1. **Файловые системы. Особенности организации устройств внешней памяти на магнитных дисках. Структуры файлов на дисках. Способы организации архивов файлов. Принципы именования.**

С точки зрения программиста приложений **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 внешней памяти и обеспечение доступа к данным.

Термин ***файловая система*** (*file system*) используется для обозначения как программной системы, управляющей файлами, так и архива файлов, хранящегося во внешней памяти.

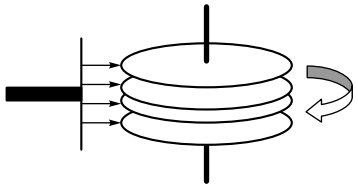
В первые десятилетия развития вычислительной техники использовались два вида устройств внешней памяти: магнитные ленты и магнитные барабаны. **Емкость магнитных лент** была достаточно **велика**, но по своей природе они обеспечивали **только последовательный доступ** к данным. **Емкость магнитной ленты пропорциональна ее длине.** Чтобы получить доступ к требуемой порции данных, нужно в среднем перемотать половину ее длины. Но чисто механическую операцию перемотки нельзя выполнить очень быстро. Поэтому быстрый произвольный доступ к данным на магнитной ленте, очевидно, невозможен.

Магнитный барабан представлял собой массивный металлический цилиндр с намагниченной внешней поверхностью и неподвижным пакетом магнитных головок. Такие устройства обеспечивали возможность достаточно быстрого **произвольного доступа к данным**, но позволяли сохранять сравнительно **небольшой объем данных**. Быстрый произвольный доступ осуществлялся благодаря высокой скорости вращения барабана и наличию отдельной головки на каждую дорожку магнитной поверхности; ограниченность объема была обусловлена наличием всего одной магнитной поверхности.

Именно требования к устройствам внешней памяти со стороны бизнес-приложений вызвали появление устройств внешней памяти со съемными пакетами магнитных дисков и подвижными головками чтения/записи. Эти устройства памяти обладали существенно большей емкостью, чем магнитные барабаны (за счет наличия нескольких магнитных поверхностей), обеспечивали удовлетворительную скорость доступа к данным в режиме произвольной выборки, а возможность смены дискового пакета на устройстве позволяла иметь архив данных практически неограниченного объема.

Магнитные диски представляют собой пакеты магнитных пластин (поверхностей), между которыми на одном рычаге движется пакет магнитных головок. Шаг движения пакета головок является дискретным, и каждому положению пакета головок логически соответствует цилиндр пакета магнитных дисков. 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, предельная емкость каждого блока составляет одно и то же число байтов. 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.



При выполнении обмена с диском аппаратура выполняет три основных действия:

• подвод головок к нужному цилиндру (обозначим время выполнения этого действия как $t_{пг}$);

• поиск на дорожке нужного блока (время выполнения — $t_{пб}$);

• собственно обмен с этим блоком (время выполнения — $t_{об}$). Тогда, как правило, $t_{пг} >> t_{пб} >> t_{об}$, потому что подвод головок — это механическое действие, причем в среднем нужно переместить головки на расстояние, равное половине радиуса поверхности, а скорость передвижения головок не может быть слишком большой по физическим соображениям. Поиск блока на дорожке требует прокручивания пакета магнитных дисков в среднем на половину длины внешней окружности; скорость вращения диска может быть существенно больше скорости дви- жения головок, но она тоже ограничена законами физики. Для выполнения же обмена нужно прокрутить пакет дисков всего лишь на угловое расстояние, соответствующее размеру блока. Таким образом, из всех этих действий в среднем наибольшее время занимает первое, и поэтому существенный выигрыш в суммарном времени обмена при считывании или записи только части блока получить практически невозможно.

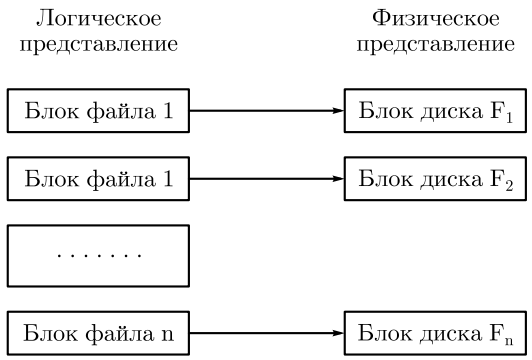
Практически во всех современных компьютерах основными устройствами внешней памяти являются магнитные диски с подвижными головками, и именно они служат для хранения файлов.

Аппаратура магнитных дисков допускает выполнение обмена с дисками порциями данных произвольного размера. Однако возможность обмениваться с магнитными дисками порциями, размеры которых меньше полного объема блока, в настоящее время в файловых системах не используется. Это связано с двумя обстоятельствами.

Во-первых, считывание или запись только части блока не приводит к существенному выигрышу в суммарном времени обмена, поскольку основная часть этого времени тратится на перемещение магнитных головок. Во-вторых, для работы с частями блоков файловая система должна обеспечить буферы оперативной памяти соответствующего размера, что существенно усложняет распределение основной памяти. Алгоритмы распределения памяти порциями произвольного размера плохи тем, что любой из них рано или поздно приводит к *внешней фрагментации* памяти. В памяти образуется большое число мелких свободных фрагментов. Их совокупный размер может быть больше размера любого требуемого буфера, но его можно выделить, только если произвести сжатие памяти, т.е. подвижку всех занятых фрагментов таким образом, чтобы они располагались вплотную один к другому. Во время выполнения операции

сжатия памяти нужно приостановить выполнение обменов, а сама эта операция занимает много времени.

Поэтому во всех современных файловых системах явно или неявно выделяется уровень, обеспечивающий работу с ***базовыми файлами*,** которые представляют собой наборы блоков, последовательно нумеруемых в адресном пространстве файла и отображаемых на физические блоки диска. Размер логического блока файла совпадает с размером физического блока диска или кратен ему; обычно размер логического блока выбирается равным размеру страницы виртуальной памяти, поддерживаемой аппаратурой компьютера совместно с операционной системой.



Исторически существует два основных подхода. При первом подходе, свойственном, например, файловой системе операционной системы компании Hewlett-Packard OpenVMS, пользователи представляют **файл как последовательность записей**. Каждая запись — это последовательность байтов, имеющая постоянный или переменный размер. Можно читать или писать записи последовательно либо позиционировать файл на запись с указанным номером.

В некоторых файловых системах допускается **структуризация записей на поля и объявление указываемых полей ключами записи**. В таких файловых системах можно потребовать выборку записи из файла по ее заданному ключу. Естественно, в этом случае файловая система поддерживает в том же (или другом, служебном) базовом файле дополнительные, невидимые пользователю, служебные структуры данных — *индексы*. Распространенные способы организации индексов ключевых файлов основаны на технике *хэширования* и *B-деревьев*. Существуют и многоключевые способы организации файлов (у одного файла объявляется несколько ключей, и можно выбирать записи по значению каждого ключа).

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

Во всех современных файловых системах обеспечивается **многоуровневое именование файлов** за счет наличия во внешней памяти **каталогов** — дополнительных файлов со

специальной структурой. Каждый каталог содержит имена каталогов и/или файлов, хранящихся 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 любом случае первое имя должно соответствовать корневому каталогу файловой системы. Вопрос заключается в том, как сопоставить этому имени корневой каталог — где его искать? В связи с этим имеются два радикально различных подхода.

Во многих системах управления файлами требуется, чтобы каждый **архив файлов (полное дерево каталогов)** целиком располагался на одном дисковом пакете или логическом диске — разделе физического дискового пакета, логически представляемом в виде отдельного диска с помощью средств операционной системы. В этом случае полное имя файла начинается с имени дискового устройства, на котором установлен соответствующий диск. Такой способ именования использовался в файловых системах компаний IBM и DEC; очень близки к этому и файловые системы, реализованные в операционных системах семейства Windows компании Microsoft. Можно назвать такую организацию поддержкой **изолированных файловых систем.**

Другой крайний вариант был реализован в файловых системах операционной системы Multics. В файловой системе Multics пользователям обеспечивалась возможность представлять всю совокупность каталогов и файлов в виде единого дерева. 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 у оператора установку необходимых дисков. Такую файловую систему можно назвать **полностью централизованной**.

Конечно, во многом централизованные файловые системы удобнее изолированных: система управления файлами выполняет больше рутинной работы. В частности, администратор файловой системы автоматически оповещается о потребности установки требуемых дисковых пакетов; система обеспечивает равномерное распределение памяти на известных ей дисковых томах; возможна организация автоматического перемещения редко используемых файлов на более медленные носители внешней памяти.

Но в таких системах возникают существенные проблемы, если требуется перенести поддерево файловой системы на другую вычислительную установку. Поскольку файлы и каталоги любого логического поддерева могут быть физически разбросаны по разным дисковым пакетам и даже магнитным лентам, для такого переноса требуется специальная утилита, собирающая все объекты требуемого поддерева на одном внешнем носителе, не входящем в состав штатных устройств централизованной файловой системы. Конечно, даже при наличии такой утилиты выполнение процедуры физической сборки требует существенного времени.

Компромиссное решение применяется в файловых системах ОС UNIX. На базовом уровне в этих файловых системах поддерживаются изолированные архивы файлов. Один из таких архивов объявляется корневой файловой системой. Это делается на

этапе генерации операционной системы, и после запуска операционная система «знает», на каком дисковом устройстве (физическом или логическом) располагается корневая файловая система. После запуска системы можно «смонтировать» корневую файловую систему и ряд изолированных файловых систем в одну общую файловую систему. Технически это осуществляется посредством создания в корневой файловой системе специальных пустых каталогов (точек монтирования).

Специальный системный вызов *mount* ОС UNIX позволяет подключить к одному из пустых каталогов корневой каталог указанного архива файлов.

Кроме того, поддерживается системный вызов unmount, «отторгающий» ранее смонтированную файловую систему от общей иерархии. Конечно, все это заметно облегчает перенос частей файловой системы на другие установки.

2. Файловые системы. Способы авторизации доступа к файлам. Организация мультидоступа.

Поскольку файловая система является 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 для каждого зарегистрированного пользователя и для каждого существующего файла файловой системы указываются действия, выполнение которых над данным файлом разрешено или запрещено данному пользователю (так называемый мандатный способ защиты — у каждого пользователя имеется или не имеется отдельный мандат для работы с каждым файлом). Применение мандатного способа авторизации доступа влечет за собой существенные накладные расходы, связанные с потребностью хранения избыточной информации и использованием этой информации для проверки правомочности доступа.

Поэтому в большинстве современных систем управления файлами применяется упрощенный подход к *авторизации* доступа к файлам, традиционно поддерживаемый, например в ОС UNIX (так называемый *дискреционный* подход). При использовании этого подхода с каждым зарегистрированным пользователем связывается пара целочисленных идентификаторов: идентификатор группы, к которой относится пользователь, и его собственный идентификатор. Этими же идентификаторами снабжается каждый процесс, запущенный от имени данного пользователя и имеющий возможность обращаться к системным вызовам файловой системы. Соответственно, при каждом файле хранится полный идентификатор пользователя (собственный идентификатор плюс идентификатор группы), который создал этот файл, и помечается, какие действия с файлом может про- изводить он сам, какие действия с файлом доступны для остальных пользователей той же группы и что могут делать с файлом пользователи других групп. Для каждого файла контролируется возможность выполнения трех действий: чтение, запись и выполнение. Хранимая информация очень компактна (два целых числа для представления идентификаторов и шкала из 9 бит для характеристики возможных действий), при проверке требуется небольшое количество действий, и этот способ контроля доступа в большинстве случаев удовлетворителен.

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.

В файловых системах обычно применяется следующий подход. В операции открытия файла (первой и обязательной операции, с которой должен начинаться сеанс работы с файлом) помимо прочих параметров указывается режим работы (чтение или изменение). Если к моменту выполнения этой операции от имени некоторого процесса *A* файл уже открыт некоторым другим процессом *B*, причем существующий режим открытия несовместим с требуемым режимом (совместимы только режимы чтения), то в зависимости от особенностей системы либо процессу *A* сообщается о невозможности открытия файла в нужном режиме, либо процесс *A* блокируется до тех пор, пока процесс *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.

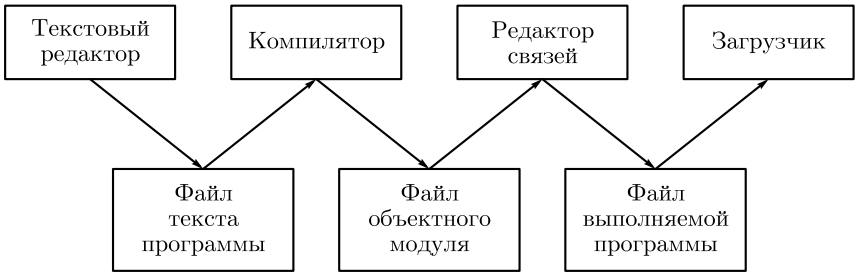
3. Области применения файловых систем. Требования к базам данных со стороны информационных систем: согласованность данных, языки запросов, восстановление согласованного состояния после сбоев, реальный режим мультидоступа.

Чаще всего файлы используются для хранения текстовых данных: документов, текстов программ и т. д. Такие файлы обычно создаются и модифицируются с помощью различных текстовых редакторов. Обычно структура текстовых файлов очень проста (с точки зрения файловой системы): это либо последовательность записей, содержащих строки текста, либо последовательность байтов, среди которых встречаются специальные символы (например, символы конца строки).

Файлы, содержащие тексты программ, используются как входные файлы компиляторов (чтобы правильно воспринять текст программы, компилятор должен понимать логическую структуру текстового файла), 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 такую структуру более сложную и специфичную для этой системы логическую структуру объектного модуля. Подчеркнем, что логическая структура объектного модуля файловой системе неизвестна; эта структура поддерживается инструментами системы программирования.

Одним словом, **файловые системы обычно обеспечивают** хранение данных с очень

«аморфной» стандартной структурой, оставляя дальнейшую структуризацию прикладным программам. В некоторых случаях использования файлов это даже хорошо, поскольку при разработке любой новой прикладной системы, опираясь на простые, стандартные и сравнительно дешевые средства файловой системы, можно реализовать те структуры хранения, которые наиболее точно соответствуют специфике данной прикладной области.



Типовая информационная система, главным образом, ориентирована на хранение, выбор и модификацию данных соответствующей прикладной области. Структура таких данных зачастую очень сложна, и, хотя структуры данных различны в разных информационных системах, между ними часто бывает много общего.

Рис. 1.4. Примитивная схема структуризации данных в информационной системе

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 общую часть информационных систем, ответственную за управление сложно структурированными данными, явилось, на мой взгляд, первой побудительной причиной создания СУБД.

Предположим, что мы хотим организовать информацинную систему, поддерживающую учет служащих в некоторой организации, которая состоит из двух файлов служащие и отделы.

Система должна «знать», что она работает с двумя информационно связанными файлами (это шаг в сторону схемы базы данных), должна иметь информацию о структуре и смысле каждого поля. Например, системе должно быть известно, что у полей СЛУ_ОТД_НОМЕР в файле СЛУЖАЩИЕ и ОТД_НОМЕР в файле ОТДЕЛЫ один и

тот же смысл – номер отдела.

Кроме того, система должна учитывать, что в ряде случаев изменение данных в одном файле должно автоматически вызывать модификацию второго файла, чтобы общее содержимое файлов было согласованным. Например, если на работу принимается новый служащий, то нужно добавить запись в файл СЛУЖАЩИЕ, а также должным образом изменить поля ОТД_СЛУ_ЗАРП и ОТД_РАЗМЕР в записи файла ОТДЕЛЫ, соответствующей отделу этого служащего.

Понятие согласованности, или целостности, данных является ключевым понятием баз данных. Фактически, если информационная система (даже такая простая, как в нашем примере) поддерживает **согласованное хранение данных в нескольких файлах**, можно говорить о том, что она **поддерживает базу данных (БД).** Если же некоторая вспомогательная система управления данными позволяет работать с несколькими файлами, обеспечивая их согласованность, можно назвать ее ***системой управления базами данных* (*СУБД*).**

Уже только требование поддержания согласованности данных в нескольких файлах не позволяет при построении информационной системы обойтись библиотекой функций: такая система должна обладать **некоторыми собственными данными (их принято называть метаданными),** определяющими целостность данных. В нашем примере информационная система должна отдельно сохранять метаданные о структуре файлов СЛУЖАЩИЕ и ОТДЕЛЫ, а также правила, определяющие условия целостности данных в этих файлах (принято считать, что правила также составляют часть метаданных).

Но обеспечение целостности данных – это далеко не все, что обычно требуется от СУБД. Начнем с того, что даже в нашем примере пользователю информационной системы будет не слишком просто получить, например, общую численность отдела, в котором работает Петр Иванович Сидоров. Придется сначала узнать номер отдела, в котором работает указанный служащий, а затем установить численность этого отдела. Было бы гораздо проще, если бы **СУБД позволяла сформулировать такой запрос на языке, более близком пользователям. Такие языки называются *языками запросов к базам данных*.** Например, на языке запросов SQL наш запрос можно было бы выразить в следующей форме (*запрос1*):

```
SELECT ОТД_РАЗМЕР
  FROM СЛУЖАЩИЕ, ОТДЕЛЫ
  WHERE СЛУ_ИМЯ = 'ПЕТР ИВАНОВИЧ СИДОРОВ' AND
    СЛУ_ОТД_НОМЕР = ОТД_НОМЕР;
```

Далее, представим себе, что в первоначальной реализации информационной системы, основанной на использовании библиотек расширенных методов доступа к файлам, обрабатывается операция принятия на работу нового служащего. Следуя требованиям согласованного изменения файлов, информационная система вставляет новую запись в файл СЛУЖАЩИЕ и собирается модифицировать соответствующую запись файла ОТДЕЛЫ (или вставлять в этот файл новую запись, если служащий является первым в своем отделе), но именно в этот момент происходит (например) аварийное выключение

питания компьютера.

Очевидно, что после перезапуска системы ее база данных будет находиться в рассогласованном состоянии. Потребуется выяснить это (а для этого нужно явно проверить соответствие данных в файлах СЛУЖАЩИЕ и ОТДЕЛЫ) и привести данные в согласованное состояние.

Настоящие СУБД берут такую работу на себя, поддерживая транзакционное управление и журнализацию изменений базы данных. Прикладная система не обязана заботиться о поддержке корректности состояния базы данных, хотя и должна знать, какие цепочки операций изменения данных являются допустимыми.

Представим теперь, что в информационной системе требуется обеспечить параллельную (например, многотерминальную) работу с базой данных служащих и отделов. 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. Таким образом, зачисление на работу Петра Ивановича Сидорова существенно затормозит получение информации о служащем Иване Сидоровиче Петрове, даже если они работают в разных отделах. Настоящие СУБД обеспечивают гораздо более тонкую синхронизацию параллельного доступа к данным.

4. Основные функции СУБД, типовая организация СУБД.

1) Непосредственное управление данными во внешней памяти

Эта функция **обеспечивает *поддержку необходимых структур внешней памяти*** как для хранения данных и метаданных, непосредственно входящих в БД, так и для служебных целей, например, для убыстрения доступа к данным (индексы).

В некоторых реализациях СУБД активно используются возможности существующих ФС, в других работа производится на уровне устройств внешней памяти.

Но подчеркнем, что в развитых СУБД пользователи в любом случае не обязаны знать, использует ли СУБД ФС, и, если использует, то как организованы файлы. В частности, в СУБД обычно поддерживается собственная система именования объектов БД.

2) Управление буферами оперативной памяти.

СУБД обычно работают с БД значительного размера; по крайней мере, этот размер обычно существенно больше объема доступной ОП.

Если при обращении к любому элементу данных будет производиться обмен с внешней памятью, то вся система будет работать со скоростью устройства внешней памяти.

Единственным способом реального увеличения этой скорости является *буферизация*

данных в оперативной памяти.

Даже если операционная система производит общесистемную буферизацию данных (как, например, 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 использованием собственной дисциплиной замены буферов.

3) Управление транзакциями.

Транзакция - это последовательность операций над БД, рассматриваемых СУБД как единое целое:

i)либо транзакция успешно выполняется, и СУБД фиксирует (выполняет операцию COMMIT) изменения БД, произведенные этой транзакцией, во внешней памяти,

ii)либо ни одно из этих изменений никак не отражается на состоянии БД.

Понятие транзакции необходимо для поддержания логической целостности БД.

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.

С управлением транзакциями в многопользовательской СУБД связаны важные понятия ***сериализации транзакций*.**

Под **сериализацией** параллельно выполняемых транзакций понимается такой порядок планирования выполнения операций, при котором суммарный эффект смеси транзакций эквивалентен эффекту их некоторого последовательного выполнения.

Понятно, что если удается добиться действительно *сериального* выполнения смеси транзакций, то для каждого пользователя, по инициативе которого образована транзакция, присутствие других транзакций будет незаметно.

Существует несколько базовых алгоритмов сериализации транзакций. Наиболее распространены алгоритмы, основанные на синхронизационных блокировках объектов БД. При использовании любого алгоритма сериализации возможны конфликты между двумя или более транзакциями по доступу к объектам БД. В этом случае для поддержки 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.

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.

Для обеспечения надежного хранения данных в БД требуется хранение избыточных данных, причем та часть данных, которая используется для восстановления БД, 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 (например, можно поддерживать 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 принято придерживаться 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.

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.

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 (*Standard Query Language*). Прежде всего, язык SQL сочетает средства SDL и DML, т.е. позволяет определять схему реляционной БД и манипулировать данными. Именование объектов БД (для реляционной БД – в основном, 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. Ограничения целостности хранятся в специальных таблицах-каталогах, и обеспечение контроля целостности БД производится на языковом уровне, т.е. при первичной обработке 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.

Типовая организация современной СУБД.

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): менеджер данных, менеджер буферов, менеджер транзакций и менеджер журнала.

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, обычно не доступным пользователям напрямую и используемым в программах, производимых компилятором SQL (или в подсистеме поддержки выполнения таких программ) и утилитах БД. При использовании архитектуры «клиент-сервер» ядро является базовой 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 процедурой, а лишь описывает в некоторой форме условия совершения желаемого действия. Поэтому компилятор должен решить, каким образом выполнять оператор языка прежде, чем произвести программу. Применяются достаточно сложные методы оптимизации операторов. 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.

5. Дореляционные модели данных.

В модели данных описывается 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.

В **структурной части** модели данных фиксируются основные логические структуры данных, которые могут применяться на уровне пользователя при организации БД, соответствующих данной модели.

Манипуляционная часть модели данных содержит спецификацию одного или нескольких языков, предназначенных для написания запросов к БД. Эти языки могут быть абстрактными, не обладающими точно проработанным синтаксисом, или законченными производственными языками (как в случае модели данных SQL). Основное назначение манипуляционной части модели данных – обеспечить эталонный «модельный» язык БД, уровень выразительности которого должен поддерживаться в реализациях СУБД, соответствующих данной модели.

Наконец, в **целостной части** модели данных (которая явно выделяется не во всех известных моделях) специфицируются механизмы ограничений целостности, которые обязательно должны поддерживаться во всех реализациях СУБД, соответствующих данной модели.

В целом ранние системы можно охарактеризовать следующим образом:

- Эти системы активно использовались в течение многих лет, задолго до появления работоспособных реляционных СУБД.
- Все ранние системы не основывались на каких-либо абстрактных моделях. 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.
- Навигационная природа ранних систем и доступ к данным на уровне записей заставляли пользователей самих 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.

Модель данных инвертированных таблиц

К числу наиболее известных и типичных представителей систем, в основе которых лежит эта модель данных, относятся СУБД Datacom/DB.

Структуры данных

База данных в модели инвертированных таблиц **похожа на БД в модели SQL, но с тем отличием, что пользователям видны и хранимые таблицы, и пути доступа к ним**. При этом:

- Строки таблиц упорядочиваются системой в некоторой физической, видимой пользователям 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.

Манипулирование данными

Поддерживаются два класса операций:

1. **Операции, устанавливающие адрес записи** и разбиваемые на два подкласса:
   - прямые поисковые операторы (например, установить адрес первой записи таблицы по некоторому пути доступа);
   - операторы, устанавливающие адрес записи при указании относительной позиции от предыдущей записи по некоторому пути доступа.
2. **Операции над адресуемыми записями.**

Вот типичный набор операций:

- `LOCATE FIRST` – найти первую запись таблицы *`T`* в физическом порядке;

возвращается адрес записи;

- `LOCATE FIRST WITH SEARCH KEY EQUAL` – найти первую запись таблицы T с заданным значением ключа поиска k; возвращается адрес записи;
- `LOCATE NEXT` – найти первую запись, следующую за записью с заданным адресом в заданном пути доступа; возвращается 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; возвращается адрес записи;
- `RETRIVE` – выбрать запись с указанным адресом;
- `UPDATE` – обновить запись с указанным адресом;
- `DELETE` – удалить запись с указанным адресом;

Ограничения целостности

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

Иерархическая модель данных

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.

На рис. 2.1 показан пример типа дерева (схемы иерархической БД). Здесь тип записи `Отдел` является предком для типов записи `Руководитель` и `Служащие`. Между типами записи поддерживаются связи (правильнее сказать, *типы* связей, поскольку реальные связи появляются в экземплярах типа дерева).

Рис. 2.1. Пример типа дерева

Все экземпляры данного типа потомка с общим экземпляром типа предка называются близнецами. Для иерархической базы данных определяется полный порядок обхода

дерева: сверху-вниз, слева-направо.

Манипулирование данными

Примерами типичных операций манипулирования иерархически организованными данными могут быть следующие:

- найти указанный экземпляр типа дерева БД (например, отдел 310);
- перейти от экземпляра одного типа записи к экземпляру другого типа записи внутри дерева (например, перейти от отдела к первому сотруднику);
- перейти от одной записи к другой в порядке обхода иерархии;
- вставить новую запись в указанную позицию;
- удалить текущую запись.

Ограничения целостности

В иерархической модели данных автоматически поддерживается целостность ссылок между предками и потомками. Основное правило: ***никакой потомок не может существовать без своего родителя***. Заметим, что аналогичная поддержка целостности по ссылкам между записями без связи «предок-потомок», не обеспечивается.

Сетевая модель данных

Типичным представителем систем, основанных на сетевой модели данных, является СУБД IDMS (Integrated Database Management System).

Сетевые структуры данных

Сетевой подход к организации данных является **расширением иерархического подхода.** В иерархических структурах запись-потомок должна иметь в точности одного предка; в сетевой структуре данных у потомка может иметься 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 записи *P* является предком только в одном экземпляре типа связи *L*;
- каждый экземпляр типа записи *C* является потомком не более чем в одном

экземпляре типа связи *L*.

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

- тип записи потомка в одном типе связи *L1* может быть типом записи предка в другом типе связи *L2* (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 *L1* и *L2* - 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.

На рис. 2.3 показан простой пример схемы сетевой БД. На этом рисунке показаны три типа записи: Отдел, Служащие и Руководитель и три типа связи: Состоит из служащих, Имеет руководителя и Является служащим.

Рис. 2.3. Пример схемы сетевой базы данных

Манипулирование данными

Вот примерный набор операций манипулирования данными:

- найти конкретную запись в наборе однотипных записей (например, служащего с именем Иванов);
- перейти от предка к первому потомку по некоторой связи (например, к первому служащему отдела 625);
- перейти к следующему потомку в некоторой связи (например, от Иванова к Сидорову);
- перейти от потомка к предку по некоторой связи (например, найти отдел, в котором работает Сидоров);
- создать новую запись;
- уничтожить запись;

- модифицировать запись;
- включить в связь;
- исключить из связи;
- переставить в другую связь и т.д.

Ограничения целостности

Имеется (необязательная) возможность потребовать для конкретного типа связи отсутствие потомков, не участвующих ни в одном экземпляре этого типа связи (как в иерархической модели).

6. Основные черты модели данных в SQL.

а) Структура данных

SQL-ориентированная база данных представляет собой **набор таблиц,** каждая из которых в любой момент времени содержит некоторое **мультимножество строк, соответствующих заголовку таблицы**. В этом состоит **первое и наиболее важное отличие модели данных SQL от реляционной** модели данных. Вторым существенным отличием является то, что **для таблицы поддерживается порядок столбцов**, соответствующий порядку их определения. Другими словами, **таблица – это вовсе не отношение**, хотя во многом они похожи.

Имеется две основных разновидности таблиц, хранимых в базе данных: т**радиционная таблица и типизированная таблица**. Традиционная таблица определяется как множество столбцов с указанными типами данных. В SQL поддерживаются следующие категории типов данных: точные числовые типы; приближенные числовые типы; типы символьных строк; типы битовых строк; типы даты и времени; типы временных интервалов; булевский тип; типы коллекций; анонимные строчные типы; типы, определяемые пользователем; ссылочные типы.

При определении типизированной таблицы указывается ранее определенный структурный тип, и если в нем содержится n атрибутов, то в таблице образуется n+1 столбец, из которых последние n столбцов с именами и типами данных, совпадающими именам и типам атрибутов структурного типа. Первый же столбец, имя которого явно задается, называется «самоссылающимся» и содержит типизированные уникальные идентификаторы строк, которые могут генерироваться системой при вставке строк в типизированную таблицу, явно указываться пользователями или состоять из комбинации значений других столбцов. Типом «самоссылающегося» столбца является ссылочный тип, ассоциированный со структурным типом типизированной таблицы. Способ генерации значений ссылочного типа указывается при определении соответствующего структурного типа и подтверждается при определении типизированной таблицы.

Манипулирование данными в SQL

Средства манипулирования данными составляют значительную часть языка SQL. Здесь же мы ограничимся общей характеристикой оператора **SQL SELECT,** предназначенного для выборки данных и имеющего следующий синтаксис:

```
SELECT [ ALL | DISTINCT ] select_item_commalist
FROM table_reference_commalist
[ WHERE conditional_expression ]
[ GROUP BY column_name_commalist ]
[ HAVING conditional_expression ]
[ ORDER BY order_item_commalist ]
```

Выборка данных производится из одной или нескольких таблиц, указываемых в разделе `FROM` запроса. В последнем случае на первом этапе выполнения оператора `SELECT` образуется одна общая таблица, получаемая из исходных таблиц путем применения операции расширенного декартова умножения. Таблицы могут быть как базовыми, реально хранимыми в базе данных (традиционными или типизированными), так и порожденными, т.е. задаваемыми в виде некоторого оператора `SELECT`. Это допускается, поскольку результатом выполнения оператора `SELECT` в его базовой форме является традиционная таблица. Кроме того, в разделе `FROM` можно указывать выражения соединения базовых и/или порожденных таблиц, результатами которых опять же являются традиционные таблицы.

На следующем шаге общая таблица, полученная после выполнения раздела, подвергается фильтрации путем вычисления для каждой ее строки логического выражения, заданного в разделе `WHERE` запроса. В отфильтрованной таблице остаются только те строки общей таблицы, для которых значением логического выражения является *`true`*.

Если в операторе отсутствует раздел `GROUP BY`, то после этого происходит формирование результирующей таблицы запроса путем вычисления выражений, заданных в списке выборки оператора `SELECT.` В этом случае список выборки вычисляется для каждой строки отфильтрованной таблицы, и в результирующей таблице появится ровно столько же строк.

При наличии раздела `GROUP BY` из отфильтрованной таблицы получается сгруппированная таблица, в которой каждая группа состоит из кортежей отфильтрованной таблицы с одинаковыми значениями столбцов группировки, задаваемых в разделе `GROUP BY`. Если в запросе отсутствует раздел `HAVING`, то результирующая таблица строится прямо на основе сгруппированной таблицы. Иначе образуется отфильтрованная сгруппированная таблица, содержащая только те группы, для которых значением логического выражения, заданного в разделе `HAVING`, является *`true`*.

Результирующая таблица на основе сгруппированной или отфильтрованной сгруппированной таблицы строится путем вычисления списка выборки для каждой группы. Тем самым, в результирующей таблице появится ровно столько строк, сколько групп содержалось в сгруппированной или отфильтрованной сгруппированной таблице.

Если в запросе присутствует ключевое слово `DISTINCT`, то из результирующей таблицы устраняются строки-дубликаты, т.е. запрос вырабатывает не мультимножество, а множество строк.

Наконец, в запросе может присутствовать еще и раздел `ORDER BY`. В этом случае результирующая таблица сортируется в порядке возрастания или убывания в соответствии со значениями ее столбцов, указанных в разделе `ORDER BY`. Результатом такого запроса является не таблица, а отсортированный список, который нельзя сохранить в базе данных. Сам же запрос, содержащий раздел `ORDER BY`, нельзя использовать в разделе `FROM` других запросов.

Приведенная характеристика средств манипулирования данными языка SQL является не вполне точной и полной. Кроме того, она отражает *семантику* оператора SQL, а не то, как он обычно исполняется в SQL-ориентированных СУБД.

Ограничения целостности в модели SQL

Как отмечалось в начале этого раздела, наиболее важным отличием модели данных SQL от реляционной модели данных является то, что таблицы SQL могут содержать мультимножества строк. Из этого, в частности, следует, что в **модели SQL отсутствует обязательное предписание об ограничении целостности сущности**. В базе данных могут существовать таблицы, для которых не определен первичный ключ. С другой стороны, если для таблицы определен первичный ключ, то для нее ограничение целостности сущности поддерживается точно так же, как это требуется в реляционной модели данных.

Ссылочная целостность в модели данных SQL поддерживается в обязательном порядке, но в трех разных вариантах, лишь один из которых полностью соответствует реляционной модели. Это связано с уже упоминавшимся в этом разделе интенсивным использованием в SQL неопределенных значений.

Кроме того, в SQL имеются развитые возможности явного определения ограничений целостности на уровне столбцов таблиц, на уровне таблиц целиком и на уровне базы данных.

7. Типы данных, наследование типов в SQL.

В SQL поддерживаются следующие категории типов данных: **точные числовые типы; приближенные числовые типы; типы символьных строк; типы битовых строк; типы даты и времени; типы временных интервалов; булевский тип; типы коллекций; анонимные строчные типы; типы, определяемые пользователем; ссылочные типы**.

Булевский тип в SQL содержит три значения – *`true`*, *`false`* и *`uknown`*. Это связано с интенсивным использованием в SQL так называемого неопределенного значения (`NULL`), которое разрешается использовать вместо значения любого типа данных.

В модели данных SQL допускается объявление двух видов типов коллекций: **типы массива и типы мультимножества.** Элементы типа коллекции могут быть любого типа данных, определенного к моменту определения данного типа коллекции. При объявлении типа мультимножества можно явно запретить наличие в его значениях элементов-дубликатов, что фактически приводит к объявлению типа множества.

Анонимный строчный тип – это безымянный структурный тип, значения которого являются строками, состоящими из элементов ранее определенных типов.

Поддерживается два вида типов данных , определяемых пользователями : **индивидуальные и структурные типы. Индивидуальный тип** – это именованный тип данных, основанный на единственном предопределенном типе. Индивидуальный тип не наследует от своего опорного типа набор операций над значениями. Чтобы выполнить некоторую операцию базового типа над значениями определенного над ним индивидуального типа, требуется явно сообщить системе, что с этими значениями нужно обращаться как со значениями базового типа. Имеется также возможность явного определения методов, функций и процедур, связанных с данным индивидуальным типом.

Структурный тип данных – это именованный тип данных, включающий один или более атрибутов любого из допустимых в SQL типа данных, в том числе другого структурного типа, типа коллекций, анонимного строчного типа и т. д. Дополнительные механизмы определяемых пользователями методов, функций и процедур позволяют определить поведенческие аспекты структурного типа. При определении структурного типа можно использовать механизм наследования от ранее определенного структурного типа.

При определении типизированных таблиц можно использовать механизм наследования. Можно определить подтаблицу типизированной супертаблицы, если структурный тип подтаблицы является непосредственным подтипом структурного типа супертаблицы. Подтаблица наследует у супертаблицы способ генерации значений ссылочного типа и все ограничения целостности, которые были специфицированы в определении супертаблицы. Дополнительно можно определить ограничения, затрагивающие новые столбцы.

8. Основные черты модели данных ODMG.

Структура данных

Объектно-ориентированная модель данных, специфицированная в ODMG 3.0, отличается от других двух моделей, прежде всего, в одном принципиальном аспекте. В модели данных SQL и истинной реляционной модели данных база данных представляет собой набор именованных контейнеров данных одного родового типа: таблиц или отношений соответственно. В объектно-ориентированной модели данных база данных – **это набор объектов (контейнеров данных) произвольного типа.**

Манипулирование данными в объектной модели

В стандарте ODMG в качестве базового средства манипулирования объектными базами данных предлагается язык **OQL** (Object Query Language). Это небольшой, но достаточно сложный язык запросов. Разработчики в целом характеризуют его следующим образом:

- OQL опирается на объектную модель ODMG (имеется в виду, что в нем поддерживаются средства доступа ко всем возможным структурам данных, допускаемых в структурной части модели).
- OQL очень близок к SQL/92. Расширения относятся к объектно-ориентированным понятиям, таким как сложные объекты, объектные идентификаторы, путевые выражения, полиморфизм, вызов операций и отложенное связывание.
- В OQL обеспечиваются высокоуровневые примитивы для работы с множествами объектов, но, кроме того, имеются настолько же эффективные примитивы для работы со структурами, списками и массивами.
- OQL **является функциональным языком**, допускающим неограниченную композицию операций, если операнды не выходят за пределы системы типов. Это является следствием того факта, что результат любого запроса обладает типом, принадлежащим к модели типов ODMG, и поэтому к результату запроса может быть применен новый запрос.
- OQL **не является вычислительно полным языком**. Он представляет собой простой язык запросов.
- Операторы языка OQL могут вызываться из любого языка программирования, для которого в стандарте ODMG определены правила связывания. И, наоборот, в запросах OQL могут присутствовать вызовы операций, запрограммированных на этих языках.
- В OQL не определяются явные операции обновления, а используются вызовы операций, определенных в объектах для целей обновления.
- В OQL обеспечивается декларативный доступ к объектам. По этой причине OQL-запросы могут хорошо оптимизироваться.
- Можно легко определить формальную семантику OQL.

Получить номера руководителей отделов и тех служащих их отделов, зарплата которых превышает 20000 руб.

```
SELECT DISTINCT STRUCT ( ОТД_РУК: D.ОТД_РУК,
СЛУ: ( SELECT E
FROM D.CONSISTS_OF AS E
WHERE E.СЛУ_ЗАРП > 20000.00 ) )
FROM ОТДЕЛЫ D
```

Здесь предполагается, что для атомарного объектного типа ОТДЕЛ определен экстент типа множества с именем ОТДЕЛЫ. В запросе перебираются все существующие объекты типа ОТДЕЛ, и для каждого такого объекта происходит переход по связи к литеральному множеству объектов типа СЛУЖАЩИЙ, соответствующих служащим, которые работают в данном отделе. На основе этого множества формируется «усеченное» множество объектов типа СЛУЖАЩИЙ, в котором остаются только объекты-служащие с зарплатой, большей 20000.00. Результатом запроса является литеральное значение-множество, элементами которого являются значения-структуры

с двумя литеральными значениями, первое из которых есть атомарное литеральное значение типа `INTEGER`, а второе – литеральное значение-множество с элементами-объектами типа `СЛУЖАЩИЙ.` Более точно, результат запроса имеет тип `set < struct { integer ОТД_РУК; bag < СЛУЖАЩИЙ > СЛУ } >`.

В совокупности результатом допустимых в OQL выражений запросов могут являться:

- коллекция объектов;
- индивидуальный объект;
- коллекция литеральных значений;
- индивидуальное литеральное значение.

Ограничения целостности в объектной модели

В соответствии с общей идеологией объектно-ориентированного подхода в модели ODMG два объекта считаются совпадающими в том и только в том случае, когда являются одним и тем же объектом, т.е. имеют один и тот же OID. О**бъекты одного объектного типа с разными OID** считаются разными, даже если обладают полностью совпадающими состояниями. Поэтому в объектной модели **отсутствует аналог ограничения целостности сущности реляционной модели данных**. Интересно, что при определении атомарного объектного типа **можно объявить ключ – набор свойств объектного класса, однозначно идентифицирующий состояние каждого объект**а, входящего в экстент этого класса. Для класса может быть объявлено несколько ключей, а может не быть объявлено ни одного ключа даже при наличии определения экстента. Но при этом определение ключа не трактуется в модели как ограничение целостности; утверждается, что объявление ключа способствует повышению эффективности выполнения запросов.

Что же касается **ссылочной целостности**, то она **поддерживается, если** между двумя атомарными объектными типами определяется связь вида «**один-ко-многим**». В этом случае объекты на стороне связи «один» рассматриваются как предки, а объекты на стороне связи «многие» – как потомки, и ООСУБД обязана следить за тем, чтобы н**е образовывались потомки без предков**.

9. Типы данных, наследование типов данных в модели ODMG.

