальный монитор, позволяющий пользователю формулировать, редактировать и выполнять наборы команд Ingres (операторов языка QUEL).

В процессе 2 выполняется лексический и синтаксический анализ операторов QUEL, модификация операторов с целью обеспечения целостности баз данных, контроля доступа, подстановки представлений, а также синхронизация параллельного доступа к базе данных.

Процесс 3 является ответственным за выполнение операторов выборки, занесения и удаления кортежей. В нем выполняется оптимизация запросов на основе техники декомпозиции сложных запросов. Кроме того, для операторов модификации кортежей производится предварительная выборка модифицируемых кортежей и подготовка их новых образов для реального выполнения модификации в процессе 4.

Наконец, в процессе 4 выполняются так называемые команды-утилиты - создания и уничтожения отношений, индексов и т.д., а также упомянутая отложенная модификация кортежей.

Процессы связаны программными каналами (pipes) ОС UNIX. Прямая информация при обработке операторов передается по каналам A, B и C. Результаты, включая сообщения об ошибках, передаются по обратным каналам D, E и F. Процессы работают строго синхронно: после посылки прямого сообщения каждый процесс дожидается получения ответного сообщения, а после посылки ответного сообщения - ждет получения очередного прямого.

Как видно, динамическая структура системы примерно одинакова в случаях интерактивного использования системы и в случае обращения к системе из прикладной программы. В последнем случае по естественным причинам отсутствует лишь процесс 1, осуществляющий функции терминального монитора.

Следует отметить, что на описанную структуру оказал большое влияние тот факт, что первый вариант Ingres реализовывался для 16-разрядных компьютеров, в которых размер виртуальной памяти процесса был весьма ограничен. Поскольку процессы системы функционировали синхронно, принципиальной выгоды от наличия нескольких процессов не было. Но подход к разбиению системы на несколько процессов позволил выработать разумную статическую структуризацию системы, в ряде компонентов которой не используются общие данные. Кроме того, с развитием системы стали использоваться и реальные возможности распараллеливания.

8.3. Структуры данных, методы доступа, интерфейсы доступа к данным.

Организация данных в базе данных Ingres отличается от организации данных в System R прежде всего тем, что на логическом уровне поддерживаются только отношения. Для каждого отношения может быть создано несколько индексов, но для индексов не поддерживаются какие-либо специальные структуры данных; они представляются также в виде отношений (для которых, правда, уже нельзя создавать индексы).

Как мы уже отмечали, каждое отношение базы данных Ingres хранится в отдельном файле ОС UNIX. Поддерживается несколько способов организации таких файлов: неключевая, основанная на хэшировании и индексно-последовательная. При любой организации кортежи отношения хранятся в специальных "первичных" страницах файлов в том же стиле, что и в System R. Соответственно, каждый кортеж обладает уникальным и не изменяемым во все время существования кортежа идентификатором (*tid*), который "почти напрямую" адресует кортеж.

При неключевой организации отношения файл состоит только из первичных страниц. Для поиска кортежей, удовлетворяющих условию выборки, требуется последовательный просмотр всех первичных страниц файла. При организации на основе хэширования файл также состоит только из первичных страниц, но расположение кортежей в страницах определяется значением

функции хэширования в зависимости от установленного ключа (части кортежа). Наконец, при индексно-последовательной организации кортежи отношения заносятся в файл в порядке возрастания установленного ключа. Для прямого доступа по ключу в том же файле поддерживается специальная индексная таблица. Заметим, что в начальных вариантах Ingres упорядоченность кортежей не поддерживалась в динамике, т.е. могла нарушаться при вставке новых или модификации существующих кортежей. Структура отношения может быть изменена в динамике путем выполнения специального оператора языка QUEL.

Для каждого из трех видов организации отношений поддерживался набор функций доступа (методов доступа) с фиксированным интерфейсом. Это позволяло добавлять новые методы доступа без требования переделки частей системы, которые ими пользовались.

Каждый набор функций включал следующие функции:

- `openr (descriptor, mode, relation-name)`
  Эта функция открывает отношение как файл ОС UNIX в режиме, определяемом значением параметра mode (на чтение или на чтение и модификацию). Кроме того, в выходной параметр *descriptor* заносится информация, характеризующая указанное отношение на основе системных каталогов. После выполнения функции openr параметр descriptor является обязательным входным параметром для всех прочих функций.
- `get(descriptor, tid, limit_tid, tuple, next_flag)`
  Если функция вызывается в режиме прямой выборки кортежа (значение параметра *next _ flag = false*), то в выходной параметр *tuple* заносится кортеж с идентификатором *tid* . При вызове в режиме сканирования (*next _ flag = true*) функция выполняет при каждом вызове последовательную выборку кортежей начиная с кортежа с идентификатором *tid* и кончая кортежем с идентификатором *limit _ tid* . Начальные установки *tid* и *limit _ tid* производятся функцией *find* .
- `find(descriptor, key, tid, match_mode)`
  Функция устанавливает в выходной параметр *tid* идентификатор первого или последнего кортежа отношения, который соответствует значению заданного ключа в соответствие с режимом, задаваемым входным параметром *match _ mode* . Если отношение имеет неключевую структуру, или если заданное значение ключа не соответствует типу ключевого атрибута отношения, в tid записывается идентификатор физически первого (или последнего) кортежа отношения.
- `paramd(descriptor, access_characteristics_structure)`
  `parami(descriptor, access_characteristics_structure)`
  Эта пара функций позволяет узнать о ключевых атрибутах отношения, использование которых может оптимизировать доступ к этому отношению. Соответствующая информация записывается в выходной параметр *access _ characteristics _ structure* и используется системой для выбора значения параметра *match _ mode* при последующих вызовах функции *find* .
- `insert(descriptor, tuple)`
  Заданный кортеж заносится в указанное отношение в соответствии со структурой отношения и значением ключевых полей.
- `replace(descriptor, tid, new_tuple)`
  `delete(descriptor, tid)`
  Функции заменяют или удаляют кортеж отношения с указанным идентификатором.
- `closer(descriptor)`
  Функция закрывает соответствующий файл ОС UNIX и, возможно, обновляет содержимое отношений-каталогов.

Заметим, что перечисленные функции работают только с указанным отношением. В частности, если для отношения определены индексы, то их автоматическая модификация при изме-

нении отношений не производится. Кроме того, функции не выполняют никаких действий по журнализации изменений или синхронизации параллельного доступа.

8.4. Общая характеристика языка QUEL. Язык программирования EQUEL.

Манипуляционная часть языка QUEL является чистой реализацией реляционного исчисления кортежей. Это означает, что в операторах указываются условия, накладываемые на кортежи, с которыми необходимо произвести соответствующие действия.

Основной набор операторов манипулирования данными включает операторы *RETRIEVE* (выбрать), *APPEND* (добавить), *REPLACE* (заменить) и *DELETE* (удалить). Перед выполнением любого из этих операторов необходимо определить используемые в них переменные кортежей, связав их с соответствующими отношениями путем выполнения оператора *RANGE*:

```
RANGE OF variable-list IS relation-name
```

Продемонстрируем основные свойства операторов QUEL на примерах. Будем использовать базу данных *СТУДЕНТЫ* и *ГРУППЫ*:

```
RANGE OF S IS СТУДЕНТЫ
   RANGE OF G IS ГРУППЫ
```

Пример 1. Выбрать имена студентов, куратором которых является Иванов.

```
RETRIEVE (S.СТУД_ИМЯ)
   WHERE (S.ГРУП_НОМЕР = G.ГРУП_НОМЕР AND G.КУРАТ_ИМЯ = "ИВАНОВ")
```

Пример 2. Занести в отношение *НЕУСПЕВАЮЩИЕ* номера студенческих билетов и имена неуспевающих студентов.

```
RETRIEVE INTO НЕУСПЕВАЮЩИЕ (S.СТУД_НОМЕР, S.СТУД_ИМЯ)
   WHERE (S.СТУД_УСП = "NO")
```

Пример 3. Вывести фамилии студентов, получающих стипендию ниже средней.

```
RETRIEVE (S.СТУД_ИМЯ)
   WHERE (S.СТУД_СТИП < AVG (S.СТУД_СТИП))
```

Как и в SQL, поддерживаются агрегатные функции *COUNT*, *SUM*, *MAX*, *MIN* и *AVG*.

Пример 4. Включить в группу 310 студента Петрова.

```
APPEND TO СТУДЕНТЫ (СТУД_ИМЯ = "ПЕТРОВ", ....)
```

Пример 5. Увеличить стипендию в 1,5 раза всем успевающим студентам.

```
REPLACE S(СТУД_СТИП BY СТУД_СТИП * 1,5)
   WHERE (S.СТУД_УСП = "YES")
```

Пример 6. Удалить из списка групп все группы, в которых не учится ни один студент.

```
DELETE G
   WHERE (G.ГРУП_РАЗМЕР = 0)
```

Кроме операторов манипулирования данными, язык QUEL содержит операторы для создания и уничтожения отношений:

```
CREATE имя_отношения (имя_атрибута IS тип_атрибута, ...)
DESTROY имя_отношения
```

а также два оператора изменения структур хранимых данных:

```
MODIFY имя_отношения TO структура_памяти ON (ключ1, ключ2, ...)
INDEX ON имя_отношения IS имя_индекса (ключ1, ключ2, ...)
```

Оператор *MODIFY* изменяет структуру хранимого отношения в соответствии с параметром *структура _ памяти* и заданным набором ключевых атрибутов. Оператор *INDEX* создает отдельную индексную структуру для заданных полей данного отношения. Созданные индексы используются системой для оптимизации выполнения операторов манипулирования данными. Согласованность содержимого отношений и индексов поддерживается системой автоматически.

Язык QUEL содержит также операторы определения ограничений целостности, представлений и ограничений доступа. На них мы остановимся немного позже.

В том виде, в каком мы его кратко описали, язык QUEL предназначен для интерактивной работы с базами данных Ingres. Для программирования прикладных информационных систем, которые должны взаимодействовать с базами данных, был разработан язык программирования EQUEL, являющийся, по существу, расширением языка программирования Си путем встраивания в него операторов языка QUEL. Язык EQUEL определяется следующим образом:

- Любой оператор языка Си является оператором языка EQUEL.
- Любой оператор языка QUEL, которому предшествуют два знака '#', является допустимым оператором языка EQUEL.
- Переменные Си-программы могут использоваться в операторах QUEL, заменяя имена отношений, имена атрибутов, элементы списка выборки или константы. Те переменные Си-программы, которые используются таким образом, должны при своем объявлении быть помечены двойным знаком '#'.
- Оператор *RETRIEVE* (без *INTO*) сопровождается составным оператором языка Си, который выполняется по одному разу для каждого выбранного кортежа.

Пример программы на языке EQUEL, выдающей номер группы по имени студента:

```
main()
{
   ## char stud_name[20];
   ## int  group_number;
   while (READ(stud_name_)
   {
      ##  RANGE OF S IS STUDENTS
      ##  RETRIEVE (group_number = G.GROUP.NUMBER)
      ##  WHERE (S.STUD_NAME = stud_name)
      {
         PRINT ("The group number of 'stud_name' is 'group_number');
      }
   }
}
```

Программа на языке EQUEL обрабатывается специальным препроцессором, который превращает ее в обычную Си-программу, содержащую вызовы Ingres с передачей в качестве параметров текстов операторов языка QUEL. Дальнейшую схему мы уже обсуждали.

8.5. Общий подход к организации представлений, ограничениям целостности и контролю доступа.

Мы объединили эти три кажущиеся не очень близкими темы, потому что в Ingres для решения соответствующих проблем применяется единый подход, основанный на модификации операторов SQL. Начнем с представлений. Как и в System R (точнее, в языке SQL), представление базы данных - это некоторый именованный запрос с именованными полями результирующего отношения.

Например, оператор

```
DEFINE VIEW GROUP310
   (STUD_NUMBER = S.STUD_NUMBER,
   STUD_NAME = S.STUD_NAME,
```

```
   STUD_STATUS = S.STUD_STATUS)
WHERE (S.GROUP_NUMBER = 310)
```

определяет представляемое отношение, включающее номера студенческих билетов и имена студентов из группы 310.

Предположим, что мы хотим теперь найти неуспевающих студентов в отношении GROUP310:

```
RANGE OF G310 IS GROUP310
RETRIEVE (G310.STUD_NAME)
WHERE (G310.STUD_STATUS = "NO")
```

Тогда после модификации этот запрос будет выглядеть следующим образом:

```
RETRIEVE (STUD_NUMBER = S.STUD_NUMBER, STUD_NAME =
          S.STUD_NAME, STUD_STATUS = S.STUD_STATUS)
WHERE (S.GROUP_NUMBER = 310 AND
       S.STUD_STATUS = "NO")
```

На тех же самых принципах построен контроль доступа к данным и контроль целостности баз данных. Например, ограничение доступа к отношению *СТУДЕНТЫ* может быть определено следующим образом:

```
DEFINE PERMIT RETRIEVE, REPLACE
       ON     S
       TO     PETROV
       AT     TTA5
       FROM   9:00 TO 17:50
       ON     MON TO FRI
       WHERE  (S.GROUP_NUMBER = 310)
```

Это означает, что Петрову разрешается читать и модифицировать отношение *СТУДЕНТЫ* с терминала TTA5 во время от 9 до 15:00 в рабочие дни недели, причем только те кортежи, которые удовлетворяют сформулированному условию. При компиляции любого оператора QUEL над отношением *СТУДЕНТЫ* этот оператор будет модифицироваться таким образом, чтобы он был выполнен при выполнении условий хотя бы одного из ограничений доступа.

Аналогично, если для отношения *СТУДЕНТЫ* определено ограничение целостности

```
DEFINE INTEGRITY
    ON     S
    WHERE  (S.STUD_STIP < 150,000)
```

то к условию любого оператора изменения кортежей отношения *СТУДЕНТЫ* будет через *AND* добавляться условия всех ограничений целостности, определенных для этого отношения.

В заключение этой лекции заметим, что конечно, в Ingres поддерживается механизм параллельных транзакций с соответствующей синхронизаций доступа и журнализация изменений баз данных. Однако нам не известны какие-либо особенности применяемых механизмов. На особенностях оптимизации операторов QUEL мы остановимся в лекции, посвященной оптимизациям в языках баз данных.

Внутренняя организация реляционных СУБД

§9. Структуры внешней памяти, методы организации индексов.

Реляционные СУБД обладают рядом особенностей, влияющих на организацию внешней памяти. К наиболее важным особенностям можно отнести следующие:

- Наличие двух уровней системы: уровня непосредственного управления данными во внешней памяти (а также обычно управления буферами оперативной памяти, управления транзакциями и журнализацией изменений БД) и языкового уровня (например, уровня, реализующего язык SQL). При такой организации подсистема нижнего уровня должна поддерживать во внешней памяти набор базовых структур, конкретная интерпретация которых входит в число функций подсистемы верхнего уровня.
- Поддержание отношений-каталогов. Информация, связанная с именованием объектов базы данных и их конкретными свойствами (например, структура ключа индекса), поддерживается подсистемой языкового уровня. С точки зрения структур внешней памяти отношение-каталог ничем не отличается от обычного отношения базы данных.
- Регулярность структур данных. Поскольку основным объектом реляционной модели данных является плоская таблица, главный набор объектов внешней памяти может иметь очень простую регулярную структуру.
- При этом необходимо обеспечить возможность эффективного выполнения операторов языкового уровня как над одним отношением (простые селекция и проекция), так и над несколькими отношениями (наиболее распространено и трудоемко соединение нескольких отношений). Для этого во внешней памяти должны поддерживаться дополнительные "управляющие" структуры - индексы.
- Наконец, для выполнения требования надежного хранения баз данных необходимо поддерживать избыточность хранения данных, что обычно реализуется в виде журнала изменений базы данных.

Соответственно возникают следующие разновидности объектов во внешней памяти базы данных:

- строки отношений - основная часть базы данных, большей частью непосредственно видимая пользователям;
- управляющие структуры - индексы, создаваемые по инициативе пользователя (администратора) или верхнего уровня системы из соображений повышения эффективности выполнения запросов и обычно автоматически поддерживаемые нижним уровнем системы;
- журнальная информация, поддерживаемая для удовлетворения потребности в надежном хранении данных;
- служебная информация, поддерживаемая для удовлетворения внутренних потребностей нижнего уровня системы (например, информация о свободной памяти).

Мы рассматривали на примерах System R и Ingres два альтернативных подхода к организации реляционной СУБД с точки разделения функций между различными компонентами. Напомним, что в СУБД System R существовала интегрированная подсистема управления данными, транзакциями и журнализацией, в то время как в Ingres управление данными, было отделено от управления транзакциями и журнализацией.

У обоих этих подходов имеются свои преимущества и недостатки. Подход System R позволяет использовать более эффективные методы за счет совместного решения проблем физической и логической синхронизации, использовании общих протоколов при управлении буферами и журнализации и т.д. Но при этом в некотором смысле подсистема нижнего уровня становится монолитом; при самой удачной ее структуризации компоненты остаются связанными общими протоколами взаимодействия. Непродуманные локальные изменения одного компонента могут

привести к фатальным последствиям для всей системы. Подход Ingres позволяет упростить структуру системы и сделать ее более гибкой, но это возможно только за счет огрубления алгоритмов: применения более грубых методов управления транзакциями; жестких протоколов журнализации и т.д.

В конечном счете любая конкретная система основывается на конкретном комплексном решении. Мы рассматриваем здесь фрагменты таких решений (эскизы).

9.1. Хранение отношений.

Существуют два принципиальных подхода к физическому хранению отношений. Наиболее распространенным является покортежное хранение отношений (кортеж является единицей физического хранения). Естественно, это обеспечивает быстрый доступ к целому кортежу, но при этом во внешней памяти дублируются общие значения разных кортежей одного отношения и, вообще говоря, могут потребоваться лишние обмены с внешней памятью, если нужна часть кортежа.

Альтернативным (менее распространенным) подходом является хранение отношения по столбцам, т.е. единицей хранения является столбец отношения с исключенными дубликатами. Естественно, что при такой организации суммарно в среднем тратится меньше внешней памяти, поскольку дубликаты значений не хранятся; за один обмен с внешней памятью в общем случае считывается больше полезной информации. Дополнительным преимуществом является возможность использования значений столбца отношения для оптимизации выполнения операций соединения. Но при этом требуются существенные дополнительные действия для сборки целого кортежа (или его части).

Поскольку гораздо более распространено хранение по строкам, мы рассмотрим немного более подробно этот способ хранения отношений. Типовой, унаследованной от System R, структурой страницы данных является следующая:

К основным характеристикам этой организации можно отнести следующие:

- Каждый кортеж обладает уникальным идентификатором (*tid*), не изменяемым во все время существования кортежа. Структура *tid* следует из приведенного выше рисунка.
- Обычно каждый кортеж хранится целиком в одной странице. Из этого следует, что максимальная длина кортежа любого отношения ограничена размерами страницы. Возникает вопрос: как быть с "длинными" данными, которые в принципе не помещаются в одной странице? Применяются несколько методов. Наиболее простым решением является хранение таких данных в отдельных (вне базы данных) файлах с заменой "длинного" данного в кортеже на имя соответствующего файла. В некоторых системах (например, в предпоследней версии СУБД Informix) такие данные хранились в отдельном наборе страниц внешней памяти, связанном физическими ссылками. Оба эти решения сильно ограничивают возможность работы с длинными данными (как, например, удалить несколько байтов из середины 2-мегабайтной строки?). В настоящее время все чаще используется метод, предложенный несколько лет тому назад в проекте Exodus, когда "длинные" данные организуются в виде B-деревьев последовательностей байтов.

- Как правило, в одной странице данных хранятся кортежи только одного отношения. Существуют, однако, варианты с возможностью хранения в одной странице кортежей нескольких отношений. Это вызывает некоторые дополнительные расходы по части служебной информации (при каждом кортеже нужно хранить информацию о соответствующем отношении), но зато иногда позволяет резко сократить число обменов с внешней памятью при выполнении соединений.
- Изменение схемы хранимого отношения с добавлением нового столбца не вызывает потребности в физической реорганизации отношения. Достаточно лишь изменить информацию в описателе отношения и расширять кортежи только при занесении информации в новый столбец.
- Поскольку отношения могут содержать неопределенные значения, необходима соответствующая поддержка на уровне хранения. Обычно это достигается путем хранения соответствующей шкалы при каждом кортеже, который в принципе может содержать неопределенные значения.
- Проблема распределения памяти в страницах данных связана с проблемами синхронизации и журнализации и не всегда тривиальна. Например, если в ходе выполнения транзакции некоторая страница данных опустошается, то ее нельзя перевести в статус свободных страниц до конца транзакции, поскольку при откате транзакции удаленные при прямом выполнении транзакции и восстановленные при ее откате кортежи должны получить те же самые идентификаторы.
- Распространенным способом повышения эффективности СУБД является кластеризация отношения по значениям одного или нескольких столбцов. Полезным для оптимизации соединений является совместная кластеризация нескольких отношений.
- С целью использования возможностей распараллеливания обменов с внешней памятью иногда применяют схему декластеризованного хранения отношений: кортежи с общим значением столбца декластеризации размещают на разных дисковых устройствах, обмены с которыми можно выполнять в параллель.

Что же касается хранения отношения по столбцам, то основная идея состоит в совместном хранении всех значений одного (или нескольких) столбцов. Для каждого кортежа отношения хранится кортеж той же степени, состоящий из ссылок на места расположения соответствующих значений столбцов. В последней лекции мы будем рассматривать особенности организации распределенных реляционных СУБД. Одним из приемов является так называемое вертикальное разделение отношений, когда в разных узлах сети хранятся разные проекции данного отношения. Хранение отношения по столбцам в некотором смысле является предельным случаем вертикального разделения отношений.

9.2. Индексы.

Как бы не были организованы индексы в конкретной СУБД, их основное назначение состоит в обеспечении эффективного прямого доступа к кортежу отношения по ключу. Обычно индекс определяется для одного отношения, и ключом является значение атрибута (возможно, составного). Если ключом индекса является возможный ключ отношения, то индекс должен обладать свойством уникальности, т.е. не содержать дубликатов ключа. На практике ситуация выглядит обычно противоположно: при объявлении первичного ключа отношения автоматически заводится уникальный индекс, а единственным способом объявления возможного ключа, отличного от первичного, является явное создание уникального индекса. Это связано с тем, что для проверки сохранения свойства уникальности возможного ключа так или иначе требуется индексная поддержка.

Поскольку при выполнении многих операций языкового уровня требуется сортировка отношений в соответствии со значениями некоторых атрибутов, полезным свойством индекса является обеспечение последовательного просмотра кортежей отношения в диапазоне значений ключа в порядке возрастания или убывания значений ключа.

Наконец, одним из способов оптимизации выполнения эквисоединения отношений (наиболее распространенная из числа дорогостоящих операций) является организация так

называемых мультииндексов для нескольких отношений, обладающих общими атрибутами. Любой из этих атрибутов (или их набор) может выступать в качестве ключа мультииндекса. Значению ключа сопоставляется набор кортежей всех связанных мультииндексом отношений, значения выделенных атрибутов которых совпадают со значением ключа.

Общей идеей любой организации индекса, поддерживающего прямой доступ по ключу и последовательный просмотр в порядке возрастания или убывания значений ключа является хранение упорядоченного списка значений ключа с привязкой к каждому значению ключа списка идентификаторов кортежей. Одна организация индекса отличается от другой главным образом в способе поиска ключа с заданным значением.

9.2.1. B-деревья

Видимо, наиболее популярным подходом к организации индексов в базах данных является использование техники B-деревьев. С точки зрения внешнего логического представления B-дерево - это сбалансированное сильно ветвистое дерево во внешней памяти. Сбалансированность означает, что длина пути от корня дерева к любому его листу одна и та же. Ветвистость дерева - это свойство каждого узла дерева ссылаться но большое число узлов-потомков. С точки зрения физической организации B-дерево представляется как мультисписочная структура страниц внешней памяти, т.е. каждому узлу дерева соответствует блок внешней памяти (страница). Внутренние и листовые страницы обычно имеют разную структуру.

В типовом случае структура внутренней страницы выглядит следующим образом:

$$N_1\ ключ_1\ N_2\ ключ_2\ \ldots N_n\ ключ_n\ N_{n+1}\ ключ_{n+1}$$

При этом выдерживаются следующие свойства:

- $ключ_1 \le ключ_2 \le \ldots \le ключ_n$;
- в странице дерева N_m находятся ключи k со значениями $ключ_m \le k \le ключ_{m+1}$.

Листовая страница обычно имеет следующую структуру:

$$ключ_1\ сп_1\ ключ_2\ сп_2\ \ldots ключ_t\ сп_t$$

Листовая страница обладает следующими свойствами:

- $ключ_1 < ключ_2 < \ldots < ключ_t$;
- $сп_r$ - упорядоченный список идентификаторов кортежей (tid), включающих значение $ключ_r$;
- листовые страницы связаны одно- или двунаправленным списком.

Поиск в B-дереве - это прохождение от корня к листу в соответствии с заданным значением ключа. Заметим, что поскольку деревья сильно ветвистые и сбалансированные, то для выполнения поиска по любому значению ключа потребуется одно и то же (и обычно небольшое) число обменов с внешней памятью. Более точно, в сбалансированном дереве, где длины всех путей от корня к листу одни и те же, если во внутренней странице помещается n ключей, то при хранении m записей требуется дерево глубиной $\log_n(m)$, где $\log_n$ вычисляет логарифм по основанию n. Если n достаточно велико (обычный случай), то глубина дерева невелика, и производится быстрый поиск.

Основной "изюминкой" B-деревьев является автоматическое поддержание свойства сбалансированности. Рассмотрим, как это делается при выполнении операций занесения и удаления записей.

При занесение новой записи выполняется:

- Поиск листовой страницы. Фактически, производится обычный поиск по ключу. Если в B-дереве не содержится ключ с заданным значением, то будет получен номер страницы, в которой ему надлежит содержаться, и соответствующие координаты внутри страницы.

- Помещение записи на место. Естественно, что вся работа производится в буферах оперативной памяти. Листовая страница, в которую требуется занести запись, считывается в буфер, и в нем выполняется операция вставки. Размер буфера должен превышать размер страницы внешней памяти.
- Если после выполнения вставки новой записи размер используемой части буфера не превосходит размера страницы, то на этом выполнение операции занесения записи заканчивается. Буфер может быть немедленно вытолкнут во внешнюю память, или временно сохранен в оперативной памяти в зависимости от политики управления буферами.
- Если же возникло переполнение буфера (т.е. размер его используемой части превосходит размер страницы), то выполняется расщепление страницы. Для этого запрашивается новая страница внешней памяти, используемая часть буфера разбивается грубо говоря пополам (так, чтобы вторая половина также начиналась с ключа), и вторая половина записывается во вновь выделенную страницу, а в старой странице модифицируется значение размера свободной памяти. Естественно, модифицируются ссылки по списку листовых страниц.
- Чтобы обеспечить доступ от корня дерева к заново заведенной страницы, необходимо соответствующим образом модифицировать внутреннюю страницу, являющуюся предком ранее существовавшей листовой страницы, т.е. вставить в нее соответствующее значение ключа и ссылку на новую страницу. При выполнении этого действия может снова произойти переполнение теперь уже внутренней страницы, и она будет расщеплена на две. В результате потребуется вставить значение ключа и ссылку на новую страницу во внутреннюю страницу-предка выше по иерархии и т.д.
- Предельным случаем является переполнение корневой страницы B-дерева. В этом случае она тоже расщепляется на две, и заводится новая корневая страница дерева, т.е. его глубина увеличивается на единицу.

При удалении записи выполняются следующие действия:

- Поиск записи по ключу. Если запись не найдена, то значит удалять ничего не нужно.
- Реальное удаление записи в буфере, в который прочитана соответствующая листовая страница.
- Если после выполнения этой подоперации размер занятой в буфере области оказывается таковым, что его сумма с размером занятой области в листовых страницах, являющихся левым или правым братом данной страницы, больше, чем размер страницы, операция завершается.
- Иначе производится слияние с правым или левым братом, т.е. в буфере производится новый образ страницы, содержащей общую информацию из данной страницы и ее левого или правого брата. Ставшая ненужной листовая страница заносится в список свободных страниц. Соответствующим образом корректируется список листовых страниц.
- Чтобы устранить возможность доступа от корня к освобожденной странице, нужно удалить соответствующее значение ключа и ссылку на освобожденную страницу из внутренней страницы - ее предка. При этом может возникнуть потребность в слиянии этой страницы с ее левым или правыми братьями и т.д.
- Предельным случаем является полное опустошение корневой страницы дерева, которое возможно после слияния последних двух потомков корня. В этом случае корневая страница освобождается, а глубина дерева уменьшается на единицу.

Как видно, при выполнении операций вставки и удаления свойство сбалансированности B-дерева сохраняется, а внешняя память расходуется достаточно экономно.

Проблемой является то, что при выполнении операций модификации слишком часто могут возникать расщепления и слияния. Чтобы добиться эффективного использования внешней памяти с минимизацией числа расщеплений и слияний, применяются более сложные приемы, в том числе:

- упреждающие расщепления, т.е. расщепления страницы не при ее переполнении, а несколько раньше, когда степень заполненности страницы достигает некоторого уровня;

- переливания, т.е. поддержание равновесного заполнения соседних страниц;
- слияния 3-в-2, т.е. порождение двух листовых страниц на основе содержимого трех соседних.

Следует заметить, что при организации мультидоступа к B-деревьям, характерного при их использовании в СУБД, приходится решать ряд нетривиальных проблем. Конечно, грубые решения очевидны, например монопольный захват B-дерева на все выполнение операции модификации. Но существуют и более тонкие решения, рассмотрение которых выходит за пределы нашего курса.

И последнее замечание относительно B-деревьев. В литературе вид рассмотренных нами деревьев принято называть $B*$ или $B+$-деревьями.

9.2.2. Хэширование.

Альтернативным и все более популярным подходом к организации индексов является использование техники хэширования. Это очень обширная тема, которая заслуживает отдельного рассмотрения. Мы ограничимся лишь несколькими замечаниями. Общей идеей методов хэширования является применение к значению ключа некоторой функции свертки (хэш-функции), вырабатывающей значение меньшего размера. Свертка значения ключа затем используется для доступа к записи.

В самом простом, классическом случае, свертка ключа используется как адрес в таблице, содержащей ключи и записи. Основным требованием к хэш-функции является равномерное распределение значение свертки. При возникновении коллизий (одна и та же свертка для нескольких значений ключа) образуются цепочки переполнения. Главным ограничением этого метода является фиксированный размер таблицы. Если таблица заполнена слишком сильно или переполнена, но возникнет слишком много цепочек переполнения, и главное преимущество хэширования - доступ к записи почти всегда за одно обращение к таблице - будет утрачено. Расширение таблицы требует ее полной переделки на основе новой хэш-функции (со значением свертки большего размера).

В случае баз данных такие действия являются абсолютно неприемлемыми. Поэтому обычно вводят промежуточные таблицы-справочники, содержащие значения ключей и адреса записей, а сами записи хранятся отдельно. Тогда при переполнении справочника требуется только его переделка, что вызывает меньше накладных расходов.

Чтобы избежать потребности в полной переделки справочников, при их организации часто используют технику B-деревьев с расщеплениями и слияниями. Хэш-функция при этом меняется динамически, в зависимости от глубины B-дерева. Путем дополнительных технических ухищрений удается добиться сохранения порядка записей в соответствии со значениями ключа. В целом методы B-деревьев и хэширования все более сближаются.

9.3. Журнальная информация.

Структура журнала обычно является сугубо частным делом конкретной реализации. Мы отметим только самые общие свойства.

Журнал обычно представляет собой чисто последовательный файл с записями переменного размера, которые можно просматривать в прямом или обратном порядке. Обмены производятся стандартными порциями (страницами) с использованием буфера оперативной памяти. В грамотно организованных системах структура (и тем более, смысл) журнальных записей известна только компонентам СУБД, ответственным за журнализацию и восстановление. Поскольку содержимое журнала является критичным при восстановлении базы данных после сбоев, к ведению файла журнала предъявляются особые требования по части надежности. В частности, обычно стремятся поддерживать две идентичные копии журнала на разных устройствах внешней памяти.

9.4. Служебная информация.

Для корректной работы подсистемы управления данными во внешней памяти необходимо поддерживать информация, которая используется только этой подсистемой и не видна подсистеме языкового уровня. Набор структур служебной информации зависит от общей организации системы, но обычно требуется поддержание следующих служебных данных:

- Внутренние каталоги, описывающие физические свойства объектов базы данных, например, число атрибутов отношения, их размер и, возможно, типы данных; описание индексов, определенных для данного отношения и т.д.
- Описатели свободной и занятой памяти в страницах отношения. Такая информация требуется для нахождения свободного места при занесении кортежа. Отдельно приходится решать задачу поиска свободного места в случаях некластеризованных и кластеризованных отношений (в последнем случае приходится дополнительно использовать кластеризованный индекс). Как мы уже отмечали, нетривиальной является проблема освобождения страницы в условиях мультидоступа.
- Связывание страниц одного отношения. Если в одном файле внешней памяти могут располагаться страницы нескольких отношений (обычно к этому стремятся), то нужно каким-то образом связать страницы одного отношения. Тривиальный способ использования прямых ссылок между страницами часто приводит к затруднениями при синхронизации транзакций (например, особенно трудно освобождать и заводить новые страницы отношения). Поэтому стараются использовать косвенное связывание страниц с использованием служебных индексов. В частности, известен общий механизм для описания свободной памяти и связывания страниц на основе B -деревьев.

§10. Управление транзакциями, сериализация транзакций.

Поддержание механизма транзакций - показатель уровня развитости СУБД. Корректное поддержание транзакций одновременно является основой обеспечения целостности баз данных (и поэтому транзакции вполне уместны и в однопользовательских персональных СУБД), а также составляют базис изолированности пользователей во многопользовательских системах. Часто эти два аспекта рассматриваются по отдельности, но на самом деле они взаимосвязаны, что и будет показано в этой лекции.

Заметим, что хотя с точки зрения обеспечения целостности баз данных механизм транзакций следовало бы поддерживать в персональных СУБД, на практике это обычно не выполняется. Поэтому при переходе от персональных к многопользовательским СУБД пользователи сталкиваются с необходимостью четкого понимания природы транзакций.

Под транзакцией понимается неделимая с точки зрения воздействия на БД последовательность операторов манипулирования данными (чтения, удаления, вставки, модификации) такая, что либо результаты всех операторов, входящих в транзакцию, отображаются в БД, либо воздействие всех этих операторов полностью отсутствует. Лозунг транзакции - "Все или ничего": при завершении транзакции оператором *COMMIT* результаты гарантированно фиксируются во внешней памяти (смысл слова commit - "зафиксировать" результаты транзакции); при завершении транзакции оператором *ROLLBACK* результаты гарантированно отсутствуют во внешней памяти (смысл слова rollback - ликвидировать результаты транзакции).

10.1. Транзакции и целостность баз данных.

Понятие транзакции имеет непосредственную связь с понятием целостности БД. Очень часто БД может обладать такими ограничениями целостности, которые просто невозможно не нарушить, выполняя только один оператор изменения БД. Например, в базе данных *СОТРУДНИКИ – ОТДЕЛЫ* естественным ограничением целостности является совпадения значения атрибута *ОТД _ РАЗМЕР* в кортеже отношения *ОТДЕЛЫ* , описывающем данный отдел (например, отдел 320), с числом кортежей отношения *СОТРУДНИКИ* таких, что значение атрибута *СОТР _ ОТД _ НОМ* равно 320. Как в этом случае принять на работу в отдел 320 нового сотрудника? Независимо от того, какая операция будет выполнена первой, вставка нового кортежа в отношение *СОТРУДНИКИ* или модификация существующего кортежа в отношении *ОТДЕЛЫ* , после выполнения операции база данных окажется в нецелостном состоянии.

Поэтому для поддержания подобных ограничений целостности допускается их нарушение внутри транзакции с тем условием, чтобы к моменту завершения транзакции условия целостности были соблюдены. В системах с развитыми средствами ограничения и контроля целостности каждая транзакция начинается при целостном состоянии БД и должна оставить это состояние целостными после своего завершения. Несоблюдение этого условия приводит к тому, что вместо фиксации результатов транзакции происходит ее откат (т.е. вместо оператора *COMMIT* выполняется оператор *ROLLBACK*), и БД остается в таком состоянии, в котором находилась к моменту начала транзакции, т.е. в целостном состоянии.

Если быть немного более точным, различаются два вида ограничений целостности: немедленно проверяемые и откладываемые. К немедленно проверяемым ограничениям целостности относятся такие ограничения, проверку которых бессмысленно или даже невозможно откладывать. Примером ограничения, проверку которого откладывать бессмысленно, являются ограничения домена (возраст сотрудника не может превышать 150 лет). Более сложным ограничением, проверку которого невозможно отложить, является следующее: зарплата сотрудника не может быть увеличена за одну операцию более, чем на 100,000 рублей. Немедленно проверяемые ограничения целостности соответствуют уровню отдельных операторов языкового уровня СУБД. При их нарушениях не производится откат транзакции, а лишь отвергается соответствующий оператор.

Откладываемые ограничения целостности - это ограничения на базу данных, а не на какие-либо отдельные операции. По умолчанию такие ограничения проверяются при конце транзакции, и их нарушение вызывает автоматическую замену оператора *COMMIT* на оператор *ROLLBACK* . Однако в некоторых системах поддерживается специальный оператор насильственной проверки ограничений целостности внутри транзакции. Если после выполнения такого оператора обнаруживается, что условия целостности не выполнены, пользователь может сам выполнить оператор *ROLLBACK* или постараться устранить причины нецелостного состояния базы данных внутри транзакции (видимо, это осмысленно только при использовании интерактивного режима работы).

И еще одно замечание. С точки зрения внешнего представления в момент завершения транзакции проверяются все откладываемые ограничения целостности, определенные в этой базе данных. Однако при реализации стремятся при выполнении транзакции динамически выделить те ограничения целостности, которые действительно могли бы быть нарушены. Например, если при выполнении транзакции над базой данных *СОТРУДНИКИ – ОТДЕЛЫ* в ней не выполнялись операторы вставки или удаления кортежей из отношения *СОТРУДНИКИ* , то проверять упоминавшееся выше ограничение целостности не требуется (а проверка подобных ограничений вызывает достаточно большую работу).

