ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ МГУ ИМЕНИ М.В. ЛОМОНОСОВА

ФОНД ВОЛЬНОЕ ДЕЛО

teach-in
ЛЕКЦИИ УЧЕНЫХ МГУ

БАЗЫ ДАННЫХ

КУЗНЕЦОВ СЕРГЕЙ ДМИТРИЕВИЧ

ВМК МГУ

БЛАГОДАРИМ ЗА ПОДГОТОВКУ КОНСПЕКТА

СТУДЕНТА ФАКУЛЬТЕТА ВМК МГУ **БРАЖНИКОВА**

АЛЕКСЕЯ АЛЕКСЕЕВИЧА

Содержание

Об авторе

Автором данного курса является Кузнецов Сергей Дмитриевич. Он закончил кафедру математической логики механико-математического факультета МГУ в 1971 г. Является специалистом в области системного программирования и баз данных. С 1995 г. преподаёт в МГУ в должности профессора кафедры системного программирования факультета ВМК. Читает лекционные курсы по проблемам построения баз данных и систем управления базами данных (СУБД), по организации ОС UNIX. Дополнительные материалы по данному курсу могут быть найдены на citforum.ru/database.

Рис. 1: Кузнецов Сергей Дмитриевич

Лекция 1

Информационная система

Информационная система (ИС) - это программный комплекс, в функции которого входят:

- поддержка надежного хранения данных в памяти компьютера
- выполнение требуемых для данного приложения преобразований информации и вычислений
- предоставление пользователям удобного и легко осваиваемого интерфейса

Классическими примерами ИС являются банковские системы, системы резервирования авиационных или железнодорожных билетов. Информационные системы также называют приложениями баз данных (database application).

Следует очень аккуратно относиться к терминам «данные», «информация» и «знание». Данные хранятся в памяти компьютера, с ними работают программы. Операционной системе (ОС) известна только структура данных, а именно их формат, какие операции над ними можно выполнять. Смысл данных системе не известен. Информация появляется в результате анализа данных (человеком или программой). Знание - это понимание как соотносятся разные куски информации. Это правила или закономерности, которые сопутствуют информации. Таким образом, выстраивается иерархия: данные $<$ информация $<$ знание.

Устройства внешней памяти

Объёмы данных, с которыми приходится работать ИС достаточно велики, а сами данные обладают достаточно сложной структурой. Надёжное и долговременное хранение информации можно обеспечить только при наличии *энергонезависимых запоминающих устройств*, сохраняющих данные после выключения электропитания. Оперативная память, например, этим свойством не обладает.

Магнитные ленты

В первые десятилетия развития вычислительной техники, т.е. в конце 40-х, начале 50-х гг. для долговременного хранения данных использовались два вида устройств внешней памяти: магнитные ленты и магнитные барабаны. На рис. 2 представлены магнитные ленты компании IBM, которые представляют из себя ленточные магнитофоны с двумя бобинами, на которые намотана магнитная лента.

Есть лентопротяжное устройство, протягивающее ленту с одной бобины на другую. Также имеются магнитные головки, отвечающие за чтение и запись на нужное место магнитной ленты.

Рис. 2: IBM 726, первый аппарат записи данных на магнитную ленту, 1952г.

Магнитные ленты обеспечивают *последовательный доступ к данным*. Это означает, что при запросе у устройства n-ого блока, необходимо физически перемотать ленту с текущей позиции на начало n-го блока (как в видеокассетах). Т.е. от блока i нам надо пройти через $i+1$, $i+2$ ... $n-1$, n. В среднем требуется перемотать половину длины магнитной ленты. Свойства магнитных лент:

- большая емкость
- только последовательный доступ к данным
- операцию перемотки нельзя выполнять слишком быстро, т.к. ленты рвутся

Магнитные барабаны

Альтернативой магнитным лентам служат магнитные барабаны. Магнитный барабан представляет из себя массивный металлический цилиндр с намагниченной внешней поверхностью, и неподвижными магнитными головками, число которых равно числу дорожек на внешней поверхности.

Данные хранятся на дорожках внешней поверхности, а скорость доступа напрямую зависит от скорости вращения магнитного барабана. Например, на БЭСМ-6

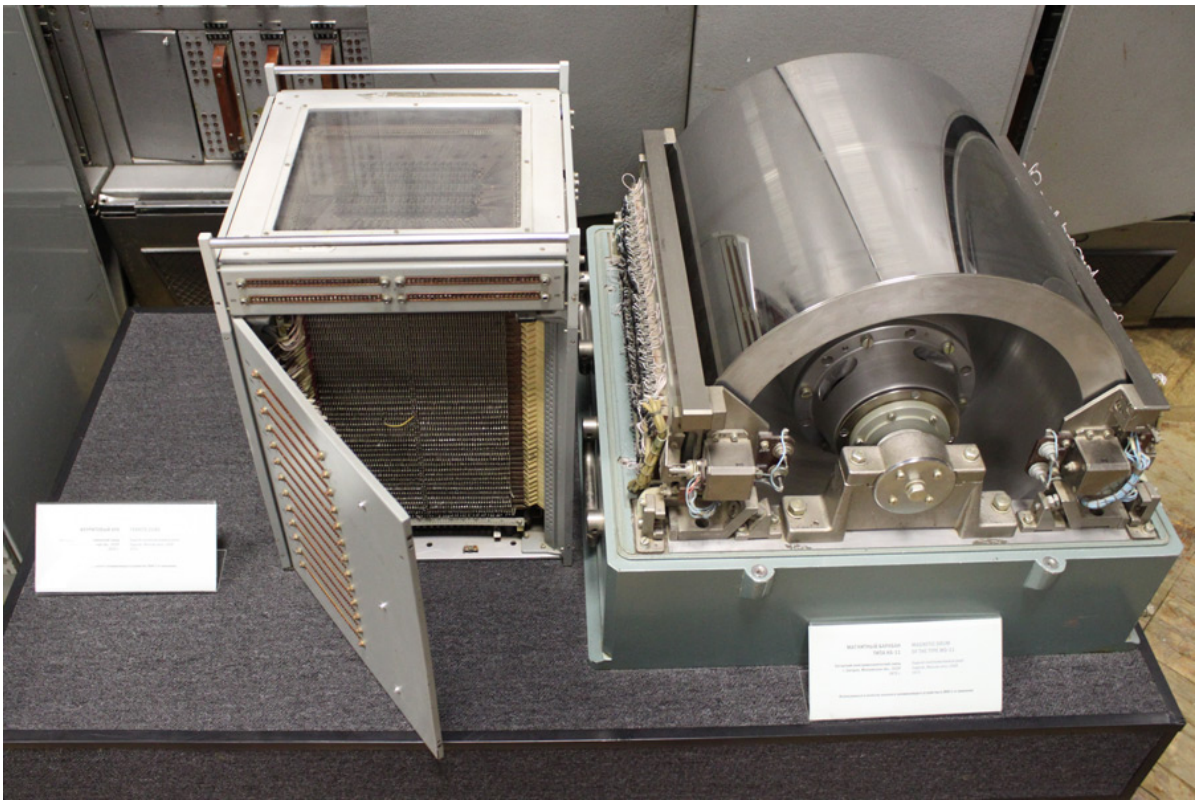


Рис. 3: Магнитный барабан в московском Политехническом музее

скорость доступа к внешней памяти была всего на один порядок меньше скорости доступа к оперативной памяти (ОП).

Магнитные барабаны обеспечивают *случайный доступ к данным*. Это означает, что заранее известно физическое расположение блока на поверхности барабана. Можно от текущего блока i сразу за счёт поворота на нужный угол встать в начало n-го блока. Т.е. не нужно последовательно проходить все промежуточные блоки между i и n. Таким образом заранее известно максимальное время доступа к случайному блоку. Свойства магнитных барабанов:

- случайный доступ к данным
- значительно быстрее чем магнитные ленты
- маленькая емкость
- тяжёлые (на БЭСМ-6 магнитные барабаны весили до 500кг)

Лекция 2

Требования к вычислительной системе

Рассмотрим какие требования предъявлялись к вычислительным системам. Как правило, серьезные вычислительные программы работают длительное время: недели, месяцы, годы. Поэтому требуется сохранять частичные результаты вычислений, чтобы при возникновении непредвиденных сбоев аппаратуры можно было продолжить выполнение расчётов с некоторой контрольной точки. Из-за последовательного доступа к данным для таких сохранений идеально подходят магнитные ленты:

- при сохранении данные последовательно сбрасываются на ленту
- при восстановлении системы после сбоя данные последовательно считываются

Второй традиционной потребностью численных программистов является максимально большой объём ОП. Она нужна, чтобы:

- обеспечить быстрый доступ к большому количеству обрабатываемых данных
- позволить выполнять сложные вычислительные программы большого объёма

Поскольку объём реально доступной ОП всегда являлся недостаточным для удовлетворения потребностей вычислений, требовалась быстрая внешняя память для организации оверлеев и/или виртуальной памяти. Из-за своей скорости для этого хорошо подходили магнитные барабаны.

Требования к информационной системе

Информационная система требует высокую среднюю скорость выполнения операций, при наличии больших объёмов данных. В этом случае только магнитных барабанов и магнитных лент оказывается недостаточно:

- магнитные барабаны не вместительны
- магнитные ленты не могут работать быстро из-за последовательной природы

Магнитные диски

Магнитный диск - это устройство внешней памяти с несколькими магнитными поверхностями и подвижными головками чтения/записи. Также предусматривалась возможность замены дискового пакета на устройстве.

Появление магнитных дисков стало революцией в истории вычислительной техники. Требования к внешней памяти со стороны бизнес-приложений были удовлетворены, т.к. эти устройства памяти:

- обладали существенно большей емкостью, чем магнитные барабаны

- обеспечивали удовлетворительную скорость доступа к данным в режиме случайного доступа
- позволяли иметь архив данных практически неограниченного объема за счёт возможности смены дискового пакета на устройстве

При выполнении обмена с диском аппаратура выполняет три основных действия:

- подвод головок к нужному цилиндру (время выполнения - $t_{\text{подвод}}$)
- поиск на дорожке нужного блока (время выполнения - $t_{\text{поиск}}$)
- обмен с этим блоком (время выполнения - $t_{\text{обмен}}$)

Механическое перемещение головок не может происходить слишком быстро, т.к. они начинают колебаться при резком останавливании, и необходимо ждать пока эти колебания затухнут. В среднем переместить головку нужно на половину радиуса поверхности.

Для поиска блока на дорожке пакета магнитных дисков в среднем прокручивается на половину длины внешней окружности. Скорость вращения диска может быть существенно больше скорости перемещения головок, но она ограничена характеристиками материала.

Для выполнения обмена нужно прокрутить пакет дисков всего лишь на угловое расстояние, соответствующее размеру блока.

В связи с этим, как правило, $t_{\text{подвод}} \geq t_{\text{поиск}} \geq t_{\text{обмен}}$. Причём подвод головок сильно длительнее времени поиска блока. Поэтому существенный выигрыш при считывании только части блока получить невозможно.

Системы управления файлами

До появление магнитных дисков каждая прикладная программа для хранения данных во внешней памяти использовала низкоуровневые программно-аппаратные средства. В связи с этим архитектура прикладных программ сильно усложнялась, а разработчики тратили время на реализацию базовых операций работы с файлами (в том числе их именование и способ организации во внешней памяти). И такая ситуация наблюдалась повсеместно: каждое приложение имело свои собственные реализации операций для работы с файлами во внешней памяти. А ведь вместо этого, разработчики могли бы больше времени уделять разработке целевых функций своих программ. Это спровоцировало появление *систем управления файлами* или *файловых систем (ФС)*.

Стоит отметить, что термин "файловая система" используется для обозначения как программной системы, управляющей файлами, так и совокупности файлов, хранящихся во внешней памяти. Было бы лучше в первом случае использовать термин «система управления файлами», оставив за термином «файловая система» только второе значение. Однако на практике его используют в обоих смыслах.

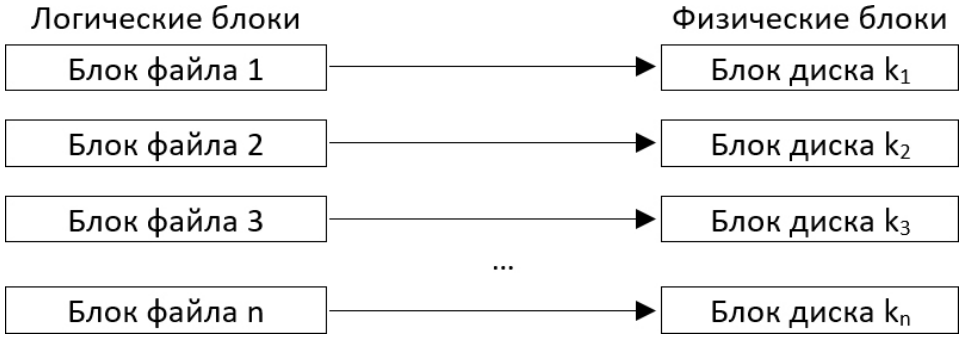


Рис. 4: Каждому логическому блоку файла соответствует физический блок диска

Первая развитая ФС была разработана специалистами IBM в середине 60-х для серии компьютеров System/360. В этой ФС поддерживались как чисто последовательные, так и индексно-последовательные файлы, а реализация во многом опиралась на возможности только появившихся к тому времени контроллеров управления дисковыми устройствами. Контроллеры обеспечивали возможность обмена с диском порциями данных произвольного размера, а также индексный доступ к файлам, т.е. поиск файла по значению ключа.

Структуры файлов

Во многих современных компьютерах основными устройствами внешней памяти являются магнитные диски с подвижными головками. Аппаратура магнитных дисков допускает выполнение обмена с дисками порциями данных произвольного размера (в том числе всего несколько байт). Однако возможность обмена порциями меньше одного блока в настоящее время не используется.

Это связано с тем, что считывание только части блока не приводит к существенному выигрышу времени обмена. Также для работы с частями блоков необходимо со стороны операционной системы (ОС) обеспечивать работу с буферами оперативной памяти произвольного размера. Но это бы приводило к внешней фрагментации памяти: в памяти существует много свободных фрагментов, размер которых в сумме больше любого требуемого буфера, но каждый отдельный свободный промежуток меньше требуемого. Для разрешения внешней фрагментации нужно выполнить дорогостоящую операцию сжатия памяти - сдвига всех фрагментов вместе, чтобы они располагались вплотную друг к другу.

Таким образом, с точки зрения современных ФС файл представляет собой набор последовательно нумеруемых логических блоков, которые отображаются на физические блоки диска (рис. 4). 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 ОС. Исторически существует два основных представления файлов.

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. Такое представление файла реализовано в операционной системе OpenVMS.

В некоторых ФС допускается структуризация *записей* на поля и объявление указываемых полей ключами *записи*. В таких ФС можно потребовать выборку конкретной *записи* из файла по заданному ключу. Такой механизм называется *индексацией*. Для реализации индексации в том же базовом файле хранятся метаданные - невидимые пользователю служебные структуры данных. Далее в курсе мы будем подробно рассматривать индексацию на основе хэширования и B-деревьев.

Файл как непрерывная последовательность байтов

Данное представление файла рассматривает файл как непрерывную последовательность байтов. Такой подход реализован в ОС UNIX. Из файла можно прочитать (или записать) указанное число байтов, предварительно выполнив позиционирование на нужный байт.

Стоит отметить, что хоть операции для работы с отдельными байтами файла в ФС UNIX и поддерживаются, но реальная организация файлов - это последовательность нумерованных блоков. Конечно, одно представление можно преобразовывать к другому. Например, OpenVMS поддерживает стандартную ФС UNIX.

Именование файлов

Современные ФС организованы в виде иерархии (дерева). В корне дерева находится корневой каталог, в узлах дерева - другие каталоги, в листьях дерева - файлы. Каждый каталог содержит имена файлов, хранящихся в данном каталоге, а также имена других каталогов. Полным именем файла является список имен каталогов и имени файла в конечном каталоге.

Такая система именования позволяет уникально идентифицировать каждый файл в ФС, и позволяет не опасаться коллизий с именами других файлов или каталогов.

В современных ФС требуется, чтобы полное дерево каталогов целиком располагалось на одном логическом диске – разделе физического дискового пакета, логически представляемом в виде отдельного диска. В этом случае полное имя файла начинается с имени дискового устройства, на котором установлен соответствующий диск. Этот способ именования использовался в ФС компании IBM, к этому также близки ФС, реализованные в ОС семейства Windows. Можно назвать такую организацию поддержкой *изолированных файловых систем*.

Другой вариант был реализован в ФС ОС Multics. В Multics пользователям представлялась вся 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. Система сама выполняет поиск файла по его имени и запрашивает у оператора установку необходимых дисков. Такую ФС можно назвать *полностью централизованной*, потому как все файлы, имеющиеся на компьютере связаны в одну иерархию.

Во многом централизованные ФС удобнее изолированных, так как система управления файлами берёт на себя больше функций:

- автоматически оповещает о потребности установки нужных дисковых пакетов
- обеспечивает равномерное распределение памяти на всех известных дисковых носителях
- облегчает резервное копирование
- автоматически перемещает редко используемые файлы на более медленные носители внешней памяти

Однако в централизованных ФС возникают проблемы, например, при переносе на другую вычислительную установку поддерева ФС. Поскольку файлы и каталоги любого логического поддерева могут быть физически разбросаны по разным дисковым устройствам, для такого переноса требуется специальная утилита, собирающая все объекты требуемого поддерева на одном внешнем носителе. Даже при наличии такой утилиты выполнение процедуры физической сборки требует существенного времени.

Авторизация доступа к файлам

Поскольку ФС является общим хранилищем файлов, принадлежащих разным пользователям, системы управления файлами должны обеспечивать авторизацию доступа к файлам. В общем случае используется мандатный способ защиты. Подход состоит в том, что для каждого зарегистрированного пользователя и для каждого существующего файла ФС указываются действия, которые над данным файлом данному пользователю разрешены или запрещены. Для его применения необходимо для каждого файла дополнительно хранить access control list - список пользователей, которым разрешено с ним что-то делать. Это влечет за собой существенные накладные расходы из-за потребности хранения избыточной информации и долгому поиску пользователю по access control list для проверки правомощности доступа.

В современных системах управления используется дискреционный подход к контролю доступа, который в большинстве случаев удовлетворителен. С каждым зарегистрированным пользователем связывается пара целочисленных идентификаторов:

- идентификатор группы пользователя
- собственный идентификатор пользователя

Полный идентификатор пользователя - это пара `<собственный идентификатор, идентификатор группы>`. Полный идентификатор присваивается каждому процессу. При каждом файле также хранится полный идентификатор пользователя, который создал этот файл, и помечается:

- какие действия с файлом может производить он сам
- какие действия доступны для остальных пользователей той же группы
- что могут делать с файлом пользователи других групп

Для каждого файла контролируется возможность выполнения трех действий: чтение, запись и выполнение. Таким образом, что 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

Синхронизация многопользовательского доступа

Если ОС поддерживает многопользовательский режим, может возникнуть ситуация, когда два процесса одновременно пытаются работать с одним и тем же файлом. Если им нужно 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.

Обычно в файловой системе используется следующий подход. При открытии файла среди прочих параметров указывается режим работы (чтение или изменение). Если к этому моменту 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.

Области разумного применения файлов

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. Эти редакторы могут быть очень простыми или многофункциональными, но обычно структура текстовых файлов очень проста (с точки зрения ФС): это либо последовательность записей, содержащих строки текста, либо последовательность байтов, среди которых встречаются специальные символы (например, символы конца строки). Сложность логической структуры текстового файла определяется текстовым редактором, но в любом случае файловая система её не знает.

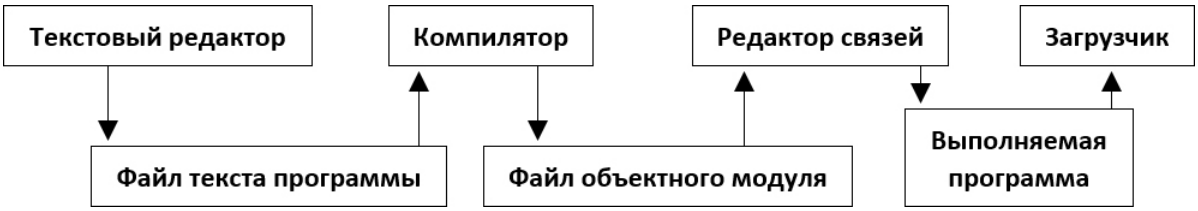


Рис. 5: Логическая структура файлов на примере системы программирования

В качестве примера рассмотрим компиляцию программы (рис. 5). 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 объектных модулей. С точки зрения файловой системы файлы объектных модулей это просто последовательность записей или байтов. С точки зрения системы программирования файлы объектных модулей имеют сложную и специфичную логическую структуру.

Здесь важным моментом является то, что для файловой системы структура файла неизвестна. Ей без разницы находится в файле текст или объектный модуль. Задача интерпретации содержимого лежит на плечах прикладных программ. Получается, что для работы с файлами логическую структуру файлов должны знать только прикладные программы, но не файловая система. ФС обеспечивает хранение слабоструктурированной информации, оставляя дальнейшую структуризацию прикладным программам. При разработке прикладной системы это даже хорошо: с помощью стандартных и сравнительно быстрых средств ФС, можно реализовать те структуры хранения, которые наиболее точно соответствуют специфике данной прикладной области.

Системы управления базами данных (СУБД)

Аналогично тому как вычислительные системы индивидуально решали задачу управления файлами (стр. 15), информационные системы боролись с похожей проблемой.

Типичная ИС ориентирована на хранение, выбор и модификацию данных соответствующей прикладной области. Структура таких данных зачастую сложна, но несмотря на то, что в разных прикладных областях разные структуры данных, между ними часто бывает много общего.

До появления систем управления базами данных, проблемы структуризации данных решались индивидуально для каждой ИС (рис. 6). Производились необходимые надстройки над файловой системой. Эти надстройки являлись существенной частью ИС и практически повторялись от одной системы к другой.

Стремление выделить общую часть ИС, ответственную за управление сложно структурированными данными, явилось первой побудительной причиной создания

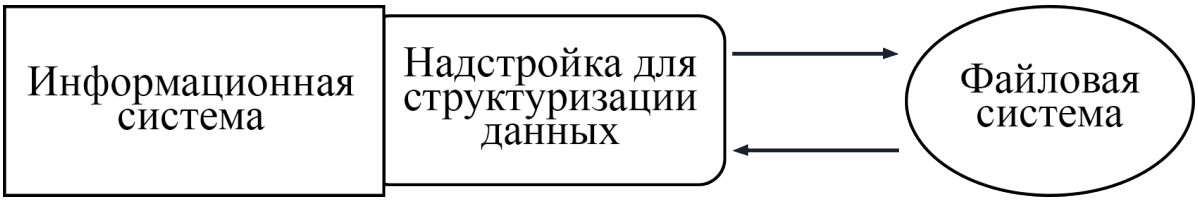


Рис. 6: Работа с ФС через дополнительную надстройку над ИС

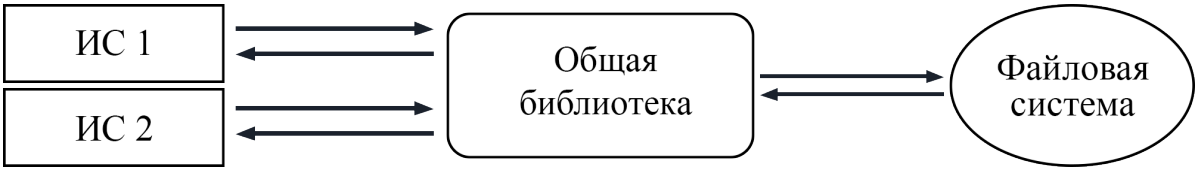


Рис. 7: Функции работы с ФС вынесены в общую библиотеку

СУБД. Возникла необходимость создания общей библиотеки программ (рис. 7), реализующей над базовой ФС более сложные методы хранения данных.

Пример информационной системы

Пусть требуется реализовать ИС, поддерживающую учет служащих некоторой организации. Должна поддерживаться следующие действия:

- выдавать списки служащих по отделам
- поддерживать возможность перевода служащего из одного отдела в другой
- обеспечивать средства поддержки приема на работу новых служащих и увольнения работающих служащих

Кроме того, для каждого отдела должна поддерживаться возможность получения:

- имени руководителя отдела
- общей численности отдела
- общей суммы зарплаты служащих отдела, среднего размера зарплаты

Для каждого служащего должна поддерживаться возможность получения:

- номера удостоверения по полному имени служащего (для простоты допустим, что имена всех служащих различны),
- полного имени по номеру удостоверения.

Пусть мы решили основывать эту ИС на файловой системе и пользоваться одним файлом `СЛУЖАЩИЕ`, расширив базовые возможности ФС за счет специальной библиотеки функций. Поскольку минимальной информационной единицей в нашем случае является служащий, в этом файле должна содержаться одна запись для каждого служащего. Чтобы можно было удовлетворить указанные выше требования, запись о служащем должна иметь следующие поля:

- `СЛУ_ИМЯ` - полное имя служащего (уникальное)
- `СЛУ_НОМЕР` - номер удостоверения (уникальный)
- `СЛУ_СТАТ` - данные о соответствии служащего занимаемой должности (для простоты «да» или «нет»)
- `СЛУ_ЗАРП` - размер зарплаты
- `СЛУ_ОТД_НОМЕР` - номер отдела
- `СЛУ_ОТД_РУК` - номер руководителя отдела

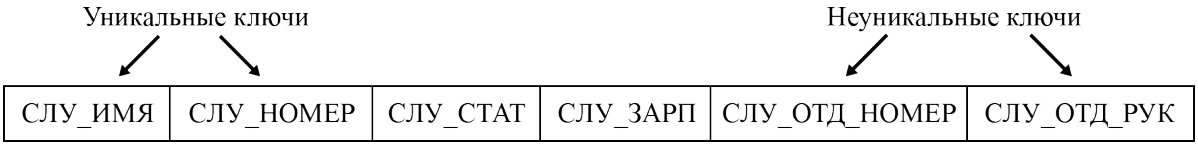


Рис. 8: Пример информационной системы

Стоит отметить, что поле `СЛУ_ИМЯ` уникальное, т.е. во всём файле не должно найтись двух служащих с одинаковым именем. А значит по имени можно однозначно идентифицировать служащего и найти значения всех остальных полей, соответствующих данному служащему. Аналогичные рассуждения справедливы для поля `СЛУ_НОМЕР`.

Уникальное поле называются *ключом* (*возможным ключом* или *уникальным ключом*). В данном случае `СЛУ_ИМЯ` и `СЛУ_НОМЕР` являются возможными ключами файла `СЛУЖАЩИЕ`. Возможные ключи очень важны, т.к. на их основе можно построить индексные структуры, которые будут существенно увеличивать скорость операций для работы с записями. Позднее в курсе механизм индексации будет подробно рассмотрен.

На рис. 8 видна структура записи служащего. Такая структура называется заголовком. Все записи служащих должны соответствовать данному заголовку.

Интересно в этом заголовке то, что поля `СЛУ_ОТД_НОМЕР` и `СЛУ_ОТД_РУК` повторяются в каждой записи служащего. Сколько служащих в отделе - столько и повторов. Это проблема хранения избыточных данных. Стоит отметить, что упомянутые поля являются неуникальными ключами, смысл чего будет понятен далее.

Как отмечалось ранее, с помощью возможных ключей можно существенно ускорить операции для работы с записями. Кроме того, хотелось бы эффективно выбрать всех служащих по номеру отдела (или по номеру руководителя отдела). Т.е. наша надстройка должна уметь это делать.

Также хочется быстро уметь считать численность отдела, считать размер средней зарплаты всего отдела, и т.д. Такие групповые функции называются *агрегатными*. Эти функции также должны поддерживаться в надстройке.

Потребности в оптимизации хранения данных и увеличения скорости операций возникают в каждой ИС! Которые, в свою очередь, решают их собственными надстройками.

Рис. 9: Схема информационной системы из двух файлов

Для улучшения ситуации можно добавить ещё один файл `ОТДЕЛЫ` (рис. 9). Введение двух файлов позволяет преодолеть большинство перечисленных неудобств. Например, теперь информация о руководителе отдела хранится только один раз. Но заметим, что это ещё более усложняет логику нашей надстройки.

Лекция 3

Таким образом информационная система должна «знать», что она работает с двумя информационно связанными файлами, должна иметь информацию о структуре и смысле каждого поля. Например, должно быть известно, что у полей СЛУ_ОТД_НОМЕР в файле СЛУЖАЩИЕ и ОТД_НОМЕР в файле ОТДЕЛЫ один и тот же смысл номера отдела.

Кроме того, ИС 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 изменить поля ОТД_СЛУ_ЗАРП и ОТД_РАЗМЕР в соответствующем отделе из файла ОТДЕЛЫ. Т.е. всегда должны выполняться следующие условия:

1) если СЛУЖАЩИЕ содержит запись, в которой СЛУ_ОТД_НОМЕР = n, то ОТДЕЛЫ содержит запись, в которой ОТД_НОМЕР = n
2) если ОТДЕЛЫ содержит запись, в которой ОТД_РУК = m, то СЛУЖАЩИЕ содержит запись, в которой СЛУ_НОМЕР = m
3) значение поля ОТД_СЛУ_ЗАРП любой записи отдел файла ОТДЕЛЫ равно сумме значений поля СЛУ_ЗАРП всех записей файла СЛУЖАЩИЕ, в которых СЛУ_ОТД_НОМЕР = ОТД_НОМЕР
4) значение поля ОТД_РАЗМЕР любой записи отдел файла ОТДЕЛЫ равно числу всех записей файла СЛУЖАЩИЕ, в которых СЛУ_ОТД_НОМЕР = ОТД_НОМЕР

Другими словами:

1) если есть служащий в отделе с номером n, то существует отдел с номером n
2) руководитель отдела является служащим
3) значение поля суммарной зарплаты отдела соответствует той, что получится при суммировании зарплат каждого служащего из данного отдела
4) значение поля размера отдела равно тому числу, что получится при подсчёте числа служащих из данного отдела

Правила 1) и 2) называются *ограничениями ссылочной целостности*. Поле СЛУ_ОТД_НОМЕР из файла СЛУЖАЩИЕ «ссылается» на поле ОТД_НОМЕР файла ОТДЕЛЫ. Поле ОТД_РУК из файла ОТДЕЛЫ «ссылается» на поле СЛУ_НОМЕР файла СЛУЖАЩИЕ.

Ранее было сказано, что поле СЛУ_ОТД_РУК является неуникальным ключом. Это означает, что БД предполагает наличие индекса по этому полю, который позволяет делать быструю выборку служащих по руководителю отдела. Однако после разбиения на два файла это имеет и другой смысл. Мы знаем, что поле СЛУ_НОМЕР является уникальным ключом, а поле СЛУ_ОТД_РУК «ссылается» на него из файла отделов. В таком случае поле СЛУ_ОТД_РУК называется внешним ключом, для которого СУБД автоматически создаёт индексную структуру. Как правило, внешние ключи являются неуникальными ключами. Например, в данном случае это

так, т.к. может быть много записей с одним и тем же руководителем. Аналогичные рассуждения верны и для поля `ОТД_РУК`.

Правила 3) и 4) называются *общими ограничениями целостности*. Общими называются потому, что такие ограничения могут быть очень общего характера. Такие ограничения наиболее сложны для проверки в вычислительном плане и, с этой точки зрения, нежелательны. Одной из задач при проектировании базы данных является уменьшение числа ограничений общего вида.

Целостность данных

Понятие целостности (или согласованности), данных является ключевым понятием баз данных (БД). База данных находится в целостном состоянии, если выполнены все ограничения целостности данной БД.

Как раз из-за необходимости в организации специальных индексных структур, необходимости поддерживать БД в целостном состоянии для любой ИС возникает необходимость в системе, которая возьмёт эти функции на себя. Такую систему называют Системой управления базами данных (СУБД).

Требование поддержки согласованности БД нельзя выполнить с помощью общей библиотеки функций, т.к. система должна обладать служебными данными (метаданными) относительно устройства конкретной БД. В нашем примере ИС должна отдельно сохранять метаданные о структуре файлов `СЛУЖАЩИЕ` и `ОТДЕЛЫ`, а также правила, определяющие условия целостности данных в этих файлах (правила также составляют часть метаданных).

Говорят, что база данных состоит из двух частей:

- экзистенциональной, содержащей реальные данные пользователей
- интенсиональной, содержащей метаданные

Языки запросов

Обеспечение целостности данных не единственное требование к СУБД. Нам необходимо сообщить базе данных какие именно данные нам нужны. Т.е. надо математически определить операции для работы с базой данных. Далее в курсе мы рассмотрим алгебру Кодда и Алгебру А, с помощью которых можно работать с базой данных. Однако запросы к базе данных с помощью этих алгебр не интуитивны, и пользователю-нематематику сложно будет работать. Поэтому важно определить интуитивно понятный *язык запросов к базам данных*.

Обычно говорят, что алгебраическая формулировка является процедурной, т.е. задающей последовательность действий для выполнения запроса, а логическая – описательной (или декларативной), поскольку она описывает свойства желаемого результата. Эти два механизма эквивалентны.

Рассмотрим поближе язык запросов SQL на примере. Допустим, мы хотим узнать общую численность отдела, в котором работает Петр Иванович Сидоров. Сначала необходимо узнать номер отдела, в котором работает указанный служащий, затем установить численность этого отдела.

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

Данный запрос называется запросом с полусоединением: условие в запросе адресуется к двум файлам – `СЛУЖАЩИЕ` и `ОТДЕЛЫ`, но данные выбираются только из файла `ОТДЕЛЫ`. Условие `СЛУ_ОТД_НОМЕР = ОТД_НОМЕР` ограничивает набор интересующих нас записей об отделах до одной записи. Если же Петр Иванович Сидоров не работает ни в одном отделе, то запрос выдаст пустой результат.

```
SELECT ОТД_РАЗМЕР FROM ОТДЕЛЫ
WHERE ОТД_НОМЕР = ( SELECT СЛУ_ОТД_НОМЕР FROM СЛУЖАЩИЕ
                    WHERE СЛУ_ИМЯ = "ПЕТР ИВАНОВИЧ СИДОРОВ" )
```

Во втором случае внутри вложенного подзапроса выбирается значение поля `СЛУ_ОТД_НОМЕР` из записи файла `СЛУЖАЩИЕ`, в которой `СЛУ_ИМЯ="ПЕТР ИВАНОВИЧ СИДОРОВ"`. Если такая запись существует, то она единственная, поскольку поле `СЛУ_ИМЯ` является уникальным ключом файла `СЛУЖАЩИЕ`. Тогда результатом выполнения подзапроса будет единственное значение – номер отдела, в котором работает Петр Иванович Сидоров. Во внешнем запросе это значение будет ключом доступа к файлу `ОТДЕЛЫ`, и снова будет выбрана только одна запись, поскольку поле `ОТД_НОМЕР` является уникальным ключом файла `ОТДЕЛЫ`. Если же на данном предприятии Петр Иванович Сидоров не работает, то подзапрос выдаст пустой результат, следовательно и внешний запрос выдаст пустой результат.

Приведенные примеры показывают, что при использовании SQL можно не задумываться о том, как будет выполняться этот запрос. Среди метаданных базы данных будет содержаться информация о том, что поле `СЛУ_ИМЯ` является ключевым для файла `СЛУЖАЩИЕ`, а поле `ОТД_НОМЕР` – для файла `ОТДЕЛЫ`. Формально доказывается, что обе формулировки эквивалентны, т.е. на любых данных всегда приводят к одному и тому же результату.

Транзакции, журнализация и многопользовательский режим

Представим себе, что в первоначальной реализации ИС, основанной на использовании библиотек расширенных методов доступа к файлам, обрабатывается операция принятия на работу нового служащего. Следуя требованиям согласованного изменения файлов, ИС вставляет новую запись в файл `СЛУЖАЩИЕ` и собирается модифицировать соответствующую запись файла `ОТДЕЛЫ`, но именно в этот момент происходит аварийное выключение питания компьютера.

После перезапуска системы, БД будет находиться в несогласованном состоянии, т.к. точно будут нарушены правила 3) и 4). Потребуется выяснить это и привести данные в согласованное состояние.

Проверку и коррекцию можно выполнить, например, следующим образом. Сгруппировать записи файла `СЛУЖАЩИЕ` по `СЛУ_ОТД_НОМЕР`. Для каждой группы:

- проверить, существует ли в `ОТДЕЛЫ` запись, в которой `ОТД_НОМЕР = СЛУ_ОТД_НОМЕР`
- если такой записи нет, то исключить всю группу из `СЛУЖАЩИЕ` и перейти к обработке следующей группы
- иначе посчитать число записей в группе и вычислить суммарную заработную плату, обновить поля `ОТД_РАЗМЕР` и `ОТД_СЛУ_ЗАРП` в соответствующем отделе и перейти к обработке следующей группы

Настоящие СУБД берут такую работу на себя. Прикладная система не обязана заботиться о поддержке целостности БД, хотя должна знать, какие цепочки операций изменения данных являются допустимыми. Например, при обновлении файла `СЛУЖАЩИЕ`, нужно обновлять значения некоторой записи в `ОТДЕЛЫ`.

Пусть в нашей ИС требуется обеспечить параллельную работу с базой данных нескольким пользователям. Нужен механизм, который организует работу так, чтобы ни один из пользователей не заметил, что помимо него кто-то ещё работает с тем же файлом. На самом деле, мы узнаем далее, что такой механизм существует и называется механизмом *транзакций*.