В объектной модели данных вводятся две разновидности типов: ***литеральные* и *объектные* типы**. **Литеральные типы данных** – это обычные типы данных, принятые в традиционных языках программирования. Они подразделяются на базовые скалярные числовые типы, символьные и булевские типы (*атомарные литералы*), конструируемые типы записей (*структур*) и *коллекций*.

Литеральный тип записи – это традиционный определяемый пользователем структурный тип, подобный структурному типу языка C или типу записи языка Pascal. Отличие состоит лишь в том, что в объектной модели атрибут типа записи может определяться не только на литеральном, но и на объектном типе, т.е. значение литерального типа записи может в качестве компонентов включать объекты. У любого

существующего объекта имеется одно и только одно местоположение , характеризующееся его идентификатором (OID). Когда в модели **говорится, что некоторое структурное значение в качестве компонента имеет некоторый объект**, то, конечно, **имеется в виду OID** этого объекта, являющийся всего лишь аналогом указательного значения в традиционных языках программирования.

Имеются четыре вида типов коллекций: **типы множеств, мультимножеств** (неупорядоченные наборы элементов, возможно, содержащие дубликаты), **списков** (упорядоченные наборы элементов, возможно, содержащие дубликаты) и **словарей** (множества пар <ключ, значение>, причем все ключи в этих парах должны быть различными). Типом элемента любой коллекции может являться любой скалярный или объектный тип за исключением того же типа коллекции.

Объектные типы в объектной модели данных по смыслу **ближе всего к понятию *класса*** в объектно-ориентированных языках программирования. У каждого объектного типа имеется операция создания и инициализации нового объекта этого типа. Эта операция возвращает значение OID нового объекта, который можно хранить в любом месте, где допускается хранение объектов данного типа, и использовать для обращения к *операциям* объекта, определенным в его объектном типе.

Имеются два вида объектных типов. Первый из них называется **атомарным объектным типом**. Нестрого говоря, при определении атомарного объектного типа указывается его внутренняя структура (набор свойств – атрибутов и связей) и набор операций, которые можно применять к объектам этого типа. Для определения атомарного объектного типа **можно использовать механизм наследования, расширяя набор свойств и/или переопределяя существующие и добавляя новые операции**.

Атрибутами называются свойства объекта, значение которых можно получить по OID объекта. Значениями атрибутов могут быть и литералы, и объекты (т.е. OID), но только тогда, когда не требуется обратная ссылка. **Связи** – это инверсные свойства. В этом случае значением свойства может быть только объект. Связи определяются между атомарными объектными типами. В объектной модели ODMG поддерживаются только бинарные связи, т.е. связи между двумя типами. Связи могут быть разновидностей «один-к-одному», «один-ко-многим» и «многие-ко-многим» в зависимости от того, сколько экземпляров соответствующего объектного типа может участвовать в связи.

Связи явно **определяются путем указания *путей обхода*.** Пути обхода указываются парами, по одному пути для каждого направления обхода связи. Например, в базе данных СЛУЖАЩИЕ-ОТДЕЛЫ служащий работает (`works`) в одном отделе, а отдел состоит (`consists of`) из множества служащих. Тогда путь обхода `consists_of` должен быть определен в объектном типе ОТДЕЛ, а путь обхода `works` – в типе СЛУЖАЩИЙ. Тот факт, что пути обхода относятся к одной связи, указывается в разделе `inverse` обоих объявлений пути обхода. Это связь «один-ко-многим». Путь обхода `consists_of` ассоциирует объект типа ОТДЕЛ с литеральным множеством объектов типа СЛУЖАЩИЙ, а путь обхода `works` ассоциирует объект типа СЛУЖАЩИЙ с объектом типа ОТДЕЛ. Пути обхода, ведущие к коллекциям объектов, могут быть упорядоченными или неупорядоченными в зависимости от вида коллекции, указанного в

объявлении пути обхода.

Второй вид – это **объектные типы коллекций**. Как и в случае использования литеральных типов коллекций, можно определять объектные типы множеств, мультимножеств, списков и словарей. Типом элемента объектного типа коллекции может быть любой литеральный или объектный тип за исключением того же типа коллекции. У объектных типов коллекций имеются предопределенные наборы операций. В отличие от литеральных типов коллекций, которые, как и все литеральные типы являются множествами значений, объектные типы коллекций обладают операцией создания объекта, имеющего, как и все объекты, собственный OID.

Интересен и важен один специальный случай неявного использования объектов типа множества. При определении атомарного объектного типа можно в качестве одного из дополнительных свойств этого типа указать, что для него должен быть создан объект типа множества, элементами которого являются объекты данного атомарного типа (**экстент** объектного структурного типа). Поскольку такой объект создается неявно, его OID неизвестен, но зато у него имеется имя, явно задающееся в определении и совпадающее с именем атомарного объектного типа. Наличие этой возможности позволяет создавать объектные базы данных, состоящие из именованных контейнеров однотипных объектов, причем в действительности эти контейнеры содержат OID'ы соответствующих объектов.

10. Основные черты истинно реляционной модели данных.

1) Структура данных

База данных в истинной реляционной модели – это **набор долговременно хранимых именованных *переменных отношений***, каждая из которых **определена на некотором типе отношений**. В каждый момент времени каждая переменная отношения базы данных содержит некоторое значение отношения соответствующего типа.

2) Манипулирование данными

Вообще говоря, в качестве эталонного средства манипулирования данными в истинной реляционной модели можно использовать реляционную алгебру Кодда. Однако Дейт и Дарвен предложили новую реляционную алгебру, названную ими **Алгеброй A**, которая **основывается на реляционных аналогах булевских операций *конъюнкции*, *дизъюнкции* и *отрицания***. Через ее операции выражаются все операции алгебры Кодда.

3) Ограничения целостности в истинно реляционной модели

В число **обязательных требований** истинной реляционной модели входит **требование определения хотя бы одного возможного ключа для каждой переменной отношения** (**возможный ключ** – это одно из подмножеств заголовка переменной отношения, обладающее свойствами первичного ключа). Кроме того, говорится, что «любое условное выражение, которое является (или логически эквивалентно) замкнутой правильно построенной формулой (WFF) реляционного исчисления, должно

быть допустимо в качестве спецификации ограничения целостности» .

Средства поддержки декларативной ссылочной целостности **фигурируют только в разделе рекомендуемых возможностей**: «В **D** [конкретную реализацию истинной реляционной модели] следует включить некоторую декларативную сокращенную форму для выражения ссылочных ограничений (называемых также ограничениями внешнего ключа)».

11. Типы данных, наследование типов в истинно реляционной модели.

В истинно реляционной модели очень большое внимание уделяется типам данных. Предлагаются **три категории типов данных**: **скалярные** типы, **кортежные** типы и **типы отношений**. **Скалярный тип данных** – это привычный инкапсулированный тип, реальная внутренняя структура которого скрыта от пользователей. Предлагаются механизмы определения новых скалярных типов и операций над ними. Типом атрибута определяемого скалярного типа может являться любой определенный к этому моменту скалярный тип, любой кортежный тип и тип отношения. Некоторые базовые скалярные типы данных должны быть предопределены в системе. В число этих типов должен входить тип `truth value` (так Дейт и Дарвен называют булевский тип) ровно с двумя значениями *`true`* и *`false`*.

Кортежный тип – это безымянный тип данных, определяемый с помощью генератора типа `TUPLE` с указанием множества пар `<имя_атрибута, тип_атрибута>` (заголовка кортежа). Типом атрибута кортежного типа может являться любой определенный к этому моменту скалярный тип, любой кортежный тип и тип отношения. Значением кортежного типа является кортеж, представляющий собой множество триплетов `<имя_атрибута, тип_атрибута, значение_атрибута>`, которое соответствует заголовку кортежа этого кортежного типа.

Тип отношения – это безымянный тип данных, определяемый с помощью генератора типа `RELATION` с указанием некоторого заголовка кортежа. Значением типа отношения является заголовок отношения, совпадающий с заголовком кортежа этого типа отношения, и тело отношения, представляющее собой множество кортежей, соответствующих этому заголовку. Кортежные типы и типы отношений не являются инкапсулированными: **имеется возможность прямого доступа к атрибутам**.

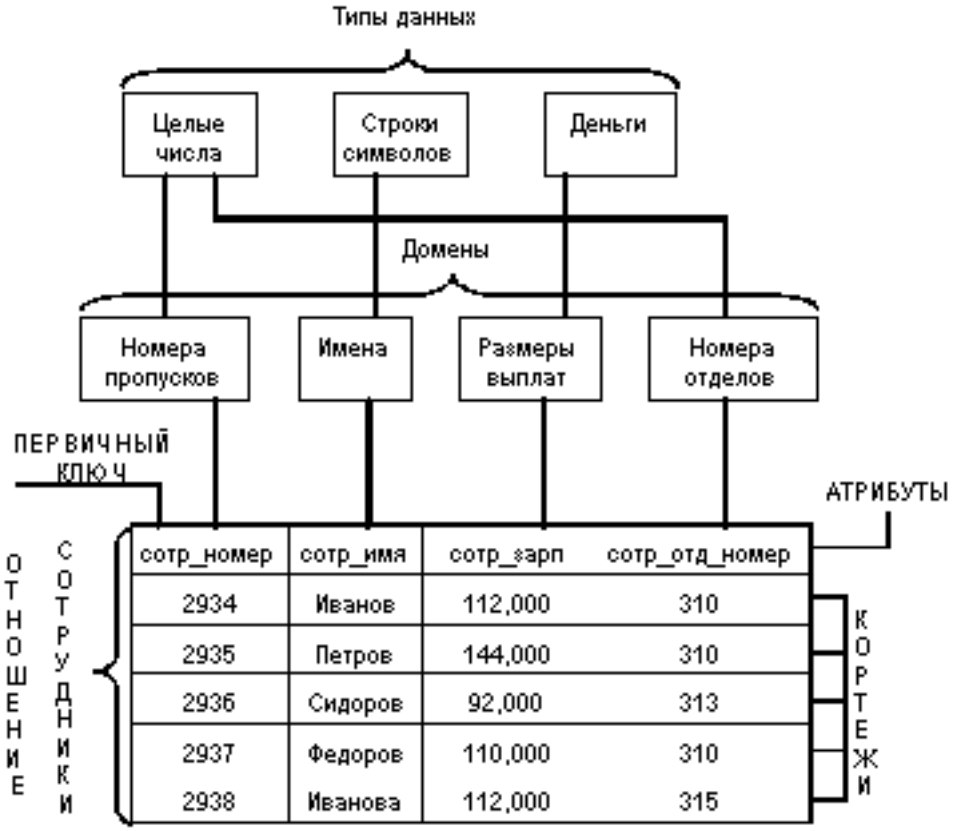
Для всех разновидностей типов данных **разработана модель множественного наследования**, позволяющая определять новые типы данных на основе уже определенных типов.

Понятно, что при таких определениях значениями атрибутов отношения могут быть не только значения произвольно сложных скалярных типов, типами атрибутов которых могут быть, в частности, отношения, но и просто отношения. Тем не менее, в Дейт и Дарвен говорят: «Каждый кортеж в [отношении] *`R`* содержит в точности одно значение *`v`* для каждого атрибута *`A`* в [заголовке отношения] *`H`*. Иными словами, *`R`* находится в *первой нормальной форме*, 1NF.» Это хорошее и понятное определение первой нормальной формы, но трудно сказать, согласился ли бы с ним Кодд.

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

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:



Тип данных

Понятие **тип данных** в реляционной модели данных полностью адекватно понятию типа данных в языках программирования. Определение типа данных состоит из **трех основных компонентов**: множества значений данного типа; набор операций, применимых к значениям типа;способ внешнего представления значений типа (литералов)

Обычно в современных реляционных БД допускается хранение символьных, числовых данных(точных и приблизительных), 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".

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. С каждым доменом связывается имя, уникальное среди имен всех доменов соответствующей базы данных.

Наиболее правильной интуитивной трактовкой понятия домена является понимание домена как допустимого потенциального ограниченного подмножества 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).

Если некоторый атрибут отношения определяется на некотором домене (как, например, атрибут сотр_имя определяется на домене имена), то в дальнейшем **ограничение домена играет роль ограничения целостности**, накладываемого на значения этого атрибута

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.

Отношение.

Для уточнения термина *отношение* выделяются понятия ***заголовка отношения*, *значения отношения* и *переменной отношения***. Кроме того, потребуется вспомогательное понятие ***кортежа*** .

***Заголовком* (или *схемой*) *отношения R* (*HR*)** называется конечное множество упорядоченных пар вида <*A*, *T*>, где *A* называется именем *атрибута*, а *T* – имя некоторого базового типа или ранее определенного домена. По определению требуется, чтобы **все имена атрибутов в заголовке отношения были различны.** В примере

заголовком отношения СЛУЖАЩИЕ является множество пар {<сотр_номер номера пропусков>, <сотр_имя, имена>, <сотр_зарп, Размеры выплат>, <сотр_отд_номер, номера отделов>}.

Если все атрибуты заголовка отношения определены на разных доменах, то, чтобы не плодить лишних имен, разумно использовать для именования атрибутов имена соответствующих доменов.

Кортежем *tR*, соответствующим заголовку *HR*, называется множество упорядоченных триплетов вида <*A*, *T*, *v*>, по одному такому триплету для каждого атрибута в *HR*. Третий элемент – *v* – триплета < *A*, *T*, *v* > должен являться допустимым значением типа данных или домена *T*

Заголовку отношения СЛУЖАЩИЕ соответствуют, например, следующие кортеж: {<сотр_номер, номера пропусков, 2934>, <сотр_имя, имена, Иванов>, <сотр_зарп, размеры выплат, 112.000>, <сотр_отд_номер, Номера отделов, 310>}

Телом *BR* отношения *R* называется произвольное множество кортежей *tR*. Одно из возможных тел отношения Сотрудники показано на рисунке. Заметим, что в общем случае могут существовать такие кортежи *tR*, которые соответствуют *HR*, но не входят в *BR*. **Значением** *VR* отношения *R* называется пара множеств *HR* и *BR*. Одно из допустимых значений отношения СОТРУДНИКИ показано на рисунке.

В изменчивой реляционной базе данных хранятся отношения, значения которых изменяются во времени. **Переменной** *VARR* называется именованный контейнер, который может содержать любое допустимое значение *VR*. Естественно, что при определении любой *VARR* требуется указывать соответствующий заголовок отношения *HR*.

Заметим, что в дальнейшем в тех случаях, когда точный смысл термина понятен по контексту, мы будем использовать не уточненный термин **отношение**, как в смысле **значение отношения**, так и в смысле **переменная отношения**.

По определению **степенью**, или **“арностью**” заголовка отношения, кортежа, соответствующего этому заголовку, тела отношения, значения отношения и переменной отношения является мощность заголовка отношения

Например, степень отношения СОТРУДНИКИ равна четырем, то есть оно является 4-арным (*кватернарным)*. Разумно считать **схемой реляционной базы** *данных* набор пар <имя_*VARR*, *HR* >, включающий имена и заголовки

всех переменных отношения, которые определены в базе данных

Реляционная база данных – это набор пар <VARr, Hr> (конечно, каждая переменная отношения в любой момент времени содержит некоторое значение-отношение, в частности, пустое).

Заметим, что в классических реляционных базах данных после определения схемы базы данных могли изменяться только значения переменных отношений. Однако теперь в большинстве реализаций допускается и изменение схемы базы данных: определение новых и изменение заголовков существующих переменных отношений. Это принято называть **эволюцией** схемы базы данных.

Первичный ключ и интуитивная интерпретация реляционных понятий

По определению, **первичным ключом** переменной отношения является такое подмножество S множества атрибутов ее заголовка, что в любое время значение первичного ключа (составное, если в состав первичного ключа входит более одного атрибута) в любом кортеже тела отношения отличается от значения первичного ключа в любом другом кортеже тела этого отношения, а никакое собственное подмножество S этим свойством не обладает. Существование первичного ключа у любого значения отношения является следствием одного из фундаментальных свойств отношений, а именно того свойства, что тело отношения является множеством кортежей.

Обычным житейским представлением отношения является ***таблица***, *заголовком* которой является схема отношения, а *строками* – кортежи отношения-экземпляра; в этом случае имена атрибутов соответствуют именам *столбцов* данной таблицы. Поэтому иногда говорят про «столбцы таблицы», имея в виду «атрибуты отношения».

Конечно, это достаточно грубая терминология, поскольку у обычных таблиц и строки, и столбцы упорядочены, тогда как атрибуты и кортежи отношений являются элементами неупорядоченных множеств. Тем не менее, когда мы перейдем к рассмотрению практических вопросов организации реляционных баз данных и средств управления, то будем использовать эту «житейскую» терминологию. Подобной терминологии придерживаются в большинстве коммерческих реляционных СУБД. Иногда также используются термины *файл* как аналог таблицы, *запись* как аналог строки и *поле* как аналог столбца.

13. Фундаментальные свойства отношений.

Отсутствие кортежей-дубликатов, первичный и возможные ключи отношений

То свойство, что т**ело любого отношения никогда не содержит кортежей-дубликатов, следует из определения тела отношения как множества кортежей**. В классической теории множеств по определению любое множество состоит из различных элементов.

Именно **из этого свойства вытекает наличие у каждого значения отношения первичного ключа** – минимального множества атрибутов, являющегося подмножеством заголовка данного отношения, составное значение которых уникально определяет кортеж отношения. Действительно, поскольку в любое время все кортежи тела любого отношения различны, у любого значения отношения свойством уникальности обладает, по крайней мере, полный набор его атрибутов. Однако в 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 – однозначного определения кортежа. Немного позже мы покажем, почему свойство минимальности первичного ключа является критически важным. Понятно, что **если у любого отношения существует набор атрибутов, обладающий свойством уникальности, то существует и минимальный набор атрибутов, обладающий свойством уникальности**.

Могут существовать значения отношения с несколькими несовпадающими минимальными наборами атрибутов, обладающими свойствами уникальности проектировщик базы данных должен решить, какое из альтернативных множеств атрибутов назвать первичным ключом, а остальные минимальные наборы атрибутов, обладающие свойством уникальности, называются *возможными ключами*.

Теперь поясним, почему проектировщику следует явно объявлять первичный и возможные ключи переменных отношений. Дело в том, что в результате этого объявления СУБД получает информацию, которая в дальнейшем будет использоваться как ограничения целостности. СУБД никогда не допустит появления в переменной отношения значения-отношения, содержащего два кортежа с одинаковым значением атрибута СЛУ_НОМЕР (определение первичного ключа для данной переменной отношения отменить нельзя). Появление двух кортежей с одинаковым значением атрибута СЛУ_ИМЯ будет также невозможно до тех пор, пока остается в силе определение {СЛУ_ИМЯ} как возможного ключа. Тем самым объявления первичного и возможных ключей дают СУБД возможность поддерживать целостность базы данных даже в случае попыток занесения в нее некорректных данных.

Наконец, вернемся к свойству минимальности первичного и возможных ключей. Как отмечалось выше, это свойство является критически важным, и важность проявляется именно при трактовке первичного и возможных ключей как ограничений целостности. В нашем примере с отношением СЛУЖАЩИЕ свойством уникальности будет обладать не только множество атрибутов {СЛУ_НОМЕР}, но и, например, множество {СЛУ_НОМЕР, СЛУ_ОТД_НОМЕР}. Но если бы мы выставили в качестве ограничения целостности требование уникальности { СЛУ _ НОМЕР , СЛУ_ОТД_НОМЕР}, то СУБД гарантировала бы отсутствие кортежей с одинаковым значением атрибута СЛУ_НОМЕР не во всем значении отношения СЛУЖАЩИЕ, а только в группах кортежей с одним и тем же значением атрибута СЛУ_ОТД_НОМЕР. Понятно, что это не соответствует смыслу моделируемой предметной области.

Отсутствие упорядоченности кортежей

Формально свойство отсутствия упорядоченности кортежей в значении отношения

также является **следствием определения тела отношения как множества кортежей**. Однако на это свойство можно взглянуть и с другой стороны. Да, то обстоятельство, что тело отношения является множеством кортежей, облегчает построение полного механизма реляционной модели данных, включая базовые средства манипулирования данными – реляционные алгебру и исчисление.

Достаточно часто у пользователей реляционных СУБД и разработчиков информационных систем вызывает раздражение тот факт, что они не могут хранить кортежи отношений на физическом уровне в нужном им порядке. И ссылки на требования реляционной теории здесь не очень уместны. Можно было бы разработать другую теорию, в которой допускались бы упорядоченные «отношения». Однако хранить упорядоченные списки кортежей в условиях интенсивно обновляемой базы данных гораздо сложнее технически, а поддержка упорядоченности влечет за собой существенные накладные расходы.

Отсутствие требования к поддержанию порядка на множестве кортежей отношения придает СУБД дополнительную гибкость 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, является не отношением, а некоторым упорядоченным списком кортежей, и он может быть только окончательным результатом, к которому уже нельзя адресовать запросы.

Отсутствие упорядоченности атрибутов

Атрибуты отношений не упорядочены, поскольку **по определению** заголовок отношения есть **множество пар** <имя атрибута, имя домена>. Для ссылки на значение атрибута в кортеже отношения всегда используется имя атрибута.

СУБД сама принимает решение о том, в каком физическом порядке следует хранить значения атрибутов кортежей (хотя обычно один и тот же физический порядок поддерживается для всех кортежей каждого отношения). Кроме того, это свойство облегчает выполнение операции модификации схем существующих отношений не только путем добавления новых атрибутов, но и путем удаления существующих.

Атомарность значений атрибутов, первая нормальная форма отношения

Значения всех атрибутов являются атомарными (вернее, скалярными). Это **следует из определения домена как потенциального множества значений скалярного типа данных**, т. е. среди значений домена не могут содержаться значения с видимой структурой, в том числе множества значений (отношения). Заметим, что это не противоречит потенциальной возможности использования при спецификации атрибутов типов данных, определяемых пользователями. Например, можно было бы добавить в схему отношения СЛУЖАЩИЕ атрибут СЛУ_ФОТО, определенный на домене (или типе данных) ФОТОГРАФИИ. **Главное** в атомарности значений атрибутов состоит в том, что реляционная **СУБД не должна обеспечивать пользователям явной видимости внутренней структуры значения**. Со всеми значениями можно обращаться

только с помощью операций, определенных в соответствующем типе данных.

Эдгар Кодд справедливо утверждал, что для моделирования большинства предметных областей можно обойтись **отношениями, атрибуты которых определены на простых доменах, элементы которых являются атомарными, не декомпозируемыми**. Он, в частности, говорил следующее: «Отношение, все домены которого являются простыми, может быть представлено двухмерным массивом … с однородными столбцами. Для отношения с одним или более непростыми доменами требуются несколько более сложные структуры данных.» Более того, он предлагал простую процедуру нормализации, приводящую отношение, значениями одного из атрибутов которых являются отношения, к нескольким отношениям над простыми доменами.

Однако и в истинной реляционной модели данных, и в модели данных SQL теперь можно определять отношения (таблицы), значениями атрибутов (столбцов) которых являются отношения (мультимножества строк). Скорее всего, это произошло по двум причинам:Во-первых, в течение многих лет реляционную модель данных (и модель SQL) многие люди критиковали за сложность представления иерархически организованных данных. Понятно, что при наличии механизма вложенных отношений иерархические данные представляются вполне естественно. Во-вторых, оказалось, что при тщательном определении системы типов, наличие атрибутов (столбцов) со значениями-отношениями никак не влияет на методы, разработанные в исходной теории реляционных БД

Нормализованные отношения составляют основу классического реляционного подхода к организации баз данных. Они обладают некоторыми ограничениями10) (не всякую информацию удобно представлять в виде плоских таблиц), но существенно упрощают манипулирование данными. Рассмотрим, например, два идентичных оператора занесения кортежа:

- зачислить служащего Кузнецова (пропуск номер 3000, зарплата 25000.00) в отдел номер 320;
- зачислить служащего Кузнецова (пропуск номер 3000, зарплата 25000.00) в отдел номер 310.

НОМЕР_ОТДЕЛА	ОТДЕЛ		
	СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП
310	2934	Иванов	22000.00
	2935	Петров	30000.00
	2937	Федоров	20000.00
313	2936	Сидоров	18000.00
315	2938	Иванова	22000.00

Рис. 3.2. Ненормализованное отношение ОТДЕЛЫ-СЛУЖАЩИЕ

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	СЛУ_ОТД_НОМЕР
2934	Иванов	22000.00	310
2935	Петров	30000.00	310
2936	Сидоров	18000.00	313
2937	Федоров	20000.00	310
2938	Иванова	22000.00	315

Рис. 3.3. Отношение СЛУЖАЩИЕ: нормализованный вариант отношения ОТДЕЛЫ-СЛУЖАЩИЕ

Если информация о служащих представлена в виде отношения СЛУЖАЩИЕ, оба оператора будут выполняться одинаково (вставить кортеж в отношение СЛУЖАЩИЕ). Если же работать с ненормализованным отношением ОТДЕЛЫ-СЛУЖАЩИЕ, то первый оператор приведет к простой вставке кортежа, а второй – к добавлению кортежа в значение-отношение атрибута ОТДЕЛ кортежа с первичным ключом 310.

При работе с ненормализованными отношениями аналогичные затруднения возникают при выполнении операций удаления и модификации кортежей.

14. Реляционная модель данных: общее понятие и составные части.

В структурной части модели фиксируется, что **единственной родовой структурой** данных, используемой в реляционных БД, **является нормализованное n-арное отношение.** Определяются понятия доменов, атрибутов, кортежей, заголовка, тела и переменной отношения.

В манипуляционной части модели определяются **два фундаментальных механизма манипулирования реляционными БД – реляционная алгебра и реляционное исчисление**. Первый механизм базируется в основном на **классической теории множеств** (с некоторыми уточнениями и добавлениями), а второй – **на классическом логическом аппарате исчисления предикатов первого порядка**. 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.

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. Это требование автоматически удовлетворяется, если в системе не нарушаются базовые свойства отношений.

На самом деле, требование целостности сущности полностью звучит следующим образом: **у любой переменной отношения должен существовать первичный ключ, и никакое значение первичного ключа в кортежах значения-отношения переменной отношения не должно содержать неопределенных значений.**

Неопределенное значение не принадлежит никакому типу данных и может присутствовать среди значений любого атрибута, определенного на любом типе данных (если это явно не запрещено при определении атрибута). Если *a* – это значение некоторого типа данных или NULL, *op* – любая двуместная «арифметическая» операция этого типа данных (например, +), а *lop* – операция сравнения значений этого типа (например, =), то по определению:

```
a op NULL = NULL
NULL op a = NULL
a lop NULL = unknown
NULL lop a = unknown
```

Здесь *unknown* – это третье значение логического, или булевского, типа, обладающее следующими свойствами:

```
NOT unknown = unknown
true AND unknown = unknown
true OR unknown = true
false AND unknown = false
false OR unknown = unknown
```

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.

Атрибут называется **внешним ключом**, если 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).

Ограничения целостности сущности и по ссылкам должны поддерживаться СУБД.

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.**

15. Реляционная алгебра Кодда.

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

Существует много подходов к определению реляционной алгебры, которые различаются наборами операций и способами их интерпретации, но, в принципе, являются более или менее равносильными. 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;
- взятия декартова 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.

Общая интерпретация реляционных операций

- При выполнении операции **объединения** (`UNION`) двух отношений **с одинаковыми заголовками** производится отношение, включающее все кортежи, которые входят хотя бы в одно из отношений-операндов.
- Операция **пересечения** (`INTERSECT`) двух отношений **с одинаковыми заголовками** производит отношение, включающее все кортежи, которые входят в оба отношения-операнда.
- Отношение, являющееся **разностью** (`MINUS`) двух отношений **с одинаковыми заголовками**, включает все кортежи, входящие в отношение-первый операнд, такие, что ни один из них не входит в отношение, которое является вторым операндом.
- При выполнении **декартова произведения** (`TIMES`) двух отношений, **пересечение заголовков которых пусто**, производится отношение, кортежи которого производятся путем объединения кортежей первого и второго операндов.
- Результатом **ограничения** (`WHERE`) отношения по некоторому условию является отношение, включающее кортежи отношения-операнда, удовлетворяющее этому условию.
- При выполнении **проекции** (`PROJECT`) отношения на заданное подмножество множества его атрибутов производится отношение, кортежи которого являются соответствующими подмножествами кортежей отношения-операнда.
- При **соединении** (`JOIN`) двух отношений по некоторому условию образуется результирующее отношение, кортежи которого производятся путем объединения кортежей первого и второго отношений и удовлетворяют этому условию.

  (*A* JOIN *B* WHERE *comp* = (*A* TIMES *B*) WHERE *comp*)

  Имеются важный частный случай соединения – *эквисоединение* (EQUIJOIN) и простое, но важное расширение операции эквисоединения – *естественное соединение* (NATURAL JOIN) Операция соединения называется операцией эквисоединения, если условие соединения имеет вид (a = b), где a и b – атрибуты разных операндов соединения.

  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 спроецированный на *ab* результат эквисоединения *A* и *B* по условию $A.c = B.c$

  Хотя операция естественного соединения выражается через операции переименования, соединения общего вида и проекции, для нее обычно используется сокращенная форма, называемая NATURAL JOIN

- **Реляционное деления** (`DIVIDE BY`) . 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 ($i = 1, 2, \ldots, m$) отношения *B* не только обладают одним и тем же именем, но и определены на одном и том же домене

  Назовем множество атрибутов $\{a_j\}$ составным атрибутом *a*, а множество

атрибутов $\{b_j\}$ – 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 (A DIVIDE BY B) является унарное отношение C $\{a\}$, тело которого состоит из кортежей v таких, что в теле отношения A содержатся кортежи $<v, w>$ такие, что множество значений $\{w\}$ включает множество значений атрибута b в отношении B

- Операция **переименования** (`RENAME`) 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, которая не вырабатывает значения) является некое отношение, можно образовывать реляционные выражения, в которых вместо отношения-операнда некоторой реляционной операции находится вложенное реляционное выражение. В построении реляционного выражения могут участвовать все реляционные операции, кроме операции присваивания. Вычислительная интерпретация реляционного выражения диктуется установленными приоритетами операций:

Операция	Приоритет
RENAME	4
WHERE	3
PROJECT	3
TIMES	2
JOIN	2
INTERSECT	2
DIVIDE BY	2
UNION	1
MINUS	1

Таблица приоритетов операций традиционной реляционной алгебры

Замкнутость реляционной алгебры и операция переименования

Каждое **значение-отношение характеризуется заголовком** (или схемой) **и телом** (или множеством кортежей). Поэтому, если нам действительно нужна алгебра, 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 можно будет строить вложенные выражения.

Заголовок отношения представляет собой множество пар <имя-атрибута, имя-домена>. Домены атрибутов результирующего отношения однозначно определяются доменами отношений-операндов. Однако с именами атрибутов результата не всегда все так просто.

Проблемы могут возникать и в случаях других двуместных операций. Для разрешения проблем в число операций реляционной алгебры вводится операция переименования. Ее следует применять в том 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 без всяких проблем.

В дальнейшем изложении мы будем предполагать применение операции переименования во всех конфликтных ситуациях. Заметим, кстати, что невозможность применения некоторых операций к произвольным парам значений отношений без предварительного переименования атрибутов отношений операндов означает, что «алгебра» Кодда не является алгеброй отношений в математическом смысле.

Особенности теоретико-множественных операций реляционной алгебры

Операции объединения, пересечения, взятия разности. Совместимость по объединению

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 в реляционной алгебре допустить возможность теоретико-множественного объединения двух произвольных отношений (с разными заголовками), 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.

Эти соображения подводят к понятию **совместимости отношений по объединению**: два отношения совместимы по объединению в том и только в том случае, когда обладают одинаковыми заголовками. В развернутой форме 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 (эта развернутая формулировка, вообще говоря, является излишней, но она пригодится нам в следующем абзаце).

Напомним, что если два отношения «почти» совместимы по объединению, т. е. совместимы во всем, кроме имен атрибутов, то до выполнения операции типа

объединения эти отношения можно сделать полностью совместимыми по объединению **путем применения операции переименования**.

Операция расширенного декартова произведения и совместимость отношений относительно этой операции

В теории множеств декартово 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, для любых двух отношений возможно получение прямого произведения. Но результат не будет отношением! Элементами результата будут не кортежи, а пары кортежей.

Поэтому в реляционной алгебре используется специализированная форма операции взятия декартова произведения – **расширенное декартово произведение отношений**. При взятии расширенного декартова произведения двух отношений элементом результирующего отношения является кортеж, который представляет собой объединение одного кортежа первого отношения и одного кортежа второго отношения.

Приведем **более точное определение *операции расширенного декартова произведения*.** Пусть имеются два отношения R1$\{a_1, a_2, \ldots, a_n\}$ и R2$\{b_1, b_2, \ldots, b_m\}$. Тогда результатом операции R1 TIMES R2 является отношение R$\{a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_m\}$, тело которого является множеством кортежей вида $\{ra_1, ra_2, \ldots, ra_n, rb_1, rb_2, \ldots, rb_m\}$ таких, что $\{ra_1, ra_2, \ldots, ra_n\}$ входит в тело R1, а $\{rb_1, rb_2, \ldots, rb_m\}$ входит в тело R2.

Поскольку схема результирующего отношения является объединением схем отношений-операндов, то очевидной проблемой может быть именование атрибутов результирующего отношения, если отношения-операнды обладают одноименными атрибутами.

Эти соображения приводят к введению понятия **совместимости по взятию расширенного декартова произведения**. Два отношения **совместимы по взятию расширенного декартова произведения** в том и только в том случае, если пересечение множеств имен атрибутов, взятых из их схем отношений, пусто. Любые два отношения всегда могут стать совместимыми по взятию декартова произведения, если применить операцию переименования к одному из этих отношений.

Следует заметить, что операция взятия декартова произведения не является слишком осмысленной на практике. Во-первых, мощность тела ее результата очень велика даже при допустимых мощностях операндов, а во-вторых, результат операции не более информативен, чем взятые в совокупности операнды. Как будет показано далее, основной смысл включения операции расширенного декартова произведения в состав реляционной алгебры Кодда 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**. **Все операции, кроме взятия разности**, являются **коммутативными**, т. е. A *OP* B = B *OP* A.

16. Алгебра A.

Пусть *r* – отношение, A – имя атрибута отношения *r*, T – имя соответствующего типа (т. е. типа или домена атрибута A), v – значение типа T. Тогда:

- заголовком H*r* отношения *r* называется множество атрибутов, т.е. упорядоченных пар вида <A, T>. По определению никакие два атрибута в этом множестве не могут содержать одно и то же имя атрибута A;
- кортеж t*r*, соответствующий заголовку H*r*, – это множество упорядоченных триплетов вида <A, T, v>, по одному такому триплету для каждого атрибута в H*r*;
- тело B*r* отношения *r* – это множество кортежей t*r*. Заметим, что (в общем случае) могут существовать такие кортежи t*r*, которые соответствуют H*r*, но не входят в B*r*.

Заметим, что заголовок – это множество (упорядоченных пар вида <A, T>), тело – это множество (кортежей t*r*), и кортеж – это множество (упорядоченных триплетов вида <A, T, v>). Элемент заголовка – это атрибут (т. е. упорядоченная пара вида <A,T>); элемент тела – это кортеж; элемент кортежа – это упорядоченный триплет вида <A, T, v>. Любое подмножество заголовка – это заголовок, любое подмножество тела – это тело, и любое подмножество кортежа – это кортеж.

Имена реляционных операций берутся в угловые скобки: <NOT>, <AND>, <OR> и т. д. В исходный базовый набор операций входят операции реляционного дополнения <NOT>, удаления атрибута <REMOVE>, переименования атрибута <RENAME>, реляционной конъюнкции <AND> и реляционной дизъюнкции <OR>.

Операция реляционного дополнения

Пусть *s* обозначает результат операции <NOT> *r*. Тогда:

- H*s* = H*r* (заголовок результата совпадает с заголовком операнда);
- B*s* = {t*s* : exists t*r* (t*r* ∉ B*r* and t*s* = t*r*) } (в тело результата входят все кортежи, соответствующие заголовку и не входящие в тело операнда).

Операция <NOT> производит дополнение *s* заданного отношения *r*. Заголовком *s* является заголовок r. Тело *s* включает все кортежи, соответствующие этому заголовку и не входящие в тело *r*.

Операция удаления атрибута

Пусть *s* обозначает результат операции *r* <REMOVE> A. Для обеспечения возможности выполнения операции требуется, чтобы существовал некоторый тип (или домен) T такой, что <A, T> ∈ H*r* (т. е. в состав заголовка отношения *r* должен входить атрибут A). Тогда:

- H*s* = H*r* minus {<A, T>}, т. е. заголовок результата получается из заголовка операнда изъятием атрибута A;
- B*s* = {t*s* : exists t*r* exists v (t*r* ∈ B*r* and v ∈ T and <A,T,v> ∈ t*r* and t*s* = t*r* minus {<A,T,v>})}, т. е. в тело результата входят все кортежи операнда, из которых удалено значение атрибута A.

Операция <REMOVE> производит отношение *s*, формируемое путем удаления указанного атрибута A из заданного отношения *r*. Операция эквивалентна взятию проекции *r* на все атрибуты, кроме A. Заголовок *s* получается теоретико-множественным вычитанием из заголовка *r* множества из одного элемента {<A, T>}. Тело *s* состоит из таких кортежей, которые соответствуют заголовку *s*, причем каждый из них является подмножеством некоторого кортежа тела отношения *r*.

Операция переименования

Пусть *s* обозначает результат операции *r* <RENAME> (A, B). Для обеспечения возможности выполнения операции требуется, чтобы существовал некоторый тип T, такой, что <A, T> ∈ H*r*, и чтобы не существовал такой тип T, что <B, T> ∈ H*r*. (Другими словами, в схеме отношения *r* должен присутствовать атрибут A и не должен присутствовать атрибут B.) Тогда:

- H*s* = (H*r* minus {<A, T>}) union {<B, T>}, т. е. в схеме результата B заменяет A;
- B*s* = {t*s* : exists t*r* exists v (t*r* ∈ B*r* and v ∈ T and <A, T, v> ∈ t*r* and t*s* = (t*r* minus {<A, T, v>}) union {<B, T, v>})}, т. е. в кортежах тела результата имя значений атрибута A меняется на B.

Операция <RENAME> производит отношение *s*, которое отличается от заданного отношения *r* только именем одного его атрибута, которое изменяется с A на B. Заголовок *s* такой же, как заголовок r, за исключением того, что пара <B, T> заменяет пару <A, T>. Тело *s* включает все кортежи тела *r*, но в каждом из этих кортежей триплет <B, T, v> заменяет триплет <A, T, v>.

Операция реляционной конъюнкции

Пусть *s* обозначает результат операции r_1 <AND> r_2. Для обеспечения возможности выполнения операции требуется, чтобы если <A, T1>$\in$Hr_1 и <A, T2>$\in$Hr_2, то T1=T2. (Другими словами, **если в двух отношениях-операндах имеются одноименные атрибуты, то они должны быть определены на одном и том же типе (домене).**)

Тогда:

- Hs = Hr_1 union Hr_2, т. е. заголовок результата получается путем объединения заголовков отношений-операндов
- Bs = { ts : exists tr_1 exists tr_2 ((tr_1 $\in$ Br_1 and tr_2 $\in$ Br_2) and ts = tr_1 union tr_2)}; обратите внимание на то, что кортеж результата определяется как *объединение кортежей операндов*; поэтому:
  - если схемы отношений-операндов имеют непустое пересечение, то операция <AND> работает как естественное соединение;
  - если пересечение схем операндов пусто, то <AND> работает как расширенное декартово произведение;
  - если схемы отношений полностью совпадают, то результатом операции является пересечение двух отношений-операндов.

Операция <AND> является реляционной конъюнкцией, в некоторых случаях выдающей в результате отношение rs, ранее называвшееся естественным соединением двух заданных отношений r_1 и r_2. Заголовок rs является объединением заголовков r_1 и r_2. Тело s включает каждый кортеж, соответствующий заголовку s и являющийся **надмножеством** некоторого кортежа из тела r_1 **и** некоторого кортежа из тела r_2.

Операция реляционной дизъюнкции

Пусть s обозначает результат операции r_1 <OR> r_2. Для обеспечения возможности выполнения операции требуется, чтобы если <A, T1> $\in$ Hr_1 и <A, T2> $\in$ Hr_2, то должно быть T1 = T2 (одноименные атрибуты должны быть определены на одном и том же типе). Тогда:

- Hs = Hr_1 union Hr_2 (из схемы результата удаляются атрибуты-дубликаты);
- Bs = { ts : exists tr_1 exists tr_2 ((tr_1 $\in$ Br_1 or tr_2 $\in$ Br_2) and ts = tr_1 union tr_2)}; очевидно, что при этом:
  - если у операндов нет общих атрибутов, то в тело результирующего отношения входят все такие кортежи ts, которые являются объединением кортежей tr_1 и tr_2, соответствующих заголовкам отношений-операндов, и хотя бы один из этих кортежей принадлежит телу одного из операндов;
  - если у операндов имеются общие атрибуты, то в тело результирующего отношения входят все такие кортежи ts, которые являются объединением кортежей tr_1 и tr_2, соответствующих заголовкам отношений-операндов, если хотя бы один из этих кортежей принадлежит телу одного из операндов, и значения общих атрибутов tr_1 и tr_2 совпадают;
  - если же схемы отношений-операндов совпадают, то тело отношения-результата является объединением тел операндов.