10.2. Изолированность пользователей.

Во многопользовательских системах с одной базой данных одновременно могут работать несколько пользователей или прикладных программ. Предельной задачей системы является обеспечение изолированности пользователей, т.е. создание достоверной и надежной иллюзии того, что каждый из пользователей работает с БД в одиночку.

В связи со свойством сохранения целостности БД транзакции являются подходящими единицами изолированности пользователей. Действительно, если с каждым сеансом работы с базой данных ассоциируется транзакция, то каждый пользователь начинает работу с согласованным состоянием базы данных, т.е. с таким состоянием, в котором база данных могла бы находиться, даже если бы пользователь работал с ней в одиночку.

При соблюдении обязательного требования поддержания целостности базы данных возможны следующие уровни изолированности транзакций:

- Первый уровень - отсутствие потерянных изменений. Рассмотрим следующий сценарий совместного выполнения двух транзакций. Транзакция 1 изменяет объект базы данных *A* . До завершения транзакции 1 транзакция 2 также изменяет объект *A* . Транзакция 2 завершается оператором *ROLLBACK* (например, по причине нарушения ограничений целостности). Тогда при повторном чтении объекта *A* транзакция 1 не видит изменений

этого объекта, произведенных ранее. Такая ситуация называется ситуацией потерянных изменений. Естественно, она противоречит требованию изолированности пользователей. Чтобы избежать такой ситуации в транзакции 1 требуется, чтобы до завершения транзакции 1 никакая другая транзакция не могла изменять объект A. Отсутствие потерянных изменений является минимальным требованием к СУБД по части синхронизации параллельно выполняемых транзакций.

- Второй уровень - отсутствие чтения "грязных данных". Рассмотрим следующий сценарий совместного выполнения транзакций 1 и 2. Транзакция 1 изменяет объект базы данных A. Параллельно с этим транзакция 2 читает объект A. Поскольку операция изменения еще не завершена, транзакция 2 видит несогласованные "грязные" данные (в частности, операция транзакции 1 может быть отвернута при проверке немедленно проверяемого ограничения целостности). Это тоже не соответствует требованию изолированности пользователей (каждый пользователь начинает свою транзакцию при согласованном состоянии базы данных и в праве ожидать видеть согласованные данные). Чтобы избежать ситуации чтения "грязных" данных, до завершения транзакции 1, изменившей объект A, никакая другая транзакция не должна читать объект A (минимальным требованием является блокировка чтения объекта A до завершения операции его изменения в транзакции 1).
- Третий уровень - отсутствие неповторяющихся чтений. Рассмотрим следующий сценарий. Транзакция 1 читает объект базы данных A. До завершения транзакции 1 транзакция 2 изменяет объект A и успешно завершается оператором $COMMIT$. Транзакция 1 повторно читает объект A и видит его измененное состояние. Чтобы избежать неповторяющихся чтений, до завершения транзакции 1 никакая другая транзакция не должна изменять объект A. В большинстве систем это является максимальным требованием к синхронизации транзакций, хотя, как мы увидим немного позже, отсутствие неповторяющихся чтений еще не гарантирует реальной изолированности пользователей.

Заметим, что существует возможность обеспечения разных уровней изолированности для разных транзакций, выполняющихся в одной системе баз данных (в частности, соответствующие операторы предусмотрены в стандарте SQL 2). Как мы уже отмечали, для поддержания целостности достаточен первый уровень. Существует ряд приложений, для которых первого уровня достаточно (например, прикладные или системные статистические утилиты, для которых некорректность индивидуальных данных несущественна). При этом удается существенно сократить накладные расходы СУБД и повысить общую эффективность.

К более тонким проблемам изолированности транзакций относится так называемая проблема кортежей-"фантомов", вызывающая ситуации, которые также противоречат изолированности пользователей. Рассмотрим следующий сценарий. Транзакция 1 выполняет оператор A выборки кортежей отношения R с условием выборки S (т.е. выбирается часть кортежей отношения R, удовлетворяющих условию S). До завершения транзакции 1 транзакция 2 вставляет в отношение R новый кортеж r, удовлетворяющий условию S, и успешно завершается. Транзакция 1 повторно выполняет оператор A, и в результате появляется кортеж, который отсутствовал при первом выполнении оператора. Конечно, такая ситуация противоречит идее изолированности транзакций и может возникнуть даже на третьем уровне изолированности транзакций. Чтобы избежать появления кортежей-фантомов, требуется более высокий "логический" уровень синхронизации транзакций. Идеи такой синхронизации (предикатные синхронизационные захваты) известны давно, но в большинстве систем не реализованы.

10.3. Сериализация транзакций.

Понятно, что для того, чтобы добиться изолированности транзакций, в СУБД должны использоваться какие-либо методы регулирования совместного выполнения транзакций.

План (способ) выполнения набора транзакций называется сериальным, если результат совместного выполнения транзакций эквивалентен результату некоторого последовательного выполнения этих же транзакций.

Сериализация транзакций - это механизм их выполнения по некоторому сериальному плану. Обеспечение такого механизма является основной функцией компонента СУБД, ответственного за управление транзакциями. Система, в которой поддерживается сериализация транзакций обеспечивает реальную изолированность пользователей.

Основная реализационная проблема состоит в выборе метода сериализации набора транзакций, который не слишком ограничивал бы их параллельность. Приходящим на ум тривиальным решением является действительно последовательное выполнение транзакций. Но существуют ситуации, в которых можно выполнять операторы разных транзакций в любом порядке с сохранением сериальности. Примерами могут служить только читающие транзакции, а также транзакции, не конфликтующие по объектам базы данных.

Между транзакциями могут существовать следующие виды конфликтов:

- W-W - транзакция 2 пытается изменять объект, измененный не закончившейся транзакцией 1;
- R-W - транзакция 2 пытается изменять объект, прочитанный не закончившейся транзакцией 1;
- W-R - транзакция 2 пытается читать объект, измененный не закончившейся транзакцией 1.

Практические методы сериализации транзакций основывается на учете этих конфликтов.

§11. Методы сериализации транзакций.

Существуют два базовых подхода к сериализации транзакций - основанный на синхронизационных захватах объектов базы данных и на использовании временных меток. Суть обоих подходов состоит в обнаружении конфликтов транзакций и их устранении. Ниже мы рассмотрим эти подходы сравнительно подробно.

Предварительно заметим, что для каждого из подходов имеются две разновидности – пессимистическая и оптимистическая. При применении пессимистических методов, ориентированных на ситуации, когда конфликты возникают часто, конфликты распознаются и разрешаются немедленно при их возникновении. Оптимистические методы основываются на том, что результаты всех операций модификации базы данных сохраняются в рабочей памяти транзакций. Реальная модификация базы данных производится только на стадии фиксации транзакции. Тогда же проверяется, не возникают ли конфликты с другими транзакциями.

Далее мы ограничимся рассмотрением более распространенных пессимистических разновидностей методов сериализации транзакций. Пессимистические методы сравнительно просто трансформируются в свои оптимистические варианты.

11.1. Синхронизационные захваты.

Наиболее распространенным в централизованных СУБД (включающих системы, основанные на архитектуре "клиент-сервер") является подход, основанный на соблюдении двухфазного протокола синхронизационных захватов объектов БД. В общих чертах протокол состоит в том, что перед выполнением любой операции в транзакции T над объектом базы данных r от имени транзакции T запрашивается синхронизационный захват объекта r в соответствующем режиме (в зависимости от вида операции).

Основными режимами синхронизационных захватов являются:

- совместный режим - S (Shared), означающий разделяемый захват объекта и требуемый для выполнения операции чтения объекта;
- монопольный режим - X (eXclusive), означающий монопольный захват объекта и требуемый для выполнения операций занесения, удаления и модификации.

Захваты объектов несколькими транзакциями по чтению совместимы, т.е. нескольким транзакциям допускается читать один и тот же объект, захват объекта 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 следующей таблице:

	X	S
-	да	да
X	нет	нет
S	нет	да

В первом столбце приведены возможные состояния объекта с точки зрения синхронизационных захватов. При этом "-" соответствует состоянию объекта, для которого не установлен никакой захват. Транзакция, запросившая синхронизационный захват объекта БД, уже захваченный другой транзакцией в несовместимом режиме, блокируется до тех пор, пока захват с этого объекта не будет снят.

Заметим, что слово "нет" в нашей таблице соответствует описанным ранее возможным случаям конфликтов транзакций по доступу к объектам базы данных (WW, RW, WR). Совместимость *S* -захватов соответствует тому, что конфликт RR не существует.

Для обеспечения сериализации транзакций (третьего уровня изолированности) синхронизационные захваты объектов, произведенные по инициативе транзакции, можно снимать только при ее завершении. Это требование порождает двухфазный протокол синхронизационных захватов - 2PL. В соответствии с этим протоколом выполнение транзакции разбивается на две фазы:

- первая фаза транзакции - накопление захватов;
- вторая фаза (фиксация или откат) - освобождение захватов.

Достаточно легко убедиться, что при соблюдении двухфазного протокола синхронизационных захватов действительно обеспечивается сериализация транзакций на третьем уровне изолированности. Основная проблема состоит в том, что следует считать объектом для синхронизационного захвата?

В контексте реляционных баз данных возможны следующие альтернативы:

- файл - физический (с точки зрения базы данных) объект, область хранения нескольких отношений и, возможно, индексов;
- отношение - логический объект, соответствующий множеству кортежей данного отношения;
- страница данных - физический объект, хранящий кортежи одного или нескольких отношений, индексную или служебную информацию;
- кортеж - элементарный физический объект базы данных.

На самом деле, когда мы говорим про операции над объектами базы данных, то любая операция над кортежем, фактически, является и операцией над страницей, в которой этот кортеж хранится, и над соответствующим отношением, и над файлом, содержащем отношение. Поэтому действительно имеется выбор уровня объекта захвата.

Понятно, что чем крупнее объект синхронизационного захвата (неважно, какой природы этот объект - логический или физический), тем меньше синхронизационных захватов будет поддерживаться в системе, и на это, соответственно, будут тратиться 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.

Разработчики многих систем начинали с использования страничных захватов, полагая это некоторым компромиссом между стремлениями сократить накладные расходы и сохранить достаточно высокий уровень параллельности транзакций. Но это не очень хороший выбор. Мы не

будем останавливаться на деталях, но заметим, что использование страничных захватов в двухфазном протоколе иногда вызывает очень неприятные синхронизационные проблемы, усложняющие организацию СУБД. В большинстве современных систем используются покортежные синхронизационные захваты.

Но при этом возникает очередной вопрос. Если единицей захвата является кортеж, то какие синхронизационные захваты потребуются при выполнении таких операций как уничтожение отношения? Было бы довольно нелепо перед выполнением такой операции потребовать захвата всех существующих кортежей отношения. Кроме того, это не предотвратило бы возможности параллельной вставки в другой транзакции нового кортежа в уничтожаемое отношение.

11.1.1. Гранулированные синхронизационные захваты.

Подобные рассуждения привели к разработки аппарата гранулированных синхронизационных захватов. 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 или X.

Теперь наиболее важное отличие, на котором, собственно, держится соответствие захватов разного уровня. Вводится специальные протокол гранулированных захватов и новые типы захватов: перед захватом объекта в режиме S или X соответствующий объект более верхнего уровня должен быть захвачен в режиме IS, IX или SIX. Что же из себя представляют эти режимы захватов?

IS (Intented for Shared lock) 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) 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) 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).

Довольно трудно описать словами все возможные ситуации. Мы ограничимся приведением полной таблицы совместимости захватов, анализируя которую можно выявить все случаи:

	X	S	IX	IS	SIX
-	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

11.1.2. 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). Давно известно, что для решения этой проблемы необходимо перейти от захватов индивидуальных объектов базы данных, к захвату усло-

вий (предикатов), которым удовлетворяют эти объекты. Проблема фантомов не возникает при использовании для синхронизации уровня отношений именно потому, что отношение как логический объект представляет собой неявное условие для входящих в него кортежей. Захват отношения - это простой и частный случай предикатного захвата.

Поскольку любая операция над реляционной базой данных задается некоторым условием (т.е. 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 { = > < } значение
```

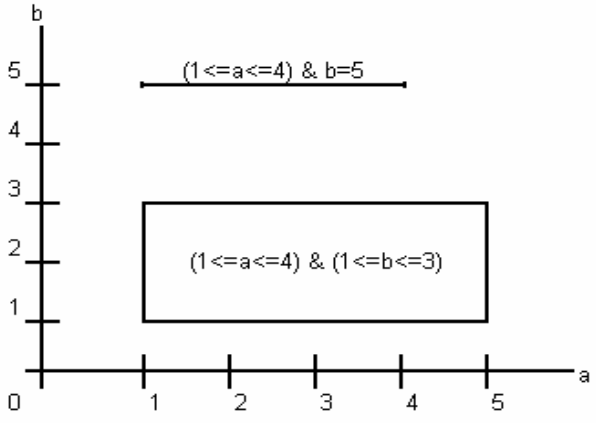
В типичных СУБД, поддерживающих двухуровневую организацию (языковой уровень и уровень управления внешней памяти), в интерфейсе подсистем управления памятью (которая обычно заведует и сериализацией транзакций) допускаются только простые условия. Подсистема языкового уровня производит компиляцию исходного 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 ($m \le n$).

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 $(1 \le a \le 4)\ and\ (b = 5)$, а транзакция 2 - условия $(1 \le a \le 5)\ and\ (1 \le b \le 3)$, эти захваты всегда совместимы.

Пример: ($n = 2$)



Заметим, что предикатные захваты простых условий описываются таблицами, немногим отличающимися от таблиц традиционных синхронизаторов.

11.1.3. Тупики, распознавание и разрушение.

Одним из наиболее чувствительных недостатков метода сериализации транзакций на основе синхронизационных захватов является возможность возникновение тупиков (deadlocks) между транзакциями. Тупики возможны при применении любого из рассмотренных нами вариантов.

Вот простой пример возникновения тупика между транзакциями $T1$ и $T2$:

- транзакции $T1$ и $T2$ установили монопольные захваты объектов $r1$ и $r2$ соответственно;
- после этого $T1$ требуется совместный захват $r2$, а $T2$ - совместный захват $r1$;
- ни одна из транзакций не может продолжаться, следовательно, монопольные захваты не будут сняты, а совместные - не будут удовлетворены.

Поскольку тупики возможны, и никакого естественного выхода из тупиковой ситуации не существует, то эти ситуации необходимо обнаруживать и искусственно устранять.

Основой обнаружения тупиковых ситуаций является построение (или постоянное поддержание) графа ожидания транзакций. Граф ожидания транзакций - это ориентированный двудольный граф, в котором существует два типа вершин - вершины, соответствующие транзакциям, и вершины, соответствующие 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.

Легко показать, что в системе существует ситуация тупика, если в графе ожидания транзакций имеется хотя бы один цикл.

Для распознавание тупика периодически производится построение графа ожидания транзакций (как уже отмечалось, иногда граф ожидания поддерживается постоянно), 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.

Предположим, что нам удалось найти цикл в графе ожидания транзакций. Что делать теперь? Нужно каким-то образом обеспечить возможность продолжения работы хотя бы для части транзакций, попавших в тупик. Разрушение тупика начинается с выбора в цикле транзакций так называемой транзакции-жертвы, т.е. транзакции, которой решено пожертвовать, чтобы обеспечить возможность продолжения работы других транзакций.

Грубо говоря, критерием выбора является стоимость транзакции; жертвой выбирается самая дешевая транзакция. Стоимость транзакции определяется на основе многофакторная оценка, 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.

Естественно, такое насильственное устранение тупиковых ситуаций является нарушением принципа изолированности пользователей, которого невозможно избежать.

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.

Еще одно замечание. Чтобы минимизировать число конфликтов между транзакциями, в некоторых СУБД (например, в Oracle) используется следующее развитие подхода. Монопольный захват объекта блокирует только изменяющие транзакции. После выполнении операции

модификации предыдущая версия объекта остается доступной для чтения в других транзакциях. Кратковременная блокировка чтения требуется только на период фиксации изменяющей транзакции, когда обновленные объекты становятся текущими.

11.2. 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 $t(T) > t(T1)$ (т.е. транзакция T является более "молодой", чем T), 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.

§12. Журнализация изменений БД.

Одним из основных требований к развитым СУБД является надежность хранения баз данных. 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 поддерживается в виде журнала изменений базы данных.

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. Тривиальной ситуацией отката транзакции является ее явное завершение оператором *ROLLBACK* . Возможны также ситуации, когда откат транзакции инициируется системой. Примерами могут быть возникновение исключительной ситуации в прикладной программе (например, деление на ноль) или выбор транзакции в качестве жертвы при обнаружении синхронизационного тупика. Для восстановления согласованного состояния базы данных при индивидуальном откате транзакции нужно устранить последствия операторов модификации базы данных, которые выполнялись в этой транзакции.
- Восстановление после внезапной потери содержимого оперативной памяти (мягкий сбой). Такая ситуация может возникнуть 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.

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. Эти локальные журналы 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.

12.1. Журнализация и буферизация.

Журнализация изменений тесно связана не только с управлением транзакциями, но и с буферизацией страниц базы данных в оперативной памяти. По причинам объективно существующей разницы в скорости работы процессоров и 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.

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 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 буфера могут выталкиваться во внешнюю память. Проблема состоит в выработке некоторой

общей политики выталкивания, которая обеспечивала бы возможности 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 вытолкнуть во внешнюю память измененный объект базы данных, то перед этим нужно гарантировать выталкивание во внешнюю память журнала 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, т.е. 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.

12.2. Индивидуальный откат транзакции.

Для того, чтобы можно было выполнить по общему журналу индивидуальный откат транзакции, все записи в журнале от 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 журнализуются. 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.

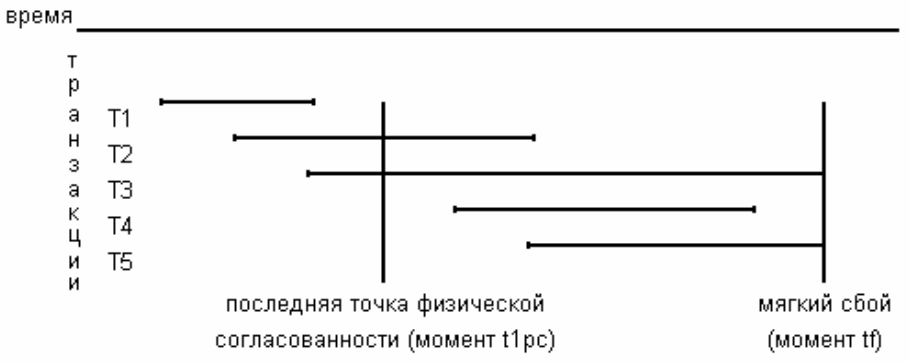
12.3. Восстановление после мягкого сбоя.

К числу основных проблем восстановление 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, т.е. часть страниц внешней памяти соответствует объекту до изменения, часть - после изменения. К такому состоянию объекта не применимы операции логического уровня.

Состояние внешней памяти базы данных называется физически согласованным, если наборы страниц всех объектов согласованы, т.е. соответствуют состоянию объекта либо после его изменения, либо до 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).

Тогда к моменту мягкого сбоя возможны следующие состояния транзакций:



Предположим, что некоторым способом удалось восстановить внешнюю память базы данных к состоянию на момент времени tlpc (как это можно сделать - немного позже). Тогда:

Для транзакции $T1$ никаких действий производить не требуется. Она закончилась до момента tlpc, 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 $t1pc$. 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 $t1pc$. С другой стороны, во внешней памяти гарантированно присутствуют результаты операций $T3$, которые были выполнены до момента $t1pc$. 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 $t1pc$ и закончиться до момента мягкого сбоя, нужно выполнить полную повторную прямую интерпретацию операций (redo).

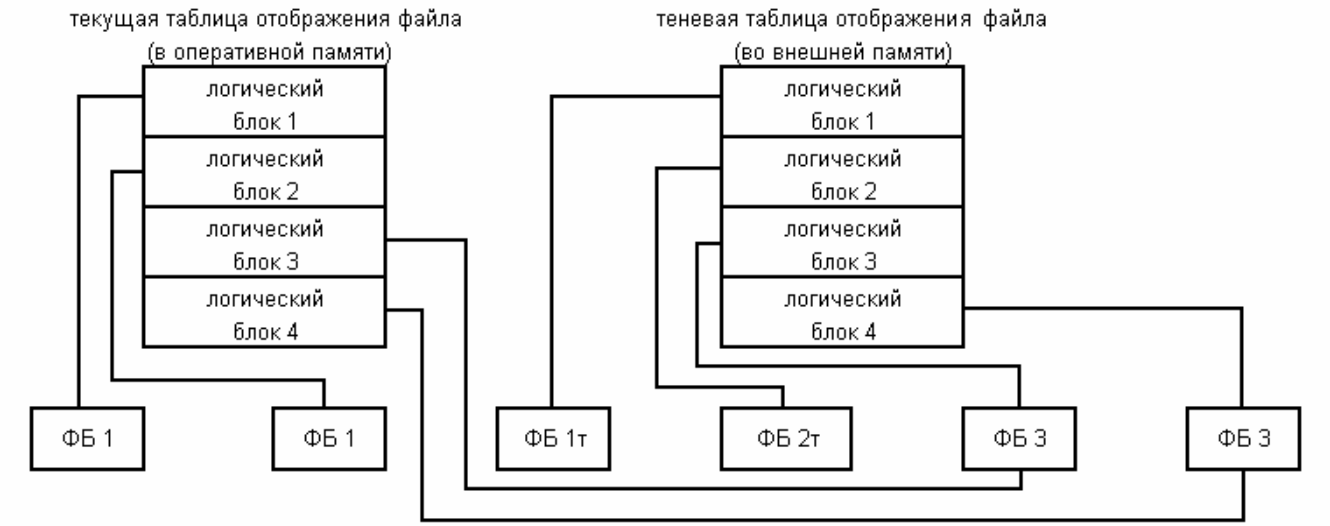
Наконец, для начавшейся после момента $t1pc$ и не успевшей завершиться к моменту мягкого сбоя транзакции $T5$ никаких действий предпринимать не требуется. Результаты операций этой транзакции полностью отсутствуют во внешней памяти базы данных.

12.4. Физическая согласованность базы данных.

Каким же образом можно обеспечить наличие точек физической согласованности базы данных, т.е. как восстановить состояние базы данных в момент tpc? Для этого используются два основных подхода: подход, основанный на использовании 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 следующем рисунке:



В контексте базы данных теневой механизм используется следующим образом. Периодически выполняются операции установления точки физической согласованности базы данных (checkpoints в System R). Для этого все логические операции завершаются, все буфера 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 $t1pc$ 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 набор согласованных наборов страниц, каждому из которых может соответствовать свой набор времени.

Для достижения 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 незакончив-шихся 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. Для того, чтобы распознать, нуждается ли страница внешней памяти базы данных в восстановлении, при выталкивании любой страницы из буфера оперативную память в нее помещается идентификатор последней записи о постраничном изменении этой страницы. Имеются и другие технические нюансы.

В этом подходе имеются два поднаправления. В первом поднаправлении 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.

12.5. 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, т.е. операции повторно выполняются в прямом смысле.

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.

Язык реляционных баз данных SQL

§13. Язык SQL. Функции и основные возможности.

13.1. SEQUEL/SQL СУБД System R.

Язык для взаимодействия с БД SQL появился в середине 70-х и был разработан в рамках проекта экспериментальной реляционной СУБД System R. Исходное название языка SEQUEL (Structered English Query Language) только частично отражает суть этого языка. Конечно, язык был ориентирован главным образом на удобную и понятную пользователям формулировку запросов к реляционной БД, но на самом деле уже являлся полным языком БД, содержащим помимо операторов формулирования запросов и манипулирования БД средства определения и манипулирования схемой БД; определения ограничений целостности и триггеров; представлений БД; возможности определения структур физического уровня, поддерживающих эффективное выполнение запросов; авторизации доступа к отношениям и их полям; точек сохранения транзакции и откатов. В языке отсутствовали средства синхронизации доступа к объектам БД со стороны параллельно выполняемых транзакций: с самого начала предполагалось, что необходимую синхронизацию неявно выполняет СУБД.

Рассмотрим эти свойства языка немного более подробно.

13.1.1. Запросы и операторы манипулирования данными.

Как известно, двумя фундаментальными языками запросов к реляционным БД являются языки реляционной алгебры и реляционного исчисления. При всей своей строгости и теоретической обоснованности эти языки редко используются в современных реляционных СУБД в качестве средств пользовательского интерфейса. Запросы на этих языках трудно формулировать и понимать. SQL представляет собой некоторую комбинацию реляционного исчисления кортежей и реляционной алгебры, причем до сих пор нет общего согласия, к какому из классических языков он ближе. При этом возможности SQL шире, чем у этих базовых реляционных языков, в частности, в общем случае невозможна трансляция запроса, сформулированного на SQL, в выражение реляционной алгебры, требуется некоторое ее расширение.

Существенными свойствами подъязыка запросов SQL являются возможность простого формулирования запросов с соединениями нескольких отношений и использование вложенных подзапросов в предикатах выборки. Вообще говоря, одновременное наличие обоих средств избыточно, но это дает пользователю при формулировании запроса возможность выбора более понятного ему варианта.

В предикатах со вложенными подзапросами в SQL System R можно употреблять теретико-множественные операторы сравнения, что позволяет формулировать квантифицированные запросы (эти возможности обычно труднее всего понимаются пользователями и поэтому в дальнейшем в SQL появились явно квантифицируемые предикаты).

Существенной особенностью SQL является возможность указания в запросе потребности группирования отношения-результата по указанным полям с поддержкой условий выборки на всю группу целиком. Такие условия выборки могут содержать агрегатные функции, вычисляемые на группе. Эта возможность SQL главным образом отличает этот язык от языков реляционной алгебры и реляционного исчисления, не содержащих аналогичных средств.

Еще одним отличием SQL является необязательное удаление кортежей-дубликатов в окончательном или промежуточных отношениях-результатах. Строго говоря, результатом оператора выборки в языке SQL является не отношение, а мультимножество кортежей. В тех случаях, когда семантика запроса требует наличия отношения, уничтожение дубликатов производится неявно.

Самый общий вид запроса на языке SQL представляет теоретико-множественное алгебраическое выражение, составленное из элементарных запросов. В SQL System R допускались все базовые теретико-множественные операции (*UNION*, *INTERSECT* и *MINUS*).

Работа с неопределенными значениями в SQL System R до конца продумана не была, хотя неявно предполагалось использование трехзначной логики при вычислении логических выражений.

Операторы манипулирования данными *UPDATE* и *DELETE* построены на тех же принципах, что и оператор выборки данных *SELECT*. Набор кортежей указанного отношения, подлежащих модификации или удалению, определяется входящим в соответствующий оператор логи-

ческим выражением, которое может включать сложные предикаты, в том числе и с вложенными подзапросами.

В операторе вставки кортежа(ей) в указанное отношение заносимый кортеж может задаваться как в литеральной форме, так и с помощью внутреннего подоператора выборки.

13.1.2. Операторы определения и манипулирования схемой БД.

В число операторов определения схемы БД SQL System R входили операторы создания и уничтожения постоянных и временных хранимых отношений (*CREATE TABLE* и *DROP TABLE*) и создания и уничтожения представляемых отношений (*CREATE VIEW* и *DROP VIEW*). В языке и в реализации System R не запрещалось использовать операторы определения схемы в пределах транзакции, содержащей операторы выборки и манипулирования данными. Допускалось, например, использование операторов выборки и манипулирования данными, в которых указываются отношения, не существующие в БД к моменту компиляции оператора. Конечно, эта возможность существенно усложняла реализацию и требовалась по существу очень редко.

Оператор манипулирования схемой БД *ALTER TABLE* позволял добавлять указываемые поля к существующим отношениям. В описании языка определялось, что выполнение этого оператора не должно приводить к недействительности ранее откомпилированных операторов над отношением, схема которого изменяется, и что значения вновь определенных полей в существующих кортежах отношения становятся неопределенными.

13.1.3. Определения ограничений целостности и триггеров.

Язык SQL System R включал очень мощные средства контроля и поддержания целостности БД. Средства контроля базировались на аппарате ограничений целостности (*ASSERTIONS*). Фактически, ограничение целостности - это логическое выражение, вычисляемое над текущим состоянием БД, ложность которого соответствует нецелостному состоянию БД. Логическое выражение ограничения целостности могло содержать любой допустимый в языке предикат.

Более точно, ограничения целостности делились на два класса: проверяемые после выполнения оператора манипулирования данными и проверяемые при завершении транзакции или при выполнении специального оператора *INFORCE INTEGRITY* . Типы предикатов, которые можно использовать в операторах определения ограничений целостности разных классов, различаются. В операторах первого класса проверяется, фактически, текущий кортеж, с которым производится манипулирование. Во втором случае проверяются указанные в ограничении целостности отношения, т.е. все их кортежи. Различается и определяемая в языке реакция системы на нарушения ограничений целостности разных классов. В первом случае нарушение ограничения целостности приводит к откату транзакции в точку, непосредственно предшествующую операции манипулирования данными, выполнение которого вызвало нарушение ограничения целостности. Во втором случае ограничение приводит к полному откату транзакции к ее началу.

Очень важным механизмом, определенным в языке SQL System R, является механизм триггеров. В контексте System R этот механизм рассматривался главным образом как средство автоматического поддержания целостности БД. При определении триггера указывалось условие проверки его применимости (имя отношения и тип операции манипулирования данными), условие применимости триггера (логическое выражение, построенное по правилам, близким к правилам для ограничений целостности первого класса) и действие, которое должно быть выполнено над БД в случае истинности условия применимости. Такое действие могло быть выражено с помощью произвольного оператора манипулирования данными. Во время выполнения действия могли срабатывать другие триггеры и т.д.

Механизмы ограничений целостности и триггеров System R являлись очень мощными и общими, но реализация их очень трудна и накладна (как уже отмечалось, триггеры так и не были реализованы в System R). Дополнительную сложность в реализации создавал тот факт, что допускалось (по крайней мере не запрещалось языком) определение ограничений целостности и триггеров в пределах той же транзакции, в которой выполняются операторы манипулирования данными. При наиболее полной реализации требовалось бы большое число дополнительных

действий во время выполнения транзакции. Кроме того, в ряде случаев отсутствие зафиксированной семантики соответствующих конструкций языка приводило к неоднозначному пониманию выполнения транзакций.

13.1.4. Представления базы данных.

В языке допускалось использование хранимых отношений БД и представляемых отношений. Наиболее удачным решением было использование для определения представлений общего аппарата операторов выборки. Любой оператор выборки может быть использован для определения представления.

В языке отсутствуют какие-либо ограничения по поводу использования представлений: в любом операторе SQL, в котором допускается использование имени хранимого отношения, допускается и использование имени представления. В SQL System R ничего не говорится о рекомендуемом способе реализации доступа к представлениям, но при любом способе эффект должен быть таким, как если бы выполнить полную материализацию представления до выполнения оператора.

Массу проблем, исследований и предложений породила потенциальная возможность выполнения операторов манипулирования данными над представлениями. Понятно, что эта возможность легко реализуема для простых представлений, но в более сложных случаях не только реализация, но и семантика операций становится нетривиальной. Кстати, в System R операторы манипулирования данными допускались только над простыми представлениями.

13.1.5. Определение управляющих структур.

Внесение в реляционный язык, каким является SQL, явных операторов порождения и уничтожения структур физического уровня, поддерживающих эффективное выполнение запросов к БД, явилось в SQL System R чисто прагматическим решением, обеспечивающим возможность всех видов работ с БД с помощью одного языка.

В SQL System R упоминаются два вида таких структур: индексы и связи (*links*). Индекс в его абстрактном языковом представлении - это инвертированный файл, обеспечивающий доступ к кортежам соответствующего отношения на основе заданных значений одного или нескольких столбцов, составляющих ключ индекса. Операторы языка позволяли создавать и уничтожать индексы, но никаким образом не давали возможности явно указать на необходимость использования существующего индекса при выполнении оператора выборки, решение об этом возлагалось на реализацию.

С помощью оператора определения индекса можно было выразить два дополнительных утверждения, касающихся логической схемы отношения и физической структуры его хранения. Использование при определении индекса ключевого слова *UNIQUE* означало, что ключ этого индекса является возможным ключом соответствующего отношения. Фактически это означает наличие дополнительного механизма определения ограничения целостности отношения. Один из индексов для данного отношения мог быть определен с ключевым словом *CLUSTERING*. Это означает требование физической кластеризации во внешней памяти кортежей отношения с равными или близкими значениями ключа индекса.

Операторы определения связи позволяли в стиле сетевой модели данных организовать во внешней памяти списки кортежей указанного отношения. Как и в случае индексов, операторы позволяли создавать и уничтожать такие списки, но не давали возможности явно указать на необходимость использования существующих списков при выполнении операторов выборки. Большая трудоемкость поддержания списков при выполнении операторов манипулирования данными и трудность выполнения оценок стоимости их использования при выполнении операторов выборки привели к тому, что механизм связей исчез из языка уже на поздней стадии проекта System R. С тех пор этот механизм, насколько нам известно, не появлялся ни в одном варианте SQL.

13.1.6. Авторизация доступа к отношениям и их полям.

Существенной особенностью языка SQL, появившейся в нем с самого начала, является обеспечение защиты доступа к данным средствами самого языка. Основная идея такого подхода состоит в том, что по отношению к любому отношению БД и любому столбцу отношения

вводится предопределенный набор привилегий. С каждой транзакцией неявно связывается идентификатор пользователя, от имени которого она выполняется (способы связи и идентификации пользователей не фиксируются в языке и определяются в реализации).

После создания нового отношения все привилегии, связанные с этим отношением и всеми его столбцами, принадлежат только пользователю-создателю отношения. В число привилегий входит привилегия передачи всех или части привилегий другому пользователю, включая привилегию на передачу привилегий. Технически передача привилегий осуществляется при выполнении оператора SQL *GRANT*. Существует также привилегия изъятия всех или части привилегий у пользователя, которому они ранее были переданы. Эта привилегия также может передаваться. Технически изъятие привилегий происходит при выполнении оператора SQL *REVOKE*.

Проверка полномочности доступа к данным происходит на основе информации о полномочиях, существующих во время компиляции соответствующего оператора SQL. Подобно тому, что мы отмечали в связи с ограничениями целостности и триггерами, в SQL System R отсутствовали какие-либо ограничения по поводу использования операторов *GRANT* и *REVOKE*. Это приводило к существенным техническим затруднениям в реализации, а иногда к неоднозначному пониманию поведения.

Долгое время подход к защите данных от несанкционированного доступа принимался практически без критики, однако в связи с распространяющимся использованием реляционных СУБД в нетрадиционных приложениях все чаще раздается критика. Если, например, в системе БД должна поддерживаться многоуровневая защита данных, соответствующую систему полномочий весьма трудно, а иногда и невозможно построить на основе средств SQL.

13.1.7. Точки сохранения и откаты транзакции.

В SQL System R существовали два специальных оператора для установки так называемых точек сохранения транзакции и для отката транзакции к ранее установленной точке сохранения. В литературе, относящейся к System R, обсуждение этих возможностей практически не содержится, из чего неявно следует, что они не были реализованы.

Прямолинейная реализация этого механизма не вызывает особых технических затруднений, но и не очень полезна, потому что после выполнения частичного отката транзакции для успешного продолжения работы прикладной программы потребовалось бы и восстановить ее состояние в соответствующей точке, а это никак не поддерживается. Понятно, что при более тщательной проработке должны быть увязаны механизмы точек сохранения и контроля целостности. Например, было бы естественно, чтобы при выполнении оператора *ENFORCE INTEGRITY*, если какие-либо ограничения целостности нарушаются, происходил автоматический откат транзакции к ближайшей точки сохранения, в которой нарушения целостности БД не было. Это значительно усложнило бы реализацию, но было бы очень полезно. Аналогично, можно было бы использовать механизм точек сохранения при автоматических откатах транзакций по причине возникновения синхронизационных тупиков.

Отметим еще два важных свойства языка SQL System R, которые в разных видах присутствуют во всех развитых последующих вариантах языка.

13.1.8. Встроенный SQL.

В SQL System R присутствуют специальные операторы, поддерживающие встраивание операторов SQL в традиционные языки программирования (в System R основным таким языком был PL/1).

Основная проблема встраивания SQL в язык программирования состояла в том, что SQL - реляционный язык, т.е. его операторы большей частью работают со множествами, в то время как в языках программирования основными являются скалярные операции. Решение SQL состоит в том, что в язык дополнительно включаются операторы, обеспечивающие покортежный доступ к результату запроса к БД.

Для этого в язык вводится понятие курсора, с которым связывается оператор выборки. Над определенным курсором можно выполнять оператор *OPEN*, означающий материализацию отношения-результата запроса, оператор *FETCH*, позволяющий выбрать очередной кортеж

результирующего отношения в память программы, и оператор *CLOSE*, означающий конец работы с данным курсором.

Дополнительную гибкость при создании прикладных программ со встроенным SQL обеспечивает возможность параметризации операторов SQL значениями переменных включающей программы.

13.1.9. Динамический SQL.

Для упрощения создания интерактивных SQL-ориентированных систем в SQL System R были включены операторы, позволяющие во время выполнения транзакции откомпилировать и выполнить любой оператор SQL.

Оператор *PREPARE* вызывает динамическую компиляцию оператора SQL, текст которого содержится в указанной переменной символьной строке включающей программы. Текст может быть помещен в переменную при выполнении программы любым допустимым способом, например, введен с терминала.

Оператор *DESCRIBE* служит для получения информации об указанном операторе SQL, ранее подготовленном с помощью оператора *PREPARE*. С помощью этого оператора можно узнать, во-первых, является ли подготовленный оператор оператором выборки, и во-вторых, если это оператор выборки, получить полную информацию о числе и типах столбцов результирующего отношения.

Для выполнения ранее подготовленного оператора SQL, не являющегося оператором выборки, служит оператор *EXECUTE*. Для выполнения динамически подготовленного оператора выборки используется аппарат курсоров с некоторыми отличиями по части задания адресов переменных включающей программы, в которые должны быть помещены значения столбцов текущего кортежа результата.

Подводя итог приведенному краткому описанию основных черт SQL System R, отметим, что несмотря на недостаточную техническую проработку, в идейном отношении язык содержал все необходимые средства, позволяющие использовать его как базовый язык СУБД.

13.2. Язык SQL в коммерческих реализациях.

В настоящее время SQL реализован практически во всех коммерческих реляционных СУБД, все фирмы провозглашают соответствие своей реализации стандарту SQL, и на самом деле реализованные диалекты SQL очень близки. Это произошло не сразу и не просто.

Наиболее близкими к System R являлись две системы фирмы IBM - SQL/DS и DB2. Как кажется (документальных подтверждений этому автор не имеет), обе эти системы прямо использовали реализацию System R. Отсюда предельная близость реализованных диалектов SQL к SQL System R. Из SQL System R были удалены только те части, которые были недостаточно проработаны (например, точки сохранения) или реализация которых вызывала слишком большие технические трудности (например, ограничения целостности и триггеры). Можно назвать этот путь к коммерческой реализации SQL движением сверху вниз.

Другой подход применялся в таких системах, как Oracle и Informix. Несмотря на различие в способе разработки этих систем, реализация SQL происходила "снизу вверх". В первых выпущенных на рынок реализациях SQL в этих системах использовалось минимальное и очень ограниченное подмножество SQL System R. В частности, в первой известной автору реализации SQL в СУБД Oracle в операторах выборки не допускалось использование вложенных подзапросов.

Тем не менее, несмотря на эти ограничения и на очень слабую на первых порах эффективность, ориентация фирм на поддержание разных аппаратных платформ и заинтересованность пользователей в переходе к реляционным системам позволили фирмам добиться коммерческого успеха и приступить к совершенствованию своих реализаций. В текущих версиях Oracle и Informix поддерживаются достаточно мощные диалекты SQL, хотя реализация иногда вызывает сомнения.

Особенностью большинства современных коммерческих СУБД, затрудняющей анализ существующих диалектов SQL, является отсутствие полного описания языка. Обычно описание разбросано по разным руководствам и перемешано с описанием специфических для данной системы языковых средств, не имеющих отношения к SQL. Тем не менее можно сказать, что

базовый набор операторов SQL, включающий операторы определения схемы БД, выборки и манипулирования данными, авторизации доступа к данным, поддержки встраивания SQL в языки программирования и операторы динамического SQL, в коммерческих реализациях относительно устоялся и более или менее соответствует стандарту.

13.3. Стандартизация SQL.

Деятельность по стандартизации языка SQL началась практически одновременно с появлением первых его коммерческих реализаций. Первый из числа имеющихся у автора документ датирован октябрем 1985 г. и является уже очередным проектом стандарта ANSI/ISO.

Понятно, что в качестве стандарта нельзя было использовать SQL System R. Во-первых, этот вариант языка не был должным образом технически проработан. Во-вторых, его слишком сложно было бы реализовать (кто знает, как бы сложилась дальнейшая история SQL, если бы были полностью реализованы все идеи System R). С другой стороны, первые коммерческие реализации языка настолько различались, что ни один из реализованных диалектов не имел шансов быть принятым в качестве стандарта.

Анализ доступных документов показывает, что процесс происходил очень сложно с использованием далеко не только научных доводов. В результате принятый в 1989 г. Международный стандарт SQL во многих частях имеет чрезвычайно общий характер и допускает очень широкое толкование. В этом стандарте полностью отсутствуют такие важные разделы, как манипулирование схемой БД и динамический SQL. Многие важные аспекты языка в соответствии со стандартом определяются в реализации.

Возможно, наиболее важными достижениями стандарта SQL являются четкая стандартизация синтаксиса и семантики операторов выборки и манипулирования данными и фиксация средств ограничения целостности БД, включающих возможности определения первичного и внешних ключей отношений и так называемых проверочных ограничений целостности, которые представляют собой подмножество ограничений целостности SQL System R первого класса. Средства определения внешних ключей позволяют легко формулировать требования так называемой целостности БД по ссылкам. Это распространенное в БД требование можно сформулировать и на основе общего механизма ограничений целостности SQL System R, но формулировка на основе понятия внешнего ключа более проста и понятна.

Осознавая неполноту стандарта SQL, на фоне завершения разработки этого стандарта специалисты различных фирм начали работу над стандартом SQL2. Эта работа также длилась несколько лет, было выпущено множество проектов стандарта, пока, наконец, в марте 1992 г. не был выработан окончательный проект стандарта. Этот стандарт существенно более полный и охватывает практически все необходимые для реализации аспекты: манипулирование схемой БД, управление транзакциями (опять появились точки сохранения) и сессиями (сессия - это последовательность транзакций, в пределах которой сохраняются временные отношения), подключение к БД, динамический SQL. Наконец стандартизованы отношения-каталоги БД, что вообще-то не связано с языком непосредственно, но очень сильно влияет на реализацию.

Удивляет отсутствие в стандарте средств управления индексами. Конечно, эти средства обычно находятся в стороне от основных операторов SQL, но автору не известна ни одна реализация, в которой бы их не было.

Наконец, одновременно с завершением работ по определению стандарта SQL2 была начата разработка стандарта SQL3. Предполагается, что SQL3 будет содержать механизм триггеров и возможность использования абстрактных типов данных. Принятие стандарта планируется только в 1995 г., будем надеяться, что на этот раз будет возможно хотя бы следить за его разработкой.

Подводя итоги этого короткого экскурса в историю стандартизации SQL, заметим, что во многом (конечно, не во всем) процесс стандартизации сводится к аккуратной технической обработке идей SQL System R, что еще раз подчеркивает уникальность этого проекта (прошло уже более 10 лет после его завершения!).

§14. Стандартный язык баз данных SQL.

В этой лекции мы коротко рассмотрим основные особенности стандарта языка SQL 1989г.

14.1. Типы данных

В языке SQL/89 поддерживаются следующие типы данных: *CHARACTER*, *NUMERIC*, *DECIMAL*, *INTEGER*, *SMALLINT*, *FLOAT*, *REAL*, *DOUBLE PRECISION*. Эти типы данных классифицируются на типы строк символов, точных чисел и приблизительных чисел.

К первому классу относится *CHARACTER*. Спецификатор типа имеет вид *CHARACTER* (lenght), где lenght задает длину строк данного типа. Заметим, что в SQL/89 нет типа строк переменного размера, хотя во многих реализациях они допускаются. Литеральные строки символов изображаются в виде 'последовательность символов' (например, 'example').

Представителями второго класса типов являются *NUMERIC*, *DECIMAL* (или *DEC*), *INTEGER* (или *INT*) и *SMALLINT*. Спецификатор типа *NUMERIC* имеет вид *NUMERIC*[(*precision* [,*scale*])]. Специфицируются точные числа, представляемые с точностью precision и масштабом scale. Здесь и далее, если опущен масштаб, то он полагается равным 0, а если опущена точность, то ее значение по умолчанию определяется в реализации.

Спецификатор типа *DECIMAL* (или *DEC*) имеет вид *NUMERIC*[(*precision* [,*scale*])]. Специфицируются точные числа, представленные с масштабом *scale* и точностью, равной или большей значения *precision*.

INTEGER специфицирует тип данных точных чисел с масштабом 0 и определяемой в реализации точностью. *SMALLINT* специфицирует тип данных точных чисел с масштабом 0 и определяемой в реализации точностью, не большей, чем точность чисел типа *INTEGER*.

Литеральные значения точных чисел в общем случае представляются в форме