СУБД как независимый системный компонент

Мы убедились, что информационная система нуждается в дополнительном функционале помимо базовых возможностей файловой системы. И более того такой функционал нужен всем информационным системам. Поэтому все эти функции были вынесены в отдельную систему, которая называется *системой управления базами данных (СУБД)*.

До сих пор мы не вычленяли СУБД из состава ИС, имея в виду общую организацию системы, подобную показанной на рис. 10.

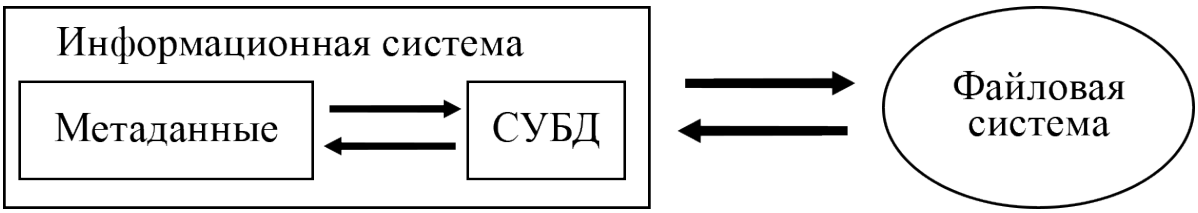


Рис. 10: СУБД неразделима от ИС

Здесь видны два дефекта. Во-первых, СУБД должна поддерживать достаточно развитую функциональность. Повторять эту функциональность в каждой ИС неразумно. С другой стороны, неясно, как обеспечить готовый к использованию компонент СУБД, который можно было бы встраивать в ИС. Во-вторых, набор файлов можно назвать базой данных только при наличии метаданных. На рис. 10 метаданные являются частью ИС, и поэтому файлы `СЛУЖАЩИЕ` и `ОТДЕЛЫ` можно эффективно использовать только в конкретно данной ИС регистрации служащих.

Предположим, предприятию нужна еще и бухгалтерская ИС. Для её работы также потребуются данные о служащих и отделах. При показанной выше организации системы возможны два варианта выполнения задачи, и ни один из них не является удовлетворительным.

1) Можно внедрить бухгалтерскую систему в состав системы регистрации служащих. Но, как правило, бухгалтерские системы покупаются в виде готовых и отдельных продуктов, не приспособленных к подобному внедрению.

2) Можно скопировать метаданные системы регистрации служащих в бухгалтерскую систему. Однако метаданные (как и данные) не обязательно являются статичными. Структура БД может со временем изменяться, могут исчезать одни правила целостности и появляться другие. Как согласовывать копии метаданных между независимыми ИС?

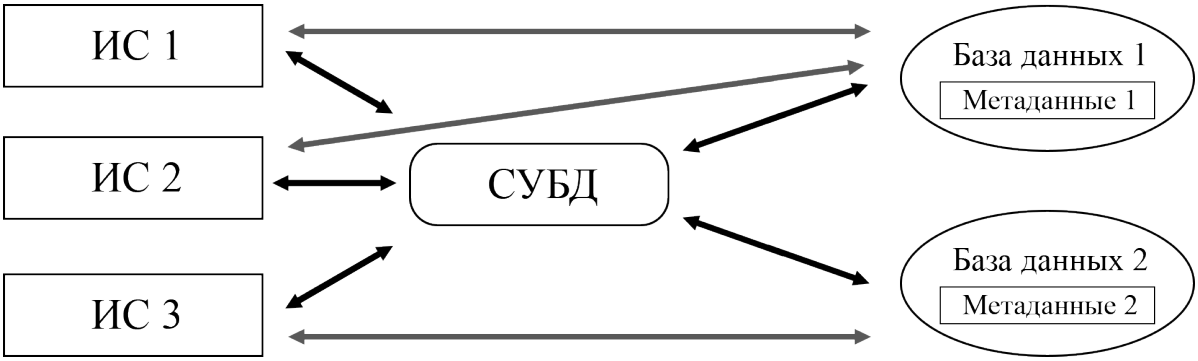


Рис. 11: СУБД отделена от информационной системы

Такими рассуждениями мы приходим к организации системы, показанной на рис. 11. Здесь изображены три информационные системы, которые через единую СУБД работают с двумя разными БД, причем первая и вторая системы работают с общей базой данных. Это возможно, поскольку метаданные каждой базы данных содержатся в самих БД. Поскольку СУБД функционирует отдельно от приложений, и её работа с БД регулируется метаданными, совместное использование одной БД двумя ИС не вызовет потери согласованности данных, и доступ к данным будет должным образом синхронизироваться.

Обратим внимание, схема на рис. 11 приближает нас к клиент-серверной архитектуре: СУБД играет роль сервера, который обслуживает нескольких клиентов - информационных систем.

Заключение

Развитие аппаратных и программных средств управления внешней памятью диктовалось потребностями ИС, для построения которых требовалась возможность надежного долговременного хранения больших объемов данных, а также обеспечение достаточно быстрого доступа к этим данным.

Системы управления файлами во внешней памяти обеспечивают минимальные потребности ИС, предоставляя средства распределения и структуризации дисковой памяти, именования файлов, авторизации доступа и поддержки многопользовательского режима. Были показаны ситуации, в которых возможностей ФС не хватает.

Перечисленные функции ИС необходимы в каждой ИС, а самостоятельная реализация функций СУБД - это очень трудная задача, которая не должна решаться каждой информационной системой. При выборе технологии построения ИС нужно тщательно оценивать и прогнозировать её потенциальные потребности в средствах управления данными. Конечно, любую ИС можно основывать на использовании промышленной, большой и мощной СУБД. Но в действительности может оказаться так, что приложение будет использовать малую часть общих возможностей СУБД. Накладные расходы (стоимость аппаратуры, лицензирование дорогостоящего ПО) могут оказаться неоправданными.

СУБД решают множество проблем, которые затруднительно или вообще невозможно решить при использовании ФС. При этом существуют маленькие приложения, которым вполне достаточно файлов.

Основные функции и компоненты СУБД

Мы выявили несколько потребностей ИС, для решения которых базовых возможностей файловой системы недостаточно:

- поддержка целостности файлов
- восстановление согласованного состояния данных после сбоев
- обеспечение параллельной работы нескольких пользователей
- поддержка языка манипулирования данными

Эти функции традиционно поддерживаются СУБД.

Непосредственное управление данными во внешней памяти

Управление данными подразумевает организацию необходимых структур во внешней памяти:

- для хранения данных и матаданных, входящих в БД
- для служебных целей (для быстрого доступа к данным с помощью индексов)

В некоторых реализациях СУБД активно используются возможности существующих ФС, в других работа производится напрямую с устройствами внешней памяти. Но подчеркнем, что в развитых СУБД пользователи в любом случае не обязаны знать как организованы файлы. В частности, в СУБД обычно поддерживается собственная система именования объектов БД.

Управление буферами оперативной памяти

Управление буферами ОП необходимо для увеличения скорости работы с данными. СУБД обычно работают с БД значительного размера (для транзакционных систем в пределах 1 петабайта, для оптических систем возможны более 1 эксабайта).

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 ОП.

Обычно размеры БД больше объёма ОП, поэтому надо определить какую именно часть буферизовать. Даже если ОС производит общесистемную буферизацию данных, то СУБД намного лучше знает какую именно часть БД лучше буферизовать. Поэтому в развитых СУБД поддерживается собственный набор буферов ОП с использованием собственной дисциплиной замены буферов.

Управление транзакциями

Управление транзакциями нужно для обеспечения целостности данных и обеспечения многопользовательского доступа. Транзакция - это последовательность операций над БД, рассматриваемых СУБД как единое целое: либо все операции внутри транзакции выполняются, либо ни одна не выполняется. Поддержание механизма транзакций является обязательным в многопользовательских СУБД.

Рассмотри подробнее как транзакции поддерживают целостность БД. Вспомним наш пример ИС. Если мы хотим принять на работу нового служащего, то нам надо обязательно добавить запись в файл служащих и обновить численность его отдела. Таким образом нам надо добавить запись и изменить поле другой записи. Эти две операции нужно выполнить в одной транзакции. Тогда в случае, если после добавления нового служащего, мы ещё не успели обновить численность, и происходит сбой системы, можно без нарушения целостности БД, удалить нового служащего и написать сообщение об ошибке. Т.е. не будет ситуации, когда применена только часть изменений, которая нарушает целостность БД.

Таким образом важным свойством транзакции является сохранение целостности БД, т.к. если транзакция исполнена, то она гарантирует целостность данных в БД, а если не исполнена, то она гарантирует, что данные в БД не изменятся.

Однако сохранение целостности БД не единственное важное свойство транзакций. При правильном управлении параллельными транзакциями, каждый пользователь может ощущать себя единственным пользователем СУБД. С этим связано важное понятие сериализации транзакций.

Сериализация транзакций - это такой порядок выполнения транзакций, при котором суммарный эффект набора параллельных транзакций эквивалентен эффекту их некоторого последовательного выполнения. Другими словами, в многопользовательской СУБД при одновременной работе нескольких пользователей надо определить в

каком порядке выполнять текущие транзакции так, чтобы результат их выполнения был таким, будто он выполняются не параллельно, а последовательно. Более подробно об этом будет изложено далее в курсе.

Если удается добиться действительно сериального выполнения набора транзакций, то для каждого пользователя, по инициативе которого образована транзакция, присутствие других транзакций будет незаметно (если не считать некоторого замедления работы по сравнению с однопользовательским режимом).

Существует несколько базовых алгоритмов сериализации транзакций. 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, произведённые 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.

Журнализация

Журнализация необходима для восстановления БД в случае сбоев. Одним из основных требований к СУБД является надежность хранения данных во внешней памяти. СУБД должна быть в состоянии восстановить последнее согласованное состояние БД после любого аппаратного или программного сбоя. Существует два вида аппаратных сбоев:

- мягкие сбои или внезапная остановка работы компьютера (например, аварийное выключение питания)
- жёсткие сбои, характеризуемые потерей информации на носителях внешней памяти

Примерами программных сбоев могут быть:

- аварийное завершение работы СУБД (по причине ошибки в программе или в результате некоторого аппаратного сбоя)
- аварийное завершение пользовательской программы, в результате чего некоторая транзакция остается незавершенной

В любом случае для восстановления БД нужно располагать некоторой дополнительной информацией. Другими словами, 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.*

Журнал - это особая часть БД, недоступная пользователям СУБД и поддерживаемая с особой тщательностью, в которую поступают записи обо всех изменениях

основной части БД. Записи в журнале могут соответствовать, например, некоторой логической операции изменения БД (удаления, вставки, изменения).

Принято использовать стратегию «упреждающей» записи в журнал, которая заключается в том, что запись об изменении любого объекта БД должна попасть в журнал раньше, чем изменённый объект попадет в основную часть БД. Такая стратегия используется в протоколе Write Ahead Log (WAL). Известно, что если в СУБД корректно соблюдается протокол WAL, то с помощью журнала можно решить все проблемы восстановления БД после любого сбоя.

Поддержка языков БД

С помощью языков БД осуществляется работа с БД. В ранних СУБД поддерживалось несколько по своим функциям языков БД. Чаще всего выделялись два:

- SDL (Schema Definition Language) - язык определения схемы БД
- DML (Data Manipulation Language) - язык манипулирования данными

SDL использовался для определения логической структуры БД, т.е. структуры БД, какой она представляется пользователям. DML содержал набор операторов манипулирования данными (вставки, удаления, изменения, выборки данных) для работы с БД.

В современных СУБД поддерживается единый интегрированный язык, который содержит все необходимые средства для работы с БД, начиная от её создания, и обеспечивающий базовый пользовательский интерфейс с БД. Для реляционных СУБД таким языком является SQL.

- SQL сочетает средства SDL и DML, т.е. позволяет определять схему реляционной БД и манипулировать данными
- именование объектов БД (для реляционной БД таблицы и столбцы) 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

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. Поддержка представлений БД производится также на языковом уровне.

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

Лекция 4

Типовая организация современной СУБД

Организация типичной СУБД и состав её компонентов соответствует рассмотренному выше набору функций. Мы выделили следующие основные функции СУБД:

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

Логически в современной реляционной СУБД можно выделить:

- наиболее внутреннюю часть – ядро СУБД (database 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 ОП, управление транзакциями и журнализацию. Соответственно, можно выделить следующие компоненты ядра:

- менеджер данных
- менеджер буферов
- менеджер транзакций
- менеджер журнала

Функции этих компонентов сильно 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 и утилитах БД. Ядро СУБД является основной резидентной частью СУБД. 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 (как правило, SQL) не процедурные,

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. Таким образом, компилятор должен решить как именно исполнить оператор до начала выполнения программы.

Применяются достаточно сложные методы оптимизации операторов. Результатом компиляции является выполняемая программа, представляемая в некоторых системах в машинных кодах. Но более часто (в машинно-независимых СУБД) представляется в выполняемом внутреннем машинно-независимом коде. В этом случае реальное выполнение оператора производится интерпретатором этого внутреннего языка.

В отдельные утилиты БД выделяют такие процедуры, которые требуют особого внимания:

- загрузка и выгрузка БД
- сбор статистики
- глобальная проверка целостности БД

Утилиты программируются с использованием интерфейса ядра СУБД, а иногда даже с проникновением внутрь ядра.

История СУБД

История СУБД началась в 1960-е гг., когда были созданы первые навигационные СУБД, основанные на иерархической, сетевой и других ранних моделях данных. В этих СУБД отсутствовали декларативные языки БД (доступ к данным явно специфицировался разработчиками ИС), имелись очень ограниченные возможности поддержки целостности БД. Наиболее известными СУБД этого времени, используемыми и сейчас, являются иерархическая СУБД IMS (IBM) и сетевая СУБД IDMS (Computer Associates).

Реляционный подход

Разработчики ранних СУБД работали с БД на низком уровне, что делало зависимыми от этой СУБД все приложения, которые с ней работают. В связи с этим, в конце 1960-х - начале 1970-х гг. был преложен *реляционный подход* к организации БД и СУБД Эдгаром Коддом (Edgar F. Codd). Новый подход предложил обобщенную структуру данных:

- которая легко отображается на реальную структуру БД во внешней памяти
- с которой можно формулировать и выполнять запросы в декларативном (описательном) стиле

Для проверки практической жизнеспособности этого подхода в 1970-е гг. были выполнены экспериментальные проекты *System R* в IBM и *Ingres* в Калифорний-

ском университете в Беркли. В этих проектах были заложены основы технологий, которые использовались в будущих коммерческих реляционных СУБД: принципы оптимизации запросов, методы управления транзакциями, появился язык SQL.

1980-е

1980-е гг. знаменательны появлением первых коммерческих реляционных СУБД (IBM DB2, Oracle и т.д.). В течение этого десятилетия за счет совершенствования технологии удалось добиться эффективности реляционных СУБД, которая не уступала эффективности ранних СУБД, в то же время обеспечивала более развитую среду разработки ИС. К концу 1980-х SQL-ориентированные системы стали главенствовать на рынке СУБД.

Одновременно с этим появились объектно-ориентированные СУБД (ОО-СУБД) (O2, ObjectStore, Versant, ...). Это направление было направленно на преодоление разрыва (impedance mismatch) между системами типов данных традиционных языков программирования и системами типов, поддерживаемыми в БД реляционными СУБД.

Проблема заключалась в том, что реляционный подход предполагает работу с БД в теоретико-множественных понятиях, а программы пишут, основываясь на атомарные модели данных (типы множества и массивов стоят на вторых ролях). Такой разрыв мешал программистам работать в едином стиле. В действительности использовался подход полнотиповых языков программирования, при котором разработчик может описывать свой собственный тип данных любой структуры, а также операции над этим типом. Также возникло направление *объектно-реляционных СУБД* (Illustra), которое преследовало те же цели, что и ОО-СУБД, но без отказа от реляционной парадигмы.

1990-е

1990-е знаменательны появлением ряда универсальных SQL-ориентированных СУБД с объектно-реляционными возможностями (Informix Universal Server, Oracle8, DB2 Universal Database). В СУБД появились возможности определения пользовательских типов данных, методов, функций и процедур. В конце десятилетия был принят стандарт SQL:1999, в котором эти возможности, а также все основные средства SQL-ориентированных СУБД были окончательно согласованы и утверждены.

В это же время продолжали развиваться ОО-СУБД, и был принят ряд стандартов ОО-модели данных, но к концу десятилетия объектно-реляционные СУБД в основном вытеснили ОО-СУБД. Появились достаточно конкурентоспособные СУБД с открытыми исходными кодами (MySQL, PostgreSQL и т.д.).

В начале 21-го века Майкл Стоунбрейкер (Michael Stonebraker) провозгласил конец эпохи универсальных СУБД и переход к набору технологий специализирован-

ных СУБД, ориентированных на поддержку отдельных классов приложений баз данных (информационных систем). Появились специализированные:

- XML-ориентированные СУБД (Tamino, Sedna)
- потоковые СУБД (Streambase)
- аналитические СУБД (Vertica, Greenplum)
- транзакционные СУБД (VoltDB)

В начале 2010-х начала появляться проблема BigData - проблема роста количества данных (прежде всего в поисковых системах, естественнонаучных исследованиях). Такой рост сделал актуальной технологию горизонтально масштабируемых СУБД. Поэтому, практически все современные СУБД являются массивно-параллельными. Та же проблема BigData привела к появлению ряда систем категории NoSQL, разработчики которых опираются на опыт области распределенных систем и не следуют канонам традиционных СУБД.

Отечественные разработчики входили и входят в международные сообщества, разрабатывающие СУБД с открытыми исходными кодами. В СССР и России в разное время имелось несколько оригинальных разработок СУБД:

- ИНЕС и НИКА (1980-е, Институт Системного Анализа РАН)
- ЛИНТЕР (компания РЕЛЭКС, Воронеж, начиная с 1990-х)
- Sedna (2000-е, ИСП РАН)

Лекция 5

Классификация СУБД

Классификация по модели данных

Прежде всего, СУБД различаются по модели данных, которая определяет архитектуру, структуры данных, методы работы с данными этой СУБД. Бывают:

- сетевые
- иерархические
- реляционные (и SQL-ориентированные)
- объектно-ориентированные
- XML-ориентированные и другие

Универсальные и специализированные СУБД

Существуют *универсальные СУБД* (Oracle, DB2, Microsoft SQL Server), которые направлены на многофункциональность, на удовлетворение всех потребностей ИС. Из-за своей всенаправленности, универсальные СУБД очень громоздкие и неповоротливые, часто они не успевают быстро адаптироваться к новым потребностям бизнеса, из-за чего бизнес теряет деньги. Считается, что именно бизнес является главной движущей силой развития СУБД (и IT продуктов в целом).

Альтернативой универсальным приходятся *специализированные СУБД* (Vertica, VoltDB и т.д.), ориентированные на эффективную поддержку одного класса ИС (например, транзакционных или аналитических систем). За счёт своей узконаправленности, они могут быстрее адаптироваться к требованиям бизнеса.

Файл-серверные, клиент-серверные и встраиваемые СУБД

В файл-серверных СУБД (Informix SE, Microsoft Access и т.д.) БД хранится на специализированном файл-сервере, а СУБД запускается на каждом клиенте. Как правило такая архитектура используется внутри локальной сети. Так как клиенты ничего друг про друга не знают, то синхронизация многопользовательского доступа реализуется на файл-сервере. Понятно, что многопользовательская синхронизация в этом случае будет на уровне ФС, что влечёт блокировки файлов. Более того, сервер только хранит данные, и не занимается их обработкой, следовательно ответ от такого файл-сервера будет в виде блока данных для каждого запроса. Из-за этого ощущается нагрузка на сеть при большом числе клиентов. Такая организация довольно старая и пригодна только при небольшом количестве клиентов (до нескольких десятков).

В клиент-серверных СУБД (Oracle, DB2, PostgreSQL и т.д.) все основные компоненты СУБД выполняются на отдельном сервере БД, и на нём же хранится БД. На клиенте находится интерфейсная (клиентская) часть СУБД и выполняется код приложения. Плюсом такой архитектуры является оптимизации на стороне клиента: всю работу с БД осуществляет сервер, по сети передаётся только обработанный ответ небольшого размера. Недостатком такой архитектуры является зависимость клиентов от сервера БД. В случае неисправности которого, ни один клиент не сможет работать с БД.

Также существенным недостатком клиент-серверных СУБД является необходимость установления прямого соединения между клиентским компьютером и сервером БД. Существует трёхзвенная клиент-серверная архитектура, которая добавляет дополнительный вспомогательный сервер приложений - прослойку между клиентами и сервером БД, который выполняет код приложения. Это позволяет полностью изолировать клиентов от конкретной БД и вся логика работы с БД ложится на плечи вспомогательного сервера. Трёхзвенная архитектура помимо повышения уровня безопасности трёхзвенная архитектура позволяет более гибко модернизировать приложения. При модернизации не нужно закачивать обновления в клиенты. Достаточно обновить только сервер приложений.

Стоит отметить, что клиент-серверная архитектура подразумевает работу с БД на слабых клиентских устройствах, но, в действительности, из года в год производительность компьютерной техники только увеличивается. Получается, что сеть загружена меньше, но вычислительные возможности клиентов не реализуют своих возможностей полностью.

Встраиваемые СУБД (BerkeleyDB, SQlite и т.д.) встраиваются в код приложений (информационные системы; клиенты) и полностью выполняется на том же компьютере и даже в том же процессе. Как правило это библиотека, которая подключается в код программы и позволяет использовать функции СУБД прямо внутри программы. Главная проблема здесь в незащищенности БД от клиента. Встраиваемые СУБД успешно применяются в Интернет- и мобильных приложениях.

Классификация по месту хранения БД

СУБД могут хранить данные во внешней памяти, и СУБД, сохраняющие данные в ОП (in-memory).

Исторически подавляющее большинство СУБД хранит БД во внешней памяти. В этом случае хранилище внешней памяти доступно СУБД через системные вызовы операционной системы и структурировано в виде блоков. Нужный блок из внешней памяти буферизуется в ОП, где с блоком происходит дальнейшая работа.

In-memory СУБД располагают всю БД в ОП и использует внешнюю память для обеспечения долговременности транзакций. За счёт расположения БД в ОП достигается очень большая скорость работы с БД (на 4 порядка выше по сравнению

с магнитными дисками). Существует несколько подходов к организации in-memory СУБД:

- Внешняя память вообще не используется, а надёжность достигается за счёт хранения реплик БД в разных узлах кластерной системы
- БД хранится целиком в ОП, а журнал изменений во внешней памяти
- Самым популярным вариант, при котором БД хранится целиком и в ОП, и во внешней памяти, однако операция чтения происходит из ОП, а запись - в обе

Стоит добавить, что появление SSD накопителей качественно изменило ситуацию, и теперь разница в скорости доступа между внешней памятью на SSD и ОП составляет всего 2 порядка. Работы по увеличению скорости внешней памяти активно ведутся и сегодня, поэтому, может быть, со временем устройства внешней памяти сравняются по скорости с ОП (или ОП станет энергонезависимой). Это бы принесло существенное увеличение производительности, сделав СУБД одноуровневой (только ОП, или только внешняя память) и сохранив надежность и долговременность хранения данных.

Классификация по типу параллельности

По типу параллельности СУБД бывают:

- однопроцессорные
- параллельные с общей памятью (shared-everything)
- параллельные с общими дисками (shared-disks)
- параллельные без использования общих ресурсов (shared-nothing)

Однопроцессорные СУБД не используют аппаратные возможности параллелизма и выполняются на одном процессоре (в частности, на одном одноядерном процессоре). До появления многоядерных процессоров такие СУБД были распространены, поскольку многопроцессорные компьютеры с общей памятью были слишком дороги и малодоступны.

Параллельные СУБД с общей памятью (Oracle, DB2) поддерживались ведущими компаниями с 1980-х гг., но стали массово распространенными только после появления многоядерных процессоров. Параллельные СУБД этого класса обеспечивают межзапросный (inter-query) или внутризапросный (intra-query) параллелизм с использованием нескольких ядер (или аппаратных потоков). Наличие общей памяти позволяет избежать пересылок данных, но требует применения сложной синхронизации.

Один из подходов, направленный на упрощение архитектуры такой СУБД, и уменьшением затрат на работу с общей памятью был реализован компанией Oracle. Допустим, имеется N ядер. Одно ядро объявляется управляющим, и оно отвечает за компиляцию запросов, а также планирует выполнение уже скомпилированных запросов на других ядрах. Все оставшиеся ядра работают обработчиками запросов.

Такой подход обеспечивает практически линейный прирост производительности при росте числа ядер. Однако здесь остаётся тонким вопрос синхронизации данных, требующий отдельного внимания.

В будущем параллельные СУБД получат качественный прирост в производительности за счёт энергонезависимой ОП и возникновения процессоров с очень большим числом ядер.

Параллельные СУБД с общими дисками (Oracle Real Application Cluster) работают на кластерах и имеют общую дисковую подсистему, в которой хранится БД. Т.е. все узлы кластера имеют собственную ОП, но общие диски внешней памяти. Эта архитектура сильно похожа на файл-серверную архитектуру, однако общие диски - это не файловые хранилища, а отдельные аппаратно-программные решения, которые работают с узлами не на уровне блоков, а выполняют части запросов. Таким образом обеспечивается межзапросный или внутризапросный параллелизм.

Параллельные СУБД без использования общих ресурсов (Greenplum, Vertica) работают на кластерах, но каждый узел со своим разделом БД. Для этого при компиляции запросов каждый запрос разбивается на части, адресуемые к соответствующему узлу кластера. Результат запроса получается объединением частичных результатов с узлов. На сегодняшний день считается, что именно shared-nothing архитектура обеспечивает наилучшее горизонтальное масштабирование при росте объема данных (для решения проблемы BigData в области аналитики).

Но в такой архитектуре возникает очень важная проблема разбиения БД по узлам кластера. Как разделить узлы, чтобы в среднем любой запрос выполнялся наиболее быстро?

Так как при обработке части запроса на конкретном узле, всё равно требуется обращение к другим узлам, то необходимо такое разбиение кластера, чтобы число таких связей в кластере было минимальным.

Известно, что задача такого разбиения NP-полная, т.е. решения, кроме полного перебора всех вариантов, не существует. Конечно, существуют эвристические подходы к организации shared-nothing систем, которые работают на конкретных запросах и конкретных данных, но никто не гарантирует надёжности стабильной работы системы при других данных и запросах.

Допустим, что задача разбиения N узлов по кластеру решена. Однако параллельные СУБД предполагают горизонтальное масштабирование. Другими словами, при добавлении нового N+1-го узла нужно получить новое разбиение кластера и перераспределить данные между узлами (согласно новому разбиению). Эта процедура перераспределения очень вычислительно сложная, и связанная с этим недоступность БД во время перераспределения просто недопустима для реального бизнеса.

Лекция 6

Модель данных

Модель данных - это набор родовых понятий и признаков, которыми должны обладать все конкретные СУБД и управляемые ими БД, если они основываются на этой модели. С помощью модели данных представляются структура объектов и отношения между ними. Они позволяют сравнивать конкретные реализации СУБД между собой, а также помогают при проектировании архитектуры СУБД. Понятие модели данных было введено Эдгаром Коддом, однако наиболее распространенная реляционная модель принадлежит Кристоферу Дейту. Модель данных состоит из *структурной*, *манипуляционной* и *целостной* частей.

Структурная часть модели данных содержит основные логические структуры данных, которые могут применяться на уровне пользователя при организации БД. Например, в модели данных SQL основным видом структур БД являются таблицы, а в объектной модели данных – объекты ранее определенных типов.

Манипуляционная часть содержит спецификацию одного или нескольких языков, предназначенных для написания запросов к БД. Эти языки могут быть:

- абстрактными, без точно проработанного синтаксиса (это свойственно языкам реляционной алгебры и реляционного исчисления, используемым в реляционной модели данных)
- или законченными производственными языками (например, SQL)

Основное назначение манипуляционной части модели данных – это обеспечить эталонный «модельный» язык БД, который должен поддерживаться в реализациях СУБД, соответствующих данной модели.

Целостная часть модели данных (которая не всегда явно выделяется) специфицирует механизмы ограничений целостности, которые обязательно должны поддерживаться во всех реализациях СУБД, соответствующих данной модели. Например, в целостной части реляционной модели данных требуется поддержка ограничения первичного ключа в любой переменной отношения, а в модели данных SQL такого требования по отношению к таблицам нет (в таблице могут быть строки, которые полностью совпадают).

Ранние модели данных

К числу ранних моделей данных относятся: иерархическая, сетевая и модель данных инвертированных таблиц.

СУБД, основанные на ранних моделях данных использовались до появления работоспособных реляционных СУБД. Такие СУБД накопили большие БД, и одной

из актуальных задач является использование этих БД совместно с современными СУБД.

Все ранние СУБД не основывались на каких-либо абстрактных моделях данных. Понятие модели данных возникло позже на основе анализа и выявления общих признаков у различных конкретных систем. Ранние СУБД были навигационными, т.е. доступ к БД производился на уровне записей (как в файловых системах). Пользователи осуществляли явную навигацию в БД, используя языки программирования, расширенные функциями СУБД. Интерактивный режим доступа к БД осуществлялся на уровне приложений, т.е. требовалось создание собственных интерфейсов.

Навигационная природа ранних СУБД 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 интерфейсами SQL. Однако в большинстве случаев это не сделало их по-настоящему реляционными системами, поскольку оставалась возможность манипулировать данными в естественном для них режиме (т.е. работать с данными так, как SQL не позволяет).

Модель данных инвертированных таблиц

К числу наиболее известных и типичных представителей систем, в основе которых лежит эта модель данных, относятся:

- СУБД Datacom/DB, разработанная в конце 1960-х гг. компанией Applied Data Research, Inc. (ADR) и принадлежащая в настоящее время компании Computer Associates
- Adabas (Adaptable Database System), которая была разработана компанией Software AG в 1971 г. и до сих пор является её основным продуктом

Структурная часть. Организация доступа к данным на основе инвертированных таблиц (индексов) используется практически во всех современных реляционных (SQL-ориентированных) СУБД, но в этих системах пользователи не имеют непосредственного доступа к инвертированным таблицам (индексам).

БД в модели инвертированных таблиц, как и в модели SQL - это набор таблиц. Однако в модели инвертированных таблиц пользователям видны и хранимые таблицы, и индексы. Строки таблиц упорядочиваются системой в некотором порядке, видимом пользователям. Для каждой таблицы можно определить произвольное число ключей поиска, для которых строятся индексы. Эти индексы автоматически поддерживаются системой (когда выполняются операции вставки строки в таблицу, удаление строки из таблицы и т.д.), но явно видны пользователям.

Манипуляционная часть. Поддерживаются два класса операций:

- Операции позиционирования (или поиска), которые позволяют по нужным критериям позиционировать на некоторые записи в таблице. Они образуют два подкласса:
  - прямые поисковые операции (например, установить адрес первой записи с конкретным значением ключа),
  - операторы, устанавливающие адрес записи относительно текущей позиции.
- Операции выборки/изменения над адресуемыми записями (после операции позиционирования).

Типичные операции позиционирования (возвращается адрес записи):

- `LOCATE FIRST` – найти первую запись таблицы T в физическом порядке
- `LOCATE FIRST WITH SEARCH KEY EQUAL` – найти первую запись таблицы T с заданным значением ключа поиска k
- `LOCATE NEXT` – найти первую запись, следующую за записью с заданным адресом в заданном пути доступа
- `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
- `LOCATE FIRST WITH SEARCH KEY GREATER` – найти первую запись таблицы T в порядке ключа поиска k со значением ключевого поля, большим заданного значения k

Типичные операции выборки/изменения:

- `RETRIVE` – выбрать запись с указанным адресом
- `UPDATE` – обновить запись с указанным адресом
- `DELETE` – удалить запись с указанным адресом
- `STORE` – включить запись в указанную таблицу (возвращает адрес записи)

Целостная часть. Общие правила определения целостности БД отсутствуют. Поддержка целостности в данном случае - это обязанность приложения. Если с БД работает только одна СУБД, то все ограничения целостности хранятся в нём, с этим можно жить. А если их больше? Тогда ограничения целостности уже могут быть разными (например, работа с разными частями БД), и их нужно учитывать, что трудно для разработчиков.

В некоторых системах поддерживаются ограничения уникальности значений некоторых столбцов таблиц (например, можно сказать что столбец `СЛУ_НОМЕР` в таблице `СЛУЖАЩИЕ` уникальный), но в основном вся поддержка целостности данных возлагается на прикладную программу.

Рис. 12: Схема иерархической БД

Иерархическая модель данных

Типичным представителем является СУБД IMS (Information Management System) компании IBM. Первая версия системы появилась в 1968 году.

Иерархическая БД состоит из упорядоченного набора деревьев, более точно - нескольких экземпляров (instances) одного типа дерева. Тип дерева состоит из одного «корневого» типа записи и упорядоченного набора типов поддеревьев, каждое из которых является некоторым типом дерева. Тип дерева представляет собой иерархически организованный набор типа записей.

Здесь везде употребляется слово «тип», так как речь идёт не о конкретных данных, а о модели данных. Т.е. именно типы, без конкретных значений полей.

На рис. 12 изображен пример типа дерева (иерархической БД). Тип записи `ОТДЕЛ` является предком для типов записей `РУКОВОДИТЕЛЬ` и `СЛУЖАЩИЕ`, а `РУКОВОДИТЕЛЬ` и `СЛУЖАЩИЕ` – потомки типа записи `ОТДЕЛ`. `РУК_ОТДЕЛ` типа записи `РУКОВОДИТЕЛЬ` содержит номер отдела, в котором работает этот руководитель (не обязательно в том же отделе, которым руководит). Между типами записи поддерживаются типы связей

Рис. 13: Экземпляр типа дерева (уже с данными), изображенного на рис. 12

(реальные связи появляются в экземплярах типа дерева). Такое типовое описание называется *схемой БД*.

Все потомки (одного типа) с общим предком называются близнецами. В нашем случае, все служащие одного отдела будут близнецами, так как имеют общего предка. Для иерархической БД определяется полный порядок обхода дерева сверху-вниз, слева-направо.

Примеры операций манипулирования иерархически организованными данными. Вся БД - это набор экземпляров типов деревьев (например, каждый отдел - это экземпляр типа записи `ОТДЕЛ`). Поэтому, можно:

- найти указанный экземпляр типа дерева БД (например, отдел 310)
- перейти от одного экземпляра типа дерева к другому
- 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 (сверху-вниз, слева-направо)
- вставить новую запись в указанную позицию
- удалить текущую запись

В иерархической модели данных автоматически поддерживается целостность ссылок между предками и потомками. Основное правило: никакой потомок не может существовать без своего родителя (потомок без родителя не может по определению быть в схеме иерархической БД). Вследствие этого получается, что при удалении Отдела, все служащие внутри этого отдела так же будут удалены.

Однако поддержка целостности между записями без связи «предок-потомок», не обеспечивается. Примером такой «внешней» ссылки является поле `РУК_НОМЕР` в экземпляре типа записи `РУКОВОДИТЕЛЬ`. Т.е., если на рис. 13 удалить Иванова из списка служащих, то запись типа `РУКОВОДИТЕЛЬ` от этого никак не пострадает.

Сетевая модель данных

Типичным представителем систем, основанных на сетевой модели данных, является СУБД IDMS (Integrated Database Management System), разработанная компанией Cullinet Software, Inc. и изначально ориентированная на использования на мейнфреймах компании IBM. В настоящее время IDMS принадлежит компании Computer Associates.

Архитектура системы основана на предложениях Data Base Task Group (DBTG) организации CODASYL (Conference on Data Systems Languages), которая отвечала за стандартизацию языка программирования COBOL. Отчет DBTG был опубликован в 1971 г., и вскоре после этого появилось несколько систем, поддерживающих архитектуру CODASYL, среди которых присутствовала и СУБД IDMS.

Сетевой подход к организации данных является расширением иерархического:

- в иерархической модели потомок должен иметь в точности одного предка
- в сетевой у потомка может иметься любое число предков (в том числе 0)

Сетевая БД состоит из набора записей и набора связей между этими записями. Тип записи определяется аналогично тому, как структура (struct) в языке Си. Тип связи всегда бинарный (связывается два типа записи). Экземпляр типа связи содержит в себе экземпляр типа предка и упорядоченный набор экземпляров типа потомка (их может быть много).

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 (предок может быть только в одном экземпляре связи)
- каждый экземпляр типа записи C является потомком не более чем в одном экземпляре типа связи L (если потомок имеет предка, то он единственный)

Типы связи формируются достаточно свободным образом:

- потомок в одной связи *L1* может быть предком в другой связи *L2*
- запись P может быть предком в любом числе связей (различных типов) (например, человек может быть и начальником отдела, и главой семьи, и т.д.)
- запись C может быть потомком в любом числе связей (различных типов)
- может существовать любое число типов связи с одним и тем же типом записи предка и одним и тем же типом записи потомка
  - Например, имеется два типа записи - *человек* и *университет*. Тогда тип связи может быть и студент-университет, и преподаватель-университет.
- 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-потомком в одной связи и потомком-предком - в другой
  - Например, муж предок жены дома, но жена начальник мужа на работе.
- предок и потомок могут быть одного типа записи (предыдущий пример с мужем и женой; они оба люди)

На рис. 14 приведён пример схемы сетевой базы данных с типами записи `ОТДЕЛ`, `СЛУЖАЩИЕ` и `РУКОВОДИТЕЛЬ` и типами связи «состоит из служащих», «имеет руководителя» и «является служащим».