Операция <OR> является реляционной дизъюнкцией и обобщением того, что ранее называлось объединением. Заголовок *s* есть объединение заголовков r_1 и r_2. Тело *s* состоит из всех кортежей, соответствующих заголовку *s* и являющихся надмножеством **либо** некоторого кортежа из тела r_1, **либо** некоторого кортежа из тела r_2.

17. Полнота алгебры A.

Покажем, что Алгебра А является полной, т. е. на основе введенных операций выражаются все операции алгебры Кодда, рассмотренной в предыдущей лекции.

К настоящему моменту в состав базовых операций Алгебры А входят операция <REMOVE> в качестве аналога операции PROJECT, а также операция переименования атрибутов <RENAME>. UNION является частным случаем операции <OR>, TIMES, INTERSECT и NATURAL JOIN – частные случаи операции <AND>. Нам осталось показать, что через операции Алгебры А выражаются операции взятия разности MINUS, ограничения (WHERE), соединения общего вида (JOIN) и реляционного деления (DIVIDE BY).

Выводимость операции взятия разности

В общем случае нетрудно доказать, что если отношения r_1 и r_2 совместимы по объединению, то r_1 MINUS $r_2 = r_1$ <AND> <NOT> r_2.

Интерпретация операции ограничения

Операция ограничения *r* WHERE comp, где *r* – отношение, а *comp* – простое условие ограничения вида (*a comp-op b*), где *a* и *b* – имена атрибутов ограничиваемого отношения, для которых осмыслена операция сравнения *comp-op*, либо вида (*a comp-op* const), где *a* – имя атрибута ограничиваемого отношения, а const – литерально заданная константа. Операцией сравнения *comp-op* может быть «=», «≠», «>», «<», «≥», «≤». Покажем на нескольких примерах, как можно выразить операцию ограничения с помощью базовых операций Алгебры А для всех простых допустимых условий.

Мы снова предположим (для упрощения примеров), что множества значений доменов, на которых определены атрибуты отношения СЛУЖАЩИЕ_1, ограничены значениями, содержащимися в теле этого отношения. Начнем с обсуждения операции WHERE с условием вида a comp-op const.

Предположим, что мы хотим найти всех служащих с заработной платой, равной 20000.00 руб. Возьмем отношение ЗАРП_20000 {СЛУ_ЗАРП}. Мы видим, что результат операции СЛУЖАЩИЕ_1 <AND> ЗАРП_20000 в точности совпадает с результатом операции СЛУЖАЩИЕ_1 WHERE СЛУ_ЗАРП = 20000.00 (рис. 5.8).

СЛУЖАЩИЕ_1

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	РУК_НОМ
2934	Иванов	22000.00	2934
2935	Петров	30000.00	2934
2936	Сидоров	18000.00	2934
2937	Федоров	20000.00	2934
2938	Иванова	22000.00	2941
2939	Сидоренко	18000.00	2941
2940	Федоренко	20000.00	2941
2941	Иваненко	22000.00	2941

ЗАРП_20000

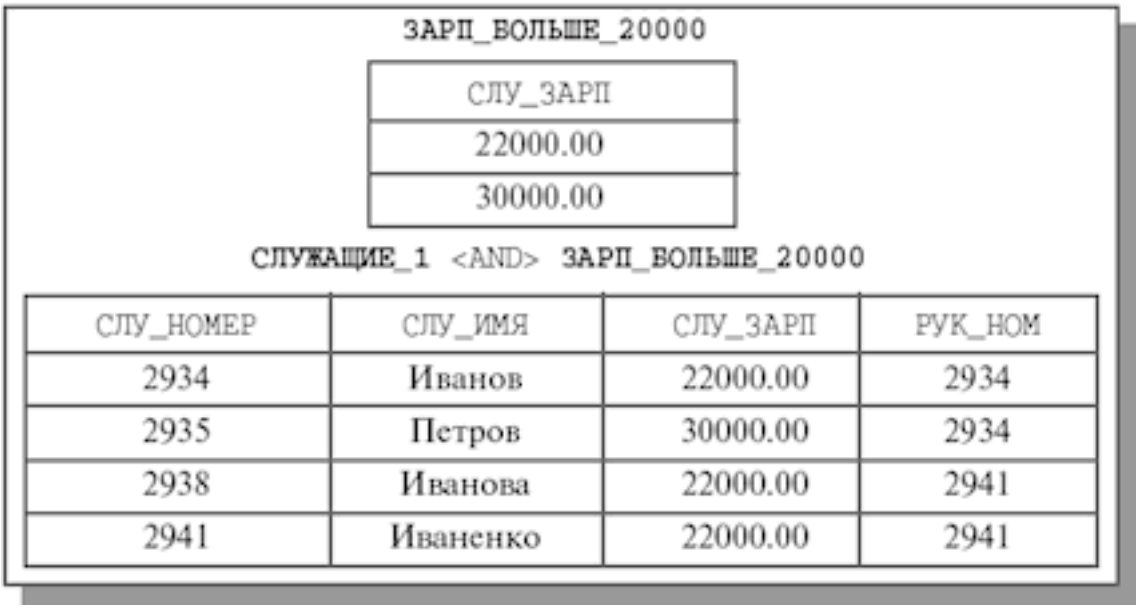
СЛУ_ЗАРП
20000.00

СЛУЖАЩИЕ_1 <AND> ЗАРП_20000

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	РУК_НОМ
2937	Федоров	20000.00	2934
2940	Федоренко	20000.00	2941

Рис. 5.8. Выражение `WHERE (a = const)` через `<AND>`

Если требуется найти служащих, чья заработная плата превышает 20000.00 руб., то возьмем отношение ЗАРП_БОЛЬШЕ_20000[18]) (рис. 5.9). Тогда снова результат операции СЛУЖАЩИЕ_1 <AND> ЗАРП_БОЛЬШЕ_20000.00 будет совпадать с результатом операции СЛУЖАЩИЕ_1 WHERE СЛУ_ЗАРП > 20000.00 (рис. 5.9).



ЗАРП_БОЛЬШЕ_20000

СЛУ_ЗАРП
22000.00
30000.00

СЛУЖАЩИЕ_1 <AND> ЗАРП_БОЛЬШЕ_20000

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	РУК_НОМ
2934	Иванов	22000.00	2934
2935	Петров	30000.00	2934
2938	Иванова	22000.00	2941
2941	Иваненко	22000.00	2941

Рис. 5.9. Выражение `WHERE (a > const)` через `<AND>`

Теперь обратимся к ограничениям с простым условием вида *a comp-op b*. Опять

начнем со случая, когда *comp-op* = «=». Предположим, что нам требуется найти данные о служащих, являющихся руководителями проектов, т. е. выполнить операцию СЛУЖАЩИЕ_1 WHERE СЛУ_НОМЕР = РУК_НОМ. Утверждается, что результат этой операции совпадает с результатом следующего выражения:

```
СЛУЖАЩИЕ_1 <AND> (((СЛУЖАЩИЕ_1 <REMOVE> СЛУ_НОМЕР) <REMOVE>
СЛУ_ИМЯ) <REMOVE> СЛУ_ЗАРП) <RENAME> (РУК_НОМ, СЛУ_НОМЕР))
```

Чтобы показать возможность выполнения операции ограничения вида r WHERE (a > b), предположим, что имеется отношение СЛУЖАЩИЕ_2 {СЛУ_НОМЕР, СЛУ_ИМЯ, СЛУ_ЗАРП, СЛУ_ПРЕМ}, причем атрибут СЛУ_ПРЕМ содержит значения премиального вознаграждения служащего.

Возьмем отношение ПРЕМ_БОЛЬШЕ_ЗАРП {СЛУ_ПРЕМ, СЛУ_ЗАРП}, тело которого включает все соответствующие заголовку кортежи {b, s} такие, что b > s. Другими словами, отношение ПРЕМ_БОЛЬШЕ_ЗАРП снова является литеральной константой типа отношения с двумя атрибутами СЛУ_ПРЕМ и СЛУ_ЗАРП. Конечно, даже в случае нашего примера мощность тела этого отношения достаточно велика. Тело отношения ПРЕМ_БОЛЬШЕ_ЗАРП показано в средней части

Результат выполнения операции СЛУЖАЩИЕ_2 <AND> ПРЕМ_БОЛЬШЕ_ЗАРП показан в нижней части Мы видим, что он совпадает с результатом операции СЛУЖАЩИЕ_2 WHERE (СЛУ_ПРЕМ > СЛУ_ЗАРП).

Аналогичным образом через операции Алгебры А выражаются операции ограничения, условия сравнения которых вида a comp_op b базируются на операциях сравнения «<», «≥», «≤», «≠».

Соединения общего вида

При наличии того факта, что операция взятия расширенного декартова произведения TIMES является частным случаем операции <AND>, после того как мы научились с помощью Алгебры А выполнять ограничения, становится очевидно, что через операции Алгебры А выражаются и соединения общего вида. В общем случае, чтобы получить результат соединения общего вида произвольных отношений A и B, нужно:

- выполнить над одним из отношений одну или несколько операций <RENAME>, чтобы избавиться от общих имен атрибутов;
- выполнить над полученными отношениями операцию <AND>, производящую расширенное декартово произведение;
- и для полученного отношения выполнить одну или несколько операций <AND> с отношениями-константами, чтобы должным образом ограничить его.

Реляционное деление

Пусть имеются отношения r_1{A, B} и r_2{B}. Утверждается, что результат r_1 DIVIDE BY r_2 совпадает с результатом выражения (r_1 PROJECT A) MINUS (((r_2 TIMES (r_1 PROJECT A)) MINUS r_1) PROJECT A) в терминах операций реляционной алгебры Кодда или (r_1 <REMOVE> B) <AND> <NOT> (((r_2 <AND> (r_1 <REMOVE> B)) <AND> <NOT> r_1) <REMOVE> B) в терминах операций Алгебры А.

Действительно, результатом выполнения операции r_1 PROJECT A является унарное отношение со схемой {A}, кортежи тела которого содержат все значения атрибута A из тела отношения r_1. Результат выражения r_2 TIMES (r_1 PROJECT A) – это бинарное отношение со схемой {A, B}, в тело которого входят все возможные комбинации значений атрибута B в теле отношения r_2 и атрибута A в теле отношения r_1. В теле результата вычисления выражения (r_2 TIMES (r_1 PROJECT A)) MINUS r_1 останутся только те кортежи, которые не входят во второй операнд, т. е. кортежи с таким значением атрибута A, что значение атрибута B, принадлежащее телу r_2, не является значением атрибута B ни в одном кортеже тела отношения r_1. Следовательно, если мы возьмем проекцию результата выражения (r_2 TIMES (r_1 PROJECT A)) MINUS r_1 на атрибут A, то в результирующем унарном отношении останутся только те значения A, которые не должны попасть в результат операции r_1 DIVIDE BY r_2. После выполнения завершающей операции MINUS мы получим желаемый результат.

СЛУЖАЩИЕ

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	ПРО_НОМ
2934	Иванов	22400.00	1
2935	Петров	29600.00	1
2936	Сидоров	18000.00	1
2937	Федоров	20000.00	1
2938	Иванова	22000.00	1
2934	Иванов	22400.00	2
2935	Петров	29600.00	2
2939	Сидоренко	18000.00	2
2940	Федоренко	20000.00	2
2941	Иваненко	22000.00	2

НОМЕРА_ПРОЕКТОВ

ПРО_НОМ
1
2

СЛУЖАЩИЕ **PROJECT** (СЛУ_НОМЕР, СЛУ_ИМЯ, СЛУ_ЗАРП)

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП
2934	Иванов	22400.00
2935	Петров	29600.00
2936	Сидоров	18000.00
2937	Федоров	20000.00
2938	Иванова	22000.00
2939	Сидоренко	18000.00
2940	Федоренко	20000.00
2941	Иваненко	22000.00

(**СЛУЖАЩИЕ** PROJECT (СЛУ_НОМЕР, СЛУ_ИМЯ, СЛУ_ЗАРП))
TIMES **НОМЕРА_ПРОЕКТОВ**

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	ПРО_НОМ
2934	Иванов	22400.00	1
2934	Иванов	22400.00	2
2935	Петров	29600.00	1
2935	Петров	29600.00	2
2936	Сидоров	18000.00	1
2936	Сидоров	18000.00	2
2937	Федоров	20000.00	1
2937	Федоров	20000.00	2
2938	Иванова	22000.00	1
2938	Иванова	22000.00	2
2939	Сидоренко	18000.00	1
2939	Сидоренко	18000.00	2
2940	Федоренко	20000.00	1
2940	Федоренко	20000.00	2
2941	Иваненко	22000.00	1
2941	Иваненко	22000.00	2

((**СЛУЖАЩИЕ** PROJECT (СЛУ_НОМЕР, СЛУ_ИМЯ, СЛУ_ЗАРП))
TIMES **НОМЕРА_ПРОЕКТОВ**) MINUS **СЛУЖАЩИЕ**

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	ПРО_НОМ
2936	Сидоров	18000.00	2
2937	Федоров	20000.00	2
2938	Иванова	22000.00	2
2939	Сидоренко	18000.00	1
2940	Федоренко	20000.00	1
2941	Иваненко	22000.00	1

(**СЛУЖАЩИЕ** PROJECT (СЛУ_НОМЕР, СЛУ_ИМЯ, СЛУ_ЗАРП))
MINUS ((((**СЛУЖАЩИЕ** PROJECT (СЛУ_НОМЕР, СЛУ_ИМЯ, СЛУ_ЗАРП)) TIMES
НОМЕРА_ПРОЕКТОВ) MINUS **СЛУЖАЩИЕ**)
PROJECT (СЛУ_НОМЕР, СЛУ_ИМЯ, СЛУ_ЗАРП))

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП
2934	Иванов	22400.00
2935	Петров	29600.00

Тем самым, мы показали, что пяти операций Алгебры А достаточно для выражения всех операций алгебры Кодда из лекции 4. Но на самом деле число операций можно еще более сократить.

18. Избыточность алгебры А.

В формальной математической логике стандартным базисом для выражения всех возможных булевских функций является набор {NOT, AND, OR} (отрицание, дизъюнкция и конъюнкция). Известно, что этот набор традиционен, но избыточен, поскольку верны тождества A AND B ≡ NOT (NOT A OR NOT B) и A OR B ≡ NOT (NOT A AND NOT B). (Эти тождества легко проверяются по таблицам истинности операций.) Оказывается (и это тоже легко проверить, опираясь на определения операций), что аналогичные тождества справедливы для операций <NOT>, <AND> и <OR> Алгебры А. Тем самым, в наборе базовых операций Алгебры А можно оставить операции <AND> и <NOT> (или <OR> и <NOT>).

Реляционные аналоги штриха Шеффера и стрелки Пирса

Более того, в алгебре логики существуют две операции, через каждую из которых выражаются все три «базовые» операции: «штрих Шеффера» – sh (A, B) ≡ NOT A OR NOT B – и «стрелка Пирса» – pi (A, B) ≡ NOT A AND NOT B.

Легко видеть, что

- sh (A, A) ≡ NOT A;
- sh (NOT A, NOT B) ≡ A OR B и
- NOT sh (A, B) ≡ A AND B.

Аналогично,

- pi (A, A) ≡ NOT A;
- pi (NOT A, NOT B) ≡ A AND B и
- NOT pi (A, B) ≡ A OR B.

Снова нетрудно проверить, что аналогичные тождества справедливы для реляционных вариантов штриха Шеффера (<sh> (r1, r2) ≡ <NOT> r1 <OR> <NOT> r2) и стрелки Пирса (<pi> (r1, r2) ≡ <NOT> r1 <AND> <NOT> r2).

Поэтому можно свести набор операций Алгебры А к трем операциям: <sh> (или <pi>), <RENAME> и <REMOVE>.

Избыточность операции переименования

Наконец, покажем, что избыточна и операция <RENAME>. Для иллюстрации снова воспользуемся отношением СЛУЖАЩИЕ. Пусть нам нужен результат операции СЛУЖАЩИЕ <RENAME> (ПРО_НОМ, НОМЕР_ПРОЕКТА) (мы по-прежнему предполагаем, что множество значений домена атрибута ПРО_НОМ ограничено значениями, представленными в теле отношения СЛУЖАЩИЕ). Возьмем бинарное отношение ПРО_НОМ_НОМЕР_ПРОЕКТА, где каждый из кортежей содержит два одинаковых значения номера проекта и в тело отношения входят все значения домена атрибута ПРО_НОМ.Вычисление выражения (СЛУЖАЩИЕ <AND> ПРО_НОМ_НОМЕР_ПРОЕКТА) <REMOVE> (ПРО_НОМ) приводит к желаемому результату.

Тем самым, можно сократить набор операций Алгебры А до двух операций: <sh> (или <pi>) и <REMOVE>

Базисом Алгебры А являются операции реляционного отрицания (дополнения), реляционной конъюнкции (или дизъюнкции) и проекции (удаления атрибута). Реляционные аналоги логических операций определяются в терминах отношений на основе обычных теоретико-множественных операций и позволяют выражать напрямую операции пересечения, декартова произведения, естественного соединения и объединения отношений. Путем комбинирования базовых операций выражаются операции переименования атрибутов, соединения общего вида, взятия разности отношений. Алгебра А позволяет лучше осознать логические основы реляционной модели, хотя, безусловно, является в меньшей степени ориентированной на практическое применение, чем алгебра Кодда

19. Реляционное исчисление кортежей.

В исчислении кортежей областями определения переменных являются тела отношений базы данных, т. е. допустимым значением каждой переменной является кортеж тела некоторого отношения.

Для определения кортежной переменной используется оператор `RANGE`. Например, для того чтобы определить переменную СЛУЖАЩИЙ, областью определения которой является отношение СЛУЖАЩИЕ, нужно употребить конструкцию

```
RANGE СЛУЖАЩИЙ IS СЛУЖАЩИЕ
```

Как уже говорилось, из этого определения следует, что в любой момент времени переменная СЛУЖАЩИЙ представляет некоторый кортеж отношения СЛУЖАЩИЕ. При использовании кортежных переменных в формулах можно ссылаться на значение атрибута переменной. Например, для того, чтобы сослаться на значение атрибута СЛУ_ИМЯ переменной СЛУЖАЩИЙ, нужно употребить конструкцию СЛУЖАЩИЙ.СЛУ_ИМЯ.

Правильно построенная формула (Well-Formed Formula, WFF) служит для выражения

условий, накладываемых на кортежные переменные.

Простые условия

Основой WFF являются **простые условия**, представляющие собой операции сравнения скалярных значений (значений атрибутов переменных или литерально заданных констант). Например, конструкции

```
СЛУЖАЩИЙ.СЛУ_НОМ = 2934
```
и

```
СЛУЖАЩИЙ.СЛУ_НОМ = ПРОЕКТ.ПРОЕКТ_РУК
```

являются простыми условиями. Первое условие принимает значение `true` в том и только в том случае, когда значение атрибута `СЛУ_НОМ` кортежной переменной `СЛУЖАЩИЙ` равно `2934`. Второе условие принимает значение `true` в том и только в том случае, когда значения атрибутов `СЛУ_НОМ` и `ПРОЕКТ_РУК` переменных `СЛУЖАЩИЙ` и `ПРОЕКТ` совпадают.

По определению, простое сравнение является WFF, а WFF, заключенная в круглые скобки, представляет собой простое сравнение.

Более сложные варианты WFF строятся с помощью логических связок `NOT`, `AND`, `OR` и `IF ... THEN` IF *a* THEN *b* = NOT *a* OR *b*

с учетом обычных приоритетов операций (`NOT > AND > OR`) и возможности расстановки скобок. Так, если `form` – WFF, а `comp` – простое сравнение, то `NOT form`, `comp AND form`, `comp OR form` и `IF comp THEN form` являются WFF.

Кванторы, свободные и связанные переменные

При построении WFF **допускается использование кванторов существования** (`EXISTS`) и **всеобщности** (`FORALL`). Если `form` – это WFF, в которой участвует переменная `var`, то конструкции `EXISTS var (form)` и `FORALL var (form)` представляют собой WFF. По определению, формула `EXISTS var (form)` принимает значение `true` в том и только в том случае, если в области определения переменной `var` найдется хотя бы одно значение (кортеж), для которого WFF `form` принимает значение `true`. Формула `FORALL var (form)` принимает значение `true`, если для всех значений переменной `var` из ее области определения WFF `form` принимает значение `true`.

Переменные, входящие в WFF, **могут быть свободными или связанными**. По определению, 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` является **связанной переменной.** 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 область ее определения.

На самом деле, **правильнее говорить не о свободных и связанных переменных, а о свободных и связанных вхождениях переменных**. Если переменная `var` является связанной в WFF `form`, то во всех WFF, включающих `form`, вне `form` может использоваться вхождение того же имени переменной `var`, которое может быть свободным или связанным, но в любом случае не имеет никакого отношения к вхождению переменной `var` в WFF `form`. Вот пример:

```
EXISTS СЛУ2 (СЛУ1.ПРО_НОМ = СЛУ2.ПРО_НОМ
  AND СЛУ1.СЛУ_НОМЕР = СЛУ2.СЛУ_НОМЕР)
  AND FORALL СЛУ2 (IF СЛУ1.ПРО_НОМ = СЛУ2.ПРО_НОМ
    THEN СЛУ1.СЛУ_ЗАРП = СЛУ2.СЛУ_ЗАРП)
```

Эта формула принимает значение `true` только для тех значений переменной `СЛУ1`, которые соответствуют служащим, участвующим в проектах с более чем одним участником, причем все участники проекта получают одну и ту же зарплату. Здесь мы имеем два связанных вхождения переменной `СЛУ2` с совершенно разным смыслом. Грубо говоря, для текущего значения переменной `СЛУ1` переменная `СЛУ2` два раза «пробежит» свою область определения – первый раз при вычислении части формулы с квантором существования, а второй при вычислении части с квантором всеобщности. Кстати, к тому же результату приведет формула с одним квантором всеобщности вида:

```
FORALL СЛУ2 (IF (СЛУ1.ПРО_НОМ = СЛУ2.ПРО_НОМ AND
     СЛУ1.СЛУ_НОМЕР ≠ СЛУ2.СЛУ_НОМЕР)
  THEN СЛУ1.СЛУ_ЗАРП = СЛУ2.СЛУ_ЗАРП)
```

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 атрибутов 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 используется несколько свободных переменных с одинаковой областью определения. Фактически применение целевого списка к области истинности WFF эквивалентно действию алгебраической операции проекции, а последний из приведенных вариантов представляет собой некоторую разновидность алгебраической операции переименования атрибута.

Выражением реляционного исчисления кортежей называется конструкция вида `target_list WHERE WFF`. Значением выражения является отношение, тело которого определяется WFF, а множество атрибутов и их имена – целевым списком.

В качестве простого примера покажем выражение реляционного исчисления кортежей, результат которого совпадает с результатом операции СЛУЖАЩИЕ `DIVIDE BY` НОМЕРА_ПРОЕКТОВ:

```
СЛУ1, СЛУ2 RANGE IS СЛУЖАЩИЕ
НОМЕР_ПРОЕКТА range is НОМЕРА_ПРОЕКТОВ
СЛУ1.СЛУ_НОМЕР, СЛУ1.СЛУ_ИМЯ, СЛУ1.СЛУ_ЗАРП
WHERE FORALL НОМЕР_ПРОЕКТА EXISTS СЛУ2
  (СЛУ1.СЛУ_НОМЕР = СЛУ2.СЛУ_НОМЕР AND
  СЛУ1.ПРО_НОМ = НОМЕРА_ПРОЕКТОВ.ПРО_НОМ)
```

Конечно, результатом этого выражения является отношение

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП
2934	Иванов	22400.00
2935	Петров	29600.00

20. Реляционное исчисление доменов.

В исчислении доменов **областью определени**я переменных являются не отношения, а **домены**. Применительно к базе данных СЛУЖАЩИЕ-ПРОЕКТЫ можно говорить, например, о доменных переменных ИМЯ (значения – допустимые имена) или НОСЛУ (значения – допустимые номера служащих).

Условия членства

Основным формальным отличием исчисления доменов от исчисления кортежей является наличие дополнительного множества предикатов, позволяющих выражать так называемые **условия членства**. Если R – это *n*-арное отношение с атрибутами a_1, a_2, ..., *an*, то условие членства имеет вид R (ai_1 : vi_1, ai_2 :

vi_2, ..., *aim* : v*im*) ($m \leq n$), где v*ij* – это либо литерально задаваемая константа, либо имя доменной 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 *aij* накладывается 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.

Для большей ясности приведем пару примеров. Для простоты будем считать, что мы определили доменные переменные, имена которых совпадают с именами атрибутов отношения СЛУЖАЩИЕ, а в случае, когда требуется несколько доменных переменных, определенных на одном домене, мы будем добавлять в конце имени цифры. WFF исчисления доменов

```
СЛУЖАЩИЕ (СЛУ_НОМ:2934, СЛУ_ИМЯ:'Иванов',
  СЛУ_ЗАРП:22400.00, ПРО_НОМ:1)
```

примет значение true в том и только в том случае, когда в теле отношения СЛУЖАЩИЕ содержится кортеж <2934, 'Иванов', 22400.00, 1>. Соответствующие значения доменных переменных образуют область истинности этой WFF. С другой стороны, WFF

```
СЛУЖАЩИЕ (СЛУ_НОМ:2934, СЛУ_ИМЯ:'Иванов',
  СЛУ_ЗАРП:22400.00, ПРО_НОМ:ПРО_НОМ)
```

будет принимать значение true для всех комбинаций явно заданных значений и допустимых значений переменной ПРО_НОМ, которые соответствуют кортежам, входящим в тело отношения СЛУЖАЩИЕ.

Выражения исчисления доменов

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

Для примера выражения исчисления доменов сформулируем с использованием исчисления доменов запрос «Выдать номера и имена служащих, не получающих минимальную заработную плату»:

```
СЛУ_НОМ, СЛУ_ИМЯ WHERE EXISTS СЛУ_ЗАРП1
(СЛУЖАЩИЕ (СЛУ_ЗАРП1) AND
  СЛУЖАЩИЕ (СЛУ_НОМ, СЛУ_ИМЯ, СЛУ_ЗАРП) AND
  СЛУ_ЗАРП > СЛУ_ЗАРП1)
```

Реляционное исчисление доменов является основой большинства языков запросов,

основанных на использовании форм.

21. Функциональные зависимости, замыкание множества функциональных зависимостей, аксиомы Армстронга, замыкание множества атрибутов. Минимальное покрытие множества.

Пусть задана переменная отношения *R*, и X и Y являются произвольными подмножествами заголовка *R* («составными» атрибутами).

В значении переменной отношения *R* **атрибут Y функционально зависит от атрибута X** в том и только в том случае, если каждому значению X соответствует в точности одно значение Y. В этом случае говорят также, что атрибут X **функционально определяет** атрибут Y (X является **детерминантом (*определителем*)** для Y, а Y **является зависимым** от X). Будем обозначать это как $R.X \longrightarrow R.Y$.

FD группы, которые должны быть верны для любого допустимого значения переменной отношения могут рассматриваться как *инварианты*, или *ограничения целостности* этой переменной отношения.

В дальнейшем нас будут интересовать только те функциональные зависимости, которые должны выполняться для всех возможных значений переменных отношений.

Заметим, что если атрибут A отношения *R* является возможным ключом, то для любого атрибута B этого отношения всегда выполняется FD $A \longrightarrow B$ (в группе (1) к этим FD относятся все FD, детерминантом которых является СЛУ_НОМ).

Итак, мы будем иметь дело с FD, которые выполняются для всех возможных состояний тела соответствующего отношения и могут рассматриваться как ограничения целостности. Поскольку они трактуются как ограничения целостности, за их соблюдением должна следить СУБД. Поэтому важно уметь сократить набор FD до минимума, поддержка которого гарантирует выполнение всех зависимостей. Мы займемся этим в следующих подразделах.

FD $A \longrightarrow B$ называется **тривиальной**, если $A \supseteq B$ (т. е. множество атрибутов A включает множество B или совпадает с множеством B).

Очевидно, что любая тривиальная FD всегда выполняется. Например, в отношении СЛУЖАЩИЕ_ПРОЕКТЫ всегда выполняется FD $\{СЛУ\_ЗАРП, ПРО\_НОМ\} \longrightarrow СЛУ\_ЗАРП$. Частным случаем тривиальной FD является $A \longrightarrow A$.

Поскольку тривиальные FD выполняются всегда, их нельзя трактовать как ограничения целостности, и поэтому они не представляют интереса с практической точки зрения.

Замыкание множества функциональных зависимостей. Аксиомы Армстронга. Замыкание множества атрибутов

Замыканием множества *FD S* является множество FD S^+, включающее все FD, логически выводимые из FD множества S.

Для начала приведем два примера FD, из которых следуют (или *выводятся*) другие FD. Будем снова пользоваться отношением СЛУЖАЩИЕ_ПРОЕКТЫ. Для этого отношения выполняется, например, FD СЛУ_НОМ⟶{СЛУ_ЗАРП, ОТД_НОМ}. Из этой FD выводятся FD СЛУ_НОМ⟶СЛУ_ЗАРП и СЛУ_НОМ⟶ОТД_НОМ.

В отношении СЛУЖАЩИЕ_ПРОЕКТЫ имеется также пара FD СЛУ_НОМ⟶ОТД_НОМ и ОТД_НОМ⟶ПРОЕКТ_РУК. Из них выводится FD СЛУ_НОМ⟶ПРОЕКТ_РУК. Заметим, что FD вида СЛУ_НОМ⟶ПРОЕКТ_РУК называются транзитивными, поскольку ПРОЕКТ_РУК зависит от СЛУ_НОМ «транзитивно», через ПРО_НОМ.

FD A⟶C называется **транзитивной**, если существует такой атрибут B, что имеются функциональные зависимости A⟶B и B⟶C и отсутствует функциональная зависимость C⟶A.

Подход к решению проблемы поиска замыкания S^+ множества FD S впервые предложил Вильям Армстронг. Им был предложен **набор правил вывода новых FD из существующих** (эти правила обычно называют аксиомами Армстронга, хотя справедливость правил доказывается на основе определения FD). Обычно принято формулировать эти правила вывода в следующей форме. Пусть A, B и C являются (в общем случае, составными) атрибутами отношения *R*. Множества A, B и C могут иметь непустое пересечение. Для краткости будем обозначать через AB A UNION B. Тогда:

1. если B⊆A, то A⟶B (**рефлексивность**);
2. если A⟶B, то AC⟶BC (**пополнение**);
3. если A⟶B и B⟶C, то A⟶C (**транзитивность**).

Истинность первой аксиомы Армстронга следует из того, что при B⊆A FD A⟶B является **тривиальной.**

Справедливость **второй** аксиомы **докажем от противного**. Предположим, что FD AC⟶BC не соблюдается. Это означает, что в некотором допустимом теле отношения найдутся два кортежа t_1 и t_2, такие, что t_1 {AC} = t_2 {AC} (a), но t_1 {BC} ≠

t_2 {BC} (b) (здесь t {A} обозначает проекцию кортежа t на множество атрибутов A). По аксиоме рефлексивности из равенства (a) следует, что t_1 {A} = t_2 {A}. Поскольку имеется FD A⟶B, должно соблюдаться равенство t_1 {B} = t_2 {B}. Тогда из неравенства (b) следует, что t_1 {C} ≠ t_2 {C}, что противоречит наличию тривиальной FD AC⟶C. Следовательно, предположение об отсутствии FD AC⟶BC не является верным, и справедливость второй аксиомы доказана.

Аналогично докажем истинность третьей аксиомы Армстронга. Предположим, что FD A⟶C не соблюдается. Это означает, что в некотором допустимом теле отношения найдутся два кортежа t_1 и t_2, такие, что t_1 {A} = t_2 {A}, но t_1 {C} ≠ t_2 {C}. Но из наличия FD A⟶B следует, что t_1 {B} = t_2 {B}, а потому из наличия FD B⟶C следует, что t_1 {C} = t_2 {C}. Следовательно, предположение об отсутствии FD A⟶C не является верным, и справедливость третьей аксиомы доказана.

Можно доказать, что система правил вывода Армстронга **полна и совершенна** *(sound and complete)* в том смысле, что для данного множества FD S любая FD, потенциально выводимая из S, может быть выведена на основе аксиом Армстронга, и применение этих аксиом не может привести к выводу лишней FD. Тем не менее Дейт по практическим соображениям предложил расширить базовый набор правил вывода **еще пятью правилами:**

4. A⟶A (**самодетерминированность**) – прямо следует из правила (1);
5. если A⟶BC, то A⟶B и A⟶C (**декомпозиция**) – из правила (1) следует, что BC ⟶B; по правилу (3) A⟶B; аналогично, из BC⟶C и правила (3) следует A⟶C;
6. если A⟶B и A⟶C, то A⟶BC (**объединение**) – из правила (2) следует, что A⟶ AB и AB⟶BC; из правила (3) следует, что A⟶BC;
7. если A⟶B и C⟶D, то AC⟶BD (**композиция**) – из правила (2) следует, что AC ⟶BC и BC⟶BD; из правила (3) следует, что AC⟶BD;
8. если A⟶BC и B⟶D, то A⟶BCD (**накопление**) – из правила (2) следует, что BC ⟶BCD; из правила (3) следует, что A⟶BCD.

Пусть заданы отношение *R*, множество Z атрибутов этого отношения (подмножество заголовка *R*, или составной атрибут *R*) и некоторое множество FD S, выполняемых для *R*. Тогда **замыканием Z над S** называется наибольшее множество Z^+ таких атрибутов Y отношения *R*, что FD Z⟶Y входит в S^+.

Алгоритм вычисления Z^+ очень прост. Один из его вариантов показан на

```
K := 0; Z[0] := Z;
DO
   K := K+1;
   Z[K] := Z[K-1];
   FOR EACH FD A→B IN S DO
      IF A⊆Z[K] THEN Z[K] := (Z[K] UNION B) END DO;
UNTIL Z[K] = Z[K-1];
Z+ := Z[K];
```

Алгоритм построения замыкания атрибутов над заданным множеством FD

Докажем корректность алгоритма по индукции. На нулевом шаге `Z[0] = Z`, FD `Z`⟶`Z[I]`, очевидно, принадлежит S^+ (тривиальная FD «выводится» из любого множества FD). Пусть для некоторого `K` выполняется FD `Z`⟶`Z[K]`, и пусть мы нашли в `S` такую FD `A`⟶`B`, что `A`⊆`Z[K]`. Тогда можно представить `Z[K]` в виде `AC`, и, следовательно, выполняется FD `Z`⟶`AC`. Но по правилу (8) мы имеем FD `Z`⟶`ACB`, т.е. FD `Z`⟶`(Z[K] UNION B)` входит во множество S^+, что переводит нас на следующий шаг индукции.

Пусть для примера имеется отношение с заголовком `{A, B, C, D, E, F}` и заданным множеством FD `S = {A`⟶`D, AB`⟶`E, BF`⟶`E, CD`⟶`F, E`⟶`C}`. Пусть требуется найти $\{AE\}^+$ над `S`. На первом проходе тела цикла DO `Z[1]` равно `AE`. В теле цикла `FOR EACH` будут найдены FD `A`⟶`D` и `E`⟶`C`, и в конце цикла `Z[1]` станет равным `ACDE`. На втором проходе тела цикла DO при `Z[2]`, равном `ACDE`, в теле цикла `FOR EACH` будет найдена FD `CD`⟶`F`, и в конце цикла `Z[2]` станет равным `ACDEF`. Следующий проход тела цикла DO не изменит `Z[3]`, и Z^+ ($\{AE\}^+$) будет равно `ACDEF`.

Алгоритм построения замыкания множества атрибутов `Z` над заданным множеством FD `S` помогает легко установить, входит ли заданная FD `Z`⟶`B` в замыкание S^+. Очевидно, что **необходимым и достаточным условием** для этого является `B`⊆Z^+, т. е. вхождение составного атрибута `B` в замыкание `Z`.

Суперключом отношения *R* называется любое подмножество `K` заголовка *R*, включающее, по меньшей мере, хотя бы один возможный ключ *R*.

Одно из следствий этого определения состоит в том, что подмножество `K` заголовка отношения *R* является суперключом тогда и только тогда, когда для любого атрибута `A` (возможно, составного) заголовка отношения *R* выполняется FD `K`⟶`A`. В терминах

замыкания множества атрибутов K является суперключом тогда и только тогда, когда K^+ совпадает с заголовком *R*.

Минимальное покрытие множества функциональных зависимостей

Множество FD S2 называется **покрытием множества** *FD* S1, если любая FD, выводимая из S1, выводится также из S2.

Легко заметить, что S2 является покрытием S1 тогда и только тогда, когда $S1^+ \subseteq S2^+$. Два множества FD S1 и S2 называются **эквивалентными**, если каждое из них является покрытием другого, т. е. $S1^+ = S2^+$.

Множество *FD S* **называется минимальным** в том и только в том случае, когда удовлетворяет следующим свойствам:

1. правая часть любой FD из S является множеством из одного атрибута (простым атрибутом);
2. детерминант каждой FD из S обладает свойством *минимальности*; это означает, что удаление любого атрибута из детерминанта приводит к изменению замыкания S^+, т. е. порождению множества FD, не эквивалентного S;
3. удаление любой FD из S приводит к изменению S^+, т. е. порождению множества FD, не эквивалентного S.

Для любого множества FD S существует (и даже может быть построено) эквивалентное ему минимальное множество S^-.

Приведем общую схему построения S^- по заданному множеству FD S. Во-первых, используя правило (5) (декомпозиции), мы можем привести множество S к эквивалентному множеству FD S1, правые части FD которого содержат только одноэлементные множества (простые атрибуты). Далее, для каждой FD из S1, детерминант D $\{D_1, D_2, \ldots, Dn\}$ которой содержит более одного атрибута, будем пытаться удалять атрибуты D_i, получая множество FD S2. Если после удаления атрибута D_i S2 эквивалентно S1, то этот атрибут удаляется, и пробуется следующий атрибут. Назовем S3 множество FD, полученное путем допустимого удаления атрибутов из всех детерминантов FD множества S1. Наконец, для каждой FD f из множества S3 будем проверять эквивалентность множеств S3 и S3 MINUS {f}. Если эти множества эквивалентны, удалим f из множества S3, и в заключение получим множество S4, которое минимально и эквивалентно исходному множеству FD S.

Пусть, например, имеется отношение *R* {A, B, C, D} и задано множество FD S = {A⟶B, A⟶BC, AB⟶C, AC⟶D, B⟶C}. По правилу декомпозиции S эквивалентно множеству S1 {A⟶B, A⟶C, AB⟶C, AC⟶D, B⟶C}. В

детерминанте FD AC⟶D можно удалить атрибут C, поскольку по правилу дополнения из FD A⟶C следует A⟶AC; по правилу транзитивности выводится FD A⟶D, поэтому атрибут C в детерминанте FD AC⟶D является избыточным. FD AB⟶C может быть удалена, поскольку может быть выведена из FD A⟶C (по правилу пополнения из этой FD выводится AB⟶BC, а по правилу декомпозиции далее выводится AB⟶C). Наконец, FD A⟶C тоже выводится по правилу транзитивности из FD A⟶B и B⟶C. Таким образом, мы получаем множество зависимостей {A⟶B, A⟶D, B⟶C}, которое является минимальным и эквивалентно S по построению.

Минимальным покрытием множества *FD S* называется любое минимальное множество FD S1, эквивалентное S.

Поскольку для каждого множества FD существует эквивалентное минимальное множество FD, **у каждого множества FD имеется хотя бы одно минимальное покрытие,** причем для его нахождения не обязательно находить замыкание исходного множества.

22. Декомпозиция без потерь и функциональные зависимости, теорема Хита.

Декомпозиция – разбиение путем проецирования отношения, находящегося в предыдущей нормальной форме, на два или более отношений, удовлетворяющих требованиям следующей нормальной формы.

Считаются правильными такие декомпозиции отношения, которые обратимы, т. е. имеется возможность собрать исходное отношение из декомпозированных отношений без потери информации. Такие декомпозиции называются **декомпозициями без потерь**.

Теорема Хита.

Пусть задано отношение *r* {A, B, C} (A, B и C, в общем случае, являются составными атрибутами) и выполняется FD A⟶B.