```
[+|-] <целое-без-знака> [.<целое-без-знака>].
```

Наконец, в классу типов данных приблизительных чисел относятся типы *FLOAT*, *REAL* и *DOUBLE PRECISION*. Спецификатор типа *FLOAT* имеет вид *FLOAT*[(*precision*)]. Специфицируются приблизительные числа с двоичной точностью, равной или большей значения *presicion*.

REAL специфицирует тип данных приблизительных чисел с точностью, определенной в реализации. *DOUBLE PRECISION* специфицирует тип данных приблизительных чисел с точностью, определенной в реализации, большей, чем точность типа *REAL*.

Литеральные значения приблизительных чисел в общем случае представляются в виде

```
<литеральное-значение-точного-числа>.<целое-со-знаком>.
```

Заметим, что хотя с использованием языка SQL можно определить схему БД, содержащую данные любого из перечисленных типов, возможность использования этих данных в прикладных системах зависит от применяемого языка программирования. Весь набор типов данных можно использовать, только если программировать на ПЛ/1. Поэтому в некоторых реализациях SQL типы данных с масштабом и точностью вообще не поддерживаются.

Хотя правила встраивания SQL в программы на языке Си не определены в SQL/89, в большинстве реализаций, поддерживающих такое встраивание, имеется следующее соответствие между типами данных SQL и типами данных Си: *CHARACTER* соответствует строкам Си; *INTEGER* соответствует *long*; *SMALLINT* соответствует *short*; *REAL* соответствует *float*; *DOUBLE PRECISION* соответствует *double* (именно такое соответствие утверждено в стандарте SQL/92).

Заметим еще, что в большинстве реализаций SQL поддерживаются некоторые дополнительные типы данных, например, *DATE*, *TIME*, *INTERVAL*, *MONEY*. Некоторые из этих типов

специфицированы в стандарте SQL/92, но в текущих реализациях синтаксические и семантические свойства таких типов могут различаться.

14.2. Средства определения схемы.

Средства определения схемы БД в стандарте SQL/89 относятся к наиболее слабым и допускающим различную интерпретацию частям стандарта. Более того, мне неизвестна ни одна реализация, в которой поддерживался бы в точности такой набор средств определения схемы.

Поэтому, чтобы добиться мобильности прикладной системы в достаточно широком классе реализаций SQL/89, необходимо тщательно локализовать компоненты определения схемы БД. Думаю, что лучше всего сосредоточить всю работу со схемой БД в одном модуле и иметь в виду, что при переходе к другой СУБД очень вероятно потребуется переделка этого модуля.

Особо отметим, что в SQL/89 вообще отсутствуют какие-либо средства изменения схемы БД: нет возможности удалить схему таблицы, добавить к схеме таблицы новый столбец и т.д. Во всех реализациях такие средства поддерживаются, но они могут различаться и синтаксисом, и семантикой.

Несмотря на отсутствие особых надежд на то, что удастся встретить реализацию, поддерживающую язык определения схем SQL/89, мы коротко опишем этот язык (без синтаксических деталей), чтобы оценить на содержательном уровне возможности SQL/89 в этой части и получить хотя бы какие-то средства сравнения разных реализаций.

14.2.1. Оператор определения схемы.

В соответствии с правилами SQL/89 каждая таблица данной БД имеет простое и квалифицированное имена. В качестве квалификатора имени выступает "идентификатор полномочий" таблицы, который обычно в реализациях совпадает с именем некоторого пользователя, и квалифицированное имя таблицы имеет вид:

```
<идентификатор полномочий>.<простое имя>
```

Подход к определению схемы в SQL/89 состоит в том, что все таблицы с одним идентификатором полномочий создаются (определяются) путем выполнения одного оператора определения схемы. При этом в стандарте не определяется способ выполнения оператора определения схемы: должен ли он выполняться только в интерактивном режиме или может быть встроен в программу, написанную на традиционном языке программирования.

В операторе определения схемы содержится идентификатор полномочий и список элементов схемы, каждый из которых может быть определением таблицы, определением представления (view) или определением привилегий. Каждое из этих определений представляется отдельным оператором SQL/89, но все они, как уже говорилось, должны быть встроены в оператор определения схемы.

Для этих операторов мы приведем синтаксис, поскольку это позволит более четко описать их особенности.

14.2.2. Определение таблицы.

Оператор определения таблицы имеет следующий синтаксис:

```
<table definition> ::=
   CREATE TABLE <table  name> (<table   element> [{,<table element>}...])
<table element> ::=
   <column definition> | <table constraint definition>
```

Кроме имени таблицы, в операторе специфицируется 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. Оператор *CREATE TABLE* определяет так называемую базовую таблицу, т.е. реальное хранилище данных.

Для определения столбцов таблицы и ограничений целостности используются специальные операторы, которые должны быть вложены в оператор определения таблицы.

14.2.3. Определение столбца.

Оператор определения столбца описывается следующими синтаксическими правилами:

```
<column definition> ::=
   <column name> <data type>
   [<default clause>] [<column constraint>...]
<default clause> ::=
   DEFAULT { <literal> | USER | NULL }
<column constraint> ::=
   NOT NULL [<unique specification>]
   | <references specification>
   | CHECK (<search condition>)
```

Как видно, кроме обязательной части, в которой определяется имя столбца и его тип данных, определение столбца может содержать два необязательных раздела: раздел значения столбца по умолчанию и раздел ограничений целостности столбца.

В разделе значения по умолчанию указывается значение, которое должно быть помещено с строку, заносимую в данную таблицу, если значение данного столбца явно не указано. 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 попытка занести в таблицу строку с неспецифицированным значением данного столбца приведет к ошибке.

Указание в разделе ограничений целостности *NOT NULL* приводит к неявному порождению проверочного ограничения целостности для всей таблицы (см. следующий подраздел) *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.

Если в разделе ограничений целостности указано ограничение по ссылкам данного столбца (`<reference specification>`), то порождается соответствующее определение ограничения по ссылкам для таблицы:

```
FOREIGN KEY(C) <reference specification>
```

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.

14.2.4. Определение ограничений целостности таблицы.

Ограничения целостности таблицы обладают следующим синтаксисом:

```
<table constraint definition> ::=
   <unique constraint definition>
   | <referential constraint definition>
   | <check constraint definition>
<unique constraint definition> ::=
   <unique specification> (<unique column list>)
<unique specification> ::= UNIQUE | PRIMARY KEY
<unique column list> ::= <column name> [{,<column name>}...]
<referential constraint definition> ::=
   FOREIGN KEY (<referencing columns>) <references specification>
<references specification> ::= REFERENCES <referenced table and columns>
<referencing columns> ::= <reference column list>
<referenced table and columns> ::= <table name> [(<reference column list>)]
```

```
<reference column list> ::= <column name> [{,<column name>}...]
<check constraint definition> ::= CHECK (<search condition>)
```

Для одной таблицы может быть задано несколько ограничений целостности, в том числе те, которые неявно порождаются ограничениями целостности столбцов. Стандарт SQL/89 устанавливает, что ограничения таблицы фактически проверяются при выполнении каждого оператора SQL.

Замечание: Наличие правильно подобранного набора ограничений БД очень важно для надежного функционирования прикладной информационной системы. Вместе с тем, в некоторых СУБД ограничения целостности практически не поддерживаются. Поэтому при проектировании прикладной системы необходимо принять решение о том, что более существенно: рассчитывать на поддержку ограничений целостности, но ограничить набор возможных СУБД, или отказаться от их использования на уровне SQL, сохранив возможность использования не самых современных СУБД.

Далее T обозначает таблицу, для которой определяются ограничения целостности.

Ограничение уникальности.

Каждое имя столбца в списке уникальности должно именовать столбец T и не должно входить в этот список более одного раза. При определении столбца, входящего в список уникальности, должно быть указано ограничение столбца $NOT\ NULL$. Среди ограничений уникальности T не должно быть более одного определения первичного ключа (ограничения уникальности с ключевым словом $PRIMARY\ KEY$).

Действие ограничения уникальности состоит в том, что в таблице T не допускается появление двух или более строк, значения столбцов уникальности которых совпадают.

Ограничение по ссылкам.

Ограничение по ссылкам от заданного набора столбцов CT таблицы T на заданный набор столбцов $CT1$ некоторой определенной к этому моменту таблицы $T1$ определяет условие на содержимое обеих этих таблиц, при котором ссылки можно считать корректными.

Если список столбцов $CT1$ явно специфицирован в определении ограничения по ссылкам, то требуется, чтобы этот список явно входил в какое-либо определение уникальности таблицы $T1$. Если же список $CT1$ не специфицирован явно в определении ограничения по ссылкам таблицы T, то требуется, чтобы в определении таблицы $T1$ присутствовало определение первичного ключа, и список $CT1$ неявно полагается совпадающим со списком имен столбцов из определения первичного ключа таблицы $T1$. Имена столбцов списков CT и $CT1$ должны именовать столбцы таблиц T и $T1$ соответственно, и не должны появляться в списках более одного раза. Списки столбцов CT и $CT1$ должны содержать одинаковое число элементов, и столбец таблицы T, идентифицируемый i-ым элементом списка CT должен иметь тот же тип, что столбец таблицы T1, идентифицируемый i-ым элементом списка $CT1$.

По определению, таблицы T и $T1$ удовлетворяют заданному ограничению по ссылкам, если для каждой строки s таблицы T такой, что все значения столбцов s, идентифицируемых списком CT, не являются неопределенными, существует строка $s1$ таблицы $T1$ такая, что значения столбцов $s1$, идентифицируемых списком $CT1$, позиционно равны значениям столбцов s, идентифицируемых списком CT. По человечески это можно сформулировать так: ограничение по ссылкам удовлетворяется, если для каждой корректной ссылки существует объект, на который она ссылается. В привычной программистам терминологии, ограничение по ссылкам не позволяет производить "висячие" ссылки, не ведущие ни к какому объекту.

Проверочное ограничение.

Проверочное ограничение специфицирует условие, которому должна удовлетворять в отдельности каждая строка таблицы T. Это условие не должно содержать подзапросов, спецификаций агрегатных функций, а также ссылок на внешние переменные или параметров. В него могут входить только имена столбцов данной таблицы и литеральные константы.

Таблица удовлетворяет проверочному ограничению целостности в том и только в том случае, когда вычисление условия для каждой строки таблицы дает true.

Замечание: В некоторых реализациях допускаются расширенные механизмы ограничений по ссылкам и проверочных ограничений. Следует быть внимательным, если не желать выходить за пределы возможностей стандарта.

14.2.5. Определение представлений.

Механизм представлений (view) является мощным средством языка SQL, позволяющим скрыть реальную структуру БД от некоторых пользователей за счет определения представления БД, которое реально является некоторым хранимым в БД запросом с именованными столбцами, а для пользователя ничем не отличается от базовой таблицы БД (с учетом технических ограничений). Любая реализация должна гарантировать, что состояние представляемой таблицы точно соответствует состоянию базовых таблиц, на которых определено представление. Обычно вычисление представляемой таблицы (материализация соответствующего запроса) производится каждый раз при использовании представления.

В стандарте SQL/89 оператор определения представления имеет следующий синтаксис:

```
<view definition> ::=
   CREATE VIEW <table name> [(<view  column  list>)]
   AS <query specification> [WITH CHECK OPTION]
<view column list> ::= <column name> [{,<column name>}...]
```

Определяемая представляемая таблица V является изменяемой (т.е. по отношению к V можно использовать операторы *DELETE* и *UPDATE*) в том и только в том случае, если выполняются следующие условия для спецификации запроса:

- В списке выборки не указано ключевое слово *DISTINCT*;
- Каждое арифметическое выражение в списке выборки представляет собой одну спецификацию столбца, и спецификация одного столбца не появляется более одного раза;
- В разделе *FROM* указана только одна таблица, являющаяся либо базовой таблицей, либо изменяемой представляемой таблицей;
- В условии выборки раздела *WHERE* не используются подзапросы;
- В табличном выражении отсутствуют разделы *GROUP BY* и *HAVING*.

Замечание: Эти ограничения являются очень сильными. В реализациях они могут быть ослаблены. Но если стремиться к мобильности, не следует пользоваться расширенными возможностями.

Если в списке выборки спецификации запроса имеется хотя бы одно арифметическое выражение, состоящее не из одной спецификации столбца, или если одно имя столбца участвует в списке выборки более одного раза, определение представления должно содержать список имен столбцов представляемой таблицы. Более просто, нужно явно именовать столбцы представляемой таблицы, если эти имена не наследуются от столбцов таблиц раздела *FROM* спецификации запроса.

Требование *WITH CHECK OPTION* в определении представления имеет смысл только в случае определения изменяемой представляемой таблицы, которая определяется спецификацией запроса, содержащей раздел *WHERE*. При наличии этого требования не допускаются изменения представляемой таблицы, которые приводят к появлению в базовых таблиц строк, не видимых в представляемой таблице (т.е. таких строк, которые не удовлетворяют условию поиска раздела *WHERE* спецификации запроса). Если *WITH CHECK OPTION* в определении представления отсутствует, такой контроль не производится.

14.2.6. Определение привилегий.

В соответствии с идеологией языка SQL контроль прав доступа данного пользователя к таблицам БД производится на основе механизма привилегий. Фактически, этот механизм состоит в том, что для выполнения любого действия над таблицей пользователь должен обладать соответствующей привилегией (реально все возможные действия описываются фиксированным стандартным набором привилегий). Пользователь, создавший таблицу, автоматически становится владельцем всех возможных привилегий на выполнение операций над этой таблицей. В

число этих привилегий входит привилегия на передачу всех или некоторых привилегий по отношению к данной таблице другому пользователю, включая привилегию на передачу привилегий. Иногда поддерживается и обратная операция изъятия привилегий от пользователя, ранее их получившего.

В SQL/89 определяется упрощенная схема механизма привилегий. Во-первых, "раздача" привилегий возможна только при определении таблицы. Во-вторых, пользователь, получивший некоторые привилегии от других пользователей, может передать их дальше только при определении схемы.

Определение привилегий производится в следующем синтаксисе:

```
<privilege definition> ::=
   GRANT <privileges>  ON <table name> TO <grantee>
   [{,<grantee>}...] [WITH GRANT OPTION]
<privileges> ::= ALL PRIVILEGES | <action> [{,<action>}...]
<action> ::=  SELECT | INSERT | DELETE
   | UPDATE [(<grant column list>)]
   | REFERENCES [(<grant column list>]
<grant column list> ::= <column name> [{,<column name>}...]
<grantee> ::= PUBLIC | <authorization identifier>
```

Смысл механизма определения привилегий в SQL/89 достаточно понятен из этого синтаксиса. Заметим лишь, что привилегией *REFERENCES* по отношению к указанным столбцам таблицы $T1$ необходимо обладать, чтобы иметь возможность при определении таблицы T определить ограничение по ссылкам между этой таблицей и существующей к этому моменту таблицей $T1$.

Еще раз заметим, что хотя в общем смысле во всех SQL-ориентированных СУБД поддерживается механизм защиты доступа на основе привилегий, реализации могут различаться в деталях. Это опять то место, которое нужно локализовывать, если стремиться к созданию мобильной прикладной системы.

§15. Язык SQL. Средства манипулирования данными.

15.1. Структура запросов.

Для того, чтобы можно было более или менее точно рассказать про структуру запросов в стандарте SQL/89, необходимо начать со сводки синтаксических правил:

```
<cursor specification> ::= <query expression> [<order by clause>]
<query expression> ::=
   <query term> |  <query expression> UNION [ALL] <query term>
<query term> ::= <query specification> | (<query expression>)
<query specification> ::=
   (SELECT [ALL | DISTINCT] <select list> <table expression>)
<select statement> ::=
   SELECT [ALL | DISTINCT] <select list>
   INTO <select target list> <table expression>
<subquery> ::=
   (SELECT [ALL | DISTINCT] <result specification> <table expression>)
<table expression> ::=
   <from clause> [<where clause>] [<group by clause>] [<having clause>]
```

Язык допускает три типа синтаксических конструкций, начинающихся с ключевого слова *SELECT* : спецификация курсора (cursor specification), оператор выборки (select statement) и подзапрос (subquery). Основой всех них является синтаксическая конструкция "табличное выражение (table expression)". Семантика табличного выражения состоит в том, что на основе последовательного применения разделов from, where, group by и having из заданных в разделе from таблиц строится некоторая новая результирующая таблица, порядок следования строк которой

не определен и среди строк которой могут находиться дубликаты (т.е. в общем случае таблица-результат табличного выражения является мультимножеством строк). На самом деле именно структура табличного выражения наибольшим образом характеризует структуру запросов языка SQL/89. Мы рассмотрим ниже структуру и смысл разделов табличного выражения ниже, но до этого немного подробнее обсудим три упомянутые конструкции, включающие табличные выражения.

15.1.1. Спецификация курсора.

Наиболее общей является конструкция "спецификация курсора". Курсор - это понятие языка SQL, позволяющее с помощью набора специальных операторов получить построчный доступ к результату запроса к БД. К табличным выражениям, участвующим в спецификации курсора, не предъявляются какие-либо ограничения. Как видно из сводки синтаксических правил, при определении спецификации курсора используются три дополнительных конструкции: спецификация запроса, выражение запросов и раздел *ORDER BY* .

Спецификация запроса.

В спецификации запроса задается список выборки (список арифметических выражений над значениями столбцов результата табличного выражения и констант). В результате применения списка выборки к результату табличного выражения производится построение новой таблицы, содержащей то же число строк, но вообще говоря другое число столбцов, содержащих результаты вычисления соответствующих арифметических выражений из списка выборки. Кроме того, в спецификации запроса могут содержаться ключевые слова *ALL* или *DISTINCT* . При наличии ключевого слова *DISTINCT* из таблицы, полученной применением списка выборки к результату табличного выражения, удаляются строки-дубликаты; при указании *ALL* (или просто при отсутствии *DISTINCT*) удаление строк-дубликатов не производится.

Выражение запросов.

Выражение запросов - это выражение, строящееся по указанным синтаксическим правилам на основе спецификаций запросов. Единственной операцией, которую разрешается использовать в выражениях запросов, является операция *UNION* (объединение таблиц) с возможной разновидностью *UNION ALL* . К таблицам-операндам выражения запросов предъявляется то требование, что все они должны содержать одно и то же число столбцов, и соответствующие столбцы всех операндов должны быть одного и того же типа. Выражение запросов вычисляется слева направо с учетом скобок. При выполнении операции *UNION* производится обычное теоретико-множественное объединение операндов, т.е. из результирующей таблицы удаляются дубликаты. При выполнении операции *UNION ALL* образуется результирующая таблица, в которой могут содержаться строки-дубликаты.

Раздел ORDER BY.

Наконец, раздел *ORDER BY* позволяет установить желаемый порядок просмотра результата выражения запросов. Синтаксис *ORDER BY* следующий:

```
<order by clause> ::=
   ORDER BY <sort specification> [{,<sort specification>}...]
<sort specification> ::=
   {<unsigned integer> | <column specification>} [ASC | DESC]
```

Как видно из этих синтаксических правил, фактически задается список столбцов результата выражения запросов, и для каждого столбца указывается порядок просмотра строк результата в зависимости от значений этого столбца (*ASC* - по возрастанию (умолчание), *DESC* - по убыванию). Столбцы можно задавать их именами в том и только в том случае, когда (1) выражение запросов не содержит операций *UNION* или *UNION ALL* и (2) в списке выборки спецификации запроса этому столбцу соответствует арифметическое выражение, состоящее только из имени столбца. Во всех остальных случаях в разделе *ORDER BY* должен указываться порядковый номер столбца в таблице-результате выражения запросов.

15.1.2. Оператор выборки.

Оператор выборки - это отдельный оператор языка SQL/89, позволяющий получить результат запроса в прикладной программе без привлечения курсора. Поэтому оператор выборки имеет синтаксис, отличающийся от синтаксиса спецификации курсора, и при его выполнении возникают ограничения на результат табличного выражения. Фактически, и то, и другое диктуется спецификой оператора выборки как одиночного оператора SQL: при его выполнении результат должен быть помещен в переменные прикладной программы. Поэтому в операторе появляется раздел *INTO*, содержащий список переменных прикладной программы, и возникает то ограничение, что результирующая таблица должна содержать не более одной строки. Соответственно, результат базового табличного выражения должен содержать не более одной строки, если оператор выборки не содержит спецификации *DISTINCT*, и таблица, полученная применением списка выборки к результату табличного выражения, не должна содержать более одной несовпадающих строк, если спецификация *DISTINCT* задана.

Замечание: В диалекте SQL СУБД Oracle поддерживается расширенный вариант оператора выборки, результатом которого не обязательно является таблица из одной строки. Такое расширение не поддерживается ни в SQL/89, ни в SQL/92.

15.1.3. Подзапрос.

Наконец, последняя конструкция SQL/89, которая может содержать табличные выражения, - это подзапрос, т.е. запрос, который может входить в предикат условия выборки оператора SQL. В SQL/89 к подзапросам применяется то ограничение, что результирующая таблица должна содержать в точности один столбец. Поэтому в синтаксических правилах, определяющих подзапрос, вместо списка выборки указано "выражение, вычисляющее значение", т.е. арифметическое выражение. Заметим еще, что поскольку подзапрос всегда вложен в некоторый другой оператор SQL, то в качестве констант в арифметическом выражении выборки и логических выражениях разделов *WHERE* и *HAVING* можно использовать значения столбцов текущих строк таблиц, участвующих в (под)запросах более внешнего уровня. Более подробно об этом см. ниже, при описании семантики табличных выражений.

15.2. Табличное выражение.

Стандарт SQL/89 рекомендует рассматривать вычисление табличного выражения как последовательное применение разделов *FROM*, *WHERE*, *ORDER BY* и *HAVING* к таблицам, заданным в списке *FROM*. Раздел *FROM* имеет следующий синтаксис:

```
<from clause> ::= FROM <table reference> ({,<table reference>}...]
<table reference> ::= <table name> [<correlation name>]
```

15.2.1. Раздел FROM.

Результатом выполнения раздела *FROM* является расширенное декартово произведение таблиц, заданных списком таблиц раздела *FROM*. Расширенное декартово произведение (расширенное, потому что в качестве операндов и результата допускаются мультимножества) в стандарте определяется следующим образом:

"Расширенное произведение R есть мультимножество всех строк r таких, что r является конкатенацией строк из всех идентифицированных таблиц в том порядке, в котором они идентифицированы. Мощность R есть произведение мощностей идентифицированных таблиц. Порядковый номер столбца в R есть $n+s$, где n - порядковый номер порождающего столбца в именованной таблице T, а s - сумма степеней всех таблиц, идентифицированных до T в разделе *FROM*".

Как видно из синтаксиса, рядом с именем таблицы можно указывать еще одно имя "correlation name". Фактически, это некоторый синоним имени таблицы, который можно использовать в других разделах табличного выражения для ссылки на строки именно этого вхождения таблицы.

Если табличное выражение содержит только раздел *FROM* (это единственный обязательный раздел табличного выражения), то результат табличного выражения совпадает с результатом раздела *FROM*.

15.2.2. Раздел WHERE.

Если в табличном выражении присутствует раздел *WHERE*, то следующим вычисляется он. Синтаксис раздела *WHERE* следующий:

```
<where clause> ::= WHERE <search condition>
<search condition> ::=
   <boolean term> ( <search condition> OR <boolean term> )
<boolean term> ::=
   <boolean factor> ( <boolean term> AND <boolean factor> )
<boolean factor> ::= [NOT] <boolean primary>
<boolean primary> ::= <predicate> | (<search condition>)
```

Вычисление раздела *WHERE* производится по следующим правилам: Пусть *R* - результат вычисления раздела *FROM*. Тогда условие поиска применяется ко всем строкам *R*, и результатом раздела *WHERE* является таблица, состоящая из тех строк *R*, для которого результатом вычисления условия поиска является *true*. Если условие выборки включает подзапросы, то каждый подзапрос вычисляется для каждого кортежа таблицы *R* (в стандарте используется термин "effectively" в том смысле, что результат должен быть таким, как если бы каждый подзапрос действительно вычислялся заново для каждого кортежа *R*).

Заметим, что поскольку SQL/89 допускает наличие в базе данных неопределенных значений, то вычисление условия поиска производится не в булевой, а в трехзначной логике со значениями *true*, *false* и *unknown* (неизвестно). Для любого предиката известно, в каких ситуациях он может порождать значение *unknown*. Булевские операции *AND*, *OR* и *NOT* работают в трехзначной логике следующим образом:

```
true AND unknown = unknown
unknown AND true = unknown
unknown AND unknown = unknown
true OR unknown = true
unknown OR true = true
unknown OR unknown = unknown
NOT unknown = unknown
```

Среди предикатов условия поиска в соответствии с SQL/89 могут находиться следующие предикаты: предикат сравнения, предикат *between*, предикат *in*, предикат *like*, предикат *null*, предикат с квантором и предикат *exists*. Сразу заметим, что во всех реализациях SQL на эффективность выполнения запроса существенно влияет наличие в условии поиска простых предикатов сравнения (предикатов, задающих сравнение столбца таблицы с константой). Наличие таких предикатов позволяет СУБД использовать индексы при выполнении запроса, т.е. избегать полного просмотра таблицы. Хотя в принципе язык SQL позволяет пользователям не заботиться о конкретном наборе предикатов в условии выборки (лишь бы они были синтаксически и семантически правильны), при реальном использовании SQL-ориентированных СУБД такие технические детали стоит иметь в виду.

Предикат сравнения.

Синтаксис предиката сравнения определяется следующими правилами:

```
<comparison predicate> ::=
   <value expression> <comp op> {<value expression> | <subquery>}
<comp op> ::=
   = | <> | < | > | <= | >=
```

Через " <> " обозначается операция "неравенства". Арифметические выражения левой и правой частей предиката сравнения строятся по общим правилам построения арифметических выражений и могут включать в общем случае имена столбцов таблиц из раздела *FROM* и константы. Типы данных арифметических выражений должны быть сравнимыми (например,

если тип столбца a таблицы A является типом символьных строк, то предикат "$a = 5$" недопустим).

Если правый операнд операции сравнения задается подзапросом, то дополнительным ограничением является то, что мощность результата подзапроса должна быть не более единицы. Если хотя бы один из операндов операции сравнения имеет неопределенное значение, или если правый операнд является подзапросом с пустым результатом, то значение предиката сравнения равно unknown.

Заметим, что значение арифметического выражения не определено, если в его вычислении участвует хотя бы одно неопределенное значение. Еще одно важное замечание из стандарта SQL/89: в контексте *GROUP BY*, *DISTINCT* и *ORDER BY* неопределенное значение выступает как специальный вид определенного значения, т.е. возможно, например, образование группы строк, значение указанного столбца которых является неопределенным. Для обеспечения переносимости прикладных программ нужно внимательно оценивать специфику работы с неопределенными значениями в конкретной СУБД.

Предикат between.

Предикат *between* имеет следующий синтаксис:

```
<between predicate> ::=
   <value expression>
   [NOT] BETWEEN <value expression> AND <value expression>
```

Результат "x *BETWEEN* y *AND* z" тот же самый, что результат "$x \geq y$ *AND* $x \leq z$". Результат "x *NOT BETWEEN* y *AND* z" тот же самый, что результат "*NOT* (x *BETWEEN* y *AND* z)".

Предикат in.

Предикат *in* определяется следующими синтаксическими правилами:

```
<in predicate> ::=
   <value expression> [NOT] IN {<subquery> | (<in value list>)}
<in value list> ::=
   <value specification> {,<value specification>}...
```

Типы левого операнда и значений из списка правого операнда (напомним, что результирующая таблица подзапроса должна содержать ровно один столбец) должны быть сравнимыми.

Значение предиката равно *true* в том и только в том случае, когда значение левого операнда совпадает хотя бы с одним значением списка правого операнда. Если список правого операнда пуст (так может быть, если правый операнд задается подзапросом), или значение "подразумеваемого" предиката сравнения $x = y$ (где x - значение арифметического выражения левого операнда) равно *false* для каждого элемента y списка правого операнда, то значение предиката *in* равно *false*. В противном случае значение предиката *in* равно *unknown*. По определению значение предиката "x *NOT IN* S" равно значению предиката "*NOT* (x *IN* S)".

Предикат like.

Предикат *like* имеет следующий синтаксис:

```
<like predicate> ::=
   <column specification> [NOT] LIKE <pattern> [ESCAPE <escape character>]
<pattern> ::=
   <value specification> <escape character> ::= <value specification>
```

Типы данных столбца левого операнда и образца должны быть типами символьных строк. В разделе *ESCAPE* должен специфицироваться одиночный символ.

Значение предиката равно *true*, если *pattern* является подстрокой заданного столбца. При этом, если раздел *ESCAPE* отсутствует, то при сопоставлении шаблона со строкой производится специальная интерпретация двух символов шаблона: символ подчеркивания ("_")

обозначает любой одиночный символ; символ процента ("%") обозначает последовательность произвольных символов произвольной длины (может быть, нулевой).

Если же раздел *ESCAPE* присутствует и специфицирует некоторый одиночный символ x, то пары символов "$x\_$" и "$x\%$" представляют одиночные символы "_" и "%" соответственно.

Значение предиката *like* есть *unknown*, если значение столбца, либо шаблона не определено.

Значение предиката "x *NOT LIKE* y *ESCAPE* z" совпадает со значением "*NOT* x *LIKE* y *ESCAPE* z".

Предикат null.

Предикат *null* описывается синтаксическим правилом:

```
<null predicate> ::= <column specification> IS [NOT] NULL
```

Этот предикат всегда принимает значения *true* или *false*. При этом значение "x *IS NULL*" равно *true* тогда и только тогда, когда значение x не определено. Значение предиката "x *NOT IS NULL*" равно значению "*NOT* x *IS NULL*".

Предикат с квантором.

Предикат с квантором имеет следующий синтаксис:

```
<quantified predicate> ::=
   <value expression> <comp op> <quantifier> <subquery>
<quantifier> ::= <all> | <some>
<all> ::= ALL
<some> ::= SOME | ANY
```

Обозначим через x результат вычисления арифметического выражения левой части предиката, а через S результат вычисления подзапроса.

Предикат "$x < comp\_op > ALL\ S$" имеет значение *true*, если S пусто или значение предиката "$x < comp\_op > s$" равно *true* для каждого s, входящего в S. Предикат "$x < comp\_op > ALL\ S$" имеет значение *false*, если значение предиката "$x < comp\_op > s$" равно *false* хотя бы для одного s, входящего в S. В остальных случаях значение предиката "$x < comp\_op > ALL\ S$" равно *unknown*.

Предикат "$x < comp\_op > SOME\ S$" имеет значение *false*, если S пусто или значение предиката "$x < comp\_op > s$" равно *false* для каждого s, входящего в S. Предикат "$x < comp\_op > SOME\ S$" имеет значение *true*, если значение предиката "$x < comp\_op > s$" равно true хотя бы для одного s, входящего в S. В остальных случаях значение предиката "$x < comp\_op > SOME\ S$" равно *unknown*.

Предикат exists.

Предикат *exists* имеет следующий синтаксис:

```
<exists predicate> ::= EXISTS <subquery>
```

Значением этого предиката всегда является *true* или *false*, и это значение равно true тогда и только тогда, когда результат вычисления подзапроса не пуст.

15.2.3. Раздел GROUP BY.

Если в табличном выражении присутствует раздел *GROUP BY*, то следующим выполняется он. Синтаксис раздела *GROUP BY* следующий:

```
<group by clause> ::=
   GROUP BY <column specification> [{,<column specification>}...]
```

Если обозначить через R таблицу, являющуюся результатом предыдущего раздела (*FROM* или *WHERE*), то результатом раздела *GROUP BY* является разбиение R на множество групп

строк, состоящего из минимального числа групп таких, что для каждого столбца из списка столбцов раздела *GROUP BY* во всех строках каждой группы, включающей более одной строки, значения этого столбца равны. Для обозначения результата раздела *GROUP BY* в стандарте используется термин "сгруппированная таблица".

15.2.4. Раздел HAVING.

Наконец, последним при вычислении табличного выражения используется раздел *HAVING* (если он присутствует). Синтаксис этого раздела следующий:

```
<having clause> ::= HAVING <search condition>
```

Раздел *HAVING* может осмысленно появиться в табличном выражении только в том случае, когда в нем присутствует раздел *GROUP BY*. Условие поиска этого раздела задает условие на группу строк сгруппированной таблицы. Формально раздел *HAVING* может присутствовать и в табличном выражении, не содержащем *GROUP BY*. В этом случае полагается, что результат вычисления предыдущих разделов представляет собой сгруппированную таблицу, состоящую из одной группы без выделенных столбцов группирования.

Условие поиска раздела *HAVING* строится по тем же синтаксическим правилам, что и условие поиска раздела *WHERE*, и может включать те же самые предикаты. Однако имеются специальные синтаксические ограничения по части использования в условии поиска спецификаций столбцов таблиц из раздела *FROM* данного табличного выражения. Эти ограничения следуют из того, что условие поиска раздела *HAVING* задает условие на целую группу, а не на индивидуальные строки.

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.

15.3. Агрегатные функции и результаты запросов.

Агрегатные функции (в стандарте SQL/89 они называются функциями над множествами) определяются в SQL/89 следующими синтаксическими правилами:

```
<set function specification> ::=
   COUNT(*) | <distinct set function> | <all set function>
<distinct set function> ::=
   { AVG | MAX | MIN | SUM | COUNT } ( DISTNICT <column specification> )
<all set function> ::=
   { AVG | MAX | MIN | SUM } ( [ALL] <value expression> )
```

Как видно из этих правил, в стандарте SQL/89 определены пять стандартных агрегатных функций: *COUNT* - число строк или значений, *MAX* - максимальное значение, *MIN* - минимальное значение, *SUM* - суммарное значение и *AVG* - среднее значение.

15.3.1. Семантика агрегатных функций.

Агрегатные функции предназначены для того, чтобы вычислять некоторое значение для заданного множества строк. Таким множеством строк может быть группа строк, если агрегатная

функция применяется к сгруппированной таблице, или вся таблица. Для всех агрегатных функций, кроме *COUNT*(*) , фактический (т.е. требуемый семантикой) порядок вычислений следующий: на основании параметров агрегатной функции из заданного множества строк производится список значений. Затем по этому списку значений производится вычисление функции. Если список оказался пустым, то значение функции *COUNT* для него есть 0, а значение всех остальных функций - *null* .

Пусть *T* обозначает тип значений из этого списка. Тогда результат вычисления функции *COUNT* - точное число с масштабом и точностью, определяемыми в реализации. Тип результата значений функций *MAX* и *MIN* совпадает с *T* . При вычислении функций *SUM* и *AVG* тип *T* не должен быть типом символьных строк, а тип результата функции - это тип точных чисел с определяемыми в реализации масштабом и точностью, если *T* - тип точных чисел, и тип приблизительных чисел с определяемой в реализации точностью, если *T* - тип приблизительных чисел.

Вычисление функции *COUNT*(*) производится путем подсчета числа строк в заданном множестве. Все строки считаются различными, даже если они состоят из одного столбца со значением null во всех строках.

Если агрегатная функция специфицирована с ключевым словом *DISTINCT* , то список значений строится из значений указанного столбца. (Подчеркнем, что в этом случае не допускается вычисление арифметических выражений!) Далее из этого списка удаляются неопределенные значения, и в нем устраняются значения-дубликаты. Затем вычисляется указанная функция.

Если агрегатная функция специфицирована без ключевого слова *DISTINCT* (или с ключевым словом *ALL*), то список значений формируется из значений арифметического выражения, вычисляемого для каждой строки заданного множества. Далее из списка удаляются неопределенные значения, и производится вычисление агрегатной функции. Обратите внимание, что в этом случае не допускается применение функции *COUNT* !

Замечание: оба ограничения, указанные в двух предыдущих абзацах, являются более техническими, чем принципиальными, и могут отсутствовать в конкретных реализациях. Тем не менее, это ограничения стандарта SQL/89, и их нужно придерживаться при мобильном программировании.

15.3.2. Результаты запросов.

Агрегатные функции можно разумно использовать в спецификации курсора, операторе выборки и подзапросе после ключевого слова *SELECT* (будем называть в этом подразделе все такие конструкции списком выборки, не забывая о том, что в случае подзапроса этот список состоит только из одного элемента), и в условии выборки раздела *HAVING* . Стандарт допускает более экзотические использования агрегатных функций в подзапросах (агрегатная функция на группе кортежей внешнего запроса), но на практике они встречаются очень редко.

Рассмотрим различные случаи применения агрегатных функций в списке выборки в зависимости от вида табличного выражения.

Если результат табличного выражения *R* не является сгруппированной таблицей, то появление хотя бы одной агрегатной функции от множества строк *R* в списке выборки приводит к тому, что *R* неявно рассматривается как сгруппированная таблица, состоящая из одной (или нуля) групп с отсутствующими столбцами группирования. Поэтому в этом случае в списке выборки не допускается прямое использование спецификаций строк *R* : все они должны находиться внутри спецификаций агрегатных функций. Результатом запроса является таблица, состоящая не более чем из одной строки, полученной путем применения агрегатных функций к *R* .

Аналогично обстоит дело в том случае, когда *R* представляет собой сгруппированную таблицу, но табличное выражение не содержит раздела *GROUP BY* (и, следовательно, содержит раздел *HAVING*). Если в случае предыдущего абзаца было два варианта формирования списка выборки: только с прямым указанием столбцов *R* или только с указанием их внутри спецификаций агрегатных функций, то в данном случае возможен только второй вариант. Результат табличного выражения явно объявлен сгруппированной таблицей, состоящей из одной группы, и

результат запроса можно формировать только путем применения агрегатных функций к этой группе строк. Опять результатом запроса является таблица, состоящая не более чем из одной строки, полученной путем применения агрегатных функций к R.

Наконец, рассмотрим случай, когда R представляет собой "настоящую" сгруппированную таблицу, т.е. табличное выражение содержит раздел $GROUP\ BY$ и, следовательно, определен по крайней мере один столбец группирования. В этом случае правила формирования списка выборки полностью соответствуют правилам формирования условия выборки раздела $HAVING$: допускает прямое использование спецификации столбцов группирования, а спецификации остальных столбцов R могут появляться только внутри спецификаций агрегатных функций. Результатом запроса является таблица, число строк в которой равно числу групп в R, и каждая строка формируется на основе значений столбцов группирования и агрегатных функций для данной группы.

§16. Использование SQL при прикладном программировании.

16.1. Язык модулей или встроенный SQL?

В стандарте SQL/89 определены два способа взаимодействия с БД из прикладной программы, написанной на традиционном языке программирования (как мы уже упоминали, SQL/89 ориентирован на использование совместно с языками Кобол, Фортран, Паскаль и ПЛ/1, но в реализациях обычно поддерживается и язык Си). Первый способ состоит в том, что все операторы SQL, с которыми может работать данная прикладная программа, собраны в один модуль и оформлены как процедуры этого модуля. Для этого SQL/89 содержит специальный подъязык - язык модулей. При использовании такого способа взаимодействия с БД прикладная программа содержит вызовы процедур модуля SQL с передачей им фактических параметров и получением ответных параметров.

Второй способ состоит в использовании так называемого встроенного SQL, когда с использованием специального синтаксиса в программу на традиционном языке программирования встраиваются операторы SQL. В этом случае с точки зрения прикладной программы оператор SQL выполняется "по месту". Явная параметризация операторов SQL отсутствует, но во встроенных операторах SQL могут использоваться имена переменных основной программы, и за счет этого обеспечивается связь между прикладной программой и СУБД.

Концептуально эти два способа эквивалентны. Более того, в стандарте устанавливаются правила порождения неявного модуля SQL по программе со встроенным SQL. Однако в большинстве реализаций операторы SQL, содержащиеся в модуле SQL, и встроенные операторы SQL обрабатываются существенно по-разному. Модуль SQL обычно компилируется отдельно от прикладной программы, в результате чего порождается набор так называемых хранимых процедур (в стандарте этот термин не используется, но распространен в коммерческих реализациях). Т.е. в случае использования модуля SQL компиляция операторов SQL производится один раз, и затем соответствующие процедуры сколько угодно раз могут вызываться из прикладной программы.

В отличие от этого, для операторов SQL, встроенных в прикладную программу, компиляция этих операторов обычно производится каждый раз при их использовании (правильнее сказать, при каждом первом использовании оператора при данном запуске прикладной программы).

Конечно, пользователи не обязаны знать об этом техническом различии в обработке двух видов взаимодействия с СУБД. Существуют и такие системы, которые производят одноразовую компиляцию встроенных операторов SQL и сохраняют откомпилированный код. Но все-таки лучше иметь это в виду.

Приведем некоторые соображения за и против каждого из этих двух способов. При использовании языка модулей текст прикладной программы имеет меньший размер, взаимодействия с СУБД более локализованы за счет наличия явных параметров вызова процедур. С другой стороны, для понимания смысла поведения прикладной программы потребуется одновременное чте-

ние двух текстов. Кроме того, как кажется, синтаксис модуля SQL может существенно различаться в разных реализациях. Встроенный SQL предоставляет возможность производства более "самосодержащихся" прикладных программ. Имеется больше оснований рассчитывать на простоту переноса такой программы в среду другой СУБД, поскольку стандарт встраивания более или менее соблюдается. Основным недостатком является некоторый PL-подобный вид таких программ, независимо от выбранного основного языка. И конечно, нужно учитывать замечания, содержащиеся в предыдущих абзацах.

Далее мы коротко опишем язык модулей и правила встраивания в соответствии со стандартом SQL/89 (еще раз заметим, что формально правила встраивания не являются частью стандарта).

16.2. Язык модулей.

Структура модуля SQL в стандарте SQL/89 определяется следующими синтаксическими правилами:

```
<module> ::=
   <module name clause>
   <language clause>
   <module autorization clause>
   [<declare cursor>...]
   < procedure >  ...
<module name clause> ::= MODULE [<module name>]
<language clause> ::= LANGUAGE { COBOL | FORTRAN | PASCAL | PLI }
<module autorization clause> ::=
   AUTHORIZATION <module autorization identifier>
<module autorization identifier> ::= <autorization identifier>
```

Существенно, что каждый модуль SQL ориентирован на использование в программах, написанных на конкретном языке программирования. Если в модуле присутствуют процедуры работы с курсорами, то все курсоры должны быть специфицированы в начале модуля. Заметим, что объявление курсора не погружается в какую-либо процедуру, поскольку это описательный, а не выполняемый оператор SQL.

16.2.1. Определение процедуры.

Процедуры в модуле SQL определяются следующими синтаксическими конструкциями:

```
<procedure> ::=
   PROCEDURE <procedure name>
   <parameter declaration>...;
   <SQL statment>;
<parameter declaration>::= <parameter name> <data type> | <SQLCODE parameter>
<SQLCODE parameter> ::= SQLCODE
<SQL statement> ::=
   <close statement>
   | <commit statement>
   | <delete statement positioned>
   | <delete statement searched>
   | <fetch statement>
   | <insert statement>
   | <open statement>
   | <rollback statement>
   | <select statement>
   | <update statement positioned>
   | <update statement searched>
```

Имена всех процедур в одном модуле должны быть различны. Любое имя параметра, содержащегося в операторе SQL процедуры, должно быть специфицировано в разделе объявления параметров. Число фактических параметров при вызове процедуры должно совпадать с числом формальных параметров, указанных при ее объявлении. Список формальных параметров каждой процедуры должен содержать ровно один параметр *SQLCODE* (код ответа процедуры; воз-

можные значения кодов ответа стандартизованы, но некоторые из них определяются в реализации).

16.3. Встроенный SQL.

Поскольку в стандарте SQL/89 не специфицированы (даже в приложениях) правила встраивания SQL в язык Си, мы приведем только общие синтаксические правила встраивания, используемые для любого языка. Это поможет оценить "степень стандартности" конкретной реализации.

```
<embedded SQL statement> ::= <SQL prefix>
   { <declare cursor>
   | <embedded exception declaration>
   | <SQL statement>}
   [<SQL terminator>]
<SQL prefix> ::= EXEC SQL
<SQL terminator> ::= END EXEC | ;
<embedded SQL declare section> ::=
   <embedded SQL begin declare>
   (<host variable definition>...]
   <embedded SQL end declare>
<embedded SQL begin declare> ::=
   <SQL prefix> BEGIN DECLARE SECTION [<SQL terminator>]
<embedded SQL end declare> ::=
   <SQL prefix> END DECLARE SECTION [<SQL terminator>]
<embedded variable name> ::= :<host identifier>
<embedded exception declaration> ::=
   WHENEVER <condition> <exception action>
<condition> ::= SQLERROR | NOT FOUND
<exception action> ::= CONTINUE | <go to>
<go to> ::= { GOTO | GO TO } <target>
<target> ::= <host identifier> | <unsigned integer>
```

Встраиваемые операторы SQL, включая объявления курсора, а также разделы объявления исключительных ситуаций и переменных основной программы, должны быть окружены скобками *EXEC SQL* и *END EXEC* . Объявление курсора должно встречаться текстуально раньше любого оператора, ссылающегося на этот курсор. Все переменные основной программы, используемые во встроенных операторах SQL, должны быть объявлены в текстуально предшествующем этому оператору разделе объявления переменных основной программы. При этом синтаксис объявления переменной соответствует синтаксису основного языка программирования, но имени переменной предшествует двоеточие.

Механизм обработки исключительных ситуаций в SQL/89 исключительно прост (можно сказать, примитивен). Можно задавать реакцию на возникновение двух видов условий: *SQLERROR* - это условие появления в переменной *SQLCODE* после выполнения встроенного оператора отрицательного значения; *NOT FOUND* - условие появления в *SQLCODE* значения +100 (этот код означает исчерпание курсора). Реакция может состоять в выполнении безусловного перехода на метку основной программы (действие *GO TO*), или отсутствовать (действие *CONTINUE*). Срабатывает тот оператор определения реакции на исключительную ситуацию, который текстуально ближе от начала программы к данному оператору SQL.

Заметим, что во многих реализациях поддерживается два вида кодов ответа при выполнении операторов SQL (встроенных или взятых из модуля): через переменную *SQLCODE* с кодами ответа, представляемыми целыми числами и через переменную *SQLSTATE* с кодами ответа, кодируемыми десятичными числами, представленными в текстовой форме. Имеется тенденция к переходу на использование только механизма *SQLSTATE* , но в стандартных реализациях должен поддерживаться механизм *SQLCODE* .

16.4. Набор операторов манипулирования данными.

В стандарте SQL/89 определен очень ограниченный набор операторов манипулирования данными. Их можно классифицировать на группы операторов, связанных с курсором; одиночных операторов манипулирования данными; и операторов завершения транзакции. Все эти операторы можно использовать как в модулях SQL, так и во встроенном SQL. Заметим, что в SQL/89 не определен набор операторов интерактивного SQL.

16.4.1. Операторы, связанные с курсором.

Операторы этой группы объединяет то, что все они работают с некоторым курсором, объявление которого должно содержаться в том же модуле или программе со встроенным SQL.

Оператор объявления курсора.

Для удобства мы повторим здесь синтаксические правила объявления курсора, приводившиеся раньше:

```
<declare cursor> ::=
   DECLARE <cursor name> CURSOR FOR <cursor specification>
<cursor specification> ::= <query expression> [<order by clause>...]
<query expression> ::=
   <query term> | <query expression> UNION [ALL] <query term>
<query term> ::= <query specification> | ( <query expression> )
<order by clause> ::=
   ORDER BY <sort specification> [{,<sort specification>}...]
<sort specification> ::=
   { <unsigned integer> | <column specification> } [ASC | DESC]
```

В объявлении курсора могут задаваться запросы наиболее общего вида с возможностью выполнения операции *UNION* и сортировкой конечного результата. Этот оператор не является выполняемым, он только связывает имя курсора со спецификацией курсора.

Оператор открытия курсора.

Оператор описывается следующим синтаксическим правилом:

```
<open statement> ::= OPEN <cursor name>
```

В реализациях встроенного SQL обычно требуется, чтобы объявление курсора текстуально предшествовало оператору открытия курсора. Оператор открытия курсора должен быть первым в серии выполняемых операторов, связанных с заданным курсором. При выполнении этого оператора производится подготовка курсора к работе над ним. В частности, в этот момент производится связывание спецификации курсора со значениями переменных основного языка в случае встроенного SQL или параметров в случае модуля.

В большинстве реализаций в случае встроенного SQL именно выполнение оператора открытия курсора приводит к компиляции спецификации курсора.

Следующие операторы можно выполнять в произвольном порядке над открытым курсором.

Оператор чтения очередной строки курсора.

Синтаксис оператора чтения следующий:

```
<fetch statement> ::= FETCH <cursor name> INTO <fetch target list>
<fetch target list> ::= <target specification>[{,<target specification>}...]
```

В операторе чтения указывается имя курсора и обязательный раздел *INTO*, содержащий список спецификаций назначения (список имен переменных основной программы в случае встроенного SQL или имен "выходных" параметров в случае модуля SQL). Число и типы данных в списке назначений должны совпадать с числом и типами данных списка выборки спецификации курсора.

Любой открытый курсор всегда имеет позицию: он может быть установлен перед некоторой строкой результирующей таблицы (перед первой строкой сразу после открытия курсора), на некоторую строку результата или за последней строкой результата.

Если таблица, на которую указывает курсор, является пустой, или курсор позиционирован на последнюю строку или за ней, то при выполнении оператора чтения курсор устанавливается в

позицию после последней строки, параметру *SQLCODE* присваивается значение 100, никакие значения не присваиваются целям, идентифицированным в разделе *INTO*.

Если курсор установлен в позицию перед строкой, то он устанавливается на эту строку, и значения этой строки присваиваются соответствующим целям.

Если курсор установлен на строку r, отличную от последней строки, то курсор устанавливается на строку, непосредственно следующую за строкой r, и значения из этой следующей строки присваиваются соответствующим целям.

Возникает естественный вопрос, каким образом можно параметризовать курсор неопределенным значением или узнать, что выбранное из очередной строки значение является неопределенным. В SQL/89 это достигается за счет использования так называемых индикаторных параметров и переменных. Если известно, что значение, передаваемое из основной программы СУБД или принимаемое основной программой от СУБД, может быть неопределенным, и этот факт интересует прикладного программиста, то спецификация параметра или переменной в операторе SQL имеет вид:

```
<parameter name>[INDICATOR]<parameter name>
```

при спецификации параметра, и

```
<embedded variable name>[INDICATOR]<embedded variable name>
```

при спецификации переменной. Отрицательное значение индикаторного параметра или индикаторной переменной (они должны быть целого типа) соответствует неопределенному значению параметра или переменной.

Оператор позиционного удаления.

Синтаксис этого оператора следующий:

```
<delete statement: positioned> ::=
   DELETE FROM <table name> WHERE CURRENT OF <cursor name>
```

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.

Оператор позиционной модификации.

Оператор описывается следующими синтаксическими правилами:

```
<update statement: positioned> ::=
   UPDATE <table name>
   SET <set clause:positioned>
   [{,<set clause:positioned>}...]
   WHERE CURRENT OF <cursor name>
<set clause: positioned> ::=
   <object column: positioned> = { <value expression> | NULL }
<object column: positioned> ::= <column name>
```

Если указанный в операторе курсор открыт и установлен на некоторую строку, и курсор определяет изменяемую таблицу, то текущая строка курсора модифицируется в соответствии с разделом *SET*. Позиция курсора не изменяется. Таблица, указанная в разделе *FROM* оператора *DELETE*, должна быть таблицей, указанной в самом внешнем разделе *FROM* спецификации курсора.

Оператор закрытия курсора.

Синтаксис этого оператора следующий:

```
<close statement> ::= CLOSE <cursor name>
```

Если к моменту выполнения этого оператора курсор находился в открытом состоянии, то оператор переводит курсор в закрытое состояние. После этого над курсором возможно выполнение только оператора *OPEN*.

16.4.2. Одиночные операторы манипулирования данными.

Каждый из операторов этой группы является абсолютно независимым от какого бы то ни было другого оператора.

Оператор выборки.

Для удобства мы повторяем синтаксис этого оператора еще раз:

```
<select statement> ::=
   SELECT [ALL | DISTINCT] <select name>
   INTO <select target list> <table expression>
<select target list>::=
   <target specification>
   [{,<target specification>}...]
```

Поскольку, как мы уже объясняли, результатом одиночного оператора выборки является таблица, состоящая не более, чем из одной строки, список целей специфицируется в самом операторе.

Оператор поискового удаления.

Оператор описывается следующим синтаксическим правилом:

```
<delete statement: searched> ::=
   DELETE FROM <table name> WHERE [<search condition>]
```

Таблица *T*, указанная в разделе *FROM* оператора *DELETE*, должна быть изменяемой. На условие поиска накладывается то условие, что на таблицу *T* не должны содержаться ссылки ни в каком вложенном подзапросе предикатов раздела *WHERE*.

Фактически оператор выполняется следующим образом: последовательно просматриваются все строки таблицы *T*, и те строки, для которых результатом вычисления условия выборки является true, удаляются из таблицы *T*. При отсутствии раздела *WHERE* удаляются все строки таблицы *T*.

Оператор поисковой модификации.

Оператор обладает следующим синтаксисом:

```
<update statement: searched> ::=
   UPDATE <table name> SET <set clause: searched>
   [{,<set clause: searched>}...] [WHERE  <search conditions>]
<set clause: searched> ::=
   <object column: searched> = { <value expression> | NULL }
<object column: searched> ::= <column name>
```

Таблица *T*, указанная в операторе *UPDATE*, должна быть изменяемой. На условие поиска накладывается то условие, что на таблицу *T* не должны содержаться ссылки ни в каком вложенном подзапросе предикатов раздела *WHERE*.

Оператор фактически выполняется следующим образом: таблица T последовательно просматривается, и каждая строка, для которой результатом вычисления условия поиска является true, изменяется в соответствии с разделом *SET*. Если арифметическое выражение в разделе *SET* содержит ссылки на столбцы таблицы *T*, то при вычислении арифметического выражения используются значения столбцов текущей строки до их модификации.

Операторы окончания транзакции.

Текущая транзакция может быть завершена успешно (с фиксацией в базе данных произведенных изменений) путем выполнения оператора *COMMIT WORK* или аварийно (с удалением из базы данных изменений, произведенных текущей транзакцией) путем выполнения оператора

ROLLBACK WORK . При выполнении любого из этих операторов производится принудительное закрытие всех курсоров, открытых к моменту выполнения оператора завершения транзакции.

16.5. Динамический SQL в Oracle V.6.

Описанный в стандарте SQL/89 набор операторов SQL предназначен для встраивания в программу на обычном языке программирования. Поэтому в этом наборе перемешаны операторы "истинного" реляционного языка запросов (например, оператор удаления из таблицы части строк, удовлетворяющих заданному значению) и операторы работы с курсорами, позволяющими обеспечить построчный доступ к таблице-результату запроса.

Понятно, что в диалоговом режиме набор операторов SQL и их синтаксис должен быть несколько другим. Весь вопрос состоит в том, как реализовывать такую диалоговую программу. Правила встраивания стандартного SQL в программу на обычном языке программирования предусматривают, что вся информация, касающаяся операторов SQL, известна в статике (за исключением значений переменных, используемых в качестве констант в операторах SQL). Не предусмотрены стандартные средства компиляции с последующим выполнением операторов, которые становятся известными только во время выполнения (например, вводятся с терминала). Поэтому, опираясь только на стандарт, невозможно реализовать диалоговый монитор взаимодействия с БД на языке SQL или другую прикладную программу, в которой текст операторов SQL возникает во время выполнения, т.е. фактически так или иначе стандарт необходимо расширять.

Один из возможных путей расширения состоит в использовании специальной группы операторов, обеспечивающих динамическую компиляцию (во время выполнения прикладной программы) базового подмножества операторов SQL и поддерживающих их корректное выполнение. Некоторый набор таких операторов входил в диалект SQL, реализованный в System R, несколько отличный набор входит в реализацию Oracle V.6 и наконец, в новом стандарте SQL/92 появилась стандартная версия динамического SQL.

Поскольку в СУБД Oracle средства динамического SQL реализованы уже сравнительно давно, имеет смысл рассмотреть сначала их, чтобы иметь основу для сравнения с SQL/92.

В дополнительный набор операторов, поддерживающих динамическую компиляцию базовых операторов SQL, входят операторы: *PREPARE* , *DESCRIBE* и *EXECUTE* .

16.5.1. Оператор подготовки.

Оператор *PREPARE* имеет синтаксис:

```
<prepare-statement> ::=
   PREPARE <statement-name> FROM <host-string-variable>
<statement-name> ::= <name>
```

Во время выполнения оператора *PREPARE* символьная строка, содержащаяся в *host – string – variable* , передается компилятору SQL, который обрабатывает ее почти таким же образом, как если бы получил в статике. Построенный при выполнении оператора *PREPARE* код остается действующим до конца транзакции или до повторного выполнения данного оператора *PREPARE* в пределах этой же транзакции.

В отличие от статически подставляемых в программу на обычном языке программирования операторов SQL, в которых связь с переменными включающей программы производится по именам (т.е. в соответствии со стандартом во встроенном операторе SQL могут употребляться просто имена переменных включающей программы), динамическая природа операторов, подготавливаемых с помощью оператора *PREPARE* , заставляет рассматривать эти имена как имена формальных параметров. Соответствие этих формальных параметров адресам переменных включающей программы устанавливается позиционно во время выполнения подготовленного оператора.

16.5.2. Оператор получения описания подготовленного оператора.

Оператор *DESCRIBE* предназначен для того, чтобы определить тип ранее подготовленного оператора, узнать количество и типы формальных параметров (если они есть) и количество и

типы столбцов результирующей таблицы, если подготовленный оператор является оператором выборки (*SELECT*).

Действие оператора *DESCRIBE* состоит в том, что в указанную область памяти прикладной программы (структура этой области фиксирована и известна пользователям) помещается информация, характеризующая ранее подготовленный оператор с заданным именем.

16.5.3. Оператор выполнения подготовленного оператора.

Оператор *EXECUTE* служит для выполнения ранее подготовленного оператора SQL типа 'N' (не требующего применения курсора) или для совмещенной подготовки и выполнения такого оператора. Синтаксис оператора *EXECUTE* :

```
<execute-statement> ::=
   EXECUTE
   {<statement-name> [USING <host-vars-list>]
   ( IMMEDIATE <host-string-variable> ) }
```

Для выполнения подготовленного оператора служит первый вариант оператора *EXECUTE* . В этом случае $< statement - name >$ должен задавать имя, употреблявшееся ранее в операторе *PREPARE* . Если в подготовленном операторе присутствуют формальные параметры, то в операторе *EXECUTE* должен задаваться список фактических параметров $< host - vars - list >$. Число и типы фактических параметров должны соответствовать числу и типам формальных параметров подготовленного оператора.

Второй вариант оператора *EXECUTE* предназначен в Oracle для совмещенной подготовки и выполнения оператора SQL типа 'N'. В этом случае параметром оператора *EXECUTE* является строка, которая должна содержать текст оператора SQL (эту строку разрешается также задавать литерально). Запрещается использование в этом операторе переменных включающей программы (формальных параметров).

16.5.4. Работа с динамическими операторами SQL через курсоры.

Для использования таких операторов используется расширение механизма курсоров стандарта SQL. Во-первых, при определении курсора можно указывать не только литеральную спецификацию курсора, но и имя оператора, вводимое с помощью оператора *PREPARE* (в этом случае оператор *PREPARE* должен текстуально находиться выше оператора *DECLARE*). Тем самым полный синтаксис оператора *DECLARE* становится следующим:

```
<declare cursor> ::=
   DECLARE <cursor name> CURSOR
   FOR { <cursor specification> | <statement-name> }
```

Далее, поскольку для такого курсора в статике неизвестна информация о входных и выходных переменных включающей программы, то используется другая форма операторов *OPEN* и *FETCH* .

Полный синтаксис этих операторов становится следующим:

```
<open statement> ::=
   OPEN <cursor name>
   [USING { <host-vars-list> | DESCRIPTOR <descr-name> }]
<fetch statement> ::=
   FETCH <cursor name>
   { INTO <fetch target list>
   ( USING <host-vars-list> )
   ( USING DESCRIPTOR <descr-name> )}
```

Как видно, предлагается два способа задания фактических входных и выходных параметров: прямое с указанием в операторах *OPEN* и/или *FETCH* списков имен переменных включающей программы и косвенное, когда число параметров и их адреса сообщаются через дополнительную структуру-дескриптор.

Первый способ предлагается использовать для работы с операторами выборки, для которых фиксирован набор формальных входных и выходных параметров. Точнее говоря, что касается выходных параметров, должны быть фиксированы число и типы элементов списка выборки.

Второй способ работы с динамически откомпилированными операторами, требующими использования курсоров, состоит в использовании дескрипторов динамически формируемых списков параметров. В этом случае вся ответственность за соответствие типов фактических и формальных параметров ложится на программиста. В результате ошибки при формировании такого списка, в частности, может быть испорчена память Си-программы.

§17. Некоторые черты SQL/92 и SQL-3.

Мы не будем даже поверхностно описывать новые возможности языка SQL в стандарте SQL/92. Это сейчас не очень осмысленно, поскольку единственной доступной реализацией SQL/92 является дорогостоящая версия Oracle V.7. Однако кажется полезным включить в наше руководство сводку операторов динамического SQL с небольшими комментариями, поскольку в SQL/92 предпринята первая попытка стандартизовать эту часть языка SQL, и это описание можно использовать хотя бы в качестве эталона при сравнении различных реализаций. В конце лекции приводится краткая сводка новых возможностей, ожидаемых в новом стандарте SQL-3, работа над которым все еще продолжается.

17.1. Оператор выделения памяти под дескриптор.

```
<allocate descriptor statement> ::=
   ALLOCATE DESCRIPTOR <descriptor name>
   [WITH MAX <occurrences>]
<occurences> ::= <simple value specification>
<descriptor name> ::= (<scope option>) <simple value specification>
  <scope option> ::= GLOBAL | LOCAL
<simple value specification> ::=
   <parameter name> (<embedded variable name>) (<literal>)
```

Комментарий: Дескриптор, это динамически выделяемая часть памяти прикладной программы, служащая для принятия информации о результате или параметрах динамически подготовленного оператора SQL или задания параметров такого оператора. Смысл того, что для выделения памяти используется оператор SQL, а не просто стандартная функция *alloc* или какая-нибудь другая функция динамического запроса памяти, состоит в том, что прикладная программа не знает структуры дескриптора и даже его адреса. Это позволяет не привязывать SQL к особенностям какой-либо системы программирования или ОС. Все обмены информацией между собственно прикладной программой и дескрипторами производятся также с помощью специальных операторов SQL (*GET* и *SET*, см. ниже).

Второй вопрос: зачем вообще выделять память под дескрипторы динамически. Это нужно потому, что в общем случае прикладная программа, использующая динамический SQL, не знает в статике число одновременно действующих динамических операторов SQL, описание которых может потребоваться. С этим же связано то, что имя дескриптора может задаваться как литеральной строкой символов, так и через строковую переменную включающего языка, т.е. его можно генерировать во время выполнения программы.

В операторе *ALLOCATE DESCRIPTOR*, помимо прочего, может указываться число описательных элементов, на которое он рассчитан. Если, например, при выделении памяти под дескриптор в разделе *WITH MAX* указано целое положительное число N, а потом дескриптор используется для описания M ($M > N$) элементов (например, M столбцов результата запроса), то это приводит к возникновению исключительной ситуации.

17.2. Оператор освобождения памяти из-под дескриптора.

```
<deallocate descriptor statement> ::= DEALLOCATE DESCRIPTOR <descriptor name>
```

Комментарий: Выполнение этого оператора приводит к освобождению памяти из-под ранее выделенного дескриптора. После этого использование имени дескриптора незаконно в любом операторе, кроме *ALLOCATE DESCRIPTOR* .

17.3. Оператор получения информации из области дескриптора SQL.

```
<get descriptor statement> ::=
   GET DESCRIPTOR <descriptor name> <get descriptor information>
<get descriptor information> ::=
   <get count>
   ( VALUE <item number> )
   <get item information>
   [{<comma> <get item information>}...]
<get count> ::=
   <simple target specification 1>
   <equals operator> COUNT
<get item information> ::=
   <simple target specification 2>
   <equals operator>
   <descriptor item name>
<item number> ::= <simple value specification>
<simple target specification 1> ::= <simple target specification>
<simple target specification 2> ::= <simple target specification>
<descriptor item name> ::=
   TYPE
   ( LENGTH )
   ( OCTET_LENGHT )
   ( RETURNED_LENGHT )
   ( RETURNED_OCTET_LENGHT )
   ( PRECISION )
   ( SCALE )
   ( DATETIME_INTERVAL_CODE )
   ( DATATIME_INTERVAL_PRECISION )
   ( NULLABLE )
   ( INDICATOR )
   ( DATA )
   ( NAME )
   ( UNNAMED )
   ( COLLATION_CATALOG )
   ( COLLATION_SCHEMA )
   ( COLLATION_NAME )
   ( CHARACTER_SET_CATALOG )
   ( CHARACTER_SET_SCHEMA )
   ( CHARACTER_SET_NAME )
<simple target specification> ::=
   <parameter name>
   ( <embedded variable name> )
```

Комментарий: Оператор *GET DESCRIPTOR* служит для выборки описательной информации, ранее размещенной в дескрипторе с помощью оператора *DESCRIBE* . За одно выполнение оператора можно получить либо число заполненных элементов дескриптора (*COUNT*), либо информацию, содержащуюся в одном из заполненных элементов.

17.4. Оператор установки дескриптора.

```
<set descriptor statement> ::=
   SET DESCRIPTOR <descriptor name> <set descriptor information>
<set descriptor information> ::=
   <set count>
   ( VALUE <item number> <set item information> )
   [{<comma> <set item information>}...]
<set count> ::= COUNT <equals operator> <simple value specification 1>
```

```
<set item information> ::=
   <descriptor item name>
   <equals operator>
   <simple value specification 2>
<simple target specification 1> ::= <simple target specification>
<simple target specification 2> ::=<simple target specification>
<item number> ::= <simple value specification>
```

Комментарий: Оператор *SET DESCRIPTOR* служит для заполнения элементов дескриптора с целью его использования в разделе *USING*. За одно выполнение оператора можно поместить значение в поле *COUNT* (число заполненных элементов), либо частично или полностью сформировать один элемент дескриптора.

17.5. Оператор подготовки.

```
<prepare statement> ::=
   PREPARE <SQL statement name> FROM <SQL statement variable>
<SQL statement variable> ::=  <simple target specification>
<preparable statement> ::=
   <preparable SQL data statement>
   | <preparable SQL schema statement>
   | <preparable SQL transaction statement>
   | <preparable SQL session statement>
   | <preparable implementation-defined statement>
<preparable SQL data statement> ::=
   <delete statement: searched>
   | <dynamic single row select statement>
   | <insert statement>
   | <dynamic select statement>
   | <update statement: searched>
   | <preparable dynamic delete statement: positioned>
   | <preparable dynamic update statement: positioned>
<preparable SQL schema statement> ::= <SQL schema statement>
<preparable SQL transaction statement> ::= <SQL transaction statement>
<preparable SQL session statement> ::= <SQL session statement>
<dynamic select statement> ::= <cursor specification>
<dynamic simple row select statement> ::= <query specification>
<SQL statement name> ::= <statement name> | <extended statement name>
<extended statement name> ::= [scope option] <simple value specification>
<cursor specification> ::=
   <query expression> [<order by clause>] [<updatability clause>]
<updatability clause> ::=
   FOR { READ ONLY | UPDATE [ OF <column name list> ] }
<query expression> ::= <non-join query expression> | <joined table>
<query specification> ::=
   SELECT [<set quantifier>] <select list> <table expression>
<set quantifier> ::= DISTINCT | ALL
```

Комментарий: Оператор *PREPARE* вызывает компиляцию и построение плана выполнения заданного в текстовой форме оператора SQL. После успешного выполнения оператора *PREPARE* с подготовленным оператором связывается указанное (литерально или косвенно) имя этого оператора, которое потом может быть использовано в операторах *DESCRIBE*, *EXECUTE*, *OPEN CURSOR*, *ALLOCATE CURSOR* и *DEALLOCATE PREPARE*. Эта связь сохраняется до явного выполнения оператора *DEALLOCATE PREPARE*.

17.6. Оператор отказа от подготовленного оператора.