- В типе связи «состоит из служащих» типом-предком является `ОТДЕЛ`, а типом-потомком – `СЛУЖАЩИЕ`. Экземпляр этого типа связи связывает экземпляр типа

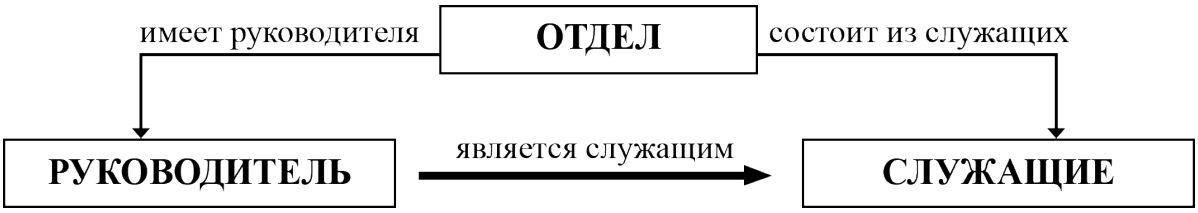


Рис. 14: Пример схемы сетевой БД

записи `ОТДЕЛ` со многими экземплярами типа записи `СЛУЖАЩИЕ`, соответствующими всем служащим данного отдела.

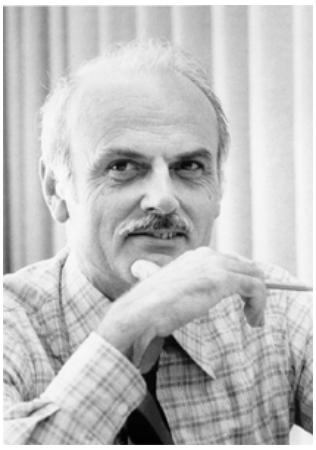
- В типе связи «имеет руководителя» типом-предком является `ОТДЕЛ`, а типом-потомком – `РУКОВОДИТЕЛЬ`. Экземпляр этого типа связи связывает экземпляр типа записи `ОТДЕЛ` с одним экземпляром типа записи `РУКОВОДИТЕЛЬ`, соответствующим руководителю данного отдела.
- В типе связи «является служащим» типом-предком является `РУКОВОДИТЕЛЬ`, а типом-потомком – `СЛУЖАЩИЕ`. Экземпляр этого типа связи связывает экземпляр типа записи `РУКОВОДИТЕЛЬ` с одним экземпляром типа записи `СЛУЖАЩИЕ`, соответствующим тому служащему, которым является данный руководитель.

Набор операций манипулирования данными в сетевой модели данных:

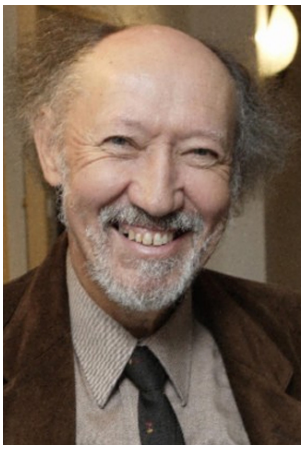
- найти конкретную запись в наборе однотипных записей (например, служащего с именем Иванов)
- перейти от предка к первому потомку по некоторой связи (например, к первому служащему отдела 625)
- перейти к следующему потомку в некоторой связи (например, от Иванова к Сидорову)
- перейти от потомка к предку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

Что касается ограничений целостности, то в сетевой модели данных имеется возможность потребовать для конкретного типа связи отсутствие потомков (как и в иерархической модели). Например, новый служащий ещё не закреплён за каким-то отделом, и первое время сам по себе.

Реляционная модель данных



(a) Эдгар Кодд



(b) Кристофер Дейт

Рис. 15

Основные идеи реляционной модели данных были изложены Эдгаром Коддом (рис. 15a) в 1969 г. в своей работе:

- E. F. Codd. Derivability, Redundancy and Consistency of Relations Stored in Large Data Banks. Заново опубликовано в ACM SIGMOD Record, March 2009 (Vol. 38, No. 1)
- Имеется русский перевод: Э.Ф. Кодд. Выводимость, избыточность и согласованность отношений, хранимых в крупных банках данных (ссылка)

Несмотря на общепризнанную значимость работ Кодда, они:

- писались на идейном уровне
- не были глубоко технически проработанными
- во многих важных местах допускали неоднозначное толкование

Поэтому эти работы невозможно было использовать как непосредственное руководство для реализации СУБД, поддерживающей реляционную модель. За прошедшие десятилетия реляционная модель развивалась в двух направлениях.

Первое направление заложил экспериментальный проект System R компании IBM. В этом проекте возник язык SQL, изначально основанный на идеях Кодда, но нарушавший некоторые принципиальные предписания реляционной модели. К настоящему времени язык SQL, по сути, образует отдельную, законченную (не реляционную) модель данных.

Второе направление, начиная с 1990-х гг., возглавлял Кристофер Дейт (рис. 15b), к которому позже примкнул Хью Дарвен. Оба этих ученых также работали в компании IBM и до 1990-х гг. внесли большой вклад в развитие языка SQL.

Однако в 1990-е гг. Дейт и Дарвен пришли к выводу, что искажения реляционной модели данных, свойственные языку SQL, слишком сильные, и пришло время предложить альтернативу, опирающуюся на неискаженные идеи Эдгара Кодда и обеспечивающую все возможности как SQL, так и объектно-ориентированного подхода к организации БД и СУБД.

Лекция 7

Новые идеи Дейта и Дарвена впервые изложены в их Третьем манифесте. Позже на основе этих идей была специфицирована отдельная модель данных. Авторы считают, что они приводят всего лишь современную и полную интерпретацию идей Кодда. С этим можно соглашаться или спорить, но бесспорен один факт – Кодд не участвовал в написании этих материалов. Далее при обсуждении реляционной модели мы будем использовать, в основном, интерпретацию Дейта и Дарвена.

Навигационная природа ранних баз данных (стр. 42) заставляла разработчиков писать слишком много деталей реализации, не относящихся к логике. Т.е. независимо от языка программирования СУБД содержит множество низкоуровневых конструкций (уровня языка ассемблера) для общения с БД. Это плохо тем, что от разработчиков требуется высокая квалификация, и они тратят время на детали низкоуровневой реализации, вместо того, чтобы сфокусироваться на логической части СУБД.

Самое главное, работающие с БД программы начинают зависеть от реализации БД. Например, если поменять что-нибудь в структуре БД, то все работающие с этой БД приложения могут пострадать. Получается, что программы сильно зависят от структуры данных. Такой недостаток свойственен всем ранним СУБД.

Реляционные структуры данных

Далее речь пойдёт о математическом описании реляционной модели, поэтому напомним разницу между понятиями множества и набора.

Множество элементов - это *неупорядоченная* совокупность элементов без повторений, обозначается $A = \{a_1, a_2, ..., a_n\}$, где a_i - элемент множества.

Набор элементов - это *упорядоченная* совокупность элементов, обозначается $A =< a_1, a_2, .., a_n >$.

Идея Кодда состояла в создании такой структуры данных, которая была бы понятна большинству разработчиков БД (не математику и не специалисту в области БД), и в то же время несла чёткий математический смысл.

В качестве такой структуры Эдгар Кодд взял неупорядоченную таблицу (без порядка строк и столбцов). Такая таблица состоит из столбцов и строк, в строках содержатся сами хранимые данные, а столбцы описывают структуру таблицы. Такая таблица со множеством столбцов $\{A_1, A_2, \ldots, A_n\}$ (A_i - имя столбца), в котором каждый столбец A_i содержит значения из конечного множества $T_i = \{v_{i1}, v_{i2}, \ldots, v_{im}\}$, в математическом смысле представляет собой отношение над множествами $\{T_1, T_2, \ldots, T_n\}$.

В математике отношением над множествами $T_1, T_2, \ldots, T_n$ называется подмножество декартова произведения этих множеств: $R = \{\{v_1, v_2, \ldots, v_n\}\} \subseteq T_1 \times ... \times T_n$, где $v_i \in T_i$. Такую таблицу Кодд назвал *отношением* (relation), запись в таблице -

кортежом, а сама модель данных получила название *реляционной модели* (relation model).

Рассмотрим структуру отношения. Отношение характеризуется парой множеств: телом (body), которое содержит кортежи, и заголовком (header), который описывает столбцы таблицы.

Заголовком отношения является множество вида $H = \{< A_1, T_1 >, < A_2, T_2 > , \ldots, < A_n, T_n >\}$. Т.е. заголовок отношения - это множество пар `<имя столбца, тип данных>`, каждая такая пара описывает один столбец.

Схема базы данных в реляционной модели данных – это набор именованных заголовков отношений вида $H_i = \{< A_i^1, T_i^1 >, < A_i^2, T_i^2 >, \ldots, < A_i^{ni}, T_i^{ni} >\}$. T_i называется *доменом* атрибута A_i. По Кодду, каждый домен T_i является подмножеством значений некоторого базового типа данных T_i^+, а значит, к его элементам применимы все операции этого базового типа. Т.е. домен накладывает ограничение на исходное множество значений.

- В конце 1960-х гг. базовыми типами данных считались типы данных распространенных тогда языков программирования.
- В IBM наиболее популярными языками были PL1 и COBOL.

Реляционная БД представляет собой набор именованных *отношений* R_i, каждое из которых обладает заголовком H_{R_i}, определенным в схеме БД, и телом B_{R_i}. B_{R_i} – это множество кортежей вида $\{< A_i^1, T_i^1, v_i^1 >, < A_i^2, T_i^2, v_i^2 >, \ldots, < A_i^{ni}, T_i^{ni}, v_i^{ni} >\}$, где $v_i^j \in T_i^j$. Имя отношения R_i совпадает с именем заголовка этого отношения H_{R_i}.

На рис. 16 приведён наглядный пример описанных множеств. На рисунке A_i - имена столбцов, T_i - тип данных, v_{ij} - сами данные.

Заголовок	<A_1,T_1>	<A_2,T_2>	...	<A_n,T_n>
Тело	<A_1,T_1,v_{11}>	<A_2,T_2,v_{12}>	...	<A_n,T_n,v_{1n}>
	<A_1,T_1,v_{21}>	<A_2,T_2,v_{22}>		<A_n,T_n,v_{2n}>

Рис. 16: Отношение состоит из заголовка и тела; каждая строка - отдельный кортеж

Для лучшего понимания на рис. 17 приведён пример отношения со столбцами «Имя» и «дата рождения» с типами string и date соответственно. В теле отношения содержится два кортежа (имя столбца и тип опущены): `{Вася, 6/10/1980}` и `{Петя, 1/19/1991}`.

<Имя, string>	<дата рождения, date>
<Имя, string, Вася>	<дата рождения, date, 6/10/1980>
<Имя, string, Петя>	<дата рождения, date, 1/19/1991>

Рис. 17: Пример отношения

Представим себе, что определена схема отношения `СПЕЦИАЛИСТ`, в которой имеются: название учебного заведения, которое он закончил; строка, перечисляющая его

навыки и т.д. Пусть в нашей базе данных требуется определить две разновидности специалистов: одна для программистов, другая для дворников. Очень хочется и тех, и других описать одной схемой, но реляционная модель данных этого не позволяет. В этом случае нужно создать два заголовка отношения (фактически одинаковых).

Чем это плохо? Предположим, требуется получить множество всех специалистов из БД. Для этого необходимо объединить множество программистов с множеством дворников. Но чтобы это сделать, надо каким-то образом убедиться, что эти заголовки отношений устроены одинаковым образом. Это проблема *структурной эквивалентности типов*, которая была незамечена в реляционной модели Дейта и Дарвена.

Манипулирование данными в реляционной модели

Отношение - это множество кортежей. Поэтому, к отношениям применимы обычные теоретико-множественные операции: объединение, пересечение, вычитание, взятие декартова произведения. Понятно, что эти операции применимы к любым телам отношений, но результатом не будет отношение, если у отношений-операндов не совпадают заголовки. Кодд предложил в качестве средства манипулирования реляционными базами данных специальный набор операций, которые гарантированно производят отношения.

Этот набор операций принято называть *реляционной алгеброй Кодда*, хотя он и не является алгеброй в математическом смысле этого термина, поскольку некоторые бинарные операции этого набора применимы не к произвольным парам отношений. В алгебре Кодда имеется десять операций:

- объединение (`UNION`)
- пересечение (`INTERSECT`)
- вычитание (`MINUS`)
- взятие расширенного декартова произведения (`TIMES`)
- переименование атрибутов (`RENAME`)
- ограничение (`WHERE`)
- проекция (`PROJECT`)
- соединение (Θ-`JOIN`)
- деление (`DIVIDE BY`)
- присваивание

При выполнении операции объединения (`UNION`) двух отношений с одинаковыми заголовками производится отношение, включающее все кортежи, входящие хотя бы в одно из отношений-операндов.

Операция пересечения (`INTERSECT`) двух отношений с одинаковыми заголовками производит отношение, включающее все кортежи, входящие 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, являющееся вторым операндом.

При выполнении расширенного декартова произведения (`TIMES`) двух отношений, пересечение заголовков которых пусто (без общих атрибутов в заголовках), производится отношение, кортежи которого производятся путем объединения кортежей первого и второго операндов.

Операция переименования (`RENAME`) производит отношение, тело которого совпадает с телом операнда, но имена атрибутов изменены. Эта операция позволяет выполнять `UNION`, `INTERSECT` и `MINUS` над отношениями с почти совпадающими заголовками (совпадающими во всем, кроме имен атрибутов) и выполнять операцию `TIMES` над отношениями, пересечение заголовков которых не является пустым.

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 Θ-соединении (Тета-соединении) (`Θ-JOIN`) двух отношений по некоторому условию (Θ) образуется результирующее отношение, кортежи которого производятся путем объединения кортежей первого и второго отношений и удовлетворяют этому условию.

У операции реляционного деления (`DIVIDE BY`) два операнда – бинарное и унарное отношения. Результат действия рассмотрим на примере. Пусть заданы отношения:

- «служащие и проекты» со столбцами `{СЛУ_НОМ,ПРО_НОМ}` - номер служащего и номер проекта, над которым он работает
- «проекты» с одним столбцом `{ПРО_НОМ}`, причём в теле указаны все номера проектов

В результате деления (`DIVIDE BY`) «служащие и проекты» на «проекты» будут получены те служащие, которые работают над всеми проектами сразу. Результирующее отношение состоит из унарных кортежей, включающих значения первого атрибута кортежей первого операнда таких, что множество значений второго атрибута (при фиксированном значении первого атрибута) включает множество значений второго операнда.

Операция присваивания (`:=`) позволяет сохранить результат вычисления реляционного выражения в существующем отношении БД.

Целостность в реляционной модели данных

Кодд предложил два декларативных механизма поддержки целостности реляционных БД, которые должны поддерживаться в любой реализующей её СУБД:

- ограничение целостности сущности (ограничение первичного ключа)
- ограничение ссылочной целостности (ограничение внешнего ключа)

Ограничение целостности сущности

Для заголовка любого отношения БД должен быть определен первичный ключ, являющийся таким минимальным подмножеством заголовка отношения, что в любом теле этого отношения, которое может появиться в БД, значение первичного ключа в любом кортеже этого тела уникально.

Под минимальностью первичного ключа понимается то, что если к множеству атрибутов первичного ключа добавить хотя бы один атрибут, то ограничение целостности нарушится, т.е. могут появляться тела отношений, которые не допускались исходным первичным ключом.

Например, в отношении «служащие» естественными образом номер служащего - первичный ключ. Если добавить к этому атрибуту ещё и номер отдела, то такая пара тоже будет уникальной, но изменится смысл. Уникальная пара `<номер служащего, номер отдела>` требует, чтобы в одном отделе не было двух служащих с одинаковым номером (а в разных могут).

Если первичный ключ не объявляется явно, то в качестве первичного ключа отношения принимается весь его заголовок. Поскольку по определению любое тело отношения с заданным заголовком является множеством, следовательно, в нем отсутствуют дубликаты, и первичный ключ, совпадающий с заголовком отношения, всегда обладает свойством уникальности.

Все столбцы просто «склеиваются» в один длинный вектор и требуется уникальность по такому объединению.

Ограничение ссылочной целостности

Вспомним пример информационной системы (стр. 21). В принципе, при использовании реляционной модели данных можно хранить все данные, соответствующие предметной области в одной таблице, как это показано на рис. 18.

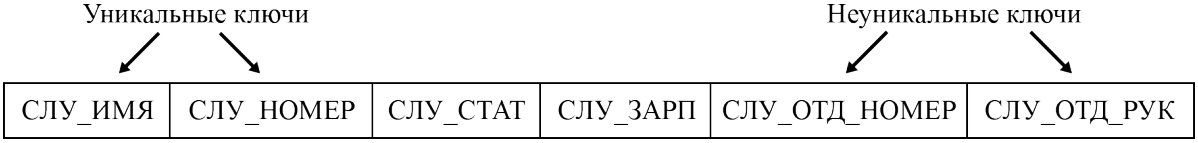


Рис. 18: Схема таблицы для хранения данных информационной системы

Такой подход приводит к избыточности хранения (данные об отделе повторяются в каждой записи о служащем этого отдела) и усложняет выполнение некоторых операций.

Рис. 19: Схема информационной системы из двух файлов

На помощь приходит двухфайловая организация базы данных из файлов `СЛУЖАЩИЕ` и `ОТДЕЛЫ` (рис. 19). Они связаны с помощью полей `СЛУ_ОТД_НОМЕР` и `ОТД_НОМЕР`. В терминах реляционной модели данных:

- `ОТД_НОМЕР` является первичным ключом в отношении `ОТДЕЛЫ`
- `СЛУ_ОТД_НОМЕР` является внешним ключом в отношении `СЛУЖАЩИЕ`, ссылающийся на `ОТДЕЛЫ`

Внешний ключ отношения R_1, который ссылкается на отношение R_2 - это подмножество заголовка H_{R_1}, которое совпадает с первичным ключом отношения R_2 (с точностью до имен атрибутов).

Тогда *ограничение ссылочной целостности* реляционной модели данных можно сформулировать следующим образом:«В любом теле отношения R_1, которое может появиться в БД, для «не пустого» значения внешнего ключа, ссылающегося на отношение R_2, в любом кортеже этого тела должен найтись кортеж в теле отношения R_2, которое содержится в БД, с совпадающим значением первичного ключа».

Легко заметить, что это почти то же самое ограничение, о котором говорилось в иерархической модели данных: никакой потомок не может существовать без своего родителя, но немного уточненное – ссылки на родителя должны быть корректными.

Современные модели данных

В 1980-х гг. в связи с развитием технологий БД стал актуальным вопрос упрощения работы со сложными типами в БД. Реляционная модель данных позволяет работать только с атомарными типами (float, double, string, int и т.д.). Но в то же время реальные СУБД требуют хранения сложных структур данных. Так вот следующие три манифеста направлены как раз на решение проблемы несоответствия типов данных (impedance mismatch).

Первый манифест

История современных моделей данных началась с 1989 г., когда группа известных специалистов в области языков программирования баз данных опубликовала статью под названием «Манифест систем объектно-ориентированных баз данных» (Первый манифест).

- Malcolm Atkinson, Francois Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley Zdonik: "The Object-Oriented Database System Manifesto", Proc. 1st International Conference on Deductive and Object-Oriented Databases, Kyoto, Japan (1989). New York, N.Y.: Elsevier Science (1990) (ссылка)
- Имеется русский перевод: М. Аткинсон и др. "Манифест систем объектно-ориентированных баз данных", СУБД, No. 4, 1995 (ссылка)

К этому времени уже было несколько реализаций объектно-ориентированных СУБД (ОО СУБД), но каждая из них опиралась на некоторое расширение объектной модели какого-либо ОО-языка программирования (Smalltalk, Object Lisp, C++), отсутствовали какие-либо общие подходы.

В Первом манифесте не предлагалась единая ОО-модель данных, но выделялся набор требований к ОО-СУБД:

- решение проблемы потери соответствия (impedance mismatch) между типами данных, используемыми в языках программирования, и типами данных, поддерживаемыми SQL-ориентированных СУБД
- обеспечение такой архитектуры СУБД, которая позволит хранить в БД данные произвольно сложной структуры

Эти требования сопровождались утверждениями об ограниченности реляционной модели данных и языка SQL и потребности использовать более развитые модели данных.

Лекция 8

Под влиянием Первого манифеста в 1991 г. возник консорциум ODMG (Object Database Management Group), задачей которого была разработка стандарта OO-модели данных.

ODMG существовал более десяти лет и опубликовал три базовых версии стандарта, последняя из которых называется ODMG 3.0 (ссылка). На этот документ мы и будем опираться в дальнейшем изложении.

Второй манифест

В ответ на публикацию Первого манифеста группа исследователей, близких к индустрии БД, в 1990 г. опубликовала документ «Манифест систем баз данных третьего поколения» (Второй манифест).

- M. Stonebraker, L. Rowe, B. Lindsay, J. Gray, M. Carey, M. Brodie, Ph. Bernstein, D. Beech. “Third-Generation Data Base System Manifesto”. Proc. IFIP WG 2.6 Conf. on Object-Oriented Databases, July 1990, ACM SIGMOD Record 19, No. 3 (September 1990) (ссылка)
- Имеется русский перевод: Стоунбрейкер М. и др. “Системы баз данных третьего поколения: Манифест”, СУБД, No. 2, 1996, (ссылка)

Второй манифест был направлен на защиту интересов крупных компаний-производителей программного обеспечения SQL-ориентированных СУБД. Соглашаясь с авторами Первого манифеста относительно потребности обеспечения развитой системы типов данных в СУБД, авторы Второго манифеста утверждали, что можно добиться аналогичных результатов, не производя революцию в области технологии БД, а эволюционно развивая технологию SQL-ориентированных СУБД.

Вслед за Вторым манифестом последовало появление *объектно-реляционных* продуктов ведущих компаний-поставщиков SQL-ориентированных СУБД (Informix Universal Server, Oracle8, IBM DB2 Universal Database).

Итак, в начале 1990-х гг. были провозглашены два манифеста, каждый из которых претендовал на роль программы будущего развития технологии БД. В первом манифесте реляционная модель данных отвергалась полностью. Во втором реляционная модель заменялась еще незрелой к тому времени моделью данных SQL, которая уже тогда была далека от реляционной модели.

Третий манифест

На защиту реляционной модели данных в её первозданном виде встали Кристофер Дейт и Хью Дарвен, опубликовавшие в 1995 г. статью, под названием «Третий манифест»:

- Hugh Darwen and C. J. Date: The Third Manifesto. ACM SIGMOD Record 24, No. 1 (March 1995)
- Имеется русский перевод: Х. Дарвин, К. Дейт. “Третий манифест”, СУБД, No. 1, 1996, (ссылка)

Третий манифест является одновременно наиболее консервативным и радикальным. Консервативность заключается в том, что авторы всеми силами утверждают необходимость и достаточность использования в системах БД следующего поколения классической реляционной модели данных. Радикальность состоит в том, что:

- авторы полностью отрицают подходы, предлагаемые в первых двух манифестах, расценивая их как необоснованные, плохо проработанные, избыточные и даже вредные (за исключением одной общей идеи о потребности обеспечения развитой системы типов)
- фактически, авторы полностью отбрасывают технологию, созданную индустрией БД за последние 40 лет, и предлагают вернуться к истокам реляционной модели данных, т.е. к начальным статьям Эдгара Кодда

Позже Дейт и Дарвен написали книгу, первое издание которой вышло в 1998 г. под названием «Foundation for Object/Relational Databases: The Third Manifesto», второе издание вышло в 2000 г. под названием «Foundation for Future Database Systems: The Third Manifesto» (имеется перевод второго издания на русский язык) и третье вышло в 2006 г. под названием «Databases, Types and the Relational Model: The Third Manifesto» (ссылка на книгу).

В этих книгах очень подробно излагается подход авторов к построению СУБД на основе, как они утверждают, истинных идей Эдгара Кодда, изложенных им в своих первых статьях про реляционную модель данных. Некоторые более поздние идеи Кодда относительно той же реляционной модели авторами отвергаются. В любом случае, Дейт и Дарвен предлагают некоторый современный вариант реляционной модели данных, который, безусловно, заслуживает внимания и изучения. Далее для определенности будем называть её *истинной реляционной моделью*. Мы рассмотрим только основные черты этой модели.

Объектно-ориентированная модель данных

Если не обращать внимания на особенности объектно-ориентированной (ОО) терминологии, то ОО-модель данных отличается от модели данных SQL и истинной реляционной модели, прежде всего, в одном принципиальном аспекте.

- В модели данных SQL и истинной реляционной модели данных БД представляет собой набор именованных контейнеров данных одного родового типа: таблиц или отношений соответственно.
- В ОО-модели данных БД – это набор объектов (контейнеров данных) произвольного типа.

В объектной модели данных вводятся две разновидности типов:

- литеральные (т.е. с явным определением области значений)
- объектные, которые делятся на два вида: атомарные объектные типы и объектные типы коллекций

Неформально говоря, любой тип данных обладает тремя характеристиками: множеством значений, набором операций и литералами. Литерал - это внешнее обозначение значения типа. На примере boolean:

- Множеством значений является $\{true, false\}$.
- Битовые операции and, or, xor и т.д.
- Литералы $true$ и $false$

Литеральные типы данных

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 (структуры) и коллекций

Литеральный тип записи – это традиционный определяемый пользователем структурный тип, подобный типу struct в языке Си. 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. Это звучит странно, но здесь все странности проистекают из особенностей ОО-терминологии.

У любого объекта имеется объектный идентификатор (OID). Этот идентификатор возникает при создании объекта и является его именем. С его помощью можно получить доступ к атрибутам и методам объекта. 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:

- множество
- мультимножество (множество, допускающее дубликаты)
- список (упорядоченный набор элементов, допускающий дубликаты)
- словарь (множество пар `<уникальный ключ, значение>`)

На практике используется только множество (в истинной реляционной модели, допустим, кроме типа множества других типов коллекций нет). Типом элемента

любой коллекции может являться любой скалярный или объектный тип, кроме того же типа коллекции.

Атомарный объектный тип

Объектные типы в объектной модели данных по смыслу ближе всего к понятию класса в ОО-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.

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.

Атрибутом называется свойство объекта, значение которого можно получить по OID объекта. Значениями атрибутов могут быть и литералы, и другие объекты, но только тогда, когда не требуется обратная ссылка.

Связи – это инверсные свойства. В этом случае значением свойства может быть только объект. Связи определяются только между атомарными объектными типами. В объектной модели ODMG поддерживаются только бинарные связи. Связи могут быть «один-к-одному», «один-ко-многим» и «многие-ко-многим» в зависимости от того, сколько экземпляров соответствующего объектного типа может участвовать в связи.

Пример связи «многие-ко-многим»: полицейские и бандиты. Каждый бандит убегает от каждого полицейского, и любой полицейский ловит любого бандита.

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 из (`consists_of`) множества служащих. Тогда путь обхода `consists_of` должен быть определен в объектном типе `ОТДЕЛ`, а путь обхода `works` – в объектном типе `СЛУЖАЩИЙ`.

Тот факт, что пути обхода относятся к одной связи, указывается в разделе `inverse` обоих объявлений пути обхода. Т.е. в служащих написано, что для связи `works` инверсной связью является `consists_of`. А в отделах наоборот указывается, что для связи `consists_of` инверсной связью является `works`.

Данная связь является связью «один-ко-многим» (один отдел, много служащих).

- Путь обхода `consists_of` ассоциирует объект типа `ОТДЕЛ` с множеством объектов типа `СЛУЖАЩИЙ`

- путь обхода `works` ассоциирует объект типа `СЛУЖАЩИЙ` с объектом типа `ОТДЕЛ`

Хотя связь является модельным понятием, другие понятия модели наталкивают на мысль, что единственным способом реализации связей является хранение в объекте OID или коллекции OID связанных объектов в зависимости от вида связи. Т.е. можно в объекте `СЛУЖАЩИЙ` объявить атрибут объектного типа `ОТДЕЛ` (который будет хранить OID отдела), а в отделе в свою очередь объявить атрибут «set_of_employees» (и хранить там множество OID сотрудников).

Объектные типы коллекций

Как и в случае литеральных типов коллекций, можно определять объектные типы множеств, мультимножеств, списков, словарей.

Типом элемента объектного типа коллекции может быть любой литеральный или объектный тип, кроме самого того типа коллекции. У объектных типов коллекций есть набор базовых операций (пересечение, вычитание, объединение и т.д.), а также операция создания объекта, обладающего собственным OID.

Интересен и важен специальный случай неявного использования объектов типа множества. При определении атомарного объектного типа в качестве дополнительного свойства можно указать, что для него должен быть создан экстент.

Экстент - это объект типа множества, элементами которого являются объекты данного атомарного объектного типа. Другими словами в экстенте сохранены OID'ы всех объектов данного атомарного типа, на данный момент существующие в БД. Поскольку экстент создается неявно, его OID неизвестен, но известно имя, которое совпадает с именем атомарного объектного типа.

Наличие этой возможности позволяет работать с ОО-БД, как с набором экстентов, что очень похоже на таблицы в SQL БД. Именно такая организация предпочтительна на практике, т.к. она позволяет работать с ОО-БД так, как будто она представляет собой набор таблиц. Т.е. можно сопоставить: экстент с таблицей, объекты со строками таблицы.

Получается, что если в ОО-БД представляется как набор экстентов, то такая БД очень похожа на современные SQL-ориентированные БД. Подробнее по данному вопросу можно прочитать в статье автора курса:

- С. Д. Кузнецов. “Объектные модели ODMG и SQL десять лет спустя: нет противоречий”. ИСП РАН, 2015 г. (ссылка)

Лекция 9

Манипулирование данными

В стандарте ODMG в качестве базового средства манипулирования объектными БД предлагается язык OQL (Object Query Language). 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).
- Синтаксис OQL очень близок к SQL/92.
- Расширения относятся к объектно-ориентированным понятиям, таким как
  - сложные объекты
  - объектные идентификаторы OID
  - путевые выражения (переход, например, от служащего к его отделу)
  - полиморфизм
  - вызов операций
  - отложенное связывание (late binding)

Полиморфизм - это возможность определения в одном контексте нескольких функций с одним именем. Есть два вида полиморфизма: полиморфизм по перегрузке и полиморфизм по включению.

Полиморфизм по перегрузке разрешается на стадии компиляции. Компилятор смотрит на аргументы функции и выбирает наиболее подходящую функцию. Например, «1 + 1» и «1f + 1f» - это две разных функции сложения. У первой аргументы - целые числа, а у второй - float.

Полиморфизм по включению позволяет при наследовании в потомке переопределить уже существующую у предка функцию. В этом случае у функции остаётся то же самое имя, а аргументы - любые. Его нельзя разрешить на стадии компиляции, поэтому такое связывание называется отложенным (late binding).

Некоторые свойства OQL:

- В OQL обеспечиваются высокоуровневые примитивы для работы с множествами объектов, со структурами и с массивами.
- OQL является функциональным языком, допускающим неограниченную композицию операций, если операнды не выходят на пределы системы типов.

- Для любого запроса можно определить тип его значения, который принадлежит к модели типов ODMG, поэтому, самое главное, к результату любого запроса может быть применён 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 определены явные операции обновления (UPDATE объектов), а используются вызовы операций, определенных в объектах для целей обновления. На момент создания считалось, что внешние операции UPDATE не должны вообще присутствовать, и можно обойтись методами самого объекта.
- Можно легко определить формальную семантику OQL.
- В OQL обеспечивается декларативный доступ к объектам. По этой причине OQL-запросы могут хорошо оптимизироваться.
  - Если для каждого атомарного объектного типа определён экстент, и любой поиск начинается с поиска в экстенте, тогда это правда.
  - Но если БД внешне представлена как набор разнотипных объектов, то возможен только навигационный доступ, т.е. надо ходить по ссылкам.

Пример запроса на языке OQL

Требуется 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 ))
FROM ОТДЕЛЫ D
```

В данном запросе `ОТДЕЛЫ` является экстентом атомарного объектного типа `ОТДЕЛ`. Для каждого отдела выбирается руководитель отдела `ОТД_РУК` (типа `INTEGER`), а также нечто под именем `СЛУ`, что является множеством служащих данного отдела со `СЛУ_ЗАРП > 20000`.

Результатом запроса является литеральное значение - множество, элементами которого являются значения-структуры с двумя литеральными значениями:

- `ОТД_РУК` - атомарное литеральное значение типа `INTEGER`
- `СЛУ` – литеральное значение - множество с элементами типа `СЛУЖАЩИЙ`

Как дальше с таким множеством работать? Предположим, что ОО-модель реализуется в клиент-серверной манере. Клиенты взаимодействуют с сервером на

языке OQL, а на сервере есть хранимые процедуры (помимо методов объектов). Если такой запрос выполняется внутри хранимой процедуры, то можно с результатом работать в терминах OQL.

Но что делать, если запрос происходит со стороны клиента? Нужно преобразовать результирующее множество в нечто, что будет понятно программе на стороне клиента. И такое преобразование совсем не тривиальное.

Например, требуется ответить на вопрос что именно скрывается под этим множеством объектов. Как абстрактные указатели в БД перевести в указатели на клиенте? Эта задача называется проблемой Свизлинга.

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 соответствии с общей идеологией ОО-подхода в модели ODMG два объекта считаются совпадающими в том и только в том случае, когда они имеют один и тот же OID. Объекты 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. Не совсем правда. Так как целостность сущности говорит, что в одном отношении (таблице) не может быть полностью совпадающих кортежей (строк).

Интересно, что при определении атомарного объектного типа можно объявить ключ – набор свойств, однозначно идентифицирующий состояние каждого объекта, 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.

Пример. Пусть есть объектный тип `ВЗРОСЛЫЙ_ЧЕЛОВЕК`, у которого есть уникальный атрибут «номер паспорта». Т.е. в экстенте класса `ВЗРОСЛЫЙ_ЧЕЛОВЕК` не должно появится двух OID с одинаковым номером паспорта. Но почему-то в объектной модели ODMG это не является ограничением целостности, а лишь указанием СУБД, что по этому ключу нужно индексировать.

Что касается ссылочной целостности, то она поддерживается, если между двумя атомарными объектными типами определена связь «один-ко-многим». В этом случае объекты на стороне связи «один» рассматриваются как предки, а объекты на

стороне связи «многие» – как потомки, и ОО-СУБД обязана следить за тем, чтобы не образовывались потомки без предков.

Модель данных SQL

Модель данных SQL в данном курсе соответствует стандарту SQL/2003. Т.к. изменения более поздних стандартах 2008, 2011, 2013 и 2016 годов на модель данных не влияли.

Типы и структуры данных

SQL-ориентированная БД 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. Допущение мультимножества связано с операцией проекции, которая может на выходе давать дубликаты.

Вторым существенным отличием является того, что для таблицы поддерживается порядок столбцов, соответствующий порядку их определения. Другими словами, таблица – это вовсе не отношение (как в реляционной модели), хотя во многом они похожи.

Имеется две основных разновидности таблиц, хранимых в БД: *традиционная* и *типизированная* таблицы.

Традиционная таблица

Традиционная таблица определяется как последовательность столбцов с указанными типами данных.

В SQL поддерживаются следующие категории типов данных:

- точные числовые (int, long int, decimal, numeric)
- приближенные числовые (float и double)
- типы символьных строк (char(N), varchar(N))
- типы битовых строк
- типы даты и времени; временных интервалов
- булевский тип
- типы коллекций
- пользовательские типы
- ссылочные типы

Точные числовые типы включают в себя привычные integer и long integer, а также decimal и numeric - тут указано кол-во десятичных знаков под значения, и кол-во знаков после запятой.

Типы символьных строк включают в себя char(N) - строка размера N (для строки меньшей длины добавятся пробелы), varchar(N) - строки не длинее N (строки меньшей длины будут занимать только свою длину), clob - строки произвольного размера (для книг, статей).

Типы битовых строк нужны для хранения картинок, звуков, видео. Могут быть фиксированной или переменной длины, а так же blob для битовых строк произвольного размера.

Типы даты и времени. Дата - это целое число дней прошедшее с Рождества Христова. Время - количество долей секунды (миллисекунд или микросекунд - зависит от точности) текущего дня. Timestamp - комбинация даты и времени.

Булевский тип в SQL

Булевский тип в SQL содержит три значения – true, false и unknown. Это связано с интенсивным использованием в SQL неопределенного значения NULL, которое разрешается использовать вместо значения любого типа данных. NULL может быть интерпретирован как отсутствие значения.

Для любой бинарной операции ob:

- A ob NULL = NULL
- NULL ob A = NULL
- NULL ob NULL = NULL

Для любой операции сравнения comp $\in \{<, \leqslant, >, \geqslant, ==\}$:

- A comp NULL = unknown
- NULL comp A = unknown
- NULL comp NULL = unknown

Какие недостатки у трёхзначной логики в модели SQL? Первое, её не хватает. Так как за unknown могут скрываться более детальные случаи:

- Просто неизвестно значение
- Для данного контекста значения не существует
  - Например, хотим хранить в одной таблице все возможные аэропланы. Их два: самолёты и планеры. Так как мы их храним в одной таблице, то там будет столбец мощности двигателя. Очевидно, у аэроплана нет двигателя, но обозначаем мы так же, как если нам она неизвестна.
- Есть информация о характеристиках неизвестного значения

– Допустим, что военнообязанный младше 27 лет не сообщает в военкомат свою точную дату рождения, но известно, что по здоровью он годен к службе в армии. Нужно-бы ему записать другой NULL, который будет отделять его от молодых людей старше 27ми лет с неизвестным возрастом.

Если использовать несколько NULL, то логика становится уже 5ти значной. На практике её не используют, потому что она неинтуитивная. Даже с трёхзначной логикой true, false, unknown не всегда понятно как работать, а с 5ти значной тем более.

Понимание 5ти значного случая осложняется ещё и тем, что в трёхзначной логике есть упорядоченность: `false < unknown < true`. `false` - это как бы 0, `true` - 1, а `unknown` - 0.5. А в 5ти значном случае такую упорядоченность представить сложно.

Лекция 10

Следует отметить, что в SQL (NULL==NULL) = unknown, но есть случаи, когда это выражение неявно равно true. Например, если из служащих мы выбираем столбец с заработной платой, и сделаем его DISTINCT (без повторов), то останется не более одной строки со значением NULL. Можно также пересчитать количество строк с NULL, то есть их можно различить. Это и означает, что при сравнении они неявно могут давать true.

Представим себе, что вычисляется булевское выражение `true and NULL`. С одной стороны, мы не знаем что скрывается за NULL и результатом будет unknown, с другой стороны, по определению NULL это должно быть NULL.

Типы коллекций

Допускается объявление двух видов типов коллекций: тип массива и тип мультимножества.

Тип массива требует указания длины массива и имеет операции доступа и изменения i-го элемента массива. Примером использования типа массива может быть случай, когда у одного человека есть несколько телефонных номеров. Тогда в таблице тип столбца «номер телефона» можно объявить массивом строк. В данном случае можно обойтись и без массива, а просто в одной строке перечислять номера через разделители. Поэтому на практике тип массива не используется.

Элементы типа коллекции могут быть любого типа данных, определенного к моменту определения данного типа коллекции. До стандарта 2003 года SQL допускал только типы массива. В SQL/2003 появились тип мультимножества и анонимный строчный тип.

Анонимный строчный тип – это безымянный структурный тип, значения которого - строки, состоящие из элементов ранее определенных типов. Можно объявить некоторый столбец C таблицы T типом мультимножества анонимного строчного типа. Таким образом в каждой строке таблицы T внутри столбца C будет другая вложенная таблица (nested table). Так можно представить любую иерархию с любой степенью вложенности.

Стоит отметить, что при объявлении типа мультимножества можно явно запретить наличие в его значениях элементов-дубликатов, что фактически приводит к объявлению типа множества.

Именно появление типа коллекций окончательно отделило модель данных SQL от реляционной модели данных, где Эдгар Кодд запрещал вложенность и требовал атомарность значений атрибутов в отношениях.

Пользовательские типы

Поддерживается два вида типов данных, определяемых пользователями: индивидуальный тип и структурный тип.

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.

Типизированная таблица

При определении типизированной таблицы указывается ранее определенный структурный тип S. Пусть в нём содержится n атрибутов. Тогда в таблице образуется n+1 столбец:

- последние n столбцов с именами и типами данных, совпадающими именам и типам атрибутов структурного типа S
- первый столбец, имя которого явно задается, называется «самоссылающимся» (self-referencing) и содержит типизированные уникальные идентификаторы строк (точно различные для любого мультимножества)
  - они могут генерироваться системой при вставке строк в типизированную таблицу
  - явно указываться пользователями
  - состоять из комбинации значений других столбцов

Типом «самоссылающегося» столбца является ссылочный тип, ассоциированный со структурным типом S. Способ генерации значений его значений указывается при определении S и подтверждается при определении типизированной таблицы.

Типизированные таблицы можно определять с использованием механизма наследования. Можно определить подтаблицу типизированной подтаблицы, тогда и только тогда, когда 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.

С типизированной таблицей можно обращаться, как с традиционной таблицей, считая, что у нее имеются неявно определенные n+1 столбец, а можно относиться к строкам типизированной таблицы, как к объектам структурного типа, OID которых содержатся в «самоссылающемся» столбце.

Тип столбца в традиционной таблице а также атрибуты в структурных типах также могут быть ссылочными.

Манипулирование данными

Язык SQL содержит один оператор выборки `SELECT`, который в действительности является целым подъязыком SQL. Рассмотрим структуру оператора `SELECT`.