Тогда *r* = (*r* PROJECT {A, B}) NATURAL JOIN (*r* PROJECT {A, C}).

Доказательство. Прежде всего, докажем, что в теле результата естественного соединения (обозначим этот результат через r_1) содержатся все кортежи тела отношения *r*. Действительно, пусть кортеж {a, b, c} $\in$ *r*. Тогда по определению операции взятия проекции {a, b} $\in$ (*r* PROJECT {A, B}) и {a, C} $\in$ (r PROJECT {A, C}). Следовательно, {a, b, c} $\in$ r_1. Теперь докажем, что в теле результата естественного соединения нет лишних кортежей, т. е. что если кортеж {a,

b, c} ∈ r1, то {a, b, c} ∈ r. Если {a, b, c} ∈ r_1, то существуют {a, b} ∈ (r PROJECT {A, B}) и {a, c} ∈ (r PROJECT {A, C}). Последнее условие может выполняться в том и только в том случае, когда существует кортеж {a, b*, c} ∈ r. Но поскольку выполняется FD A⟶B, то b = b* и, следовательно, {a, b, c} = {a, b*, c}.

23. Проектирование реляционных баз данных с использованием нормализации: первая, вторая и третья нормальные формы.

Весь процесс проектирования базы данных осуществляется 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 «улучшенными» свойствами. Процесс проектирования представляет собой процесс нормализации схем отношений, причем каждая следующая нормальная форма обладает свойствами, в некотором смысле, лучшими, чем предыдущая.

Атомарность **значения** трактуется в том смысле, что значение типизировано, и с этим значением можно работать только с помощью операций соответствующего типа данных.

Каждой нормальной форме соответствует определенный набор ограничений, и отношение находится в некоторой нормальной форме, если удовлетворяет свойственному ей набору ограничений. Примером может служить **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 состоят в следующем:

- **каждая следующая нормальная форм**а в некотором смысле **лучше предыдущей** нормальной формы;
- при переходе к следующей нормальной форме **свойства предыдущих нормальных форм сохраняются**.

В основе процесса проектирования лежит метод нормализации, т. е. декомпозиции отношения, находящегося в предыдущей нормальной форме, на два или более.

FD с минимальным детерминантом называется **минимальной слева**.

Атрибут B минимально зависит от атрибута A, если выполняется минимальная слева FD A⟶B.

Если некоторые функциональные зависимости атрибутов от возможного ключа не являются минимальными). Это приводит к так называемым аномалиям обновления. Под **аномалиями обновления** понимаются трудности, с которыми приходится сталкиваться при выполнении операций добавления кортежей в отношение (`INSERT`), удаления кортежей (`DELETE`) и модификации кортежей (`UPDATE`).

Переменная отношения находится во **второй нормальной форме** *(2NF)* тогда и только тогда, когда она находится в первой нормальной форме, и каждый неключевой атрибут минимально функционально зависит от первичного ключа.

Любая переменная отношения, находящаяся в 1NF, но не находящаяся в 2NF, может быть приведена к набору переменных отношений, находящихся в 2NF. В результате декомпозиции мы получаем набор проекций исходной переменной отношения, естественное соединение значений которых воспроизводит значение исходной переменной отношения (т. е. это декомпозиция без потерь).

Функциональные зависимости переменной отношения порождают некоторые аномалии обновления, если существует транзитивная FD

Переменная отношения находится в **третьей нормальной форме (3NF)** в том и только в том случае, когда она находится во второй нормальной форме, и каждый неключевой атрибут нетранзитивно функционально зависит от первичного ключа.

Любое отношение, находящееся в 2NF, но не находящееся в 3NF, может быть приведено к набору отношений, находящихся в 3NF. Мы получаем набор проекций исходного отношения, естественное соединение которых воспроизводит исходное отношение (т. е. это декомпозиция без потерь).

24. Проектирование реляционных баз данных с использованием нормализации: теорема Риссонена, нормальная форма Бойса-Кодда.

Отношения являются независимыми проекциями отношения, если они могут обновляться независимо и при этом результат их естественного соединения всегда будет таким, как если бы обновлялось исходное отношение.

Теорема Риссанена

Проекции r_1 и r_2 отношения r являются **независимыми** тогда и только тогда, когда:

- каждая FD в отношении r логически следует(Т.е. выводится на основе аксиом Армстронга) из FD в r_1 и r_2;
- общие атрибуты r_1 и r_2 образуют возможный ключ хотя бы для одного из этих отношений.

Атомарным отношением называется отношение , которое невозможно декомпозировать на независимые проекции. Далеко не всегда для неатомарных (не являющихся атомарными) отношений требуется декомпозиция на атомарные проекции. Например, отношение СЛУЖ2 {СЛУ_НОМ, СЛУ_ЗАРП, ПРО_НОМ} с множеством FD {СЛУ_НОМ⟶СЛУ_ЗАРП, СЛУ_НОМ⟶ПРО_НОМ} не является атомарным (возможна декомпозиция на независимые проекции СЛУЖ3 {СЛУ_НОМ, СЛУ_ЗАРП} и СЛУЖ4 {СЛУ_НОМ, ПРО_НОМ}). Но эта декомпозиция не улучшает свойства отношения СЛУЖ2 и поэтому не является осмысленной. Другими словами, при выборе способа декомпозиции нужно стремиться к получению независимых проекций, но не обязательно атомарных.

До сих пор в определениях нормальных форм мы предполагали, что у декомпозируемого отношения имеется только один возможный ключ. На практике чаще всего бывает именно так. Но имеется один частный случай, который (почти) удовлетворяет требованиям 2NF и 3NF, но, тем не менее, порождает аномалии обновления. Это тот случай, когда у отношения имеется несколько возможных ключей, и некоторые из этих возможных ключей «перекрываются», т. е. содержат общие атрибуты.

Причиной аномалий является то, что **в требованиях 2NF и 3NF не требовалась минимальная функциональная зависимость от первичного ключа атрибутов, являющихся компонентами других возможных ключей**. Проблему решает нормальная форма, которую исторически принято называть нормальной формой Бойса-Кодда и которая является уточнением 3NF в случае наличия нескольких перекрывающихся возможных ключей.

Переменная отношения находится в **нормальной форме Бойса-Кодда (BCNF)** в том и только в том случае, когда любая выполняемая для этой переменной отношения нетривиальная и минимальная FD имеет в качестве детерминанта некоторый

возможный ключ данного отношения.

25. Многозначные зависимости, теорема Фейджина, четвертая

мы имеем дело с новым видом зависимости, впервые обнаруженным Роном Фейджином в 1971 г. Фейджин назвал зависимости этого вида многозначными (multi-valued dependency – MVD). Как мы увидим немного позже, MVD является обобщением понятия FD.

[В отношении СЛУЖ_ПРО_ЗАДАН выполняются две MVD: СЛУ_НОМ$\twoheadrightarrow$ПРО_НОМ и СЛУ_НОМ$\twoheadrightarrow$СЛУ_ЗАДАН. Первая MVD означает, что каждому значению атрибута СЛУ_НОМ соответствует определяемое только этим значением множество значений атрибута ПРО_НОМ. Другими словами, в результате вычисления алгебраического выражения

(СЛУЖ_ПРО_НОМ WHERE (СЛУ_НОМ = СН AND СЛУ_ЗАДАН = СЗ)) PROJECT {ПРО_НОМ}

для фиксированного допустимого значения СН и любого допустимого значения СЗ мы всегда получим одно и то же множество значений атрибута ПРО_НОМ. Аналогично трактуется вторая MVD.]

В переменной отношения *R* с атрибутами A, B, C (в общем случае, составными) имеется **многозначная зависимость** B от A (A$\twoheadrightarrow$B) в том и только в том случае, когда множество значений атрибута B, соответствующее паре значений атрибутов A и C, зависит от значения A и не зависит от значения C.

Лемма Фейджина

В отношении *R* {A, B, C} выполняется MVD A$\twoheadrightarrow$B в том и только в том случае, когда выполняется MVD A$\twoheadrightarrow$C.

Доказательство достаточности условия леммы. Пусть выполняется MVD A$\twoheadrightarrow$B. Пусть имеется некоторое удовлетворяющее этой зависимости значение *r* переменной отношения *R*, a обозначает значение атрибута A в некотором кортеже тела B_r, $a\ \{b\}$ – множество значений атрибута B, взятых из всех кортежей B_r, в которых значением атрибута A является a. Предположим, что для этого значения a MVD A$\twoheadrightarrow$C не выполняется. Это означает, что существуют такое допустимое значение c атрибута C и такое значение $b \in \{b\}$, что кортеж $\{a, b, c\} \notin B_r$. Но это противоречит наличию MVD A$\twoheadrightarrow$B. Следовательно, если выполняется MVD A$\twoheadrightarrow$B, то выполняется и

MVD A$\rightarrow\!\!\!\!\rightarrow$C. Аналогично можно доказать необходимость условия леммы.

Таким образом, MVD A$\rightarrow\!\!\!\!\rightarrow$B и A$\rightarrow\!\!\!\!\rightarrow$C всегда составляют пару. Поэтому обычно их представляют вместе в форме A $\rightarrow\!\!\!\!\rightarrow$ B | C.

FD является частным случаем MVD, когда множество значений зависимого атрибута обязательно состоит из одного элемента. Таким образом, если выполняется FD A$\rightarrow$B, то выполняется и MVD A$\rightarrow\!\!\!\!\rightarrow$B .

Теорема Фейджина

Пусть имеется переменная отношения *R* с атрибутами A, B, C (в общем случае, составными). Отношение *R* декомпозируется без потерь на проекции {A, B} и {A, C} тогда и только тогда, когда для него выполняется MVD A $\rightarrow\!\!\!\!\rightarrow$ B | C.

Докажем достаточность условия теоремы. Пусть *r* является некоторым допустимым значением переменной отношений *R*. Пусть a есть значение атрибута A в некотором кортеже тела B_r, {b} – множество значений атрибута B, взятых из всех кортежей тела B_r, в которых значением атрибута A является a, и {c} – множество значений атрибута C, взятых из всех кортежей тела B_r, в которых значением атрибута A является a. Тогда очевидно, что в тело значения *r* PROJECT {A, B} будут входить все кортежи вида {a, b_i}, где $b_i \in$ {b}, и если некоторый кортеж {a, b_j} входит в тело значения отношения *r* PROJECT {A, B}, то $b_j \in$ {b}. Аналогичные рассуждения применимы к *r* PROJECT {A, C}. Очевидно, что из этого следует, что при наличии многозначной зависимости A $\rightarrow\!\!\!\!\rightarrow$ B | C в переменной отношения *R*{A, B, C} декомпозиция *r* на проекции *r* PROJECT {A, B} и *r* PROJECT {A, C} является декомпозицией без потерь.

Для доказательства необходимости условия теоремы предположим, что декомпозиция переменной отношения *R* {A, B, C} на проекции *R* PROJECT {A, B} и *R* PROJECT {A, C} является декомпозицией без потерь для любого допустимого значения *r* переменной отношения *R*. Мы должны показать, что в теле B_r значения-отношения *r* поддерживается ограничение

IF ({a, b_1, c_1}; $\in B_r$ AND {a, b_2, c_2} $\in B_r$)

THEN ({a, b_1, c_2} $\in B_r$ AND {a, b_2, c_1} $\in B_r$)

Действительно, пусть в B_r входят кортежи {a, b_1, c_1} и {a, b_2, c_2}. Предположим, что {a, b_1, c_2} $\notin B_r$ OR a, b_2, c_1 $\notin B_r$. Но в тело значения

отношения *r* PROJECT {A, B} входят кортежи {a, b_1} и {a, b_2}, а в тело значения переменной отношения *r* PROJECT {A, C} – {a, c_1} и {a, c_2};. Очевидно, что в тело значения естественного соединения *r* PROJECT {A, B} NATURAL JOIN *r* PROJECT {A, C} войдут кортежи {a, b_1, c_2} и {a, b_2, c_1}, и наше предположение об отсутствии по крайней мере одного из этих кортежей в B_r противоречит исходному предположению о том, что декомпозиция *r* на проекции *r* PROJECT {A, B} и *r* PROJECT {A, C} является декомпозицией без потерь. Тем самым, теорема Фейджина полностью доказана.

Переменная отношения r находится в **четвертой нормальной форме (4NF)** в том и только в том случае, когда она находится в BCNF, и все MVD r являются FD с детерминантами – возможными ключами отношения r.

26. Зависимости проекции-соединения. Пятая нормальная форма.

Будем называть **n-декомпозируемым** отношением отношение, которое может быть декомпозировано без потерь на *n* проекций.

В переменной отношения *R* с атрибутами (возможно, составными) A и B MVD A$\twoheadrightarrow$B называется **тривиальной,** если либо A$\subseteq$B, либо A UNION B совпадает с заголовком отношения *R*.

Тривиальная MVD всегда удовлетворяется. При A$\subseteq$B она вырождается в тривиальную FD. В случае A UNION B = H*R* требования многозначной зависимости соблюдаются очевидным образом.

Пусть задана переменная отношения *R*, и A, B, …, Z являются произвольными подмножествами заголовка *R* (составными, перекрывающимися атрибутами). В переменной отношения *R* удовлетворяется **зависимость проекции/соединения** *(Project-Join Dependency – PJD)* *(A, B, …, Z) тогда и только тогда, когда любое допустимое значение *r* переменной отношения *R* можно получить путем естественного соединения проекций этого значения на атрибуты A, B, …, Z.

В переменной отношения *R* PJD *(A, B, …, Z) называется **подразумеваемой возможными ключами** в том и только в том случае, когда каждый составной атрибут A, B, …, Z является суперключом *R*, т. е. включает хотя бы один возможный ключ *R*.

В переменной отношения *R* зависимость проекции/соединения *(A, B, …, Z) называется **тривиальной**, если хотя бы один из составных атрибутов A, B, …, Z совпадает с заголовком *R*.

Легко убедиться, что нетривиальные PJD, подразумеваемые возможными ключами, существуют во всех отношениях с арностью, большей двух, первичный ключ которых не совпадает с заголовком отношения.

Переменная отношения *R* находится в **пятой нормальной форме, или в нормальной форме проекции/соединения** (5NF, или PJ/NF – Project-Join Normal Form) в том и только в том случае, когда каждая нетривиальная PJD в *R* подразумевается возможными ключами *R*.

Отношения, находящиеся в 4NF, как правило, находятся и в 5NF.

5NF является «окончательной» нормальной формой, которой можно достичь в процессе нормализации на основе проекций. «Окончательность» понимается в том смысле, что у отношения, находящегося в 5NF, отсутствуют аномалии обновлений, которые можно было бы устранить путем его декомпозиции. Такие отношения далее нормализовать бессмысленно.

Процесс проектирования реляционной базы на основе метода нормализации преследует две основных цели:

- **избежать избыточности хранения данных;**
- **устранить аномалии обновления отношений.**

27. Семантические модели данных.

Потребность проектировщиков баз данных в **более удобных и мощных средствах** моделирования предметной области привела к появлению семантических моделей данных. **Основным назначением** семантических моделей является **обеспечение возможности выражения семантики данных**. Чаще всего на практике семантическое моделирование используется на **первой стадии проектирования базы данных.** 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 .

Основным достоинством данного подхода является **отсутствие потребности в дополнительных программных средствах**, поддерживающих семантическое моделирование. Требуется только знание основ выбранной семантической модели и правил преобразования концептуальной схемы в реляционную схему.

Построение мощной и наглядной концептуальной схемы БД **позволяет более полно оценить специфику моделируемой предметной области** и избежать возможных ошибок на стадии проектирования схемы реляционной БД. На этапе семантического моделирования **производится важная документация** (хотя бы в виде вручную нарисованных диаграмм и комментариев к ним), которая может оказаться очень полезной не только при проектировании схемы реляционной БД, но и при эксплуатации, сопровождении и развитии уже заполненной БД.

Без семантического моделирования можно обойтись, если число таблиц не превышает десяти, но оно совершенно необходимо, если БД включает более сотни таблиц. Для справедливости заметим, что процедура создания концептуальной схемы вручную с ее последующим преобразованием в реляционную схему БД затруднительна в случае больших БД (содержащих несколько сотен таблиц)

История систем **автоматизации проектирования баз данных** (CASE-средств проектирования БД) **началась с автоматизации процесса рисования диаграмм**, проверки их формальной корректности, обеспечения средств долговременного хранения диаграмм и другой проектной документации.Наличие электронного архива проектной документации помогает при эксплуатации, администрировании и сопровождении базы данных. Так как имеется четкая методика преобразования концептуальной схемы БД в реляционную схему, то ее решили включить в состав системы проектирования баз данных.

Подавляющее **большинство** подобных систем, представленных на рынке, **обеспечивает автоматизированное преобразование диаграммных концептуальных схем баз данных**, представленных в той или иной семантической модели данных, **в реляционные схемы,** специфицированные чаще всего на языке SQL.

В типичной схеме SQL-ориентированной БД могут **содержаться определения многих объектов ограничений целостности общего вида**, триггеров и хранимых процедур, которые невозможно сгенерировать автоматически на основе концептуальной схемы. Поэтому **на завершающем этапе проектирования реляционной схемы снова требуется ручная работа проектировщика**.

.Если создатели семантической модели данных предоставляют язык (например, диаграммный), используя который проектировщики БД на основе исходной информации о предметной области могут сформировать концептуальную схему БД, то почему бы не реализовать программу, которая сама генерирует концептуальную схему БД в соответствующей семантической модели, используя исходную информацию о предметной области?

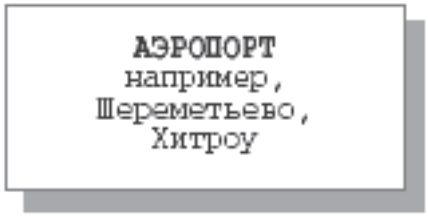
Хотя неизвестны коммерческие CASE-средства проектирования БД, поддерживающие такой подход, экспериментальные системы успешно существовали. Они представляли собой интегрированные системы проектирования с автоматизированным созданием концептуальной схемы на основе интервью с экспертами предметной области и последующим преобразованием концептуальной схемы в реляционную схему . Как правило, CASE-средства, автоматизирующие преобразование концептуальной схемы БД в реляционную, производят реляционную схему базы данных в третьей нормальной форме. Нормализация более высокого уровня усложняет программную реализацию и редко требуется на практике.

28. Семантическая модель Entity-Relationship.(Сущность-Связи)

Модель была предложена **Питером Ченом** (Peter Chen) в 1976 г. Моделирование предметной области **базируется на использовании графических диаграмм, включающих небольшое число разнородных компонентов**. Рассмотрим упрощенный

вариант этой диаграммной модели, достаточный для понимания основных особенностей проектирования реляционных баз данных с использованием ER-моделей

Основными понятиями ER-модели являются **сущность**, **связь** и **атрибут**. **Сущность** – это реальный или представляемый объект, информация о котором должна сохраняться и быть доступной. 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 экземпляров этого типа.



При определении типа сущности **необходимо гарантировать**, что **каждый экземпляр сущности может 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 типами сущностей. Как и сущность, связь – это **типовое понятие**, все экземпляры обоих связываемых типов сущностей подчиняются устанавливаемым правилам связывания. Поэтому **правильнее говорить о типе связи**, **устанавливаемой между типами сущности**, и об экземплярах типа связи, устанавливаемых между экземплярами типа сущности. В **обсуждаемом здесь варианте** ER-модели эта **ассоциация всегда является бинарной** и может существовать между двумя разными типами сущностей или между типом сущности и им же самим (рекурсивная связь). В любой связи **выделяются два конца** (в соответствии с существующей парой связываемых сущностей), на каждом из которых **указываются имя конца связи**, **степень конца связи** (сколько экземпляров данного типа сущности должно присутствовать в каждом экземпляре данного типа связи), **обязательность связи** (т. е. любой ли экземпляр данного типа сущности должен участвовать в некотором экземпляре данного типа связи).

Связь представляется в виде **ненаправленной линии**, соединяющей две сущности или ведущей от сущности к ней же самой. При этом в месте «стыковки» связи с сущностью используются:

- **трехточечный вход** в прямоугольник сущности, если для этой сущности в связи могут (или должны) использоваться **много** (*many*) экземпляров сущности;
- **одноточечный вход**, если в связи может (или должен) участвовать только **один** экземпляр сущности.

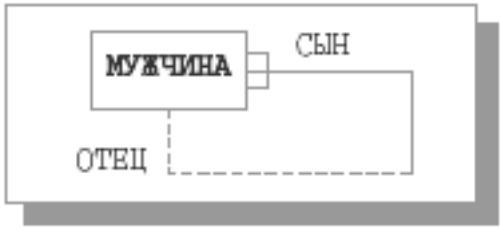
Обязательный конец связи изображается **сплошной** линией, а **необязательный** –

прерывистой линией.

Рис. 10.2. Пример типа связи

Лаконичная устная трактовка изображенной диаграммы состоит в следующем:

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

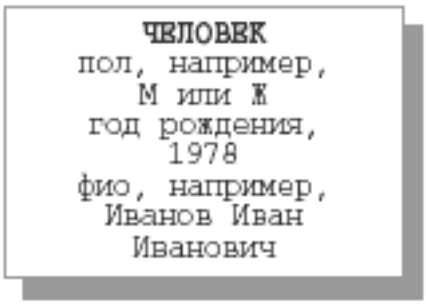


Лаконичная устная трактовка изображенной диаграммы состоит в следующем:

- каждый МУЖЧИНА является сыном одного и только одного МУЖЧИНЫ;
- каждый МУЖЧИНА может являться отцом одного или более МУЖЧИН.

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 типа сущности ЧЕЛОВЕК с указанными атрибутами показан на рисунке.



Пример типа сущности с атрибутами

С технической точки зрения **атрибуты** типа сущности в ER-модели **похожи** на атрибуты **отношения** в реляционной модели данных. И в том, и в другом случаях введение именованных атрибутов вводит некоторую типовую структуру данных, имя

которой совпадает с именем типа сущности в случае ER-модели или с именем переменной отношения в случае реляционной модели. Этой типовой структуре должны следовать все экземпляры типа сущности или все кортежи отношения. Но имеется и **важное отличие**. Напомним, что в реляционной модели данных атрибут определяется как упорядоченная пара <имя_атрибута, имя_домена> (или <имя_атрибута, имя_базового_типа_данных>, если понятие домена не поддерживается). Заголовок отношения, определяемый как множество таких пар, представляет собой полный аналог структурного типа данных в языках программирования.

При определении атрибутов типа сущности в ER-модели указание домена атрибута не является обязательным, хотя это и возможно. Обсудим, чем вызвана эта возможность «ослабленного» определения атрибутов. Прежде всего, как отмечалось в разделе , семантические **модели данных используются для построения концептуальных схем** БД, и эти схемы преобразуются в реляционные схемы БД, которые поддерживаются той или иной СУБД. Поскольку производители CASE-средств проектирования реляционных БД стремятся не связывать обеспечиваемые ими возможности семантического моделирования с конкретной реализацией СУБД, они стимулируют откладывание строгого определения типов атрибутов до стадии полного определения реляционной схемы.

Кроме того, напомним, что при определении атрибута отношения допускается использование имен атрибутов, совпадающих с именами своих доменов (это два разных пространства имен, и наличие одинаковых имен у атрибутов и доменов не вызывает коллизий). Поэтому при определении атрибутов типов сущности **можно так подбирать их имена, что они в дальнейшем будут подсказывать, какие домены** у этих атрибутов имеются в виду. Пониманию предполагаемой сути доменов способствует и возможность указания примеров значений атрибутов.

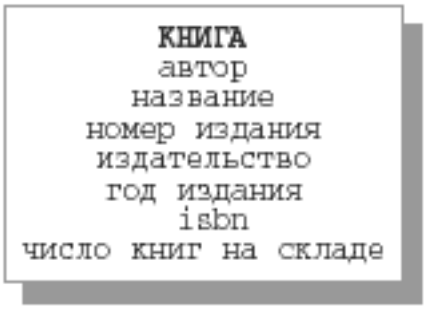
Уникальные идентификаторы типов сущности

При определении типа сущности **необходимо гарантировать, что каждый экземпляр сущности является отличимым от любого другого экземпляра той же сущности**. Поскольку сущность является абстракцией реального или представляемого объекта внешнего мира, это требование нужно иметь в виду уже при выборе кандидата в типы сущности. Например, предположим, что проектируется база данных для поддержки работы книжного склада. На складе могут храниться произвольные части тиража любого издания любой книги. Может ли в этом случае индивидуальная книга являться прообразом типа сущности? Утверждается, что нет, поскольку отсутствует возможность различения книг одного издания. Для книжного склада прообразом типа сущности будет **набор одноименных книг** одного автора, вышедших в одном издании. Одним из атрибутов этого типа сущности будет число книг в наборе. Но когда книга поступает в библиотеку и ей присваивается уникальный библиотечный номер, она становится разумным прообразом типа сущности. Плохо устроены библиотеки, в которых не различаются индивидуальные книги (даже одноименные книги одного автора, вышедшие в одном издании).

Но при проектировании базы данных мало того, чтобы проектировщик убедился в правильном выборе типов сущности, гарантирующем различие экземпляров каждого

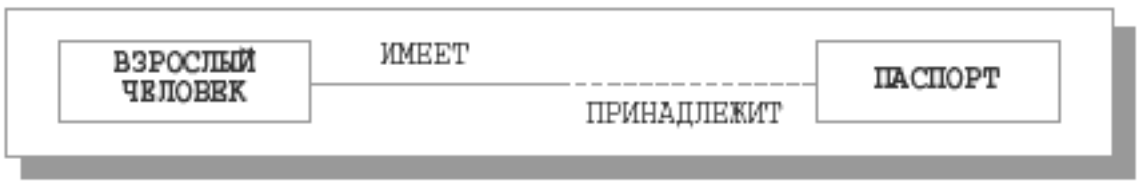
типа сущности. **Необходимо сообщить системе автоматизации проектирования БД, каким образом будут различаться эти экземпляры**, т. е. **сообщить, как конструируются уникальные идентификаторы экземпляров каждого типа сущности**. В ER-модели у экземпляра типа сущности не может быть назначаемого пользователем имени или назначаемого системой внешнего уникального идентификатора. Экземпляр типа сущности может идентифицироваться только своими индивидуальными характеристиками, а они представляются значениями атрибутов и экземплярами типов связи, связывающими данный экземпляр типа сущности с экземплярами других типов сущности или этого же типа сущности. Поэтому **уникальным идентификатором сущности может быть атрибут, комбинация атрибутов, связь, комбинация связей или комбинация связей и атрибутов**, уникально отличающая любой экземпляр сущности от других экземпляров сущности того же типа.

Приведем несколько примеров. На рисунке показан тип сущности КНИГА, пригодный для использования в базе данных книжного склада. При издании любой книги в любом издательстве (кроме пиратских, которыми мы для простоты пренебрежем) ей присваивается уникальный номер – ISBN. Понятно, что значение атрибута isbn будет уникально идентифицировать **партию книг на складе.** Кроме того, конечно, в качестве уникального идентификатора годится и комбинация атрибутов <автор, название, номер издания, издательство, год издания>.



Тип сущности, экземпляры которого идентифицируются атрибутами

На рисунке диаграмма включает два связанных типа сущности. У каждого взрослого человека имеется один и только один паспорт (мы снова не берем в расчет особый случай, когда у одного человека имеется несколько паспортов), и каждый паспорт может принадлежать только одному взрослому человеку (некоторые уже готовые паспорта могут быть еще никому не выданы). Тогда связь человека с его паспортом (конец связи ИМЕЕТ) уникально идентифицирует взрослого человека, т. е., грубо говоря, паспорт определяет взрослого человека. Поскольку могут существовать паспорта, еще не выданные какому-либо человеку, эта связь не является уникальным идентификатором сущности ПАСПОРТ.



Тип сущности, экземпляры которого идентифицируются связью

Как и в случае схем реляционных баз данных, для ER-диаграмм **вводится понятие нормальных форм**, причем их смысл очень близко соответствует смыслу нормальных форм отношений. Заметим, что определения нормальных форм ER-диаграмм делают более понятным смысл нормализации схем отношений.

Первая нормальная форма ER-диаграммы

В первой нормальной форме ER-диаграммы **устраняются атрибуты, содержащие множественные значения**, т. е. производится выявление неявных сущностей, «замаскированных» под атрибуты.

Вторая нормальная форма ER-диаграммы

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

Третья нормальная форма ER-диаграммы

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

Более сложные элементы ER-модели

До сих пор мы рассматривали только самые основные и наиболее очевидные понятия ER-модели данных. К числу некоторых более сложных элементов модели относятся следующие.

- **Подтипы и супертипы сущностей**. Подобно тому как это делается в языках программирования с развитыми типовыми системами (например, в языках объектно-ориентированного программирования), в ER-модели поддерживается возможность определения нового типа сущности путем наследования некоторого супертипа сущности.
- ***Уточняемые степени связи.*** Иногда бывает полезно определить возможное количество экземпляров сущности, участвующих в данной связи (например, ввести ограничение, связанное с тем, что служащему разрешается участвовать не более чем в трех проектах одновременно). Для выражения этого семантического ограничения разрешается указывать на конце связи ее максимально допустимую или обязательную степень.
- ***Взаимно исключающие связи.*** Для заданного типа сущности можно определить такой набор типов связи с другими типами сущности, что для каждого экземпляра заданного типа сущности может (если набор связей является необязательным) или должен (если набор связей обязателен) существовать экземпляр только одной связи из этого набора.

- ***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 специфицировать при определении связи.
- **Домены**. Как и в случае реляционной модели данных, в некоторых случаях полезна возможность определения потенциально допустимого множества значений атрибута сущности (домена).

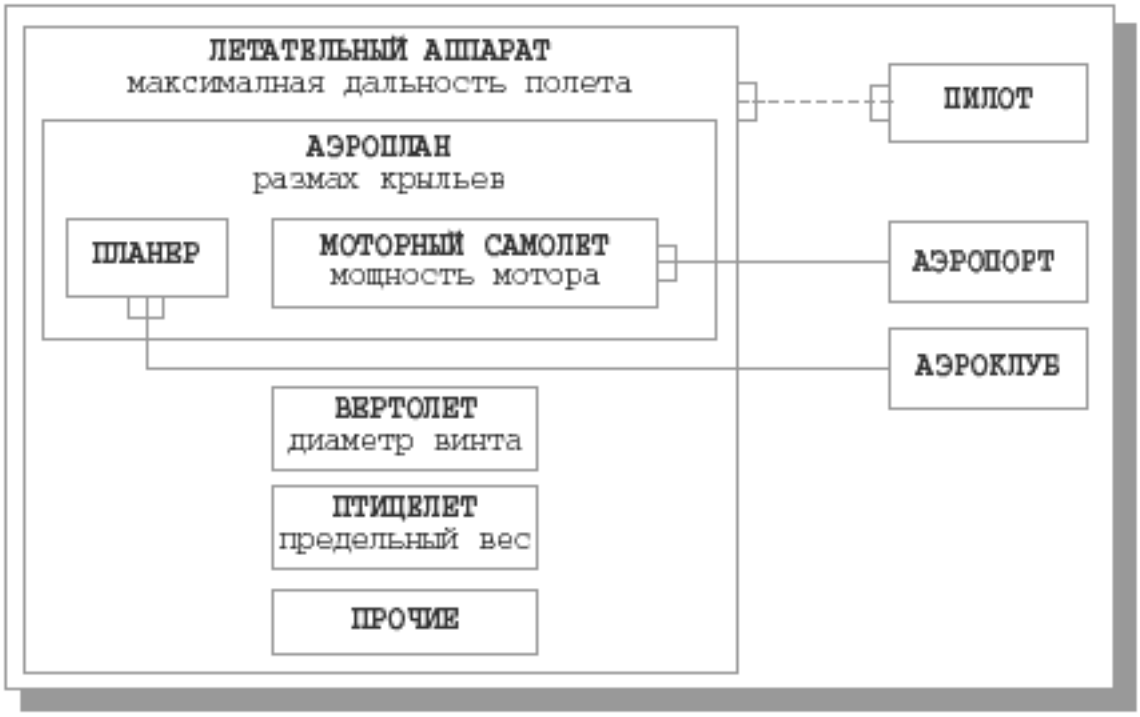
Наследование типов сущности и типов связи

Сущность может быть расщеплена на два или большее число взаимно исключающих подтипов, 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 использования ER-модели при проектировании баз данных показывает, что в большинстве случаев оказывается достаточно двух-трех уровней.

Если у типа сущности A имеются подтипы $B_1, B_2, \ldots, Bn$, то:

- (a) любой экземпляр типа сущности $B_1, B_2, \ldots, Bn$ является экземпляром типа сущности A (**включение**);
- (b) если a является экземпляром типа сущности A, то a является экземпляром некоторого подтипа сущности Bi ($i = 1, 2, \ldots, n$) (**отсутствие собственных экземпляров у супертипа сущности**);
- (c) ни для каких подтипов Bi и Bj ($i, j = 1, 2, \ldots, n$) не существует экземпляра, типом которого одновременно являются типы сущности Bi и Bj (**разъединенность подтипов**).

Тип сущности, на основе которого определяются подтипы, называется ***супертипом.*** Как мы видели выше, подтипы должны образовывать полное множество, т. е. любой экземпляр супертипа должен относиться к некоторому подтипу. Иногда для обеспечения такой полноты приходится определять дополнительный подтип ПРОЧИЕ.



Супертипы и подтипы сущности

Диаграмма изображает ЛЕТАТЕЛЬНЫЙ АППАРАТ, который должен быть АЭРОПЛАНОМ, ВЕРТОЛЕТОМ, ПТИЦЕЛЕТОМ или ДРУГИМ ЛЕТАТЕЛЬНЫМ АППАРАТОМ. Если начинать от подтипа (например, сущности ВЕРТОЛЕТ), то это ВЕРТОЛЕТ, который относится к типу ЛЕТАТЕЛЬНОГО АППАРАТА. Если начинать от подтипа, который является одновременно супертипом, то это АЭРОПЛАН, который относится к типу ЛЕТАТЕЛЬНОГО АППАРАТА и должен быть ПЛАНЕРОМ или МОТОРНЫМ САМОЛЕТОМ.

В механизме наследования ER-модели **допускается наличие двух или более разбиений сущности на подтипы.**

29. Получение реляционной схемы из ER-диаграммы.

Опишем типовую многошаговую процедуру преобразования ER-диаграммы в реляционную (более точно, в SQL-ориентированную) схему базы данных.

Каждый **простой тип сущности превращается в таблицу**. (**Простым типом** сущности называется тип сущности, не являющийся подтипом и не имеющий подтипов.) Имя сущности становится именем таблицы. **Экземплярам** типа сущности соответствуют **строки** соответствующей таблицы.

Каждый **атрибут** становится **столбцом** таблицы **с тем же именем**; может выбираться более точный формат представления данных. Столбцы, соответствующие **необязательным** атрибутам, **могут содержать неопределенные** значения; **столбцы**, соответствующие **обязательным** атрибутам, – **не могут**.

Компоненты **уникального идентификатора сущности** превращаются в **первичный ключ таблицы**. Если имеется **несколько** возможных уникальных идентификаторов, для первичного ключа **выбирается наиболее характерный**. **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, и в общем случае может привести к зацикливанию). Для именования этих столбцов используются имена концов связей и/ или имена парных типов сущностей.

Связи «**многие к одному**» (и «**один к одному**») становятся **внешними ключами,** т. е. образуется **копия** уникального идентификатора сущности **на конце связи «один»**, и соответствующие столбцы составляют внешний ключ таблицы, соответствующей типу сущности на конце связи «многие». **Необязательные связи** соответствуют столбцам внешнего ключа, **допускающим наличие неопределенных значений**; **обязательные** связи – столбцам, **не допускающим неопределенных значений**. Если между двумя типами сущности A и B имеется связь «один к одному», то соответствующий внешний ключ по желанию проектировщика может быть объявлен как в таблице A, так и в таблице B. Чтобы отразить в определении таблицы ограничение, которое заключается в том, что степень конца связи должна равняться единице, соответствующий (возможно, составной) столбец должен быть дополнительно специфицирован как возможный ключ таблицы (в случае использования языка SQL для этого служит спецификация UNIQUE).

Для поддержки связи «многие ко многим» между типами сущности A и B **создается дополнительная таблица** AB с двумя столбцами, один из которых содержит уникальные идентификаторы экземпляров сущности A, а другой – уникальные идентификаторы экземпляров сущности B. Обозначим через УИД(C) уникальный идентификатор экземпляра C некоторого типа сущности C. Тогда, если в экземпляре связи «многие ко многим» участвуют экземпляры $a_1, a_2, \ldots, a_n$ типа сущности A и экземпляры $b_1, b_2, \ldots, b_m$ типа сущности B, то в таблице AB должны присутствовать все строки вида {УИД(a_i), УИД(b_j)} для $i = 1, 2, \ldots, n$, $j = 1, 2, \ldots, m$. Понятно, что, используя таблицы A, B и AB, с помощью стандартных реляционных операций можно найти все пары экземпляров типов сущности, участвующих в данной связи.

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

Представление в реляционной схеме супертипов и подтипов сущности

В этом подразделе мы предполагаем, что реляционная схема базы данных проектируется в расчете на использование обычной SQL-ориентированной СУБД, не поддерживающей объектно-реляционные расширения.

Если в концептуальной схеме (ER-диаграмме) присутствуют подтипы, то возможны два способа их представления в реляционной схеме:

- (a) **собрать все подтипы в одной таблице;**
- (b) **для каждого подтипа образовать отдельную таблицу.**

При применении способа (a) таблица создается для максимального супертипа (типа сущности, не являющегося подтипом), а для подтипов могут создаваться представления. Таблица содержит столбцы, соответствующие каждому атрибуту (и связям) каждого подтипа. В таблицу **добавляется, по крайней мере, один столбец, содержащий код ТИПА**; **он становится частью первичного ключа**. Для каждой строки таблицы значение этого столбца определяет тип сущности, экземпляру которого соответствует строка. Столбцы этой строки, которые соответствуют атрибутам и связям, **отсутствующим в данном типе сущности**, должны содержать **неопределенные значения**.

При использовании второго метода для каждого подтипа первого уровня (непосредственного подтипа максимального супертипа) создается отдельная таблица. Для более глубоких уровней наследования применяется первый метод. Супертип воссоздается с помощью объединения проекций таблиц, соответствующих подтипам, на заголовок таблицы супертипа (т.е. из всех таблиц подтипов выбираются общие столбцы – столбцы супертипа)

У каждого способа есть свои достоинства и недостатки. К **достоинствам первого способа** (одна таблица для супертипа и всех его подтипов) можно отнести следующее:

- **соответствие логике супертипов и подтипов**; поскольку любой экземпляр любого подтипа является экземпляром супертипа, логично хранить вместе все строки, соответствующие экземплярам супертипа;
- обеспечение **простого доступа к экземплярам супертипа** и не слишком сложный доступ к экземплярам подтипов;
- возможность обойтись **небольшим числом таблиц.**

Недостатки метода (a):

- прикладная программа, имеющая дело с одной таблицей супертипа, должна включать **дополнительную логику работы с разными наборами столбцов** (в зависимости от значения столбца ТИП) и **разными ограничениями целостности** (в зависимости от особенностей связей, определенных для подтипа);
- общая для всех подтипов таблица потенциально может стать **узким местом при многопользовательском доступе по причине возможности блокировки таблицы целиком**;
- для индивидуальных столбцов подтипов должна допускаться возможность содержать неопределенные значения; таким образом, потенциально в общей таблице будет содержаться **много неопределенных значений**, что при использовании некоторых РСУБД может потребовать значительного объема внешней памяти.

Достоинства метода (b) состоят в следующем:

- действуют **более понятные правила работы с подтипами** (каждому подтипу соответствует одноименная таблица);
- **упрощается логика приложений**; каждая программа работает только с нужной таблицей.

Недостатки метода (b):

- в общем случае требуется **слишком много отдельных таблиц**;
- **работа с экземплярами супертипа** на основе представления, объединяющего таблицы супертипов, **может оказаться недостаточно эффективной**;
- поскольку множество экземпляров супертипа является объединением множеств экземпляров подтипов, **не все** РСУБД **могут обеспечить выполнение операций модификации экземпляров супертипа**.

Представление в реляционной схеме взаимно исключающих связей

Существуют два способа формирования схемы реляционной БД при наличии взаимно исключающих связей (имеются в виду связи «один ко многим», причем конец связи «многие» находится на стороне сущности, для которой связи являются взаимно исключающими):

- (a) **общее хранение внешних ключей**;
- (b) **раздельное хранение внешних ключей**.

Понятно, что если имеются взаимно исключающие связи упомянутой категории, то в таблице, соответствующей сущности, для которой связи являются взаимно исключающими, необходимо хранить внешние ключи. **Если внешние ключи всех потенциально связанных таблиц имеют общий формат**, то можно применить способ (a), т. е. создать **два столбца: идентификатор связи и идентификатор сущности** (возможно, составной). Столбец идентификатора связи используется для различения связей, покрываемых дугой исключения. В столбце (столбцах) идентификатора сущности хранятся значения уникального идентификатора сущности на дальнем конце соответствующей связи.