```
<deallocate prepared statement> ::= DEALLOCATE PREPARE <SQL statement name>
```

Комментарий: Выполнение этого оператора приводит к тому, что ранее подготовленный оператор SQL, связанный с указанным именем оператора, ликвидируется, и, соответственно, имя оператора становится неопределенным. Если подготовленный оператор являлся оператором

выборки, и к моменту выполнения оператора *DEALLOCATE* существовал открытый курсор, связанный с именем подготовленного оператора, то оператор *DEALLOCATE* возвращает код ошибки. Если же для подготовленного оператора выборки существовал неоткрытый курсор, образованный с помощью оператора *ALLOCATE CURSOR* , то этот курсор ликвидируется. Если курсор объявлялся оператором *DECLARE CURSOR* , то такой курсор переходит в состояние, существовавшее до выполнения оператора *PREPARE* . Если с курсором был связан подготовленный оператор (динамический *DELETE* или *UPDATE*), то для этих операторов выполняется неявный оператор *DEALLOCATE* .

17.7. Оператор запроса описания подготовленного оператора.

```
<describe statement> ::=
   <describe input statement> | <describe output statement>
<describe input statement> ::=
   DESCRIBE INPUT <SQL statement name> <using descriptor>
<describe output statement> ::=
   DESCRIBE [OUTPUT] <SQL statement name> <using descriptor>
<using clause> ::= <using arguments> | <using descriptor>
<using arguments> ::= { USING | INTO } <argument> [{<comma> <argument>}...]
<argument> ::= <target specification>
<using descriptor> ::= { USING | INTO } SQL DESCRIPTOR <descriptor name>
<target specification> ::=
   <parameter specification> | <variable specification>
<parameter specification> ::= <parameter name> [<indicator parameter>]
<indicator parameter> ::= [INDICATOR] <parameter name>
<variable specification> ::= <embedded variable name> [<indicator variable>]
<indicator variable> ::= [INDICATOR] <embedded variable name>
```

Комментарий: При выполнении оператора *DESCRIBE* происходит заполнение указанного в операторе дескриптора информацией, описывающей либо результат ранее подготовленного оператора SQL (если это оператор выборки), либо количество и типы параметров подготовленного оператора. В < *using descriptor* > здесь полагается писать *USING SQL DESCRIPTOR* .

17.8. Оператор выполнения подготовленного оператора.

```
<execute statement> ::=
   EXECUTE <SQL statement name>
   ( <result using clause> )
   ( <parameter using clause> )
<result using clause> ::= <using clause>
<parameter using clause> ::= <using clause>
```

Комментарий: Оператор *EXECUTE* может быть применен к любому ранее подготовленному оператору SQL, кроме < *dynamic select statement* > . Если это оператор < *dynamic* sin *gle row select statement* > , то оператор *EXECUTE* должен содержать раздел < *result using class* > с ключевым словом *INTO* . В любом случае число фактических параметров, задаваемых через разделы using, должно соответствовать числу формальных параметров, определенных в подготовленном операторе SQL.

17.9. Оператор подготовки с немедленным выполнением.

```
<execute immediate statement> ::= EXECUTE IMMEDIATE <SQL statement variable>
```

Комментарий: При выполнении оператора *EXECUTE IMMEDIATE* производится подготовка и немедленное выполнение заданного в текстовой форме оператора SQL. При этом подготавливаемый оператор не должен быть оператором выборки, не должен содержать формальных параметров и комментариев.

17.10. Оператор объявления курсора над динамически подготовленным оператором выборки.

```
<dynamic declare cursor> ::=
   DECLARE <cursor name> [INSENSITIVE] [SCROLL]
   CURSOR FOR <statement name>
```

Комментарий: Как определяется в новом стандарте, для всех операторов *DECLARE CURSOR* , курсоры фактически создаются при начале транзакции и уничтожаются при ее завершении. Заметим, что в этом операторе < *cursor name* > и < *statement name* > - прямо заданные идентификаторы.

17.11. Оператор определения курсора над динамически подготовленным оператором выборки.

```
<allocate cursor statement> ::=
   ALLOCATE <extended cursor name> [INSENSITIVE] [SCROLL]
   CURSOR FOR <extended statement name>
<extended cursor name> ::=
   [<scope option>] <simple value specification>
```

Комментарий: Курсоры, определяемые с помощью оператора *ALLOCATE CURSOR* , фактически создаются при выполнении такого оператора и уничтожаются при выполнении оператора *DEALLOCATE PREPARE* или при завершении транзакции. В этом операторе имена курсора и подготовленного оператора SQL могут задаваться не только в литеральной форме, но и через переменные. < *scope option* > относится к области видимости имен: в пределах текущего модуля или в пределах текущей сессии.

17.12. Оператор открытия курсора, связанного с динамически подготовленным оператором выборки.

```
<dynamic open statement> ::= OPEN <dynamic cursor name> [<using clause>]
```

Комментарий: По сути, оператор открытия курсора, связанного с динамически подготовленным оператором SQL, отличается от статического случая только возможным наличием раздела using, в котором задаются фактические параметры оператора выборки. Кроме того, имя курсора может задаваться через переменную.

17.13. Оператор чтения строки по курсору, связанному с динамически подготовленным оператором выборки.

```
<dynamic fetch statement> ::=
   FETCH [[<fetch orientation>] FROM]
   <dynamic cursor name> <using clause>
```

Комментарий: По сути, оператор чтения по курсору, связанному с динамически подготовленным оператором SQL, отличается от статического случая только возможным наличием раздела *using* , в котором задается размещение значений текущей строки результирующей таблицы. Кроме того, имя курсора может задаваться через переменную.

17.14. Оператор закрытия курсора, связанного с динамически подготовленным оператором выборки.

```
<dynamic close statement> ::= CLOSE <dynamic cursor name>
```

Комментарий: По сути, оператор закрытия курсора, связанного с динамически подготовленным оператором SQL, отличается от статического случая только тем, что имя курсора может задаваться через переменную.

17.15. Оператор позиционного удаления по курсору, связанному с динамически подготовленным оператором выборки.

```
<dynamic delete statement: positioned> ::=
```

```
DELETE FROM <table name>
WHERE CURRENT OF <dynamic cursor name>
```

Комментарий: По сути, оператор позиционного удаления по курсору, связанному с динамически подготовленным оператором SQL, отличается от статического случая только тем, что имя курсора может задаваться через переменную.

17.16. Оператор позиционной модификации по курсору, связанному с динамически подготовленным оператором выборки.

```
<dynamic update statement: positioned> ::=
   UPDATE <table name>
   SET <set clause> [{<comma> <set clause>}...]
   WHERE CURRENT OF <dynamic cursor name>
```

Комментарий: По сути, оператор позиционной модификации по курсору, связанному с динамически подготовленным оператором SQL, отличается от статического случая только тем, что имя курсора может задаваться через переменную.

17.17. Подготавливаемый оператор позиционного удаления.

```
<preparable dynamic delete statement: positioned> ::=
   DELETE [FROM <table name>]
   WHERE CURRENT OF <cursor name>
```

Комментарий: Основной смысл появления этого и следующего операторов состоит в том, что сочетание курсора, определенного на динамически подготовленном операторе выборки, и статически задаваемых операторов удаления и модификации по этому курсору, выглядит довольно нелепо. Поэтому в стандарте появились динамически подготавливаемые позиционные операторы удаления и вставки. Естественно, что выполняться они должны с помощью оператора *EXECUTE*.

17.18. Подготавливаемый оператор позиционной модификации.

```
<preparable dynamic update statement: positioned> ::=
   UPDATE [<table name>]
   SET <set clause> [{<comma> <set clause>}...]
   WHERE CURRENT OF <cursor name>
```

Комментарий: Смотри предыдущий пункт.

Если внимательно сравнивать средства динамического SQL СУБД Oracle V.6 и стандарта SQL/92, то видно, что Oracle практически вкладывается в стандарт, если не считать небольших синтаксических различий и (что существенно более важно) разного стиля работы с дескрипторами. Думается, что примерно такая же ситуация имеет место в других СУБД, поддерживающих динамический SQL.

17.19. Сводка новых возможностей SQL-3.

В стандарте SQL/92 по сравнению со стандартом SQL/89 язык был расширен главным образом количественно, хотя даже этих количественных расширений оказалось достаточно для того, чтобы стандарт SQL/92 не удалось полностью реализовать до сих пор в большинстве коммерческих СУБД. Поскольку SQL/92 не удовлетворял значительной части претензий, исторически предъявляемых к языку SQL, был сформирован новый комитет, который должен выработать стандарт языка с качественными расширениями. Язык SQL-3 пока не сформирован полностью, многие аспекты продолжают обсуждаться. Поэтому к приводимой здесь сводке возможностей нужно относиться как к сугубо предварительной.

17.19.1. Типы данных.

Набор встроенных типов данных предполагается расширить типами *BOOLEAN* и *ENUMERATED*. Хотя по причине поддержки неопределенных значений языку SQL свойственно применение трехзначной логики, тип *BOOLEAN* содержит только два возможных значения *true* и *false* false. Для представления значения unknown рекомендуется использовать *NULL*,

что, конечно, не вполне естественно. Перечисляемый тип *ENUMERATED* обладает свойствами, подобными свойствам перечисляемых типов в языках программирования.

Расширены возможности работы с неопределенными значениями. Появился новый оператор *CREATE NULL CLASS* , позволяющий ввести именованный набор именованных неопределенных значений. При определении домена можно явно указать имя класса неопределенных значений, появление которых допустимо в столбцах, связанных с этим доменом. Смысл каждого неопределенного значения интерпретируется на уровне пользователей.

Предполагается включение в язык возможности использования определенных пользователями типов данных. Видимо, будут иметься возможности определения абстрактных типов данных с произвольно сложной внутренней структурой на основе таких традиционных возможностей агрегирования и структуризации как *LIST* , *ARRAY* , *SET* , *MULTISET* и *TUPLE* , а также возможности определения объектных типов с соответствующими методами в стиле объектно-ориентированного подхода.

Появляется возможность использования принципов наследования свойств существующей таблицы (супертаблицы) при определении новой таблицы (подтаблицы). Подтаблица наследует от супертаблицы все определения столбцов и первичного ключа. Другая возможность - создать таблицу, "подобную" существующей в том смысле, что в новой таблице наследуются определения некоторых столбцов существующей таблицы.

17.19.2. Некоторые другие свойства SQL-3.

Одной из проблем реализации языка SQL всегда являлась проблема распознавания "изменяемости" соединений. Как известно, если представление включает соединение общего вида, то теоретически невозможно определить, можно ли однозначно интерпретировать операции обновления такого представления. Однако существует несколько важных классов соединений, которые заведомо являются изменяемыми. В SQL-3 предполагается выделить эти классы с помощью специальных синтаксических конструкций.

Наконец-то появляется возможность определения триггеров как комбинации спецификаций события и действия. Действие определяется как SQL-процедура, в которой могут использоваться как операторы SQL, так и ряд управляющих конструкций. На самом деле, этот механизм очень близок к тому, который реализован в Oracle V.7.

Что касается управления транзакциями, то происходит возврат к старой идее System R о возможности установки внутри транзакции точек сохранения (savepoints). В операторе *ROLLBACK* можно указать идентификатор ранее установленной точки сохранения, и тогда будет произведен откат транзакции не к ее началу, а к этой точке сохранения.

Как видно, можно ожидать наличия в SQL-3 многих интересных и полезных возможностей. Однако даже промежуточные проекты стандарта включают почти в два раза больше страниц, чем стандарт SQL/92. Поэтому трудно ожидать быстрой реализации этого стандарта после его принятия (а многие вообще сомневаются, что этот стандарт будет когда-либо реализован).

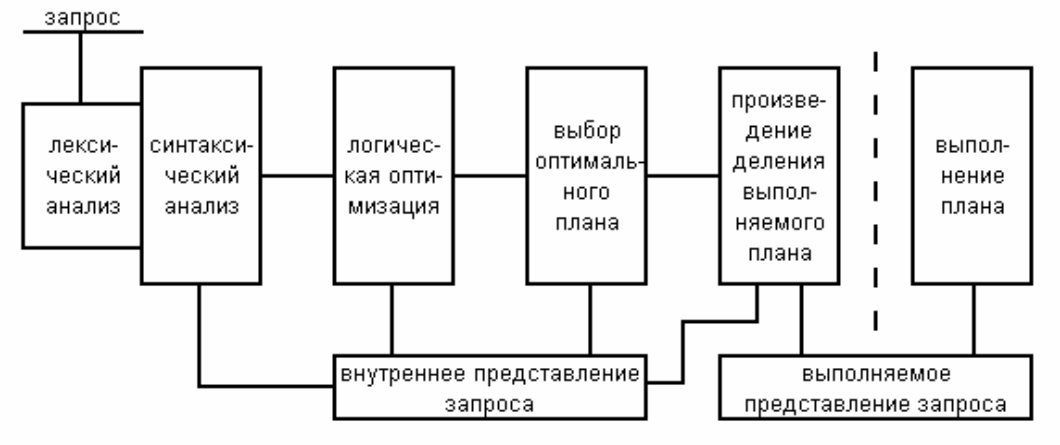
Компиляторы языка SQL

§18. Компиляторы SQL. Проблемы оптимизации.

Говоря про оптимизацию запросов в реляционных СУБД, обычно имеют в виду такой способ обработки запросов, когда по начальному представлению запроса путем его преобразований вырабатывается процедурный план его выполнения, наиболее оптимальный при существующих в базе данных управляющих структурах. Соответствующие преобразования начального представления запроса выполняются специальным компонентом СУБД - оптимизатором, и оптимальность производимого им плана запроса носит условный характер: план оптимален в соответствии с критериями, заложенными в оптимизатор.

18.1. Общая схема обработки запроса.

Можно представить, что обработка поступившего в систему запроса состоит из фаз, изображенных ниже.



На первой фазе запрос, заданный на языке запросов, подвергается лексическому и синтаксическому анализу. При этом вырабатывается его внутреннее представление, отражающее структуру запроса и содержащее информацию, которая характеризует объекты базы данных, упомянутые в запросе (отношения, поля и константы). Информация о хранимых в базе данных объектах выбирается из каталогов базы данных. Внутреннее представление запроса используется и преобразуется на следующих стадиях обработки запроса. Форма внутреннего представления должна быть достаточно удобной для последующих оптимизационных преобразований.

На второй фазе запрос во внутреннем представлении подвергается логической оптимизации. Могут применяться различные преобразования, "улучшающие" начальное представление запроса. Среди преобразований могут быть эквивалентные, после проведения которых получается внутреннее представление, семантически эквивалентное начальному (например, приведение запроса к некоторой канонической форме), Преобразования могут быть и семантическими: получаемое представление не является семантически эквивалентным начальному, но гарантируется, что результат выполнения преобразованного запроса совпадает с результатом запроса в начальной форме при соблюдении ограничений целостности, существующих в базе данных. После выполнения второй фазы обработки запроса его внутреннее представление остается непроцедурным, хотя и является в некотором смысле более эффективным, чем начальное.

Третий этап обработки запроса состоит в выборе на основе информации, которой располагает оптимизатор, набора альтернативных процедурных планов выполнения данного запроса в соответствии с его внутреннем представлением, полученным на второй фазе. Для каждого плана оценивается предполагаемая стоимость выполнения запроса. При оценках используется статистическая информация о состоянии базы данных, доступная оптимизатору. Из полученных альтернативных планов выбирается наиболее дешевый, и его внутреннее представление теперь соответствует обрабатываемому запросу.

На четвертом этапе по внутреннему представлению наиболее оптимального плана выполнения запроса формируется выполняемое представление плана. Выполняемое представление плана может быть программой в машинных кодах, если, как в случае System R, система ориентирована на компиляцию запросов в машинные коды, или быть машинно-независимым, но более удобным для интерпретации, если, как в случае INGRES, система ориентирована на интерпретацию запросов. В нашем случае это непринципиально, поскольку четвертая фаза обработки запроса уже не связана с оптимизацией.

Наконец, на пятом этапе обработки запроса происходит его реальное выполнение. Это либо выполнение соответствующей подпрограммы, либо вызов интерпретатора с передачей ему для интерпретации выполняемого плана.

18.2. Синтаксическая оптимизация запросов.

При классическом подходе к организации оптимизаторов запросов на этапе логической оптимизации производятся эквивалентные преобразования внутреннего представления запроса, которые "улучшают" начальное внутреннее представление в соответствии с фиксированными стратегиями оптимизатора. Характер "улучшений" связан со спецификой общей организации оптимизатора, в частности, с особенностями процедуры поиска возможных процедурных планов запросов, выполняемой на третьей фазе обработки запроса.

Поэтому трудно привести полную характеристику и классификацию методов логической оптимизации. Мы ограничимся несколькими примерами, а также рассмотрим один частный, но важный класс логических преобразований, касающихся сложных запросов, выраженных на языке SQL.

18.2.1. Простые логические преобразования запросов.

Очевидный класс логических преобразований запроса составляют преобразования предикатов, входящих в условие выборки, к каноническому представлению. Имеются в виду предикаты, содержащие операции сравнения простых значений. Такой предикат имеет вид

```
арифметическое выражение op арифметическое выражение
```

где "*op* " - 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 op константное арифметическое выражение" (эта форма предиката – простой предикат селекции - очень полезна при выполнении следующего этапа оптимизации). Если начальное представление предиката имеет вид $(a+5)(A\ op\ 36$ (малыми буквами обозначены имена объемлющих переменных, а большими - имена полей отношений), то каноническим представлением такого предиката может быть $A\ op\ 36/(a+5)$.

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 op 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 op - это равенство). Если в начальном представлении предикат имеет вид $5(A-a(B\ op\ b$, то его каноническое представление - $A\ op\ (b+a(B)/5$.

Наконец, для рассматриваемых предикатов более общего вида имеет смысл приведение предиката к каноническому представлению вида "арифметическое выражение op константное арифметическое выражение", где выражения правой и левой частей также приведены к каноническому представлению; например, в выражениях полностью раскрыты скобки и произведено лексикографическое упорядочение. 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, т.е. потенциально большое число раз.

При приведении предикатов к каноническому представлению можно вычислять константные выражения и избавляться от логических отрицаний.

Следующий 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. Фрагмент логического выражения $1/4\ (A > B)\ AND\ (A < 5)\ 1/4$ можно заменить на $1/4\ FALSE\ 1/4$ Возможны и более "умные" упрощения. Например, фрагмент логического выражения $1/4\ (A > B)\ AND\ (B = 5)\ 1/4$ можно заменить на $1/4\ (A > 5)\ 1/4$. Такие упрощения могут оказаться существенными для дальнейшей обработки запроса: в запросе с логическим условием первого вида предполагалось соединение отношений; после преобразования запрос уже не требует соединения.

18.2.2 Преобразования запросов с изменением порядка реляционных операций.

В традиционных оптимизаторах распространены логические преобразования, связанные с изменением порядка выполнения реляционных операций. Примером соответствующего правила преобразования в терминах реляционной алгебры может быть следующее (A и B - имена отношений):

```
(A JOIN B) WHERE restriction-on-A AND restriction-on-B
```

эквивалентно выражению

```
(A WHERE restriction-on-A) JOIN (B WHERE restriction-on-B)
```

Здесь *JOIN* обозначает реляционный оператор естественного соединения отношений; *A WHERE restriction* - оператор ограничения отношения A в соответствии с предикатом *restriction*.

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 к алгебраической форме запросов на языке SQL. Можно выявить две основные побудительные причины преобразований запросов на SQL к алгебраической форме. Первой, на наш взгляд, менее важной причиной может быть стремление к использованию реляционной алгебры в качестве унифицированного внутреннего интерфейса реляционной СУБД. Такой подход распространен при использовании специализированных машин баз данных, на основе которых реализуются различные интерфейсы доступа к базам данных. Интерфейс машины баз данных должен быть унифицирован (например, быть алгебраическим), а все остальные интерфейсы, включая интерфейс на основе SQL, приводятся к алгебраическому.

Второй причиной, особенно важной в контексте проблем оптимизации, является 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.

18.2.3 Приведение запросов со 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, т.е., 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 вся-

кий 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.

Ниже Ri обозначает i-е отношение базы данных; Ck - k-е поле (столбец) отношения.

Предикаты, допустимые в запросах языка SQL, можно разбить на следующие четыре группы:

- Простые предикаты. Это предикаты вида $Ri.Ck\ op\ X$, где X - константа или список констант, и op - оператор скалярного сравнения ($=, !=, >, \geq, <, \leq$) или оператор проверки вхождения во множество ($IS\ IN$, $IS\ NOT\ IN$).
- Предикаты со вложенными подзапросами. Это предикаты вида $Ri.Ck\ op\ Q$, где Q - блок запроса, а op может быть таким же, как для простых предикатов. Предикат может также иметь вид $Q\ op\ Ri.Ck$. В этом случае оператор принадлежности ко множеству заменяется на $CONTAINS$ или $DOES\ NOT\ CONTAIN$. Эти две формы симметричны. Достаточно рассматривать только одну.
- Предикаты соединения. Это предикаты вида $Ri.Ck\ op\ Rj.Cn$, где $Ri\ != Rj$ и op - оператор скалярного сравнения.
- Предикаты деления. Это предикаты вида $Qi\ op\ Qj$, где Qi и Qj - блоки запросов, а op может быть оператором скалярного сравнения или оператором проверки вхождения в множество.

Приведенная классификация является упрощением реальной ситуации в SQL. Не рассматриваются предикаты соединения общего вида, включающие арифметические выражения с полями более чем двух отношений.

Каноническим представлением запроса на n отношениях называется запрос, содержащий $n-1$ предикат соединения и не содержащий предикатов со вложенными подзапросами. Фактически, каноническая форма - это алгебраическое представление запроса.

Ниже приводятся два примера канонических форм запросов с предикатами разного типа. Соответствующая техника существует и для других видов предикатов.

```
SELECT Ri.Ck FROM Ri WHERE Ri.Ch IS IN
   SELECT Rj.Cm FROM Rj WHERE Ri.Cn = Rj.Cp
```

эквивалентно

```
SELECT Ri.Ck FROM Ri, Rj
   WHERE Ri.Ch = Rj.Cm AND Ri.Cn = Rj.Cp
SELECT Ri.Ck FROM Ri
   WHERE Ri.Ch = SELECT AVG (Rj.Cm) FROM Rj WHERE Rj.Cn = Ri.Cp
```

эквивалентно

```
SELECT Ri.Ck FROM Ri, Rt
   WHERE Ri.Ch = Rt.Cm AND  Ri.Cp = Rt.Cn - Rt ( Cp, Cn ) =
   SELECT Rj.Cp, AVG (Rj.Cn) FROM  Rj
   GROUP BY Rj.Cp
```

Разумность таких преобразований обосновывается тем, что оптимизатор получает возможность выбора большего числа способов выполнения запросов. Часто открывающиеся после преобразований способы выполнения более эффективны, чем планы, используемые в традиционном оптимизаторе System R.

При использовании в оптимизаторе запросов подобного подхода не обязательно производить формальные преобразования запросов. Оптимизатор должен в большей степени исполь-

зовать 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 на наших лекциях.

18.3. Семантическая оптимизация запросов.

Рассмотренные преобразования запросов основывались на семантике языка запросов, но в них не использовалась семантика базы данных, к которой адресуется запрос. Любое преобразование может быть произведено независимо от того, какая конкретная база данных имеется в виду. На самом же деле, при каждой истинно реляционной базе данных хранится и некоторая семантическая информация, определяющая, например, целостность базы данных.

Если говорить о базах данных, поддерживаемых System R, то эта информация хранится в системных каталогах базы данных в виде заданных ограничений целостности. Поскольку СУБД гарантирует целостность базы данных, то ограничения целостности можно рассматривать как аксиомы, в окружении которых формулируются запросы к базе данных.

18.3.1. Преобразования запросов на основе семантической информации.

В качестве начальных примеров преобразований запросов на основе семантической информации рассмотрим подходы известных СУБД System R и INGRES к обеспечению работы с базами данных через представления. Эти преобразования не были ориентированы на оптимизацию запросов, но имеют к ней непосредственное отношение.

Представление базы данных (view) в терминах языков SQL и QUEL - это именованный каталогизированный запрос, представляющий собой с точки зрения пользователей такой же объект базы данных, как и отношение. Представления могут использоваться в запросах так же, как и хранимые отношения (с ограничениями на использование их в операторах модификации базы данных).

Семантика представлений требует, чтобы они были точными: в момент выполнения запроса над представлением получаемая информация должна точно соответствовать текущему состоянию хранимой части базы данных. Выполнение запроса над представлением требует его материализации, т.е. вычисления отношения, определяемого запросом, который связан с представлением.

Подход System R и INGRES основан на том, что представления хранятся в каталогах базы данных во внутренней форме, получаемой после выполнения грамматического разбора (а также, возможно, логической оптимизации) соответствующего запроса. При обработке запроса над представлением до выполнения фазы логической оптимизации производится слияние внутренних форм запроса и представления. Образуется некоторая новая преобразованная внутренняя форма, и над ней производится вся последующая обработка запроса, включая логическую оптимизацию и выбор оптимального плана выполнения запроса. При этом оптимизатор получает больше информации о данном запросе и может принимать более правильные решения. Приведем простой пример.

Пусть представление определено как

```
DEFINE VIEW V (C2) AS SELECT C1 FROM R WHERE C1 > 6
```

Запрос формулируется следующим образом:

```
SELECT C2 FROM V WHERE C2 < 6
```

Если бы запрос компилировался в расчете на явную материализацию представления, был бы выбран способ его выполнения, основанный на последовательном сканировании V с выбором кортежей, удовлетворяющих условию. Запрос так бы и выполнялся, чтобы в конце концов выдать пустой результат. При слиянии же внутренних форм запроса и представления мы получили бы внутреннюю форму, эквивалентную запросу

```
SELECT C1 FROM R WHERE C1 < 6 AND C1 >6
```

Уже на фазе логической оптимизации можно было бы установить, что он эквивалентен запросу

```
SELECT C1 FROM R WHERE FALSE
```

из чего следовало бы, что результат запроса пуст.

Очевидно, что в ряде случаев этот способ обработки запросов над представлениями базы данных позволяет добиться более эффективного выполнения запроса за счет предоставления оптимизатору большей информации. Та же идея лежит и в основе семантической оптимизации запросов: использование при оптимизации запроса семантической информации, хранящейся в базе данных независимо от данного запроса.

Другим примером преобразований запросов, имеющих непосредственное отношение к оптимизации, являются преобразования запросов на модификацию базы данных для удовлетворения существующих в базе данных ограничений целостности. Этот подход впервые был применен в СУБД INGRES, но используется и в других системах, например, в System R.

Ограничением целостности называется сохраняемое в каталогах базы данных логическое выражение, составленное из предикатов над объектами базы данных. База данных находится в целостном состоянии, если удовлетворяются все заданные ограничения целостности.

Если задан запрос на модификацию базы данных, включающей ограничения целостности, удовлетворение которых требуется при выполнении любой модификации, то можно добавить соответствующие ограничениям предикаты к логическому условию запроса.

Пусть база данных, характеризующая структуру организации, состоит из отношений *EMP* и *DEPT* . В отношении *EMP* регистрируются служащие организации. Схема этого отношения *EMP* (*EMP#* , *EMPNAME* , *DEPT#*); поле *EMP#* содержит уникальный номер служащего, поле *EMPNAME* - имя служащего, а *DEPT#* - номер отдела организации, в котором работает данный сотрудник. Отношении *DEPT* хранит информацию об отделах и имеет схему *DEPT* (*DEPT#* , *EMPCOUNT*), где в поле *EMPCOUNT* хранится общее число сотрудников данного отдела. Пусть задано ограничение целостности

```
ASSERT B IMMEDIATE ON EMP: EMP.DEPT# = 6
```

Это ограничение означает запрещение занесения, удаления и модификации кортежей в отношении *EMP* со значением поля *DEPT#* , отличным от 6. Пусть обрабатывается запрос на удаление кортежа

```
DELETE EMP WHERE EMPNAME = 'Brown'
```

Если при компиляции запроса не учитывать наличие ограничения целостности, то корректным способом выполнения такого запроса будет следующий: последовательно выбирать кортежи, у которых значением поля *EMPNAME* является 'Brown'; проверять удовлетворение очередного кортежа ограничению целостности, и если проверка удовлетворительна, удалить кортеж.

Если же ограничения целостности учитываются при компиляции, то можно слить внутренние формы запроса и соответствующих ограничений целостности, а потом произвести совместную оптимизацию. В нашем случае после слияния образуется внутренняя форма, эквивалентная запросу

```
DELETE EMP WHERE EMPNAME = 'Brown' AND DEPT# = 6
```

При выполнении такого запроса не понадобятся дополнительные вызовы проверок ограничений целостности второго типа, поскольку они все уже включены в логическое условие выполнения операции удаления. Кроме того, оптимизатор получает большую свободу в выборе способа выполнения запроса. Например, если отделы малочисленны, и для отношения EMP поддержи-

вается индекс на поле $DEPT\#$, то, возможно, наиболее оптимальным способом выполнения запроса будет сканирование отношения через индекс по $DEPT\#$ с условием $DEPT\# = 6$ с удалением всякого выбираемого таким образом кортежа со значением поля $EMPNAME$, равным 'Brown'. Тем самым, преобразующие запрос действия, не направленные специально на оптимизацию, могут способствовать более эффективному выполнению запроса. Эффективность выполнения запроса удается повысить за счет использования знаний, независимо хранящихся в базе данных.

Рассмотренные примеры показывают, что идея семантической оптимизации в принципе не нова. Имеется и принципиальная разница между рассмотренными выше преобразованиями запросов и теми, которые производятся при семантической оптимизации. Основное отличие состоит в том, что когда производятся слияния внутренней формы запроса с внутренней формой представления или внутренними формами ограничений целостности, мы обязаны полностью и однозначно использовать внешнюю информацию, независимо от того, будут ли это способствовать оптимизации запроса: условие выборки представления должно быть целиком добавлено через AND к условию выборки запроса; то же относится и к набору логических условий ограничений целостности. Иначе семантика запроса над представлением или оператора модификации базы данных будет нарушена.

18.3.2. Использование семантической информации при оптимизации запросов.

Семантическая оптимизация основана на наличии в базе данных семантической информации, которую не обязательно использовать при обработке запроса, но использование которой может привести к его более оптимальному выполнению. Если семантическая оптимизация имеет дело только со знаниями, представленными в виде ограничений целостности базы данных, то при семантической оптимизации производится множество преобразованных внутренних представлений запроса; при каждом преобразовании используется некоторый поднабор ограничений целостности. Если, например, в базе данных определены два ограничения целостности A и B с логическими условиями $F1$ и $F2$ и обрабатывается запрос с условием выборки F, то в ходе семантической оптимизации будут получены внутренние представления, эквивалентные запросам с условиями F, $F\ AND\ F1$, $F\ AND\ F2$ и $F\ AND\ F1\ AND\ F2$.

Каждое из полученных внутренних представлений подвергается полной дальнейшей обработке, включая логическую оптимизацию и выбор оптимального плана выполнения; из полученных планов (все они семантически эквивалентны, т.е. вырабатывают один и тот же результат) выбирается наиболее дешевый, который и становится реальным планом выполнения исходного запроса.

Для разумного применения семантической оптимизации удобно представлять ограничения целостности базы данных в импликативной форме. Тогда можно добиться более осмысленного порядка преобразований. Пусть, например, в нашей базе данных о структуре организации отношение EMP расширено полями $STATUS$ и $SALARY$. Значения поля $STATUS$ характеризуют должность служащего, а поле $SALARY$ - его оклад. Может быть наложено ограничение целостности, выражающееся в том, что оклад, превышающий 40.000, может быть назначен только начальникам отделов:

```
ASSERT A ON  EMP:
   IF SALARY > 40.000 THEN STATUS = 'Manager'
```

Предположим, что обрабатывается запрос

```
SELECT EMPNAME, STATUS FROM EMP WHERE SALARY = 20.000
```

Если не учитывать импликативного характера ограничения целостности, то по общим правилам будет произведено слияние внутренних представлений запроса и ограничения целостности, которое заведомо ничего не даст. Если же рассматривать ограничение целостности как правило преобразования, а левую часть импликации как условие применения правила, то слия-

ние производиться и не будет, что в общем случае сэкономит время обработки запроса. Но для запроса

```
SELECT EMPNAME, STATUS FROM EMP WHERE SALARY > 40.000
```

правило преобразования применимо и приводит к семантически эквивалентному запросу

```
SELECT EMPNAME, STATUS FROM EMP WHERE STATUS = 'Manager'
```

выполнение которого может быть более эффективным.

После преобразования запроса в соответствии с некоторым правилом к полученному представлению может оказаться применимым другое правило и т.д. Возможно появление циклов преобразований. Проблема построения полного набора семантически эквивалентных представлений запроса на основе заданного набора правил в общем случае является весьма сложной. Точное решение этой проблемы может потребовать затрат, соизмеримых с затратами на выполнение запроса по наиболее оптимальному плану. Поэтому, вообще говоря, необходимо применение эвристических алгоритмов, сокращающих пространство поиска, т.е. задающих условие прекращения генерации новых представлений. Эвристики основываются на минимизации взвешенной суммы стоимостей семантической оптимизации и выполнения запроса. Примером грубой эвристики может быть следующий: оптимизация производится до тех пор, пока затраты на нее не превзойдут оценочную стоимость наиболее эффективного из всех найденных планов выполнения запроса.

18.4. Выбор и оценка альтернативных планов выполнения запросов.

Оптимизирующие преобразования, которые мы рассматривали выше, оставляли внутреннее представление запроса непроцедурным.

Процедурным представлением или планом выполнения запроса называется такое его представление, в котором детализирован порядок выполнения операций доступа к базе данных физического уровня. Как правило, в реляционных СУБД выделяется подсистема управления данными на физическом уровне. В System R, такая подсистема называется RSS (Relational Storage System) и обеспечивает простой интерфейс, позволяющий последовательно просматривать кортежи отношений, удовлетворяющие заданным условиям на значения полей, с использованием индексов или простым сканированием страниц базы данных. Кроме того, RSS позволяет производить отсортированные временные файлы и заносить, удалять и модифицировать индивидуальные кортежи. Аналогичные подсистемы явно или неявно выделяются во всех подобных СУБД.

Естественно, что до выполнения запроса необходимо выработать его процедурное представление. Поскольку оно, вообще говоря, выбирается неоднозначно, необходимо выбрать среди альтернативных планов запроса один в соответствии с некоторыми критериями. Как правило, критерием выбора плана выполнения запроса является минимизация стоимости выполнения.

Тем самым, при обработке запроса на стадии, следующей за логической оптимизацией, решаются две задачи. Первая задача: исходя из внутреннего представления запроса и информации, характеризующей управляющие структуры базы данных (например, индексы), выбрать набор потенциально возможных планов выполнения данного запроса. Вторая задача: оценить стоимость выполнения запроса в соответствии с каждым альтернативным планом и выбрать план с наименьшей стоимостью.

18.4.1. Генерация планов.

При традиционном подходе к организации оптимизаторов обе задачи решаются на основе фиксированных встроенных в оптимизатор алгоритмов. Оптимизатор может быть рассчитан на то, что ограничение любого отношения в соответствии с заданным предикатом может быть выполнено путем некоторого последовательного просмотра отношения. Так, запрос

```
SELECT EMPNAME FROM EMP WHERE DEPT# = 6 AND SALARY > 15.000
```

может выполняться последовательным сканированием отношения EMP с выбором кортежей с $DEPT\#=6$ и $SALARY>15.000$; сканированием отношения через индекс $I1$, определенный на

поле $DEPT\#$, с условием доступа к индексу $DEPT\#=6$ и условием выборки кортежа $SALARY>15.000$; наконец, сканированием отношения через индекс $I2$, определенный на поле $SALARY$, с условием доступа к индексу $SALARY>15.000$ и условием выборки кортежа $DEPT\#=6$.

Аналогично, фиксированы и стратегии выполнения более сложных операций - реляционных соединений отношений, вычисления агрегатных функций на группах кортежей отношения и т.д. Например, в System R для (экви)соединения двух отношений используются две основные стратегии: метод вложенных циклов и метод сортировок со слияниями.

Компонент оптимизатора, генерирующий выполняемые планы запросов, имеет достаточно сложную организацию; генерация плана выполнения сложного запроса - это многоэтапный процесс, в ходе которого учитываются свойства создаваемых при выполнении запроса по данному плану временных объектов базы данных. Например, пусть запрос задан над тремя отношениями и в нем имеются два предиката соединения:

```
SELECT R1.C1, R2.C2, R3.C3 FROM R1, R2, R3
   WHERE R1.C4 = R2.C5 AND R2.C5 = R3.C6.
```

Тогда, если в плане запроса выбирается порядок выполнения соединений сначала $R1$ с $R2$, а затем полученного временного отношения - с $R3$, и при этом для выполнения первого соединения выбирается метод сортировок со слиянием, то временное отношение будет заведомо отсортировано по $C5$, и одна сортировка не потребуется, если и второе соединение будет выполняться тем же методом.

Компонент оптимизатора, ведающий порождением множества альтернативных планов выполнения запроса, базируется на стратегиях декомпозиции запроса на элементарные составляющие и стратегиях выполнения элементарных составляющих. Первая группа стратегий определяет пространство поиска оптимального плана выполнения запроса, вторая направлена на то, чтобы в этом пространстве действительно находились эффективные планы выполнения запроса. Ключом к обеспечению эффективного выполнения сложного запроса является наличие эффективных стратегий выполнения элементарных составляющих. Это очень важный вопрос, но здесь мы его не касаемся: оптимизатор запросов пользуется заданными стратегиями. Рассмотрим более актуальную для оптимизатора проблему - обоснованный выбор плана выполнения запроса из множества альтернативных планов.

18.4.2. Оценка стоимости плана запроса.

После генерации множества планов выполнения запроса на основе разумных стратегий декомпозиции и эффективных стратегий выполнения элементарных операций нужно выбрать один план, в соответствии с которым будет происходить реальное выполнение запроса. При неверном выборе запрос будет выполнен неэффективно. Прежде всего необходимо определить, что понимается под эффективностью выполнения запроса. Это понятие неоднозначно и зависит от специфики операционной среды СУБД. В одних условиях можно считать, что эффективность выполнения запроса определяется временем его выполнения, т.е. реактивностью системы по отношению к обрабатываемым ею запросам. В других условиях определяющей является общая пропускная способность системы по отношению к смеси параллельно выполняемых запросов. Тогда мерой эффективности запроса можно считать количество системных ресурсов, требуемых для его выполнения и т.д.

Следуя принятой терминологии, мы будем говорить о стоимости плана выполнения запроса, определяемой ресурсами процессора и устройств внешней памяти, которые расходуются при выполнении запроса.

В традиционных реляционных СУБД выделяется подсистема управления доступом к данным на внешней памяти (RSS в System R). Данные на внешней памяти традиционно хранятся в блокированном виде; база данных занимает некоторый набор блоков одного или нескольких дисковых томов. Предполагается, что средства доступа к блокам внешней памяти порождают несравненно меньшие накладные расходы.

Как правило, подсистема управления доступом к данным на внешней памяти осуществляет буферизацию блоков базы данных в оперативной памяти. Каждый блок базы данных, прочитанный в оперативную память для выполнения запроса, сохраняется в одном из буферов буферного пула СУБД, пока не будет вытеснен из него другим блоком базы данных. Эта особенность СУБД существенна для повышения общей эффективности системы, но не учитывается (за исключением частного, но важного случая) при оценках стоимостей планов выполнения запроса. В любом случае компонент стоимости выполнения запроса, связанный с ресурсами устройств внешней памяти, монотонно зависит от числа блоков внешней памяти, доступ к которым потребуется при выполнении запроса.

С другой стороны, число блоков внешней памяти, доступ к которым требуется при выполнении запроса, монотонно зависит от числа кортежей, затрагиваемых запросом.

Из этого следует важность показателя предиката ограничения, характеризующего долю кортежей отношения, которые удовлетворяют данному предикату, и называемого степенью селективности предиката. Степени селективности простых предикатов вида $R.C\ op\ const$, где $R.C$ задает имя поля отношения базы данных, op - операция сравнения ($=, !=, >, <, >=<, <=$), а $const$ - константа, являются основой для оценок стоимости планов запроса. Точность оценок степеней селективности определяет точность общих оценок и правильность выбора оптимального плана запроса.

Степень селективности предиката $R.C\ op\ const$ зависит от вида операции сравнения, значения константы и распределения значений поля C отношения R в базе данных. Первые два параметра селективности могут быть известны при проведении оценок, но истинные распределения значений полей отношений, как правило, неизвестны. Существует ряд подходов к приближенным определениям распределений на основе статистической информации. Далее мы рассмотрим наиболее интересные из них.

Если известна степень селективности предиката ограничения отношения, то тем самым известна и мощность результирующего отношения (обычно мощности хранимых отношений известны при обработке запроса). Но в оценочных формулах учитывается необходимое для выполнения запроса число обменов с внешней памятью. Конечно, число кортежей является верхней оценкой требуемого числа обменов, но эта оценка может быть очень завышенной. Все зависит от распределения кортежей по блокам внешней памяти. В ряде случаев можно получить более точную оценку числа блоков.

Наконец, для сложных запросов, включающих, например, соединения более двух отношений, требуется оценивать размеры возникающих в процессе выполнения запроса промежуточных отношений. Чтобы оценить мощность результата соединения двух отношений, вообще говоря, необходимо знать степень селективности предиката соединения по отношению к прямому произведению отношений-операндов. В общем случае степень селективности такого предиката невозможно определить на основе простой статистической информации. Обычно применяются достаточно грубые эвристические оценки, хотя предлагаются и подходы, обеспечивающие большую точность.

Подход System R базируется на двух основных предположениях о распределениях значений полей отношений: предполагается, что значения полей всех отношений базы данных распределены равномерно и что значения любых двух полей распределены независимо. Первое предположение позволяет оценивать селективность простых предикатов на основе скудной статистической информации о базе данных. На втором предположении основываются оценки числа блоков, в которых располагается известное количество кортежей. Эти два предположения являются предметом критики System R. Они сделаны исключительно в целях упрощения оптимизатора и не могут быть теоретически обоснованы. Можно привести примеры баз данных, для которых эти предположения не оправданы. В этих случаях оценки оптимизатора System R будут неверны.

В каталогах базы данных для каждого отношения R сохраняется число кортежей в данном отношении (T) и число блоков внешней памяти, в которых располагаются кортежи отношения

(N); для каждого поля C отношения хранится число различных значений этого поля (CD), минимальное хранимое значение этого поля ($CMin$) и максимальное значение ($CMax$).

При наличии такой информации с учетом первого базового предположения степени селективности простых предикатов вычисляется просто (и точно, если распределение на самом деле равномерно). Пусть $SEL(P)$ - степень селективности предиката P.

Тогда

```
SEL (R.C = const) = CD / (CMax - CMin)
```

(при равномерном распределении степень селективности такого предиката не зависит от значения константы).