```
SELECT [DISTINCT] списокСтолбцов1
FROM списокССылокНаТаблицы
[WHERE условноеВыражение1]
[GROUP BY списокСтолбцов2]
[HAVING условноеВыражение2]
[ORDER BY списокСтолбцов3]
```

Выборка данных производится из одной или нескольких таблиц, указываемых в разделе `FROM` запроса. Если таблиц несколько, то на первом этапе выполнения оператора `SELECT` образуется одна общая таблица, получаемая из исходных таблиц путем операции расширенного декартова умножения.

Таблицы могут быть как базовыми, реально хранимыми в базе данных (традиционными или типизированными), так и порожденными, т.е. являющиеся результатом некоторого оператора `SELECT`. Это допускается, поскольку результатом выполнения любого оператора `SELECT` (без `ORDER BY`) является традиционная таблица.

Кроме того, в разделе `FROM` можно указывать соединенные таблицы. Т.е. можно прямо в `SELECT` операторе соединять базовые и порождённые таблицы, в результате опять же будет традиционная таблица.

После `FROM` мы получили таблицу - естественное расширение всех таблиц. В нашей результирующей таблице количество столбцов будет равно сумме количества столбцов каждой таблицы.

На следующем шаге выполняется фильтрация и остаются только те строки, для которых выполняется `условноеВыражение1` раздела `WHERE`. Помним, что логика трёхзначная, поэтому строки с результатом unknown не войдут в результирующую таблицу. Получили подмультимножество расширенного декартова произведения таблиц из `FROM`.

Если в операторе `SELECT` отсутствует раздел `GROUP BY`, то после этого происходит формирование результирующей таблицы запроса путем вычисления выражений, заданных в списке выборки оператора `SELECT`.

Список выборки оператора `SELECT` - это скалярные выражения, тип результата которых - скаляр (int, char, varchar, bool, date, и т.д.). Например, это функции без параметров, которые могут быть в запросе `CURRENTTIME()`, `DATE()`, `CURRENTUSER()`; или строки, алгебраические выражения `(90+900)/2-51`, и т.д. Также в списке выборки `SELECT` может быть вложенный подзапрос, с единственным ограничением - результатом запроса должен быть скаляр.

Список выборки вычисляется для каждой строки отфильтрованной таблицы, и в результирующей таблице появится ровно столько же строк.

Лекция 11

Стоит отдельно упомянуть, что операции `UPDATE`, `DELETE` и `INSERT` избыточны с точки зрения логики. Поясним что имеется в виду. В SQL есть операция сохранения результата запроса, и т.к. результатом `SELECT` является таблица (если не использовать `ORDER BY`), то изменение в БД можно трактовать как операцию присваивания. Т.е. вместо старой таблицы присваиваем результат некоторого `SELECT`. Операции изменения БД удобны на практике, но избыточны логически. Поэтому, в курсе подробно разговариваем только об операторе `SELECT`.

Возвращаясь к разделу `FROM`, следует добавить, что порождённые таблицы могут также быть созданы с помощью операций объединения таблиц: `LEFT JOIN`, `RIGHT JOIN`, `OUTER JOIN`, на которых мы подробно останавливаться не будем.

Пусть операторе `SELECT` присутствует раздел `GROUP BY`, то к результирующей таблице T, полученной после `WHERE` секции, строится сгруппированная таблица.

Построение сгруппированной таблицы. В `GROUP BY` указывается список столбцов таблицы T. Далее начинается группировка по первому столбцу (сортировка по первому столбцу, игнорируя остальные). Таким образом соседними строками становятся строки с одинаковым первым столбцом. Далее проводим сортировку по второму столбцу. В итоге получатся группы, у которых и первый и второй столбцы одинаковые.

`GROUP BY` нужен для того, чтобы выбрать, к примеру, минимальную, максимальную, среднюю зарплату среди сотрудников отдела. В результате получится сгруппированная таблица, в которой каждая строка соответствует одному отделу.

Раздел `HAVING` действует аналогично разделу `WHERE`, только условие применяется не к отдельным строкам, а к группам. Таким образом отсекаются группы, которые не подходят под условие секции `HAVING`. Синтаксические правила для выражения в `HAVING` такие же, как и в `WHERE`, но использовать можно только те столбцы, которые были указаны в `GROUP BY` секции.

Выражение также может включать агрегатные функции:

- `COUNT(*)` - посчитать число элементов внутри группы
- `min(столбец)` - минимальное значение столбца внутри группы
- `max(столбец)` - максимальное значение столбца внутри группы
- и другие

Если в запросе `SELECT` отсутствует `GROUP BY`, но присутствует `HAVING`, то вся таблица рассматривается как одна группа. И в результате будет только одна строка.

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

Если требуется, чтобы в результате запроса `SELECT` получилась таблица, то обязательно требуется отсутствие `ORDER BY` секции.

В этом случае результирующая таблица так же, как и в `GROUP BY`, сортируется. Отличие в том, что тут можно указать порядок: по возрастанию или по убыванию.

Результатом такого запроса является не таблица, а отсортированный список, который нельзя сохранить в БД. 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.

Например, в реальной реализации `SELECT` запроса, условие `WHERE` разбивается на подусловия для каждой отдельной таблицы, и применяется ещё до расширенного декартового произведения.

Ограничения целостности

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 БД могут присутствовать таблицы, для которых первичный ключ не определен. Если для таблицы определен первичный ключ, то для нее ограничение целостности сущности поддерживается точно так же, как это требуется в реляционной модели данных. А именно, что в значение столбца (совокупности столбцов), выбранным как первичный ключ, должно быть определёно (не NULL) и уникально.

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 NULL.

Есть три спецификации (matching) внешнего ключа и первичного ключа в таблице, на которую идёт ссылка: `FULL`, `SIMPLE`, `PARTIAL`.

`FULL` (полная) спецификация, требует, чтобы для любого внешнего ключа существовала строка с точно таким же первичным ключом (т.е. нельзя ссылаться на несуществующую строку). Точно такое же ограничения сущности требуется в реляционной модели.

`SIMPLE` (простая) спецификация требует для внешнего ключа либо полностью соответствовать некоторому первичному ключу, либо (в случае составного ключа во всех столбцах) быть NULL. Т.е. внешние ключи с частью столбцов указанных, а частью NULL недопускаются.

`PARTIAL` (частичная) спецификация требует либо полной неопределённости во всех столбцах, либо полного соответствия первичному ключу, либо когда лишь часть столбцов NULL, то найдётся такая строка, что указанные столбцы внешнего ключа совпадают со столбцами первичного ключа. В этом случае внешний ключ может ссылаться на несколько строк. Эта спецификация довольно оторвана от реальности и непонятно, когда её можно разумно применить.

Кроме того, помимо ограничения целостности и сущности в SQL имеются развитые возможности явного определения ограничений целостности:

- на уровне домена
- на уровне таблицы
- на уровне базы данных

Домен - это именованный объект БД, который определяется на существующем типе данных и имеет ограничения по множеству значений. Если столбец определяется на домене, то для столбца наследуются ограничения домена.

Ограничения на уровне таблицы подразумевает, что задаётся логическое выражение для строк таблицы. И нельзя добавить строку, для которой значение этого логического выражения true или unknown.

Ограничения на уровне БД позволяет определить ограничение из нашего примера `СЛУЖАЩИЕ-ОТДЕЛЫ`. Для каждой строки Отдел значение столбца размер отдела должно совпадать с количеством служащих в этом отделе. Выполнение таких ограничений обычно проверяется при конце транзакции, и в случае нарушения вся транзакция откатывается.

Истинная РМД

Кристофер Дейт и Хью Дарвен поставили перед собой трудную задачу: показать, что на основе идей Эдгара Кодда можно реализовать СУБД, которая обеспечит возможности по части представления и хранения данных сложной структуры, не меньшие тех, которые обеспечивают объектные и SQL-ориентированные СУБД.

Основные пути для хранения сложной структуры данных, использованные в объектных и SQL-ориентированных СУБД - это:

- возможность определения новых структурных типов
- возможность использования типов коллекций

Однако есть тезис Кодда о нормализации отношений: «В реляционной БД должны содержаться только отношения с атрибутами, определенными на «доменах, элементы которых являются атомарными (не составными) значениями». Другими словами Кодд не допускает множества, отношения, структуры и т.д.

С другой стороны, Кодд не возражал против строк символов, которые являются массивом символов. Почему массив символов разрешён, а массив чисел нет - скольз-

кий момент. Казалось бы, этот тезис напрочь стирает возможность создавать такие же сложные структуры, как в объектных и SQL-ориентированных СУБД.

Дейт пишет: «Я согласен с Коддом, что желательно оставаться в рамках логики первого порядка, если это возможно. В то же время я отвергаю идею "атомарных значений по крайней мере, в смысле абсолютной атомарности. В Третьем манифесте мы допускаем наличие доменов, содержащих значения произвольной сложности (они могут быть даже отношениями). Тем не менее, мы остаемся в рамках логики первого порядка.»

Если учесть, что цитировалась первая официальная публикация Кодда по поводу реляционной модели данных, то трудно сказать, что Дейт очень уж строго следует всем его заветам. Те постулаты Кодда, которые вредят достижению цели Третьего манифеста, просто отвергаются.

В истинно реляционной модели очень большое внимание уделяется типам данных. Предлагаются три категории типов данных:

- скалярные типы
- кортежные типы
- типы отношений

Скалярный тип данных

Скалярный тип данных – это привычный инкапсулированный тип, реальная внутренняя структура которого скрыта от пользователей. Больше всего похож на структурный тип в SQL.

Типом атрибута определяемого скалярного типа может являться любой определенный к этому моменту:

- скалярный тип
- любой кортежный тип
- тип отношения

Предлагаются механизмы определения новых скалярных типов и операций над ними. Для любого определяемого скалярного отношения есть два вида его представления: *реальное* и *возможное*.

Реальное представление скалярного типа может вести только главный разработчик, который начал его с нуля. Например, если мы определяем скалярный тип point (точка на плоскости), то в качестве реального представление можно выбрать декартово. Каждая точка представляется набором их двух float (x, y) с соответствующими операциями сложения, вычитания и т.д.

Возможное представление - это то, что видно пользователям, которые с этим типом хотят работать. Продолжая пример с point, в качестве возможного представления можно выбрать то же самое декартово, которое будет совпадать с реальным.

А вторым возможным представлением может быть в полярной системе координат. В этом случае надо указать как из возможного представления получить реальное.

Возможных представлений может быть сколько угодно. Допускаются возможные представления над другими возможными представлениями (опять же нужно указать отображение).

Хоть скалярный тип данных и инкапсулирован, т.е. внутренняя структура скрыта от пользователя. Однако Дейт и Давен говорят, что у каждого скалярного типа есть столько псевдопеременных, сколько атрибутов в его любом возможном представлении.

Например, в случае с point в полярном возможном представлении есть две псевдопеременные: a - длина радиуса до точки, и α - полярный угол. Для любой пары (x, y) можно получить (a, α), и наоборот, изменив (a, α), поменять (x, y).

Если вспомнить неформальное определение типа данных, то тип данных обладает тремя характеристиками:

- множеством значений
- набором операций
- литералами (Как внешним образом представить значения этого типа?)

Что касается литералов, то всё более-менее понятно, пока типы данных простые Но что такое литерал типа множества или массива? В связи с этим для каждого возможного представления предопределяется операция `SELECT`, которая по заданным значениям псевдопеременных выбирает соответствующие значения из множества значений реального представления.

Например, из возможного представления (a, α), сделав `SELECT (1,П/4)`, можно получить реальное значение точки.

Некоторые базовые скалярные типы данных должны быть предопределены в системе. В число этих типов должен входить тип «truth value» (так они называют булевский тип), который принимает true или false. Почему они ограничились двоичной логикой?

В начале 90-х гг. Дейт и Дарвен лютые противники SQL. В частности, они долгое время потратили на то, чтобы привести примеры, где SQL выдаёт неправильные результаты. В том числе, они против неопределённых значений NULL и трёхзначной логики. Во втором издании их истинной реляционной модели Дейт предложил подход отказа от неопределённых значений, оставшись в рамках двузначной логики.

Дейт предложил следующее решение. Допустим мы знаем, что в БД будет хранится отношение `СЛУЖАЩИЙ`, и, например, в атрибуте «год рождения» иногда приходится помещать информацию о том, что данные неизвестны. Пусть тип данных этого столбца $Date$ (дата), тогда определим новый тип данных $Date'$ со множеством значений $Date$ + специальное значение C.

Условия на C:

- $C == C = true$
- $C > C = C < C = C! = C = false$
- $\forall d \in Date$, $d\ op\ C = false$, где op - любая операция сравнения

В случае, если дата неизвестна, будет указано значение C. К сожалению, если так определять это специальное значение, то по правилам логики оказывается, что C меньше минимального значения этого типа и больше максимального.

Возможно, в связи с этим в третьем издании истинной реляционной модели специальные значения были убраны совсем. Но в итоге была принято решение Хью Дарвена, которое логически верное, но на практике бессмысленное.

Состоит оно в следующем. Пусть есть отношение, в котором n атрибутов не могут содержать неопределённое значение, а n+1-й может. Тогда разобьём это отношение на два: в первом будет n атрибутов, для которых мы значение n+1-го (пусть это будет возраст) не знаем, а во втором отношении будут все n+1 атрибутов.

Как получить информацию обо всех служащих. При запросе данных мы расширяем отношение без возраста и добавляем n+1-й атрибут «возраст», значение которого разработчик приложения задаёт сам. А дальше объединем эти два отношения.

Это объединение происходит на на уровне базы данных, а на уровне приложения. Грубо говоря, разработчик сам решил «закостылить» отсутствие значения, а в БД всё чистенько.

Допустим, что столбцов, которые допускают неопределённые значения, не один, а k. Тогда, следуя такому решению, нужно будет создать 2^k таблиц, что на практике, понятное дело, катастрофично.

Лекция 12

Кортежный тип данных

Заметим, что в истинной РМД отсутствует понятие домена, т.к. оно логически избыточное. Скалярный тип позволяет сделать то же, что и домен (из реляционной модели Кодда).

Кортежный тип очень похож на анонимный строчный тип SQL. Кортежный тип – это безымянный тип, определяемый с помощью генератора типа TUPLE с указанием некоторого заголовка кортежа - множества вида `<имя_атрибута, тип_атрибута>`.

Типом атрибута определяемого кортежного типа может являться любой определенный к этому моменту:

- скалярный тип
- кортежный тип
- тип отношения

Значением кортежного типа является кортеж, представляющий собой множество триплетов `<имя_атрибута, тип_атрибута, значение_атрибута>`, которое соответствует заголовку. В БД отдельно хранить кортежи нельзя, обязательно нужны отношения.

Тип отношения – это безымянный тип данных, определяемый с помощью генератора типа RELATION с указанием некоторого заголовка кортежа. Значение типа отношения состоит из заголовка и тела:

- заголовок совпадает с заголовком кортежа этого типа отношения
- тело - это множество кортежей, соответствующих этому заголовку

Кортежные типы и типы отношений не являются инкапсулированными, т.е. имеется возможность прямого доступа к атрибутам. Разница не очень понятна, т.к. и до атрибутов инкапсулированного скалярного типа можно добраться, взяв в качестве возможного представления реальное.

Для всех разновидностей типов данных разработана модель множественного наследования, позволяющая определять новые типы данных на основе уже определенных типов. Модель наследования по Дейту и Дарвену не является частью истинной реляционной модели данных.

Наследование по Дейту и Дарвену - это наследование по ограничению (inheritance by constraint): при определении подтипа какого-то супертипа, то множество значений подтипа содержится во множестве значений супертипа.

Другими словами, при наследовании мы только сужаем множество значений, но никак не можем расширить.

Одиночное наследование

Пусть есть некоторый тип T, от которого порождаются два подтипа: T_1 и T_2. Обозначим $D(A)$ - множество значений типа A. Условия на T_1, T_2:

- $D(T_1) \cup D(T_2) = D(T)$
- $D(T_1) \cap D(T_2) = \varnothing$

Пример. Пусть имеются яблоки, которые нужно разделить на красные и зелёные. Для разделения яблок по цвету, требуется чтобы хотя бы в одном представлении типа *яблоки* был атрибут цвет. В подтипе нельзя определить новый атрибут, если он не выражается с помощью уже существующих атрибутов супертипа.

Множественное наследование

Пример. Пусть люди (супертип) в университете делятся на студентов и преподавателей (два подтипа). Но есть студенты, которые одновременно ещё и преподают, поэтому неплохо было бы иметь тип, который включает всех студентов-преподавателей. С другой стороны необходимо, чтобы тип студенты и преподаватели имели непустое пересечение, потому что именно их пересечение составляют те студенты, которые являются преподавателями.

При множественном наследовании получаем, что должен быть супертип, от которого наследуются несколько подтипов. И если у этих подтипов имеются непустые пересечения, то от них нужно унаследовать подтип, включающий их пересечения. Такое наследование отличается от принятого в языках программирования. Например, нельзя добавить новый атрибут к подтипу, если он не выражается через существующие атрибуты. А также получается, что при множественном наследовании множество значений уменьшается ещё сильнее, чем при одиночном наследовании.

При определении подтипов можно переопределить операции. Например, студенты получают стипендию, преподаватели - зарплату, а студенты-преподаватели и то, и другое. Значит операцию начисления зарплаты для них нужно сделать свою.

Допустим, у студента есть номер комнаты (room number), в которой он живёт. А у преподавателя room number - это номер его кабинета. Что делать со студентом-преподавателем? Можно, например, вообще запретить такое наследование. Второе решение - переименовать в одном из типов атрибут. Такое нежелательно, т.к. эти типы данных уже могут быть использованы в приложениях (надо везде поменять). Третье решение - это переименование одноимённых атрибутов в момент создания подтипа (именно в подтипе; супертипы никак не затрагиваются).

Отложенное связывание

Дейт и Дарвен говорят, что привязывать определения операций к определению типа данных неправильно. Т.к. если мы определяем новый тип данных, то операций,

которые включают значения только этого типа данных гораздо меньше, чем опреаций над разными типами данных.

Например, пусть мы определяем тип integer. Тогда можно определить операцию integer + integer. Но как определить integer + float? Не внутри же класса. Поэтому определять операции надо вне типа.

Далее возникает вопрос какую операцию использовать при полиморфизме. Может быть много операций сложения, и несколько могут подходить одновременно. Как решить какую вызывать?

Дейт и Дарвен говорят, что нужно находить ту реализацию операции, которая наиболее близка к реальным типам аргументов.

При таких определениях типов данных значениями атрибутов отношения могут быть не только значения произвольно сложных скалярных типов, но и отношения.

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 бы с ним Кодд.

Переменная отношения - это именованный объект, значение которого может изменяться с течением времени. Переменная отношения в разное время - это различные таблицы базы данных, у которых разные строки, но одинаковые столбцы.

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, названной Алгеброй А, которая основывается на реляционных аналогах булевских операций *конъюнкции, дизъюнкции и отрицания.*

Ограничения целостности

Стоит отметить, что в истинной РМД нет понятия первичного ключа, его аналогом выступает возможный ключ.

Возможный ключ – это одно из подмножеств заголовка переменной отношения, обладающее свойствами первичного ключа.

В число обязательных требований истинной реляционной модели входит требование определения хотя бы одного возможного ключа для каждой переменной отношения.

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».

Более подробно о правильно построенной формуле будет в следующей лекции.

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)».

Заключение по моделям данных

Кратко рассмотрены особенности трех ранних моделей данных:

- модели инвертированных таблиц
- иерархической модели
- сетевой модели данных

Представлена исходная реляционная модель данных, определенная Эдгаром Коддом. Описаны основные черты трех современных моделей данных, системы типов данных которых позволяют сохранять в БД и обрабатывать данные произвольно сложной структуры:

- ОО-модель данных
- модель данных SQL
- истинно реляционная модель данных

Реляционные алгебра и исчисление

С подачи Кодда в реляционной модели данных имеются два базовых механизма манипулирования реляционными данными:

- Основанная на теории множеств реляционная алгебра.
- Основанное на математической логике реляционное исчисление, базирующееся:
  - на исчислении кортежей
  - на исчислении доменов

Оба механизма обладают одним важным свойством: они замкнуты относительно понятия отношения. Т.е. выражения реляционной алгебры и формулы реляционного исчисления определяются над отношениями реляционных БД, и результатом их «вычисления» также являются отношения.

Поэтому, любое выражение или формула могут интерпретироваться как отношения, что позволяет использовать их в других выражениях или формулах исчисления кортежей.

Кодд доказал, что эти два механизма логически и алгебраически эквивалентны, т.е. для любого выражения реляционной алгебры можно построить эквивалентную формулу реляционного исчисления, и наоборот. Эквивалентность подразумевает, что при интерпретации формулы получится такой же результат, как при реляционном выражении.

Алгебра А Дейта и Дарвена

Рассмотрим базовые операции Алгебры А.

Действующие понятия:

- A - имя атрибута; T - имя типа (домена).
- Атрибут - это упорядоченная пара вида $<A, T>$.
- Заголовок Hr – это множество атрибутов.
- Кортеж tr – это множество упорядоченных триплетов вида $<A, T, v>$. v - конкретное значение.
- Тело Br – это множество кортежей tr.
- Элемент заголовка - это атрибут.
- Элемент тела – это кортеж.
- Элемент кортежа – это упорядоченный триплет вида $<A, T, v>$.
- Любое подмножество (в т.ч. пустое) заголовка – это заголовок.
- Любое подмножество (в т.ч. пустое) тела – это тело.
- Любое подмножество (в т.ч. пустое) кортежа – это кортеж.

Важное отметить, что по определению никакие два атрибута в заголовке не могут содержать одно и то же имя атрибута A.

Реляционное дополнение

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

- $Hs = Hr$
- $Bs = \{ts : \exists tr(tr \notin Br \wedge ts = tr\}$

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.

Пример. В тип данных `ДОПУСТИМЫЕ_ПРОЕКТЫ`, на котором определен атрибут `ПРО_НОМ` отношения `ПРОЕКТЫ`, входит пять значений $\{1, 2, 3, 4, 5\}$. Тогда:

ПРОЕКТЫ

ПРО_НОМ
1
2

<NOT> ПРОЕКТЫ

ПРО_НОМ
3
4
5

Рис. 20: Пример реляционного дополнения

Удаление атрибута

Пусть s обозначает результат операции r<REMOVE>A. Тогда:

- $Hs = Hr \setminus \{<A, T>\}$
- $Bs = \{ts : \exists tr \exists v(tr \in Br \wedge <A, T, v> \in tr \wedge ts = tr \setminus \{<A, T, v>\})\}$

СОТРУДНИКИ

СОТР_НОМЕР	СОТР_ИМЯ	СОТР_ЗАРП	ПРО_НОМ
2934	Иванов	21400	1
2935	Петров	24400	2
2936	Сидоров	19200	1
2937	Федоров	21000	1

СОТРУДНИКИ <REMOVE> ПРО_НОМ

СОТР_НОМЕР	СОТР_ИМЯ	СОТР_ЗАРП
2934	Иванов	21400
2935	Петров	24400
2936	Сидоров	19200
2937	Федоров	21000

Рис. 21: Пример удаления атрибута

Операция <REMOVE> аналогична операции `PROJECT` (проецирования отношения). Отличие в том, что в `PROJECT` указывается подмножество заголовка, которое должно остаться, то здесь указывается атрибут, который будет удалён.

Переименование атрибутов

Пусть s обозначает результат операции r <RENAME> (A, B). Для выполнения операции в схеме отношения r должен присутствовать атрибут A и не должен присутствовать атрибут B. Тогда:

- $Hs = (Hr \setminus \{<A, T>\}) \cup \{<B, T>\}$
- $Bs = \{ts : \exists tr \exists v(tr \in Br \wedge <A, T, v> \in tr \wedge ts = (tr \setminus \{<A, T, v>\}) \cup \{<B, T, v>\})\}$

Операция <RENAME> производит отношение s, отличающееся от отношения r только именем одного его атрибута, меняющегося с A на B.

Заголовок s такой же, как заголовок r, но пара $<B, T>$ меняется на $<A, T>$. Тело s включает все кортежи тела r, но в каждом из этих кортежей триплет $<B, T, v>$ меняется на $<A, T, v>$.

Реляционная конъюнкция

Реляционная конъюнкция - это самая важная операция Алгебры А.

Пусть s обозначает результат операции $r1$ <AND> $r2$. Для выполнения операции требуется, чтобы если в $r1$ и $r2$ имеются одноименные атрибуты, то они должны быть определены на одном и том же типе (домене). Тогда:

- $Hs = Hr1 \cup Hr2$
- $Bs = \{ts : \exists tr1\ \exists tr2(tr1 \in Br1 \wedge tr2 \in Br2 \wedge (ts = tr1 \cup tr2))\}$

Операция реляционной конъюнкции может вести себя по-разному:

- 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.

Лекция 13

Реляционная дизъюнкция

Пусть s обозначает результат операции $r1$ <OR> $r2$. Для выполнения операции требуется, чтобы если в $r1$ и $r2$ имеются одноименные атрибуты, то они должны быть определены на одном и том же типе (домене). Тогда:

- $Hs = Hr1 \cup Hr2$
- $Bs = \{ts : \exists tr1 \exists tr2((tr1 \in Br1 \vee tr2 \in Br2) \wedge ts = tr1 \cup tr2)\}$

ПРОЕКТЫ_1

НАЗВАНИЕ	РУКОВОДИТЕЛЬ
ПРОЕКТ1	Иванов
ПРОЕКТ2	Сидоров

НОМЕРА_ПРОЕКТОВ

НОМЕР
1
2

ПРОЕКТЫ_1 <OR> НОМЕРА_ПРОЕКТОВ

НАЗВАНИЕ	РУКОВОДИТЕЛЬ	НОМЕР
ПРОЕКТ1	Иванов	1
ПРОЕКТ2	Иванов	1
ПРОЕКТ3	Иванов	1
ПРОЕКТ1	Сидоров	1
ПРОЕКТ2	Сидоров	1
ПРОЕКТ3	Сидоров	1
ПРОЕКТ1	Иванов	2
ПРОЕКТ2	Иванов	2
ПРОЕКТ3	Иванов	2
ПРОЕКТ1	Сидоров	2
ПРОЕКТ2	Сидоров	2
ПРОЕКТ3	Сидоров	2
ПРОЕКТ1	Иванов	3
ПРОЕКТ2	Сидоров	3

Рис. 22: Пример реляционной дизъюнкции

На рис. 22 показаны отношения `ПРОЕКТЫ_1` и `НОМЕРА_ПРОЕКТОВ`, а также результат их реляционной дизъюнкции. Для простоты введены ограничения:

- `НАЗВАНИЕ` принимает значения `ПРОЕКТ1, ПРОЕКТ2, ПРОЕКТ3`.
- `РУКОВОДИТЕЛЬ` принимает значения `Иванов, Сидоров`.
- `НОМЕР` принимает значения `1, 2, 3`.

Для номеров 1 и 2 перебираются все 6 возможных комбинаций названия и руководителя. А вот кортежа с номером 3 в `НОМЕРА_ПРОЕКТОВ` нет. Значит, мы можем только к существующим кортежам в `ПРОЕКТЫ_1` добавить номер 3.

Полнота Алгебры А

Вспомним операции алгебры Кодда:

- объединение (`UNION`)
- пересечение (`INTERSECT`)
- вычитание (`MINUS`)
- взятие расширенного декартова произведения (`TIMES`)
- переименование атрибутов (`RENAME`)
- ограничение (`WHERE`)
- проекция (`PROJECT`)
- соединение (`Θ-JOIN`)
- деление (`DIVIDE BY`)
- присваивание

Покажем, что Алгебра A является полной, т.е. на основе введенных операций выражаются все операции алгебры Кодда.

- `UNION` - частный случай <OR>
- `INTERSECT`, `TIMES` - частные случаи <AND>
- `RENAME` присутствует
- `PROJECT` выражается через <REMOVE>
- присваивание присутствует

Покажем, что через операции Алгебры A выражаются операции `MINUS`, `WHERE`, `Θ-JOIN`, `DIVIDE BY`.

Операция MINUS

Аналогично результату из теории множеств, отрицание выражается через пересечение и отрицание. $r1$ `MINUS` $r2 = r1$ <AND> (<NOT>$r2$).

Операция WHERE

Покажем, как можно выразить операцию ограничения `WHERE` с помощью операций Алгебры A. Далее comp-op $\in \{=, \neq, <, \leq, >, \geq\}$.

Условие вида *A* comp-op const

Рассмотрим как выразить на языке Алгебры A операцию `WHERE` с условием вида A comp-op const на примерах. На рис. 23 изображены отношения `СОТРУДНИКИ_1` и `ЗАРП_11000`. Предположим (для упрощения примеров), что множества значений

доменов, на которых определены атрибуты отношения СОТРУДНИКИ_1, ограничены значениями, содержащимися в теле этого отношения.

СОТРУДНИКИ_1

СОТР_НОМЕР	СОТР_ИМЯ	СОТР_ЗАРП	РУК_НОМЕР
2934	Иванов	11400	2934
2935	Петров	14400	2934
2936	Сидоров	9200	2934
2937	Федоров	11000	2934
2938	Иванова	11000	2941
2939	Сидоренко	9200	2941
2940	Федоренко	11000	2941
2941	Иваненко	11000	2941

ЗАРП_11000

СОТР_ЗАРП
11000

Рис. 23: Схемы отношений СОТРУДНИКИ_1 и ЗАРП_11000

Пример. Пусть требуется ограничить содержимое СОТРУДНИКИ_1 условием WHERE СОТР_ЗАРП=11000. В таком случае стоит сделать операцию реляционной конъюнкции СОТРУДНИКИ_1 <AND> ЗАРП_11000. Заметим, что ЗАРП_11000 в данном случае - это специально построенное константное отношение. Результат после реляционной конъюнкции показан на рис. 24.

СОТРУДНИКИ_1 <AND> ЗАРП_11000

СОТР_НОМЕР	СОТР_ИМЯ	СОТР_ЗАРП	РУК_НОМЕР
2937	Федоров	11000	2934
2938	Иванова	11000	2941
2940	Федоренко	11000	2941
2941	Иваненко	11000	2941

Рис. 24: WHERE СОТР_ЗАРП=11000 выражается через операцию <AND>

Пример. Ограничим содержимое СОТРУДНИКИ_1 условием WHERE СОТР_ЗАРП≠11000. Так как атрибут СОТР_ЗАРП принимает значения $\{9200, 11000, 11400, 14400\}$, то для выражения ограничения WHERE СОТР_ЗАРП≠11000 достаточно выполнить операцию <AND> с отношением <NOT> ЗАРП_11000 (рис. 25).

Пример. Из-за конечности допустимых значений СОТР_ЗАРП легко представить и другие условия: СОТР_ЗАРП<11000={9200}, СОТР_ЗАРП>11000={11400,14400}. В общем случае, когда СОТР_ЗАРП принимает не четыре значения, а, например, 2^{32} или любое другое число (но конечное), ситуация аналогична. Таким образом, ограничения вида WHERE A comp-op const выражаются через операцию <AND>, путём построения вспомогательных константных отношений.

СОТРУДНИКИ_1 <AND> <NOT> ЗАРП_11000 <NOT> ЗАРП_11000

СОТР_НОМЕР	СОТР_ИМЯ	СОТР_ЗАРП	РУК_НОМЕР
2934	Иванов	11400	2934
2935	Петров	14400	2934
2936	Сидоров	9200	2934
2939	Сидоренко	9200	2941

СОТР_ЗАРП
9200
11400
14400

Рис. 25: `WHERE СОТР_ЗАРП≠11000` выражается через операцию <AND>

Условие вида A comp-op B

Пример. Рассмотрим ситуацию, когда оба параметра являются атрибутами. Допустим, мы хотим наложить на `СОТРУДНИКИ_1` (рис. 23) ограничение `WHERE СОТР_НОМЕР = РУК_НОМЕР`.

Удалим все атрибуты из `СОТРУДНИКИ_1` кроме `РУК_НОМЕР`. Для этого надо вызвать операцию <REMOVE> для каждого другого атрибута. Результат показан на рис. 26 слева. Оставшийся столбец переименуем в `СОТР_НОМЕР` (результат справа).

РУК_НОМЕР
2934
2941

СОТР_НОМЕР
2934
2941

Рис. 26: Удаляем все столбцы кроме `РУК_НОМЕР` и переименовываем в `СОТР_НОМЕР`

Получившееся отношение пересекаем операцией <AND> с исходным отношением `СОТРУДНИКИ_1`. Результат и полная последовательность операций показаны на рис. 27.

СОТРУДНИКИ_1 <AND> ((СОТРУДНИКИ_1 <REMOVE> СОТР_НОМЕР <REMOVE> СОТР_ИМЯ <REMOVE> СОТР_ЗАРП) <RENAME> (РУК_НОМЕР, СОТР_НОМЕР))

СОТР_НОМЕР	СОТР_ИМЯ	СОТР_ЗАРП	РУК_НОМЕР
2934	Иванов	11400	2934
2941	Иваненко	11000	2941

Рис. 27: `WHERE` $A = B$ выражается через <REMOVE>, <RENAME> и <AND>

Таким образом, мы выразили операцию ограничения `WHERE` с условием вида A comp-op B через операции Алгебры А <REMOVE>, <RENAME> и <AND>.

Пример. Рассмотрим как реализуется ограничение с другими операциями сравнения. Допустим, мы хотим наложить на `СОТРУДНИКИ_1` (рис. 23) ограничение `WHERE СОТР_НОМЕР > РУК_НОМЕР`. Идея аналогична: построим подходящее вспомогательное отношение и выполним операцию <AND>. На рис. 28 показано нужное вспомогательное отношение, с которым нужно сделать операцию <AND>. Обратите внимание, что для его построения не нужно знаний о содержимом отношения `СОТРУДНИКИ_1`, надо знать только область определения атрибута.

Конечно, никто не требует реализовывать эти операции на практике именно таким образом. Здесь важна именно теоретическая возможность выражения операций алгебры Кодда через операции Алгебры А.

СОТР_НОМЕР	РУК_НОМЕР
2935	2934
...	...
2941	2934
2936	2935
...	...
2941	2935
...	...
2941	2940

Рис. 28: В данном случае будет $7 + 6 + 5 + 4 + 3 + 2 + 1 = 28$ подходящих кортежей

Операция Θ-JOIN

Операция Θ-JOIN выражается через TIMES и WHERE, которые обе выражаются через операции Алгебры А:

- TIMES является частным случаем <AND>.
- WHERE выражается через <REMOVE>, <RENAME> и <AND>.

Операция DIVIDE BY

Пусть имеются отношения $r1\{A, B\}$ и $r2\{B\}$. Утверждается, что следующие два выражения эквивалентны:

- $r1$ DIVIDE BY $r2$
- ($r1$ PROJECT A) MINUS ((($r2$ TIMES ($r1$ PROJECT A)) MINUS $r1$) PROJECT A)

Разберём последовательно почему это так:

- Результатом выполнения операции $r1$ PROJECT A является унарное отношение со схемой $\{A\}$, кортежи которого содержат все значения атрибута A из тела отношения $r1$.
- Результат выражения $r2$ TIMES ($r1$ PROJECT A) – это отношение со схемой $\{A, B\}$, в тело которого входят все возможные комбинации значений B в теле отношения $r2$ и атрибута A в теле $r1$.
- В теле результата выражения ($r2$ TIMES ($r1$ PROJECT A)) MINUS $r1$ останутся только кортежи с таким значением атрибута A, которые не должны попасть в результат операции $r1$ DIVIDE BY $r2$.

- Проецируем результат предыдущего выражения на A и получаем множество тех A, которые не принимают всех значений B из второго операнда.
- Выполнение завершающей операции `MINUS` дает желаемый результат.

Таким образом, приведённые выражения эквивалентны. Но нам уже известно, что операции `PROJECT`, `MINUS`, `TIMES` выражаются через операции Алгебры А, а значит и `DIVIDE BY` тоже выражается через операции Алгебры А. Использовав похожие рассуждения на те, что выше, можно убедиться, что следующие выражения эквивалентны:

- $r1$ `DIVIDE BY` $r2$
- ($r1$ <REMOVE> B) <AND> <NOT> (((($r2$ <AND> ($r1$ <REMOVE> B)) <AND> <NOT> $r1$) <REMOVE> B)

Избыточность Алгебры А

Покажем избыточность Алгебры А. Для этого вспомним, что в математической логике полной системой называется та, из которой выводятся все возможные функции алгебры. Вопрос: сколько нужно иметь функций в Алгебре А и каких, чтобы система всё ещё была полной?

Стандартным полным базисом является $\{\neg, \wedge, \vee\}$ (отрицание, конъюнкция, дизъюнкция). Этот набор избыточен, поскольку верны тождества де Моргана:

- $A \wedge B = \neg(\neg A \vee \neg B)$
- $A \vee B = \neg(\neg A \wedge \neg B)$

В математической логике существуют полные базисы из одной функции:

- $sh(A, B) = \neg A \vee \neg B$ - «штрих Шеффера»
- $pi(A, B) = \neg A \wedge \neg B$ - «стрелка Пирса»

Полнота этих базисов доказывается выводом стандартного полного базиса $\{\neg, \wedge, \vee\}$. Покажем полноту базиса из штриха Шеффера:

- $sh(A, A) = \neg A$
- $sh(\neg A, \neg B) = A \vee B$
- $\neg sh(A, B) = A \wedge B$

Таким образом, базис из одного штриха Шеффера является полным. Аналогичные рассуждения справедливы и для стрелки Пирса:

- $pi(A, A) = \neg A$
- $\neg pi(A, B) = A \vee B$
- $pi(\neg A, \neg B) = A \wedge B$

Нам интересны эти факты о полноте потому, что все эти формулы справедливы и для Алгебры A. Покажем, что из реляционного штриха Шеффера <sh> выводятся <AND>, <OR> и <NOT>.

Реляционный аналог штриха Шеффера

Пусть s обозначает результат операции <sh>$(r1, r2)$. Тогда:

- $Hs = Hr1 \cup Hr2$
- $Bs = \{ts : \exists tr1 \exists tr2((tr1 \notin Br1 \vee tr2 \notin Br2) \wedge (ts = tr1 \cup tr2))\}$

При $r1 = r2 = r$ получим определение, идентичное операции <NOT>:

- $Hs = Hr$
- $Bs = \{ts : \exists tr(tr \notin Br \wedge ts = tr)\}$

Таким образом, из <sh> мы вывели <NOT>.

Далее рассмотрим выражение <sh>(<NOT>$r1$,<NOT>$r2$). Из-за того, что аргументы с отрицанием, в определении <sh>$(r1, r2)$ $tr1 \notin Br1$ и $tr2 \notin Br2$ поменяются на $tr1 \in Br1$ и $tr2 \in Br2$:

- $Hs = Hr1 \cup Hr2$
- $Bs = \{ts : \exists tr1 \exists tr2((tr1 \in Br1 \vee tr2 \in Br2) \wedge (ts = tr1 \cup tr2))\}$

Т.е. мы получили операцию <OR> из <sh>.

Наконец, рассмотрим выражение <NOT><sh>$(r1, r2)$. Опять же посмотрим как изменится определение <sh>$(r1, r2)$ из-за предшествующего отрицания. По определению <NOT> условие $(tr1 \notin Br1 \vee tr2 \notin Br2)$ поменяется на противоположное $(tr1 \in Br1 \wedge tr2 \in Br2)$:

- $Hs = Hr1 \cup Hr2$
- $Bs = \{ts : \exists tr1 \exists tr2((tr1 \in Br1 \wedge tr2 \in Br2) \wedge (ts = tr1 \cup tr2))\}$

И мы получаем в точности определение операции <AND> (тут мы воспользовались тождеством де Моргана).

Таким образом, три операции <AND>, <OR> и <NOT> можно заменить одним реляционным штрихом Шеффера <sh>. И теперь в нашем арсенале осталось три операции: <sh>, <RENAME>, <REMOVE> (и операция присваивания результата, само собой).

Аналогично можно вывести <AND>, <OR> и <NOT> из реляционного аналога стрелки Пирса <pi>, но мы на этом останавливаться не будем.

Избыточность <RENAME>

Оказывается, что операция переименования <RENAME> является логически избыточной и выводится через <sh> и <REMOVE>, оставляя всего две базовые операции.

На рис. 29 показано отношение СОТРУДНИКИ, в котором мы хотим переименовать атрибут ПРО_НОМ на НОМЕР_ПРОЕКТА.

СОТРУДНИКИ

СОТР_НОМЕР	СОТР_ИМЯ	СОТР_ЗАРП	ПРО_НОМ
2934	Иванов	11400	1
2935	Петров	14400	1
2936	Сидоров	9200	1
2937	Федоров	11000	1
2938	Иванова	11000	2
2939	Сидоренко	9200	2
2940	Федоренко	11000	2
2941	Иваненко	11000	2

ПРО_НОМ_НОМЕР_ПРОЕКТА

ПРО_НОМ	НОМЕР_ПРОЕКТА
1	1
2	2
3	3
4	4
5	5

Рис. 29: Вспомогательное отношение ПРО_НОМ_НОМЕР_ПРОЕКТА

Пусть атрибут ПРО_НОМ принимает значения на домене $\{1, 2, 3, 4, 5\}$. На первом шаге создадим вспомогательное отношение ПРО_НОМ_НОМЕР_ПРОЕКТА с кортежами вида $\{k, k\}$ всей области определения ПРО_НОМ (рис. 29). К результату применим операцию реляционной конъюнкции <AND> и получим расширенное отношение, где в каждом кортеже появился новый атрибут НОМЕР_ПРОЕКТА, со значением равным ПРО_НОМ (рис. 30).

СОТРУДНИКИ <AND> ПРО_НОМ_НОМЕР_ПРОЕКТА

СОТР_НОМЕР	СОТР_ИМЯ	СОТР_ЗАРП	ПРО_НОМ	НОМЕР_ПРОЕКТА
2934	Иванов	11400	1	1
2935	Петров	14400	1	1
2936	Сидоров	9200	1	1
2937	Федоров	11000	1	1
2938	Иванова	11000	2	2
2939	Сидоренко	9200	2	2
2940	Федоренко	11000	2	2
2941	Иваненко	11000	2	2

Рис. 30: Расширенное отношение

Осталось только удалить старый атрибут ПРО_НОМ, и мы получили эквивалент операции <RENAME>.

Таким образом, мы выразили <RENAME> через <AND> и <REMOVE>. Получается, что в нашем базисе осталось всего две функции: <sh> и <REMOVE>.

Очевидно, <sh> нельзя выразить через <REMOVE>, но можно ли наоборот? Оказывается, что операцию удаления исключить нельзя, т.к. она уменьшает размерность операнда, а <sh> (или <pi>) нет (т.е. у них свойства разные).

Реляционное исчисление

Допустим, в БД есть отношения `СЛУЖАЩИЕ {СЛУ_НОМ,СЛУ_ИМЯ,СЛУ_ЗАРП,ПРО_НОМ}` и `ПРОЕКТЫ {ПРО_НОМ,ПРОЕКТ_РУК,ПРО_ЗАРП}`. Требуется узнать номера и имена служащих, руководящих проектами с `ПРО_ЗАРП > 18000`. На языке алгебры Кодда запрос выглядит следующим образом:

```
(СЛУЖАЩИЕ JOIN ПРОЕКТЫ WHERE
(СЛУ_ИМЯ = ПРОЕКТ_РУК AND ПРО_ЗАРП > 18000)) PROJECT (СЛУ_ИМЯ, СЛУ_НОМ)
```

А вот данный запрос на языке реляционного исчисления:

```
СЛУЖАЩИЙ.СЛУ_ИМЯ, СЛУЖАЩИЙ.СЛУ_НОМ WHERE EXISTS
(СЛУЖАЩИЙ.СЛУ_ИМЯ = ПРОЕКТ.ПРОЕКТ_РУК AND ПРОЕКТ.ПРО_ЗАРП > 18000)
```

Тут мы оперируем переменными `СЛУЖАЩИЙ` и `ПРОЕКТ`. Прочитать можно так: «Выдать значения `СЛУ_ИМЯ` и `СЛУ_НОМ` для каждого такого кортежа служащих, что существует кортеж проектов со значением `ПРОЕКТ_РУК = СЛУ_НОМ` и значением `ПРО_ЗАРП` большим 18000».

Мы указали характеристики результирующего отношения, но ничего не сказали о способе его формирования. В этом случае система сама должна решить, какие операции и в каком порядке нужно выполнить над отношениями `СЛУЖАЩИЕ` и `ПРОЕКТЫ`.

Обычно говорят, что алгебраическая формулировка является процедурной, т.е. задающей последовательность действий для выполнения запроса, а логическая – описательной (или декларативной), поскольку она описывает свойства желаемого результата. Эти два механизма эквивалентны.

В основе реляционного исчисления лежат понятия переменной и правильно построенной формулы (Well-formed formula или WFF). В зависимости от того, что является областью определения переменной, различают исчисление кортежей и исчисление доменов. В исчислении кортежей допустимым значением переменной является кортеж некоторого отношения. В исчислении доменов допустимым значением переменной является значение некоторого домена.

Реляционное исчисление кортежей

Для определения кортежной переменной используется оператор `RANGE`. Пример определения кортежной переменной `СЛУЖАЩИЙ` на отношении `СЛУЖАЩИЕ`:

```
RANGE СЛУЖАЩИЙ IS СЛУЖАЩИЕ
```

При использовании кортежных переменных в формулах можно ссылаться на значение атрибута переменной (например, `СЛУЖАЩИЙ.СЛУ_ИМЯ`).

Правильно построенные формулы

Правильно построенная формула (Well-Formed Formula, WFF) служит для выражения условий, накладываемых на кортежные переменные. WFF принимает два значения: или true, или false.

Основой WFF являются простые сравнения, представляющие собой операции сравнения ($>, \geq, <, \leq, =, !=$) значений атрибутов переменных или литерально заданных констант. Пример простых сравнений:

```
СЛУЖАЩИЙ.СЛУ_НОМ = 2934
СЛУЖАЩИЙ.СЛУ_НОМ = ПРОЕКТ.ПРОЕКТ_РУК
СЛУЖАЩИЙ.СЛУ_ЗАРП > 18000
```

Простое сравнение является WFF. Заключённая в скобки WFF является простым сравнением.

Более сложные варианты WFF строятся с помощью логических операций NOT, AND, OR и IF ... THEN. С приоритетом операций NOT $>$ AND $>$ OR и возможностью расстановки скобок. Пусть f1,f2 – WFF, тогда следующие формулы тоже являются WFF:

- NOT f1
- f1 AND f2
- f1 OR f2
- IF f1 THEN f2

Пример WFF с использованием операций:

```
IF СЛУЖАЩИЙ.СЛУ_ИМЯ = 'Иванов'
THEN (СЛУЖАЩИЙ.СЛУ_ЗАРП >= 22400.00 AND СЛУЖАЩИЙ.ПРО_НОМ = 1)
```

Правильно построенные формулы нужны для того, чтобы с их помощью реализовывать запросы к БД. Например, для реализации запроса выше нужно пройтись по всем кортежам отношения `СЛУЖАЩИЕ` и применить WFF к каждому кортежу. Результатом запроса будет множество тех кортежей, для которых WFF равна true.

Рассмотрим WFF с двумя переменными: `СЛУЖАЩИЙ.СЛУ_ИМЯ = ПРОЕКТ.ПРОЕКТ_РУК`. Переменных в данной формуле две, значит обходим кортежи не одного отношения, а декартово произведение двух отношений. Эквивалентный запрос в алгебре Кодда: `СЛУЖАЩИЕ JOIN ПРОЕКТЫ WHERE СЛУ_ИМЯ = ПРОЕКТ_РУК`.

Одной из реализаций может быть следующая. Сделаем внешний цикл - обход по кортежам отношения `СЛУЖАЩИЕ` и внутренний цикл - обход по проектам. Применим WFF к каждой паре `<кортеж СЛУЖАЩИЕ, кортеж ПРОЕКТЫ>`. В результат запишем те пары, на которых WFF равно true. Такой метод реализации называется методом вложенных циклов (nested loops join).

Лекция 14

Метод вложенных циклов подходит для условий произвольного вида. Однако на практике чаще всего соединение происходит по равенству аргументов (эквисоединение). Рассмотрим два алгоритма, которые оптимизированы под эквисоединения: sort match и hash join.

Алгоритмы эквисоединения Sort match и Hash match

Рассмотрим сначала алгоритм Sort match. Пусть задана WFF с двумя переменными: `СЛУЖАЩИЙ.СЛУ_ИМЯ = ПРОЕКТ.ПРОЕКТ_РУК`. Отсортируем кортежи `СЛУЖАЩИЕ` по возрастанию `СЛУ_ИМЯ`, а кортежи `ПРОЕКТЫ` по `ПРОЕКТ_РУК`. Получим два отсортированных списка кортежей. Возьмём первый кортеж из служащих и первый кортеж из проектов. Далее возможна одна из трёх ситуаций:

- Если `СЛУ_ИМЯ<ПРОЕКТ_РУК`, то в служащих переходим к следующему кортежу
- Если `СЛУ_ИМЯ=ПРОЕКТ_РУК`, то заносим пару кортежей в результат и в проектах переходим к следующему
- Если `СЛУ_ИМЯ>ПРОЕКТ_РУК`, то в проектах переходим к следующему кортежу

Сложность алгоритма получается линейной O(n+m) + сложность сортировки. Недостаток этого алгоритма заключается в необходимости сортировки данных. Для больших объёмов данных алгоритмов быстрой сортировки нет. Однако даже с сортировкой это быстрее чем O(nm).

Рассмотри алгоритм Hash match. Алгоритм получает на вход два множества кортежей и условие соединения. Результатом его работы является таблица с результатами соединения. Меньшее из двух входных множеств помещается в специальную структуру данных в памяти: хеш-таблицу, которая обеспечивает очень высокую скорость поиска. Затем для каждого кортежа большего множества выполняется поиск значений, соответствующих условию соединения. Результаты помещаются в выходную таблицу. При относительно небольшом размере меньшего множества кортежей это самый эффективный алгоритм для эквисоединения.

Кванторы

При построении WFF допускается использование квантора существования `EXISTS` и квантора всеобщности `FORALL`. Пусть `var` - кортежная переменная, а `form` - правильно построенная формула (WFF). Тогда следующие формулы являются WFF:

- `EXISTS var (form)` - принимает значение true, если в области определения переменной var найдется хотя бы один кортеж, для которого `form` принимает значение true.

- `FORALL var (form)` - принимает значение true, если для всех значений переменной `var` из её области определения `form` принимает значение true.
- В остальных случаях обе формулы равны false.

Переменные, входящие в WFF, могут быть свободными или связанными. Все переменные в WFF без кванторов являются свободными. Если же квантор был использован: `EXISTS var (form)` или `FORALL var (form)`, то в `form` переменная `var` является связанной. Связанная переменная не видна за пределами `form`.

Пример. Пусть `СЛУ1` и `СЛУ2` представляют собой две кортежные переменные, определенные на отношении `СЛУЖАЩИЕ`.

```
EXISTS СЛУ2 (СЛУ1.СЛУ_ЗАРП > СЛУ2.СЛУ_ЗАРП)
```

Данная WFF для текущего кортежа переменной `СЛУ1` принимает значение true только в том случае, если во всем отношении `СЛУЖАЩИЕ` найдется кортеж (ассоциированный со `СЛУ2`) со значением атрибута `СЛУ_ЗАРП` удовлетворяло внутреннему условию сравнения. Эта формула принимает значение true только на служащих, не получающих минимальную зарплату.

```
FORALL СЛУ2 (СЛУ1.СЛУ_ЗАРП >= СЛУ2.СЛУ_ЗАРП)
```

Данная WFF для текущего кортежа переменной `СЛУ1` принимает значение true только в том случае, если для всех кортежей отношения `СЛУЖАЩИЕ` (ассоциированных со `СЛУ2`) значения атрибута `СЛУ_ЗАРП` удовлетворяют условию сравнения. Эта формула принимает значение true только для служащих, получающих максимальную зарплату.

В обоих формулах переменная `СЛУ2` является связанной (кванторы существования и всеобщности),`СЛУ1` - свободной (нет кванторов).

Наиболее очевидный способ интерпретации обеих обсуждавшихся выше формул следующий:

- Просматривать область определения свободной кортежной переменной `СЛУ1`.
- Для каждого очередного кортежа из области определения `СЛУ1` просматривать область определения связанной переменной `СЛУ2` до тех пор, пока не будет установлено истинностное значение формулы для данного кортежа `СЛУ1`:
  - в случае наличия квантора существования процесс просмотра для `СЛУ2` можно остановить после нахождения первого кортежа, для которого значением подформулы, находящейся под знаком квантора, станет true
  - при наличии квантора всеобщности необходимо просмотреть всю область определения СЛУ2)

Мы снова получаем два цикла, как и при интерпретации WFF с двумя свободными переменными.

Одна и та же переменная СЛУ2 может быть использована несколько раз под разными кванторами (или вложенными кванторами). К ним нужно относиться, как к разным переменным. Например:

```
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
- второй при вычислении части с квантором всеобщности

Кванторы можно трактовать как булевские функции (функции, принимающие значения true или false) над множеством значений связанной кортежной переменной. Можно ввести в реляционное исчисление числовые функции над множествами:

- `MIN` (минимальное значение)
- `MAX` (максимальное значение)
- `AVG` (среднее значение) и т.д.

В этом случае можно было бы написать следующую WFF:

```
СЛУ1.СЛУ_ЗАРП > MIN СЛУ2.СЛУ_ЗАРП (СЛУ1.ПРО_НОМ = СЛУ2.ПРО_НОМ)
```

В области истинности которой (множество всех кортежей, удовлетворяющих WFF) содержатся все кортежи отношения `СЛУЖАЩИЕ`, соответствующие тем служащим, которые получают заработную плату, превышающую минимальную зарплату служащих, участвующих в том же проекте.

Целевые списки и выражения реляционного исчисления

WFF обеспечивают средства формулировки условия выборки из отношений базы данных. Чтобы можно было использовать исчисление для реальной работы с БД, требуется целевой список (target list), который определяет набор и имена атрибутов результирующего отношения. Целевой список строится из целевых элементов, каждый из которых может представляться одним из следующих способов:

- `var.attr`, где `var` – имя свободной переменной соответствующей WFF, а `attr` – имя атрибута отношения, на котором определена переменная `var` (например, `СЛУЖАЩИЙ.СЛУ_НОМ`)
- `var`, что эквивалентно указанию всех атрибутов определяющего отношения: `var.attr1, var.attr2, ..., var.attrn`

- `new_name = var.attr`, где `new_name` – переименование соответствующего атрибута в результирующем отношении (требуется когда в WFF используется несколько свободных переменных с одинаковой областью определения)

Применение целевого списка к области истинности WFF эквивалентно действию алгебраической операции `PROJECT`, а последний из приведенных вариантов представляет собой некоторую разновидность операции `RENAME`.

Выражением реляционного исчисления кортежей называется конструкция вида `target_list WHERE WFF`. В качестве примера использования целевого списка покажем формулу, эквивалентную 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 = НОМЕР_ПРОЕКТА.ПРО_НОМ)
```

Заключение

Почему важен аппарат реляционного исчисления кортежей? Во-первых, этот аппарат важен, т.к. это изобретение Эдгара Кодда. Во-вторых, в истории БД существовал язык Quel, который целиком основывался на реляционном исчислении кортежей. Этот язык был реализован в проекте Ingres. В некоторых аспектах язык Quel удобнее языка SQL, но тем не менее победил SQL. Скорее всего это связано с тем, что SQL - разработка IBM, а Quel - университетская разработка. Ресурсы компании IBM несравнимо больше. Вероятно, именно из-за маркетинга SQL вышел в лидеры.

Реляционное исчисление доменов

В исчислении доменов областью определения переменных являются не отношения, а домены. Применительно к базе данных `СЛУЖАЩИЕ-ПРОЕКТЫ` можно говорить, например, о доменных переменных `СЛУ_ИМЯ` и `СЛУ_НОМЕР`.

В исчислении доменов также есть понятие правильно построенной формулы (WFF). Все правила построения точно такие же, как и в исчислении кортежей. Однако использовать исчисление доменов как язык доступа к БД нельзя. Например, с квантором `FORALL` областью истинности WFF является подмножество декартова произведения доменов (домен - это вся область определения; домены хоть конечны, но огромны). На помощь приходят условия членства.

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-мерное отношение с заголовком $\{< X_1, T_1 >, < X_2, T_2 >, \ldots, < X_n, T_n >\}$, то $R(X_{i1} : v_{i1}, X_{i2} : v_{i2}, \ldots, X_{im} : v_{im})(m \leq n, \{Xij\} \subset \{Xk\})$ является условием членства, где v_{ij} - либо литерально задаваемая константа, либо имя некоторой 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 a_{ij} накладывается жёсткое условие, не зависящее от текущих значений доменных переменных.
- Если v_{ij} – имя доменной переменной, то условие членства может принимать разные значения при разных значениях этой переменной.

Пример. Будем считать, что мы определили доменные переменные, имена которых совпадают с именами атрибутов отношения СЛУЖАЩИЕ.

```
СЛУЖАЩИЕ (СЛУ_НОМ:2934, СЛУ_ИМЯ:'Иванов', СЛУ_ЗАРП:22400, ПРО_НОМ:1)
```

Эта формула примет значение true в том и только в том случае, когда в теле отношения СЛУЖАЩИЕ содержится кортеж <2934, 'Иванов', 22400, 1>. Соответствующие значения доменных переменных образуют область истинности этой WFF.

```
СЛУЖАЩИЕ (СЛУ_НОМ:2934, СЛУ_ИМЯ:'Иванов', СЛУ_ЗАРП:22400, ПРО_НОМ:ПРО_НОМ)
```

Данная WFF примет значение true для кортежей с заданными значениями доменных переменных и любым значением ПРО_НОМ.

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 запрос «Выдать номера и имена служащих, не получающих минимальную заработную плату»:

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

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

Заключение по реляционному исчислению

Мы две реляционные алгебры: алгебру Кодда и алгебру А. С формальной точки зрения можно было бы обойтись одним из вариантов, поскольку их выразительные средства эквивалентны.

В алгебре Кодда базовыми операциями являются переименование атрибутов, объединение, пересечение, взятие разности, декартово произведение, проекция и ограничение. Операция соединения общего вида, хотя и включается в алгебру, является вторичной и явно представляется через другие операции. Фундаментальная же в реляционном подходе операция естественного соединения выражается через соединение общего вида и в алгебру не включается. В терминах алгебры Кодда проще всего определяются алгебраические черты языка SQL, в частности общая семантика оператора `SELECT`.

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, 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.

На примере реляционного исчисления видна возможность декларативной логической формулировки запросов. В этом случае выполнение запроса происходит путем интерпретации логической формулы, а не вычисления алгебраического выражения. Были рассмотрены два варианта реляционного исчисления, первый из которых – реляционное исчисление кортежей – был определен сравнительно полно, а для второго – реляционного исчисления доменов – были только отмечены и проиллюстрированы основные отличительные черты.

Проектирование РБД на основе учета FD

При проектировании БД надо ответить на два основных вопроса:

- Каким образом отобразить объекты предметной области в объекты модели данных, чтобы это отображение не противоречило семантике предметной области и было, по возможности, наилучшим (эффективным, удобным и т.д.)? Это называется *проблемой логического проектирования БД.*
- Каким образом в конкретной СУБД, расположить данные во внешней памяти? Какие дополнительные структуры (например, индексы) потребовать и т.д.? Это называется *проблемой физического проектирования БД.*

В случае реляционных баз данных трудно предложить какие-либо общие рецепты по части физического проектирования. Здесь слишком многое зависит от используемой СУБД. Поэтому мы ограничимся вопросами логического проектирования реляционных баз данных, которые существенны при использовании любой реляционной СУБД.

Более того, мы не будем касаться очень важного аспекта проектирования – определения ограничений целостности общего вида.

Например стоит ли хранить в отношении `ОТДЕЛЫ` среднюю заработную плату отдела и поставить ограничение, которое будет контролировать чтобы это средняя ЗП совпадала с действительной. Дело в том, что при использовании СУБД с развитыми механизмами ограничений целостности (например, SQL-ориентированных систем) трудно предложить какой-либо универсальный подход к определению ограничений целостности.

Эти ограничения могут иметь произвольно сложную форму, и их формулировка пока относится скорее к области искусства, чем инженерного мастерства. Самое большее, что предлагается по этому поводу в литературе, это автоматическая проверка непротиворечивости набора ограничений целостности.

Поэтому будем считать, что проблема проектирования РБД состоит в принятии решении из каких отношений должна состоять БД и какие атрибуты должны быть у этих отношений.

Мы рассмотрим классический подход, при котором весь процесс проектирования БД 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.

Каждой нормальной форме соответствует определенный набор ограничений. Будем говорить, что отношение находится в некоторой нормальной форме, если удовлетворяет свойственному ей набору ограничений.

Примером может служить ограничение первой нормальной формы – значения всех атрибутов всех отношений атомарны. Поскольку требование первой нормальной формы является базовым требованием классической РМД, то будем считать что исходные отношения уже в первой нормальной форме.

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 форма 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 - декомпозиции отношения, находящегося в предыдущей нормальной форме, на два или более отношений, которые удовлетворяют требованиям следующей нормальной формы.

Функциональная зависимость

Понятие функциональной зависимости (Functional Dependency – FD) очень важно для проектирования РБД, поэтому стоит обсудить соответствующие теоретические вопросы.

Среди этих вопросов наибольший интерес представляют

- замыкания и покрытия множеств функциональных зависимостей
- аксиомы Армстронга
- теорема Хита о достаточном условии декомпозиции отношения без потерь

Вспомним понятие переменной отношения. Переменная отношения - это именованный объект, значение которого может изменяться с течением времени. Переменная отношения в разное время - это различные таблицы базы данных, у которых разные строки, но одинаковые столбцы.

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). Обозначение $r.X \to r.Y$.

На рис. 31 изображено отношение `СЛУЖАЩИЕ_ПРОЕКТЫ {СЛУ_НОМ, СЛУ_ИМЯ, СЛУ_ЗАРП, ПРО_НОМ, ПРО_РУК}`. `СЛУ_НОМ` является первичным ключом отношения `СЛУЖАЩИЕ`, поэтому для этого отношения выполняется `FD СЛУ_НОМ` $\to$ `СЛУ_ИМЯ`.

СЛУЖАЩИЕ_ПРОЕКТЫ

СЛУ_НОМ	СЛУ_ИМЯ	СЛУ_ЗАРП	ПРО_НОМ	ПРО_РУК
2934	Иванов	22400	1	Иванов
2935	Петров	29000	1	Иванов
2936	Сидоров	23000	1	Иванов
2937	Федоров	20600	1	Иванов
2938	Иванова	22000	2	Иваненко
2939	Сидоренко	18000	2	Иваненко
2940	Федоренко	20400	2	Иваненко
2941	Иваненко	21600	2	Иваненко

Рис. 31: Отношение СЛУЖАЩИЕ_ПРОЕКТЫ

В конкретно данном случае выполняются еще и следующие FD (1):

- СЛУ_НОМ → СЛУ_ИМЯ
- СЛУ_НОМ → СЛУ_ЗАРП
- СЛУ_НОМ → ПРО_НОМ
- СЛУ_НОМ → ПРО_РУК
- {СЛУ_НОМ, СЛУ_ИМЯ} → СЛУ_ЗАРП
- {СЛУ_НОМ, СЛУ_ИМЯ} → ПРО_НОМ
- {СЛУ_НОМ, СЛУ_ИМЯ} → {СЛУ_ЗАРП, ПРО_НОМ}
- ...
- ПРО_НОМ → ПРО_РУК

Поскольку имена всех служащих различны, то выполняются и такие FD (2):

- СЛУ_ИМЯ → СЛУ_НОМ
- СЛУ_ИМЯ → СЛУ_ЗАРП
- СЛУ_ИМЯ → ПРО_НОМ

Более того, для конкретно данных кортежей выполняется и FD (3):

- СЛУ_ЗАРП → ПРО_НОМ

Выполнение каждой из этих FD означает, что по значению левой части формулы можно узнать значение в правой части. Рассмотрим чем отличаются FD (1) от FD (2) и FD (3).

Логично предположить, что СЛУ_НОМ должны быть всегда различны, а у каждого проекта имеется только один руководитель. Поэтому FD (1) должны быть верны для любого допустимого значения переменной отношения СЛУЖАЩИЕ_ПРОЕКТЫ и могут рассматриваться как инварианты, или ограничения целостности этой переменной отношения.

FD (2) основаны на менее естественном предположении о том, что имена всех служащих различны. Это соответствует действительности для примера из рис. 31, но добавим второго Иванова, и FD (2) не будут выполняться.

FD (3) основаны на совсем неестественном предположении, что никакие двое служащих, участвующие в разных проектах, не получают одинаковую зарплату. 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 $A \to B$. Обратите внимание, в отношении `СЛУЖАЩИЕ_ПРОЕКТЫ` наличие FD `ПРО_НОМ` $\to$ `ПРОЕКТ_РУК` приводит к некоторой избыточности этого отношения. Имя руководителя проекта является характеристикой проекта, а не служащего, но в нашем случае содержится в теле отношения столько раз, сколько служащих работает над проектом.

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. Важно уметь сократить набор FD до минимума, поддержка которого гарантирует выполнение всех зависимостей (т.к. чем больше ограничений СУБД должна контролировать, тем медленнее операции изменения БД).

Лекция 15

Тривиальная функциональная зависимость

FD $A \to B$ называется тривиальной, если $B \subseteq A$. Например, `FD {СЛУ_ЗАРП, ПРО_НОМ}` $\to$ `СЛУ_ЗАРП` является тривиальной. Очевидно, любая тривиальная FD всегда выполняется (по определению FD). Частным случаем тривиальной FD является FD $A \to A$. Тривиальные FD не представляют интереса с практической точки зрения. Однако в теоретических рассуждениях их наличие необходимо учитывать.

Практика проектирования БД

Было сказано, что важно найти минимально возможный набор FD, который тем не менее сохраняет нужные свойства у БД (более конкретно об этих свойствах далее). Но как получить этот самый избыточный набор FD?

Обычно это делается группой экспертов предметной области, для которой БД проектируется. Происходит работа с клиентами, и выясняется какая БД им нужна. Допустим, выяснилось, что некоторой организации нужна БД, которая будет использоваться одновременно для бухгалтерии и учёта кадров. Тогда нужно выяснить у отдела кадров и бухгалтерии что им нужно хранить, а также узнать как они отличают кадров между собой (что является первичным ключом). Может быть такое, что у сотрудников просто нет идентификационного номера, и их отличают между собой по `<ФИО, дата рождения>`. После получения этой первичной схемы БД интересен вопрос: «Все ли атрибуты в этой БД можно получить по ключам?». Иначе говоря, нужно функционально вывести всё что возможно. Возникает понятие логически выводимых FD.

Логически выводимые функциональные зависимости

Для понимания логической выводимости рассмотрим два примера.

Пример. Из `СЛУ_НОМ` $\to$ `{СЛУ_ЗАРП, СЛУ_ИМЯ}` выводятся:

- `СЛУ_НОМ` $\to$ `СЛУ_ЗАРП`
- `СЛУ_НОМ` $\to$ `СЛУ_ИМЯ`

Рассмотрим на конкретных значениях. На рис. 31 по значению `СЛУ_НОМ=2934`, мы однозначно находим по таблице пару `{СЛУ_ЗАРП=22400, СЛУ_ИМЯ=Иванов}`. Значит, можно установить зависимость с каждым из атрибутов по отдельности:

- `СЛУ_НОМ=2934` $\to$ `СЛУ_ЗАРП=22400`
- `СЛУ_НОМ=2934` $\to$ `СЛУ_ИМЯ=Иванов`

Пример. Пусть есть две функциональные зависимости:

- `СЛУ_НОМ` → `ПРО_НОМ`
- `ПРО_НОМ` → `ПРОЕКТ_РУК`

Из них логически выводится `СЛУ_НОМ` → `ПРОЕКТ_РУК`. Эта FD называется транзитивной, поскольку `ПРОЕКТ_РУК` зависит от `СЛУ_НОМ` «транзитивно», через атрибут `ПРО_НОМ`.

Транзитивная FD

FD $A \to C$ называется транзитивной, если существует такой атрибут B, что имеются FD $A \to B$ и $B \to C$ и отсутствует явная FD $A \to C$.

Замыкание множества FD

Замыкание множества FD S - это множество всех логически выводимых из S функциональных зависимостей. Обозначается S^+.

Подход к решению проблемы поиска замыкания S^+ множества FD S впервые предложил Вильям Армстронг.

- William Ward Armstrong. Dependency Structures of Data Base Relationships. Proceedings of IFIP Congress 74, 1974, pp. 580-583
- Русский перевод: Вильям Армстронг. “Структуры зависимостей в отношениях баз данных” (ссылка)

Армстронг предложил набор правил вывода новых функциональных зависимостей из существующих (эти правила называются аксиомами Армстронга).

Аксиомы Армстронга

Пусть A, B, C являются (в общем случае, составными) атрибутами отношения R. Множества A, B, C могут иметь непустое пересечение. Обозначим $AC = (A \cup C)$. Тогда:

1) если $B \subseteq A$, то $A \to B$ (рефлексивность)
2) если $A \to B$, то $AC \to BC$ (пополнение)
3) если $(A \to B) \wedge (B \to C)$, то $A \to C$ (транзитивность)

Аксиома рефлексивности

Аксиома рефлексивности истинна, т.к. если $B \subseteq A$, то FD $A \to B$ является тривиальной.

Аксиома пополнения

Докажем от противного. Обозначим $t\{A\} = t$ `PROJECT` A.

Пусть FD $A \to B$ соблюдается, а FD $AC \to BC$ нет. Тогда в теле значения переменной отношения R найдутся два кортежа $t1$ и $t2$, такие, что

- $t1\{AC\} = t2\{AC\}$ (a)
- $t1\{BC\} \neq t2\{BC\}$ (b)

Из $A \subseteq AC$ и (a) по аксиоме рефлексивности следует, что $t1\{A\} = t2\{A\}$. Из FD $A \to B$, следует $t1\{B\} = t2\{B\}$. Из (b) следует, что $t1\{C\} \neq t2\{C\}$. Из $C \subseteq AC$ и (a) по аксиоме рефлексивности следует, что $t1\{C\} = t2\{C\}$.

Исходное предположение привело к противоречию. Вторая аксиома доказана.

Аксиома транзитивности

Докажем от противного.

Пусть FD $A \to B$ и FD $B \to C$ соблюдаются, а $A \to C$ нет. Это означает, что в некотором допустимом теле отношения найдутся два кортежа $t1$ и $t2$, такие, что $t1\{A\} = t2\{A\}$ и $t1\{C\} \neq t2\{C\}$. Из FD $A \to B$ следует, что $t1\{B\} = t2\{B\}$. Из FD $B \to C$ следует, что $t1\{C\} = t2\{C\}$.

Исходное предположение привело к противоречию. Третья аксиома доказана.

Система правил вывода Армстронга полна и совершенна (sound and complete) в том смысле, что для данного множества FD S любая FD, логически 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) $A \to A$ (самодетерминированность)
   - Прямо следует из 1)
5) если $A \to BC$, то $A \to B$ и $A \to C$ (декомпозиция)
   - Из рефлексивности (1): $BC \to B$. Из транзитивности (3): $A \to B$
   - Из рефлексивности (1): $BC \to C$. Из транзитивности (3): $A \to C$
6) если $A \to B$ и $C \to D$, то $AC \to BD$ (композиция)
   - Из аксиомы пополнения (2): $AC \to BC$ и $BC \to BD$
   - Из транзитивности (3): $AC \to BD$
7) если $A \to B$ и $A \to D$, то $A \to BD$ (объединение)
   - Частный случай композиции при $C = A$

8) если $A \to BC$ и $C \to D$, то $A \to BCD$ (накопление)
   - Пополним $C \to D$ с помощью BC: $BC \to BCD$
   - Из транзитивности (3): $A \to BCD$

Замыкание множества атрибутов

Пусть заданы переменная отношения r, множество Z атрибутов этого отношения и некоторое множество FD S, выполняемых для r. Тогда замыканием Z над S называется наибольшее множество Z^+ таких атрибутов Y, что (FD $Z \to Y$) $\in S^+$.

Другими словами, в Z^+ помимо атрибутов Z добавятся новые, функционально зависимые от атрибутов в Z.

Алгоритм вычисления Z^+