Если результирующие внешние ключи не относятся к одному домену, то приходится прибегать к использованию способа **(b)**, т. е. создавать для каждой связи, покрываемой дугой исключения, явные столбцы внешних ключей; **все эти столбцы могут содержать неопределенные значения**.

Преимущество подхода (a) состоит в том, что **в таблице, соответствующей сущности**, появляется **всего два дополнительных столбца**. Очевидным недостатком является **усложнение выполнения операции соединения**: чтобы воспользоваться для соединения одной из альтернативных связей, нужно сначала произвести ограничение таблицы в соответствии с нужным значением столбца, содержащего идентификаторы связей.

При использовании подхода (b) **соединения являются явными** (и естественными). Недостаток состоит в том, что требуется иметь **столько столбцов, сколько имеется альтернативных связей**. Кроме того, в каждом из таких столбцов будет содержаться **много неопределенных значений**, хранение которых потенциально может привести к серьезным накладным расходам внешней памяти.

Рис. 10.14. Возможные модификации ER-диаграмм, позволяющие избежать взаимно исключающих связей

Модификация, показанная на рис. 10.14 (b), основана на том наблюдении, что коль скоро связи являются альтернативными, то они разделяют множество экземпляров сущности `A` на два или более непересекающихся подмножества, которые могут лежать в основе определения подтипов `A1` и `A2`. Это хороший вариант, если такие подтипы могут пригодиться еще для чего-нибудь. Например, в случае взаимно исключающей связи, представленной на рис. 10.12, у исправных и неисправных самолетов могут имется несовпадающие множества атрибутов (скажем, у типа сущности ИСПРАВНЫЕ САМОЛЕТЫ может иметься атрибут `дата завершения гарантийного срока`, а у типа сущности НЕИСПРАВНЫЕ САМОЛЕТЫ – атрибут `тип неисправности`). С другой стороны, как отмечалось в предыдущем разделе, для использования этого подхода требуется возможность динамического изменения типа существующего экземпляра.

Модификация, показанная на рис. 10.14 (c), основана на том наблюдении, что коль скоро типы сущности `B` и `C` участвуют в альтернативной связи, то, по всей видимости, у этих сущностей имеется что-то общее. Возможно, их было бы правильнее определять как подтипы некоторого общего типа сущности. Заметим, что пример с рис. 10.12 явно

демонстрирует, что далеко не всегда типы сущности, участвующие в альтернативной связи, обладают общими чертами. Создание общего супертипа для типов сущности ПИЛОТ и АВИАРЕМОНТНОЕ ПРЕДПРИЯТИЕ представляется весьма странной идеей.

На этом мы заканчиваем краткую экскурсию в семантическое моделирование с использованием ER-диаграмм.

30. Диаграммы классов языка UML.

Диаграммой классов в терминологии UML называется диаграмма, на которой показан набор классов (и некоторых других сущностей, не имеющих явного отношения к проектированию БД), а также связей между этими классами. Кроме того, диаграмма классов **может включать комментарии и ограничения**. Ограничения могут неформально задаваться на естественном языке или же могут формулироваться на языке объектных ограничений OCL (Object Constraints Language).52) Чуть позже мы обсудим эту тему более подробно.

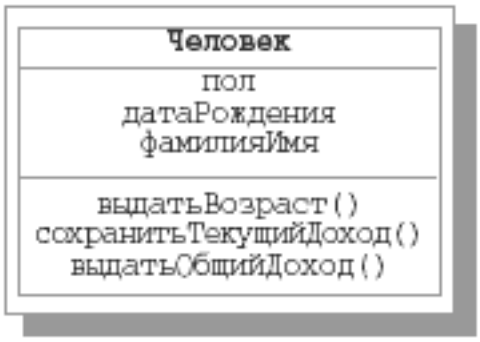
Классом называется именованное описание совокупности объектов с общими атрибутами, операциями, связями и семантикой. Графически класс изображается в виде **прямоугольника**. У **каждого класса должно быть имя** (текстовая строка), уникально **отличающее его от всех других классов**. При формировании имен классов в UML допускается использование произвольной комбинации **букв, цифр и даже знаков препинания**. Однако на практике рекомендуется использовать в качестве имен классов к**ороткие и осмысленные прилагательные и существительные, каждое из которых начинается с заглавной буквы**.

Атрибутом класса называется **именованное свойство класса, описывающее множество значений, которые могут принимать экземпляры этого свойства**. Класс может иметь **любое число атрибутов** (в частности, не иметь ни одного атрибута). Свойство, выражаемое атрибутом, является свойством моделируемой сущности, общим для всех объектов данного класса. Так что атрибут является абстракцией состояния объекта. **Любой атрибут любого объекта класса должен иметь некоторое значение**.

Имена атрибутов представляются в разделе класса, расположенном **под именем класса**. Хотя UML не накладывает ограничений на способы создания имен атрибутов (имя атрибута может быть произвольной текстовой строкой), на практике рекомендуется использовать **короткие прилагательные и существительные, отражающие смысл соответствующего свойства класса**. **Первое слово в имени атрибута рекомендуется писать с прописной буквы**, а все **остальные** слова – **с заглавной**.

Операцией класса называется **именованная услуга**, которую можно запросить у любого объекта этого класса. Операция – это абстракция того, **что можно делать с объектом**. Класс может содержать **любое число операций** (в частности, не содержать ни одной операции). Набор операций класса является общим для всех объектов данного класса.

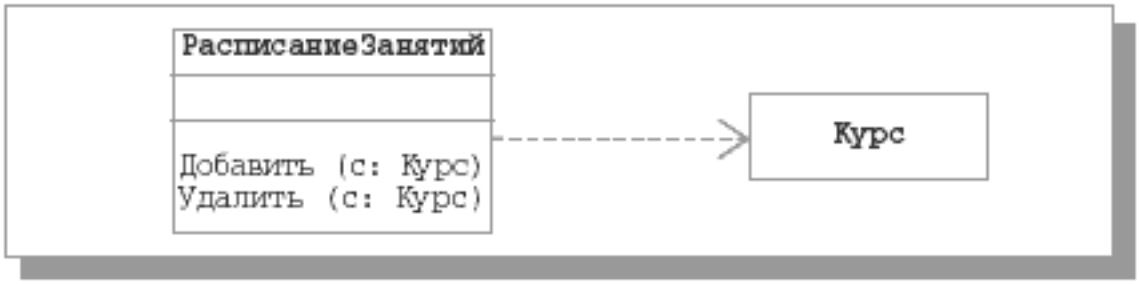
Операции класса определяются в разделе, расположенном **ниже раздела с атрибутами**. При этом можно ограничиться **только указанием имен операций**, оставив детальную спецификацию выполнения операций на более поздние этапы моделирования. Для именования операций рекомендуется использовать **глагольные формы**, соответствующие ожидаемому поведению объектов данного класса. Описание операции может также содержать ее **сигнатуру**, **т. е. имена и типы всех параметров**, а если операция является функцией, **то и тип ее значения**. Класс Человек с определенными операциями показан на рис. 11.3.



Класс Человек с операциями

В диаграмме классов могут участвовать **связи трех разных категорий**: **зависимость** (dependency), **обобщение** (generalization) и **ассоциация** (association). При проектировании реляционных БД наиболее **важны вторая и третья** категории связей, поэтому о связях-зависимостях будет сказано только самое основное.

Зависимостью называют **связь по применению**, когда изменение в спецификации одного класса может повлиять на поведение другого класса, **использующего первый класс**. **Чаще** всего зависимости **применяют в диаграммах классов**, чтобы отразить в сигнатуре операции одного класса тот факт, что параметром этой операции могут быть объекты другого класса. Понятно, что если интерфейс второго класса изменяется, это влияет на поведение объектов первого класса.



Зависимость показывается **прерывистой линией со стрелкой**, **направленной к классу, от которого имеется зависимость**. Очевидно, что связи-зависимости существенны для объектно-ориентированных систем (в том числе и для ООБД). При проектировании реляционных БД непонятно, что делать с зависимостями

Связью-обобщением называется связь между общей сущностью, называемой **суперклассом**, или родителем, и более специализированной разновидностью этой сущности, называемой **подклассом**, или потомком. Обобщения иногда называют *связями «is a»*, имея в виду, что класс-потомок является частным случаем класса-предка. Класс-потомок наследует все атрибуты и операции класса-предка, но в нем могут быть определены дополнительные атрибуты и операции.

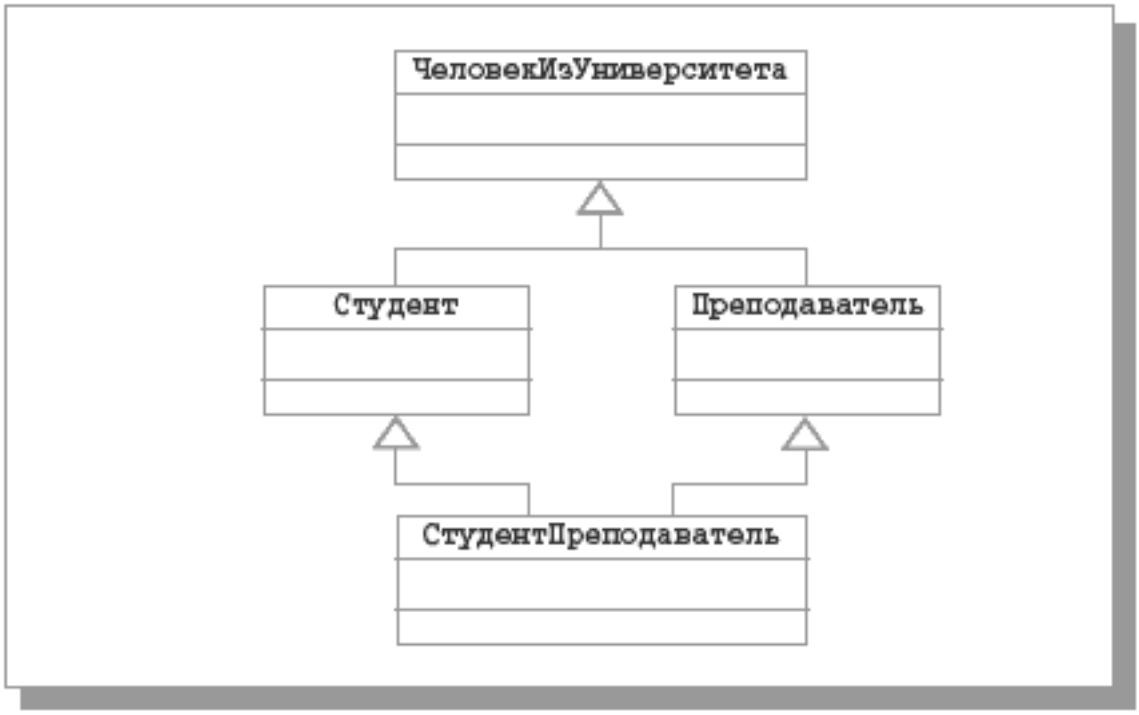
Объекты класса-потомка могут использоваться везде, где могут использоваться объекты класса-предка. Это свойство называют **полиморфизмом по включению**, имея в виду, что объекты потомка можно считать включаемыми во множество объектов класса-предка. Графически обобщения изображаются в виде **сплошной линии с большой незакрашенной стрелкой, направленной к суперклассу**.

Иерархии *одиночного наследования*: у каждого подкласса имеется только один суперкласс.

Иерархия одиночного наследования классов

Одиночное наследование является достаточным в большинстве случаев применения связи-обобщения.

Однако в UML **допускается и множественное наследование**, когда один подкласс определяется на основе нескольких суперклассов.



Пример множественного наследования классов

Следует отметить, что **множественное наследование**, помимо того что не слишком часто требуется на практике, **порождает ряд проблем**, из которых одной из наиболее известных является **проблема именования атрибутов и операций в подклассе**, полученном путем множественного наследования. Например, предположим, что при образовании подклассов Студент и Преподаватель в них обоих был определен атрибут с именем номерКомнаты. Очень вероятно, что для объектов класса Студент значениями этого атрибута будут номера комнат в студенческом общежитии, а для объектов класса Преподаватель – номера служебных кабинетов. Как быть с объектами класса СтудентПреподаватель, для которых существенны оба одноименных атрибута (у студента-преподавателя могут иметься и комната в общежитии, и служебный кабинет)? На практике применяется одно из следующих решений:

1. **запретить образование подкласса** СтудентПреподаватель, **пока в одном из суперклассов не будет произведено переименование атрибута** номерКомнаты;
2. **наследовать это свойство только от одного из суперклассов**, так что, например, значением атрибута номерКомнаты у объектов класса

СтудентПреподаватель всегда будут номера служебных кабинетов;

3. **унаследовать в подклассе оба свойства, но автоматически переименовать оба атрибута**, **чтобы прояснить их смысл**; назвать их, например, номерКомнатыСтудента и номерКомнатыПреподавателя.

Ни одно из решений не является полностью удовлетворительным. Первое решение требует возврата к ранее определенному классу, имена атрибутов и операций которого, возможно, уже используются в приложениях. Второе решение нарушает логику наследования, не давая возможности на уровне подкласса использовать все свойства суперклассов. Наконец, третье решение заставляет использовать длинные имена атрибутов и операций, которые могут стать недопустимо длинными, если процесс множественного наследования будет продолжаться от полученного подкласса.

Но, конечно, сложность проблемы именования атрибутов и операций несопоставимо меньше сложности реализации множественного наследования в реляционных БД. Поэтому при использовании UML для проектирования реляционных БД нужно очень осторожно использовать наследование классов вообще и **стараться избегать множественного наследования**.

Ассоциацией называется структурная связь, показывающая, что **объекты одного класса некоторым образом связаны с объектами другого или того же самого класса**. Допускается, чтобы оба конца ассоциации относились к одному классу. **В ассоциации могут связываться два класса, и тогда она называется бинарной**. Допускается **создание ассоциаций**, связывающих сразу *n* классов (они называются **n-арными ассоциациями**). Графически ассоциация изображается в виде л**инии, соединяющей класс сам с собой или с другими классами**.

С понятием ассоциации связаны четыре важных дополнительных понятия: **имя, роль, кратность и агрегация**. Во-первых, ассоциации может быть присвоено **имя, характеризующее природу связи**. Смысл имени уточняется с помощью **черного треугольника, который располагается над линией связи справа или слева от имени ассоциации**. Этот треугольник **указывает направление чтения имя связи**. Пример именованной ассоциации показан на рисунке. Треугольник показывает, что именованная ассоциация должна читаться как «Студент учится в Университете».

Пример именованной ассоциации

Другим способом именования ассоциации является **указание роли каждого класса**, участвующего в этой ассоциации. Роль класса, как и имя конца связи в ER-модели, **задается именем, помещаемым под линией ассоциации ближе к данному классу**. На рисунке показаны две ассоциации между классами Человек и Университет, в которых эти классы играют разные роли.

Две ассоциации с разными ролями классов

В общем случае, для ассоциации **могут задаваться и ее собственное имя, и имена ролей классов**. Это связано с тем, что класс может играть одну и ту же роль в разных ассоциациях, так что в общем случае пара имен ролей классов не идентифицирует ассоциацию. С другой стороны, в простых случаях, когда между двумя классами определяется только одна ассоциация, **можно вообще не связывать с ней дополнительные имена**.

Кратностью (multiplicity) роли ассоциации называется характеристика, указывающая, **сколько объектов класса** с данной ролью может или должно участвовать в каждом экземпляре ассоциации (в UML **экземпляр ассоциации называется *соединением – link***, но мы не будем здесь использовать этот термин, чтобы не создавать путаницу – все-таки трудно одновременно говорить про *связи*, *ассоциации* и *соединения*, имея в виду разные понятия). Наиболее распространенным способом задания кратности роли ассоциации является **указание конкретного числа или диапазона**. Например, указание «`1`» говорит о том, что **каждый объект класса с данной ролью** должен участвовать в некотором экземпляре данной ассоциации, причем в каждом экземпляре ассоциации может участвовать ровно один объект класса с данной ролью. Указание диапазона «`0..1`» говорит о том, что не все объекты класса с данной ролью обязаны участвовать в каком-либо экземпляре данной ассоциации, но в каждом экземпляре ассоциации может участвовать только один объект. Аналогично, указание диапазона «`1..*`» говорит о том, что все объекты класса с данной ролью должны участвовать в некотором экземпляре данной ассоциации, и в каждом экземпляре ассоциации должен участвовать хотя бы один объект (верхняя граница не задана). Толкование диапазона «`0..*`» является очевидным расширением случая «`0..1`».

В более сложных (но крайне редко встречающихся на практике) случаях определения кратности можно использовать списки диапазонов. Например, список «`2, 4..6, 8..*`» говорит о том, что все объекты класса с указанной ролью должны участвовать в некотором экземпляре данной ассоциации, и в каждом экземпляре ассоциации должны участвовать два, от четырех до шести или более семи объектов класса с данной ролью.

На диаграмме классов с рисунке показано, что произвольное (может быть, нулевое) число людей являются служащими произвольного числа университетов. Каждый университет обучает произвольное (может быть, нулевое) число студентов, но каждый студент может быть студентом только одного университета.

Ассоциации с указанными кратностями ролей

Обычная ассоциация между двумя классами характеризует связь между равноправными сущностями: оба класса находятся на одном концептуальном уровне. Но иногда в диаграмме классов требуется отразить тот факт, что ассоциация между двумя классами имеет специальный вид «часть-целое». В этом случае класс «целое» имеет более высокий концептуальный уровень, чем класс «часть». Ассоциация такого рода называется **агрегатной**. Графически агрегатные ассоциации изображаются в виде простой ассоциации с **незакрашенным ромбом на стороне класса-«целого».**

Пример агрегатной ассоциации

Бывают случаи, когда связь «части» и «целого» настолько сильна, что уничтожение «целого» приводит к уничтожению всех его «частей». Агрегатные ассоциации, обладающие таким свойством, называются **композитными**, или просто композициями. При наличии композиции объект-часть может быть частью только одного объекта-целого (композита). При обычной агрегатной ассоциации «часть» может одновременно принадлежать нескольким «целым». Графически композиция изображается в виде простой ассоциации, дополненной **закрашенным ромбом** со стороны «целого».

Пример композитной агрегатной ассоциации

Любой факультет является частью одного университета, и ликвидация университета приводит к ликвидации всех существующих в нем факультетов (хотя во время существования университета отдельные факультеты могут ликвидироваться и создаваться).

Заметим, что в контексте проектирования реляционных БД **агрегатные и в**

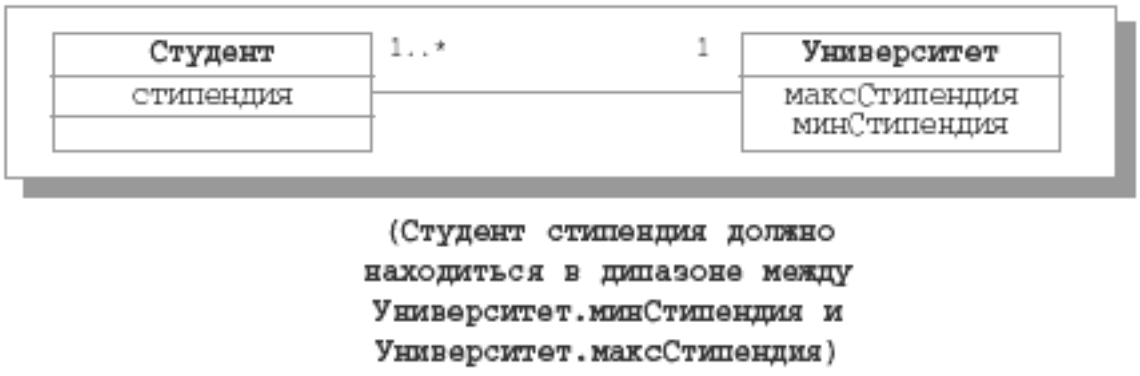
особенности композитные ассоциации влияют только на способ поддержки ссылочной целостности. В частности, композитная связь является явным указанием на то, что ссылочная целостность между «целым» и «частями» должна поддерживаться путем каскадного удаления частей при удалении целого.

При наличии простой ассоциации между двумя классами (например, ассоциации между классами Студент и Университет предполагается возможность навигации между объектами, входящими в один экземпляр ассоциации. Если известен конкретный объект-студент, то должна обеспечиваться возможность узнать соответствующий объект-университет. Если известен конкретный объект-университет, то должна обеспечиваться возможность узнать все соответствующие объекты-студенты. Другими словами, если не оговорено иное, то навигация по ассоциации может проводиться в обоих направлениях. Однако бывают случаи, когда желательно ограничить направление навигации для некоторых ассоциаций. В этом случае на линии ассоциации ставится стрелка, указывающая направление навигации.

Ассоциация с указанным направлением навигации

31. Язык объектных ограничений OCL.

Как уже отмечалось, в диаграммах классов могут указываться ограничения целостности, которые должны поддерживаться в проектируемой БД. В UML допускаются два способа определения ограничений: на естественном языке и на языке OCL.



Ограничение, выраженное на естественном языке

Более точный и лаконичный способ формулировки ограничений обеспечивает язык OCL (Object Constraints Language). Вот общая характеристика этого языка.

Из языка **UML** в OCL **заимствованы**, в первую очередь, следующие концепции:

- **класс, атрибут, операция;**
- **объект (экземпляр класса);**
- **ассоциация;**
- **тип данных** (включая набор предопределенных типов Boolean, Integer, Real и String);
- **значение** (экземпляр типа данных).

Для понимания языка OCL существенны определяемые в UML традиционные для объектных моделей данных различия между объектом некоторого класса и значением некоторого типа:

- объект обладает уникальным идентификатором и может сравниваться с другими объектами только по значению идентификатора; следствием этого является возможность определения операций над множествами объектов в терминах их идентификаторов;
- объект может быть ассоциирован через бинарную связь с другими объектами, что позволяет определить в OCL операцию перехода от данного объекта к связанным с ним объектам;
- в то же время значение является «чистым значением» в том смысле, что:
    - при сравнении двух значений проверяются сами эти значения;
    - кроме того, значения не могут участвовать в связях, поскольку понятие связи определено только для объектов классов.

В дополнение к скалярным типам данных, заимствованным из UML, в OCL предопределены структурные типы, которые являются разновидностями типов коллекций (collection):

- математическое множество (set), неупорядоченная коллекция, не содержащая одинаковых элементов;
- мультимножество (bag), неупорядоченная коллекция, которая может содержать повторяющиеся элементы-дубликаты;
- последовательность (sequence), упорядоченная коллекция, которая может содержать элементы-дубликаты.

В OCL **элементами каждого из трех типов коллекций могут быть либо объекты, либо значения**.

Язык OCL предназначен, главным образом, **для определения ограничений целостности данных**, соответствующих модели, которая представлена в терминах диаграммы классов **UML**. OCL может применяться для определения ограничений, описывающих пред- и постусловия операций классов, и ограничений, представляющих собой инварианты классов. При проектировании реляционных баз данных возможность определения пред- и постусловий операций вряд ли может оказаться существенной.

Под **инвариантом класса** в OCL понимается **условие, которому должны удовлетворять все объекты данного класса**. Если говорить более точно, инвариант класса – это **логическое выражение**, вычисление которого должно давать **`true`** при **создании любого объекта данного класса и сохранять истинное значение в течение всего времени существования этого объекта**. При определении инварианта требуется указать имя класса и выражение, определяющее инвариант указанного класса.

Синтаксически это выглядит следующим образом:

```
context <class_name> inv:
<OCL-выражение>
```

Здесь `<class-name>` является именем класса, для которого определяется инвариант, `inv` – ключевое слово, говорящее о том, что определяется именно инвариант, а не ограничение другого вида, и `context` – ключевое слово, которое говорит о том, что контекстом следующего после двоеточия OCL-выражения являются объекты класса `<class-name>`, т. е. OCL-выражение должно принимать значение `true` для всех объектов этого класса.

Заметим, что OCL является **типизированным языком,** поэтому у каждого выражения имеется некоторый тип. Естественно, что **OCL-выражение** в инварианте класса должно быть логического типа.

В общем случае OCL-выражение в определении инварианта основывается на композиции операций, которым посвящена большая часть определения языка. В спецификации языка эти операции условно разделены на следующие группы:

- операции над значениями предопределенных в UML (**скалярных)** типов данных;
- операции над **объектами**;
- операции над **множествами**;
- операции над **мультимножествами**;
- операции над **последовательностями.**

В OCL поддерживаются следующие заимствованные из определения UML скалярные типы данных: Boolean, Integer, Real и String.

В OCL определены **три операции над объектами**:

- **получение значения атрибута**;
- **переход по соединению**,
- **вызов операции класса** (последняя операция для целей проектирования реляционных БД несущественна).

Для записи этих трех операций используется **«точечная нотация»**. Например, результатом выражения вида

```
<объект>.<имя атрибута>
```

является текущее значение атрибута с именем `имя атрибута`, если объект имеет такой атрибут. В противном случае использование подобного выражения приводит к возникновению **ошибки типа**.

Результатом применения к объекту операции **перехода по соединению** (экземпляру связи-ассоциации) является **коллекция**, **содержащая все объекты**, которые

ассоциированы с данным объектом через указываемое соединение. Это соединение идентифицируется именем роли, противоположной по отношению к данному объекту. Таким образом, синтаксис выражения перехода по соединению следующий:

```
<объект>.<имя роли, противоположенной по отношению к объекту>
```

Синтаксически **операции над коллекциями записываются в нотации, аналогичной точечной**, но вместо точки используется стрелка (⟶). Таким образом, общий синтаксис применения операции к коллекции следующий:

```
<коллекция> ⟶ <имя операции> (<список фактических параметров>)
```

В OCL определены **три одноименных операции `select`**, которые обрабатывают заданное **множество**, **мультимножество** или **последовательность на основе заданного логического выражения над элементами коллекции**. Результатом каждой операции является **новое множество**, **мультимножество** или **последовательность**, соответственно, из тех элементов входной коллекции, для которых результатом вычисления логического выражения является `true`.

Аналогично набору операций `select`, в OCL определены **три операции `collect`**, параметрами которых являются **множество, мультимножество или последовательность** и некоторое выражение над элементами соответствующей коллекции. **Результатом** является **мультимножество** для операций `collect`, определенных над **множествами** и **мультимножествами**, и **последовательность** для операции `collect`, определенной над последовательностью. При этом результирующая коллекция соответствующего типа (коллекция значений или объектов) состоит из результатов применения выражения к каждому элементу входной коллекции. Операция `collect` **используется**, главным образом, в тех случаях, когда **от заданной коллекции** объектов **требуется перейти** к некоторой другой коллекции объектов, которые ассоциированы с объектами исходной коллекции через некоторое соединение. В этом случае выражение над элементом исходной коллекции **основывается на операции перехода по соединению.**

В OCL определены **три одноименных операции `exists`** над множеством, мультимножеством и последовательностью, дополнительным параметром которых является логическое выражение. В результате каждой из этих операций **выдается `true`** в том и только в том случае, когда хотя бы для одного элемента входной коллекции значением логического выражения является `true`. В противном случае результатом операции является `false`. **Операции `forAll`** отличаются от операций `exists` тем, что в результате каждой из них выдается `true` в том и только в том случае, когда для всех элементов входной коллекции результатом вычисления логического выражения является `true`. В противном случае результатом операции будет `false`. Операция **`size`** применяется к коллекции и выдает число содержащихся

в ней элементов.

Параметрами **двуместных операций `union`, `intersect`, `symmetricDifference`** являются две коллекции, причем в OCL операции определены почти для всех возможных комбинаций типов коллекции. Не будем рассматривать все определения этих операций и кратко упомянем только две из них. Результатом операции `union`, определенной над множеством и мультимножеством, является мультимножество, т. е. из результата объединения таких двух коллекций дубликаты не исключаются. Результатом же операции `union`, определенной над двумя множествами, является множество, т. е. в этом случае возможные дубликаты должны быть исключены.

11.3.4. Примеры инвариантов

В заключение обзора языка OCL приведем примеры четырех инвариантов, выраженных на этом языке.

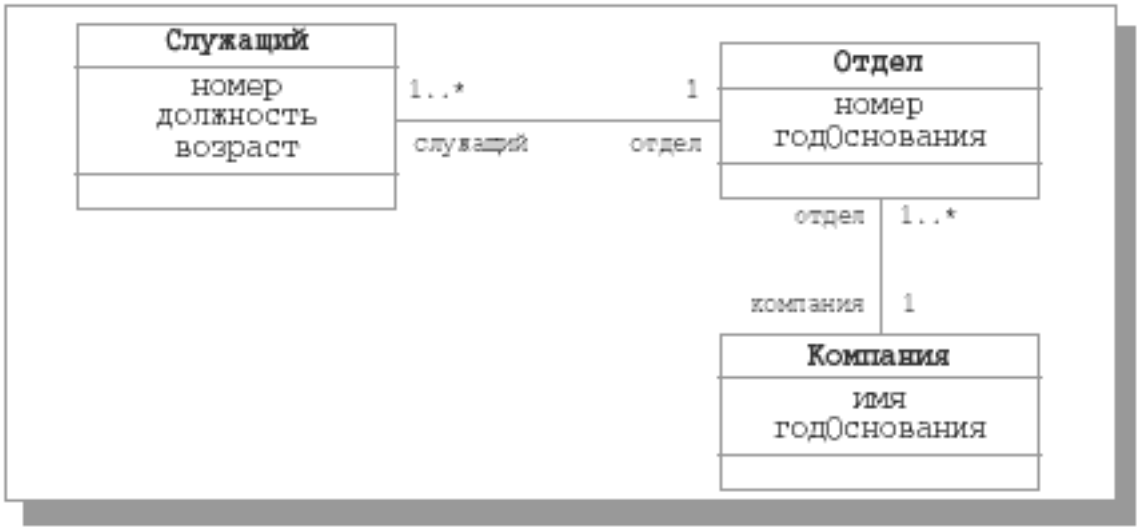


Диаграмма классов, используемая для примеров на языке OCL

Определить ограничение «возраст служащих должен быть больше 18 и меньше 100 лет».

```
context Служащий inv:
self.возраст > 18 and self.возраст < 100
```

Условие инварианта накладывает требуемое ограничение на значения атрибута `возраст`, определенного в классе `Служащий`. В условном выражении инварианта ключевое **слово** `self` обозначает **текущий объект класса-контекста** инварианта.

Выразить на языке OCL ограничение, в соответствии с которым в отделах с номерами больше 5 должны работать служащие старше 30 лет.

```
context Отдел inv:
self.номер ≤ 5 or
```

```
self.служащий → select (возраст ≤ 30) → size () = 0
```
В этом случае условное выражение инварианта будет вычисляться для каждого объекта класса Отдел

Тот же инвариант можно сформулировать в контексте класса Сотрудник:

```
context Сотрудник inv:
self.возраст > 30 or self.отдел.номер ≤ 5
```

Определить ограничение, в соответствии с которым у каждого отдела должен быть менеджер, и любой отдел должен быть основан не раньше соответствующей компании:

```
context Отдел inv:
self.служащий → exists (должность = "manager") and
self.компания.годОснования ≤ self.годОснования
```

Условие четвертого инварианта ограничивает максимально возможное количество служащих компании числом 1000:

```
context Компания inv:
self.отдел → collect (служащие) → size ( ) < 1000
```

Плюсы и минусы использования языка OCL при проектировании реляционных БД очевидны. Язык **позволяет формально и однозначно** (без двусмысленностей, свойственных естественным языкам) **определять ограничения целостности БД** в терминах ее концептуальной схемы. Скорее всего, наличие подобной проектной документации будет полезным для сопровождения БД, даже если придется преобразовывать инварианты OCL в ограничения целостности SQL вручную.

К отрицательным сторонам использования OCL относится, прежде всего, **сложность языка и неочевидность некоторых его конструкций**. Кроме того, строгость синтаксиса и линейная форма языка в некотором роде **противоречат наглядности и интуитивной ясности диаграммной части UML**. Да, в инвариантах OCL используются те же понятия и имена, что и в соответствующей диаграмме классов, но используются совсем в другой манере. И последнее. **Трудно доказать или опровергнуть как предположение, что на языке OCL можно выразить любое ограничение целостности, которое можно определить средствами SQL**, так и утверждение, что на языке OCL нельзя выразить такой инвариант, для которого окажется невозможным сформулировать эквивалентное ограничение целостности на языке SQL. Неизвестны работы, в которых бы сравнивалась выразительная мощность этих языков в связи с ограничениями целостности реляционных БД.

Если не обращать внимания на различия в терминологии, то выполняются практически **те же шаги, что и в случае преобразования в схему реляционной БД ER-диаграммы**. Поэтому ограничимся только некоторыми рекомендациями, специфичными для диаграмм классов.

Рекомендация 1. Прежде чем определять в классах операции, **подумайте, что вы будете делать с этими определениями в среде целевой РСУБД**. Если в этой среде поддерживаются хранимые процедуры, то, возможно, некоторые операции могут быть реализованы именно с помощью такого механизма. Но если в среде РСУБД поддерживается механизм определяемых пользователями функций, возможно, он окажется более подходящим.

Рекомендация 2. Помните, что сравнительно эффективно в РСУБД реализуются только ассоциации видов **«один ко многим» и «многие ко многим».** Если в созданной диаграмме классов имеются ассоциации «один к одному», следует задуматься о целесообразности такого проектного решения. Реализация в среде РСУБД ассоциаций с точно заданными кратностями ролей возможна, но требует определения дополнительных триггеров, выполнение которых понизит эффективность.

Рекомендация 3. Для технологии реляционных БД **агрегатные и в особенности композитные ассоциации неестественны**. Подумайте о том, что вы хотите получить в реляционной БД, объявив некоторую ассоциацию агрегатной. Скорее всего, ничего.

Рекомендация 4. В спецификации UML говорится о том, что, определяя **однонаправленные связи**, вы можете способствовать эффективности доступа к некоторым объектам. Для технологии реляционных баз данных поддержка такого объявления **вызовет дополнительные накладные расходы** и тем самым снизит эффективность.

Рекомендация 5. **Не злоупотребляйте возможностями OCL**.

Диаграммы классов UML – это мощный инструмент для создания концептуальных схем баз данных, но, как известно, все хорошо в меру.

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

Система управления реляционными базами данных System R.

В тех случаях, когда терминология System R расходится с реляционной терминологией, предпочтение будет отдаваться терминологии System R. В частности, это касается использования термина «**поле таблицы**» вместо термина «**атрибут отношения**».

Базовым понятием System R является понятие **таблицы** (приближенный к реализации аналог основного понятия реляционного подхода **отношения**; иногда, в зависимости от контекста, мы будем использовать и этот термин). **Таблица** – это регулярная структура данных, состоящая из конечного набора однотипных записей – кортежей. Каждый

кортеж одной таблицы состоит из конечного (и одинакового) числа полей кортежа, причем *i*-тое поле каждого кортежа одной таблицы может содержать данные только одного типа, и **набор допустимых типов данных в System R предопределен и фиксирован.**

В силу **регулярности структуры таблицы понятие поля кортежа расширяется до понятия поля таблицы**. Тогда *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 таблицы 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.

При выполнении проекта System R преследовались следующие основные цели:

- 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 таблицы, 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, допуская параллельную модификацию 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**.

Язык

Основой System R является **язык SEQUEL** (который достаточно быстро был переименован в SQL). 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.

Целостность

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 ***утверждениями целостности*** **(assertion)**, также задаются средствами языка SQL. Любой оператор языка выполняется в границах некоторой **транзакции** – последовательности операторов языка, неделимой в смысле состояния базы данных. Неделимость означает, что все изменения базы данных, 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 . Язык SQL System R 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 транзакции, инициированный по причине нарушения условия целостности, производится **до ближайшей точки сохранения, в которой условия целостности соблюдены**.

Журналирование

В System R используется специальный набор данных – **журнал**, в который помещаются записи обо всех операциях всех транзакций, изменяющих состояние базы данных. При откате транзакции происходит процесс **обратного выполнения транзакции** (undo), в ходе которого в обратном порядке выполняются все изменения, запомненные в журнале.

Ограничения доступа

В языке SQL System R имеется средство определения так называемых **триггеров** (trigger), позволяющих автоматически поддерживать целостность базы данных при модификациях ее объектов. В SQL System R **триггер** – это каталогизированная операция модификации, для которой задано условие ее автоматического выполнения.

Язык SQL содержит средства определения **представлений**. **Представление** – это каталогизированный именованный запрос на выборку данных (из одной или нескольких таблиц). 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

Авторизация доступа к базе данных также **основана на средствах SQL**. При создании любого объекта базы данных пользователь, выполняющий эту операцию, становится полновластным владельцем этого объекта, т.е. может выполнять по отношению к этому объекту любую допустимую операцию SQL. 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.

Параллельная работа

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 пользователей**, конкурирующих с ним за доступ к базе данных, 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 блокировок всех изменяемых объектов базы данных**.

Надежность

Одним из основных требований к СУБД вообще и к System R в частности является обеспечение надежности баз данных по отношению к различного рода сбоям. 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 уже отмечалось, такое восстановление происходит путем обратного выполнения транзакции на основе информации о внесенных ею изменениях, запомненной в журнале. **На информации журнала также основано восстановление базы данных и после сбоев другого рода**.

Эффективность

Что касается естественных требований к эффективности системы, то здесь основные решения связаны со **спецификой физической организации баз данных во внешней**

памяти, использованием техники **индексированного доступа к данным**, **буферизацией используемых страниц базы данных в основной памяти и развитой техникой оптимизации SQL-запросов**, производимой на стадии их компиляции.

Структурная организация System R согласуется с поставленными при ее разработке целями и выбранными решениями. Основными структурными компонентами System R являются система управления реляционными данными **(Relational Data System – RDS**), состоящая, по существу, из **компилятора языка SQL** и подсистемы поддержки откомпилированных операторов, и система управления реляционной памятью (**Relational Storage System** – RSS).

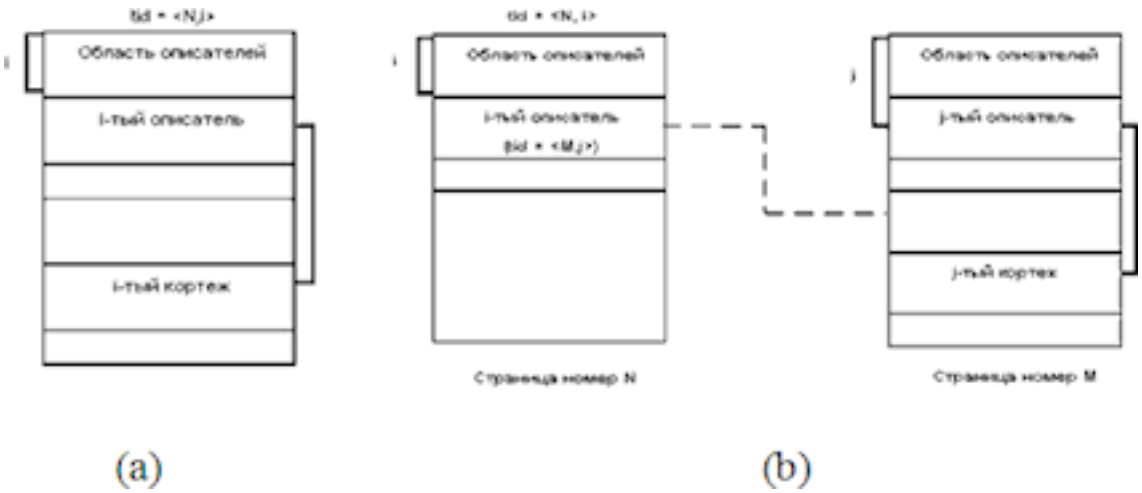
Компилятор запросов использует интерфейс RSS для доступа к разнообразной справочной информации (каталоги таблиц, индексов, прав доступа, условий целостности, условных воздействий и т.д.) и производит рабочие программы, выполняемые в дальнейшем также с использованием интерфейса RSS.

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.

33. Организация памяти в базах данных System R, B - деревья.

Как уже говорилось, 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 килобайтам**; 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 одной или нескольких таблиц**. Фундаментальным понятием RSS является **идентификатор кортежа** *(tuple identifier – tid)*. Гарантируется **неизменяемость** tid'a во все время существования кортежа в базе данных независимо от перемещений кортежа внутри страницы и даже при перемещении кортежа в другую страницу. **Потребность в перемещении** кортежей возникает по той причине, что **кортеж**, занесенный в некоторую таблицу базы данных, вообще говоря, во время своего существования **может увеличиваться в размерах** (если к этой таблице добавляется новое поле, или если в ней имеется хотя бы одно поле, типом данных которого являются строки символов переменного размера). Реально **tid представляет собой пару** <номер страницы, индекс описателя кортежа в странице>. При этом кортеж может реально располагаться в данной странице (рис. 12.1a) или в другой странице (рис. 12.1b).