```
SEL (R.C > const) = (CMax - const) / (CMax - CMin)
```

и т.д.

При оценках числа блоков, в которых могут располагаться кортежи, удовлетворяющие предикату $R.C\ op\ const$, различаются случаи кластеризованности или некластеризованности отношения по полю C. Эти оценки имеют смысл только при рассмотрении варианта сканирования отношения с использованием индекса на поле C. При просмотре отношения без использования индекса понадобится всегда обратиться ровно к N блокам внешней памяти.

Предположения о равномерности распределений значений атрибутов отношений позволяют достаточно просто оценивать и мощности отношений, являющихся результатами двухместных реляционных и теоретико-множественных операций над отношениями.

Оценки селективности предикатов используются и для оценки затрат ресурсов процессора. Предполагается, что основная часть процессорной обработки производится в RSS. Поэтому процессорные затраты измеряются числом обращений к RSS, которое соответствует числу кортежей, получаемых при сканировании хранимого или временного отношения, т.е. селективностью логического выражения, состоящего из простых предикатов (условие выборки уровня RSS). Предположения о равномерности и независимости значений разных полей отношения позволяют легко оценить селективность логического выражения, составленного из простых предикатов, при известных их степенях селективности.

На самом деле в System R не оценивается селективность предикатов в чистом виде. Оценки оптимизатора основываются на фиксированном наборе элементарных оценочных формул, опирающихся на статистическую информацию о базе данных. Каждая формула соответствует некоторому элементарному действию системы; выполнение любого запроса состоит в комбинации некоторых элементарных действий. Примерами элементарных действий могут быть выполнение ограничения отношения путем его сканирования через кластеризованный индекс, сортировка отношения, соединение двух ранее отсортированных отношений и т.д. Общая оценка плана выполнения запроса производится в итерационном процессе, начиная с оценок элементарных операций над хранимыми отношениями.

18.4.3. Более точные оценки.

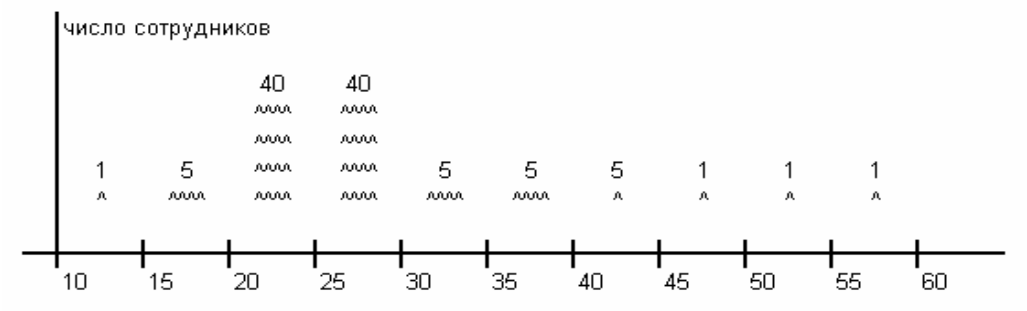
Перейдем к рассмотрению подходов к более точным оценкам стоимости выполнения планов запроса. Эти подходы можно разбить на два класса. При использовании подходов первого класса оптимизатор сохраняет жесткую структуру, аналогичную структуре оптимизатора System R, но проведение оценок основывается на более точной статистической информации, характеризующей распределения значений. Предложения второго класса более революционны и исходят из того, что для произведения планов выполнения запросов и их оценок оптимизатор должен снабжаться некоторой информацией, характерной для конкретной области приложений.

При отказе от предположения о равномерности распределения значений поля отношения необходимо уметь установить реальное распределение значений. Существует два базовых подхода к оценкам распределения значений поля отношения: параметрический и основанный на методе сигнатур. Подход System R является тривиальным частным случаем метода параметрической оценки распределения - любое распределение оценивается как равномерное. В более раз-

витом подходе было предложено использовать для оценки реального распределения значений поля отношения серию распределений Пирсона, в которую входят распределения от равномерного до нормального. Распределение выбирается из серии путем вычисления нескольких параметров на основе выборок реально встречающихся значений. Примеры практического применения этого подхода нам неизвестны.

Метод оценки распределения на основе сигнатур в целом можно описать следующим образом. Область значений поля разбивается на несколько интервалов. Для каждого интервала некоторым образом устанавливается число значений поля, попадающих в этот интервал. Внутри интервала значения считаются распределенными по некоторому фиксированному закону (как правило, принимается равномерное приближение). Рассмотрим два альтернативных подхода, связанных с сигнатурным описанием распределений.

При традиционном подходе область значений поля разбивается на N интервалов равного размера, и для каждого интервала подсчитывается число значений полей из кортежей данного отношения, попадающих в интервал. Предположим, что EMP расширено еще одним полем AGE - возраст сотрудника. Пусть всего в организации работает 60 сотрудников в возрасте от 10 до 60 лет. Тогда гистограмма, изображающая распределение значений поля AGE может иметь вид, показанный ниже на рисунке. Гистограмма построена исходя из разбиения диапазона значений поля AGE на 10 интервалов.

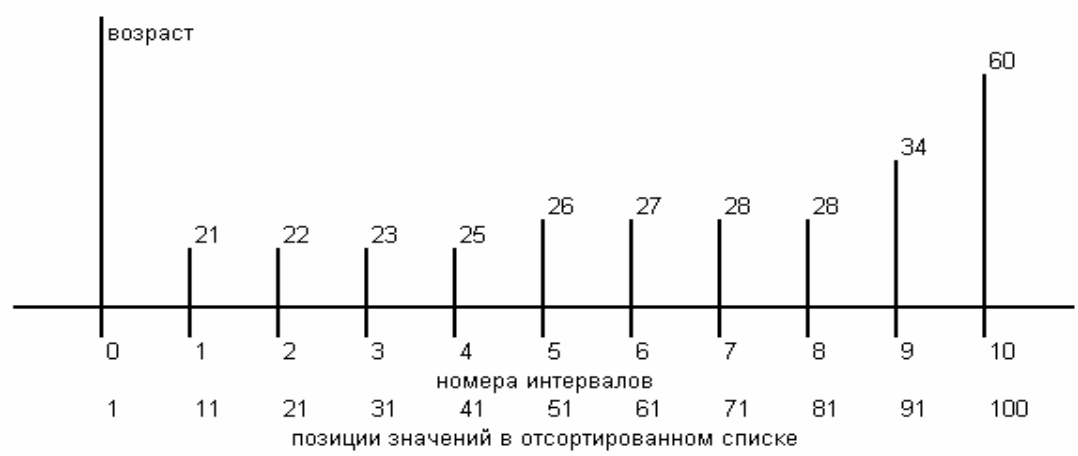


Рассмотрим, как можно оценивать селективность простых предикатов, задаваемых на поле AGE, с использованием такой гистограммы. Пусть в интервал значений AGE S_i попадает K_i значений. Тогда $SEL(EMP.AGE = const)$, если значение константы попадает в интервал значений S_i, можно оценить следующим образом: $0 \le SEL(EMP.AGE = const) \le K_i / T$ (T - общее число кортежей в отношении EMP). Отсюда средняя оценка степени селективности предиката - $K_i /(2(T))$. Например, $SEL(AGE = 29)$ оценивается в 40/200 = 0.2, а $SEL(AGE = 16)$ оценивается в 5/200 = 0.025. Это, конечно, существенно более точные оценки, чем те, которые можно получить, исходя из предположений о равномерности распределений. Но не так хорошо обстоят дела с оценками селективности простых предикатов с неравенствами.

Например, пусть требуется оценить степень селективности предиката $EMP.AGE < const$. Если значение константы попадает в интервал S_i, и SUM_i - суммарное количество значений AGE, попадающих в интервалы $S_1, S_2, ..., S_i$, то $SUM_{i-1} / T \le SEL(AGE < const) \le SUM_i / T$. Тогда средняя оценка степени селективности $(SUM_{i-1} + SUM_i)/(2(T))$, и ошибка оценки может достигать половины веса подобласти, в которую попадает значение константы предиката. Самое неприятное, что ошибка оценки зависит от значения константы и тем больше, чем больше значений поля содержится в соответствующем интервале гистограммы. Например, $SEL(AGE < 29)$ оценивается как $46/100 \le SEL(AGE < 29) \le 86/100$, откуда оценка степени селективности (46 + 86) / 200 = 0.66; при этом ошибка оценки может достигать 0.2. В то же время $SEL(AGE < 49)$ оценивается существенно более точно.

Для устранения этого дефекта был предложен другой подход к описанию распределений значений поля отношения. Идея подхода состоит в том, что множество значений поля разби-

вается на интервалы вообще говоря разного размера, чтобы в каждый интервал (кроме, вообще говоря, последнего) попадало одинаковое число значений поля. Количество интервалов выбирается исходя из ограничений по памяти, и чем оно больше, тем точнее получаемые оценки. При разбиении области значений на десять интервалов получаемая псевдогистограммная картина распределений значений поля AGE показана на рисунке ниже.



Область значений поля AGE отношения EMP разбита на 10 интервалов таким образом, что в каждый интервал попадает ровно по 10 значений поля AGE. Интервалы имеют разные размеры. Граничные значения интервалов показаны над вертикальными линиями. В псевдогистограмме допустимы интервалы, правая и левая граница которых совпадают, например, интервал (28,28). Он образовался по причине наличия в отношении EMP большого (большего десяти) числа кортежей со значением $AGE = 28$.

При использовании "псевдогистограммы" ошибки в оценках степеней селективности предикатов с операцией, отличной от равенства, уменьшаются. Размер ошибки не зависит от значения константы и уменьшается при увеличении числа интервалов.

Недостатком метода псевдогистограмм по сравнению с методом гистограмм является необходимость сортировки отношения по значениям поля для построения псевдогистограммы распределений значений этого поля. Известен подход, позволяющий получить достоверную псевдогистограмму без необходимости сортировки всего отношения.

Подход основывается на статистике Колмогорова, из которой применительно к случаю реляционных баз данных следует, что если из отношения выбирается образец из 1064 кортежей, и b - доля кортежей в образце со значениями поля $C < V$, то с достоверностью 99% доля кортежей во всем отношении со значениями поля $C < V$ находится в интервале $[b - 0.05, b + 0.05]$. При уменьшении мощности образца достоверность, естественно, уменьшается.

СУБД в архитектуре "клиент-сервер"

§19. Архитектура "клиент-сервер".

Применительно к системам баз данных архитектура "клиент-сервер" интересна и актуальна главным образом потому, что обеспечивает простое и относительно дешевое решение проблемы коллективного доступа к базам данных в локальной сети. В некотором роде системы баз данных, основанные на архитектуре "клиент-сервер", являются приближением к распределенным системам баз данных, конечно, существенно упрощенным приближением, но зато не требующим решения основного набора проблем действительно распределенных баз данных.

19.1. Открытые системы.

Реальное распространение архитектуры "клиент-сервер" стало возможным благодаря развитию и широкому внедрению в практику концепции открытых систем. Поэтому мы начнем с краткого введения в открытые системы.

Основным смыслом подхода открытых систем является упрощение комплексирования вычислительных систем за счет международной и национальной стандартизации аппаратных и программных интерфейсов. Главной побудительной причиной развития концепции открытых систем явились повсеместный переход к использованию локальных компьютерных сетей и те проблемы комплексирования аппаратно-программных средств, которые вызвал этот переход. В связи с бурным развитием технологий глобальных коммуникаций открытые системы приобретают еще большее значение и масштабность.

Ключевой фразой открытых систем, направленной в сторону пользователей, является независимость от конкретного поставщика. Ориентируясь на продукцию компаний, придерживающихся стандартов открытых систем, потребитель, который приобретает любой продукт такой компании, не попадает к ней в рабство. Он может продолжить наращивание мощности своей системы путем приобретения продуктов любой другой компании, соблюдающей стандарты. Причем это касается как аппаратных, так и программных средств и не является необоснованной декларацией. Реальная возможность независимости от поставщика проверена в отечественных условиях.

Практической опорой системных и прикладных программных средств открытых систем является стандартизованная операционная система. В настоящее время такой системой является UNIX. Фирмам-поставщикам различных вариантов ОС UNIX в результате длительной работы удалось придти к соглашению об основных стандартах этой операционной системы. Сейчас все распространенные версии UNIX в основном совместимы по части интерфейсов, предоставляемых прикладным (а в большинстве случаев и системным) программистам. Как кажется, несмотря на появление претендующей на стандарт системы Windows NT, именно UNIX останется основой открытых систем в ближайшие годы.

Технологии и стандарты открытых систем обеспечивают реальную и проверенную практикой возможность производства системных и прикладных программных средств со свойствами мобильности (portability) и интероперабельности (interoperability). Свойство мобильности означает сравнительную простоту переноса программной системы в широком спектре аппаратно-программных средств, соответствующих стандартам. Интероперабельность означает упрощения комплексирования новых программных систем на основе использования готовых компонентов со стандартными интерфейсами.

Использование подхода открытых систем выгодно и производителям, и пользователям. Прежде всего открытые системы обеспечивают естественное решение проблемы поколений аппаратных и программных средств. Производители таких средств не вынуждаются решать все проблемы заново; они могут по крайней мере временно продолжать комплексировать системы, используя существующие компоненты.

Заметим, что при этом возникает новый уровень конкуренции. Все производители обязаны обеспечить некоторую стандартную среду, но вынуждены добиваться ее как можно лучшей реализации. Конечно, через какое-то время существующие стандарты начнут играть роль сдерживания прогресса, и тогда их придется пересматривать.

Преимуществом для пользователей является то, что они могут постепенно заменять компоненты системы на более совершенные, не утрачивая работоспособности системы. В частности, в этом кроется решение проблемы постепенного наращивания вычислительных, информационных и других мощностей компьютерной системы.

19.2. Клиенты и серверы локальных сетей.

В основе широкого распространения локальных сетей компьютеров лежит известная идея разделения ресурсов. Высокая пропускная способность локальных сетей обеспечивает эффективный доступ из одного узла локальной сети к ресурсам, находящимся в других узлах.

Развитие этой идеи приводит к функциональному выделению компонентов сети: разумно иметь не только доступ к ресурсами удаленного компьютера, но также получать от этого

компьютера некоторый сервис, который специфичен для ресурсов данного рода и программные средства для обеспечения которого нецелесообразно дублировать в нескольких узлах. Так мы приходим к различению рабочих станций и серверов локальной сети.

Рабочая станция предназначена для непосредственной работы пользователя или категории пользователей и обладает ресурсами, соответствующими локальным потребностям данного пользователя. Специфическими особенностями рабочей станции могут быть объем оперативной памяти (далеко не все категории пользователей нуждаются в наличии большой оперативной памяти), наличие и объем дисковой памяти (достаточно популярны бездисковые рабочие станции, использующие внешнюю память дискового сервера), характеристики процессора и монитора (некоторым пользователям нужен мощный процессор, других в большей степени интересует разрешающая способность монитора, для третьих обязательно требуются средства убыстрения графики и т.д.). При необходимости можно использовать ресурсы и/или услуги, предоставляемые сервером.

Сервер локальной сети должен обладать ресурсами, соответствующими его функциональному назначению и потребностям сети. Заметим, что в связи с ориентацией на подход открытых систем, правильнее говорить о логических серверах (имея в виду набор ресурсов и программных средств, обеспечивающих услуги над этими ресурсами), которые располагаются не обязательно на разных компьютерах. Особенностью логического сервера в открытой системе является то, что если по соображениям эффективности сервер целесообразно переместить на отдельный компьютер, то это можно проделать без потребности в какой-либо переделке как его самого, так и использующих его прикладных программ.

Примерами сервером могут служить:

- сервер телекоммуникаций, обеспечивающий услуги по связи данной локальной сети с внешним миром;
- вычислительный сервер, дающий возможность производить вычисления, которые невозможно выполнить на рабочих станциях;
- дисковый сервер, обладающий расширенными ресурсами внешней памяти и предоставляющий их в использование рабочим станциями и, возможно, другим серверам;
- файловый сервер, поддерживающий общее хранилище файлов для всех рабочих станций;
- сервер баз данных фактически обычная СУБД, принимающая запросы по локальной сети и возвращающая результаты.

Сервер локальной сети предоставляет ресурсы (услуги) рабочим станциям и/или другим серверам.

Принято называть клиентом локальной сети, запрашивающий услуги у некоторого сервера и сервером - компонент локальной сети, оказывающий услуги некоторым клиентам.

19.3. Системная архитектура "клиент-сервер".

Понятно, что в общем случае, чтобы прикладная программа, выполняющаяся на рабочей станции, могла запросить услугу у некоторого сервера, как минимум требуется некоторый интерфейсный программный слой, поддерживающий такого рода взаимодействие (было бы по меньшей мере неестественно требовать, чтобы прикладная программа напрямую пользовалась примитивами транспортного уровня локальной сети). Из этого, собственно, и вытекают основные принципы системной архитектуры "клиент-сервер".

Система разбивается на две части, которые могут выполняться в разных узлах сети, - клиентскую и серверную части. Прикладная программа или конечный пользователь взаимодействуют с клиентской частью системы, которая в простейшем случае обеспечивает просто надсетевой интерфейс. Клиентская часть системы при потребности обращается по сети к серверной части. Заметим, что в развитых системах сетевое обращение к серверной части может и не понадобиться, если система может предугадывать потребности пользователя, и в клиентской части содержатся данные, способные удовлетворить его следующий запрос.

Интерфейс серверной части определен и фиксирован. Поэтому возможно создание новых клиентских частей существующей системы (пример интероперабельности на системном уровне).

Основной проблемой систем, основанных на архитектуре "клиент-сервер", является то, что в соответствии с концепцией открытых систем от них требуется мобильность в как можно более широком классе аппаратно-программных решений открытых систем. Даже если ограничиться UNIX-ориентированными локальными сетями, в разных сетях применяется разная аппаратура и протоколы связи. Попытки создания систем, поддерживающих все возможные протоколы, приводит к их перегрузке сетевыми деталями в ущерб функциональности.

Еще более сложный аспект этой проблемы связан с возможностью использования разных представлений данных в разных узлах неоднородной локальной сети. В разных компьютерах может существовать различная адресация, представление чисел, кодировка символов и т.д. Это особенно существенно для серверов высокого уровня: телекоммуникационных, вычислительных, баз данных.

Общим решением проблемы мобильности систем, основанных на архитектуре "клиент-сервер" является опора на программные пакеты, реализующие протоколы удаленного вызова процедур (RPC - Remote Procedure Call). При использовании таких средств обращение к сервису в удаленном узле выглядит как обычный вызов процедуры. Средства RPC, в которых, естественно, содержится вся информация о специфике аппаратуры локальной сети и сетевых протоколов, переводит вызов в последовательность сетевых взаимодействий. Тем самым, специфика сетевой среды и протоколов скрыта от прикладного программиста.

При вызове удаленной процедуры программы RPC производят преобразование форматов данных клиента в промежуточные машинно-независимые форматы и затем преобразование в форматы данных сервера. При передаче ответных параметров производятся аналогичные преобразования.

Если система реализована на основе стандартного пакета RPC, она может быть легко перенесена в любую открытую среду.

19.4. Серверы баз данных.

Термин "сервер баз данных" обычно используют для обозначения всей СУБД, основанной на архитектуре "клиент-сервер", включая и серверную, и клиентскую части. Такие системы предназначены для хранения и обеспечения доступа к базам данных.

Хотя обычно одна база данных целиком хранится в одном узле сети и поддерживается одним сервером, серверы баз данных представляют собой простое и дешевое приближение к распределенным базам данных, поскольку общая база данных доступна для всех пользователей локальной сети.

19.4.1. Принципы взаимодействия между клиентскими и серверными частями.

Доступ к базе данных от прикладной программы или пользователя производится путем обращения к клиентской части системы. В качестве основного интерфейса между клиентской и серверной частями выступает язык баз данных SQL.

Это язык по сути дела представляет собой текущий стандарт интерфейса СУБД в открытых системах. Собирательное название SQL-сервер относится ко всем серверам баз данных, основанных на SQL. Соблюдая предосторожности при программировании, некоторые из которых были рассмотрены на предыдущих лекциях, можно создавать прикладные информационные системы, мобильные в классе SQL-серверов.

Серверы баз данных, интерфейс которых основан исключительно на языке SQL, обладают своими преимуществами и своими недостатками. Очевидное преимущество - стандартность интерфейса. В пределе, хотя пока это не совсем так, клиентские части любой SQL-ориентированной СУБД могли бы работать с любым SQL-сервером вне зависимости от того, кто его произвел.

Недостаток тоже довольно очевиден. При таком высоком уровне интерфейса между клиентской и серверной частями системы на стороне клиента работает слишком мало программ СУБД. Это нормально, если на стороне клиента используется маломощная рабочая станция. Но если клиентский компьютер обладает достаточной мощностью, то часто возникает желание возложить на него больше функций управления базами данных, разгрузив сервер, который является узким местом всей системы.

Одним из перспективных направлений СУБД является гибкое конфигурирование системы, при котором распределение функций между клиентской и пользовательской частями СУБД определяется при установке системы.

19.4.2. Преимущества протоколов удаленного вызова процедур.

Упоминавшиеся выше протоколы удаленного вызова процедур особенно важны в системах управления базами данных, основанных на архитектуре "клиент-сервер".

Во-первых, использование механизма удаленных процедур позволяет действительно перераспределять функции между клиентской и серверной частями системы, поскольку в тексте программы удаленный вызов процедуры ничем не отличается от удаленного вызова, и следовательно, теоретически любой компонент системы может располагаться и на стороне сервера, и на стороне клиента.

Во-вторых, механизм удаленного вызова скрывает различия между взаимодействующими компьютерами. Физически неоднородная локальная сеть компьютеров приводится к логически однородной сети взаимодействующих программных компонентов. В результате пользователи не обязаны серьезно заботиться о разовой закупке совместимых серверов и рабочих станций.

19.4.3. Типичное разделение функций между клиентами и серверами.

В типичном на сегодняшний день случае на стороне клиента СУБД работает только такое программное обеспечение, которое не имеет непосредственного доступа к базам данных, а обращается для этого к серверу с использованием языка SQL.

В некоторых случаях хотелось бы включить в состав клиентской части системы некоторые функции для работы с "локальным кэшем" базы данных, т.е. с той ее частью, которая интенсивно используется клиентской прикладной программой. В современной технологии это можно сделать только путем формального создания на стороне клиента локальной копии сервера базы данных и рассмотрения всей системы как набора взаимодействующих серверов.

С другой стороны, иногда хотелось бы перенести большую часть прикладной системы на сторону сервера, если разница в мощности клиентских рабочих станций и сервера чересчур велика. В общем-то при использовании RPC это сделать нетрудно. Но требуется, чтобы базовое программное обеспечение сервера действительно позволяло это. В частности, при использовании ОС UNIX проблемы практически не возникают.

19.4.4. Требования к аппаратным возможностям и базовому программному обеспечению клиентов и серверов.

Из предыдущих рассуждений видно, что требования к аппаратуре и программному обеспечению клиентских и серверных компьютеров различаются в зависимости от вида использования системы.

Если разделение между клиентом и сервером достаточно жесткое (как в большинстве современных СУБД), то пользователям, работающим на рабочих станциях или персональных компьютерах, абсолютно все равно, какая аппаратура и операционная система работают на сервере, лишь бы он справлялся с возникающим потоком запросов.

Но если могут возникнуть потребности перераспределения функций между клиентом и сервером, то уже совсем не все равно, какие операционные системы используются.

Распределенные базы данных

§20. Распределенные БД.

Основная задача систем управления распределенными базами данных состоит в обеспечении средства интеграции локальных баз данных, располагающихся в некоторых узлах вычислительной сети, с тем, чтобы пользователь, работающий в любом узле сети, имел доступ ко всем этим базам данных как к единой базе данных.

При этом должны обеспечиваться:

- простота использования системы;

- возможности автономного функционирования при нарушениях связности сети или при административных потребностях;
- высокая степень эффективности.

20.1. Разновидности распределенных систем

Возможны однородные и неоднородные распределенные базы данных. В однородном случае каждая локальная база данных управляется одной и той же СУБД. В неоднородной системе локальные базы данных могут относиться даже к разным моделям данных. Сетевая интеграция неоднородных баз данных - это актуальная, но очень сложная проблема. Многие решения известны на теоретическом уровне, но пока не удается справиться с главной проблемой – недостаточной эффективностью интегрированных систем.

Заметим, что более успешно практически решается промежуточная задача - интеграция неоднородных SQL-ориентированных систем. Понятно, что этому в большой степени способствует стандартизация языка SQL и общее следование производителей СУБД принципам открытых систем.

Мы ограничимся рассмотрением проблем однородных распределенных СУБД на примере System R*.

20.2. Распределенная система управления базами данных System R*

Основную цель проекта можно сформулировать следующим образом: обеспечить средства интеграции локальных баз данных System R, располагающихся в узлах вычислительной сети, с тем, чтобы пользователь, работающий в любом узле сети, имел доступ ко всем этим базам данных так, как если бы они были централизованы. При этом должны обеспечиваться:

- легкость использования системы;
- возможности автономного функционирования при нарушениях связности сети или при административных потребностях;
- высокая степень эффективности.

Для решения этих проблем было необходимо принять ряд проектных решений, касающихся декомпозиции исходного запроса, оптимального выбора способа выполнения запроса, согласованного выполнения транзакций, обеспечения синхронизации, обнаружения и разрешения распределенных тупиков, восстановления состояния баз данных после разного рода сбоев узлов сети.

Легкость использования системы достигается за счет того, что пользователи System R* (разработчики прикладных программ и конечные пользователи) остаются в среде языка SQL, т.е. могут продолжать работать в тех же внешних условиях, что и в System R (и SQL/DS и DB2). Возможность использования SQL основывается на обеспечении System R* прозрачности местоположения данных. Система автоматически обнаруживает текущее местоположение упоминаемых в запросе пользователя объектов данных; одна и та же прикладная программа, включающая предложения SQL, может быть выполнена в разных узлах сети. При этом в каждом узле сети на этапе компиляции запроса выбирается наиболее оптимальный план выполнения запроса в соответствии с расположением данных в распределенной системе.

Обеспечению автономности узлов сети в System R* уделяется очень большое внимание. Каждая локальная база данных администрируется независимо от других. Возможны автономное подключение новых пользователей, смена версии автономной части системы и т.д. Система спроектирована таким образом, что в ней не требуются централизованные службы именования объектов или обнаружения тупиков. В индивидуальных узлах не требуется наличие глобального знания об операциях, выполняющихся в других узлах сети; работа с доступными базами данных может продолжаться при выходе из строя отдельных узлов сети или линий связи.

Высокая степень эффективности системы является одним из наиболее ключевых требований к распределенным системам управления базами данных вообще и к System R* в частности. Для достижения этой цели используются два основных приема.

Во-первых, как и в System R, в System R* выполнению запроса предшествует его компиляция. В ходе этого процесса производится поиск употребляемых в запросе имен объектов баз данных в распределенном каталоге и замена имен на внутренние идентификаторы; проверка

прав доступа пользователя, от имени которого производится компиляция, на выполнение соответствующих операций над базами данных и выбор наиболее оптимального глобального плана выполнения запроса, который затем подвергается декомпозиции и по частям рассылается в соответствующие узлы сети, где производится выбор оптимальных локальных планов выполнения компонентов запроса и происходит генерация модулей доступа в машинных кодах. В результате множество действий производится на стадии компиляции до реального выполнения запроса. Обработанная посредством прекомпилятора System R* прикладная программа, включающая предложения SQL, может в дальнейшем выполняться много раз без дополнительных накладных расходов. Использование распределенного каталога, распределенная компиляция и оптимизация запросов являются наиболее интересными и оригинальными аспектами проекта System R*.

Вторым средством повышения эффективности системы является возможность перемещения удаленных отношений в локальную базу данных. Диалект SQL, используемый в System R*, включает предложение *MIGRATE TABLE* , при выполнении которого указанное отношение переносится в локальную базу данных. Это средство, находящееся в распоряжении пользователей, конечно, в ряде случаев может помочь добиться более эффективного прохождения транзакций. Естественно, как и для всех операций, операция *MIGRATE* по отношению к указанному отношению доступна не любому пользователю, а лишь тем, которые обладают соответствующим правом.

Прежде, чем перейти к более детальному изложению наиболее интересных аспектов реализации System R*, упомянем некоторые средства, которые разработчики этой системы предполагали реализовать на начальной стадии проекта, но которые реализованы не были (причем некоторые из них, видимо, и не будут никогда реализованы). Предполагалось иметь в системе средства горизонтального и вертикального разделения отношений распределенной базы данных, средства дублирования отношений в нескольких узлах с поддержкой согласованности копий и средства поддержания мгновенных снимков состояния баз данных в соответствии с заданным запросом.

Для задания горизонтального разделения отношений в SQL была введена конструкция вида:

```
DISTRIBUTE TABLE <table-name> HORIZONTALLY INTO
   <name> WHERE <predicate> IN SEGMENT <segment-name site>
   ...
   <name> WHERE <predicate> IN SEGMENT <segment-name site>
```

При выполнении предложения такого типа указанное отношение разбивалось на ряд подотношений, содержащих кортежи, удовлетворяющие соответствующему предикату из раздела *WHERE* , и каждое полученное таким образом подотношение посылалось в казанный узел для хранения в сегменте с указанным именем. Гарантируется согласованное состояние разделов при изменении отношения.

Вертикальное разделение производилось с помощью оператора:

```
DISTRIBUTE TABLE <table-name> VERTICALLY INTO
   <name> WHERE <column-name-list> IN SEGMENT <segment-name site>
   ...
   <name> WHERE <column-name-list> IN SEGMENT <segment-name site>
```

При выполнении такого предложения также образовывался набор подотношений с помощью проекции заданного отношения на атрибуты из заданного списка. Каждое полученное подотношение затем посылалось для хранения в сегменте с указанным именем в соответствующий узел. После этого система ответственна за поддержание согласованного состояния образованных разделов.

Горизонтальное и вертикальное разделение отношений реально не используются в System R*, хотя очевидно, что выполнение собственно оператора *DISTRIBUTE* никаких технических трудностей не вызывает. Трудности возникают при обеспечении согласованности разделов

(смотри ниже). Кроме того, разделенные отношения очень трудно использовать. В соответствии с идеологией системы учет наличия разделов отношения в разных узлах сети должен производить оптимизатор, т.е. количество потенциально возможных планов выполнения запросов, которые должны оцениваться оптимизатором, еще более возрастает. При том, что в распределенной системе число возможных планов и так очень велико, и оптимизатор работает на пределе сложности, разумным образом использовать разделенные отношения невозможно. Разработчики оптимизатора System R* не были в состоянии учитывать разделенность отношений. Поэтому и вводить в систему разделенные отношения пока бессмысленно.

Для задания требования поддержки копий отношения в нескольких узлах сети предлагалось использовать новую конструкцию SQL:

```
DISTRIBUTE TABLE <table-name> REPLICATED INTO
   <name> IN SEGMENT <segment-name site>
   ...
   <name> IN SEGMENT <segment-name site>
```

При выполнении такого предложения должна была производиться рассылка копий указанного отношения для хранения в именованных сегментах указанных узлов сети. Система должна автоматически поддерживать согласованность копий.

Как и в случае разделенных отношений, кроме существенных проблем поддержания согласованности копий, проблемой является и разумное использование копий, наличие которых должно было бы учитываться оптимизатором.

Создание мгновенного снимка состояния баз данных в соответствии с заданным запросом на выборку должно было производиться с использованием новой конструкции SQL:

```
DEFINE SNAPSHOT <snapshot-name> (<attribute-list>)
   AS <query> REFRESHED EVERY <period>
```

При выполнении предложения фактически производится выполнение указанного в нем запроса на выборку, а результирующее отношение сохраняется под указанным в предложении именем в локальной базе данных в том узле, в котором выполняется предложение. После этого мгновенный снимок периодически обновляется в соответствии с запомненным запросом.

Можно обновить мгновенный снимок, не дожидаясь истечения временного интервала, указанного в определении, путем выполнения предложения
REFRESH SNAPSHOT < snapshot − name > .

Разумное использование мгновенных снимков более реально, чем использование разделенных отношений и копированных отношений, поскольку их можно в некотором смысле рассматривать как материализованные представления базы данных. Имя мгновенного снимка можно было бы использовать прямо в запросе на выборку там, где можно использовать имена базовых отношений или представлений. Большие проблемы связаны с обновлением отношений через их мгновенные снимки, поскольку в момент обновления содержимое мгновенного снимка может расходиться с текущим содержимым базового отношения.

По отношению к мгновенным снимкам проблем поддержания согласованного состояния мгновенного снимка и базовых отношений не существует, поскольку автоматическое согласование не требуется. Что же касается разделенных отношений и раскопированных отношений, то для них эта проблема общая и достаточно трудная. Во-первых, согласование разделов и копий вызывает существенные накладные расходы при выполнении операций модификации хранимых отношений. Для этого требуется выработка и соблюдение специальных протоколов модификации.

Во-вторых, введение копированных отношений обычно производится не столько для увеличения эффективности системы, сколько для увеличения доступности данных при нарушении связности сети. В системах, в которых применяется этот подход, при нарушении связности сети работа с распределенной базой данных обычно продолжается только в одной из образовавшихся подсетей. При этом для выбора подсети используются алгоритмы голосования; решение прини-

мается на основе учета количества связных узлов сети. Применяются и другие подходы, но все они очень дорогостоящие, а самое главное, они плохо согласуются с базовым подходом System R* по поводу выбора способа выполнения запроса на стадии его компиляции. Поэтому, как нам кажется, в System R* никогда не будут реализованы средства, позволяющие тем или иным способом поддерживать копии отношений в нескольких узлах сети.

Далее мы рассмотрим аспекты проекта System R*, которые нашли отражение в ее реализации и являются на наш взгляд наиболее интересными: средства именования объектов и организацию распределенного каталога баз данных; подход к распределенным компиляции и выполнению запросов; особенности использования представлений; средства оптимизации запросов; особенности управления транзакциями; средства синхронизации и распределенный алгоритм обнаружения синхронизационных тупиков.

20.2.1. Именование объектов и организация распределенного каталога.

Напомним прежде всего, что полное имя отношения (базового или представления) в базе данных System R имеет вид имя-пользователя.имя-отношения, где имя-пользователя идентифицирует пользователя - создателя отношения, а имя-отношения - это то имя, которое было указано в предложениях *CREATE TABLE* или *CREATE VIEW* . В запросах можно указывать либо это полное имя отношения, либо его локальное имя. Во втором случае при компиляции используются стандартные правила дополнения локального имени до полного с использованием в качестве составляющей имя-пользователя идентификатора пользователя, от имени которого выполняется компиляция.

В System R* используется развитие этого подхода. Системное имя отношения включает четыре компонента: идентификатор пользователя-создателя отношения; идентификатор узла сети, в котором выполнялась операция создания отношения; локальное имя отношения, присвоенное ему при создании; идентификатор узла, в котором отношение располагалось непосредственно после своего создания (напомним, что отношение может перемещаться из одного узла в другой при выполнении операции *MIGRATE*).

В запросе на SQL можно использовать системные имена объектов, но разрешается использовать и короткие локальные имена (либо локальное имя, квалифицированное именем пользователя). При этом возможны две интерпретации локального имени. Оно может интерпретироваться как часть системного имени, и в этом случае по умолчанию дополняется до системного, исходя из идентификатора узла, в котором производится компиляция, и идентификатора пользователя, от имени которого она производится (если имя пользователя не указано явно). Вторая возможная интерпретация локального имени заключается в рассмотрении его как имени ранее определенного синонима системного имени.

Для определения синонимов SQL расширен оператором вида:

```
DEFINE SYNONYM <relation-name> AS <system-wide-name>
```

При выполнении такого предложения в локальный каталог заносится соответствующая информация.

Таким образом, при компиляции запроса всегда можно определить системные имена всех употребляемых в нем отношений: либо они явно указаны, либо могут быть получены на основе информации из локальных отношений-каталогов.

Концепция распределенного каталога System R* основана на наличии у каждого объекта распределенной базы данных уникального системного имени. Принято следующее соглашение: информация о размещении любого объекта базы данных (идентификатор текущего узла, в котором размещен объект) сохраняется в локальном каталоге того узла, в котором объект располагался непосредственно после создания (родового узла).

Следовательно, для получения полной информации об отношении в общем случае необходимо сначала воспользоваться локальным каталогом узла, в котором происходит компиляция, затем обратиться к удаленному каталогу родового узла данного отношения и в заключение воспользоваться каталогом текущего узла. Таким образом, для получения точной системной ин-

формации о любом отношении распределенной базы данных может потребоваться самое большее два удаленных доступа к отношениям-каталогам.

Применяется некоторая оптимизация этой процедуры. В локальном каталоге узла могут храниться копии элементов каталога других узлов (своего рода кэш-каталог). Согласованность копий элементов каталога не поддерживается. Эта информация используется на первой стадии компиляции запроса (мы рассматриваем распределенную компиляцию в следующем подразделе), а затем, на второй стадии, если информация, касающаяся некоторого объекта, оказалась неточной, она уточняется на основе локального каталога того узла, в котором объект хранится в настоящее время. Обнаружение некорректности копии элемента каталога производится за счет наличия при каждом элементе каталога номера версии. Если учесть достаточную инерционность системной информации, эта оптимизация может оказаться существенной.

20.2.2. Распределенная компиляция запросов.

Как мы уже отмечали, запросы на языке SQL до своего реального выполнения подвергаются компиляции. Как и в случае System R компиляция запроса может производиться на стадии прекомпиляции прикладной программы, написанной на традиционном языке программирования (PL/1, Cobol, ассемблер) с включением предложений SQL, или в динамике выполнения транзакции при выполнении предложения *PREPARE* . С точки зрения пользователей процесс компиляции в System R* приводит к тем же результатам, что и в System R: для каждого предложения SQL образуется программа к машинных кодах (секция модуля доступа), вызовы которой помещаются в текст исходной прикладной программы.

Однако, в действительности процесс компиляции запроса в System R* намного более сложен, чем в System R, что и естественно по причине гораздо более сложных сетевых взаимодействий, которые потребуются при реальном выполнении транзакции. Распределенная компиляция запросов в System R* включает множество технических ухищрений и тонкостей. Мы не будем касаться их всех в этой статье по причинам недостатка информации и ограниченности объема. Рассмотрим только общую схему распределенной компиляции.

Будем называть главным узлом тот узел сети, в котором инициирован процесс компиляции предложения SQL, и дополнительными узлами - те узлы, которые вовлекаются в этот процесс в ходе его выполнения. На самом грубом уровне процесс компиляции можно разбить на следующие фазы:

- В главном узле производится грамматический разбор предложения SQL с построением внутреннего представления запроса в виде дерева. На основе информации из локального каталога главного узла и удаленных каталогов дополнительных узлов производится замена имен объектов, фигурирующих в запросе, на их системные идентификаторы.
- В главном узле генерируется глобальный план выполнения запроса, в котором учитывается лишь порядок взаимодействий узлов при реальном выполнении запроса. Для выработки глобального плана используется расширение техники оптимизации, применяемой в System R. Глобальный план отображается в преобразованном соответствующим образом дереве запроса.
- Если в глобальном плане выполнения запроса участвуют дополнительные узлы, производится его декомпозиция на части, каждую из которых можно выполнить в одном узле (например, локальная фильтрация отношения в соответствии с заданным в условии выборки предикате ограничения). Соответствующие части запроса (во внутреннем представлении) рассылаются в дополнительные узлы.
- В каждом узле, участвующем в глобальном плане выполнения запроса (главном и дополнительных) выполняется завершающая стадия выполнения компиляции. Эта стадия включает, по существу, две последние фазы процесса компиляции запроса в System R: оптимизацию и генерацию машинных кодов. Производится проверка прав пользователя, от имени которого производится компиляция, на выполнение соответствующих действий; происходит обработка представлений базы данных (здесь имеются тонкости, связанные с тем, что представления могут включать удаленные отношения; ниже мы еще остановимся на этом, а пока будем считать, что в запросе употребляются только имена ба-

зовых отношений); осуществляется локальная оптимизация обрабатываемой части запроса в соответствии с имеющимися индексами; наконец, производится генерация кода.

20.2.3. Управление транзакциями и синхронизация.

Выполнение транзакции в распределенной системе управления базами данных System R*, естественно, является распределенным. Транзакция начинается в главном узле при обращении к какой-либо секции ранее подготовленного (на этапе компиляции) модуля доступа. Как и в System R, модуль доступа загружается в виртуальную память задачи, обращение к секции модуля доступа - это вызов подпрограммы. Однако, в отличие от System R, эта подпрограмма, кроме своего локального программного кода и вызовов функций RSS, содержит еще и вызовы удаленных подсекций модуля доступа. Эти вызовы интерпретируются в духе вызовов удаленных процедур. Тем самым выполнение одной транзакции, инициированной в некотором узле сети *A* влечет, вообще говоря, инициирование транзакций в дополнительных узлах. Основной новой по сравнению с System R проблемой является проблема согласованного завершения распределенной транзакции, чтобы результаты ее выполнения во всех затронутых ею узлах были либо отображены в состояние локальных баз данных, либо полностью отсутствовали.

Для достижения этой цели в System R* используется двухфазный протокол завершения распределенной транзакции. Этот протокол является общеупотребимым в распределенных системах баз данных и описан во многих литературных источниках. Поэтому мы здесь опишем его очень кратко и неформально.

Для описания протокола используется следующая модель. Имеется ряд независимых транзакций-участников распределенной транзакции, выполняющихся под управлением транзакции-координатора. Решение об окончании распределенной транзакции принимается координатором. После этого выполняется первая фаза завершения транзакции, когда координатор передает каждому из участников сообщение "подготовиться к завершению". Получив такое сообщение, каждый участник переходит в состояние готовности как к немедленному завершению транзакции, так и к ее откату. В терминах System R* это означает, что буфер журнала с записями об изменениях базы данных участника выталкиваются на внешнюю память, но синхронизационные захваты не снимаются. После этого каждый участник, успешно выполнивший подготовительные действия, посылает координатору сообщение "готов к завершению". Если координатор получает такие сообщения ото всех участников, то он начинает вторую фазу завершения, рассылая всем участникам сообщение "завершить транзакцию", и это считается завершением распределенной транзакции. Если не все участники успешно выполнили первую фазу, то координатор рассылает всем участникам сообщение "откатить транзакцию", и тогда эффект воздействия распределенной транзакции на состояние баз данных отсутствует.

По отношению к особенностям реализации двухфазного протокола завершения транзакции в System R* заметим еще следующее. В качестве координатора выступает транзакция, выполняющаяся в главном узле, т.е. та, по инициативе которой возникли дополнительные транзакции. Тем самым, наличие центрального координирующего узла не требуется, что соответствует требованию автономности узлов. Для откатов транзакций используется базовый механизм точек сохранения System R. Наконец, классический протокол двухфазного завершения оптимизирован, чтобы сократить число необходимых сообщений.

Как и в System R, согласованность состояния баз данных при параллельном выполнении нескольких транзакций в System R* обеспечивается на основе механизма синхронизационных захватов объектов базы данных при соблюдении двухфазного протокола захватов. Напомним, что это означает разбиение каждой транзакции с точки зрения синхронизации на две фазы – рабочую фазу, на которой захваты только устанавливаются, и фазу завершения, когда все захваты объектов базы данных, произведенные данной транзакцией, снимаются. Синхронизация производится в точности так же, как и в System R: каждая транзакция-участник обращается к локальной базе данных через RSS своего узла. Основной новой проблемой является проблема возможных распределенных тупиков, которые могут возникнуть между несколькими распределенными транзакциями, выполняющимися параллельно. (Тупики между транзакциями - участниками одной распределенной транзакции невозможны, поскольку все участники получают один общий

идентификатор транзакции и не конфликтуют по синхронизации). Для обнаружения распределенных синхронизационных тупиков в System R* применяется оригинальный распределенный алгоритм, не нарушающий требования автономности узлов сети и минимизирующий число передаваемых по сети сообщений и необходимую процессорную обработку.

Основная идея алгоритма состоит в том, что в каждом узле периодически производится анализ на предмет существования тупика с использованием информации о связях транзакций по ожиданию ресурсов, локальной в данном узле и полученной от других узлов. При проведении этого анализа обнаруживаются либо циклы ожиданий, что означает наличие тупика, либо потенциальные циклы, которые необходимо уточнить в других узлах. Эти потенциальные циклы представляются в виде специального вида строк. Строка представляет собой по сути дела список транзакций. Все транзакции упорядочены в соответствии со значениями своих идентификаторов ("номеров транзакций"). Строка передается для дальнейшего анализа в следующий узел (узел, в котором выполняется самая правая в строке транзакция) только в том случае, если номер первой транзакции в строке меньше номера последней транзакции. (Это оптимизация, уменьшающая число передаваемых по сети сообщений). Этот процесс продолжается до обнаружения тупика.

Если обнаруживается наличие синхронизационного тупика, он разрушается за счет уничтожения (отката) одной из транзакций, входящей в цикл. В качестве жертвы выбирается транзакция, выполнившая к этому моменту наименьший объем работы. Эта информация также передается по сети вместе со строками, описывающими связи транзакций по ожиданию.

20.3. Интегрированные или федеративные системы и мультибазы данных.

Направление интегрированных или федеративных систем неоднородных БД и мульти-БД появилось в связи с необходимостью комплексирования систем БД, основанных на разных моделях данных и управляемых разными СУБД.

Основной задачей интеграции неоднородных БД является предоставление пользователям интегрированной системы глобальной схемы БД, представленной в некоторой модели данных, и автоматическое преобразование операторов манипулирования БД глобального уровня в операторы, понятные соответствующим локальным СУБД. В теоретическом плане проблемы преобразования решены, имеются реализации.

При строгой интеграции неоднородных БД локальные системы БД утрачивают свою автономность. После включения локальной БД в федеративную систему все дальнейшие действия с ней, включая администрирование, должны вестись на глобальном уровне. Поскольку пользователи часто не соглашаются утрачивать локальную автономность, желая тем не менее иметь возможность работать со всеми локальными СУБД на одном языке и формулировать запросы с одновременным указанием разных локальных БД, развивается направление мульти-БД. В системах мульти-БД не поддерживается глобальная схема интегрированной БД и применяются специальные способы именования для доступа к объектам локальных БД. Как правило, в таких системах на глобальном уровне допускается только выборка данных. Это позволяет сохранить автономность локальных БД.

Как правило, интегрировать приходится неоднородные БД, распределенные в вычислительной сети. Это в значительной степени усложняет реализацию. Дополнительно к собственным проблемам интеграции приходится решать все проблемы, присущие распределенным СУБД: управление глобальными транзакциями, сетевую оптимизацию запросов и т.д. Очень трудно добиться эффективности.

Как правило, для внешнего представления интегрированных и мульти-БД используется (иногда расширенная) реляционная модель данных. В последнее время все чаще предлагается использовать объектно-ориентированные модели, но на практике пока основой является реляционная модель. Поэтому, в частности, включение в интегрированную систему локальной реляционной СУБД существенно проще и эффективнее, чем включение СУБД, основанной на другой модели данных.

Современные направления исследований и разработок

Конечно, несмотря на всю их привлекательность, классические реляционные системы управления базами данных являются ограниченными. Они идеально походят для таких традиционных приложений, как системы резервирования билетов или мест в гостиницах, а также банковских систем, но их применение в системах автоматизации проектирования, интеллектуальных системах обучения и других системах, основанных на знаниях, часто является затруднительным. Это прежде всего связано с примитивностью структур данных, лежащих в основе реляционной модели данных. Плоские нормализованные отношения универсальны и теоретически достаточны для представления данных любой предметной области. Однако в нетрадиционных приложениях в базе данных появляются сотни, если не тысячи таблиц, над которыми постоянно выполняются дорогостоящие операции соединения, необходимые для воссоздания сложных структур данных, присущих предметной области.

Другим серьезным ограничением реляционных систем являются их относительно слабые возможности по части представления семантики приложения. Самое большее, что обеспечивают реляционные СУБД,- это возможность формулирования и поддержки ограничений целостности данных. Как мы отмечали в лекции 6, после проектирования реляционной базы данных многие знания проектировщика остаются зафиксированными в лучшем случае на бумаге по причине отсутствия в системе соответствующих выразительных средств.

Осознавая эти ограничения и недостатки реляционных систем, исследователи в области баз данных выполняют многочисленные проекты, основанные на идеях, выходящих за пределы реляционной модели данных. По всей видимости, какая-либо из этих работ станет основой систем баз данных будущего. Следует заметить, что тематика современных исследований, относящихся к базам данных, исключительно широка. В завершающей части курса мы приведем только короткий обзор наиболее важных направлений.

§21. Системы управления базами данных следующего поколения.

В этом разделе очень кратко рассматриваются основные направления исследований и разработок в области так называемых постреляционных систем, т.е. систем, относящихся к следующему поколению (хотя термин "next-generation DBMS" зарезервирован для некоторого подкласса современных систем).

Хотя отнесение СУБД к тому или иному классу в настоящее время может быть выполнено только условно (например, иногда объектно-ориентированную СУБД O2 относят к системам следующего поколения), можно отметить три направления в области СУБД следующего поколения. Чтобы не изобретать названий, будем обозначать их именами наиболее характерных СУБД.

- ***Направление Postgres.*** Основная характеристика: максимальное следование (насколько это возможно с учетом новых требований) известным принципам организации СУБД (если не считать коренной переделки системы управления внешней памятью).
- ***Направление Exodus/Genesis.*** Основная характеристика: создание собственно не системы, а генератора систем, наиболее полно соответствующих потребностям приложений. Решение достигается путем создания наборов модулей со стандартизованными интерфейсами, причем идея распространяется вплоть до самых базисовых слоев системы.
- ***Направление Starburst.*** Основная характеристика: достижение расширяемости системы и ее приспосабливаемости к нуждам конкретных приложений путем использования стандартного механизма управления правилами. По сути дела, система представляет собой некоторый интерпретатор системы правил и набор модулей-действий, вызываемых в соответствии с этими правилами. Можно изменять наборы правил (существует специаль-

ный язык задания правил) или изменять действия, подставляя другие модули с тем же интерфейсом.

В целом можно сказать, что СУБД следующего поколения - это прямые наследники реляционных систем. Тем не менее, различные направления систем третьего поколения стоит рассмотреть отдельно, поскольку они обладают некоторыми разными характеристиками.

21.1. Ориентация на расширенную реляционную модель.

Одним из основных положений реляционной модели данных является требование нормализации отношений: поля кортежей могут содержать лишь атомарные значения. Для традиционных приложений реляционных СУБД - банковских систем, систем резервирования и т.д. - это вовсе не ограничение, а даже преимущество, позволяющее проектировать экономные по памяти БД с предельно понятной структурой. Запросы с соединениями в таких системах сравнительно редки, для динамической поддержки целостности используются соответствующие средства SQL.

Однако с появлением эффективных реляционных СУБД их стали пытаться использовать и в менее традиционных прикладных системах - САПР, системах искусственного интеллекта и т.д. Такие системы обычно оперируют сложно структурированными объектами, для реконструкции которых из плоских таблиц реляционной БД приходится выполнять запросы, почти всегда требующие соединения отношений. В соответствии с требованиями разработчиков нетрадиционных приложений появилось направление исследований баз сложных объектов. Основной смысл этого направления состоит в том, что в руки проектировщиков даются настолько же мощные и гибкие средства структуризации данных, как те, которые были присущи иерархическим и сетевым системам базам данных.

Однако важным отличием является то, что в системах баз данных, поддерживающих сложные объекты, сохраняется четкая граница между логическим и физическим представлениями таких объектов. В частности, для любого сложного объекта (произвольной сложности) должна обеспечиваться возможность перемещения или копирования его как единого целого из одной части базы данных в другую ее часть или даже в другую базу данных. Это очень обширная область исследований, в которой затрагиваются вопросы моделей данных, структур данных, языков запросов, управления транзакциями, журнализации и т.д. Во многом эта область соприкасается с областью объектно-ориентированных БД. (И в этой области настолько же плохо обстоят дела с теоретическим обоснованием.)

Близкое, но, вообще говоря, основанное на других принципах направление представлено системами баз данных, основанных на реляционной модели, в которой не обязательно поддерживается первая нормальная форма отношений. Напомним, что требование атомарности значений, которые могут храниться в элементах кортежей отношений, является базовым требованием классической реляционной модели. Приведение исходного табличного представления предметной области к "плоскому" виду является обязательным первым шагом в процессе проектирования реляционной базы данных на основе принципов нормализации. С другой стороны, абсолютно очевидно, что такое "уплощение" таблиц хотя и является необходимым условием получения неизбыточной и "правильной" схемы реляционной базы данных, в дальнейшем потенциально вызывает выполнение многочисленных соединений, наличие которых может свести на нет все преимущества "хорошей" схемы базы данных.

Так вот, в "ненормализованных" реляционных моделях данных допускается хранение в качестве элемента кортежа кортежей (записей), массивов (регулярных индексированных множеств данных), регулярных множеств элементарных данных, а также отношений. При этом такая вложенность может быть, по существу, неограниченной. Если внимательно продумать эти идеи, то станет понятно, что они приводят (только) к логически обособленным (от физического представления) возможностям иерархической модели данных. Но это уже не так уж и мало, если учесть, что к настоящему времени фактически полностью сформировано теоретическое основание реляционных баз данных с отказом от нормализации. Скорее всего, в этой теории все еще имеются темные места (они наличествуют даже в классической реляционной теории), но тем не менее большинство известных теоретических результатов реляционной теории уже распростра-

нено на ненормализованную модель, и даже такой пурист реляционной модели, как Дейт, полагает возможным использование ограниченной и контролируемой реляционной модели в SQL-3.

21.2. Абстрактные типы данных.

Одной из наиболее известных СУБД третьего поколения является система Postgres, а создатель этой системы М.Стоунбрекер, по всей видимости, является вдохновителем всего направления. В Postgres реализованы многие интересные средства: поддерживается темпоральная модель хранения и доступа к данным (см. ниже) и в связи с этим абсолютно пересмотрен механизм журнализации изменений, откатов транзакций и восстановления БД после сбоев; обеспечивается мощный механизм ограничений целостности; поддерживаются ненормализованные отношения (работа в этом направлении началась еще в среде Ingres), хотя и довольно странным способом: в поле отношения может храниться динамически выполняемый запрос к БД.

Одно свойство системы Postgres сближает ее со свойствами объектно-ориентированных СУБД. В Postgres допускается хранение в полях отношений данных абстрактных, определяемых пользователями типов. Это обеспечивает возможность внедрения поведенческого аспекта в БД, т.е. решает ту же задачу, что и ООБД, хотя, конечно, семантические возможности модели данных Postgres существенно слабее, чем у объектно-ориентированных моделей данных. Основная разница состоит в том, что системы класса Postgres не предполагают наличия языка программирования, одинаково понимаемого как внешней системой программирования, так и системой управления базами данных. Если с использованием такой системы программирования определяются типы данных, хранимых в базе данных, то СУБД оказывается не в состоянии контролировать безопасность этих определений, т.е. то отсутствует гарантия, что при выполнении процедур абстрактных типов данных не будет разрушена сама база данных.

Заметим, что в середине 1995 г. компания Sun Microsystems объявила о выпуске нового продукта - языка и семейства интерпретаторов под названием Java. Язык Java является расширенным подмножеством языка Си++. Основные изменения касаются того, что язык является пооператорно интерпретируемым (в стиле языка Бейсик), а программы, написанные на языке Java, гарантированно безопасны (в частности, при выполнении любой программы не может быть поврежден интерпретатор). Для этого, в частности, из языка удалена арифметика над указателями. В то же время Java остается мощным объектно-ориентированным языком, включающим развитые средства определения абстрактных типов данных. Компания Sun продвигает язык Java с целью расширения возможностей службы Всемирной Паутины (World Wide Web) Internet (основная идея состоит в том, что из сервера WWW в клиенты передаются не данные, а объекты, методы которых запрограммированы на языке Java и интерпретируются на стороне клиента; этот подход, в частности, решает проблему нестандартизованного представления мультимедийной информации). Однако, как кажется, интерпретируемый и безопасный язык типа Java может быть успешно применен и в системах баз данных, допускающих хранение данных с типами, определенными пользователями.

21.3. Генерация систем баз данных, ориентированных на приложения.

Идея очень проста: никогда не станет возможно создать универсальную систему управления базами данных, которая будет достаточна и не избыточна для применения в любом приложении. Например, если посмотреть на использование универсальных коммерческих СУБД (например, Oracle или Informix) в российской действительности, то можно легко увидеть, что по крайней мере в 90% случаев применяется не более чем 30% возможностей системы. Тем не менее, приложение несет всю тяжесть поддерживающей его СУБД, рассчитанной на использование в наиболее общих случаях.

Поэтому очень заманчиво производить не законченные универсальные СУБД, а нечто вроде компиляторов компиляторов (compiler compiler), позволяющих собрать систему баз данных, ориентированную на конкретное приложение (или класс приложений). Рассмотрим простые примеры:

В системах резервирования проездных билетов запросы обычно настолько просты (например, "выдать очередное место на рейс SU 645"), что нет особого смысла производить широко-

масштабную оптимизацию запросов. С другой стороны, информация, хранящаяся в базе данных настолько критична (кто из нас не сталкивался с проблемой наличия двух или более билетов на одно место?), что особо важным является гарантированные синхронизация обновлений базы данных и ее восстановление после любого сбоя.

С другой стороны, в статистических системах запросы могут быть произвольно сложными (например, "выдать количество холостых особей мужского пола, проживающих в России и имеющих не менее трех зарегистрированных детей"), что вызывает необходимость использования развитых средств оптимизации запросов. С другой стороны, поскольку речь идет о статистике, здесь не требуется поддержка строгой сериализации транзакций и точного восстановления базы данных после сбоев. (Поскольку речь идет о статистической информации, потеря нескольких ее единиц обычно не существенна.)

Поэтому желательно уметь генерировать систему баз данных, возможности (и соответствующие накладные расходы) которой в достаточной степени соответствуют потребностям приложения. На сегодняшний день на коммерческом рынке такие "генерационные" системы отсутствуют (например, при выборе сервера системы Oracle вы не можете отказаться от каких-либо ненужных для вашего приложения его свойств или потребовать наличия некоторых дополнительных свойств). Однако существуют как минимум два экспериментальных прототипа - Genesis и Exodus.

Обе эти генерационные системы основаны прежде всего на принципах модульности и точного соблюдения установленных интерфейсов. По сути дела, системы состоят из минимального ядра (развитой файловой системы в случае Exodus) и технологического механизма программирования дополнительных модулей. В проекте Exodus этот механизм основывается на системе программирования E, которая является простым расширением Си++, поддерживающим стабильное хранение данных во внешней памяти. Вместо готовой СУБД предоставляется набор "полуфабрикатов" с согласованными интерфейсами, из которых можно сгенерировать систему, максимально отвечающую потребностям приложения.

21.4. Оптимизация запросов, управляемая правилами.

В лекции 18 мы коротко рассмотрели проблемы оптимизации запросов, которые приходится решать в компиляторах языков баз данных. Возможно, главным выводом, который следовало бы сделать на основе материалов этой лекции, является то, что оптимизатор запросов - это наиболее громоздкий, сложный и критичный компонент СУБД. Все разработчики систем управления базами данных согласны с тем, что на оптимизации запросов экономить нельзя. Чем большее количество вариантов выполнения запроса анализируется и чем более точные оценки стоимости плана выполнения запроса применяются, тем более вероятно, что запрос будет выполнен эффективно.

Главная неприятность, связанная с оптимизаторами запросов, состоит в том, что отсутствует принятая технология их программирования. Обычно оптимизатор представляет собой аморфный набор относительно независимых процедур, которые жестко связаны с другими компонентами компилятора. По этой причине очень трудно менять стратегии оптимизации или качественно их расширять (делать это приходится, поскольку оптимизация вообще и оптимизация запросов, в частности, в принципе является эмпирической дисциплиной, а хорошие эмпирические алгоритмы появляются только со временем).

Каким же образом можно решать эту проблему? Имеются компромиссные решения, не выводящие за пределы традиционной технологии производства компиляторов. В основном все они связаны с применением тех или иных инструментальных средств, обеспечивающих автоматизацию построения компиляторов. Среди них отметим технологию, примененную Ричардом Столлманом в его семействе компиляторов gcc, а также инструментальный пакет Cocktail, разработанный в Германском университете города Карлсруе. Основным производственным достоинством gcc является применение единого языка в качестве средства внутреннего представления программы. Высокоуровневый лиспоподобный язык RTL используется на всех фазах компиляции gcc, что позволяет применять одни и те же преобразующие процедуры на разных стадиях оптимизации программы (вплоть до стадии машинно-зависимых оптимизаций).

В пакете Cocktail обеспечивается набор универсальных, настраиваемых процедур преобразования графов внутреннего представления программы. В некотором смысле Cocktail можно рассматривать как специализированный язык для написания компиляторов (компиляторов любых языков, а не только процедурных языков программирования или декларативных языков баз данных). Как утверждается, Cocktail позволяет повысить производительность труда разработчиков компиляторов в 2-3 раза.

Однако наиболее революционный подход среди известных автору был применен в экспериментальной постреляционной системе компании IBM Starburst. В некотором смысле этот подход является развитием идеи Столлмана, примененной при реализации широко популярного редактора Emacs. Напомним, что в основе этого редактора лежит интерпретатор расширенного диалекта языка Common Lisp. Сам этот интерпретатор написан на языке Си, а основная часть редактора написана на языке Лисп. Это позволяет, среди прочего, добавлять в редактор новые возможности, не покидая его среды: вы просто пишете новый текст на Лиспе и объявляете соответствующую функцию подключенной к редактору.

Система Starburst основана на применении продукционной системы. Эта система является, по существу, виртуальной машиной, в которой выполняются все компоненты СУБД, начиная от компилятора языка баз данных (расширенного варианта языка SQL) и заканчивая подсистемой непосредственного исполнения запросов. Сама СУБД представляет собой набор продукционных правил, каждое из которых вызывается продукционной системой при возникновении соответствующего события и выполняет некоторое действие, которое, в свою очередь, может привести к возникновению события, активизирующего другое правило. Правила представляются на специальном языке. Поддерживается набор предопределенных правил низкого уровня, обеспечивающих интерфейс с подсистемой управления внешней памятью (конечно, по соображениям эффективности эта подсистема написана не на продукционном языке).

Очевидно, что такая организация системы обеспечивает максимальную гибкость. Например, чтобы внедрить в оптимизатор запросов некоторую новую стратегию выполнения (например, расширить применяемый набор методов выполнения эквисоединения) достаточно дополнительно написать одно или несколько новых правил, связанных с событием требования выполнить соединение. Тем самым, Starburst может использоваться (и реально используется в научно-исследовательских лабораториях компании IBM) как мощное и гибкое средство исследования методов оптимизации запросов. Конечно, сомнительно, что технология, положенная в основу Starburst, позволит этой системе конкурировать с такими выполненными в традиционной манере коммерческими СУБД, как DB2, Oracle, Informix и т.д.

21.5. Поддержка исторической информации и темпоральных запросов.

Обычные БД хранят мгновенный снимок модели предметной области. Любое изменение в момент времени t некоторого объекта приводит к недоступности состояния этого объекта в предыдущий момент времени. Самое интересное, что на самом деле в большинстве развитых СУБД предыдущее состояние объекта сохраняется в журнале изменений, но возможности доступа со стороны пользователя нет.

Конечно, можно явно ввести в хранимые отношения явный временной атрибут и поддерживать его значения на уровне приложений. Более того, в большинстве случаев так и поступают. Недаром в стандарте SQL появились специальные типы данных date и time. Но в таком подходе имеются несколько недостатков: СУБД не знает семантики временного поля отношения и не может контролировать корректность его значений; появляется дополнительная избыточность хранения (предыдущее состояние объекта данных хранится и в основной БД, и в журнале изменений); языки запросов реляционных СУБД не приспособлены для работы со временем.

Существует отдельное направление исследований и разработок в области темпоральных БД. В этой области исследуются вопросы моделирования данных, языки запросов, организация данных во внешней памяти и т.д. Основной тезис темпоральных систем состоит в том, что для любого объекта данных, созданного в момент времени t_1 и уничтоженного в момент времени t_2, в БД сохраняются (и доступны пользователям) все его состояния во временном интервале $[t_1, t_2]$.

Исследования и построения прототипов темпоральных СУБД обычно выполняются на основе некоторой реляционной СУБД. Как и в случае дедуктивных БД темпоральная СУБД - это надстройка над реляционной системой. Конечно, это не лучший способ реализации с точки зрения эффективности, но он прост и позволяет производить достаточно глубокие исследования.

Примером кардинального (но, может быть, преждевременного) решения проблемы темпоральных БД может служить СУБД Postgres. Эта система была спроектирована и разработана М.Стоунбрекером для исследований и обучения студентов в университете г.Беркли, и он безбоязненно шел в ней на самые смелые эксперименты.

Главными особенностями системы управления памятью в Postgres являются, во-первых, то, что в ней не ведется обычная журнализация изменений базы данных и мгновенно обеспечивается корректное состояние базы данных после перевызова системы с утратой состояния оперативной памяти. Во-вторых, система управления памятью поддерживает исторические данные. Запросы могут содержать временные характеристики интересующих объектов. Реализационно эти два аспекта связаны.

Основное решение состоит в том, что при модификациях кортежа изменения производятся не на месте его хранения, а заводится новая запись, куда помещаются измененные поля. Эта запись содержит, кроме того, данные, характеризующие транзакцию, производившую изменения (в том числе и время ее завершения), и подшивается в список к изменявшемуся кортежу. В системе поддерживается уникальная идентификация транзакций и имеется специальная таблица транзакций, хранящаяся в стабильной памяти. Таким образом, после сбоев просто не следует обращать внимание на хвостовые записи списков, относящиеся к незакончившемся транзакциям. Синхронизация поддерживается на основе обычного двухфазного протокола захватов.

Отдельный компонент системы осуществляет архивацию объектов базы данных. Он производит сборку разросшихся списков изменявшихся кортежей и записывает их в область архивного хранения. К этой области тоже могут адресоваться запросы, но уже только на чтение.

Система ориентирована на использование оптических дисков с разовой записью и стабильной оперативной памяти (хотя бы небольшого объема). При наличии таких технических средств она выигрывает по эффективности даже при работе в традиционном режиме по сравнению со схемой с журнализацией. Однако возможна работа и на традиционной аппаратуре, тогда эффективность системы слегка уступает традиционным схемам.

Соответствующие возможности работы с историческими данными заложены в язык Postquel (и в этом его главное отличие от последних вариантов Quel). Возможна выборка информации, хранившейся в базе данных в указанное время, в указанном временном интервале и т.д. Кроме того, имеется возможность создавать версии отношений и допускается их последующая модификация с учетом изменений основных вариантов.

§22. Объектно-ориентированные СУБД.

Направление объектно-ориентированных баз данных (ООБД) возникло сравнительно давно. Публикации появлялись уже в середине 1980-х. Однако наиболее активно это направление развивается в последние годы. С каждым годом увеличивается число публикаций и реализованных коммерческих и экспериментальных систем.

Возникновение направления ООБД определяется прежде всего потребностями практики: необходимостью разработки сложных информационных прикладных систем, для которых технология предшествующих систем БД не была вполне удовлетворительной.

Конечно, ООБД возникли не на пустом месте. Соответствующий базис обеспечивают как предыдущие работы в области БД, так и давно развивающиеся направления языков программирования с абстрактными типами данных и объектно-ориентированных языков программирования.

Что касается связи с предыдущими работами в области БД, то на наш взгляд наиболее сильное влияние на работы в области ООБД оказывают проработки реляционных СУБД и следующее хронологически за ними семейство БД, в которых поддерживается управление сложными

объектами. Кроме того, исключительное влияние на идеи и концепции ООБД и, как кажется, всего объектно-ориентированного подхода оказал подход к семантическому моделированию данных. Достаточное влияние оказывают также развивающиеся параллельно с ООБД направления дедуктивных и активных БД.

Среди языков и систем программирования наибольшее первичное влияние на ООБД оказал Smalltalk. Этот язык сам по себе не является полностью пионерским, хотя в нем была введена новая терминология, являющаяся теперь наиболее распространенной в объектно-ориентированном программировании. На самом деле, Smalltalk основан на ряде ранее выдвинутых концепций.

Большое число опубликованных работ не означает, что все проблемы ООБД полностью решены. Как отмечается в Манифесте группы ведущих ученых, занимающихся ООБД, современная ситуация с ООБД напоминает ситуацию с реляционными системами середины 1970-х. При наличии большого количества экспериментальных проектов (и даже коммерческих систем) отсутствует общепринятая объектно-ориентированная модель данных, и не потому, что нет ни одной разработанной полной модели, а по причине отсутствия общего согласия о принятии какой-либо модели. На самом деле имеются и более конкретные проблемы, связанные с разработкой декларативных языков запросов, выполнением и оптимизацией запросов, формулированием и поддержанием ограничений целостности, синхронизацией доступа и управлением транзакциями и т.д.

Тематика ООБД очень широка, объем этой лекции не позволяет рассмотреть все вопросы. Тем не менее, мы постараемся в систематической манере проанализировать наиболее важные аспекты ООБД.

22.1. Связь объектно-ориентированных СУБД с общими понятиями объектно-ориентированного подхода.

В наиболее общей и классической постановке объектно-ориентированный подход базируется на следующих концепциях:

- объекта и идентификатора объекта;
- атрибутов и методов;
- классов;
- иерархии и наследования классов.

Любая сущность реального мира в объектно-ориентированных языках и системах моделируется в виде объекта. Любой объект при своем создании получает генерируемый системой уникальный идентификатор, который связан с объектом все время его существования и не меняется при изменении состояния объекта.

Каждый объект имеет состояние и поведение. Состояние объекта - набор значений его атрибутов. Поведение объекта - набор методов (программный код), оперирующих над состоянием объекта. Значение атрибута объекта - это тоже некоторый объект или множество объектов. Состояние и поведение объекта инкапсулированы в объекте; взаимодействие объектов производится на основе передачи сообщений и выполнении соответствующих методов.

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.

Допускается порождение нового класса на основе уже существующего класса – наследование. 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, назы-

ваемый решеткой классов. Объект подкласса считается принадлежащим любому суперклассу этого класса.

Одной из более поздних идей объектно-ориентированного подхода является идея возможного переопределения атрибутов и методов суперкласса в подклассе (перегрузки методов). Эта возможность увеличивает гибкость, но порождает дополнительную проблему: при компиляции объектно-ориентированной программы могут быть неизвестны структура и программный код методов объекта, хотя его класс (в общем случае - суперкласс) известен. Для разрешения этой проблемы применяется так называемый метод позднего связывания, означающий, по сути дела, интерпретационный режим выполнения программы с распознаванием деталей реализации объекта во время выполнения посылки сообщения к нему. Введение некоторых ограничений на способ определения подклассов позволяет добиться эффективной реализации без потребностей в интерпретации.

Как видно, при таком наборе базовых понятий, если не принимать во внимание возможности наследования классов и соответствующие проблемы, объектно-ориентированный подход очень близок к подходу языков программирования с абстрактными (или произвольными) типами данных.

С другой стороны, если абстрагироваться от поведенческого аспекта объектов, объектно-ориентированный подход весьма близок к подходу семантического моделирования данных (даже и по терминологии). Фундаментальные абстракции, лежащие в основе семантических моделей, неявно используются и в объектно-ориентированном подходе. На абстракции агрегации основывается построение сложных объектов, значениями атрибутов которых могут быть другие объекты. Абстракция группирования - основа формирования классов объектов. На абстракциях специализации/обобщения основано построение иерархии или решетки классов.

Видимо, наиболее важным новым качеством ООБД, которого позволяет достичь объектно-ориентированный подход, является поведенческий аспект объектов. В прикладных информационных системах, основывавшихся на БД с традиционной организацией (вплоть до тех, которые базировались на семантических моделях 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 гарантирования согласованности этих структурной (статической) и поведенческой (динамической) частей. В среде ООБД проектирование, разработка и сопровождение прикладной системы становится 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. Это определяется потребностями долговременного хранения объектов во внешней памяти, ассоциативного доступа к объектам, обеспечения согласованного состояния ООБД в условиях мультидоступа и тому подобных возможностей, свойственных базам данных. Выделяются три аспекта, отсутствующие в традиционной парадигме, но требующиеся в ООБД.

Первый аспект касается потребности в средствах спецификации знаний при определении класса (ограничений целостности, правил дедукции и т.п.). Второй аспект - потребность в механизме определения разного рода семантических связей между объектами вообще говоря разных классов. Фактически это означает требование полного распространения на ООБД средств семантического моделирования данных. Потребность в использовании абстракции ассоциирования отмечается и в связи с использовании ООБД в сфере автоматизированного проектирования и инженерии. Наконец, третий аспект связан с пересмотром понятия класса. В контексте ООБД оказывается более удобным рассматривать класс как множество объектов данного типа, т.е. одновременно поддерживать понятия и типа и класса объектов.

Как мы отмечали во введении, в сообществе исследователей ООБД и разработчиков систем отсутствует полное согласие, но в большинстве практических работ используется некоторое расширение объектно-ориентированного подхода.

22.2. Объектно-ориентированные модели данных.

Первой формализованной и общепризнанной моделью данных была реляционная модель Кодда. В этой модели, как и во всех следующих, выделялись три аспекта - структурный, целостный и манипуляционный. Структуры данных в реляционной модели основываются на плоских нормализованных отношениях, ограничения целостности выражаются с помощью средств логики первого порядка и, наконец, манипулирование данными осуществляется на основе реляционной алгебры или равносильного ей реляционного исчисления. Как отмечают многие исследователи, своим успехом реляционная модель данных во многом обязана тому, что опиралась на строгий математический аппарат теории множеств, отношений и логики первого порядка. Разработчики любой конкретной реляционной системы считали своим долгом показать соответствие своей конкретной модели данных общей реляционной модели, которая выступала в качестве меры "реляционности" системы.

Основные трудности объектно-ориентированного моделирования данных проистекают из того, что такого развитого математического аппарата, на который могла бы опираться общая объектно-ориентированная модель данных, не существует. В большой степени поэтому до сих пор нет базовой объектно-ориентированной модели. С другой стороны, некоторые авторы утверждают, что общая объектно-ориентированная модель данных в классическом смысле и не может быть определена по причине непригодности классического понятия модели данных к парадигме объектной ориентированности.

Один из наиболее известных теоретиков в области моделей данных Беери предлагает в общих чертах формальную основу ООБД, далеко не полную и не являющуюся моделью данных в традиционном смысле, но позволяющую исследователям и разработчикам систем ООБД по крайней мере говорить на одном языке (если, конечно, предложения Беери будут развиты и получат поддержку). Независимо от дальнейшей судьбы этих предложений мы считаем полезным кратко их пересказать.

Во-первых, следуя практике многих ООБД, предлагается выделить два уровня моделирования объектов: нижний (структурный) и верхний (поведенческий). На структурном уровне поддерживаются сложные объекты, их идентификация и разновидности связи "isa". База данных - это набор элементов данных, связанных отношениями "входит в класс" или "является атрибутом". Таким образом, БД может рассматриваться как ориентированный граф. Важным моментом является поддержание наряду с понятием объекта понятия значения (позже мы увидим, как много на этом построено в одной из успешных объектно-ориентированных СУБД O2).

Важным аспектом является четкое разделение схемы БД и самой БД. В качестве первичных концепций схемного уровня ООБД выступают типы и классы. Отмечается, что во всех системах, использующих только одно понятие (либо тип, либо класс), это понятие неизбежно перегружено: тип предполагает наличие некоторого множества значений, определяемого структурой данных этого типа; класс также предполагает наличие множества объектов, но это множество определяется пользователем. Таким образом, типы и классы играют разную роль, и для строгости и недвусмысленности требуется одновременная поддержка обоих понятий.

Беери не представляет полной формальной модели структурного уровня ООБД, но выражает уверенность, что текущего уровня понимания достаточно, чтобы формализовать такую модель. Что же касается поведенческого уровня, предложен только общий подход к требуемому для этого логическому аппарату (логики первого уровня недостаточно).

Важным, хотя и недостаточно обоснованным предположением Беери является то, что двух традиционных уровней - схемы и данных - для ООБД недостаточно. Для точного определения ООБД требуется уровень мета-схемы, содержимое которой должно определять виды объектов и связей, допустимых на схемном уровне БД. Мета-схема должна играть для ООБД такую же роль, какую играет структурная часть реляционной модели данных для схем реляционных баз данных.

Имеется множество других публикаций, относящихся к теме объектно-ориентированных моделей данных, но они либо затрагивают достаточно частные вопросы, либо используют слишком серьезный для этого обзора математический аппарат (например, некоторые авторы определяют объектно-ориентированную модель данных на основе теории категорий).

Для иллюстрации текущего положения дел мы кратко рассмотрим особенности конкретной модели данных, применяемой в объектно-ориентированной СУБД O2 (это, конечно, тоже не модель данных в классическом смысле).

В O2 поддерживаются объекты и значения. Объект - это пара (идентификатор, значение), причем объекты инкапсулированы, т.е. их значения доступны только через методы - процедуры, привязанные к объектам. Значения могут быть атомарными или структурными. Структурные значения строятся из значений или объектов, представленных своими идентификаторами, с помощью конструкторов множеств, кортежей и списков. Элементы структурных значений доступны с помощью предопределенных операций (примитивов).

Возможны два вида организации данных: классы, экземплярами которых являются объекты, инкапсулирующие данные и поведение, и типы, экземплярами которых являются значения. Каждому классу сопоставляется тип, описывающий структуру экземпляров класса. Типы определяются рекурсивно на основе атомарных типов и ранее определенных типов и классов с применением конструкторов. Поведенческая сторона класса определяется набором методов.

Объекты и значения могут быть именованными. С именованием объекта или значения связана долговременность его хранения (persistency): любые именованные объекты или значения долговременны; любые объект или значение, входящие как часть в другой именованный объект или значение, долговременны.

С помощью специального указания, задаваемого при определении класса, можно добиться долговременности хранения любого объекта этого класса. В этом случае система автоматически порождает значение-множество, имя которого совпадает с именем класса. В этом множестве гарантированно содержатся все объекты данного класса.

Метод - программный код, привязанный к конкретному классу и применимый к объектам этого класса. Определение метода в O2 производится в два этапа. Сначала объявляется сигнатура метода, т.е. его имя, класс, типы или классы аргументов и тип или класс результата. Методы могут быть публичными (доступными из объектов других классов) или приватными (доступными только внутри данного класса). На втором этапе определяется реализация класса на одном из языков программирования O2 (подробнее языки обсуждаются в следующем разделе нашего обзора).

В модели O2 поддерживается множественное наследование классов на основе отношения супертип/подтип. В подклассе допускается добавление и/или переопределение атрибутов и методов. Возможные при множественном наследовании двусмысленности (по именованию атрибутов и методов) разрешаются либо путем переименования, либо путем явного указания источника наследования. Объект подкласса является объектом каждого суперкласса, на основе которого порожден данный подкласс.

Поддерживается предопределенный класс "Object", являющийся корнем решетки классов; любой другой класс является неявным наследником класса "Object" и наследует предопределенные методы ("*is _ same* ", "*is _ value _ equal* " и т.д.).

Специфической особенностью модели O2 является возможность объявления дополнительных "исключительных" атрибутов и методов для именованных объектов. Это означает, что конкретный именованный объект-представитель класса может обладать типом, являющимся подтипом типа класса. Конечно, с такими атрибутами не работают стандартные методы класса, но специально для именованного объекта могут быть определены дополнительные (или переопределены стандартные) методы, для которых дополнительные атрибуты уже доступны. Подчеркивается, что дополнительные атрибуты и методы привязываются не к конкретному объекту, а к имени, за которым в разные моменты времени могут стоять вообще говоря разные объекты. Для реализации исключительных атрибутов и методов требуется развитие техники позднего связывания.

В следующем разделе мы среди прочего рассмотрим особенности языков программирования и запросов системы O2, которые, конечно, тесно связаны со спецификой модели данных.

22.3. Языки программирования объектно-ориентированных баз данных.

Как отмечают многие исследователи и разработчики, объектно-ориентированная система БД представляет собой объединение системы программирования и СУБД (альтернативная, но не более проясняющая суть дела точка зрения состоит в том, что объектно-ориентированная СУБД - это СУБД, основанная на объектно-ориентированной модели данных).

22.3.1. Потеря соответствия между языками программирования и языками запросов в реляционных СУБД.

Мы уже говорили, что основная практическая надобность в ООБД связана с потребностью в некоторой интегрированной среде построения сложных информационных систем. В этой среде должны отсутствовать противоречия между структурной и поведенческой частями проекта и должно поддерживаться эффективное управление сложными структурами данных во внешней памяти. В отличие от случая реляционных систем, где при создании приложения приходится одновременно использовать ориентированный на работу со скалярными значениями процедурный язык программирования и ориентированный на работу со множествами декларативный язык запросов (это принято называть потерей соответствия - impedance mismatch), языковая среда ООБД - это объектно-ориентированная система программирования, естественно включающая средства работы с долговременными объектами. "Естественность" включения средств работы с БД в язык программирования означает, что работа с долговременными (хранимыми во внешней БД) объектами должна происходить на основе тех же синтаксических конструкций (и с той же семантикой), что и работа со временными, существующими только во время работы программы объектами.

Эта сторона ООБД наиболее близка родственному направлению языков программирования баз данных. Языки программирования ООБД и БД во многих своих чертах различаются только терминологически; существенным отличием является лишь поддержание в языках первого класса подхода к наследованию классов. Кроме того, языки второго класса, как правило, более развиты как в отношении системы типов, так и в отношении управляющих конструкций.

Другим аспектом языкового окружения ООБД является потребность в языках запросов, которые можно было бы использовать в интерактивном режиме. Если доступ к объектам внешней БД в языках программирования ООБД носит в основном навигационный характер, то для языков запросов более удобен декларативный стиль. Декларативные языки запросов к ООБД менее развиты, чем языки программирования ООБД, и при их реализации возникают существенные проблемы. В следующем разделе мы рассмотрим имеющиеся подходы и их ограничения более подробно. Но начнем с языков программирования ООБД.

22.3.2. Языки программирования ООБД как объектно-ориентированные языки с поддержкой стабильных (persistent) объектов.

К настоящему моменту нам неизвестен какой-либо язык программирования ООБД, который был бы спроектирован целиком заново, начиная с нуля. Естественным подходом к построению такого языка было использование (с необходимыми расширениями) некоторого существующего объектно-ориентированного языка. Начало расцвета направления ООБД совпало с пиком популярности языка Smalltalk-80. Этот язык оказал большое влияние на разработку первых систем ООБД, и, в частности, использовался в качестве языка программирования. Во многом опирается на Smalltalk и известная коммерчески доступная система GemStone.

Трудности с эффективной практической реализацией языка Smalltalk побудили разработчиков систем ООБД к поиску альтернативных базовых языков. Известная близость объектно-ориентированного и функционального подходов к программированию позволяет достаточно успешно опираться на функциональные языки программирования. В частности, язык Лисп (Common Lisp) является основой проекта ORION. В этом проекте Лисп является и инструментальным языком, и базой объектно-ориентированного языка программирования в среде ORION.

Потребности в еще более эффективной реализации заставляют использовать в качестве основы объектно-ориентированного языка языки более низкого уровня. Например, в системе VBASE

наряду со специально разработанным языком TDL, предназначенным для определения типов, используется объектно-ориентированное расширение языка Си - COP (C Object Processor). В уже упоминавшемся проекте O2 наряду с функциональным объектно-ориентированным языком программирования используются два объектно-ориентированных расширения языков Бейсик и Си. При этом, насколько можно судить по публикациям, наибольшее распространение среди пользователей этой системы (она уже коммерчески доступна) получил язык CO2, являющийся расширением языка Си. Возможно это связано лишь с широкой (и все более возрастающей) популярностью языка Си (и его объектно-ориентированного потомка Си++), ставшего поистине девизом "настоящих программистов". Может быть причины более глубинны (например, языки более высокого уровня слишком ограничительны для программистов-профессионалов; недаром большинство современных реализаций языков более высокого уровня выполняются именно на языке Си). Тем не менее, современная ситуация именно такова, и мы считаем полезным привести краткое описание основных особенностей языка CO2.

22.3.3. Примеры языков программирования ООБД.

Прежде всего, CO2 не является полностью самостоятельным языком. Этот язык входит в многоязыковую среду O2 и предназначен для программирования методов ранее определенных классов. Определение классов, сигнатур методов (фактически, прототипов функций в терминологии языка Си) и имен постоянно хранимых значений и объектов производится с использованием отдельного языка определения схемы БД.

Имя любого объекта трактуется как указатель на значение этого объекта; разименование производится с помощью обычного оператора Си '*'. Доступ к значению объекта возможен только из метода его класса, если только при перечислении методов оператор '*' не объявлен явно публичным.

Поддерживается операция порождения нового объекта указанного класса. В отличие от языка Си++ в CO2 невозможно совместить создание нового объекта с его инициализаций (понятие метода-конструктора начального значения объекта в CO2 не поддерживается). Для инициализации необходимо либо явно обратиться к соответствующему методу класса с указанием вновь созданного объекта (поддерживается соответствующий механизм "передачи сообщений", означающий на самом деле вызов функции), либо воспользоваться оператором '*' и явно присвоить новое значение, если '*' - публичный оператор для данного класса.

CO2 включает средства конструирования значений-кортежей, множеств и списков. Понятие значения-кортежа фактически эквивалентно понятию значения-структуры обычного языка Си (с тем отличием, что элементами кортежа могут являться объекты, множества и списки). Для значений-множеств и списков поддерживаются операции добавления и изъятия элементов, а также набор теоретико-множественных операций (и конкатенации для списков).

Основой манипулирования объектами, хранимыми в БД, является расширенное по сравнению с языком Си средство итерации. Итератор применим к значениям-множествам или спискам. Фактически он означает последовательное применение оператора-тела цикла ко всем элементам множества или списка. Если мы вспомним, что долговременно хранимому классу объектов неявно соответствуют одноименное значение-множество с элементами-объектами данного класса, то становится понятно, что итератор языка CO2 обеспечивает явную навигацию в классах объектов. Единственное, что остается от привычных пользователям СУБД языков запросов, - это ограниченная возможность указания характеристик требуемых в цикле объектов (это делается путем использования оператора разименования и явного указания условий на атрибуты; конечно, для этого нужно, чтобы оператор '*' был объявлен публичным в данном классе).

Разработчики O2 подчеркивают, что они умышленно сделали CO2 более бедным по возможностям, чем, например, язык Си++, потому что многое по части управления объектами берет на себя общий менеджер объектов системы, явно вызываемый из рабочей программы.

22.4. Языки запросов объектно-ориентированных баз данных.

Потребность в поддержании в объектно-ориентированной СУБД не только языка (или семейства языков) программирования ООБД, но и развитого языка запросов в настоящее время

осознается практически всеми разработчиками. Система должна поддерживать легко осваиваемый интерфейс, прямо доступный конечному пользователю в интерактивном режиме.

22.4.1. Явная навигация как следствие преодоления потери соответствия.

Наиболее распространенный подход к организации интерактивных интерфейсов с объектно-ориентированными системами баз данных основывается на использовании обходчиков. В этом случае конечный интерфейс обычно является графическим. На экране отображается схема (или подсхема) ООБД, и пользователь осуществляет доступ к объектам в навигационном стиле. Некоторые исследователи считают, что в этом случае разумно игнорировать принцип инкапсуляции объектов и предъявлять пользователю внутренность объектов. В большинстве существующих систем ООБД подобный интерфейс существует, но всем понятно, что навигационный язык запросов - это в некотором смысле шаг назад по сравнению с языками запросов даже реляционных систем. Ведутся активные поиски подходов к организации декларативных языков запросов к ООБД.

22.4.2. Ненавигационные языки запросов.

Беери отмечает существование трех подходов. Первый подход - языки, являющиеся объектно-ориентированными расширениями языков запросов реляционных систем. Наиболее распространены языки с синтаксисом, близким к известному языку SQL. Это связано, конечно, с общим признанием и чрезвычайно широким распространением этого языка. В частности, в своем Манифесте третьего поколения СУБД М. Стоунбрекер и его коллеги по комитету перспективных систем БД утверждают необходимость поддержания SQL-подобного интерфейса во всех СУБД следующего поколения. Мы уже видели, какое влияние оказывает эта точка зрения на развитие языка SQL.

Второй подход основывается на построении полного логического объектно-ориентированного исчисления. По поводу построения такого исчисления имеются теоретические работы, но законченный и практически реализованный язык запросов нам неизвестен. Видимо к этому же направлению строго теоретически обоснованных языков запросов можно отнести и работы, основанные на алгебраической теории категорий.

Наконец, третий подход основывается на применении дедуктивного подхода. В основном это отражает стремление разработчиков к сближению направлений дедуктивных и объектно-ориентированных БД.

Независимо от применяемого для разработки языка запросов подхода перед разработчиками встает одна концептуальная проблема, решение которой не укладывается в традиционное русло объектно-ориентированного подхода. Понятно, что основой для формулирования запроса должен служить класс, представляющий в ООБД множество однотипных объектов. Но что может представлять собой результат запроса? Набор основных понятий объектно-ориентированного подхода не содержит подходящего к данному случаю понятия. Обычно из положения выходят, расширяя базовый набор концепций множества объектов и полагая, что результатом запроса является некоторое подмножество объектов-экземпляров класса. Это довольно ограничительный подход, поскольку автоматически исключает возможность наличия в языке запросов средств, аналогичных реляционному оператору соединения. Кратко рассмотрим особенности нескольких конкретных декларативных языков запросов к ООБД.

В языке запросов объектно-ориентированной СУБД ORION полностью поддерживается принцип инкапсуляции объектов. В реализованном варианте языка запросы могут основываться только на одном классе (предлагался подход к определению запроса на нескольких классах в стиле расширения семантики реляционного оператора соединения). Синтаксис языка ориентирован на SQL. Очень развит набор допустимых предикатов селекции. В частности, для атрибута, доменом которого является суперкласс, можно указать имя интересующего пользователя подкласса.

Язык запросов системы Iris находится в значительной степени под влиянием реляционной парадигмы. Даже название этого языка OSQL отражает его тесную связь с реляционным языком SQL. По сути дела, OSQL - это реляционный язык, рассчитанный на работу с ненормализован-

ными отношениями. Естественно, при таком подходе в OSQL нарушается инкапсуляция объектов.

На наш взгляд, особый интерес представляет декларативный язык запросов системы O2 RELOOP. В общих словах, это декларативный язык запросов с SQL-ори-ентированным синтаксисом, основанный на специально разработанной для модели O2 алгебре объектов и значений. (Кстати, это не единственная работа в направлении построения алгебры для объектно-ориентированных моделей данных.) Особенно впечатляющим качеством языка RELOOP является естественность его построения в общем контексте модели O2. Запрос задается всегда на значении-множестве или списке. Если мы вспомним, что долговременному классу в O2 соответствует одноименное значение-множество, то тем самым можно определить запрос на любом хранимом классе. Результатом запроса может являться объект, значение-множество или значение-список. При этом элементами значений-множеств могут являться объекты (простая выборка), либо значения-кортежи с элементами-объектами разных классов (например). В совокупности эти особенности языка позволяют формулировать запросы над несколькими классами (специфическое соединение, порождающее не новые объекты, а кортежи из существующих объектов), а также употреблять вложенные подзапросы.

22.4.3. Проблемы оптимизации запросов.

Как обычно, основной целью оптимизации запроса в системе ООБД является создание оптимального плана выполнения запроса с использованием примитивов доступа к внешней памяти ООБД.

Оптимизация запросов хорошо исследована и разработана в контексте реляционных БД. Известны методы синтаксической и семантической оптимизации на уровне непроцедурного представления запроса, алгоритмы выполнения элементарных реляционных операций, методы оценок стоимости планов запросов.

Конечно, объекты могут иметь существенно более сложную структуру, чем кортежи плоских отношений, но не это различие является наиболее важным. Основная сложность оптимизации запросов к ООБД следует из того, что в этом случае условия выборки формулируются в терминах "внешних" атрибутов объектов (методов), а для реальной оптимизации (т.е. для выработки оптимального плана) требуются условия, определенные на "внутренних" атрибутах (переменных состояния).

На самом деле похожая ситуация существует и в РСУБД при оптимизации запроса над представлением БД. В этом случае условия также формулируются в терминах внешних атрибутов (атрибутов представления), и в целях оптимизации запроса эти условия должны быть преобразованы в условия, определенные на атрибутах хранимых отношений. Хорошо известным методом такой "предоптимизации" является подстановка представлений, которая часто (хотя и не всегда в случае использования языка SQL) обеспечивает требуемые преобразования. Альтернативным способом выполнения запроса над представлением (иногда единственным возможным) является материализация представления.

В системах ООБД ситуация существенно усложняется двумя обстоятельствами. Во-первых, методы обычно программируются на некотором процедурном языке программирования и могут иметь параметры. Т.е. в общем случае тело метода представляет из себя не просто арифметическое выражение, как в случае определения атрибутов представления, а параметризованную программу, включающую ветвления, вызовы функций и методов других объектов. Вторая сложность связана с возможным и распространенным в ООП поздним связыванием: точная реализация метода и даже структура объекта может быть неизвестна во время компиляции запроса.

Одним из подходов к упрощению проблемы является открытие видимости некоторых (наиболее важных для оптимизации) внутренних атрибутов объектов. В этом контексте достаточно было бы открыть видимость только для компилятора запросов, т.е. фактически запретить переопределять такие переменные в подклассах. С точки зрения пользователя такие атрибуты выглядели бы как методы без параметров, возвращающие значение соответствующего типа. С нашей точки зрения лучше было бы сохранить строгую инкапсуляцию объектов (чтобы избавить при-

ложение от критической зависимости от реализации) и обеспечить возможности тщательного проектирования схемы ООБД с учетом потребностей оптимизации запросов.

Общий подход к предоптимизации условия выборки для одного (супер)класса объектов может быть следующим (мы предполагаем, что условия формулируются с использованием логики предикатов первого порядка без кванторов; в предикатах могут использоваться методы соответствующего класса, константы и операции сравнения):

Шаг A: Преобразовать логическую формулу условия к конъюнктивной нормальной форме (КНФ). Мы не останавливаемся на способе выбора конкретной КНФ, но естественно, должна быть выбрана "хорошая" КНФ (например, содержащая максимальное число атомарных конъюнктов).

Шаг B: Для каждого конъюнкта, включающего методы с известным во время компиляции телом, заменить вызовы методов на их тела с подставленными параметрами. (Для простоты будем предполагать, что параметры не содержат вызовов функций или методов других объектов.)

Шаг C: Для каждого такого конъюнкта произвести все возможные упрощения, т.е. вычислить все, что можно вычислить в статике. Хотя в общем виде эта задача является очень сложной, при разумном проектировании ООБД в число методов должны будут войти методы с предельно простой реализацией, задавать условия на которых будет очень естественно. Такие условия будут упрощаться очень эффективно.

Шаг D: Если теперь появились конъюнкты, представляющие собой простые предикаты сравнения на основе переменных состояния и констант, использовать эти конъюнкты для выработки оптимального плана выполнения запроса. Если же такие конъюнкты получить не удалось, единственным способом "отфильтровать" (супер)класс объектов является его последовательный просмотр с полным вычислением (возможно упрощенного) логического выражения для каждого объекта.

Понятно, что возможности оптимизации будут зависеть от особенностей языка программирования, который используется для программирования методов, от особенностей конкретного языка запросов и от того, насколько продуманно спроектирована схема ООБД. В частности, желательно, чтобы используемый язык программирования стимулировал максимально дисциплинированный стиль программирования методов объектов. Язык запросов должен разумно ограничивать возможности пользователей (в частности, в отношении параметров методов, участвующих в условиях запросов). Наконец, в классах схемы ООБД должны содержаться простые методы, не переопределяемые в подклассах и основанные на тех переменных состояния, которые служат основой для организации методов доступа.

Заметим, что указанные ограничения не влекут зависимости прикладной программы от особенностей реализации ООБД, поскольку объекты остаются полностью инкапсулированными. Использование в условиях запросов простых методов должно стимулироваться не требованиями реализации, а семантикой объектов.

22.5. Примеры объектно-ориентированных СУБД.

В настоящее время ведется очень много экспериментальных и производственных работ в области объектно-ориентированных СУБД. Больше всего университетских работ, которые в основном носят исследовательский характер. Но уже несколько лет назад отмечалось существование по меньшей мере тринадцати коммерчески доступных систем ООБД. Среди них уже упоминавшиеся в нашем обзоре системы O2, ORION, GemStone и Iris.

Рассмотрим особенности организации двух из них - ORION и O2.

22.5.1. Проект ORION.

Проект ORION осуществлялся с 1985 по 1989 г. фирмой MCC под руководством известного еще по работам в проекте System R Вона Кима. Под названием ORION на самом деле скрывается семейство трех СУБД: ORION-1 - однопользовательская система; ORION-1SX, предназначенная для использования в качестве сервера в локальной сети рабочих станций; ORION-2 – полностью распределенная объектно-ориентированная СУБД. Реализация всех систем производилась с использованием языка Common Lisp на рабочих станциях (и их локальных сетях) Symbolics 3600 с ОС Genera 7.0 и SUN-3 в среде ОС UNIX.

Основными функциональными компонентами системы являются подсистемы управления памятью, объектами и транзакциями. В ORION-1 все компоненты, естественно, располагаются на одной рабочей станции; в ORION-1SX - разнесены между разными рабочими станциями (в частности, управление объектами производится на рабочей станции-клиенте). Применение в ORION-1SX для взаимодействия клиент-сервер механизма удаленного вызова процедур позволило использовать в этой системе практически без переделки многие модули ORION-1. Сетевые взаимодействия основывались на стандартных средствах операционных систем.

В число функций подсистемы управления памятью входит распределение внешней памяти, перемещение страниц из буферов оперативной памяти во внешнюю память и наоборот, поиск и размещение объектов в буферах оперативной памяти (как принято в объектно-ориентированных системах, поддерживаются два представления объектов - дисковое и в оперативной памяти; при перемещении объекта из буфера страниц в буфер объектов и обратно представление объекта изменяется). Кроме того, эта подсистема ответственна за поддержание вспомогательных индексных структур, предназначенных для ускорения выполнения запросов.

Подсистема управления объектами включает подкомпоненты обработки запросов, управления схемой и версиями объектов. Версии поддерживаются только для объектов, при создании которых такая необходимость была явно указана. Для схемы БД версии не поддерживаются; при изменении схемы отслеживается влияние этого изменения на другие компоненты схемы и на существующие объекты. При обработке запросов используется техника оптимизации, аналогичная применяемой в реляционных системах (т.е. формируется набор возможных планов выполнения запроса, оценивается стоимость каждого из них и выбирается для выполнения наиболее дешевый).

Подсистема управления транзакциями обеспечивает традиционную сериализуемость транзакций, а также поддерживает средства журнализации изменений и восстановления БД после сбоев. Для сериализации транзакций применяется разновидность двухфазного протокола синхронизационных захватов с различной степенью гранулированности. Конечно, при синхронизации учитывается специфика ООБД, в частности, наличие иерархии классов. Журнал изменений обеспечивает откаты индивидуальных транзакций и восстановление БД после мягких сбоев (архивные копии БД для восстановления после поломки дисков не поддерживаются).

22.5.2. Проект O2.

Проект O2 выполнялся французской компанией Altair, образованной специально для целей проектирования и реализации объектно-ориентированной СУБД. Начало проекта датируется сентябрем 1986 г., и он был рассчитан на пять лет: три года на прототипирование и два года на разработку промышленного образца. После успешного завершения проекта для сопровождения системы и ее дальнейшего развития была организована новая чисто коммерческая компания O2.

Прототип системы функционировал в режиме клиент/сервер в локальной сети рабочих станций SUN с соответствующим разделением функций между сервером и клиентами.

Основными компонентами системы (не считая развитого набора интерфейсных средств) являются интерпретатор запросов и подсистемы управления схемой, объектами и дисками. Управление дисками, т.е. поддержание базовой среды постоянного хранения обеспечивает система WiSS, которую разработчики O2 перенесли в окружение ОС UNIX.

Наибольшую функциональную нагрузку несет компонент управления объектами. В число функций этой подсистемы входят:

- управление сложными объектами, включая создание и уничтожение объектов, выборку объектов по именам, поддержку предопределенных методов, поддержку объектов со внутренней структурой-множеством, списком и кортежем;
- управление передачей сообщений между объектами;
- управление транзакциями;
- управление коммуникационной средой (на базе транспортных протоколов TCP/IP в локальной сети Ethernet);

- отслеживание долговременно хранимых объектов (напомним, что в O2 объект хранится во внешней памяти до тех пор, пока достижим из какого-либо долговременно хранимого объекта);
- управление буферами оперативной памяти (аналогично ORION, представление объекта в оперативной памяти отличается от его представления на диске);
- управление кластеризацией объектов во внешней памяти;
- управление индексами.

Несколько слов про управление транзакциями. Различаются режимы, когда допускается параллельное выполнение транзакций, изменяющих схему БД, и когда параллельно выполняются только транзакции, изменяющие внутренность БД. Первый режим обычно используется на стадии разработки БД, второй - на стадии выполнения приложений. Средства восстановления БД после сбоев и откатов транзакций также могут включаться и выключаться. Наконец, поддерживается режим, при котором все постоянно хранимые объекты загружаются в оперативную память при начале транзакции для увеличения скорости работы прикладной системы.

Компонент управления схемой БД реализован над подсистемой управления объектами: в системе поддерживаются несколько невидимых для программистов классов и в том числе классы "Class" и "Method", экземплярами которых являются, соответственно, объекты, определяющие классы, и объекты, определяющие методы. (Как видно, ситуация напоминает реляционные системы, в которых тоже обычно поддерживаются служебные отношения-каталоги, описывающие схему БД.) Удаление класса, который не является листом иерархии классов или используется в другом классе или сигнатуре какого-либо метода, запрещено.

Даже приведенное краткое описание особенностей двух объектно-ориентированных СУБД показывает прагматичность современного подхода к организации таких систем. Их разработчики не стремятся к полному соблюдению чистоты объектно-ориентированного подхода и применяют наиболее простые решения проблем. Пока в сообществе разработчиков объектно-ориентированных систем БД не видно работы, которая могла бы сыграть в этом направлении роль, аналогичную роли System R по отношению к реляционным системам. Правда и проблемы ООБД гораздо более сложны, чем решаемые в реляционных системах.

§23. Системы баз данных, основанные на правилах.

В этой очень краткой лекции мы рассмотрим последнюю тему этого курса - системы баз данных, основанные на правилах. Более точно можно было бы сказать, что наша завершающая лекция посвящается системам баз данных, в которых правила играют существенно более важную роль, чем в традиционных реляционных системах. Это уточнение необходимо по той причине, что правила используются для разных целей в любой развитой СУБД.

23.1. Экстенсиональная и интенсиональная части базы данных.

Если внимательно присмотреться к тому, что реально хранится в базе данных, то можно заметить наличие трех различных видов информации. Во-первых, это информация, характеризующая структуры пользовательских данных (описание структурной части схемы базы данных). Такая информация в случае реляционной базы данных сохраняется в системных отношениях-каталогах и содержит главным образом имена базовых отношений и имена и типы данных их атрибутов. Во-вторых, это собственно наборы кортежей пользовательских данных, сохраняемых в определенных пользователями отношениях. Наконец, в-третьих, это правила, определяющие ограничения целостности базы данных, триггеры базы данных и представляемые (виртуальные) отношения. В реляционных системах правила опять же сохраняются в системных таблицах-каталогах, хотя плоские таблицы далеко не идеально подходят для этой цели.

Информация первого и второго вида в совокупности явно описывает объекты (сущности) реального мира, моделируемые в базе данных. Другими словами, это явные факты, предоставленные пользователями для хранения в БД. Эту часть базы данных принято называть экстенсиональной.

Информация третьего вида служит для руководства СУБД при выполнении различного рода операций, задаваемых пользователями. Ограничения целостности могут блокировать выполнение операций обновления базы данных, триггеры вызывают автоматическое выполнение специфицированных действий при возникновении специфицированных условий, определения представлений вызывают явную или косвенную материализацию представляемых таблиц при их использовании. Эту часть базы данных принято называть интенсиональной; она содержит не непосредственные факты, а информацию, характеризующую семантику предметной области.

Как видно, в реляционных базах данных наиболее важное значение имеет экстенсиональная часть, а интенсиональная часть играет в основном вспомогательную роль. В системах баз данных, основанных на правилах, эти две части как минимум равноправны.

23.2. Активные базы данных.

По определению БД называется активной, если СУБД по отношению к ней выполняет не только те действия, которые явно указывает пользователь, но и дополнительные действия в соответствии с правилами, заложенными в саму БД.

Легко видеть, что основа этой идеи содержалась в языке SQL времени System R. На самом деле, что есть определение триггера или условного воздействия, как не введение в БД правила, в соответствии с которым СУБД должна производить дополнительные действия? Плохо лишь то, что на самом деле триггеры не были полностью реализованы ни в одной из известных систем, даже и в System R. И это не случайно, потому что реализация такого аппарата в СУБД очень сложна, накладна и не полностью понятна.

Среди вопросов, ответы на которые до сих пор не получены, следующие. Как эффективно определить набор вспомогательных действий, вызываемых прямым действием пользователя? Каким образом распознавать циклы в цепочке "действие-условие-действие-..." и что делать при возникновении таких циклов? В рамках какой транзакции выполнять дополнительные условные действия и к бюджету какого пользователя относить возникающие накладные расходы?

Масса проблем не решена даже для сравнительно простого случая реализации триггеров SQL, а задача ставится уже гораздо шире. По существу, предлагается иметь в составе СУБД продукционную систему общего вида, условия и действия которой не ограничиваются содержимым БД или прямыми действиями над ней со стороны пользователя. Например, в условие может входить время суток, а действие может быть внешним, например, вывод информации на экран оператора. Практически все современные работы по активным БД связаны с проблемой эффективной реализации такой продукционной системы.

Вместе с тем, по нашему мнению, гораздо важнее в практических целях реализовать в реляционных СУБД аппарат триггеров. Заметим, что в проекте стандарта SQL3 предусматривается существование языковых средств определения условных воздействий. Их реализация и будет первым практическим шагом к активным БД (уже появились соответствующие коммерческие реализации).

23.3. Дедуктивные базы данных.

По определению, дедуктивная БД состоит из двух частей: экстенциональной, содержащей факты, и интенциональной, содержащей правила для логического вывода новых фактов на основе экстенциональной части и запроса пользователя.

Легко видеть, что при таком общем определении SQL-ориентированную реляционную СУБД можно отнести к дедуктивным системам. Действительно, что есть определенные в схеме реляционной БД представления, как не интенциональная часть БД. В конце концов не так уж важно, какой конкретный механизм используется для вывода новых фактов на основе существующих. В случае SQL основным элементом определения представления является оператор выборки языка SQL, что вполне естественно, поскольку результатом оператора выборки является порождаемая таблица. Обеспечивается и необходимая расширяемость, поскольку представления могут определяться не только над базовыми таблицами, но и над представлениями.

Основным отличием реальной дедуктивной СУБД от реляционной является то, что и правила интенциональной части БД, и запросы пользователей могут содержать рекурсию. Можно спорить о том, всегда ли хороша рекурсия. Однако возможность определения рекурсивных пра-

вил и запросов дает возможность простого решения в дедуктивных базах данных проблем, которые вызывают большие проблемы в реляционных системах (например, проблемы разборки сложной детали на примитивные составляющие). С другой стороны, именно возможность рекурсии делает реализацию дедуктивной СУБД очень сложной и во многих случаях неразрешимой эффективно проблемой.

Мы не будем здесь более подробно рассматривать конкретные проблемы, применяемые ограничения и используемые методы в дедуктивных системах. Отметим лишь, что обычно языки запросов и определения интенциональной части БД являются логическими (поэтому дедуктивные БД часто называют логическими). Имеется прямая связь дедуктивных БД с базами знаний (интенциональную часть БД можно рассматривать как БЗ). Более того, трудно провести грань между этими двумя сущностями; по крайней мере, общего мнения по этому поводу не существует.

Какова же связь дедуктивных БД с реляционными СУБД, кроме того, что реляционная БД является вырожденным частным случаем дедуктивной? Основным является то, что для реализации дедуктивной СУБД обычно применяется реляционная система. Такая система выступает в роли хранителя фактов и исполнителя запросов, поступающих с уровня дедуктивной СУБД. Между прочим, такое использование реляционных СУБД резко актуализирует задачу глобальной оптимизации запросов.

При обычном применении реляционной СУБД запросы обычно поступают на обработку по одному, поэтому нет повода для их глобальной (межзапросной) оптимизации. Дедуктивная же СУБД при выполнении одного запроса пользователя в общем случае генерирует пакет запросов к реляционной СУБД, которые могут оптимизироваться совместно.

Конечно, в случае, когда набор правил дедуктивной БД становится велик, и их невозможно разместить в оперативной памяти, возникает проблема управления их хранением и доступом к ним во внешней памяти. Здесь опять же может быть применена реляционная система, но уже не слишком эффективно. Требуются более сложные структуры данных и другие условия выборки. Известны частные попытки решить эту проблему, но общего решения пока нет.

Содержание