```
K=0
Z[0]=Z
DO
  K=K+1
  Z[K] = Z[K-1]
  FOREACH ((FD A→B) IN S)
    IF A ⊆ Z[K] THEN Z[K]=Z[K] ∪ B
UNTIL Z[K] == Z[K-1]
Z+ = Z[K]
```

Данный алгоритм строит Z^+ итерационно (последовательными приближениями). Докажем корректность алгоритма с помощью метода математической индукции.

1) На нулевом шаге `Z[0]=Z` и выполняется тривиальная `FD Z→Z[0]`.
2) Пусть для некоторого числа `K` выполняется `FD Z→Z[K]`.
3) Пусть `FD A→B ∈ S` и `A⊆Z[K]`. Обозначим `AC=Z[K]`. Тогда выполняются `FD A→B` и `FD Z→AC`. Из правила накопления верно, что `FD Z→(Z[K] ∪ B)`. Т.к. `Z[K+1]=Z[K] ∪ B`, то `FD Z→Z[K+1]`. Т.е. мы перешли на следующий шаг индукции.

Т.к. Z конечно, то на некотором шаге алгоритм завершит свою работу.

Пример работы алгоритма

Пусть для примера имеется отношение с заголовком $\{A, B, C, D, E, F\}$ и заданным множеством FD $S = \{A \to D, AB \to E, BF \to E, CD \to F, E \to C\}$. Пусть требуется найти $(AE)^+$ над S.

На первом проходе цикла DO $Z[1] = AE$. Внутри `FOREACH` будут найдены $A \to D$ и $E \to C$. В конце цикла $Z[1] = ACDE$.

На втором проходе цикла DO $Z[2] = ACDE$. Внутри `FOREACH` будет найдена $CD \to F$. В конце цикла $Z[2] = ACDEF$.

В конце следующего прохода цикла DO $Z[3] == Z[2]$, поэтому $(AE)^+ = ACDEF$.

Алгоритм построения замыкания множества атрибутов Z над заданным множеством FD S помогает легко установить, входит ли заданная FD $Z \to B$ в S^+. Необходимым и достаточным условием для этого является $B \subseteq Z^+$, т.е. вхождение составного атрибута B в замыкание Z.

Суперключ отношения

Суперключом переменной отношения r называется любое подмножество K заголовка Hr, включающее, по меньшей мере, хотя бы один возможный ключ r.

Следствием определения является то, что подмножество K заголовка Hr является суперключом $\Leftrightarrow \forall A \subseteq Hr$ верно, что FD $K \to A$.

В терминах замыкания множества атрибутов: K - суперключ $\Leftrightarrow K^+ = Hr$.

Покрытие множества FD

Множество FD $S2$ называется покрытием множества FD $S1$, если любая FD, выводимая из $S1$, выводится также из $S2$. $S2$ является покрытием $S1 \leftrightarrow S1^+ \subseteq S2^+$. Два множества FD $S1$ и $S2$ называются эквивалентными, если $S1^+ = S2^+$.

Минимальное множество FD

Множество FD S называется минимальным (или неприводимым), если:

1) правая часть любой FD из S является простым атрибутом
2) любая FD из S минимальна слева: удаление $\forall$ атрибута из левой части приводит к изменению замыкания S^+
3) удаление любой FD из S приводит к изменению S^+

Здесь очень важно понятие минимальности FD слева («слева» иногда опускается). Другими словами, минимальность FD обозначает, что в детерминанте отсутствует меньшее подмножество атрибутов, из которого можно также вывести данную FD.

Пример. Рассмотрим отношение `СЛУЖАЩИЕ_ПРОЕКТЫ {СЛУ_НОМ, СЛУ_ИМЯ, СЛУ_ЗАРП, ПРО_НОМ, ПРО_РУК}`. Пусть `СЛУ_НОМ` является единственным возможным ключом. Тогда множество FD `{СЛУ_НОМ` $\to$ `СЛУ_ИМЯ, СЛУ_НОМ` $\to$ `СЛУ_ЗАРП, СЛУ_НОМ` $\to$ `ПРО_НОМ, ПРО_НОМ` $\to$ `ПРО_РУК}` является минимальным:

- в правых частях FD этого множества находятся простые атрибуты
- все левые части являются простыми атрибутами
- удаление любой FD явно приводит к изменению замыкания множества FD

Построение минимального множества FD

Для любого множества FD S может быть построено эквивалентное ему минимальное множество S_{-}. Общая схема построения S_{-} следующая.

Правило декомпозиции: $A \to BC$, то $A \to B$ и $A \to C$.

1) Используя декомпозицию, мы можем привести множество S к эквивалентному множеству FD $S1$, правые части FD которого содержат только простые атрибуты.
2) Для любой FD из $S1$ удаляем поочерёдно по одному атрибуту из левой части. Если замыкание изменилось, то атрибут возвращаем. Результирующее множество обозначим за $S2$.
3) Поочерёдно удаляем каждую FD f из $S2$. Оставляем только те f, которые изменяют замыкание. Полученное множество S_{-}является минимальным и эквивалентным исходному.

Приведём пример.

Пусть имеется переменная отношения r с заголовком $\{A, B, C, D\}$ и задано множество FD $S = \{A \to B, A \to BC, AB \to C, AC \to D, B \to C\}$.

После декомпозиции получили эквивалентное множество $S1 = \{A \to B, A \to C, AB \to C, AC \to D, B \to C\}$.

В левой части FD $AC \to D$ можно удалить атрибут C, т.к.

- по правилу пополнения из $A \to C$ следует $A \to AC$
- по аксиоме транзитивности получим $A \to D$.

Но это не всё. Покажем, что из FD $\{A \to C, A \to D\}$ выводится $AC \to D$.

- пополним из $A \to D$ до $AC \to CD$
- декомпозицией $AC \to CD$ получим $AC \to D$

Т.е. эти системы эквивалентны, поэтому можно удалить атрибут C.

Аналогично из левой части $AB \to C$ можно удалить атрибут B.

Удаляем $A \to C$, т.к. она выводится по аксиоме транзитивности из $A \to B$ и $B \to C$.

Построили минимальное эквивалентное множество $S_{-} = \{A \to B, A \to D, B \to C\}$.

Минимальное покрытие множества FD

Минимальным покрытием множества FD S называется любое минимальное множество FD S_{-}, эквивалентное S.

СЛУЖАЩИЕ_ПРОЕКТЫ

СЛУ_НОМ	СЛУ_ИМЯ	СЛУ_ЗАРП	ПРО_НОМ	ПРО_РУК
2934	Иванов	22000	1	Иванов
2941	Иваненко	22000	2	Иваненко

Декомпозиция 1

СЛУ_НОМ	ПРО_НОМ	ПРО_РУК
2934	1	Иванов
2941	2	Иваненко

СЛУ_НОМ	СЛУ_ИМЯ	СЛУ_ЗАРП
2934	Иванов	22000
2941	Иваненко	22000

Декомпозиция 2

СЛУ_ЗАРП	ПРО_НОМ	ПРО_РУК
22000	1	Иванов
22000	2	Иваненко

СЛУ_НОМ	СЛУ_ИМЯ	СЛУ_ЗАРП
2934	Иванов	22000
2941	Иваненко	22000

Рис. 32: Две возможных декомпозиции

Декомпозиция без потерь и функциональные зависимости

Далее рассмотрим проектирование реляционных БД на основе нормализации, т.е. декомпозиции отношения, находящегося в предыдущей нормальной форме, на два или более отношений, удовлетворяющих требованиям следующей нормальной формы.

Декомпозициями без потерь называются те, в результате которых имеется возможность собрать исходное отношение из декомпозированных отношений без потери информации.

На рис. 32 приведён пример двух декомпозиций. В результате первой декомпозиции информация не была потеряна: про каждого служащего можно узнать имя, размер зарплаты, номер выполняемого проекта и имя руководителя проекта.

Вторая декомпозиция не дает возможности получить данные о проекте служащего, поскольку Иванов и Иваненко получают одинаковую зарплату, следовательно, эта декомпозиция приводит к потере информации.

Что же привело к тому, что одна декомпозиция является декомпозицией без потерь, а вторая – нет?

На рис. 33 показан результат естественного соединения получившихся отношений. Как видно, в случае декомпозиции (2), образовались лишние кортежи, которых изначально не было. Именно их наличие связано с потерей исходной информации.

Интуитивно понятно, что это происходит из-за отсутствия функциональных зависимостей `СЛУ_ЗАРП` $\rightarrow$ `ПРО_НОМ` и `СЛУ_ЗАРП` $\rightarrow$ `ПРО_РУК`, но точнее причину потери информации мы объясним несколько позже.

Корректность декомпозиции 1 следует из теоремы Хита, которая показывает достаточное условие декомпозиции без потерь.

Естественное соединение декомпозиции 1

СЛУ_НОМ	СЛУ_ИМЯ	СЛУ_ЗАРП	ПРО_НОМ	ПРО_РУК
2934	Иванов	22000	1	Иванов
2941	Иваненко	22000	2	Иваненко

Естественное соединение декомпозиции 2

СЛУ_НОМ	СЛУ_ИМЯ	СЛУ_ЗАРП	ПРО_НОМ	ПРО_РУК
2934	Иванов	22000	1	Иванов
2934	Иванов	22000	2	Иваненко
2941	Иваненко	22000	1	Иванов
2941	Иваненко	22000	2	Иваненко

Рис. 33: Две возможных декомпозиции

Декомпозиция отношения $R\{A, B, C\}$ на проекции $R1 = R\{A, B\}$ и $R2 = R\{A, C\}$ называется декомпозицией без потерь, если $R = R1$ `NATURAL JOIN` $R2$.

Теорема Хита

Пусть A, B, C - непересекающиеся множества атрибутов отношения $R\{A, B, C\}$. Декомпозиция отношения R на проекции $R1 = R\{A, B\}$ и $R2 = R\{A, C\}$ будет декомпозицией без потерь, если выполняется функциональная зависимость $A \to B$.

Доказательство.

Покажем, что $R \subseteq (R1$ `NATURAL JOIN` $R2)$. Пусть кортеж $(a, b, c) \in B_R$. По определению проекции $(a, b) \in B_{R1}$ и $(a, c) \in B_{R2}$. По определению естественного соединения (a, b, c) содержится в теле $R1$ `NATURAL JOIN` $R2$.

Покажем, что $R \supseteq (R1$ `NATURAL JOIN` $R2)$. Пусть кортеж (a, b, c) содержится в теле $R1$ `NATURAL JOIN` $R2$. Тогда по определению операции естественного соединения $(a, b) \in B_{R1}$ и $(a, c) \in B_{R2}$. Из последнего условия по определению проекции, найдётся такое значение b_1 атрибута B, что $(a, b1, c) \in B_R$. Т.к. выполняется функциональная зависимость $A \to B$, то $b1 = b$. Мы показали включения в обе стороны.

Теорема доказана.

Пример выполнения теоремы Хита виден на рис. 34. Из-за FD `СЛУ_НОМ` $\to$ `ОТД_НОМ` возможна декомпозиция без потерь.

СЛУЖАЩИЕ_ОТДЕЛЫ_ПРОЕКТЫ

СЛУ_НОМ	ОТД_НОМ	ПРО_НОМ
2934	7	1
2934	7	2
2941	13	1
2941	13	2

Декомпозиция без потерь

СЛУ_НОМ	ПРО_НОМ
2934	1
2934	2
2941	1
2941	2

СЛУ_НОМ	ОТД_НОМ
2934	7
2941	13

Рис. 34: Иллюстрация использования теоремы Хита

Минимально зависимые атрибуты

Атрибут B минимально зависит от атрибута A, если выполняется минимальная слева FD $A \to B$ (удаление $\forall$ атрибута из A приводит к изменению замыкания). Например, в отношении СЛУЖАЩИЕ_ПРОЕКТЫ выполняются FD:

- СЛУ_НОМ $\to$ СЛУ_ЗАРП
- {СЛУ_НОМ, СЛУ_ИМЯ} $\to$ СЛУ_ЗАРП

Первая FD является минимальной слева, а вторая - нет.

Диаграммы FD

Рис. 35: Диаграмма отношения СЛУЖАЩИЕ_ПРОЕКТЫ

С помощью диаграмм FD можно представить минимальное множество FD. На рис. 35 видно минимальное множество FD отношения СЛУЖАЩИЕ_ПРОЕКТЫ. В левой части диаграммы все стрелки начинаются с атрибута СЛУ_НОМ, который является единственным возможным ключом. Стрелка от СЛУ_НОМ к ПРО_РУК отсутствует, т.к. эта FD является транзитивной (через ПРО_НОМ) и поэтому не входит в минимальное множество FD.

Минимальные FD и вторая нормальная форма

Рассмотрим «плохое» отношение, при котором возникают так называемые аномалии обновления. Пусть задана переменная отношения СЛУЖАЩИЕ_ПРОЕКТЫ_ЗАДАНИЯ {СЛУ_НОМ, СЛУ_УРОВ, СЛУ_ЗАРП, ПРО_НОМ, СЛУ_ЗАДАН}. Атрибуты СЛУ_УРОВ и СЛУ_ЗАДАН содержат данные о разряде служащего и о задании, которое выполняет служащий в данном проекте. На рис. 36 показано некоторое допустимое тело отношения.

Будем считать, что заработная плата определяется разрядом, и что каждый служащий может участвовать в нескольких проектах, но в каждом проекте он выполняет только одно задание. Тогда единственно возможным ключом этого отношения является составной атрибут {СЛУ_НОМ, ПРО_НОМ}.

На рис. 37 показана диаграмма FD отношения СЛУЖАЩИЕ_ПРОЕКТЫ_ЗАДАНИЯ. Обратим внимание на FD СЛУ_НОМ $\to$ СЛУ_УРОВ и СЛУ_НОМ $\to$ СЛУ_ЗАРП, которые выделены красным цветом. В этих FD в левой части находится атрибут СЛУ_НОМ, который

СЛУЖАЩИЕ_ПРОЕКТЫ_ЗАДАНИЯ

СЛУ_НОМ	СЛУ_УРОВ	СЛУ_ЗАРП	ПРО_НОМ	СЛУ_ЗАДАН
2934	2	22400	1	A
2935	3	29600	1	B
2936	1	20000	1	C
2937	1	20000	1	D
2934	2	22400	2	D
2935	3	29600	2	C
2936	1	20000	2	B
2937	1	20000	2	A

Рис. 36: Ненормализованное отношение `СЛУЖАЩИЕ_ПРОЕКТЫ_ЗАДАНИЯ`

Рис. 37: Диаграмма FD отношения `СЛУЖАЩИЕ_ПРОЕКТЫ_ЗАДАНИЯ`

является частью возможного ключа `{СЛУ_НОМ,ПРО_НОМ}`, из-за чего происходят так называемые аномалии обновления. Под аномалиями обновления понимаются трудности, с которыми приходится сталкиваться при операциях вставки, удаления и модификации кортежей.

Например, при модификации данных мы вынуждены модифицировать несколько кортежей для одного служащего при изменении его разряда. Также при изменении зарплаты у конкретного разряда, прийдётся менять кортежи всех служащих с данным разрядом. Таким образом, хранение избыточных данных - это аномалия модификации данных.

Также аномалией модификации является то, что при нельзя отстранить служащего от единственного проекта. Хотя характерна ситуация, когда между проектами возникают перерывы, не приводящие к увольнению служащих.

При добавлении нового служащего нельзя не включать его в какой-то проект, т.к. `ПРО_НОМ` является частью возможного ключа и не может содержать неопределенных значений.

Как же улучшить ситуацию и исправить перечисленные аномалии обновления? Ответ - сделать декомпозицию без потерь на отношения, которые находятся во второй нормальной форме.

Взглянем ещё раз на диаграмму FD на рис. 37. Т.к. выполняются FD `СЛУ_НОМ` $\rightarrow$ `СЛУ_УРОВ` и `СЛУ_НОМ` $\rightarrow$ `СЛУ_ЗАРП`, то выполняется FD `СЛУ_НОМ` $\rightarrow$ `{СЛУ_ЗАРП, СЛУ_УРОВ}`. Применим теорему Хита для A=`{СЛУ_НОМ}`, B=`{СЛУ_ЗАРП, СЛУ_УРОВ}`, C=`{ПРО_НОМ, СЛУ_ЗАДАН}`. Получаем, что т.к. выполняется FD $A \rightarrow B$, то декомпозиция на рис. 38 является декомпозицией без потерь.

СЛУЖ

СЛУ_НОМ	СЛУ_УРОВ	СЛУ_ЗАРП
2934	2	22400
2935	3	29600
2936	1	20000
2937	1	20000

СЛУЖ_ПРО_ЗАДАН

СЛУ_НОМ	ПРО_НОМ	СЛУ_ЗАДАН
2934	1	A
2935	1	B
2936	1	C
2937	1	D
2934	2	D
2935	2	C
2936	2	B
2937	2	A

Рис. 38: Декомпозиция без потерь на отношения в 2NF

Посмотрим что произошло с аномалиями обновления.

- Чтобы добавить нового служащего, который еще не участвует ни в каком проекте, достаточно добавить его в `СЛУЖ`.
- Если кто-то из служащих прекращает работу над проектом, достаточно удалить соответствующий кортеж из `СЛУЖ_ПРО_ЗАДАН`.
- При увольнении служащего удаляем кортежи из обоих отношений.
- При изменении разряда служащего, нужно изменить один кортеж в `СЛУЖ`.

Таким образом, аномалии обновления разрешились, потому что мы разбили отношение `СЛУЖАЩИЕ_ПРОЕКТЫ_ЗАДАНИЯ` на `СЛУЖ` и `СЛУЖ_ПРО_ЗАДАН`, которые находятся *во второй нормальной форме*.

Вторая нормальная форма (2NF)

Переменная отношения R находится в 2NF тогда и только тогда, когда:

- она находится в первой нормальной форме
- каждый неключевой атрибут минимально функционально зависит от (каждого) возможного ключа R

Неключевым атрибутом называется атрибут, который не входит ни в один возможный ключ.

Как в нашем случае нарушалось определение 2NF? Рассмотрим неключевой атрибут `СЛУ_УРОВ`. По определению 2NF, он должен минимально зависеть от возможного ключа `{СЛУ_НОМ, ПРО_НОМ}`, но на самом деле зависит от его части `СЛУ_НОМ`. Минимальность нарушена.

Переменные отношения `СЛУЖ` и `СЛУЖ_ПРО_ЗАДАН` находятся в 2NF, т.к. все неключевые атрибуты минимально зависят от первичных ключей `СЛУ_НОМ` и `{СЛУ_НОМ, ПРО_НОМ}` соответственно.

Лекция 16

Аномалии обновления из-за транзитивных FD

В приведённом примере обозначенные аномалии обновления были разрешены, но можно и дальше улучшать свойства отношений путём дальнейшей нормализации.

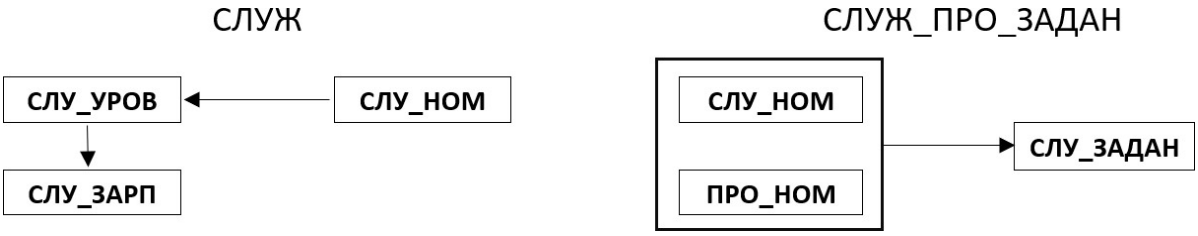


Рис. 39: Диаграмма декомпозированных отношений

Рассмотрим переменную отношения `СЛУЖ` (рис. 39). Зарплата `СЛУ_ЗАРП` хранится в каждом кортеже с одним и тем же разрядом. Это вызывает следующие трудности:

- При модификации ЗП у разряда, необходимо изменить много кортежей вместо одного. Также для изменения разряда служащего, надо в каждом кортеже менять его заработную плату.
- При увольнении последнего служащего с данным разрядом мы утратим информацию о наличии такого разряда и соответствующем размере зарплаты.
- Если мы хотим добавить новый разряд, то обязательно нужен служащий с таким разрядом.

Решение

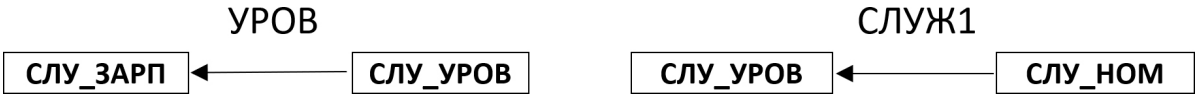


Рис. 40: Декомпозиция `СЛУЖ` на `УРОВ` и `СЛУЖ1`

По теореме Хита (A=`СЛУ_НОМ`, B=`СЛУ_УРОВ`, C=`СЛУ_ЗАРП`) это декомпозиция без потерь. Результирующие отношения находятся в *третьей нормальной форме*. Все аномалии обновления, описанные выше были разрешены

Третья нормальная форма (3NF)

Переменная отношения находится в 3NF тогда и только тогда, когда:

- она находится во второй нормальной форме

- ни один неключевой атрибут не находится в транзитивной функциональной зависимости от возможного ключа

Как видно, до декомпозиции отношения СЛУЖ имело транзитивную FD от возможного ключа СЛУ_НОМ → СЛУ_ЗАРП. Из-за этого нарушалось определение 3NF и возникали аномалии обновления.

Заметим, что отношение СЛУЖ_ПРО_ЗАДАН (справа на рис. 39) уже в 3NF.

Хоть аномалий обновления и разрешились, но не третья нормальная форма их устранила. Ведь мы могли разбить переменную отношения СЛУЖ как показано на рис. 41.

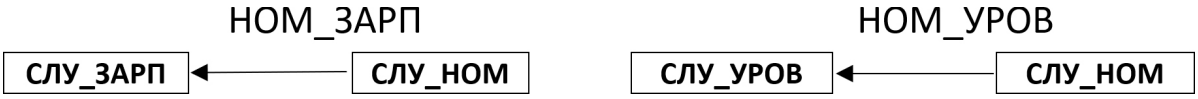


Рис. 41: Плохая декомпозиция СЛУЖ

Это так же декомпозиция без потерь, и результирующие отношения НОМ_ЗАРП и НОМ_УРОВ так же в 3NF, но аномалии не разрешены.

FD СЛУ_УРОВ → СЛУ_ЗАРП трансформируется в ограничение целостности сразу для двух отношений (такого рода ограничения целостности называются ограничениями базы данных, и их поддержка гораздо более накладна с технической точки зрения).

Понятно, что в процессе нормализации декомпозиция отношения на независимые проекции является предпочтительной. Необходимые и достаточные условия независимости проекций отношения обеспечивает теорема Риссанена.

Теорема Риссанена

Проекции $r1$ и $r2$ отношения r являются независимыми тогда и только тогда, когда:

- каждая FD в отношении r логически следует из FD в $r1$ и $r2$,
- общие атрибуты $r1$ и $r2$ образуют возможный ключ хотя бы для одного из этих отношений.

Мы не будем приводить доказательство этой теоремы, но продемонстрируем её верность на примере двух упомянутых ранее декомпозиций отношения СЛУЖ.

Декомпозиция на рис. 41 не удовлетворяет первому условию теоремы. Например, нельзя вывести FD СЛУ_УРОВ → СЛУ_ЗАРП. С другой стороны, декомпозиция на рис. 40 удовлетворяет обоим условиям теоремы Риссанена, а значит проекции являются независимыми.

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:

- FD СЛУ_НОМ → СЛУ_ЗАРП
- FD СЛУ_НОМ → ПРО_НОМ

не является атомарным, т.к. можно разбить на независимые проекции:

- {СЛУ_НОМ, СЛУ_ЗАРП}
- {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.

Перекрывающиеся возможные ключи и нормальная форма Бойса-Кодда

Рассмотрим случай, когда у переменной отношения имеется несколько возможных ключей, и некоторые из этих возможных ключей «перекрываются», т.е. содержат общие атрибуты.

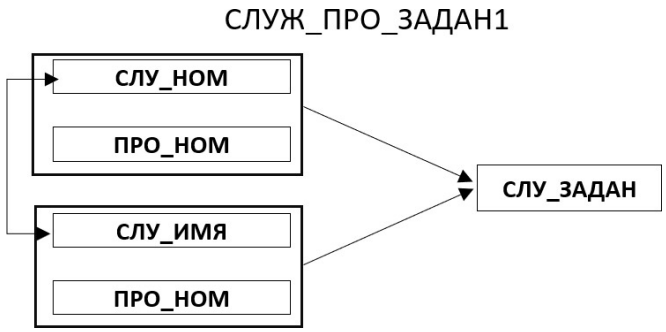


Рис. 42: Диаграмма FD переменной отношения СЛУЖ_ПРО_ЗАДАН1

Пусть задана переменная отношения СЛУЖ_ПРО_ЗАДАН1 с множеством FD, показанным на рис. 42. В этом отношении служащие уникально идентифицируются как по номерам удостоверений, так и по именам. Поэтому существуют FD:

- FD СЛУ_НОМ → СЛУ_ИМЯ
- FD СЛУ_ИМЯ → СЛУ_НОМ

Но один служащий может участвовать в нескольких проектах, поэтому возможными ключами являются

- {СЛУ_НОМ, ПРО_НОМ}

- {СЛУ_ИМЯ, ПРО_НОМ}

СЛУЖ_ПРО_ЗАДАН1 находится в 2NF, поскольку единственный неключевой атрибут СЛУ_ЗАДАН минимально зависит от обоих возможных ключей. Более того, нет транзитивной зависимости СЛУ_ЗАДАН от возможных ключей. Поэтому данная переменная отношения находится в 3NF. Однако аномалии обновления присутствуют.

СЛУЖ_ПРО_ЗАДАН1

СЛУ_НОМ	СЛУ_ИМЯ	ПРО_НОМ	СЛУ_ЗАДАН
2934	Иванов	1	A
2941	Иваненко	2	B
2934	Иванов	2	B
2941	Иваненко	1	A

Рис. 43: Некоторое допустимое тело СЛУЖ_ПРО_ЗАДАН1

На рис. 43 показано некоторое значение переменной отношения СЛУЖ_ПРО_ЗАДАН1. Если мы хотим изменить СЛУ_ИМЯ, то нам требуется изменить все кортежи, соответствующие данному служащему. Другими словами, опять возникает аномалия обновления. Проблему решает нормальная форма, которую исторически принято называть *нормальной формой Бойса-Кодда* (BCNF), и которая является уточнением 3NF в случае наличия нескольких перекрывающихся возможных ключей.

Нормальная форма Бойса-Кодда (BCNF)

Переменная отношения находится в BCNF тогда и только тогда, когда детерминанты всех её функциональных зависимостей являются возможными ключами.

Заметим, что, если в переменной отношения имеется только один возможный ключ, то любое отношение, находящееся в нормальной форме Бойса-Кодда, одновременно находится во второй и третьей нормальных формах Это утверждение доказывается методом «от противного» на основе определений 2NF и 3NF.

НОМ_ИМЯ

СЛУ_НОМ	СЛУ_ИМЯ
2934	Иванов
2941	Иваненко

НОМ_ПРО_ЗАДАН

СЛУ_НОМ	ПРО_НОМ	СЛУ_ЗАДАН
2934	1	A
2941	2	B
2934	2	B
2941	1	A

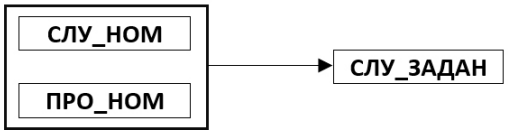


Рис. 44: Декомпозиция СЛУЖ_ПРО_ЗАДАН1 на два отношения в BCNF

Переменная отношения `СЛУЖ_ПРО_ЗАДАН1` может быть приведена к BCNF декомпозицией на рис. 44. Эта декомпозиция является декомпозицией без потерь по теореме Хита при `СЛУ_НОМ` $\rightarrow$ `СЛУ_ИМЯ`.

Всегда ли следует стремиться к BCNF?

Рассмотрим переменную `СЛУЖ_ПРО_ЗАДАН {СЛУ_НОМ, СЛУ_ЗАДАН, ПРО_НОМ}`, в которой:

- у разных проектов не может быть одинакового задания
- каждый служащий может участвовать в нескольких проектах, но может выполнять в каждом проекте только одно задание
- одинаковое задание в каждом проекте могут выполнять несколько служащих

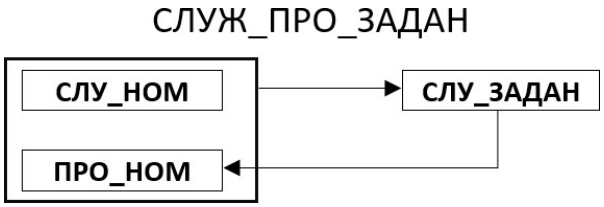


Рис. 45: Диаграмма с возможным ключом `{СЛУ_НОМ, ПРО_НОМ}`

Тогда получаем два возможных ключа: `{СЛУ_НОМ,ПРО_НОМ}` и `{СЛУ_НОМ,СЛУ_ЗАДАН}`. Для определённости первичным ключом выбран `{СЛУ_НОМ, ПРО_НОМ}` (рис. 45). Переменная отношения `СЛУЖ_ПРО_ЗАДАН` удовлетворяет требованиям 2NF и 3NF: отсутствуют не минимальные FD неключевых атрибутов от возможных ключей (поскольку нет неключевых атрибутов) и отсутствуют транзитивные FD. Однако из-за наличия `СЛУ_ЗАДАН` $\rightarrow$ `ПРО_НОМ` это отношение не находится в BCNF. Поэтому присутствуют аномалии обновления. Например, невозможно удалить данные о единственном служащем, выполняющем задание в некотором проекте, не утратив информацию об этом задании.

ЗАДАН_ПРО_НОМ

СЛУ_ЗАДАН	ПРО_НОМ
A	1
B	2
C	1

СЛУ_НОМ_ЗАДАН

СЛУ_НОМ	СЛУ_ЗАДАН
2934	A
2935	A
2936	B

Рис. 46: Декомпозиция `СЛУЖ_ПРО_ЗАДАН` на два отношения в BCNF

На рис. 46 показана декомпозиция на `ЗАДАН_ПРО_НОМ` и `СЛУ_НОМ_ЗАДАН`, которые находятся в BCNF. Возможными ключами являются `СЛУ_НОМ` и `СЛУ_ЗАДАН`. Действительно, теперь можно хранить информацию о заданиях, даже если над ними никто не работает.

Представим, что в отношение `СЛУ_НОМ_ЗАДАН` нам нужно добавить кортеж `<2934, C>`. Тогда получим, что служащий со `СЛУ_НОМ=2934` в первом проекте будет работать

сразу над заданиями A и C (рис. 46). А мы изначально потребовали, чтобы один служащий делал только одно задание в одном проекте. Как же предотвратить добавление этого кортежа? Вспомним теорему Риссанена о независимых проекциях.

Теорема Риссанена. Проекции $r1$ и $r2$ отношения r являются независимыми тогда и только тогда, когда:

- каждая FD в отношении r логически следует из FD в $r1$ и $r2$
- общие атрибуты $r1$ и $r2$ образуют возможный ключ хотя бы для одного из этих отношений

В нашем случае FD `{СЛУ_НОМ, ПРО_НОМ}` $\rightarrow$ `СЛУ_ЗАДАН` не выводится из FD `СЛУ_НОМ` $\rightarrow$ `СЛУ_ЗАДАН` и `СЛУ_ЗАДАН` $\rightarrow$ `ПРО_НОМ`. Т.е. по теореме Риссанена получившиеся проекции не являются независимыми, и это приводит к тому, что ограничение целостности (связанное с кортежом `<2934, C>`) становится ограничением базы данных. Ограничения базы данных вычислительно очень затраты, поэтому всегда нужно внимательно оценивать плюсы и минусы последствий нормализации.

Промежуточные итоги по нормализации

Были рассмотрены три начальные нормальные формы отношений – вторую и третью нормальные формы и нормальную форму Бойса-Кодда, – которые производятся путем декомпозиции без потерь исходного отношения на две проекции, где отсутствуют аномалии изменений, существовавшие в исходном отношении по причине наличия функциональных зависимостей с нежелательными свойствами.

При проектировании реляционной базы данных почти всегда добиваются второй нормальной формы всех входящих в базу данных отношений. В часто обновляемых базах данных обычно стараются обеспечить третью нормальную форму отношений. На нормальную форму Бойса-Кодда внимание обращают гораздо реже, поскольку на практике ситуации, в которых у отношения имеется несколько составных перекрывающихся возможных ключей, встречаются нечасто.

Проектирование РБД: дальнейшая нормализация

Функциональные зависимости и нормальные формы, основанные на учете «аномальных» функциональных зависимостей, являются естественными и легко понимаемыми, поскольку в их основе лежит понятие функционального отображения. Было бы замечательно, если бы ликвидация в ходе нормализации аномальных FD гарантировала отсутствие аномалий обновления отношений.

К сожалению, такой гарантии нет. Иногда в переменных отношений требуется поддержка более сложных ограничений целостности, для выражения которых понятие функции оказывается недостаточным.

Класс зависимостей, опирающихся на понятие функционала (обобщения понятия функции), обнаружил в 1970-е гг. Рональд Фейджин. Он назвал такие зависимости

многозначными, поскольку в них одному значению детерминанта соответствует множество значений зависимого атрибута (в отличие от функции, где только одно значение).

Наличие в переменной отношения многозначных зависимостей, не являющихся FD от возможного ключа, приводит к аномалиям обновления таких отношений. Фейджин показал, что в этом случае возможна декомпозиция таких отношений на две проекции, для которых подобные аномалии обновления не проявляются. Такие проекции находятся в *четвертой нормальной форме* (4NF).

Позже Фейджин установил, что отношениям в 4NF, при наличии некоторых ограничений, свойственны аномалии обновления. Эти аномалии невозможно устранить путем проецирования отношения на две проекции, требуется декомпозиция на три или большее число отношений. Такие ограничения получили название *зависимостей проекции/соединения.*

Отношение, в котором есть нетривиальная зависимость проекции/соединения, может быть декомпозировано на три или большее число проекций, в которых зависимости проекции/соединения следуют из возможного ключа. Такие проекции находятся в *пятой нормальной форме*, или *нормальной форме проекции/соединения.* В отношениях, находящихся в 5NF, отсутствуют аномалии обновления, которые можно было бы устранить путем декомпозиции, и поэтому при достижении 5NF процесс проектирования РБД на основе нормализации естественным образом завершается.

Пример многозначной зависимости

Предположим, что каждый служащий может участвовать в нескольких проектах, но в каждом проекте, в котором он участвует, им должны выполняться одни и те же задания.

СЛУЖ_ПРО_ЗАДАН

СЛУ_НОМ	ПРО_НОМ	СЛУ_ЗАДАН
2934	1	A
2934	1	B
2934	2	A
2934	2	B
2941	1	A
2941	1	D

Рис. 47: Некоторое допустимое значение переменной отношения `СЛУЖ_ПРО_ЗАДАН`

На рис. 47 видно, что служащий с номером 2934 выполняет задания $\{A, B\}$, а служащий 2941 - задания $\{A, D\}$. При данной формулировке единственным возможным ключом является весь заголовок `{СЛУ_НОМ,ПРО_НОМ, СЛУ_ЗАДАН}`. Обозначим x, y, z - значения атрибутов `СЛУ_НОМ`, `ПРО_НОМ` и `СЛУ_ЗАДАН` соответственно. Кортеж $(x, y, z) \in B$ означает, что служащий с номером x выполняет в проекте y задание z.

Поскольку для каждого служащего указываются все проекты, в которых он участвует, и все задания, которые он должен выполнять в этих проектах, для каждого допустимого значения СЛУЖ_ПРО_ЗАДАН должно выполняться следующее ограничение (B - тело отношения).

- Если $(x, y_1, z_1) \in B \wedge (x, y_2, z_2) \in B$,
- то $(x, y_2, z_1) \in B \wedge (x, y_1, z_2) \in B$.

Наличие такого ограничения (как мы скоро увидим, это ограничение порождается наличием многозначной зависимости) приводит к тому, что при работе с отношением СЛУЖ_ПРО_ЗАДАН проявляются аномалии обновления.

Добавление кортежа. Если уже участвующий в проектах служащий присоединяется к новому проекту, то к телу переменной отношения СЛУЖ_ПРО_ЗАДАН требуется добавить столько кортежей, сколько заданий выполняет этот служащий.

Удаление кортежей. Если служащий прекращает участие в проектах, то отсутствует возможность сохранить данные о заданиях, которые он может выполнять.

Модификация кортежей. При изменении одного из заданий служащего необходимо изменить значение атрибута СЛУ_ЗАДАН в стольких кортежах, в скольких проектах участвует служащий.

Решение

Трудности, связанные с обновлением СЛУЖ_ПРО_ЗАДАН, решаются путем его декомпозиции на две переменные отношения:

- СЛУЖ_ПРО_НОМ {СЛУ_НОМ, ПРО_НОМ}
- СЛУЖ_ЗАДАНИЕ {СЛУ_НОМ, СЛУ_ЗАДАН}

Это декомпозиция без потерь (мы докажем это позже с помощью теоремы Фейджина), которая решает перечисленные выше проблемы с обновлением переменной отношения СЛУЖ_ПРО_ЗАДАН.

Добавление кортежа. Если некоторый уже участвующий в проектах служащий присоединяется к новому проекту, то к телу значения переменной отношения СЛУЖ_ПРО_НОМ требуется добавить один кортеж, соответствующий новому проекту.

Удаление кортежей. Если служащий прекращает участие в проектах, то данные о заданиях, которые он может выполнять, остаются в отношении СЛУЖ_ЗАДАНИЕ.

Модификация кортежей. При изменении одного из заданий служащего необходимо изменить значение атрибута СЛУ_ЗАДАН в одном кортеже отношения СЛУЖ_ЗАДАНИЕ.

Заметим, что переменная отношения СЛУЖ_ПРО_ЗАДАН находится в BCNF, поскольку все атрибуты заголовка отношения входят в состав единственно возможного ключа. В этом отношении вообще отсутствуют нетривиальные FD. Поэтому ранее обсуждавшиеся принципы нормализации здесь неприменимы, но, тем не менее, мы получили полезную декомпозицию.

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

В отношении `СЛУЖ_ПРО_ЗАДАН` выполняются две MVD:

- `СЛУ_НОМ` $\to\to$ `ПРО_НОМ`
- `СЛУ_НОМ` $\to\to$ `СЛУ_ЗАДАН`

Первая MVD означает, что каждому значению атрибута `СЛУ_НОМ` соответствует определяемое только этим значением множество значений атрибута `ПРО_НОМ`. Другими словами первой 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`.

Формальное определение MVD

Пусть R - отношение, и A, B, C - непересекающиеся множества его атрибутов. Тогда множество атрибутов B многозначно зависят от A (обозначение $A \to\to B$) тогда и только тогда, когда множество значений атрибута B, соответствующее паре значений атрибутов (A, C), зависит от значения A и не зависит от значения C.

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

Многозначные зависимости обладают свойством «двойственности», которое демонстрирует лемма Фейджина.

Лемма Фейджина

Пусть X, Y, Z - непересекающиеся множества атрибутов отношения $R\{X, Y, Z\}$. В отношении $R\{X, Y, Z\}$ выполняется многозначная зависимость $X \to\to Y$ тогда и только тогда, когда выполняется многозначная зависимость $X \to\to Z$.

Достаточность. Пусть имеется $X \to\to Y$. Надо доказать, что $X \to\to Z$.

Возьмём произвольный кортеж (x_0, y_0, z_0) из тела B_R. Обозначим $Y(x_0), Z(x_0)$ - множества значений атрибутов Y и Z соответственно, взятых из всех кортежей B_R, в которых значением атрибута X является x_0. Предположим, что для значения x_0 атрибута X многозначная зависимость $X \to\to Z$ не выполняется. Тогда существует значение $z \in Z(x_0)$, что для него найдётся $y \in Y(x_0)$, что кортеж $(x0, y, z) \notin B_R$. Но это противоречит многозначной зависимости $X \to\to Y$.

Необходимость. Доказывается аналогично с заменой Y на Z.

Лемма доказана.