Идентификатор кортежа и расположение кортежа в странице данных

Как показывает рисунок, в каждой странице данных имеются **две области: область хранения описателей кортежей** и **область хранения самих кортежей**. Один из остроумных приемов, примененных в System R, состоит в том, что **обе эти области являются динамическими**, т.е. **в странице данных заранее не резервируется место под описатели кортежей**. Легко видеть, что **выделение фиксированной части страницы данных под описатели кортежей** (вмещающей, скажем, *k* описателей) потенциально привело бы к потери памяти в этой странице, поскольку **при размещении в ней *k* кортежей очень маленького размера пропадало бы место в области хранения кортежей, а при размещении *p<k* крупных кортежей полностью заполнялась бы область хранения кортежей**, но пропадало бы место в области описателей. Для динамического распределения памяти внутри страницы **память на описатели кортежей выделяется вниз** от начала страницы, **а память для хранения кортежей – вверх от конца страницы**.

Второй вариант хранения кортежей возникает в том случае, когда **некоторый кортеж после своего создания был размещен системой в странице с номером N,** а после обновления с увеличением размера **перестал помещаться в этой странице**, и **система была вынуждена разместить его в странице с номером M**. Тогда **исходный tid этого кортежа не изменится**, но **его описатель в странице N будет содержать не координаты кортежа в данной странице, а новый tid**, указывающий на реальное положение кортежа в странице M. Легко видеть, что применение такого подхода позволяет ограничиться максимум **одним уровнем косвенности** (если данный кортеж в какой-то момент времени перестанет помещаться в странице M, и система переместит его в страницу P, то достаточно будет изменить косвенную ссылку на этот кортеж в странице N, и его исходный tid не изменится).

Поскольку допускается нахождение в одной странице данных кортежей разных таблиц, каждый **кортеж должен**, **кроме содержательной части**, **включать служебную информацию**, **идентифицирующую таблицу, которой принадлежит данный кортеж**. Кроме того, в System R (точнее, в языке SQL) **допускается динамическое добавление полей к существующим таблицам**. При этом реально происходит лишь **модификация описателя таблицы в таблице-каталоге таблиц**. В существующем кортеже таблицы **новое поле возникает только при модификации этого кортежа, затрагивающей новое поле**. Это позволяет избежать массовой перестройки хранимой

таблицы при добавлении к ней новых полей, но, естественно, требует хранения при кортеже дополнительной служебной информации, определяющей реальное число полей в данном кортеже. (Заметим, что удалять существующие поля существующей таблицы в SQL System R не разрешалось.)

На основе наличия уникальных, обеспечивающих почти прямой доступ к кортежам и не изменяемых во время существования кортежей **tid'ов** в System R **поддерживаются** дополнительные управляющие структуры – **индексы**. Каждый **индекс определяется на одном или нескольких полях таблицы**, значения которых составляют его ключ, и позволяет производить **прямой поиск по ключу кортежей** (их tid'ов) и последовательное сканирование таблицы 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 System R указания системе первичного ключа таблицы (фактически, набора первичного и всех возможных ключей таблицы).

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 **из-за своей *сильной ветвистости* обладают небольшой глубиной**. Кроме того, **B+-деревья сохраняют *порядок ключей* в листовых блоках иерархии**, что позволяет производить последовательное сканирование таблицы 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**. **System R была первой системой**, в которой для организации индексов использовались B+-деревья.

Видимо, **наиболее важной особенностью физической организации баз данных в System R** является **возможность обеспечения *кластеризации* связанных кортежей одной или нескольких таблиц**. **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 некоторого класса запросов**.

В окончательном варианте System R **существует только одно средство определения условий кластеризации таблицы – объявить до заполнения таблицы один** (и **только один**) **индекс, определенный на полях этой таблицы, кластеризованным**. Тогда, **если заполнение таблицы 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 кластеризованной таблицы **оставляется некоторое резервное свободное пространство**. При последующих вставках кортежей в такую таблицу система стремится поместить каждый кортеж в одну из страниц данных, в которых уже находятся кортежи этой таблицы с такими же (или близкими)

значениями полей кластеризации. Естественно, что **поддерживать идеальную кластеризацию таблицы можно только до определенного предела**, пока не исчерпается резервная память в страницах. Далее этого предела степень кластеризации таблицы начинает уменьшаться, и для **восстановления идеальной кластеризации таблицы требуется физическая реорганизация таблицы** (ее можно произвести средствами SQL).

Очевидным **преимуществом кластеризации** таблицы является то, что при последовательном сканировании кластеризованной таблицы с использованием кластеризованного индекса **потребуется ровно столько чтений страниц данных из внешней памяти, сколько страниц занимают кортежи этой таблицы**. Следовательно, **при правильно выбранных критериях кластеризации 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 таблицы** 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 одной таблицы**). В языке SEQUEL (до того момента, когда его стали называть SQL) **существовали средства определения связей в иерархической** манере: можно было объявить некоторую таблицу родительской по отношению к той же или другой таблице-потомку. При этом указывались поля родительской таблицы и таблицы-потомка, в соответствии со значениями которых образовывалась иерархия. **Правила** построения были очень простыми – **проводились связи от кортежа родительской таблицы ко всем кортежам таблицы-потомк**а с теми же значениями полей связывания. На самом деле, **все кортежи таблицы-потомка с общим значением полей связывания образовывали кольцевой списо**к, **на который проводилась одна связь из соответствующего кортежа родительской таблицы**.

Следует заметить, что этот способ использования механизма связей поддерживался в ранних версиях SEQUEL. **В интерфейсе RSS System R** этого периода д**опускалась возможность произвольной установки 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 одной таблицы допускалось создание многих связей: кортеж таблицы м**ог 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 таблиц. **Основной смысл** такой кластеризации заключался **в возможности оптимизации выполнения некоторых запросов**, включающих (экви)соединение двух связанных таблиц в соответствии со значениями полей связывания.

В более поздних публикациях, посвященных System R, упоминания о **механизме связей** исчезли, из чего можно заключить, что **разработчики отказались от его**

использования.

Основными причинами отказа от использования связей были следующие. Во-первых, **средства построения** связей, обеспечиваемые RSS, **были очень низкого уровня**, гораздо более низкого, чем средства поддержки индексов. Если при занесении, удалении или обновлении кортежа RSS обеспечивала автоматическую коррекцию всех индексов, то для коррекции связей требовалось выполнить ряд дополнительных обращений к RSS, из-за чего **время выполнения этих операций, конечно, увеличивалось.**

Во-вторых, при реализации этого механизма возникают д**ополнительные 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 преимуществами, предоставляемыми механизмом связей**. Действительно, максимального эффекта от использования связей можно достичь только при выполнении операции соединения двух таблиц, кластеризованных по этой связи, если поле соединения совпадает с полем связывания и условия, накладываемые на родительскую таблицу, выделяют в нем ровно один кортеж. Очевидно, что такие запросы на практике редки.

Кроме таблиц и индексов при работе System R во внешней памяти могут располагаться еще и временные объекты – ***списки*** *(list)*. **Список** – это временная структура данных, создаваемая с целью оптимизации выполнения SQL-запроса, **содержащая некоторые кортежи хранимой таблицы базы данных**, не имеющая имени и, следовательно, **не видимая на уровне интерфейса SQL**. Кортежи списка **могут быть упорядочены по возрастанию или убыванию полей** соответствующей таблицы. 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 и**сходя из здравого смысла можно предположить, что они устроены не так, как таблицы** (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).

Хэширование

Альтернативным и достаточно популярным подходом к организации индексов является использование техники хэширования. Общей идеей является применение к значению ключа некоторой функции свертки *(*хэш-функции*)*, вырабатывающей значение меньшего размера. **Значение хэш-функции 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 (одним из распространенных видов

«хороших» хэш-функций являются функции, выдающие **остаток от деления значения ключа на некоторое простое число**). **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).

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 на 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). В B+-дереве внутренние и листовые страницы обычно имеют разную структуру.

Структура внутренней страницы: • ключ1 <= ключ2 <=… ключm;

• в странице дерева Nm находятся ключи k со значениями ключm <= k <= ключm+1.

Структура листовой страницы:

• ключ1 < ключ2 < ... < ключk;

•списокr – упорядоченный список идентификаторов кортежей (tid), включающих значение ключr;

• 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 B+-деревья являются сильно ветвистыми и сбалансированными, 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, е**сли во внутренней странице помещается** `n` **ключей**, то **при хранении** `m` **записей требуется дерево глубиной** $\log_n(m)$. Если `n` достаточно велико (обычный случай), то глубина дерева невелика, и производится быстрый поиск. **Основной «изюминкой» B+-деревьев является автоматическое поддержание свойства сбалансированности**.

При занесение новой записи выполняются следующие действия. 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 основной 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 странице, 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+-деревaВ 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

34. Интерфейс ядра System R - RSS.

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 во внешней памяти для постоянных объекто**в или **в основной памяти для временных объектов**. **Замена** имен объектов базы данных на их идентификаторы **производится компилятором SQL** на основе информации, черпаемой им из системных таблиц-каталогов.

Можно выделить следующие группы операций:

- операции **сканирования** таблиц и списков;
- операции **создания и уничтожения постоянных и временных объектов** базы данных;
- операции **модификации** таблиц и списков;
- операция **добавления поля** к таблице;
- операции **управления** прохождением **транзакций**;
- операция **явной синхронизации**.

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

Для таблицы возможны **два режима** сканирования: **прямое** сканирование и сканирование **через индекс**. При **прямом** сканировании единственным параметром операции `OPEN` является **идентификатор таблицы** (включающий и идентификатор сегмента, в котором эта таблица хранится). По причине того, что в System R допускается размещение в одной странице данных кортежей нескольких таблиц, прямое сканирование предполагает **последовательный просмотр** всех **страниц сегмента** с выделением в них кортежей, входящих в данную таблицу; это очень **дорогой способ сканирования таблицы**. При этом **порядок выборки кортежей определяется их физическим размещением в страницах сегмента**, т.е. предопределен системой.

При начале сканирования таблицы через индекс **в число параметров операции** `OPEN` **входит идентификатор какого-либо индекса**, определенного ранее на полях этой таблицы. Кроме того, **можно указать диапазон сканирования в терминах значений поля** (полей), **составляющего ключ индекса**. При открытии сканирования через индекс **производится начальная установка указателя сканирования в позицию листа B-дерева индекс**а, соответствующую левой границе заданного диапазона. Процесс сканирования состоит в **последовательном продвижении по листовым вершинам индекса** до достижения правой границы диапазона сканирования с выборкой идентификаторов кортежей и чтением соответствующих кортежей. Легко видеть, что в **худшем случае может потребоваться столько чтений страниц данных из внешней памяти, сколько идентификаторов кортежей 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 прямом сканировании таблицы, единственным параметром операции `OPEN` является **идентификатор списка**, но, в отличие от прямого сканирования таблицы **это сканирование максимально эффективно**: читаются **только страницы, содержащие 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 аргумента других операций этой группы**.

Операция **`NEXT` выполняет чтение следующего кортежа указанного сканирования, удовлетворяющего условию данной операции. Условие** представляет собой **дизъюнктивную нормальную форму** простых условий, **накладываемых на значения указанных полей таблицы**. **Простое условие** – это условие вида `номер-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 таблицы 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 кортежа достигается правая граница диапазона сканирования (правая граница значения ключа при сканировании через индекс или последний кортеж таблицы или списка при прямом сканировании), то 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 **спецификатор структуры объекта**, т**.е. число полей объекта и спецификаторы их типов**. Кроме того, при спецификации полей таблицы указывается разрешение или запрещение наличия неопределенных значений полей в

кортежах этой таблицы 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 совпадение со специальной литеральной константой NULL**.

В результате выполнения этих операций **заводится описатель** в служебной таблице описателей таблиц или основной памяти (в зависимости от того, создается ли постоянный объект или временный), и **вырабатывается идентификатор объекта, который служит входным параметром других операций, относящихся к соответствующему объекту** (в частности, параметром операции `OPEN` при открытии сканирования 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 полей, составляющих ключ**. 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 служебной таблице описателей индексов и возвращается идентификатор индекса (который, в частности, используется в качестве аргумента операции открытия сканирования таблицы через индекс).

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 таблицы. 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 непустой таблицы запрещено** , поскольку соответствующую кластеризацию таблицы без ее 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` и `DROP TABLE`) требуют дополнительных накладных расходов в связи с **необходимостью обеспечения возможности откатов транзакции,** для чего требуется выполнение массовых обратных действий. Особенно сильно это затрагивает операцию уничтожения непустых таблиц, поскольку требует **журнализации всех кортежей, содержащихся в них к моменту уничтожения**. Поэтому, хотя **уничтожение непустых таблиц и не запрещено**, нужно иметь в виду, что это очень дорогостоящая

операция.

Группа операций **модификации таблиц и списков** включает операции **вставки** кортежа в таблицу или список (`INSERT`), **удаления кортежа из таблицы** (`DELETE`) и **обновления кортежа в таблице** (`UPDATE`).

Параметрами операции вставки кортежа являются **идентификатор таблицы 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 кластеризованную таблицу 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 некластеризованную таблицу место под кортеж выделяется **в первой подходящей странице данных**. Наконец, при вставке кортежа в **список он помещается в конец списка**.

При занесении кортежа в таблицу **производится коррекция всех индексов**, определенных на этой таблице. 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 таблицу вырабатывается идентификатор нового кортежа**, который выдается в качестве результата операции и может быть в дальнейшем использован как прямой параметр операций удаления и модификации кортежей таблицы. При **занесении кортежа в список значение идентификатора кортежа не вырабатывается** (для списков допускается только последовательное сканирование и добавление новых кортежей в конец списка; над ними нельзя определить индексов, и поэтому косвенная адресация кортежей списков через их идентификаторы не требуется).

Операции удаления и модификации кортежей допускаются только для кортежей таблиц . Естественно , что для выполнения этих операций необходимо **идентифицировать соответствующий кортеж**. В интерфейсе RSS допускаются два способа такой идентификации: **с помощью идентификатора кортежа** (явная адресация) и **с использованием идентификатора открытого к этому времени сканирования**. Первый вариант возможен, поскольку идентификатор 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 выполняется операция `DELETE` или операция `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

сканирования теряется, и его идентификатор нельзя использовать для идентификации кортежа, пока не будет выполнена следующая операция NEXT.

Единственным параметром операции DELETE является **идентификатор кортежа или идентификатор сканирования**. Параметры операции UPDATE включают, кроме этого, **спецификацию изменяемых полей кортежа** (список номеров полей и их новых значений) . Среди значений могут находиться литеральные изображения неопределенных значений, если соответствующие поля таблицы допускают хранение неопределенных значений. При выполнении операции DELETE производится коррекция всех индексов, определенных на данной таблице. 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»** BUILDLIST, позволяющую за одно обращение к RSS построить список, отсортированный в соответствии со значениями заданных полей. Эта операция включает сканирование заданной таблицы 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 в новом списке допускаются или не допускаются кортежи-дубликаты**.

Операция RSS **добавления поля к существующей таблице 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 данной таблицы** в служебной таблице описателей таблиц. 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).

Это не отмечалось раньше, но на самом деле при вызове любой операции 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 позиций сканирования. Откат **может быть произведен до начала транзакции** (в этом случае о восстановлении позиций сканирования говорить бессмысленно) или до установленной ранее в транзакции точки сохранения.

Точка сохранения устанавливается с помощью операции `SAVE`. При выполнении этой операции **запоминаются состояние сканов данной транзакции**, открытых к моменту выполнения `SAVE`, и **координаты последней записи об изменениях в базе данных в журнал**е, 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 **аргумент 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 установки указанной точки сохранения. Кроме того, **по локальной информации в оперативной памяти , привязанной к транзакции , восстанавливается состояние ее сканов**. Откат к началу транзакции инициируется также вызовом 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**. **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 вр*е*менным 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 блокировок 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 явную синхронизационную блокировку указанной таблицы** (параметром операции является идентификатор таблицы). Выполнение операции `LOCK` **гарантирует, что никакая другая транзакция до конца данной не сможет изменить эту таблицу** (вставить в нее новый кортеж, удалить или модифицировать существующий), если установлена блокировка в режиме чтения, или **даже прочитать любой** кортеж этой таблицы, если установлена монопольная

блокировка.

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 было бы реализовать SQL с использованием оставшейся части операций. Операция `LOCK` введена в интерфейс RSS для возможности оптимизации выполнения запросов.

Дело в том, что, как видно из описания интерфейса RSS, этот **интерфейс является покортежным**. 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 table_name`, 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 всей таблицы.

Но ситуации, в которых 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**.

35. ACID-транзакции. Средства СУБД для поддержки свойств атомарности, согласованности, изолированности и постоянства хранения.

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 в многопользовательских системах.

Более точно, в современных СУБД **поддерживается понятие транзакции, характеризуемое аббревиатурой ACID** (Atomicy, Consistency, Isolation и Durability). В соответствии с этим понятием **под транзакцией разумеется последовательность операций над базой данных, обладающая следующими свойствами**.

- **Атомарность (Atomicy)**. Это свойство означает, что **результаты всех** операций, успешно выполненных в пределах транзакции, должны быть отражены в состоянии базы данных, либо в состоянии базы данных **не должно быть** отражено действие **ни одной операции** (конечно, здесь речь идет об операциях, изменяющих состояние базы данных). Свойство атомарности, которое часто называют свойством "**все или ничего**", позволяет относиться к транзакции, как к динамически образуемой **составной операции над базой данных** (в общем случае состав и порядок выполнения операций, выполняемых внутри транзакции, становится известным только на стадии выполнения).
- **Согласованность (Consistency)**. В классическом смысле это свойство означает, что **транзакция может быть успешно завершена с *фиксацией* результатов** своих операций только в том случае, к**огда действия операций не нарушают**

целостность **базы данных**, т.е. удовлетворяют набору ограничений целостности, определенных для этой базы данных. Это свойство расширяется тем, что во время выполнения транзакции **разрешается устанавливать точки согласованности и явным образом проверять ограничения целостности.** (С точки зрения автора, в контексте баз данных т**ермины *согласованность* и *целостность* эквивалентны**. Единственным к**ритерием согласованности данных является их удовлетворение ограничениям целостности**, т.е. база данных находится в согласованном состоянии тогда и только тогда, когда она находится в целостном состоянии.)

- **Изоляция (Isolation)**. Требуется, чтобы **две одновременно (параллельно или квазипараллельно) выполняемые транзакции никоим образом не действовали одна на другую**. Другими словами, **результаты выполнения операций транзакции *T1* не должны быть видны никакой другой транзакции *T2* до тех пор, пока транзакция *T1* не завершится успешным образом**.
- **Долговечность (Durability)**. После успешного завершения транзакции **все изменения**, которые были внесены в состояние базы данных операциями этой транзакции, **должны гарантированно сохраняться**, даже в случае сбоев аппаратуры или программного обеспечения.
- 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.

Атомарность

В этом смысле 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 состоянии базы данных, либо воздействие всех этих операторов полностью отсутствует.

Лозунгом транзакции является «Все или ничего»: при завершении транзакции **оператором** `COMMIT` (высокоуровневый аналог операции `END TRANSACTION` в интерфейсе RSS, см. лекцию 12) р**езультаты гарантированно фиксируются во внешней памяти** (смысл термина *commit* состоит в запросе «фиксации» результатов транзакции); **при завершении транзакции оператором** `ROLLBACK` (высокоуровневый аналог операции `RESTORE` в интерфейсе RSS, см. лекцию 12) результаты гарантированно отсутствуют во внешней памяти (смысл термина *rollback* состоит в запросе ликвидации результатов транзакции).

Понятие транзакции имеет непосредственную связь с понятием целостности базы данных. Очень часто база данных может обладать такими ограничениями целостности, которые просто невозможно не нарушить, выполняя только один оператор изменения базы данных.

Поэтому для поддержки 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 базы данных 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 база данных остается в таком состоянии, в котором находилась к моменту начала транзакции, т.е. в целостном состоянии.

Более точно, 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 (например, 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 100000 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` с откатом транзакции до ее начала или **до установленной ранее точки сохранения** или **постараться устранить причины** нецелостного состояния базы данных внутри транзакции (видимо, это осмысленно только при использовании интерактивного режима работы).

Заметим, что концептуально в момент завершения транзакции проверяются все откладываемые ограничения целостности, определенные в этой базе данных. Однако в реализации стремятся при выполнении транзакции динамически выделить те ограничения целостности, которые действительно могли бы быть нарушены. Например, если при выполнении транзакции над базой данных СЛУЖАЩИЕ–ОТДЕЛЫ в ней не выполнялись операторы вставки или удаления кортежей из отношения СЛУЖАЩИЕ, то проверять упоминавшееся выше ограничение целостности не требуется (а для проверки подобных ограничений требуется достаточно большая работа).

Понятно, что описанный механизм поддержки целостности баз данных обеспечивает требуемое свойство транзакций: **никакая транзакция не может быть зафиксирована, если ее действия нарушили целостность базы данных**. Однако в этом подходе имеются **два серьезных дефекта**.

Во-первых, **если** при выполнении транзакции **не устанавливать точки сохранения и не проверять периодически соответствие текущего состояния базы данных** (с точки

зрения данной транзакции) ограничениям целостности, то **долговременно выполняемая транзакция вполне вероятно может быть «откачена» системой при выполнении завершающего оператора** `COMMIT`. Конечно, это означает непроизводительный расход системных ресурсов и времени пользователей. Во-вторых, **чем длиннее транзакция, модифицирующая состояние базы данных, тем потенциально больше ограничений целостности придется проверять при ее завершении** и тем накладнее становится оператор `COMMIT`.

Для решения этой проблемы, авторы **предлагают отказаться от откладываемых ограничений целостности базы данных**, а вместо этого **ввести составные операторы изменения базы данных** (нечто наподобие блоков `BEGIN … END`, поддерживаемых в языках программирования). После выполнения каждого такого блока (или отдельного оператора изменения базы данных, используемого без операторов начала и конца блока) база данных должна находится в целостном состоянии. Если составной оператор нарушает ограничение целостности, то он целиком отвергается, и вырабатывается соответствующий код ошибки. Транзакция в этом случае не откатывается. Понятно, что при использовании такого подхода **при выполнении оператора** `COMMIT` **не требуется проверять ограничения целостности, и каждая зафиксированная транзакция будет оставлять базу данных в целостном состоянии.**

Интересно, что для реализации описанного подхода не требуются какие-либо новые механизмы, кроме точек сохранения транзакции, насильственной проверки ограничений целостности и частичных откатов транзакций, а отмеченные ранее проблемы снимаются.

Изолированность транзакций

В многопользовательских системах с одной базой данных одновременно может 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 базой данных в одиночку**.

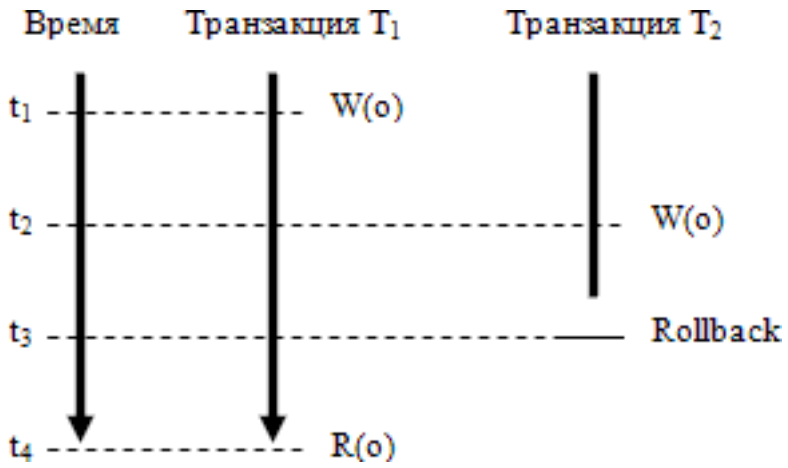
В связи со свойством сохранения целостности базы данных **транзакции являются подходящими единицами изолированности пользователей**. Действительно, если с каждым сеансом работы пользователя или приложений с базой данных ассоциируется транзакция, то **каждый пользователь начинает работу с согласованным 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 поддержки целостности базы данных возможно наличие нескольких уровней изолированности транзакций.

Отсутствие потерянных изменений (первый уровень изолированности)

Рассмотрим сценарий совместного выполнения двух транзакций. В момент времени t_1 транзакция T_1 изменяет объект базы данных `o` (выполняет операцию `W(o)`). До

завершения транзакции T_1 в момент времени $t_2 > t_1$ транзакция T_2 также изменяет объект o. В момент времени $t_3 > t_2$ транзакция T_2 завершается оператором ROLLBACK (например, по причине нарушения ограничений целостности).



Потерянные изменения

Тогда при повторном чтении объекта o (выполнении операции R(o)) в момент времени $t_4 > t_3$ транзакция T_1 **не видит своих изменений этого объекта, произведенных ранее** (в частности, из-за этого может не удастся фиксация этой транзакции, что, возможно, повлечет потерю изменений у еще одной транзакции и т.д.).

Такая ситуация называется ситуацией **потерянных изменений**. Естественно, она противоречит требованию изолированности пользователей. Чтобы избежать такой ситуации в транзакции T_1 требуется, чтобы **до завершения транзакции T_1 никакая другая транзакция не могла изменять никакой измененный транзакцией T_1** объект o (в частности, достаточно заблокировать доступ по изменению к объекту o до завершения транзакции T_1). **Отсутствие потерянных изменений является минимальным требованием к СУБД при обеспечении изолированности одновременно выполняемых транзакций**.

Отсутствие чтения «грязных» данных (второй уровень изолированности)

Рассмотрим сценарий совместного выполнения транзакций T_1 и T_2, показанный на рис. 13.2. В момент времени t_1 транзакция T_1 изменяет объект базы данных o (выполняет операцию W(o)). В момент времени $t_2 > t_1$ транзакция T_2 читает объект o (выполняет операцию R(o)). Поскольку транзакция T_1 еще не завершена, **транзакция T_2 видит несогласованные *«грязные» данные***. В частности, в момент времени $t_3 > t_2$ транзакция T_1 может завершиться откатом (например, по причине нарушения ограничений целостности).

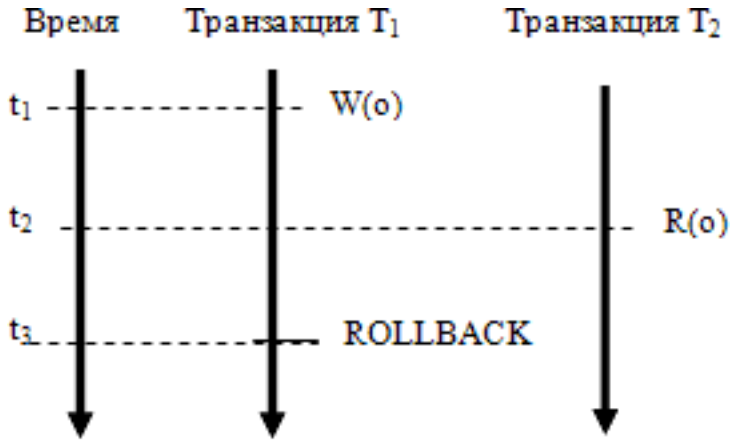


Рис. 13.2. «Грязные» чтения

Эта ситуация тоже **не соответствует требованию изолированности пользователей (каждый пользователь начинает свою транзакцию при согласованном состоянии базы данных и имеет право видеть только согласованные данные). Чтобы избежать ситуации чтения "грязных" данных, до завершения транзакции** T_1, изменившей объект базы данных `o`, **никакая другая транзакция не должна читать объект** `o` (например, достаточно заблокировать доступ по чтению к объекту `o` до завершения изменившей его транзакции T_1).

Отсутствие неповторяющихся чтений (третий уровень изоляции)

Рассмотрим сценарий совместного выполнения транзакций T_1 и T_2, показанный на рис. 13.3. В момент времени t_1 транзакция T_1 читает объект базы данных `o` (выполняет операцию `R(o)`). До завершения транзакции T_1 в момент времени $t_2 > t_1$ транзакция T_2 изменяет объект `o` (выполняет операцию `W(o)`) и успешно завершается оператором `COMMIT`. В момент времени $t_3 > t_2$ транзакция T_1 повторно читает объект `o` и видит его измененное состояние.

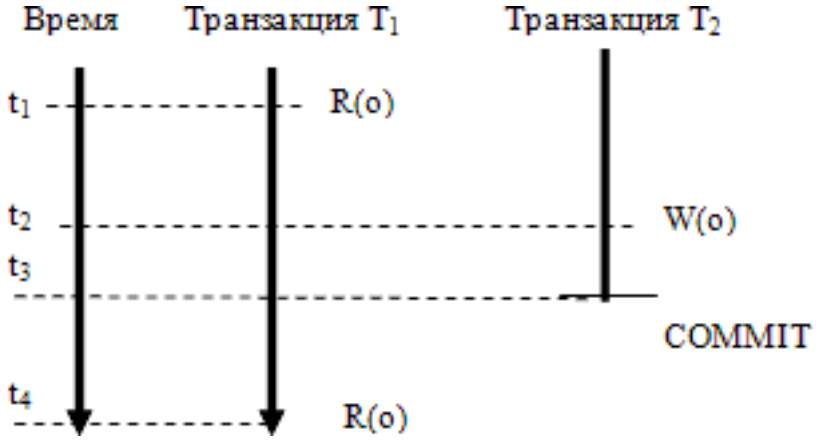


Рис. 13.3. Неповторяющиеся чтения

Чтобы избежать **неповторяющихся чтений, до завершения транзакции** T_1 **никакая**

другая транзакция не должна изменять объект o (для этого достаточно заблокировать доступ по записи к объекту o до завершения транзакции T_1). Часто это является **максимальным требованием к средствам обеспечения изолированности транзакций**, хотя, как будет видно 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 (кстати, соответствующие операторы были предусмотрены уже в стандарте SQL:1992). Как уже отмечалось, для корректного соблюдения ограничений целостности достаточен первый уровень. Существует ряд приложений, которым хватает первого уровня изолированности (например, прикладные или системные статистические утилиты, для которых некорректность индивидуальных данных несущественна). При этом удается существенно сократить накладные расходы СУБД и повысить общую эффективность.

Проблема фантомов

К более тонким проблемам изолированности транзакций относится так называемая **проблема кортежей-«фантомов»**, приводящая к ситуациям, которые также противоречат изолированности пользователей. Рассмотрим сценарий, показанный на рис. 13.4.

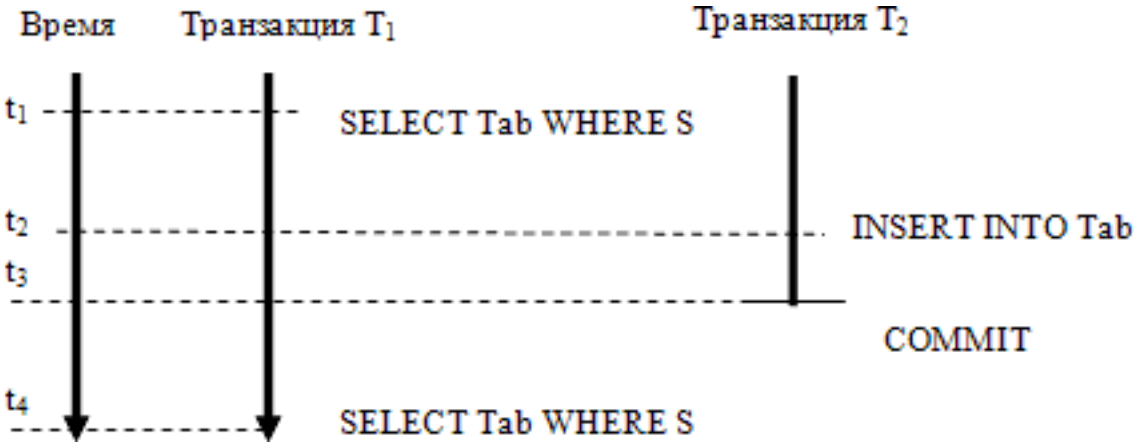


Рис. 13.4. Проблема фантомов

В момент времени t_1 транзакция T_1 выполняет оператор выборки кортежей таблицы Tab с условием выборки S (т.е. выбирается часть кортежей таблицы Tab, удовлетворяющих условию S). **До завершения транзакции** T_1 в момент времени $t_2 > t_1$ транзакция T_2 **вставляет в таблицу** Tab новый кортеж r, удовлетворяющий условию S, и успешно завершается. В момент времени $t_3 > t_2$ транзакция T_1 повторно выполняет тот же оператор выборки, 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 изоляции транзакций. Идеи требуемого механизма (предикатные синхронизационные блокировки) появились также еще во время выполнения проекта System R, но в большинстве систем не реализованы.

Постоянство хранения.

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 (т.е. должно поддерживаться свойство *долговечности (durability)* транзакций);
- результаты незафиксированных транзакций должны отсутствовать в восстановленном состоянии базы данных (в противном случае состояние базы данных могло бы оказаться не целостным).

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.
- Восстановление после внезапной потери содержимого оперативной памяти (мягкий сбой).
- Восстановление после поломки основного внешнего носителя базы данных (жесткий сбой).

Во всех трех случаях основой восстановления является хранение избыточных 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 основной (правильнее сказать, в виртуальной) памяти СУБД. Кроме того, **поддерживается общий журнал изменений базы данных, используемый для восстановления состояния базы**

данных после мягких и жестких сбоев.

Данный подход позволяет быстро выполнять индивидуальные откаты транзакций, но приводит к дублированию информации в локальных и общем журналах. Поэтому чаще используется второй вариант – поддержка только общего журнала изменений базы данных, который используется и при выполнении индивидуальных откатов.

36. Сериализация транзакций, виды конфликтов транзакций и порождаемые феномены поведения транзакций. Двухфазный протокол синхронизированных блокировок.

Чтобы добиться изолированности транзакций, в СУБД **должны использоваться какие-либо методы регулирования совместного выполнения транзакций**.

Пусть в системе одновременно выполняется некоторое множество транзакций $S = \{T_1, T_2, \ldots, Tn\}$. **План (способ) выполнения набора транзакций** S (в котором, вообще говоря, чередуются или реально параллельно выполняются операции разных транзакций) **называется сериальным, если результат совместного выполнения транзакций эквивалентен результату некоторого последовательного выполнения этих же транзакций** $(Ti_1, Ti_2, \ldots, Tin)$.

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 чередование их операций или реальную 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 свойства сериальности. Примерами могут служить только читающие транзакции, а также транзакции, не конфликтующие по объектам базы данных.

Между транзакциями T_1 и T_2 **могут существовать следующие виды конфликтов**:

- **W/W** – транзакция T_2 пытается изменять объект, измененный не закончившейся транзакцией T_1 (наличие такого конфликта может привести к возникновению **ситуации потерянных изменений**);
- **R/W** – транзакция T_2 пытается изменять объект, прочитанный не закончившейся транзакцией T_1 (наличие такого конфликта может привести к возникновению ситуации **неповторяющихся чтений**);
- **W/R** – транзакция T_2 пытается читать объект, измененный не закончившейся

транзакцией T_1 (наличие такого конфликта может привести к возникновению ситуации **«грязного» чтения**).

Практические методы сериализации транзакций основываются на учете этих конфликтов.

Наиболее распространенным в централизованных СУБД (включающих системы, основанные на архитектуре «клиент-сервер») является подход, основанный на **соблюдении двухфазного протокола синхронизационных захватов объектов баз данных** (Two-Phase Locking Protocol, 2PL). В общих чертах подход состоит в том, что перед выполнением любой операции в транзакции T над объектом базы данных o от имени транзакции T **запрашивается синхронизационная блокировка объекта o в соответствующем режиме (в зависимости от вида операции).**

Основными режимами синхронизационных блокировок являются следующие:

- **совместный режим** – S (Shared), означающий совместную (по чтению) блокировку объекта и требуемый для выполнения операции чтения объекта;
- **монопольный режим** – X (eXclusive), означающий монопольную (по записи) блокировку объекта и требуемый для выполнения операций вставки, удаления и модификации объекта.

Блокировки одних и тех же объектов по чтению несколькими транзакциями **совместимы**, т.е. **нескольким транзакциям допускается одновременно читать один и тот же объект**. **Блокировка объекта одной транзакцией по чтению не совместима с блокировкой другой транзакцией того же объекта по записи**, т.е. **никакой транзакции нельзя изменять объект, читаемый некоторой транзакцией** (кроме самой этой транзакции), и **никакой транзакции нельзя читать объект, изменяемый некоторой транзакцией** (кроме самой этой транзакции). **Блокировки одного и того же объекта по записи разными транзакциями не совместимы**, т.е. никакой транзакции нельзя изменять объект, изменяемый некоторой транзакцией (кроме самой этой транзакции). Правила совместимости захватов одного объекта разными транзакциями приведены в таблице 10.1.

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**.

Таблица 9.1. Совместимость блокировок S и X

	X	S
-	да	да
X	нет	нет

S	нет	да

Заметим, что слово «нет» (отсутствие совместимости блокировок) в этой таблице соответствует описанным ранее возможным случаям конфликтов транзакций по доступу к объектам базы данных (W/W, R/W, W/R). Совместимость S-блокировок соответствует тому, что конфликт R/R не существует.

Для обеспечения сериализации транзакций (третьего уровня изолированности) **синхронизационные блокировки 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 (выполнение операций над базой данных) – **накопление блокировок**;
- вторая фаза (фиксация или откат) – **снятие блокировок**.

Достаточно легко убедиться, что **при соблюдении двухфазного протокола** синхронизационных блокировок действительно **обеспечивается сериализация** транзакций на **третьем уровне изолированности**. Также легко видеть, что для обеспечения отсутствия потерянных данных достаточно блокировать в режиме X изменяемые объекты базы данных и удерживать эти блокировки до конца транзакции, а для обеспечения отсутствия чтения «грязных» данных достаточно блокировать в режиме X изменяемые объекты до конца транзакции и блокировать в режиме S читаемые объекты на время выполнения операции чтения.

Основная проблема состоит в том, **что следует считать объектом для синхронизационного захвата**? В контексте реляционных баз данных возможны следующие альтернативы:

- **файл** (сегмент в терминах System R) – физический (с точки зрения базы данных) **объект,** область хранения нескольких таблиц и, возможно, индексов;
- **таблица** – логический объект, соответствующий множеству кортежей данной таблицы;
- **страница данных** – физический объект, хранящий кортежи одной или нескольких таблиц, индексную или служебную информацию;
- **кортеж** – элементарный физический объект базы данных.

На самом деле, любая операция над объектом базы данных фактически воздействует и на объемлющие его объекты. Например, операция над кортежем является и операцией над страницей, в которой этот кортеж хранится, и над соответствующей таблицей, и над файлом, содержащим таблицу. Поэтому действительно имеется выбор уровня объекта блокировки.

Понятно, что для поддержки блокировок требуются системные ресурсы, и что **чем крупнее объект синхронизационного захвата (**неважно, какой природы этот объект – логический или физический), **тем меньше синхронизационных блокировок будет поддерживаться в системе**, и на это, соответственно, **будут тратиться меньшие накладные расходы**. Более того, **если устанавливать блокировки на уровне файлов или таблиц, то будет решена даже проблема фантомов** (если это не ясно сразу,

посмотрите еще раз описание проблемы фантомов и определение двухфазного протокола захватов).

Но вся беда в том, что при использовании для блокировок крупных объектов **возрастает вероятность конфликтов транзакций** и, тем самым, у**меньшается допускаемая степень чередования их операций или реального параллельного выполнения**. Фактически, при укрупнении объекта синхронизационной блокировки мы умышленно огрубляем ситуацию и видим конфликты в тех ситуациях, в которых на самом деле конфликтов нет.

Разработчики многих систем **начинали с использования страничных блокировок**, полагая это некоторым компромиссом между стремлениями сократить накладные расходы и сохранить достаточно высокий уровень параллельности транзакций. Но это не очень хороший выбор. Не будем останавливаться на деталях, но заметим, что использование страничных блокировок в двухфазном протоколе иногда вызывает очень неприятные синхронизационные проблемы, усложняющие организацию СУБД (коротко говоря, эти проблемы связаны с тем, что страницы приходится блокировать на двух разных уровнях – уровне управления буферами страниц в основной памяти и уровне выполнения логических операций). **В большинстве современных систем используются покортежные синхронизационные блокировки**.

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

37. Гранулированные и предикатные блокировки.

Гранулированные блокировки.

При применении этого подхода синхронизационные **блокировки могут запрашиваться по отношению к объектам разного уровня: файлам, таблицам и кортежам**. Требуемый **уровень объекта определяется тем, какая операция выполняется** (например, для выполнения операции уничтожения таблицы объектом синхронизационной блокировки должна быть вся таблица, а для выполнения операции удаления кортежа – этот кортеж). Объект любого уровня может **быть заблокирован в режиме S или X.**

Для согласования блокировок разного уровня вводятся специальный протокол гранулированных блокировок и новые типы блокировок. Коротко говоря, перед установкой блокировки на некоторый объект базы данных в режиме S или X соответствующий объект верхнего уровня должен быть заблокирован в режиме IS, IX или SIX.

Блокировка в режиме IS (Intented for Shared lock) некоторого составного объекта o базы данных означает намерение заблокировать некоторый объект o', входящий в o, в

совместном режиме (режиме S). Например, при намерении читать кортежи из таблицы `Tab` эта таблица должна быть заблокирована в режиме IS (а до этого в таком же режиме должен быть заблокирован файл, в котором располагается таблица `Tab`).

Блокировка в режиме IX (Intented for eXclusive lock) некоторого составного объекта `o` базы данных означает намерение заблокировать некоторый объект `o'`, входящий в `o`, в монопольном режиме (режиме X). Например, для удаления кортежей из таблицы `Tab` эта таблица должна быть заблокирована в режиме IX (а до этого в таком же режиме должен быть заблокирован файл, в котором располагается таблица `Tab`).

Блокировка в режиме SIX (Shared, Intented for eXclusive lock) некоторого составного объекта `o` базы данных означает совместную блокировку всего этого объекта с намерением впоследствии блокировать какие-либо входящие в него объекты в монопольном режиме (режиме X). Например, если выполняется длинная операция просмотра таблицы `Tab` с возможностью удаления некоторых просматриваемых кортежей, то экономичнее всего заблокировать таблицу `Tab` в режиме SIX (а до этого заблокировать в режиме IS файл, в котором располагается таблица `Tab`).

Совместимость блокировок S, X, IS, IX и SIX

	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

В таб. 9.2 приведена таблица совместимости блокировок S, X, IS, IX и SIX. Немного поясним правила совместимости. Должно быть понятно, что для атомарных объектов разумны только блокировки в режимах S и X, для которых правила совместимости остаются такими же, как были показаны в таб. 9.1. Пусть теперь o – это некоторый составной объект.

Тогда блокировка объекта o в режиме X в транзакции T_1 не совместима с блокировкой этого объекта в режимах X, S, IX, IS или SIX в транзакции T_2. Действительно, блокировка объекта o в режиме X в транзакции T_1 направлена на то, чтобы изменять объект o целиком. Несовместимость блокировки объекта o в режиме X в транзакции

T_1 с его блокировкой в режиме X или IX в транзакции T_2 устраняет конфликты транзакций T_1 и T_2 вида W/W. Несовместимость блокировки объекта o в режиме X в транзакции T_1 с его блокировкой в режиме S или IS в транзакции T_2 устраняет конфликты транзакций T_1 и T_2 вида W/R. Наконец, несовместимость блокировки объекта o в режиме X в транзакции T_1 с его блокировкой в режиме SIX в транзакции T_2 устраняет конфликты транзакций T_1 и T_2 вида W/R и W/W.

Блокировка объекта o в режиме S в транзакции T_1 совместима с блокировкой этого объекта в режимах S или IS в транзакции T_2, поскольку эти блокировки в транзакциях T_1 и T_2 направлены только на то, чтобы только читать некоторые объекты o', входящие в o. Блокировка объекта o в режиме S в транзакции T_1 не совместима с блокировкой этого объекта в режимах X, IX или SIX в транзакции T_2, поскольку любая из этих блокировок направлена на то, чтобы изменять в транзакции T_2 объект o целиком или какой-либо объект o', входящий в o. Несовместимость блокировки объекта o в режиме S в транзакции T_1 с блокировкой этого объекта в режимах X, IX или SIX в транзакции T_2, тем самым, устраняет конфликты транзакций T_1 и T_2 вида R/W.

Блокировка объекта o в режиме IX в транзакции T_1 совместима с блокировкой этого же объекта в режимах IS или IX в транзакции T_2. Действительно, блокировка объекта o в режиме IX в транзакции T_1 направлена на то, чтобы в этой транзакции изменять какой-либо объект o', входящий в o, а блокировка этого же объекта в режиме IS в транзакции T_2 – на то, чтобы читать в транзакции T_2 какой-либо объект o'', входящий в o. Если объекты o' и o'' – разные, то конфликт транзакций T_1 и T_2 не возникнет. Если $o' = o''$, то перед изменением этот объект будет заблокирован в транзакции T_1 в режиме X, а перед чтением – в транзакции T_2 в режиме S. Несовместимость этих блокировок позволит избежать конфликта транзакций T_1 и T_2 вида W/R, и для этого не требуется несовместимость блокировок IX и IS объекта o. Аналогично обосновывается совместимость блокировок IX и IX. Блокировка IX не совместима с блокировкой S, поскольку иначе мог бы проявиться конфликт транзакций T_1 и T_2 вида W/R. Блокировка IX не совместима с блокировкой X, поскольку иначе мог бы проявиться конфликт транзакций T_1 и T_2 вида W/W. Наконец, блокировка IX не совместима с блокировкой SIX, поскольку иначе мог бы проявиться конфликт транзакций T_1 и T_2 вида W/R или W/W.

Блокировка объекта o в режиме IS в транзакции T_1 совместима с блокировкой этого же объекта в режимах S, IS, IX или SIX в транзакции T_2. Совместимость с блокировкой в режиме S или IS уже обосновывалась. Покажем, что блокировка объекта o в режиме IS в транзакции T_1 совместима с блокировкой того же объекта в

режиме IX в транзакции T_2. Действительно, блокировка объекта o в режиме IS в транзакции T_1 направлена на то, чтобы в этой транзакции читать какой-либо объект o', входящий в o, а блокировка этого же объекта в режиме IX в транзакции T_2 – на то, чтобы в транзакции T_2 изменять какой-либо объект o'', входящий в o. Если объекты o' и o'' – разные, то конфликт транзакций не возникнет. Если $o' = o''$, то перед чтением этот объект будет заблокирован в транзакции T_1 в режиме S, а перед изменением – в транзакции T_2 в режиме X. Несовместимость этих блокировок позволит избежать конфликта транзакций T_1 и T_2 вида R/W, и для этого не требуется несовместимость блокировок IS и IX объекта o. Аналогично можно показать совместимость блокировок IS и SIX. Несовместимость блокировок IS и X очевидна, поскольку иначе мог бы проявиться конфликт транзакций T_1 и T_2 вида R/W.

Блокировка объекта o в режиме SIX в транзакции T_1 позволяет этой транзакции читать любой объект o', входящий в o, без его дополнительной блокировки и изменять любой объект o', входящий в o, с его предварительной блокировкой в режиме X. Эта блокировка совместима с блокировкой объекта o в режиме IS в транзакции T_2. Действительно, блокировка объекта o в режиме IS в транзакции T_2 направлена на то, чтобы в транзакции T_2 читать какой-либо объект o', входящий в o. Перед этим в транзакции T_2 должна быть установлена блокировка объекта o' в режиме S. К этому моменту у объекта o' может отсутствовать явная блокировка, установленная в транзакции T_1, что, в соответствии с семантикой блокировки SIX, означает наличие неявной блокировки o' по чтению. Очевидно, что в этом случае конфликт транзакций T_1 и T_2 не возникает. К этому же моменту у объекта o' может иметься блокировка в режиме X, установленная в транзакции T_1. В этом случае запрос блокировки объекта o' в режиме S удовлетворен не будет, и конфликт транзакций T_1 и T_2 вида W/R будет предотвращен без потребности в несовместимости блокировок SIX и IS. Блокировка объекта o в режиме SIX в транзакции T_1 не совместима с блокировкой объекта o в режиме X в транзакции T_2, поскольку иначе мог бы проявиться конфликт транзакций T_1 и T_2 вида R/W. Блокировка объекта o в режиме SIX в транзакции T_1 не совместима с блокировкой объекта o в режиме S или IS в транзакции T_2, поскольку иначе мог бы проявиться конфликт транзакций T_1 и T_2 вида W/R при доступе к некоторым объектам o', входящим в o. Наконец, блокировка объекта o в режиме SIX в транзакции T_1 не совместима с блокировкой объекта o в режиме IX или SIX в транзакции T_2, поскольку иначе мог бы проявиться конфликт транзакций T_1 и T_2 вида R/W при доступе к некоторым объектам o', входящим в o.

Предикатные блокировки.

Несмотря на привлекательность метода **гранулированных синхронизационных захватов**, следует отметить, что он **не решает проблему фантомов** (**если, конечно, не ограничиться использованием блокировок таблиц в режимах S и X**). Давно известно, что для решения этой проблемы **необходимо перейти** от блокировок индивидуальных («физических») объектов базы данных, **к блокировке условий** (предикатов), которым удовлетворяют эти объекты. **Проблема фантомов не возникает при использовании для блокировок уровня таблиц** именно потому, что таблица как логический объект представляет собой неявное условие для входящих в него кортежей. **Блокировка таблицы** – это **простой и частный случай предикатной блокировки.**

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 **синхронизационную блокировку** 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 блокировок**. Ясно, что без этого использовать предикатные блокировки для сериализации транзакций невозможно, а в общей форме проблема неразрешима.

Один из компромиссных подходов предлагался участниками проекта System R. Подход основывался на том, что **при открытии сканирования таблицы по индексу в RSS передается дополнительная информация (диапазон сканирования), которая ограничивает множество кортежей, среди которых не должны возникать фантомы**.

Опираясь на наличие этой информации, предлагалось ввести в систему блокировок System R **элементы предикатных блокировок**. Заметим сначала, что в System R блокировки сегментов (файлов), таблиц и кортежей технически трактовались единообразно, как блокировки идентификаторов кортежей (tid'ов). **При блокировке кортежа на самом деле блокировался его tid**. При блокировке сегмента или таблицы на самом деле блокировался tid описателя соответствующего объекта во внутренних таблицах-каталогах сегментов или таблиц.

Предлагалось расширить систему синхронизации, разрешив применять блокировки к паре «идентификатор индекса, интервал значений ключа этого индекса». К такой паре можно было применять блокировки в любом из допустимых режимов, причем две такие блокировки считались совместимыми в том и только в том случае, если они были совместимы в соответствии с приведенной таб. 13.2 или указанные диапазоны значений ключей не пересекались.

При наличии такой возможности, если открывается сканирование таблицы через индекс, то таблица блокируется в режиме IS, и в этом же режиме блокируется пара «идентификатор индекса, диапазон сканирования». При занесении (удалении) кортежа таблица блокируется в режиме IX, и в этом же режиме для каждого индекса, определенного на данной таблице отношении, **блокируется пара «идентификатор индекса, значение ключа из затрагиваемого операцией кортежа»**. Это позволяет избежать конфликтов читающих транзакций с теми изменяющими транзакциями,

которые затрагивают диапазоны сканирования читающих транзакций. При этом решается проблема фантомов, и **параллельность транзакций ограничивается «по существу»**, т.е. только в тех случаях, когда их параллельное выполнение создает проблемы.

Заметим сразу, что описанное решение проблемы фантомов далеко от идеального. Во-первых, по-прежнему **при сканировании таблиц без использования индексов отсутствие фантомов можно гарантировать только при блокировке всего отношения в режиме S**. Во-вторых, даже при сканировании по индексу **условие реальной выборки кортежа часто может быть гораздо строже простого указания диапазона сканирования**, а это значит, что блокировка этого диапазона будет слишком сильной, т.е. затронет более широкое множество кортежей, чем то, которое будет реальным результатом сканирования.

Известно следующее более совершенное решение. Будем называть простым условием конъюнкцию простых предикатов сравнения, имеющих вид имя_поля { = > < } значение. В типичных СУБД, поддерживающих двухуровневую организацию (языковой уровень и уровень управления внешней памяти), в интерфейсе подсистемы управления памятью (которая обычно заведует и сериализацией транзакций) допускаются только простые условия. Подсистема языкового уровня производит компиляцию оператора SQL со сложным условием в последовательность обращений к подсистеме управления памятью, в каждом из которых содержатся только простые условия.

Более точно, **простое условие явно указывается в операции открытия сканирования таблицы** (напрямую или через индекс; в последнем случае оно конъюнктивно соединяется с условием, задаваемым диапазоном сканирования). Кроме того, **при открытии сканирования всегда можно указать, для какой цели оно будет использоваться**: для выборки кортежей, для их удаления или для их обновления (это известно компилятору SQL). Кроме того, **неявные условия задаются операциями вставки и удаления кортежей** (конъюнктивное логическое выражение, состоящее из простых предикатов вида имя_поля = значение для всех полей таблицы), а т**акже операциями обновления кортежей** (конъюнктивное логическое выражение, состоящее из простых предикатов вида имя_поля = значение для всех обновляемых полей таблицы). Поэтому в случае типовой организации SQL-ориентированной СУБД **простые условия можно использовать как основу предикатных захватов**.

Для простых условий совместимость предикатных блокировок легко определяется на основе следующей геометрической интерпретации. Пусть Tab – таблица с полями a_1, a_2, ..., an, а m_1, m_2, ..., mn – множества допустимых значений a_1, a_2, ..., an соответственно (естественно, все эти множества – конечные). Тогда можно сопоставить Tab конечное n-мерное пространство возможных значений кортежей Tab. Легко видеть, что любое простое условие, представляющее собой конъюнкцию простых предикатов, «вырезает» в этом пространстве k-мерный прямоугольник ($k \le n$).

Достаточно очевидно следующее утверждение:

Пусть имеются два простых условия $\texttt{scond}_1$ и $\texttt{scond}_2$. Пусть транзакция $\texttt{T}_1$ запрашивает блокировку $\texttt{scond}_1$, а транзакция $\texttt{T}_2$ – $\texttt{scond}_2$ в режимах, которые были бы несовместимы, если бы $\texttt{scond}_1$ и $\texttt{scond}_2$ являлись не условиями, а объектами базы данных (S-X, X-S, X-X). Эти блокировки совместимы в том и только в том случае, когда прямоугольники, соответствующие $\texttt{scond}_1$ и $\texttt{scond}_2$, не пересекаются.

Это утверждение действительно очевидно (каждому *k*-мерному прямоугольнику в *n*-мерном пространстве возможных значений кортежей `Tab` соответствует некоторое подмножество возможных значений кортежей, и отсутствие пересечения у двух прямоугольников гарантирует отсутствие конфликтов транзакций), но для наглядности на рис. 13.5 приводится иллюстрирующий пример, показывающий, что в каких бы режимах не требовала транзакция $\texttt{T}_1$ блокировки условия `(0 < a < 5) & (b = 5)`, а транзакция $\texttt{T}_2$ – блокировки условия `(0 < a <6) & (0 < b <4)`, эти блокировки всегда будут совместимы.

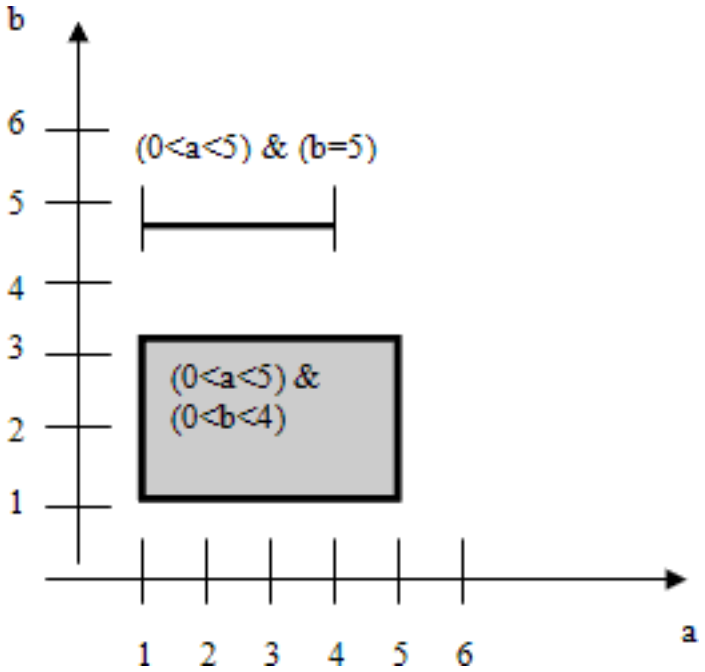


Рис. 13.5. Простые условия, блокировки которых совместимы

Интересно, что **при поддержке такой системы блокировок простых условий можно обойтись без гранулированных блокировок**. В частности, чтобы гарантированно заблокировать таблицу целиком, достаточно заблокировать условие $\&_{1 \le i \le n}$ `(min(mi) < ИМЯ_ПОЛЯi < max(mi))`. **Чтобы заблокировать базу данных, достаточно заблокировать условие, являющееся конъюнкцией условий блокировки всех таблиц этой базы данных.**

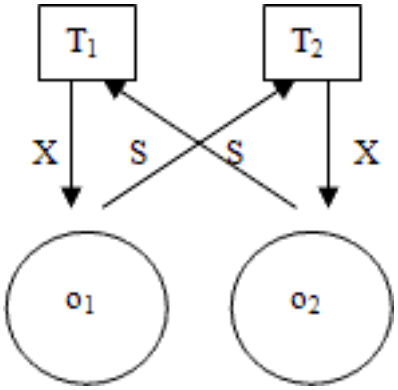
Заметим, что блокировки простых условий описываются таблицами, немногим

отличающимися от таблиц традиционных синхронизаторов с гранулированными блокировками. Поэтому введение в СУБД механизма предикатных блокировок не приводит к значительным усложнениям.

38. Синхронизационные тупики, способы их обнаружения и разрушения.

Одним из **наиболее чувствительных недостатков метода сериализации транзакций на основе синхронизационных блокировок является возможность возникновение тупиков** (deadlocks) **между транзакциями**. Синхронизационные тупики возможны при применении любого из рассмотренных выше вариантов механизмов блокировок.

На рисунке показан простой сценарий возникновения синхронизационного тупика между транзакциями T_1 и T_2:



Ситуация синхронизационного тупика между транзакциями T_1 и T_2

- транзакции T_1 и T_2 устанавливают монопольные блокировки объектов o_1 и o_2 соответственно;
- после этого T_1 требуется совместная блокировка объекта o_2, а T_2 – совместная блокировка объекта o_1;
- ни одно из этих требований блокировки не может быть удовлетворено, следовательно, ни одна из транзакций не может продолжаться; поэтому монопольные блокировки объектов никогда не будут сняты, а требования совместных блокировок не будут удовлетворены.

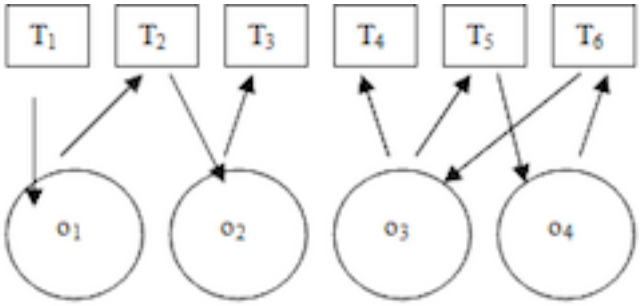
Поскольку тупики возможны, и никакого естественного выхода из тупиковой ситуации не существует, то эти ситуации необходимо обнаруживать и искусственно устранять.

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** (будем изображать их прямоугольниками), и **вершины, соответствующие объектам блокировок** (будем изображать их окружностями). В этом графе **дуги соединяют только вершины-транзакции с вершинами-объектами**. **Дуга из вершины-транзакции к вершине-объекту**

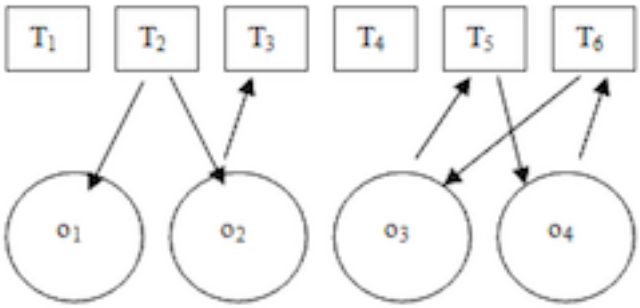
существует в том и только в том случае, **если для этой транзакции имеется удовлетворенная блокировка** данного объекта. **Дуга из вершины-объекта к вершине-транзакции существует** тогда и только тогда, когда эта **транзакция ожидает удовлетворения запроса блокировки данного объекта**. Легко показать, что в системе **существует тупиковая ситуация в том и только в том случае, когда в графе ожидания транзакций имеется хотя бы один цикл**. Простейший пример графа ожидания транзакций с циклом показан на рис. 13.6.

Для распознавания тупиковых ситуаций 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**.

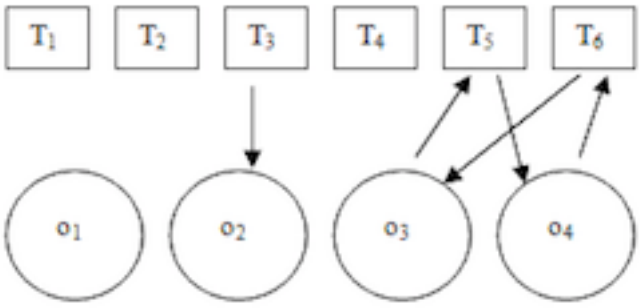
Пример применения алгоритма редукции к графу ожидания транзакций показан на рис. 13.7 (в целях упрощения примера предполагается, что **все блокировки являются монопольными, т.е. для каждой вершины-объекта имеется не более одной входящей дуги**). В этом случае редукция состоит в том, что, прежде всего, из графа ожидания (начальное состояние которого показано на рис. 13.7 (a)) **удаляются все дуги, исходящие из вершин-транзакций, в которые не входят дуги из вершин-объектов**. (Это основывается на том разумном предположении, что транзакции, не ожидающие удовлетворения запроса блокировок, могут успешно завершиться и освободить блокировки). Кроме того, **удаляются дуги, входящие в вершины-транзакции, из которых не исходят, ведущие к вершинам-объектам** (транзакции, ожидающие удовлетворения блокировок, но не удерживающие заблокированные объекты, не могут быть причиной тупика). **Для тех вершин-объектов, для которых не осталось входящих дуг, но существуют исходящие**, **ориентация одной из исходящих дуг (выбираемой произвольным образом) изменяется на противоположную** (это моделирует удовлетворение запроса блокировки). Состояние графа после выполнения первого шага редукции показано на рис. 13.7 (b). После этого снова повторяются описанные действия (состояние графа после выполнения второго шага редукции показано на рис. 13.7 (c)), и так до тех пор, пока не прекратится удаление дуг. Если в графе остались дуги, то они обязательно образуют цикл (см. рис. 13.7 (c)).



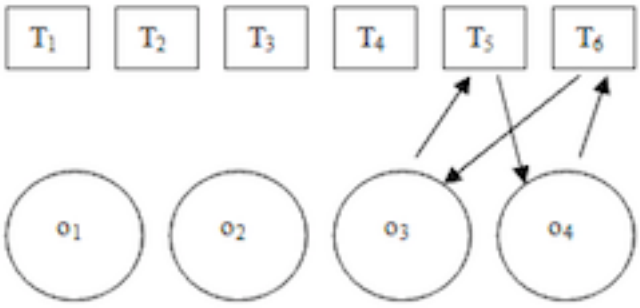
(a) Исходное состояние графа ожидания



(b) Первый шаг



(c) Второй шаг



(d) Конечный вид графа ожидания

Рис. 13.7. Применение алгоритма редукции к графу ожидания транзакций

Предположим теперь, что н**ам удалось найти цикл в графе ожидания транзакций**.

Разрушение тупиков

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**.

Выбрать «жертву» не так уж легко, поскольку для этого могут использоваться различные, зачастую противоречивые критерии. С одной стороны, было бы разумно жертвовать наиболее «богатой» транзакцией, т.е. той **транзакцией, которая удерживает наиболее число блокировок объектов**. В этом случае после принудительно завершения такой транзакции освободилось бы наибольшее число объектов, что с большой вероятностью привело бы к исчезновению тупиковой ситуации. Но, с другой стороны, «богатая» транзакция, скорее всего, выполнялась дольше других транзакций. **На ее выполнение уже затрачено большое количество системных ресурсов и, вероятно, она скоро завершится самостоятельно**. Поэтому этот выбор может оказаться в системном отношении **не самым удачным**.

Можно пожертвовать самой «молодой» транзакцией, которая существует в системе в течение наименьшего времени. Такую транзакцию менее всего жалко, поскольку она еще н**е успела израсходовать много системных ресурсов**. Но, с другой стороны, такая транзакция **не могла и накопить много блокировок**, и поэтому ее насильственное завершение **вряд ли поможет устранить тупиковую ситуацию**.

Можно выбрать транзакцию-жертву случайным образом из всех транзакций, попавших в тупик. Возможно, что в среднем этот подход привел бы к хорошим результатам. Но, к сожалению, в нем **не учитывается возможная приоритетность транзакций**. Было бы не слишком хорошо, например, жертвовать транзакцией, запущенной от имени руководителя организации.

Поэтому **обычно при выборе транзакции-жертвы используется многофакторная оценка** ее стоимости, **в которую с разными весами входят время выполнения, число накопленных блокировок, приоритет и т.д.** В качестве «жертвы» выбирает транзакция, для которой эта оценка выдает наиболее подходящий результат.

После выбора транзакции-жертвы выполняется откат этой транзакции, который **может носить полный или частичный** (до некоторой точки сохранения) характер. При этом, естественно, освобождаются блокировки, 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 распределенных СУБД**, в которых транзакции могут выполняться в разных узлах сети. Поэтому в таких системах обычно используются другие методы сериализации транзакций.

39. Сериализация транзакций на основе временных меток.

Альтернативный метод сериализации транзакций, **хорошо работающий в условиях редкого возникновения конфликтов транзакций и не требующий построения графа ожидания транзакций, основан на использовании временных меток**. **Основная идея метода временны*х* меток** (Timestamp Ordering, TO), у которого существует множество разновидностей, состоит в следующем: если транзакция T_1 началась раньше транзакции T_2, то **система обеспечивает такой сериальный план, как если бы транзакция T_1 была целиком выполнена до начала T_2**.

Для этого каждой транзакции T **предписывается временн*а*я метка** t(T), **соответствующая времени начала выполнения транзакции** T. При выполнении операции над объектом o **транзакция T помечает его своими идентификатором, временн*о*й меткой и типом операции (чтение или изменение)**.

Перед выполнением операции над объектом o транзакция T_2 выполняет следующие действия:

- Проверяет, **помечен ли объект** o какой-либо транзакцией T_1. **Если не помечен, то помечает этот объект** своей временн*о*й меткой и **типом операции и выполняет операцию**. Конец действий.
- Иначе транзакция T_2 **проверяет, не завершилась ли** транзакция T_1, **пометившая** этот объект. **Если** транзакция T_1 **закончилась**, то T_2 **помечает объек**т o и выполняет свою операцию. Конец действий.
- **Если транзакция T_1 не завершилась**, то T_2 **проверяет конфликтность операций**. Если операции **неконфликтны, то** при объекте o **запоминается идентификатор транзакции T_2, остается или проставляется временн*а*я метка с меньшим значением**, и транзакция T_2 **выполняет свою операцию**.
- **Если операции транзакций T_2 и T_1 конфликтуют**, то если $t(T_1) > t(T_2)$ (т.е. транзакция T_1 является более «молодой», чем T_2), **то производится откат T_1 и всех других транзакций, идентификаторы которых сохранены при объекте** o, и T_2 выполняет свою операцию.
- **Если же $t(T_1) < t(T_2)$** (T_1 «старше» T_2), то **производится откат T_2**; T_2 **получает новую временну*ю* метку и начинается заново**.

К недостаткам метода TO 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 временн*ы*е метки с отношением полного порядка (это отдельная большая наука).

Но в распределенных системах эти недостатки окупаются тем, ч**то не нужно распознавать тупики, а как мы уже отмечали, построение графа ожидания в**

распределенных системах стоит очень дорого.

40. Версионный вариант алгоритма временных меток.

Основная идея алгоритмов сериализации транзакций, описываемых в этом разделе, состоит в том, что в базе данных допускается существование **нескольких «версий» одного и того же объекта**. Эти алгоритмы, главным образом, направлены на преодоление конфликтов транзакций категорий R/W и W/R, **позволяя выполнять операции чтения над некоторой предыдущей версией объекта базы данных**. В результате операции чтения выполняются без задержек и тупиков, свойственных механизмам синхронизационных блокировок, а также без некоторых откатов, возможных при применении метода временны*х* меток, описанного в предыдущем подразделе.

Алгоритмы управления транзакциями, основанные на поддержке версий, достаточно широко распространены в области SQL-ориентированных СУБД. В частности, подобные алгоритмы используются в СУБД Oracle и PostgreSQL. В дальнейшем в этом подразделе будем называть алгоритмы этой категории *версионными* алгоритмами.

Версионный вариант алгоритма временных меток

Одним из наиболее старых и простых версионных алгоритмов является **версионный вариант алгоритма временных меток** *(Multiversion Timestamp Ordering, MVTO)*. Как и в простом методе временны*х* меток, описанном в предыдущем подразделе, в алгоритме MVTO **порядок выполнения операций одновременно выполняемых транзакций задается порядком временны*х* меток**, которые получают транзакции во время старта. **Временны*е* метки также используются для идентификации версий данных при чтении и модификации** – каждая **версия получает временную метку той транзакции, которая ее записала**. Алгоритм MVTO **не только следит за порядком выполнения операций транзакций**, но **также отвечает за трансформацию операций над объектами базы данных в операции над версиями этих объектов**, т.е. каждая **операция над объектом базы данных** o **преобразуется в соответствующую операцию над некоторой версией объект**а o.

При описании алгоритма будем использовать следующие обозначения. Как и раньше, **временну*ю* метку, полученную транзакцией** T_i в начале ее работы, будем обозначать как $t(T_i)$. **Операция чтения объекта** базы данных o, выполняемая в транзакции T_i, будет обозначаться как $R_i(o)$. Для обозначения того, что **транзакция** T_i **читает версию объекта базы данных** o, **созданную транзакцией** T_k, будем использовать запись $R_i(o_k)$. Для обозначения того, что **транзакция** T_i **записывает версию элемента данных** o, будем использовать запись $W_i(o_i)$.

Алгоритм MVTO работает следующим образом.

- Любая операция $R_i(o)$ **преобразуется** в операцию $R_i(o_k)$, где o_k – это

версия объекта o, помеченная **наибольшей временной меткой** $t(T_k)$, такой что $t(T_k) \leq t(T_i)$. Другими словами, транзакции T_i для чтения дается версия объекта o, созданная транзакцией T_k, которая не моложе T_i, но старше любой другой транзакции T_n, создававшей свою версию объекта o.

- При обработке операции $W_i(o)$ выполняются следующие действия:
  - **если к этому времени некоторой незафиксированной транзакцией** T_n **уже выполнена некоторая операция** $R_n(o_k)$, такая что $t(T_k) \leq t(T_i) < t(T_n)$, то операция $W_i(o)$ не выполняется, а **транзакция** T_i **откатывается**;
  - в противном случае $W_i(o)$ преобразуется в $W_i(o_i)$, т.е. **образуется еще одна версия объекта** o.
- **При откате любой транзакции уничтожаются все созданные ею версии объектов базы данных и откатываются все транзакции, прочитавшие хотя бы одну из этих версий.** Тем самым, откаты транзакций могут быть **«каскадными»**.
- **Выполнение операции фиксации транзакции** T_i (COMMIT) **откладывается до того момента, когда завершатся все транзакции**, **записавшие версии данных, прочитанные** T_i. **Легко видеть, что без соблюдения этого требования не соблюдалось бы свойство долговечности (durability) транзакций, поскольку при откате некоторых транзакций потребовалось бы откатывать и ранее зафиксированные транзакции.**

Преимущества алгоритма MVTO лучше всего иллюстрируются поведением транзакций T_1 и T_2 (см. рис. 13.8). **При использовании блокировок между ними возник бы синхронизационный тупик**, **а при использовании обычного метода временных меток одна из транзакций подверглась бы откату**. Однако при применении версий такие неприятности не возникают из-за того, что первая транзакция читает «старые» версии объектов o и ω.

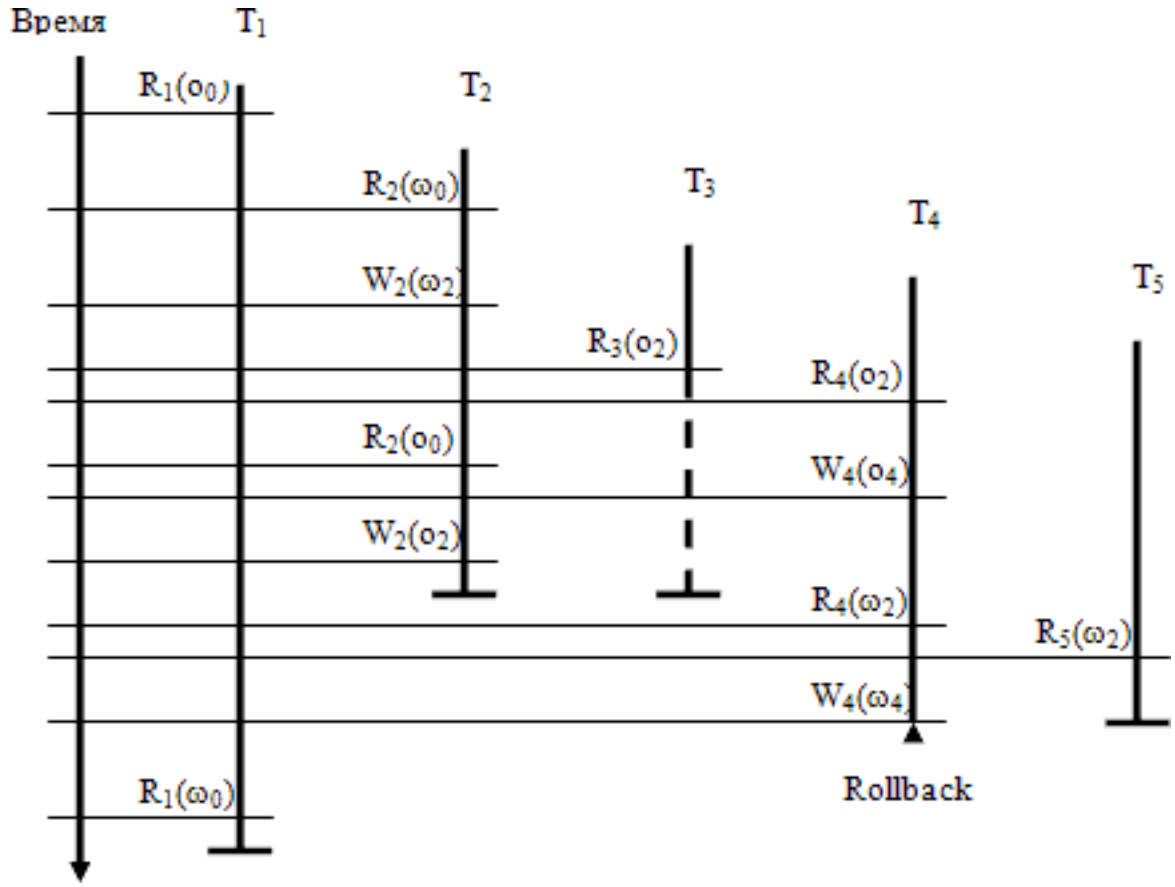


Рис. 13.8. Пример работы алгоритма MVTO

Транзакция T_3 ожидает фиксации транзакции T_2 перед своим собственным завершением (на рис. 13.8 это показано пунктирной линией). Это происходит потому, что транзакция T_3 прочитала версию o_2 объекта o, образованную еще не зафиксированной транзакцией.

Транзакция T_4 пытается создать версию ω_4 объекта ω после того, как еще не зафиксированная транзакция T_5 (начавшаяся позже) уже прочитала более раннюю версию ω_4. Поэтому транзакция T_5 не сможет «увидеть» изменения объекта ω, произведенные транзакцией T_4. Следовательно, сериализация транзакций в порядке получения ими временных меток становится невозможной, и приходится произвести откат транзакции T_4.

Итак, **основными преимуществами алгоритм**а MVTO **является отсутствие задержек и откатов при выполнении операций чтения**, а **основным недостатком – возможность возникновение каскадных откатов** транзакций при выполнении операций записи. Кроме того, **в базе данных может накапливаться произвольное число версий одного и того же объекта**, **и определение того, какие версии больше не требуются, является серьезной технической проблемой**.

41. Версионный вариант двухфазного протокола синхронизационных блокировок.

При описании **двухверсионного варианта протокола** *2PL (Two-Version Two-Phase Locking Protocol, 2V2PL)* будем называть **текущими** версиями объектов базы данных **версии, созданные зафиксированными транзакциями с наиболее поздним временем фиксации**; **незафиксированными** версиями – **версии, созданные еще незавершившимися транзакциями**. При следовании протоколу 2V2PL **в каждый момент времени существует не более одной незафиксированной версии каждого объекта базы данных.**

Операции любой транзакции T_i над объектом базы данных o обрабатываются следующим образом:

- **операция** $R_i(o)$ **немедленно выполняется над текущей** версией объекта o;
- **операция** $W_i(o)$, приводящая к созданию новой версии объекта o, **выполняется только после завершения (фиксации или отката) транзакции, создавшей незафиксированную версию объекта** o;
- **выполнение операции** `COMMIT` **откладывается** до тех пор, **пока не завершатся все транзакции** T_k, **прочитавшие текущие версии объектов базы данных, которые должны замениться незафиксированными версиями этих объектов, созданными транзакцией** T_i.

Для реализации такого поведения используются **три типа блокировок**:

- **RL** (Read Lock) – в этом режиме блокируется любой объект базы данных o **перед выполнением операции чтения его текущей версии**; удержание этой блокировки до конца транзакции гарантирует, что при повторном чтении объекта o **будет прочитана та же версия этого объекта**;
- **WL** (Write Lock) – в этом режиме блокируется любой объект базы данных o **перед выполнением операции , приводящей к созданию новой (незафиксированной) версии** этого объекта; удержание этой блокировки до конца транзакции гарантирует, что в любой момент времени **будет существовать не более одной незафиксированной версии любого объекта базы данных**;
- **CL** (Commit Lock) – **блокировка устанавливается во время выполнения операции** `COMMIT` транзакции и **затрагивает любой объект базы данных, новую версию которого создала данная транзакция**; удовлетворение этой блокировки для данной транзакции **гарантирует, что завершились все транзакции, читавшие текущие версии объектов**, **новые версии которых были созданы при выполнении данной транзакции**, и, следовательно, их можно заменить.

Таблица 9.3. Таблица совместимости «версионных» блокировок

	RL(o)	WL(o)	CL(o)
RL(o)	да	да	нет
WL(o)	да	нет	нет
CL(o)	нет	нет	нет

Как видно, операция чтения может блокироваться только на время фиксации транзакции, заменяющей текущую версию требуемого объекта базы данных. Для выполнения операции записи требуется долговременная монопольная блокировка соответствующего объекта базы данных, которая, однако, в этом случае совместима с блокировкой этого же объекта по чтению (поскольку в действительности блокируются разные версии этого объекта). И, конечно, как и во всех схемах сериализации транзакций на основе блокировок, здесь возможны синхронизационные тупики.

42. Версионно-блокировочный протокол сериализации для поддержки только читающих транзакций.

В заключение обсудим гибридный протокол, **поддерживающий эффективное выполнение транзакций, не изменяющих состояние базы данных** (*Multiversion Protocol for Read-Only Transactions, ROMV*). При применении этого протокола п**ри образовании каждой транзакции явно указывается ее тип – только читающая** (read-only) **или изменяющая** (update) транзакция. В только читающих транзакциях допускается использование **только операций чтения** объектов базы данных, а в изменяющих транзакциях – **операций и чтения, и записи**.

Изменяющие транзакции выполняются в соответствии **с обычным протоколом 2PL**, т.е. **перед выполнением операции чтения или записи** объекта базы данных o этот объект **должен быть заблокирован в режиме** S или X соответственно, и **блокировки объектов удерживаются до конца изменяющей транзакции**. **Каждая операции записи объекта o создает его новую версию**, **которая при завершении транзакции помечается временн*о*й меткой, соответствующей моменту фиксации этой транзакции.**

Каждая только читающая транзакция при своем образовании **получает соответствующую временную метку**. При выполнении операции чтения объекта базы данных o транзакция **получает доступ к версии объекта o**, **образованной изменяющей транзакцией**, **которая хронологически последней зафиксировалась к моменту образования данной читающей транзакции**.

Основным плюсом протокола ROMV по сравнению с ранее описанным протоколом 2V2PL является **принципиальное отсутствие синхронизационных задержек при выполнении операций чтения только читающих транзакций**. Если сравнивать ROMV с MVTO, то **он выигрывает в принципиальном отсутствии откатов только читающих транзакций**. Конечно, **при работе изменяющих транзакций возможно возникновение синхронизационных тупиков и откатов**, и здесь т**ребуется использовать обычные методы распознавания и разрушения тупиков**.