Таким образом, MVD $X \to\to Y$ и $X \to\to Z$ всегда составляют пару. Поэтому их представляют вместе в виде $X \to\to Y|Z$ (читается "многозначная зависимость Y или Z от X").

Критерий декомпозиции без потерь сформулирован в теореме Фейджина, которая является обобщением теоремы Хита.

Теорема Фейджина

Пусть A, B, C - непересекающиеся множества атрибутов отношения $R\{A, B, C\}$. Декомпозиция отношения R на проекции $R1 = R\{A, B\}$ и $R2 = R\{A, C\}$ будет декомпозицией без потерь тогда и только тогда, когда выполняется многозначная зависимость $A \to\to B|C$.

Достаточность. Пусть имеется $A \to\to B|C$. Надо доказать, что $R1$ `NATURAL JOIN` $R2 = R$.

Покажем, что $R \subseteq (R1$ `NATURAL JOIN` $R2)$. Пусть кортеж $(a, b, c) \in B_R$. По определению проекции $(a, b) \in B_{R1}$ и $(a, c) \in B_{R2}$. По определению естественного соединения (a, b, c) содержится в теле $R1$ NATURAL JOIN $R2$.

Покажем, что $R \supseteq (R1$ `NATURAL JOIN` $R2)$. Пусть кортеж (a, b, c) содержится в теле $R1$ `NATURAL JOIN` $R2$. Тогда по определению операции естественного соединения $(a, b) \in B_{R1}$ и $(a, c) \in B_{R2}$. По определению проекции, найдётся такое значение c_1 атрибута C, что $(a, b, c_1) \in B_R$. И найдётся такое значение b_1 атрибута B, что $(a, b1, c) \in B_R$. Т.к. выполняется многозначная зависимость $A \to\to B|C$, то $(a, b, c) \in B_R$. Мы показали включения в обе стороны. *Достаточность доказана.*

Необходимость. Пусть декомпозиция отношения R на проекции $R1 = R\{A, B\}$ и $R2 = R\{A, C\}$ является декомпозицией без потерь. Докажем, что $A \to\to B|C$.

Пусть кортежи $r_1 = (a, b, c_1) \in B_R$ и $r_2 = (a, b_1, c) \in B_R$. Надо доказать, что $(a, b, c) \in B_R$. По определению проекции, $(a, b) \in B_{R1}$ и $(a, c) \in B_{R2}$. Тогда (a, b, c) содержится в теле $R1$ `NATURAL JOIN` $R2$, а в силу того, что декомпозиция является декомпозицией без потерь, этот кортеж содержится и в R. *Необходимость доказана.*

Теорема доказана.

Теорема Фейджина обеспечивает основу для декомпозиции отношений, удаляющей «аномальные» многозначные зависимости, с приведением отношений в четвертую нормальную форму.

Четвёртая нормальная форма (4NF)

Переменная отношения r находится в 4NF тогда и только тогда, когда она находится в BCNF, и все многозначные зависимости r являются функциональными зависимости с детерминантами – возможными ключами r.

Обратим внимание, что 4NF является BCNF, в которой многозначные зависимости вырождаются в функциональные.

Отношение `СЛУЖ_ПРО_ЗАДАН` не находится в 4NF, поскольку детерминант (левая часть) MVD `СЛУ_НОМ` $\twoheadrightarrow$ `ПРО_НОМ` и `СЛУ_НОМ` $\twoheadrightarrow$ `СЛУ_ЗАДАН` не является возможным ключом, и эти MVD не являются функциональными. С другой стороны, отношения `СЛУЖ_ПРО_НОМ` и `СЛУЖ_ЗАДАНИЕ` находятся в BCNF и не содержат MVD, отличных от FD с детерминантом – возможным ключом. Поэтому они находятся в 4NF.

Важно отметить, что на практике 4NF практически не используется. Первая причина состоит в том, что получить на практике информацию о том, что нужно поддерживать ограничения многозначной зависимости. Вторая причина в том, что многозначная зависимость на практике - довольно экзотичное явление. Например, составляется некоторая БД по поводу курсов и преподавателей. Любой из преподавателей может вести любой курс из заданного множества и может при подготовке любого курса может использовать одно и то же множество учебников.

До сих пор мы говорили о декомпозиции путём декомпозиции на две проекции. Однако бывают случаи, когда декомпозиция без потерь на две проекции невозможна, но можно произвести декомпозицию без потерь на большее число проекций.

Будем называть отношение n-декомпозируемым, если оно может быть декомпозировано без потерь на n проекций. До сих пор мы имели дело с 2-декомпозируемыми отношениями. Обратим внимание, что любое отношение является 1-декомпозируемым.

Тривиальная многозначная зависимость

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

СЛУ_ПРО_ЗАДАН

СЛУ_НОМ	ПРО_НОМ	СЛУ_ЗАДАН
2934	1	A
2934	1	B
2934	2	A
2941	1	A

СЛУ_ПРО

СЛУ_НОМ	ПРО_НОМ
2934	1
2934	2
2941	1

ПРО_ЗАДАН

ПРО_НОМ	СЛУ_ЗАДАН
1	A
1	B
2	A

СЛУ _ЗАДАН

СЛУ_НОМ	СЛУ_ЗАДАН
2934	A
2934	B
2941	A

Рис. 48: Пример 3-декомпозируемого отношения

На рис. 48 показано отношение СЛУ_ПРО_ЗАДАН, у которого весь заголовок {СЛУ_НОМ, ПРО_НОМ, СЛУ_ЗАДАН} является возможным ключом. На этом же рисунке показаны все три возможные проекции (на два атрибута): СЛУ_ПРО, ПРО_ЗАДАН и СЛУ_ЗАДАН.

Оказывается, что как бы мы не пробовали соединять проекции, восстановить исходное отношение СЛУ_ПРО_ЗАДАН не получится (рис. 49). Т.е. отношение СЛУ_ПРО_ЗАДАН не является 2-декомпозируемым. Однако если соединить все три проекции в любом порядке, то мы получим изначальное отношение. Другими словами, СЛУ_ПРО_ЗАДАН является 3-декомпозируемым отношением.

СЛУ_ПРО NATURAL JOIN СЛУ_ЗАДАН

СЛУ_НОМ	ПРО_НОМ	СЛУ_ЗАДАН
2934	1	A
2934	1	B
2934	2	A
2941	1	A
2934	2	B

СЛУ_ПРО NATURAL JOIN ПРО_ЗАДАН

СЛУ_НОМ	ПРО_НОМ	СЛУ_ЗАДАН
2934	1	A
2934	1	B
2934	2	A
2941	1	A
2941	1	B

СЛУ_ЗАДАН NATURAL JOIN ПРО_ЗАДАН

СЛУ_НОМ	ПРО_НОМ	СЛУ_ЗАДАН
2934	1	A
2934	1	B
2934	2	A
2941	1	A
2941	2	A

Лишний кортеж

Рис. 49: СЛУ_ПРО_ЗАДАН нельзя восстановить по двум проекциям

Утверждение о том, что отношение СЛУ_ПРО_ЗАДАН восстанавливается без потерь путем естественного соединения его проекций СЛУ_ПРО, ПРО_ЗАДАН и СЛУ_ЗАДАН эквивалентно следующему утверждению:

- Если $(a, b) \in B_{\text{СЛУ\_ПРО}}$ и $(b, c) \in B_{\text{ПРО\_ЗАДАН}}$ и $(a, c) \in B_{\text{СЛУ\_ЗАДАН}}$
- то $(a, b, c) \in B_{\text{СЛУ\_ПРО\_ЗАДАН}}$

Это обычное ограничение реального мира, которое для переменной отношения СЛУ_ПРО_ЗАДАН может быть сформулировано на естественном языке следующим образом: «Если служащий с номером a участвует в проекте b, и в проекте b выполняется задание c, и служащий с номером a выполняет задание c, то служащий с номером a в проекте b выполняет задание c». Чтобы восстановить без потерь отношение СЛУ_ПРО_ЗАДАН при любом допустимом теле, должно поддерживаться следующее ограничение:

- Если $\{(a_1, b, c), (a, b_1, c), (a, b, c_1)\} \subseteq B_{\text{СЛУ\_ПРО\_ЗАДАН}}$
- то $(a_1, b_1, c_1) \in B_{\text{СЛУ\_ПРО\_ЗАДАН}}$

В общем виде такое ограничение называется зависимостью проекции/соединения.

Зависимость проекции/соединения

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

Лекция 17

Предположим, что для переменной отношения СЛУ_ПРО_ЗАДАН выполняется PJD* ({СЛУ_НОМ, ПРО_НОМ},{ПРО_НОМ, СЛУ_ЗАДАН},{СЛУ_НОМ, СЛУ_ЗАДАН}) (рис. 48). Наличие такой PJD обеспечивает возможность декомпозиции этой переменной отношения на три проекции.

Возникает вопрос, зачем нужна такая декомпозиция на три проекции, т.е. чем плохо исходное отношение СЛУ_ПРО_ЗАДАН? Ответ: «этому отношению свойственны аномалии обновления».

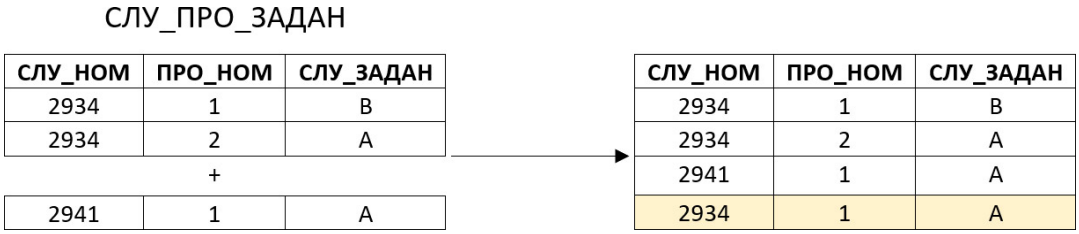


Рис. 50: Аномалия при добавлении кортежей

Добавление кортежей. Если к СЛУ_ПРО_ЗАДАН на рис. 50 добавить (2941,1,A), то надо добавить и кортеж (2934,1,A). Иначе будет нарушено требование зависимости проекции/соединения.

Удаление кортежей. При удалении кортежа (2934,1,A) (на рис. справа) нужно удалить и кортеж (2941,1,A). Иначе опять же будет нарушено требование PJD. Обратим внимание, что можно удалить (2941,1,A) и не трогать (2934,1,A).

Подразумеваемая возможными ключами PJD

В переменной отношения r PJD* $(A, B, \ldots, Z)$ называется подразумеваемой возможными ключами тогда и только тогда, когда каждый составной атрибут $A, B, \ldots, Z$ является суперключом r (включает хотя бы один возможный ключ r).

Тривиальная зависимость проекции/соединения

В переменной отношения r PJD* $(A, B, \ldots, Z)$ называется тривиальной, если хотя бы один из составных атрибутов $A, B, \ldots, Z$ совпадает с заголовком r.

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

Например, если в СЛУ_ПРО_ЗАДАН атрибут СЛУ_НОМ - первичный ключ, то имеется PJD* ({СЛУ_НОМ, ПРО_НОМ}, {СЛУ_НОМ, СЛУ_ЗАДАН}) (это следует из теоремы Хита). Но такие зависимости проекции/соединения неинтересны с точки зрения проекти-

рования базы данных, поскольку не порождают аномалий обновления. Поэтому общепринятое определение пятой нормальной формы выглядит следующим образом.

Пятая нормальная форма (5NF)

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

Таким образом, чтобы распознать, что данная переменная отношения r находится в 5NF, необходимо знать все возможные ключи r и все PJD этой переменной отношения. Обнаружение всех зависимостей соединения является нетривиальной задачей, и для её решения нет общих методов. Поэтому на практике проектирование реляционных баз методом нормализации обычно завершается после достижения 4NF, и отношения, находящиеся в 4NF, как правило, находятся и в 5NF.

Зачем же тогда была введена эта труднодостижимая пятая нормальная форма?

Ответ на этот естественный вопрос состоит в том, что 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. Другими словами, такие отношения далее нормализовать бессмысленно. Как дальше отношение не декомпозируй, оно всё равно будет в 5NF.

Заключение

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

- избежать избыточности хранения данных
- устранить аномалии обновления отношений

Насколько это актуально сейчас, когда объемы доступных носителей внешней памяти непрерывно возрастают, а их стоимость падает?

Начнем с того, что теория реляционных баз данных и методы их проектирования на основе нормализации разрабатывались в расчете на реляционную модель данных. Подавляющее большинство существующих в настоящее время баз данных и средств управления ими опирается на модель данных SQL.

Основным отличием модели данных SQL от реляционной модели данных является то, что таблица модели SQL может содержать мультимножества строк, и поэтому для таблицы, вообще говоря, может быть не определен никакой возможный ключ. Если потребовать от проектируемой SQL-базы данных наличия хотя бы одного возможного ключа для каждой таблицы целевой базы данных, то практиче-

ски все методы нормализации остаются пригодными. Для применимости методов нормализации требуется, чтобы в составе значений любого возможного ключа не допускалось присутствие неопределенных значений (в модели данных SQL это не является обязательным). Для любого определяемого пользователями типа данных необходимо наличие операции сравнения значений этого типа по равенству. А ведь в модели данных SQL можно определить структурный тип данных, который нельзя сравнивать по равенству. В таком случае нельзя производить даже банально операцию естественного соединения. Другими словами, SQL-ориентированную базу данных можно проектировать как реляционную базу данных, если должным образом ограничить используемые средства модели данных SQL. Далее под «реляционными» базами данных будут пониматься SQL-ориентированные БД, не противоречащие требованиям реляционной модели данных.

Вернёмся к вопросу об актуальности нормализации БД на сегодняшний день. Теория реляционных баз данных и методы их проектирования активно развивались в 1980-х г. Ситуация в области технологии аппаратуры и программного обеспечения тогда была совсем иной, чем сегодня, и хорошо нормализованные реляционные базы данных в значительной степени способствовали росту эффективности приложений. В то время реляционные базы преимущественно использовались в информационных системах оперативной обработки транзакций (On-Line Transaction Processing – OLTP). Характерные примеры таких систем – банковские системы, системы резервирования билетов и мест в гостиницах. Системам категории OLTP свойственны частые обновления базы данных, поэтому аномалии обновлений, даже если их корректировка производится СУБД автоматически, могут заметно снижать эффективность приложения. Сегодня на переднем крае приложений баз данных находятся системы категории оперативной аналитической обработки (On-Line Analytical Processing – OLAP). В подобных системах, в частности, системах поддержки принятия решений, базы данных в основном используются для выборки данных, поэтому аномалиями обновлений можно пренебречь, а объем этих баз настолько огромен, что можно пренебречь и избыточностью хранения. Допустим, база данных занимает 4 петабайта. Это настолько много, что будет там 4 или 8 - не особенно важно. Значит ли это, что подход к проектированию реляционных баз данных методом нормализации утратил свою роль? Категорически нет!

Мир приложений баз данных в настоящее время огромен. Сегодня любое маломальски приличное предприятие использует хотя бы одно приложение баз данных – бухгалтерские, складские, кадровые системы. Это системы категории OLTP с частым обновлением данных и умеренными запросами к базе данных, не вызывающими соединений многих отношений. Для небольших компаний важны как эффективность информационных систем, так и стоимость используемых аппаратно-программных средств. Правильно спроектированные, хорошо нормализованные реляционные базы данных помогают решению корпоративных проблем. Да, любое (правильно) развивающееся предприятие рано или поздно приходит к использованию систем категории OLAP, например, некоторой разновидности систем поддержки принятия решений (Decision Support System – DSS). В базах данных таких систем обновления очень

редки, а запросы могут иметь произвольную сложность, включая соединения многих отношений.

Во-первых, технологически правильно для системы OLAP поддерживать отдельную базу данных (обычно подобные базы данных называют хранилищами данных – DataWarehouse). Технологически правильно из-за больших объёмов данных. Т.к. для точной аналитики нужна история, из которой можно выводить прогнозы.

Во-вторых, основными источниками данных для построения таких хранилищ данных являются базы данных систем OLTP. Т.е. надо опираться именно на операционные данные этой самой компании, а не какой-либо ещё. Так что актуальность правильно спроектированных баз данных OLTP-систем не уменьшается, а постоянно возрастает.

Следует ли из этого, что принципы нормализации непригодны для проектирования баз данных OLAP-приложений? И снова категорическое нет! Возможно, окончательная схема такой базы данных должна быть денормализована из соображений повышения эффективности выполнения запросов. Но чтобы получить правильную денормализованную схему, нужно сначала понять, как выглядит нормализованная схема. Поэтому основной вывод можно сформулировать следующим образом: «Пока мы остаемся в мире реляционных баз данных, для правильного проектирования базы данных необходимо понимать принципы нормализации, воспринимая их не как догму, а как руководство к действию».

Проектирование РБД с помощью концептуальных схем

Широкое распространение SQL-ориентированных СУБД и их использование в самых разнообразных приложениях показывает, что реляционная модель данных достаточна для моделирования разнообразных предметных областей. Однако проектирование реляционной базы данных в терминах отношений на основе кратко рассмотренного нами в двух предыдущих лекциях механизма нормализации часто представляет собой очень сложный и неудобный для проектировщика процесс. Ограниченность реляционной модели проявляется в следующих аспектах.

Модель не обеспечивает достаточных средств для представления смысла данных. При проведении опроса для создания начального представления о базе данных, проектировщики получают гораздо больше информации, чем её можно использовать в процессе нормализации. Это разнообразные ограничения целостности, которые не выражаются в терминах зависимостей. Всё, что выходит за пределы первичных и внешних ключей должно держаться в голове проектировщика, и порой даже не понятно как это задокументировать. Другими словами,

- семантика реальной предметной области должна независимым от модели способом представляться в голове проектировщика
- в частности, это относится к проблеме представления ограничений целостности, выходящих за пределы ограничений первичного и внешнего ключа

Хотя весь процесс проектирования происходит на основе учета функциональных и других зависимостей, реляционная модель не предоставляет какие-либо формализованные средства для представления этих зависимостей. В реляционной модели данных нет явного общего механизма, позволяющего отделить объекты предметной области («сущности») от связей между ними.

Например, при нормализации отношения `СЛУЖАЩИЕ` мы получили два отношения: `СЛУЖАЩИЕ` и `ОТДЕЛЫ`. А в `СЛУЖАЩИХ` в каждом кортеже хранится номер отдела. В этом случае этот атрибут не является свойством служащего, это связь служащего с отделом. Смотря на структуру таблицы, понять, что это связь довольно трудно.

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 данных является обеспечение возможности выражения семантики данных.

Чаще всего семантическое моделирование используется на первой стадии проектирования базы данных (когда мы её ещё не привязываем к конкретной СУБД и SQL). 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. Следует заметить, что многие начинающие проектировщики баз данных недооценивают важность семантического моделирования вручную. Зачастую это воспринимается как дополнительная и излишняя работа.

Эта точка зрения абсолютно неверна. Во-первых, 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. Во-вторых, 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. Автору курса неоднократно приходилось наблюдать ситуации, в которых отсутствие такого рода документации существенно затрудняет внесение даже небольших изменений в схему существующей реляционной БД.

Конечно, такая документация важна в случаях, когда проектируемая БД содержит не слишком малое число таблиц. Скорее всего, 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 её последующим преобразованием в реляционную схему БД затруднительна в случае больших БД (содержащих несколько сотен таблиц).

CASE-средства проектирования БД

В связи с этим, начали появляться CASE-средства для автоматизации процесса проектирования БД. Они включают:

- отрисовку диаграмм
- проверку их формальной корректности
- обеспечение средств долговременного хранения диаграмм и другой проектной документации

Конечно, компьютерная поддержка работы с диаграммами для проектировщика БД очень полезна. Наличие электронного архива проектной документации помогает при эксплуатации, администрировании и сопровождении базы данных. Но система, которая ограничивается поддержкой рисования диаграмм, проверкой их корректности и хранением, напоминает текстовый редактор, поддерживающий ввод, редактирование и проверку синтаксической корректности конструкций некоторого языка программирования, но существующий отдельно от компилятора. Т.е. корректность проверить мы можем, но по заданной концептуальной схеме БД скомпилировать реляционную БД мы не можем. Тогда логично такую возможность сделать.

Кажется естественным желание расширить такой редактор функциями компилятора, и это действительно возможно, поскольку известна техника компиляции конструкций языка программирования в коды целевого компьютера. Но коль скоро имеется четкая методика преобразования концептуальной схемы БД в реляционную схему, то почему бы не выполнить программную реализацию соответствующего «компилятора» и не включить её в состав системы проектирования баз данных? Эта идея показалась разумной производителям CASE-средств проектирования БД, и подавляющее большинство подобных систем, 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.

Может возникнуть вопрос, почему в предыдущем предложении говорится про «автоматизированное», а не про «автоматическое» преобразование? Все дело в том, что в типичной схеме SQL-ориентированной БД могут содержаться определения многих объектов, которые невозможно сгенерировать автоматически на основе концептуальной схемы:

- ограничения целостности общего вида
- триггеры

- хранимые процедуры

Поэтому на завершающем этапе проектирования реляционной схемы снова требуется ручная работа проектировщика.

Еще раз обратим внимание на то, какой ход рассуждений привел нас к выводу о возможности автоматизации процесса преобразования концептуальной схемы БД в реляционную схему. Если создатели семантической модели данных предоставляют методику преобразования концептуальных схем в реляционные схемы, то почему бы не реализовать программу, которая производит те же преобразования, следуя той же методике?

Зададимся теперь другим, но, по существу, схожим вопросом. 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.

Рассмотрим устройство одной такой экспериментальной системы. В ней существовал настраиваемый стандартный опросник, по которому эксперты стандартной области отвечали. Набрав соответствующее число мнений экспертов, система рисовала эскиз предметной области. Этот эскиз показывался одному выбранному эксперту, который оценивал и говорил похоже это на правду или нет. Если не похоже, то под его руководством производился дополнительный опрос экспертов области для уточнения концептуальной схемы. Далее полученный эскиз автоматически преобразовывался в реляционную схему в 3NF.

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.

Семантическая модель Entity-Relationship (ER)

На основе ER-модели (модели "сущность-связь") создано большинство современных подходов к проектированию БД. Эта модель была предложена Питером Ченом (Peter Chen) в 1976г.

- Peter Pin-Shan Chen. The Entity-Relationship Model-Toward a Unified View of Data. ACM Transactions on Database Systems, Volume 1, Number 1, 1976.
- Русский перевод: «Модель "сущность-связь" – шаг к единому представлению о данных» (ссылка на статью).

Моделирование предметной области базируется на использовании графических диаграмм, включающих небольшое число разнородных компонентов. Простота и наглядность представления концептуальных схем баз данных в ER-модели привели к её широкому распространению в CASE-системах, поддерживающих автоматизированное проектирование баз данных.

Среди множества разновидностей ER-моделей одна из наиболее популярных и развитых применяется в CASE-системе компании Oracle. Рассмотрим упрощенный вариант этой диаграммной модели, достаточный для понимания основных особенностей проектирования реляционных баз данных с использованием ER-моделей.

Основные понятия ER-модели

Основными понятиями ER-модели являются *сущность*, *связь* и *атрибут*. Сущность – это реальный или представляемый объект, информация о котором должна сохраняться и быть доступной. Это «определение» позаимствовано из одного из ранних вариантов документации CASE-системы Oracle. Понятно, что на самом деле мы имеем дело с тавтологией, поскольку, во-первых, пытаемся определить термин сущность через не определенный термин объект, а во-вторых, попытки определения термина объект настолько же безнадежны. Обычно авторы пытаются оправдываться тем, что в подобном контексте имеется в виду «житейское», а не сколько-нибудь формализованное понятие объекта. Конечно, от этого не становится легче, поскольку понятие сущности должно пониматься в достаточно точном смысле. Но эта тавтология традиционна для области семантического моделирования. В этой области стремятся максимально избегать формальностей.

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. Было бы правильнее всегда использовать термины тип сущности и экземпляр типа сущности, но во избежание многословности в тех случаях, где это не приводит к двусмысленности, мы будем использовать термин сущность в значении типа сущности. Для большей выразительности и лучшего понимания имя сущности может сопровождаться примерами конкретных экземпляров этого типа.

На рис. 51 изображена сущность АЭРОПОРТ с примерными экземплярами «Шереметьево» и «Хитроу». Эта диаграмма примитивная, но тем не менее, несет важную информацию. Во-первых, она показывает, что в базе данных будут содержаться однотипные структуры данных (экземпляры сущности), описывающие аэропорты. Во-вторых, поскольку в жизни существует несколько точек зрения на аэропорты, приведенные примеры аэропортов позволяют несколько сузить допустимый набор

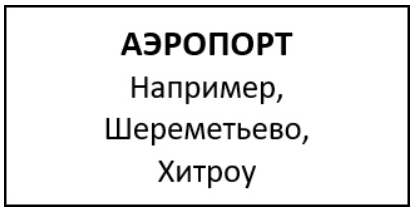


Рис. 51: Пример диаграммы сущности АЭРОПОРТ

точек зрения. В нашем случае приведены примеры международных аэропортов, так что, скорее всего, имеется точка зрения пассажира или пилота международных авиарейсов.

При определении типа сущности необходимо гарантировать, что каждый экземпляр типа сущности может быть отличим от любого другого экземпляра того же типа сущности. Это требование в некотором роде аналогично требованию отсутствия кортежей-дубликатов в реляционных таблицах.

Связь сущностей

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. Тем не менее, как и в случае типа сущности, для краткости мы будем часто использовать термин связь в значении типа связи.

В обсуждаемом здесь варианте ER-модели связи всегда являются бинарными, то есть соединяющими два типа сущности, и они могут существовать между двумя разными типами сущностей или между типом сущности и им же самим (в этом случае связь называется рекурсивной). В общем случае связи могут быть n-арные, не только бинарные. Однако, однажды компания Oracle убедила автора курса в том, что бинарных связей всегда хватает для проектирования БД. Как утверждает Oracle: «Если в какой-то момент кажется, что необходима связь более чем двух типов сущности, то скорее всего при проектировании была потеряна какая-то сущность».

Пример. Рыбак из множества рыбаков может ловить рыбу на озере из множества озёр на лодке из множества лодок. Получаем три сущности: РЫБАК, ОЗЕРО, ЛОДКА. Если надо зафиксировать тот факт, что конкретный рыбак ловит рыбу на конкретном озере на конкретной лодке, то нужна связь трёх типов сущности. Но если подумать, то оказывается, что потерян очень важный тип сущности - РЫБАЛКА. Какие у рыбалки есть свойства, которых нет ни у рыбака, ни в озера, ни у лодки? Во-первых, это улов. Во-вторых, это застолье, относящееся к конкретной рыбалке. При добавлении новой сущности РЫБАЛКА, мы получаем бинарную связь с рыбаком, бинарную связь с озером и бинарную связь с лодкой. Из таких рассуждений похоже, что бинарных связей более чем достаточно.

Лекция 18

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)

В некоторых вариантах ER-модели конец связи называют *ролью* связи в данной сущности. Тогда можно говорить об *имени роли*, *степени роли* и *обязательности роли* связи в данной сущности.

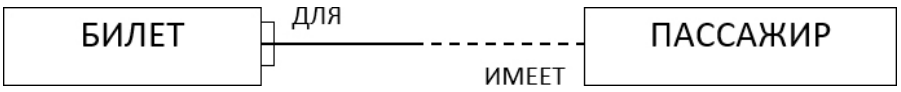


Рис. 52: Пример связи двух сущностей

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

- трёхточечный 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 (one) экземпляр сущности

Обязательный конец связи изображается сплошной линией, а необязательный – прерывистой линией.

На рис. 52 связь между сущностями БИЛЕТ и ПАССАЖИР, связывает билеты и пассажиров. Конец связи с именем «для» трёхточечный, а значит у одного пассажира может быть несколько билетов, причем каждый билет должен быть связан с каким-либо пассажиром. Конец связи с именем «имеет» показывает, что каждый билет может принадлежать только одному пассажиру, причем пассажир не обязан иметь даже один билет. Устная трактовка диаграммы: "Каждый БИЛЕТ предназначен для одного и только одного ПАССАЖИРА. Каждый ПАССАЖИР может иметь один или более БИЛЕТОВ".

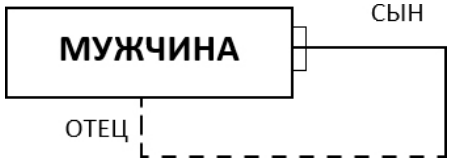


Рис. 53: Пример рекурсивной связи

На рис. 53 изображена рекурсивная связь, связывающая сущность МУЖЧИНА с ней же самой. Конец связи с именем «сын» определяет тот факт, что несколько мужчин могут быть сыновьями одного отца. Конец связи с именем «отец» означает, что не у каждого мужчины должны быть сыновья.

Атрибут сущности

Атрибут сущности - это 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 примерам.

Некоторые атрибуты могут помечаться как необязательные. Значения таких атрибутов не обязаны присутствовать во всех экземплярах данного типа сущности.

Атрибуты типа сущности в ER-модели похожи на атрибуты отношения в реляционной модели данных. Введение именованных атрибутов позволяет относиться к сущности как к типовой структуре данных, которой должен удовлетворять каждый экземпляр данной сущности. Т.е. диаграмма сущности изображает будущий заголовок отношения в реляционной модели.

Но имеется и важное отличие. 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. Отличие состоит в том, что при определении атрибутов типа сущности в ER-модели указание домена атрибута не является обязательным, хотя это и возможно. Посмотрим, чем вызвана эта возможность «ослабленного» определения атрибутов.

Семантические модели данных используются для построения концептуальных схем БД, и эти схемы преобразуются в реляционные схемы БД, которые поддерживаются разными СУБД. Несмотря на то, что в настоящее время типовые возможности реляционных СУБД (РСУБД) в основном стандартизованы (на основе стандарта языка SQL), детали базового набора типов данных и средств определения доменов в разных системах могут различаться. Поскольку производители CASE-средств проектирования реляционных БД стремятся не связывать свои возможности моделирования с конкретной реализацией СУБД, они стимулируют откладывание строгого определения типов атрибутов до стадии полного определения реляционной схемы.

Кроме того, при определении заголовка 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. Более того, желательно при определении имён атрибутов, выбирать те, которые будут подсказывать какие домены у этих атрибутов имеются в виду.

Пониманию предполагаемой сути доменов способствует и возможность указания примеров значений атрибутов. Например, на рис. 54 имеется атрибут год рождения

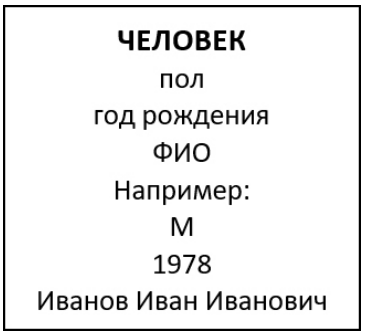


Рис. 54: Сущность ЧЕЛОВЕК с атрибутами и примером значений

с примерным значением 1978. Исходя из этого значения можно сделать вывод, что значения этого атрибута должны быть с точностью до года, а значит в реляционной схеме базовым типом данных этого атрибута будет темпоральный тип «ДАТА».

Обратим внимание, что данный атрибут можно было бы назвать «дата», тогда и сам атрибут, и его домен были бы с одинаковым именем, и это не вызвало бы ошибок.

При определении типа сущности необходимо гарантировать, что каждый экземпляр сущности отличим 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 сущности? Другими словами, можем ли мы выделить индивидуальную книгу в отдельную сущность? Не можем, т.к. нельзя различить книги одного и того же издания. А вот набор книг с одним именем одного автора, вышедших в одном издании, может быть прообразом сущности. Т.к. книги из двух разных изданий отличить можно. Одним из атрибутов этого типа сущности будет число книг в наборе.

С другой стороны, однажды книга уходит со склада и поступает в библиотеку, то ей присваивается уникальный библиотечный номер. В этом случае индивидуальная книга является разумным прообразом сущности.

Уникальные идентификаторы типов сущности