Кроме того, при использовании протокола ROMV в базе данных **может возникать произвольное число версий объектов**. **Требуется создание специального сборщика мусора, который должен удалять ненужные версии данных**. Простейший сборщик мусора **удаляет все неиспользуемые версии, значения временных меток которых меньше значения временной метки старейшей активной только читающей транзакции**.

43. Ситуации, требующие восстановления базы данных. Индивидуальные откаты транзакций. Понятие журнала.

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** (т.е. должно поддерживаться **свойство *долговечности*** *(durability)* транзакций);
- **результаты незафиксированных транзакций должны отсутствовать в восстановленном состоянии базы данных** (в противном случае состояние базы данных могло бы оказаться не целостным).

Это, собственно, и означает, что **восстанавливается последнее по времени согласованное состояние базы данных**.

Специальный набор данных – **журнал**, в который помещаются записи обо всех операциях всех транзакций, изменяющих 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 разрушении 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 основной памяти) и т.д**. Ситуация характеризуется потерей той части базы данных, которая к моменту сбоя содержалась в буферах оперативной памяти СУБД**.
- **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 хранение избыточных 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 основной (правильнее сказать, в виртуальной) памяти СУБД**. Кроме того, **поддерживается общий журнал изменений базы данных, используемый для восстановления состояния базы данных после мягких и жестких сбоев**.

Данный подход **позволяет быстро выполнять индивидуальные откаты транзакций**, **но приводит к дублированию информации в локальных и общем журналах**. Поэтому **чаще используется второй вариант – поддержка только общего журнала изменений базы данных**, который **используется и при выполнении индивидуальных откатов**. Здесь мы рассматриваем именно этот вариант.

Индивидуальный откат транзакции

Для **обеспечения возможности индивидуального отката транзакции по общему журналу все записи в журнале от данной транзакции связываются в обратный список**. В начале списка для незавершенных транзакций **находится запись о последнем изменении базы данных, произведенном данной транзакцией**. Заметим, что в этом случае хронологически последние записи могут быть еще не вытолкнуты во внешнюю память журнала и могут находиться в буфере основной памяти. **Для закончившихся транзакций** (индивидуальные откаты которых уже невозможны) **началом списка является запись о конце транзакции**, которая обязательно вытолкнута во внешнюю память журнала, т.е. весь список находится во внешней

памяти. **Концом списка всегда служит первая запись об изменении базы данных, произведенном данной транзакцией**. Обычно в каждой записи проставляется у**никальный 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 незавершенных транзакций)** выполняется следующим образом:

- **Выбирается очередная журнальная 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 журнализуется**. Собственно для индивидуального отката это не нужно, но **при выполнении индивидуального отката транзакции может произойти мягкий сбой**, **при восстановлении после которого потребуется откатить транзакции, для которых не полностью выполнен индивидуальный откат**.
- **При успешном завершении отката в журнал заносится запись о конце транзакции**. **С точки зрения журнала такая транзакция является зафиксированной.**

44. Управление буферами основной памяти.

Журнализация операций изменения базы данных тесно **связана не только с управлением транзакциями**, **но и с буферизацией блоков базы данных в основной памяти**. По причинам **объективно существующей разницы в скорости работы процессоров и основной памяти и устройств внешней памяти** (эта разница в скорости существовала, существует, и будет существовать всегда) буферизация блоков базы данных в основной памяти является единственным реальным способом достижения приемлемой эффективности СУБД. **Без поддержки буферизации** базы данных **СУБД работала бы со скоростью магнитных дисков**, т.е. на несколько порядков медленнее, чем если бы обработка данных происходила в основной памяти.

Если бы каждая запись об изменении базы данных, **которая должна поступить в журнал** при выполнении любой операции обновления базы данных, **реально немедленно перемещалась бы во внешнюю память**, это п**ривело бы к существенному замедлению работы системы**. Фактически, тогда **каждая операция обновления базы данных выполнялась бы со скоростью магнитного диска**. Поэтому **записи в журнал тоже буферизуются: при нормальной работе буфер выталкивается во внешнюю память журнала только при полном заполнении записями**. Более точно, для буферизации записей журнала обычно **используются два буфера**. После полного заполнения **первый буфер выталкивается** на магнитный диск, и **пока совершается этот обмен**, журнальные **записи размещаются во втором буфере**. К моменту конца обмена заполняется второй буфер, он выталкивается во внешнюю память, а журнальные записи снова размещаются в первом буфере и т.д.

Здесь следует заметить, что здесь идет **речь об использовании буферов** (и базы

данных, и журнала), **располагающихся именно в физической основной памяти**, **управляемой непосредственно СУБД**, **а не виртуальной памяти СУБД, управляемой операционной системой**. Использование буферов виртуальной памяти является практически бессмысленным делом, поскольку в этом случае операционная система, руководствуясь своими собственными стратегиями управления основной памяти, в любой момент может удалить буферную страницу СУБД из основной памяти и перенести ее копию во внешнюю память в область свопинга. Тогда при следующей попытке записи СУБД в эту страницу возникнет прерывание, при обработке которого операционная система подкачает страницу в основную память, выполнив совершенно не ожидаемый СУБД обмен с внешней памятью.

Нельзя надеяться на то, что операционная система настолько грамотно управляет основной памятью, что нужные страницы виртуальной памяти СУБД в нужное время будут находиться в основной памяти. Операционная система просто не обладает достаточной информацией, чтобы всегда принимать правильные решения. Правильно управлять своей буферной памятью может только сама СУБД, «отбирающая» у операционной системы часть физической основной памяти для размещения в ней буферов базы данных и журнала.

В развитых (вернее сказать, правильно организованных) СУБД поддерживается собственная стратегия замещения страниц буферного пула. Задача, которую решает СУБД, очень похожа на задачу, которую решает операционная система при управлении виртуальной памятью.

В случае операционной системы, **если некоторый процесс требует обеспечения доступа к странице виртуальной памяти, отсутствующей в основной памяти, и нет свободных страниц основной памяти, в соответствии с некоторым критерием выбирается некоторая занятая страница основной памяти, освобождается** (т.е. изымается из виртуальной памяти какого-то процесса и, может быть, копируется на диск) и подключается к виртуальной памяти запросившего процесса с предварительным считыванием с диска нужных данных.

В случае СУБД, если **при выполнении некоторой операции в некоторой транзакции требуется доступ к некоторому блоку базы данных**, **и копия этого блока отсутствует в буферном пуле, СУБД должна выделить какую-либо страницу буферного пула, считать в нее с диска требуемый блок базы данных и предоставить доступ к этой странице запросившей операции**. Конечно, в буферном пуле **может не оказаться свободных страниц**, и тогда **СУБД в соответствии с некоторым критерием находит некоторую занятую страницу, освобождает ее** (возможно, выталкивает во внешнюю память).

Основная разница между этими случаями состоит в критерии выборки занятой страницы для «откачки». Не будем обсуждать здесь стратегии замещения страниц, используемые в операционных системах. Заметим лишь, что почти всегда **операционная система стремится заменить страницу, к которой предположительно дольше всего не будет обращений**, но, поскольку предвидение будущего невозможно, **оно аппроксимируется прошлым**. В частности, в одном из популярных алгоритмов замещения страниц LRU (Least Recently Used) принимается предположение, что **дольше всего в будущем не потребуется та страница, к которой дольше всего не**

обращались в прошлом.

В стратегии замещения страниц буферного пула СУБД тоже чаще всего используется **некоторая разновидность алгоритма LRU**. Но, как уже отмечалось выше, **СУБД располагает большей информацией о страницах буферного пула, чем операционная система о страницах основной памяти**.

Например, **если в некоторой транзакции выполняется сканирование некоторой таблицы без использования индекса**, и **при выполнении операции** `NEXT` **был затребован доступ к некоторому блоку базы данных** (с соответствующим перемещением копии этого блока в некоторую страницу буферного пула), **то подсистема управления буферным пулом «знает», что эта страница еще точно потребуется до тех пор, пока не будет прочитан последний кортеж сканируемой таблицы, располагающийся в данной странице**. Более того, **СУБД «знает», какой блок базы данных потребуется после завершения просмотра кортежей данного блока, и может заранее переместить его копию в некоторую страницу буферного пула.**

Кроме того, **некоторые блоки базы данных заведомо требуются чаще других блоков.** Например, **при любом просмотре таблицы на основе некоторого индекса гарантированно потребуется доступ к корневому блоку соответствующего B-дерева. При вставке кортежа в любую таблицу или удалении из нее кортежа будет необходимо должным образом изменить все определенные для нее индексы, и для этого тоже гарантированно потребуется доступ к корневым блокам всех соответствующих B-деревьев.**

Поэтому в стратегии замещения страниц буферного пула базы данных обычно **используется алгоритм LRU с приоритетами страниц** (грубо говоря, **высокоприоритетные страницы стареют, т.е. становятся кандидатами на замещение, медленнее, чем низкоприоритетные страницы**). В частности, **страницы, содержащие копии корневых блоков индексов, являются настолько высокоприоритетными, что обычно никогда не замещаются.** Кроме того, **поддерживается предварительное считывание в буферную память копий блоков, доступ к которым вскоре понадобится.**

.45. Физическая синхронизация.

Поскольку в СУБД может одновременно («параллельно») выполняться несколько транзакций, вполне **реальна ситуация**, **когда в двух одновременно выполняемых операциях требуется доступ к одному и тому же блоку базы данных** (т.е. к одной и той же буферной странице, содержащей копию этого блока). Понятно, что в одновременном доступе для чтения содержимого блока ничего плохого нет, но параллельное изменение блока может привести к непредсказуемым результатам.

Следует заметить, что, **вообще говоря**, **координацию параллельного доступа к страницам буферного пула не обеспечивает логическая синхронизация**, используемая для сериализации транзакций. Например, предположим, что **в двух параллельно выполняемых транзакциях одновременно выполняются операции**

модификации кортежей, у одного из которых tid = (*n*, 1), а у другого tid = (*n*, 2). **Если в СУБД используются блокировки на уровне кортежей**, **то система допустит параллельное выполнение этих двух операций**, и **они будут одновременно изменять страницу, содержащую копию блока базы данных с номером *n***. При выполнении обеих операций **может потребоваться перемещение кортежей внутри этого блока**, и понятно, что в результате ничего хорошего, скорее всего, не получится. Аналогично, **логическая синхронизация может легко допустить параллельное выполнение нескольких операций**, **требующих обновления одного и того же индекса**. **Некоординированное параллельное обновление B-дерева с большой вероятностью приводит к разрушению его структуры**.

Поэтому при выполнении операций уровня RSS **необходимо поддерживать дополнительную «физическую» синхронизацию**, в которой **единицами блокировки служат страницы буферного пула (или блоки) базы данных**. В пределах операции **перед чтением из страницы буферного пула** (блока базы данных) требуется запросить у подсистемы управления буферным пулом **блокировку соответствующей страницы (блока) в режиме S, а перед записью в страницу (в блок) – ее блокировку в режиме X**.

Но **блокировки страниц буферного пула нужны не только для координации параллельного доступа к страницам при параллельном выполнении транзакций**. При выполнении операций **уровня RSS** могут возникать **ошибки, обнаруживаемые в середине операции**, уже после того, как **одна или несколько страниц буферного пула (блоков базы данных) было изменено**. Например, **может выполняться операция вставки кортежа в некоторую таблицу, нарушающая уникальность некоторого индекса,** определенного над этой таблицей. **Нарушение уникальности** этого индекса **будет обнаружено при попытке вставить в него новый ключ,** но **до этого новый кортеж уже мог быть размещен в блоке данных**, и **некоторые индексы уже могли быть успешно обновлены**.

При обнаружении ошибки операции **нужно ликвидировать все ее следы в базе данных** и **выдать соответствующий код ошибки** на уровень RDS. **Проще всего сделать это, произведя обратные изменения всех страниц** (блоков базы данных), которые были изменены при прямом выполнении операции. Но **для этого требуется, чтобы все страницы (блоки базы данных), заблокированные при выполнении операции, оставались заблокированными до конца этой операции**.

Тем самым, **для подсистемы управления буферным пулом операции уровня RSS являются (почти) тем же, чем являются транзакции для подсистемы управления транзакциями**. **Достаточным условием корректного выполнения операций является соблюдение двухфазного протокола синхронизационных блокировок над страницами буферного пула в пределах операций**.

Заметим (хотя и без подробных объяснений), что **это условие не является необходимым**. Каждую операцию уровня RSS **можно разбить на последовательность « микроопераций » и потребовать соблюдения двухфазного протокола синхронизационных блокировок в пределах микроопераций**. Например, операцию `INSERT` уровня RSS можно разбить на следующие микрооперации:

1) нахождение блока данных для вставки;
2) вставка кортежа в найденный блок;
3) обновление индекса 1;
…….
n) обновление индекса *n*,

где *n* – число индексов, определенных для данной таблицы. **Общий принцип состоит в том, что в пределах одной микрооперации блокируются все блоки базы данных, которые обязаны быть изменены согласованным образом.**

46. Протокол Write Ahead Log.

Имеются два вида буферов – **буфера журнала и буферный пул страниц основной памяти**, – **которые содержат связанную информацию**. И те, и другие **буфера могут выталкиваться во внешнюю память**. **Основной причиной** выталкивания буфера журнала является его **полное заполнение журнальными записями**. **Страницы буферного пула базы данных чаще всего выталкиваются во внешнюю память, когда требуется переместить в основную память некоторый блок базы данных, а свободных страниц в буферном пуле нет**. Тогда срабатывает алгоритм замещения страниц, выбирается страница, содержимое которой, вероятно, дольше всего не потребуется, и эта страница (если ее содержимое изменялось) выталкивается в соответствующий блок внешней памяти базы данных. **Проблема состоит в выработке некоторой общей политики выталкивания, которая обеспечивала бы возможность восстановления состояния базы данных после сбоев**.

Заметим, что **эта проблема не возникает при индивидуальных откатах транзакций**, поскольку **в этих случаях содержимое основной памяти не утрачено**, и при **восстановлении можно пользоваться содержимым как буфера журнала, так и буферных страниц базы данных**. Но **если произошел мягкий сбой, и содержимое буферов утрачено, то 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 буферных страниц базы данных является то, что **запись об изменении объекта базы данных должна оказаться во внешней памяти журнала раньше, чем измененный объект окажется во внешней памяти базы данных**. Соответствующий протокол журнализации (и управления буферизацией) называется WAL (Write Ahead Log, **«пиши сначала в журнал»**) и состоит в том, что **если требуется вытолкнуть во внешнюю память буферную страницу, содержащую измененный объект базы данных, то перед этим нужно гарантировать выталкивание во внешнюю память журнала буферной страницы журнала, содержащей запись об изменении этого объекта**.

При следовании протоколу WAL, 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 журнала 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 завершенная 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 транзакций, зафиксированных до момента сбоя**.

Самым 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**.

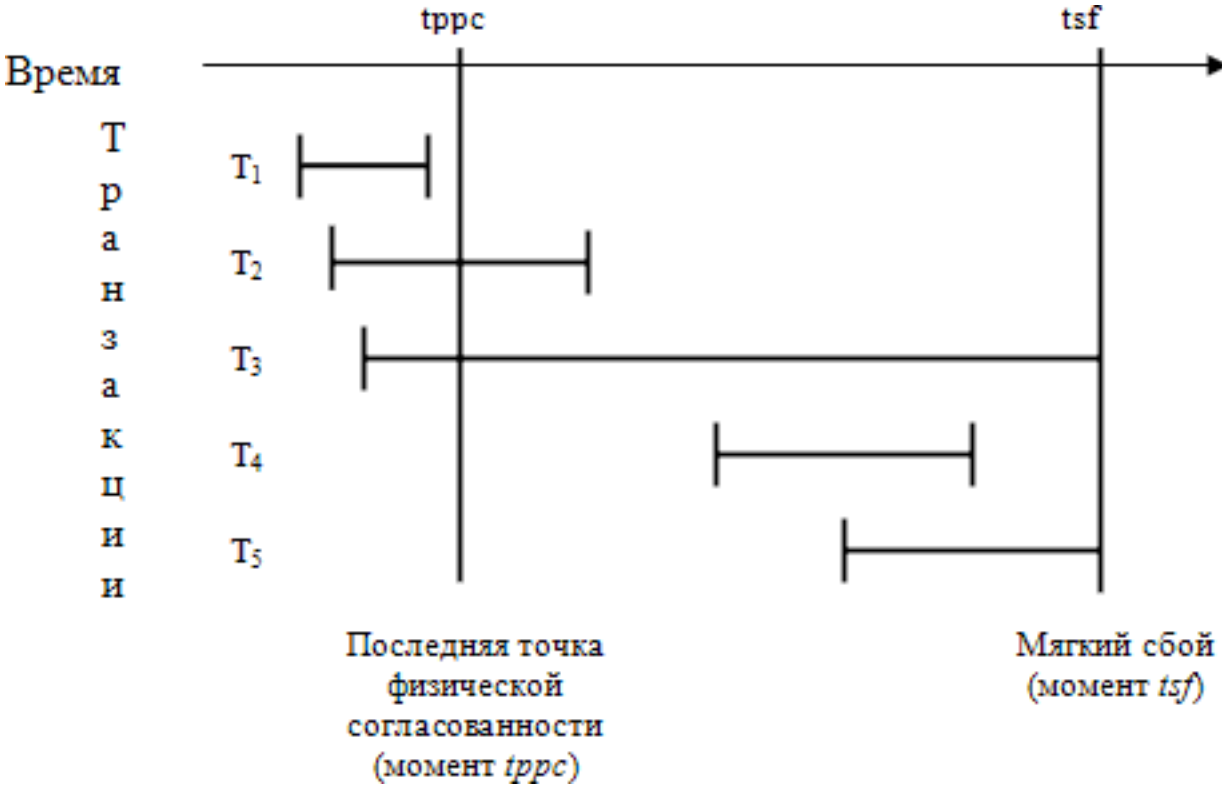
47. Физически согласованное состояние базы даных. Восстановление базы данных после мягкого сбоя.

К числу основных проблем восстановления после мягкого сбоя относится то, что одна **логическая операция изменения базы данных может изменять несколько физических блоков базы данных, например, блок данных и несколько блоков индексов.** Блоки базы данных буферизуются в оперативной памяти и выталкиваются независимо. **После мягкого сбоя набор блоков внешней памяти базы данных может оказаться несогласованным,** т.е. **часть блоков внешней памяти соответствует объекту до изменения, часть – после изменения**. Например, в результате выполнения операции `UPDATE` соответствующий кортеж мог переместиться в другой блок. В этом случае **изменяются два блока**: **в описатель кортежа в его исходном блоке записывается его новый tid**, **а в новом блоке размещается сам модифицированный кортеж**. Очевидно, что если хотя бы один из этих блоков не попал во внешнюю память базы данных к моменту мягкого сбоя, то при восстановлении не удастся вернуть кортеж на его прежнее место. Другими словами, к такому состоянию внешней памяти базы данных не применимы операции логического уровня.

Состояние внешней памяти базы данных называется **физически согласованным**, если **наборы страниц всех объектов согласованы**, **т.е. соответствуют состоянию любого 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 обсудим, как можно достичь

физической согласованности. Назовем такие точки ppc (point of physical consistency).



Возможные состояния транзакций к моменту мягкого сбоя

Предположим, что каким-то образом удалось восстановить внешнюю память базы данных к состоянию на момент времени tppc (как это можно сделать, обсудим в следующем подразделе). Тогда восстановление последнего по времени логически целостного состояния базы данных производится следующим образом.

- Для транзакции T_1 **никаких действий производить не требуется**. Она закончилась до момента *tppc*, и все ее результаты гарантированно отражены во внешней памяти базы данных.
- Для транзакции T_2 **нужно повторно выполнить** (*redo*) **последовательность операций, которые выполнялись после установки точки физически согласованного состояния в момент** *tppc*. Действительно, во внешней памяти полностью отсутствуют следы операций, которые выполнялись в транзакции T_2 после момента *tppc*. Следовательно, повторное прямое (по смыслу и хронологии) выполнение операций транзакции T_2 корректно и приведет к логически согласованному состоянию базы данных. (Поскольку транзакция T_2 успешно завершилась до момента мягкого сбоя *tfs*, в журнале содержатся записи обо всех изменениях базы данных, произведенных этой транзакцией.)
- Для транзакции T_3 **нужно выполнить в обратном направлении (undo) ту часть операций, которую она успела выполнить до момента** *tppc*. Действительно, во внешней памяти базы данных полностью отсутствуют результаты операций T_3, которые были выполнены после момента *tppc*. С

другой стороны, во внешней памяти гарантированно присутствуют результаты операций T_3, которые были выполнены до момента *tppc*. Следовательно, обратное выполнение (по смыслу и хронологии) операций T_3 корректно и приведет к согласованному состоянию базы данных. (Поскольку транзакция T_3 не завершилась к моменту мягкого сбоя *tfs*, при восстановлении необходимо устранить все последствия ее выполнения.)

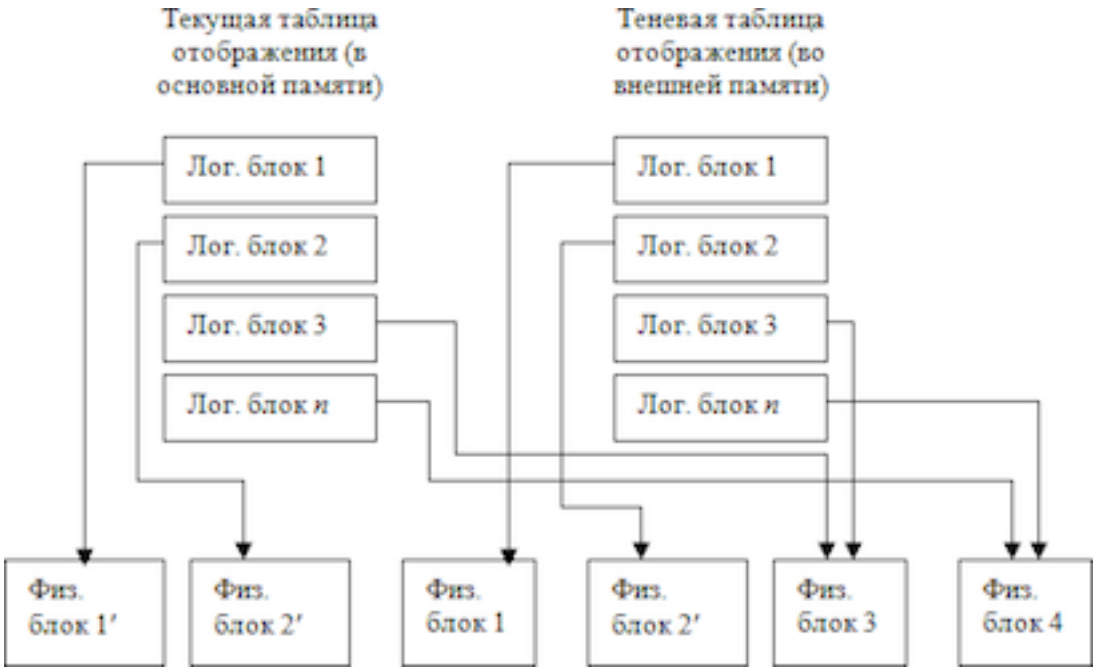
- Для транзакции T_4, которая успела начаться после момента *tppc* и закончиться до момента мягкого сбоя *tfs*, **нужно произвести полное повторное выполнение операций в прямом направлении.** (Поскольку транзакция T_4 успешно завершилась до момента мягкого сбоя *tfs*, в журнале содержатся записи обо всех изменениях базы данных, произведенных этой транзакцией).
- Наконец, для транзакции T_5, начавшейся после момента *tppc* и не успевшей завершиться к моменту мягкого сбоя *tfs*, н**икаких действий предпринимать не требуется. Результаты операций этой транзакции полностью отсутствуют во внешней памяти базы данных.**

48. Способы восстановления физически согласованного состояния.

Каким же образом можно обеспечить наличие точек физической согласованности базы данных, т.е. как восстановить состояние базы данных в момент *tppc*? Для этого используются два основных подхода: подход, основанный на использовании **теневого механизма**, и подход, в котором применяется **журнализация постраничных изменений базы данных.**

Теневой механизм

Теневой механизм был изначально предложен для поддержания целостности файлов при аварийном отключении питания компьютера. Файл представляется как набор блоков внешней памяти, для доступа к которым поддерживается таблица отображения. 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 основной памяти) изменяется, а теневая остается 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 основную память теневую таблицу отображения.**



. Теневой механизм для файлов

В контексте базы данных теневой механизм используется следующим образом. **Периодически выполняются операции установки точки физической согласованности базы данных.** При выполнении этой операции все **логические операции завершаются, все страницы буферного пула базы данных, содержимое которых отличается от содержимого соответствующих блоков внешней памяти, выталкиваются**. Теневая таблица отображения файлов (сегментов) базы данных заменяется текущей таблицей отображения (правильнее сказать, **текущая таблица отображения записывается на место теневой**).

Здесь имеется некоторая **проблема**, состоящая в том, что **в любой момент времени теневая** таблица отображения **должна быть корректной**, т.е. **соответствовать некоторому ранее зафиксированному физически целостному состоянию базы данных.** Для этого необходимо обеспечить **атомарность операции замены теневой таблицы отображения**. **В общем случае** таблица отображения может занимать несколько блоков внешней памяти, и для записи текущей таблицы отображения на место теневой таблицы в этом случае **потребуется несколько обменов с дисками**. Если в промежутке между этими обменами возникнет **мягкий сбой**, то будет благополучно утрачена текущая таблица отображения и **безнадежно испорчена теневая таблица**, т.е. мы просто лишимся возможности восстанавливаться за счет использования последнего физически согласованного состояния базы данных.

Чтобы это не произошло, **во внешней памяти поддерживаются две области хранения таблицы отображения файлов** (будем называть их областями `A` и `B`). Кроме того, в отдельном блоке внешней памяти **хранится флаг** `F`, показывающий, **какая из этих областей в данный момент содержит действующую теневую таблицу отображения** (назовем соответствующие значения флага F_A и F_B). Тогда, если сохраненным во внешней памяти значением флага является F_A, то текущая таблица отображения записывается в область `B`. Если эта операция выполняется успешно, то в **блок флага**

записывается значение F_B. Считается, что **операция записи одного блока на диск является атомарной**. Если эта операция заканчивается успешно, это означает, что новая теневая таблица отображения хранится в области B. Если же запись текущей таблицы отображения в область B не удалась, или если не выполнилась операция записи блока с флагом F, то продолжает действовать старая теневая таблица отображения.

Восстановление хронологически последнего сохраненного физически согласованного состояния базы данных происходит мгновенно: текущая таблица отображения заменяет теневой таблицей (при восстановлении просто считывается действующая 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 данны**х.

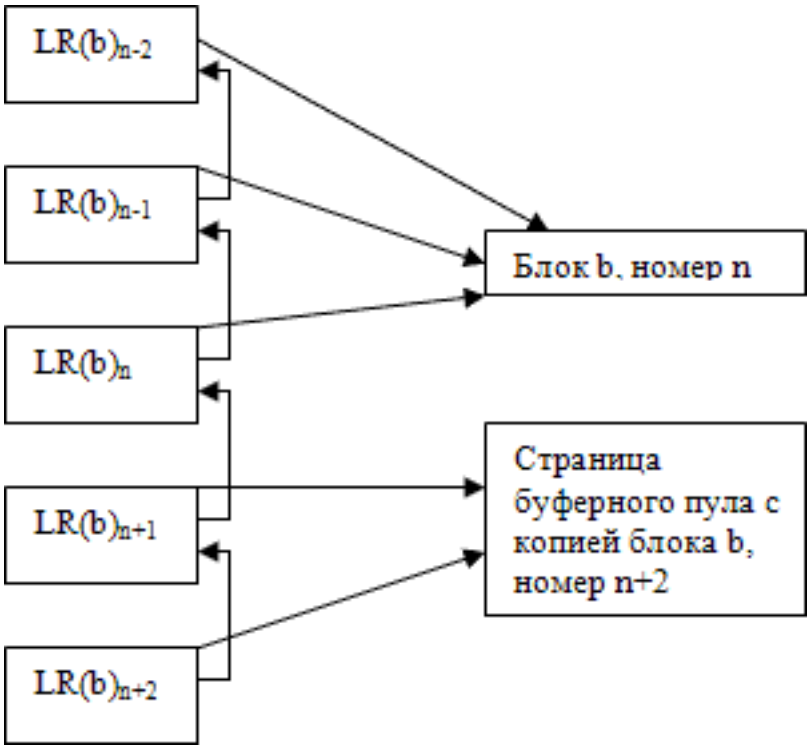
Журнализация постраничных изменений

Возможен другой подход, при использовании которого **наряду с логической журнализацией операций изменения базы данных производится журнализация постраничных изменений**. **Первый этап** восстановления после мягкого сбоя состоит в постраничном **откате недовыполненных** 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. Вообще, выполнение логических операций уровня RSS носит транзакционный характер. В частности, как уже отмечалось выше, при выполнении логической операции обновления базы данных, вообще говоря, изменяется несколько блоков базы данных. Для обеспечения возможности отката отдельной операции (а это может потребоваться, например, если обнаруживается нарушение свойства уникальности какого-либо индекса) приходится до конца операции монопольно блокировать все страницы буферного пула базы данных, содержащие копии изменяемых этой операцией блоков базы данных.

Чтобы распознать, нуждается ли страница внешней памяти базы данных в восстановлении, **при выталкивании любой страницы из буферного пула основной памяти в нее помещается номер последней записи о постраничном изменении этой страницы**. **Этот же номер запоминается в самой записи**. Тогда, чтобы понять, нужно ли применить данную запись о постраничном изменении соответствующего блока внешней памяти для восстановления состояния этого блока, **требуется всего лишь сравнить номер, содержащийся в этом блоке, с номером, содержащимся в журнальной записи**. **Если в блоке содержится номер, меньший номера журнальной записи, то это означает, что буферная страница, в которой выполнялось соответствующее изменение, не была к моменту мягкого сбоя вытолкнута во внешнюю память, и применять данную запись для восстановления соответствующего блока внешней памяти не требуется.**

Для иллюстрации на рис. 14.3 показано пять записей об изменении блока b с номерами $n-2$, $n-1$, n, $n+1$, $n+2$. В блоке b содержится номер n. Это означает, что в состоянии блока отражены результаты операций изменения блока, соответствующих журнальным записям LR(b)n, LR(b)n_{-1} и LR(b)n_{-2}. Изменения блока, произведенные

операциями, которым соответствуют две хронологически последние журнальные записи $\mathtt{LR(b)}n_{+1}$ и $\mathtt{LR(b)}n_{+2}$, в его состоянии во внешней памяти не отражены, поскольку не было выполнено выталкивание во внешнюю память страницы буферного пула, содержащей копию блока b. Поэтому при восстановлении состояния блока требуется выполнить обратные операции изменения блока b, соответствующие журнальным записям $\mathtt{LR(b)}n$, $\mathtt{LR(b)}n_{-1}$ и $\mathtt{LR(b)}n_{-2}$.



Нумерация записей об изменении блока

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 распространено поддержание **отдельного (короткого) журнала постраничных изменений**. Такой журнал обычно называют ***физическим* журналом,** поскольку он содержит записи об изменении физических объектов – блоков внешней памяти. В отличие от этого, журнал логических операций принято называть ***логическим* журналом,** поскольку в нем содержатся записи об операциях над **логическими объектами – кортежами.**

Как уже отмечалось, **логический и физический журналы имеют разную природу**. Во-первых, **логический** журнал должен поддерживать **как обратное выполнение журнализованных операций** (*undo*), **так и их повторное прямое выполнение** (*redo*). В отличие от этого, **от физического журнала требуется только поддержка**

обратного выполнения постраничных операций.

Во-вторых, **логический журнал** обычно **начинает заполняться заново только после выполнения операций резервного копирования базы данных или архивирования самого журнала**. **До этого времени он линейно растет**. Понятно, что в любом случае для размещения журнала выделяется внешняя память ограниченного размера. **Предельный размер журнала определяется администратором базы данных и должен согласовываться с размером интервала времени, через которое производится резервное копирование базы данных.**

Физический журнал существует сравнительно недолгое время (интервал времени между соседними операциями установки точки физической согласованности базы данных) и, как правило, **занимает существенно меньшее дисковое пространство, чем логический журнал**. При выполнении операции **установки точки физической согласованности** выполняются следующие действия:

- **прекращают инициироваться новые логические операции;**
- **после завершения всех выполняемых логических операций происходит выталкивание во внешнюю память всех модифицированных страниц буферного пула;**
- **формируется и выталкивается во внешнюю память логического журнала специальная запись о точке физически согласованного состояния;**
- **в случае успешного предыдущего действия разрешается инициация новых логических операций, и физический журнал пишется заново**.

Предпоследняя операция является атомарной (это опять же запись одного блока на диск): **если она успешно выполняется**, то **при следующем восстановлении после мягкого сбоя будет использоваться новая точка физически согласованного состояния**, иначе **ситуация воспринимается как мягкий сбой с восстановлением логически согласованного состояния базы данных от предыдущей точки физически согласованного состояния** (с оповещением об этом администратора базы данных).

49. Архивация базы данных и журнала. Восстановление базы данных после жесткого сбоя.

Потенциальное переполнение логического журнала регулируется следующим образом. **На пути к достижению максимально возможного размера журнала устанавливаются «желтая» и «красная» зоны.** Когда записи в журнал достигают **«желтой» зоны, выдается предупреждение администратору базы данных и прекращается образование новых транзакций. Если все существующие транзакции завершаются до достижения «красной» зоны**, **автоматически выполняется архивация базы данных или логического журнала. Если какие-то транзакции не успевают завершиться до достижения «красной» зоны журнала, выполняется их аварийный откат, после чего производится архивация базы данных или журнала.** Естественно, размер «желтой» и «красной» зон логического журнала должен устанавливаться администратором базы данных с учетом максимально допустимого

числа одновременно существующих транзакций и их возможной протяженности.

Понятно, что для восстановления последнего согласованного состояния базы данных после жесткого сбоя **журнала изменений базы данных явно недостаточно.** Самый простой способ восстановления основывается на использовании **логического журнала и архивной копии базы данных.**

Восстановление начинается с обратного копирования (на исправный носитель) 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.**

Очевидно, что после этого будет получено хронологически последнее до момента жесткого сбоя логически согласованное состояние базы данных.

Следует заметить, что при некоторой дисциплине выполнения логических операций над базой данных при восстановлении базы данных после жесткого сбоя **можно просто последовательно повторно выполнять операции в соответствии с журнальными записями, не обращая внимание на то, в каких транзакциях они выполнялись до жесткого сбоя.** В частности, **если сериализация транзакций основывается на блокировках объектов, то эта дисциплина заключается в том, что при выполнении операции в штатном режиме нужно сначала дождаться удовлетворения блокировки изменяемого объекта, затем поместить запись в буфер логического журнала, и только после этого реально выполнять операцию.**

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 незавершенных 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 здесь следует еще раз обсудить, касается производства архивных копий базы данных и/или журнала. Самый простой способ состоит в **архивировании базы данных по явному указанию администратора или при переполнении журнала**. Но **можно выполнять архивацию базы данных реже, чем переполняется журнал. Вместо базы данных можно архивировать сам журнал.** В пределе для полного восстановления базы данных после жесткого сбоя достаточно иметь **исходную архивную копию базы данных, последовательность архивных копий журналов и последний логический журнал.** Может показаться, что

восстановление базы данных на основе таких архивных источников будет занимать недопустимо большое время, однако здесь возможна значительная оптимизация.

Во-первых, **архивированный логический журнал можно сжимать.** Для этого **для каждого объекта базы данных нужно найти последовательность журнальных записей, относящихся к этому объекту, в хронологическом порядке и заменить их одной записью, соответствующей операции над объектом, результат которой эквивалентен результату последовательного выполнения журнализованных операций из построенной последовательности.** Например, на рис. 14.4 показан процесс сжатия последовательности журнальных записей, соответствующих последовательности операций над кортежем, у которого tid = *k* и имеются четыре целочисленных поля . Заметим , что если хронологически последней в последовательности является запись, соответствующая операции `DELETE`, то после сжатия этой последовательности она станет пустой.

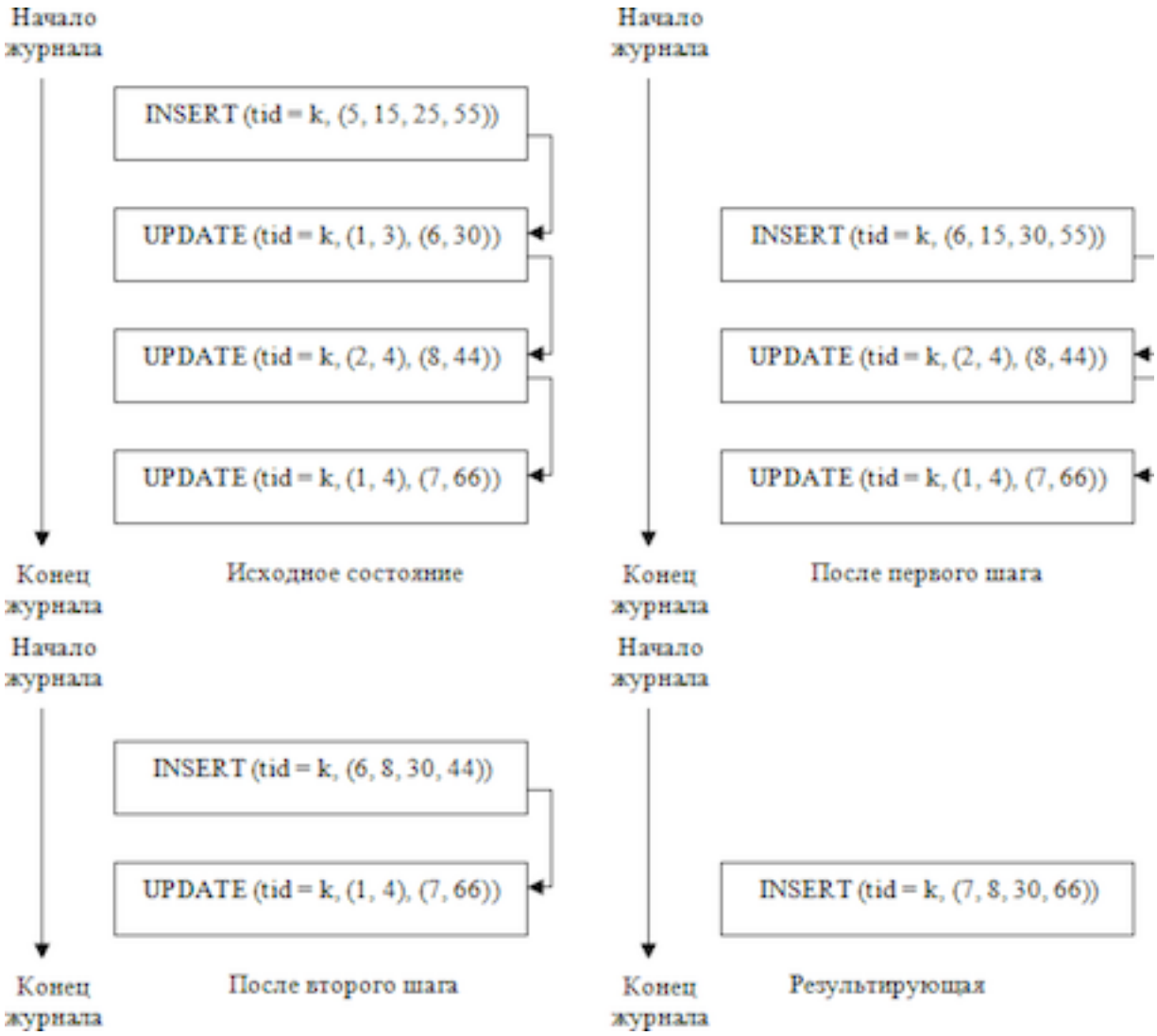


Рис. 14.4. Процесс сжатия последовательности журнальных записей

Во-вторых, точно таким же образом **можно совместно сжать два хронологически последовательных полных или сжатых журнала.** Таким образом, для восстановления

после жесткого сбоя **можно воспользоваться исходной архивной копией, одним сжатым архивным журналом и последним логическим журналом.** Снова могут возникнуть сомнения относительно сложности и продолжительности процесса сжатия журнала. Но здесь следует заметить, что **эта работа может выполняться на отдельном компьютере в режиме off-line. Кроме того, если имеется архивная копия базы данных, сжатый архивный журнал и набор еще не сжатых архивных журналов, то этого уже достаточно для восстановления, так что сроки завершения процесса полного сжатия не являются критическими.**