Но при проектировании БД 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. указать уникальные идентификаторы данной 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 атрибутами и связями. Поэтому уникальным идентификатором может быть:

- атрибут (или комбинация атрибутов)
- связь (или комбинация связей)
- комбинация атрибутов и связей

Рис. 55: Пример сущности КНИГА в на библиотечном складе

На рис. 55 показан тип сущности КНИГА, используемый в БД книжного склада. Что в данном случае является уникальным идентификатором? При издании любой книги в приличном издательстве ей присваивается уникальный номер – ISBN. Значение атрибута «isbn» уникально идентифицирует партию книг на складе. Кроме того, конечно, в качестве уникального идентификатора годится и комбинация атрибутов <автор, название, номер издания, издательство, год издания>.

Рис. 56: Связь человека с паспортом уникально идентифицирует человека

Рассмотрим другой пример. На рис. 56 изображены два связанных типа сущности. У каждого обычного взрослого человека имеется один и только один паспорт (некоторые уже готовые паспорта могут быть еще никому не выданы). Тогда связь человека с его паспортом уникально идентифицирует взрослого человека. Однако могут существовать паспорта, еще не выданные человеку, эта связь не является уникальным идентификатором сущности ПАСПОРТ.

На рис. 57 имеются три связанных типа сущности. Профессора обладают знаниями в нескольких учебных дисциплинах. Преподавание каждой дисциплины доступно нескольким профессорам. Другими словами, между сущностями ПРОФЕССОР и ДИСЦИПЛИНА определена связь «многие ко многим».

Каждый профессор может готовить курсы по любой доступной ему дисциплине. Каждой дисциплине может быть посвящено несколько учебных курсов. Но каждый

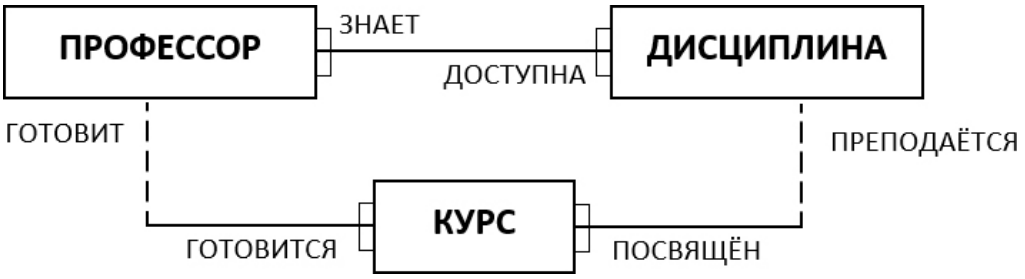


Рис. 57: Курс уникально идентифицируется парой связей

профессор может готовить только один курс по любой доступной ему дисциплине, и каждый курс может быть посвящен только одной дисциплине.

Тем самым, каждый КУРС уникально идентифицируется профессором и дисциплиной, т.е. парой связей с именами концов ГОТОВИТСЯ и ПОСВЯЩЁН на стороне сущности КУРС. Заметим, что сущности ПРОФЕССОР и ДИСЦИПЛИНА связями не идентифицируются.

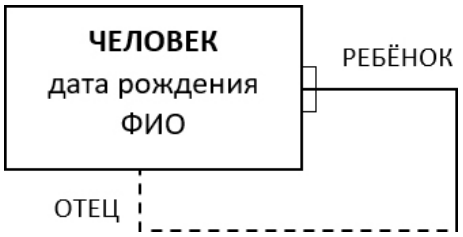


Рис. 58: ЧЕЛОВЕК уникально идентифицируется по ФИО, дате рождения и отцу

Наконец, на рис. 58 приведён пример типа сущности, уникальный идентификатор которого является комбинацией атрибутов и связей. У каждого человека могут быть дети, и у каждого человека имеется отец. Тогда, если предположить, что близнецам, появившимся на свет одновременно, не дают одинаковых имен, то уникальным идентификатором типа сущности ЧЕЛОВЕК может быть комбинация атрибутов <дата рождения, ФИО> и связь с именем конца РЕБЕНОК.

Нормальные формы ER-диаграмм

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. Мы приведем только очень краткие и неформальные определения трех первых нормальных форм. Конечно, можно было бы ввести дальнейшие нормальные формы ER-диаграмм, аналогичные нормальной форме Бойса-Кодда, 4NF и 5NF, но на практике к такой нормализации обычно не прибегают, а общие идеи должны быть понятны и так.

Первая нормальная форма

Рис. 59: Ненормализованная схема

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

На рис. 59 показана диаграмма, в которой тип сущности АЭРОДРОМ не удовлетворяет требованию первой нормальной формы. Здесь несущественны атрибуты сущности АВИАРЕМОНТНОЕ ПРЕДПРИЯТИЕ, но сущность АЭРОДРОМ помимо атрибутов, отражающих собственные характеристики аэродромов (длина взлетно-посадочной полосы, число ангаров и т.д.) содержит атрибут «самолёты», множественное значение которого характеризует все самолёты, приписанные к этому аэродрому.

Очевидно, что самолёты нуждаются в ремонте и должны обслуживаться некоторым авиаремонтным предприятием. Но поскольку самолёты являются частью сущности АЭРОДРОМ, единственным способом фиксации этого факта на диаграмме является проведение связи «многие ко многим» между типами сущности АЭРОДРОМ и АВИАРЕМОНТНОЕ ПРЕДПРИЯТИЕ. Таким образом выражается то соображение, что для ремонта разных самолётов, приписанных к одному аэродрому, могут использоваться разные авиаремонтные предприятия, и каждое авиаремонтное предприятие может обслуживать несколько аэродромов.

Чем плоха эта ситуация? Прежде всего, тем, что скрывается тот факт, что авиаремонтное предприятие ремонтирует самолёты, а не аэродромы. Связь же между типами сущности АЭРОДРОМ и АВИАРЕМОНТНОЕ ПРЕДПРИЯТИЕ на самом деле означает, что любой аэродром из группы аэродромов обслуживается любым авиаремонтным предприятием из группы таких предприятий. Проблема состоит именно в том, что значением атрибута «самолёты» является множество экземпляров типа сущности САМОЛЁТ, и этот тип сущности сам обладает атрибутами и связями.

На рис. 60 показана нормализованная ER-диаграмма, в которой добавлен тип сущности САМОЛЁТ. Связь между сущностями АЭРОПОРТ и САМОЛЁТ показывает, что к одному аэродрому приписывается несколько самолётов. Связь между сущностями САМОЛЁТ и АВИАРЕМОНТНОЕ ПРЕДПРИЯТИЕ означает, что каждый самолёт из группы самолётов обслуживается любым транспортным предприятием из некоторой группы таких предприятий. Эта ER-диаграмма находится

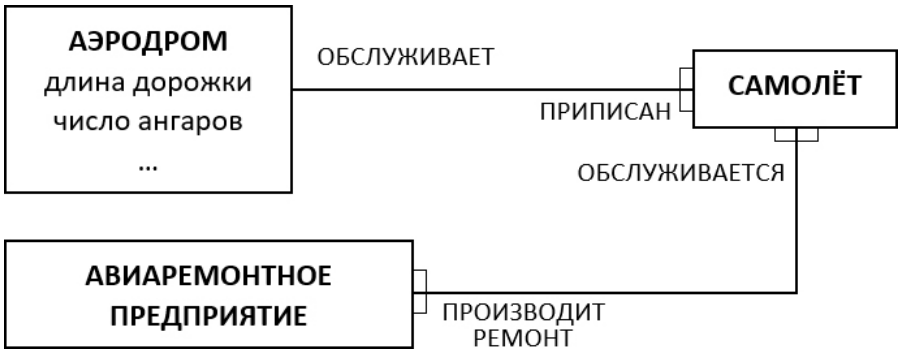


Рис. 60: Диаграмма со всеми сущностями в 1NF

в первой нормальной форме и, как мы видим, правильнее отображает реальную ситуацию.

Вторая нормальная форма

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

Рис. 61: ЭЛЕМЕНТ РАСПИСАНИЯ не находится в 2NF

На рис. 61 показана диаграмма, на которой тип сущности ЭЛЕМЕНТ РАСПИСАНИЯ не удовлетворяет требованиям второй нормальной формы. Элемент расписания - это то, что показывается на табло. Они предназначены для сохранения данных о рейсах самолётов, вылетающих в течение дня. Некоторыми важными характеристиками рейса являются номер рейса, аэропорт вылета, аэропорт назначения, дата и время вылета, бортовой номер самолёта, тип самолёта.

Если говорить про российские авиационные компании, то:

- у каждого рейса имеется заранее приписанный ему номер (уникальный среди всех других имеющихся номеров рейсов)
- не все рейсы совершаются каждый день, поэтому характеристикой конкретного рейса является дата и время его совершения

- бортовой номер самолёта определяется парой `<номер рейса, дата-время вылета>`
- каждый день много рейсов прибывает в один город, поэтому связь между сущностями ЭЛЕМЕНТ РАСПИСАНИЯ и ГОРОД - «многие к одному»

Уникальным идентификатором типа сущности ЭЛЕМЕНТ РАСПИСАНИЯ является пара атрибутов `<номер рейса, дата-время вылета>`. Если вернуться к терминам функциональных зависимостей, то между атрибутами этой сущности имеются следующие FD:

- `{номер рейса, дата-время вылета}` $\rightarrow$ `бортовой номер самолёта`
- `номер рейса` $\rightarrow$ `аэропорт вылета`
- `номер рейса` $\rightarrow$ `аэропорт назначения`
- `бортовой номер самолёта` $\rightarrow$ `тип самолёта`

Кроме того, очевидно, что каждый экземпляр связи с сущностью ГОРОД также определяется значением атрибута номер рейса. Налицо нарушение требования второй нормальной формы. Мы получаем не только избыточное хранение значений атрибутов аэропорт вылета и аэропорт назначения в каждом экземпляре типа сущности ЭЛЕМЕНТ РАСПИСАНИЯ с одним и тем же значением номера рейса. Искажается и затемняется смысл связи с сущностью ГОРОД. Можно подумать, что в разные дни один и тот же рейс прибывает в разные города.

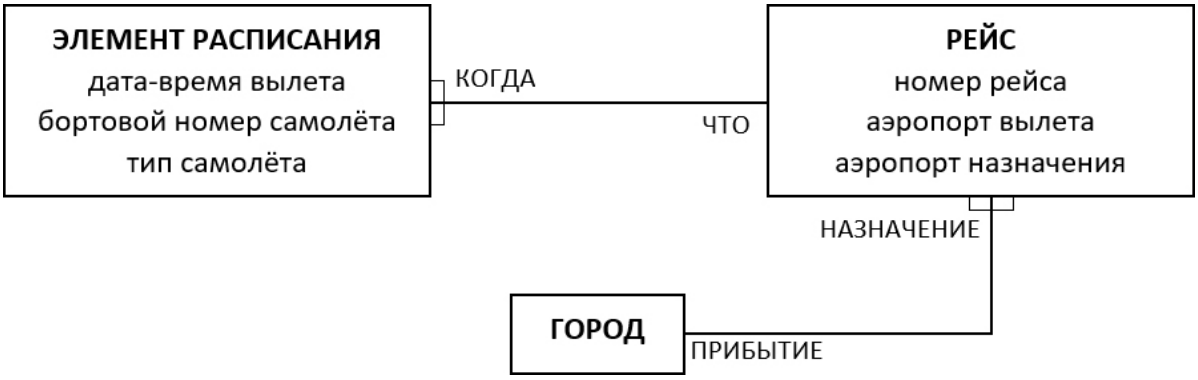


Рис. 62: Все сущности находятся в 2NF

На рис. 62 показана нормализованная диаграмма, в котором все сущности находятся во второй нормальной форме. Теперь имеются три типа сущности: РЕЙС, ЭЛЕМЕНТ РАСПИСАНИЯ и ГОРОД. Уникальным идентификатором сущности РЕЙС является «номер рейса», уникальный идентификатор сущности ЭЛЕМЕНТ РАСПИСАНИЯ состоит из конца связи КОГДА и атрибута «дата-время вылета».

Свойства второй нормальной формы удовлетворяются, т.к. ни в одном типе сущности нет атрибутов, определяемых частью уникального идентификатора.

Третья нормальная форма

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

Взглянем еще раз на тип сущности ЭЛЕМЕНТ РАСПИСАНИЯ (рис. 62). Видно, что каждый день каждый рейс выполняется только одним самолётом, поэтому бортовой номер самолёта полностью зависит от уникального идентификатора (указанного выше). Однако бортовой номер является уникальной характеристикой каждого самолёта, и от этой характеристики зависит «тип самолёта». Т.е. получаем избыточное хранение типа самолёта в каждом экземпляре сущности ЭЛЕМЕНТ РАСПИСАНИЯ.

Между уникальным идентификатором и другими атрибутами типа сущности ЭЛЕМЕНТ РАСПИСАНИЯ имеются следующие функциональные зависимости:

- `{КОГДА, дата-время вылета}` → `бортовой номер самолёта`
- `бортовой номер самолёта` → `тип самолёта`

Имеется транзитивная FD `{КОГДА, дата-время вылета}` → `тип самолёта`, и наличие этой FD вызывает нарушение требования третьей нормальной формы.

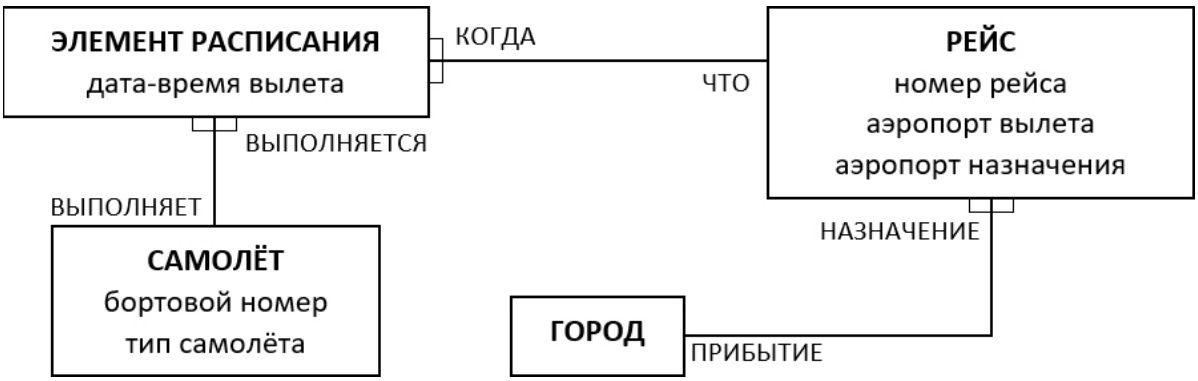


Рис. 63: Все сущности находятся в 3NF

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:

- *Подтипы и супертипы сущностей.* В ER-модели поддерживается возможность наследования типа сущности от одного супертипа сущности (одиночное наследование).
- *Уточняемые степени связи.* Определение возможного количества экземпляров сущности, участвующих в данной связи.
- *Взаимно-исключающие связи.* (Подробнее далее.)

- *Каскадные удаления экземпляров сущностей.* Некоторые связи бывают настолько сильными (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».
- *Домены.* Аналогично случаю реляционной модели данных, есть возможность на существующем типе данных определить домен.

Наследование

Тип сущности при проектировании РБД может быть расщеплен на два или более 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.

Тип сущности, на основе которого определяются подтипы, называется супертипом. Особенности механизма наследования в ER-модели определяются следующими правилами. Если у типа сущности A имеются подтипы $B_1, B_2, \ldots, B_n$ то:

- любой экземпляр типа сущности $B_1, B_2, \ldots, B_n$ является экземпляром типа сущности A (включение)
- если a является экземпляром типа сущности A, то a является экземпляром некоторого подтипа B_i, где $i = 1, 2, \ldots, n$ (отсутствие собственных экземпляров у супертипа)
- ни для каких подтипов B_i и B_j, где $i, j = 1, 2, \ldots, n; i \neq j$ не существует экземпляра, типом которого одновременно являются типы сущности B_i и B_j (разъединённость подтипов)

Проводя аналогию с наследованием в языке программирования C++:

- объект класса-потомка также является объектом базового класса, поэтому правило включения выполняется
- правило отсутствия собственных экземпляров у супертипа аналогично абстрактному классу в C++
- правило разъединённости подтипов соответствует наследованию в C++, т.к. любые два класса-потомка уникально различимы, даже если имеют идентичное описание

На рис. 64 показан супертип ЛЕТАТЕЛЬНЫЙ АППАРАТ и его подтипы АЭРОПЛАН, ВЕРТОЛЁТ, ПТИЦЕЛЁТ и ПРОЧИЕ. У подтипа АЭРОПЛАН имеются два собственных подтипа – ПЛАНЕР и МОТОРНЫЙ САМОЛЁТ. Для супертипа ЛЕТАТЕЛЬНЫЙ АППАРАТ определен атрибут «максимальная дальность полё-

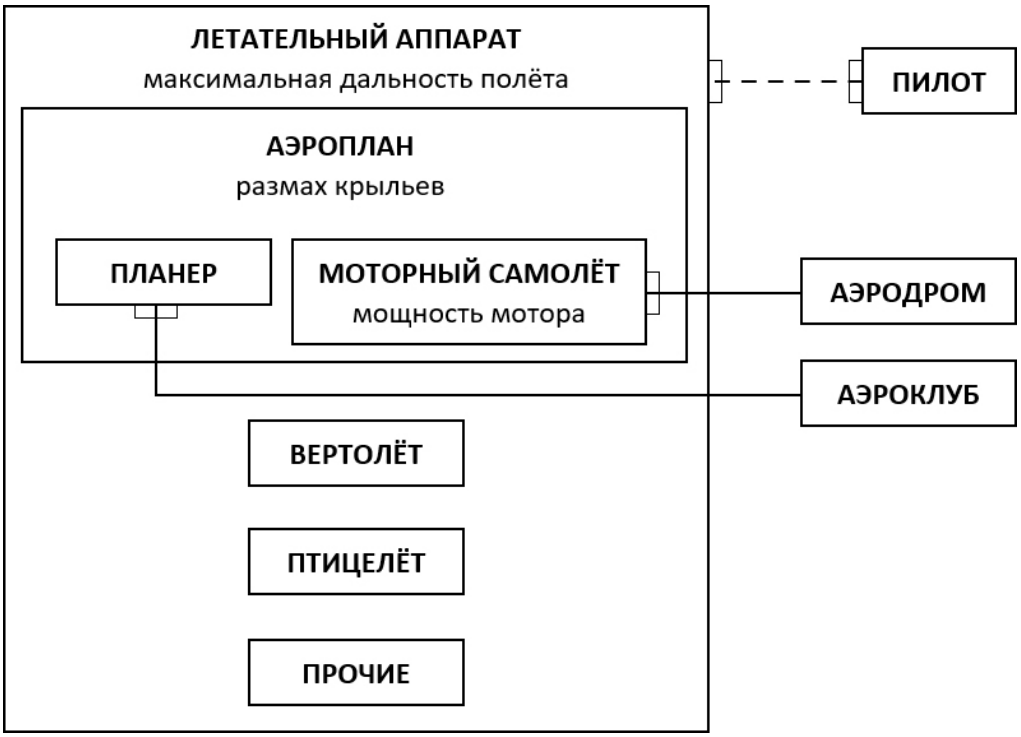


Рис. 64: Пример наследования в ER-модели

та» и необязательная связь «многие ко многим» с типом сущности ПИЛОТ. Этот атрибут и связь наследуется каждым из подтипов.

У подтипа АЭРОПЛАН определяется один дополнительный атрибут, так что у данного типа сущности имеются два атрибута «максимальная дальность полёта» и «размах крыльев» и одна унаследованная связь с типом сущности ПИЛОТ.

У подтипа МОТОРНЫЙ САМОЛЁТ супертипа АЭРОПЛАН определяется один дополнительный атрибут «мощность мотора» и одна обязательная связь с типом сущности АЭРОДРОМ. Тем самым, у типа сущности МОТОРНЫЙ САМОЛЁТ имеются три атрибута: два унаследованных – «максимальная дальность полёта» и «размах крыльев» и один собственный – «мощность мотора», а также две связи: одна унаследованная – с типом сущности ПИЛОТ и одна собственная – с типом сущности АЭРОДРОМ.

Понятно, что для типа сущности ПРОЧИЕ, бессмысленно определять собственные атрибуты и связи, так что свойства этого типа будут совпадать со свойствами его супертипа.

Как же следует понимать эту диаграмму?

- Если начинать от супертипа, то диаграмма изображает ЛЕТАТЕЛЬНЫЙ АППАРАТ, который должен быть АЭРОПЛАНОМ, ВЕРТОЛЁТОМ, ПТИЦЕЛЁТОМ или ПРОЧИМ ЛЕТАТЕЛЬНЫМ АППАРАТОМ.
- Если начинать от подтипа (например, сущности ВЕРТОЛЁТ), то это ВЕРТОЛЁТ, 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 САМОЛЁТОМ.

В механизме наследования ER-модели также допускается наличие двух или более разбиений сущности на подтипы. Например, тип сущности ЧЕЛОВЕК может быть расщеплен на подтипы по профессиональному признаку (ПРОГРАММИСТ, ДОЯРКА и т. д.), а может быть расщеплен и по половому признаку (МУЖЧИНА, ЖЕНЩИНА). Однако автор курса обращает внимание, что конкретно данную возможность не понятно каким образом реализовать в SQL-ориентированных СУБД, поэтому далее она рассматриваться не будет.

Изображать наследование в виде вкладывания, как это делается в ER-модели, удобно, т.к. это позволяет сразу видеть общую картину. Нам отлично видно устройство типа ЛЕТАТЕЛЬНЫЙ АППАРАТ.

С другой стороны, представим себе, что мы рисуем сложную диаграмму, в которой указаны детали аэроклубов: в диаграмме учтены и люди, которые летают, и места куда летают, и прочие характеристики аэроклубов. Понятно, что они не имеют никакого отношения к аэродромам, у которых есть своя инфраструктура, т.е. дороги, гостиницы, предприятия общепита и т.д. Все эти детали на одной диаграмме приводят к загромождённости и трудности восприятия, когда информации слишком много.

Далее будут рассмотрены UML-диаграммы классов, в которых наследование изображается стрелочками. Но стрелочки не решают проблемы загромождённости и непонятно насколько они лучше или хуже изображения наследования вкладыванием.

Взаимно-исключающие связи

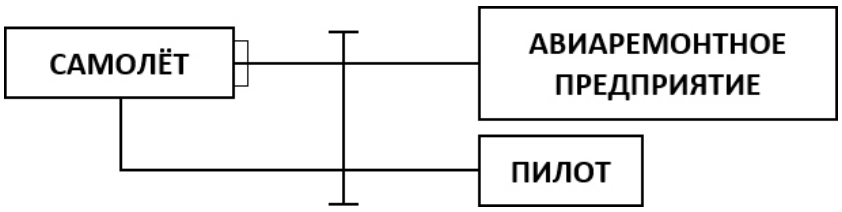


Рис. 65: Диаграмма с взаимно-исключающими связями

На рис. 65 показана диаграмма из двух сущностей с взаимно-исключающими связями. Она показывает, что если самолёт находится в рабочем состоянии, то у него имеется один пилот. Если же самолёт неисправен, то его обслуживает авиаремонтное предприятие, в этот момент пилота за самолётом не закреплено.

Взаимно-исключающие связи требуют, чтобы существовал экземпляр только одной связи из заданного набора связей. В данном случае для каждого экземпляра типа сущности САМОЛЕТ должен существовать экземпляр одной из указанных связей. Для экземпляров типа сущности САМОЛЕТ, соответствующих исправным

самолётам, должен существовать экземпляр связи «один к одному» с экземпляром типа сущности ПИЛОТ, а экземпляры, соответствующие неисправным самолётам, должны участвовать в экземпляре типа связи «многие ко одному» с экземпляром типа сущности АВИАРЕМОНТНОЕ ПРЕДПРИЯТИЕ.

Диаграмма с взаимно-исключающими связями может быть преобразована к диаграмме без взаимно-исключающих связей путем введения подтипов.

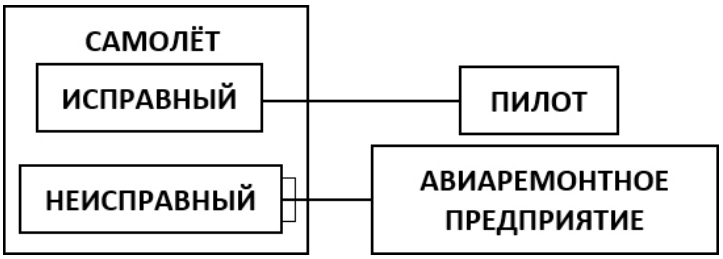


Рис. 66: Эквивалентная диаграмма с использованием подтипов

Поскольку любой самолёт может быть либо исправным, либо неисправным, можно корректным образом ввести два подтипа самолёта: ИСПРАВНЫЙ и НЕИСПРАВНЫЙ. На уровне супертипа САМОЛЁТ связи не определяются. Для подтипа ИСПРАВНЫЙ определяется обязательная связь «один к одному» с типом сущности ПИЛОТ, а для подтипа НЕИСПРАВНЫЙ определяется обязательная связь «многие к одному» с типом сущности АВИАРЕМОНТНОЕ ПРЕДПРИЯТИЕ.

Получение реляционной схемы из ER-диаграммы

Базовые приемы

Опишем типовую процедуру преобразования ER-диаграммы в реляционную схему БД. Предполагается использование «традиционных» средств определения данных SQL, не включающих возможности определения структурных типов данных с поддержкой механизма наследования типов и типизированных таблиц. К сожалению, отсутствует общепризнанная методология проектирования SQL-ориентированных БД, в которых используются «объектные» расширения SQL, поэтому мы их опускаем.

Итак, сами преобразования. Простым типом сущности называется тип сущности, не являющийся подтипом и не имеющий подтипов. Каждый простой тип сущности превращается в таблицу:

- Имя сущности становится именем таблицы.
- Каждый атрибут становится столбцом таблицы; может выбираться более точный формат представления данных.

  - Столбцы, соответствующие необязательным атрибутам, могут содержать неопределенные значения; столбцы, соответствующие обязательным атрибутам не могут.

- Экземпляр типа сущности становится строкой таблицы.
- Уникальный идентификатор сущности становятся первичным ключом таблицы.
  - 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 равна единице, соответствующий (возможно, составной) столбец должен быть дополнительно специфицирован как возможный ключ таблицы.
- 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 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)>, где $i = 1, 2, \ldots, n; j = 1, 2, \ldots, m$.
  - Понятно, что, используя таблицы A, B и AB, с помощью стандартных реляционных операций можно найти все пары экземпляров типов сущности, участвующих в данной связи «многие ко многим».

Стоит отметить, что ещё при проектировании с помощью семантической ER-диаграммы надо осторожно относиться к связям «один-к-одному» (особенно обязательным с обоих концов!), т.к. часто бывает, что на самом деле они являются частью

одной логической сущностью, т.е. например не стоит из сущности ИЗОБРАЖЕНИЕ выделять отдельную сущность ЦВЕТ. Это неправильно, т.к. нет осмысленного уникального идентификатора цвета. Его уникальный идентификатор - это само значение цвета, т.е. весь заголовок (например, `<r,g,b,a>` в случае цветовой модели rgba). С другой стороны, не нужно склеивать в одну сущность сущности ПАСПОРТ и ЧЕЛОВЕК, т.к. хоть по паспорту и можно уникально идентифицировать человека, но он не является его атрибутом (человек может ещё не иметь паспорта из-за возраста, или иметь несколько паспортов, или ещё не получить уже изготовленный паспорт и т.д.).

Супертипы и подтипы

Существует два способа представить наследование сущностей в реляционной схеме. Рассмотрим их на примере. Пусть имеется супертип ЧЕЛОВЕК с подтипами ПРОГРАММИСТ, ДОЯРКА, ДВОРНИК.

Первый способ заключается в *создании единой таблицы* для всех трёх подтипов. Эта таблица содержит столбцы, соответствующие каждому атрибуту (и связям) каждого подтипа, а также один специальный столбец «код подтипа». Для каждой строки таблицы значение этого столбца определяет конкретный подтип, которому соответствует строка. Например, можно указать 0 для программиста, 1 для доярки и 2 для дворника. Столбцы, которые соответствуют атрибутам и связям, отсутствующим в данном типе сущности, должны содержать неопределенные значения.

К достоинствам такого способа можно отнести:

- обеспечение простого доступа к экземплярам супертипа и не слишком сложный доступ к экземплярам подтипов
- возможность обойтись небольшим числом таблиц

К недостаткам можно отнести:

- приложения, работающие с одной таблицей супертипа, должны содержать дополнительный программный код для работы с разными наборами столбцов (в зависимости от значения столбца «кода типа») 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 (например, работаем с доярками, но заблокированы ещё и дворники с программистами)
- потенциально в общей таблице будет содержаться много неопределенных значений, что может привести к непроизводительному расходу внешней памяти

Второй способ предполагает *создание отдельной таблицы для каждого подтипа.* В этом случае для получения всех кортежей супертипа нужно объединить проекции таблиц подтипов, на заголовок таблицы супертипа. Другими словами, из всех таблиц

подтипов выбираются общие столбцы супертипа. В нашем примере это соответствует созданию трёх таблиц: ДОЯРКИ, ДВОРНИКИ, ПРОГРАММИСТЫ.

Достоинства:

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

Недостатки:

- 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

Взаимно-исключающие связи

Ранее было показано, что ER-диаграмму с взаимно-исключающими связями можно преобразовать к диаграмме с подтипами. Однако можно построить схему SQL-ориентированной базы данных и без такого преобразования.

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):

- определение таблицы с одним столбцом для представления всех взаимно-исключающих связей, т.е. общее хранение внешних ключей
- определение таблицы, в которой каждой взаимно-исключающей связи соответствует отдельный столбец, т.е. 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 первый способ, т.е. создать два столбца, содержащие идентификатор связи и уникальный идентификатор соответствующей сущности (второй столбец может быть составным). Столбец идентификатора связи используется для различения связей, покрываемых дугой исключения.

Если результирующие внешние ключи не относятся к одному домену, то приходится прибегать к использованию второго способа, т.е. создавать для каждой связи,

покрываемой дугой исключения, явные столбцы внешних ключей; каждый из этих столбцов может содержать неопределенные значения.

Преимущество первого подхода состоит в том, что в таблице, соответствующей сущности с взаимно-исключающими связями, появляется всего два дополнительных столбца. 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 второго подхода 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 может привести к излишнему расходу внешней памяти.

Заключение

На этом мы заканчиваем краткую экскурсию в семантическое моделирование с использованием ER-диаграмм. Основной целью этого раздела было ознакомление с семантическими моделями данных на примере упрощенного варианта ER-модели. Представленный вариант ER-модели, с одной стороны, является достаточно развитым, чтобы можно было почувствовать общую специфику семантических моделей данных, а с другой стороны, не перегружен деталями и излишними понятиями, затрудняющими общее понимание подхода. С практической точки зрения наибольшую пользу могут принести рассмотренные приемы перехода от ER-диаграмм к схеме реляционной БД. Особенно могут пригодиться рекомендации по представлению в реляционной схеме связей «многие ко многим», подтипов и супертипов сущности и взаимно-исключающих связей.

Лекция 19

Диаграммы классов языка UML

Рассмотрим основные понятия диаграмм классов языка UML и возможности применения этой диаграммной модели для проектирования реляционных БД. По UML написано много книг в том числе на русском языке, но порой их сложно читать из-за сложностей перевода на русский язык английских терминов. Автор курса рекомендует книги Александра Михайловича Вендорова по языку UML.

UML (Unified Modeling Language) является языком объектно-ориентированного моделирования. Он разработан консорциумом OMG (Object Management Group) и имеет много общего с объектными моделями, на которых основана технология распределенных объектных систем CORBA, и объектной моделью ODMG (Object Data Management Group).

При основании ODMG, язык IDL (Interface Definition Language) был основой для языка ODL (Object Definition Language) и, соответственно, для объектно-ориентированной модели данных ODMG. Далее модель ODMG была использована для формализации языка UML.

Язык UML состоит из нескольких подъязыков (они все графические, диаграммные), в число которых входит подъязык диаграмм классов. Эти подъязыки (спецификации) называются метамоделями. В терминах UML концептуальная схема (схема, которую создаёт проектировщик) называется моделью. Модель может быть представлена из метамоделей.

Для нас интересно то, что метамодели так же основываются на единой спецификации - метаметамодели, которая описывает устройство метамоделей. Самое интересное, что метаметамодель (которая является логическим корнем) описывается через диаграммы классов UML. Например, в метаметамодели *объект* определяется с помощью диаграмм классов, в которых уже используется понятие объекта. Поэтому здесь нет никакой речи о формальном определении метаметамодели.

Поэтому термины «сущность» и «объект» используются настолько же неформально, как и термин «сущность» в ER-модели. По-прежнему приходится опираться на интуицию и здравый смысл: *Объект* - это контейнер, содержащий типизированные значения, и по отношению к которому можно применять некоторый заданный набор операций.

Когда-то были надежды, что язык UML будет полностью формально определённым, т.е. что будет определена формальная семантика всех его подъязыков. К сожалению, этого так и не произошло по причине того, что сам язык UML неформально определён, вводить формальную семантику его подъязыков не имеет смысла. Полная формальная семантика иногда была бы очень полезна: например, для CASE-систем, которые пытаются компилировать модели языка UML в термины других языков.

Стоит отдельно отметить, что терминология в диаграммах классов UML используется отличная от той, что в ER-модели. Однако по своей сути эти модели эквивалентны, т.е. всё, что можно представить при семантическом моделировании с использованием диаграмм классов, можно представить с использованием ER-модели, и наоборот.

Часто можно слышать утверждение, что ER-модель является структурной семантической моделью, а метаметамодель UML является объектно-ориентированной. Но средства одного выражаются через средства другого, поэтому оно не имеет много смысла.

UML позволяет моделировать разные виды систем: чисто программные, чисто аппаратные, программно-аппаратные, смешанные (включающие деятельность людей) и т.д. Если возможности диаграмм классов несущественно отличаются от возможностей ER-диаграмм, почему о них идёт речь в этом курсе? Это связано с тем, что в реальном проекте почти наверняка будут использованы именно UML-диаграммы (диаграммы активности, диаграммы способов использования и другие).

Основные понятия языка UML

Диаграмма классов (в терминологии UML) - это диаграмма, на которой показан набор классов (и некоторых других сущностей, не имеющих явного отношения к проектированию БД), а также связей между этими классами. Диаграмма классов может включать комментарии и ограничения.

Как и в модели ER-диаграмм, в UML для родового обозначения связей используется термин relationship. Во многих переводах книг про UML на русский язык вместо термина «связь» применяется термин «отношение». Как и в предыдущем разделе, мы будем использовать термин «связь».

Для диаграмм классов UML могут задаваться ограничения на естественном языке или же на языке объектных ограничений OCL (Object Constraints Language). Язык OCL является частью общей спецификации UML, но, в отличие от других частей языка, имеет не графическую, а линейную нотацию (как в обычных языках программирования). Более подробно язык OCL будет рассмотрен далее.

Класс

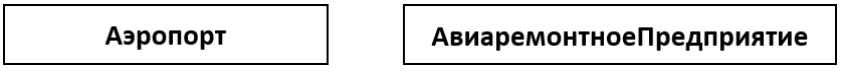


Рис. 67: Пример двух классов без атрибутов

Класс - это именованное описание совокупности объектов с общими атрибутами, операциями, связями и семантикой. На рис. 67 приведён пример двух классов.

У каждого класса должно быть уникальное имя (текстовая строка). При формировании имен классов в UML допускается использование произвольной комбинации букв, цифр и даже знаков препинания. На практике в качестве имен классов рекомендуется использовать короткие и осмысленные прилагательные и существительные без пробелов, каждое слово начинается с заглавной буквы.

Атрибут

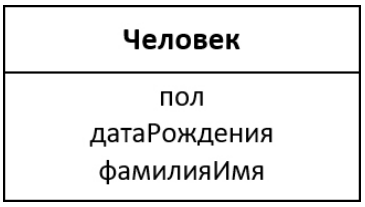


Рис. 68: Класс с атрибутами

Атрибут класса - это именованное свойство класса, описывающее множество значений, которые могут принимать экземпляры этого свойства. Множество всех атрибутов класса описывает структуру этого класса. Класс может иметь любое число атрибутов (в том числе ноль). Свойство, выражаемое атрибутом, является свойством моделируемой сущности, общим для всех объектов данного класса, т.е. атрибут является абстракцией состояния объекта.

Имена атрибутов представляются в разделе класса, расположенном под именем класса. На практике для имен атрибутов 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.

Операции класса

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

Операции класса определяются в разделе, расположенном ниже раздела с атрибутами. Можно ограничиться только указанием имен операций, оставив детальную спецификацию выполнения операций на более поздние этапы моделирования. Для именования операций рекомендуется использовать глаголы, соответствующие ожидаемому поведению объектов данного класса. Описание операции может также содержать имена и типы всех параметров (и тип возвращаемого значения).

На рис. 69 показан класс Человек с трёмя операциями (например, со следующей функциональностью):

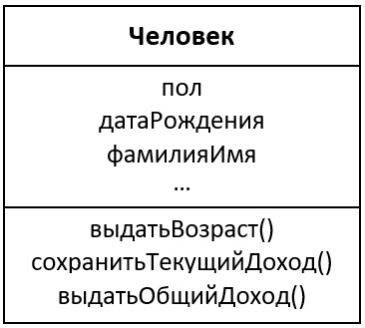


Рис. 69: Класс с атрибутами и операциями

- выдатьВозраст - возвращает возраст человека, как разность текущей даты и значения атрибута датаРождения
- сохранитьТекущийДоход - позволяет зафиксировать в состоянии объекта сумму и дату поступления дохода данного человека
- выдатьОбщийДоход - возвращает суммарный доход данного человека за указанное время

Заметим, что состояние объекта (значение атрибутов) меняется при выполнении только второй операции. Результаты первой и третьей операций формируются на основе текущего состояния объекта.

Связь-зависимость

В диаграмме классов могут участвовать связи трёх категорий:

- зависимость (dependency)
- обобщение (generalization)
- ассоциация (association)

Зависимость - это связь, показывающая, что изменение описании одного класса может повлиять на поведение другого класса, использующего первый класс. Как правило зависимость отражает тот факт, что в сигнатуре операции одного класса параметром является объект другого класса. Понятно, что если интерфейс второго класса изменяется, это влияет на поведение объектов первого класса.

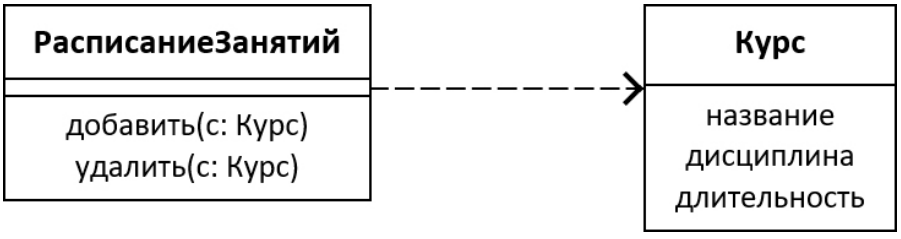


Рис. 70: Диаграмма со связью-зависимостью

На рис. 70 показана диаграмма со связью-зависимостью. Класс РасписаниеЗанятий имеет две операции: добавить и удалить, параметрами которых является объект класса Курс. При изменении интерфейса класса Курс изменится поведение объектов класса РасписаниеЗанятий. На диаграмме связь-зависимость изображается прерывистой линией со стрелкой, направленной к классу, от которого имеется зависимость.

Связи-зависимости существенны для объектно-ориентированных систем. При проектировании традиционных РБД непонятно, как использовать информацию о наличии связей-зависимостей между классами.

Связь-обобщение

Обобщение - это связь между суперклассом (родителем) и подклассом (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.

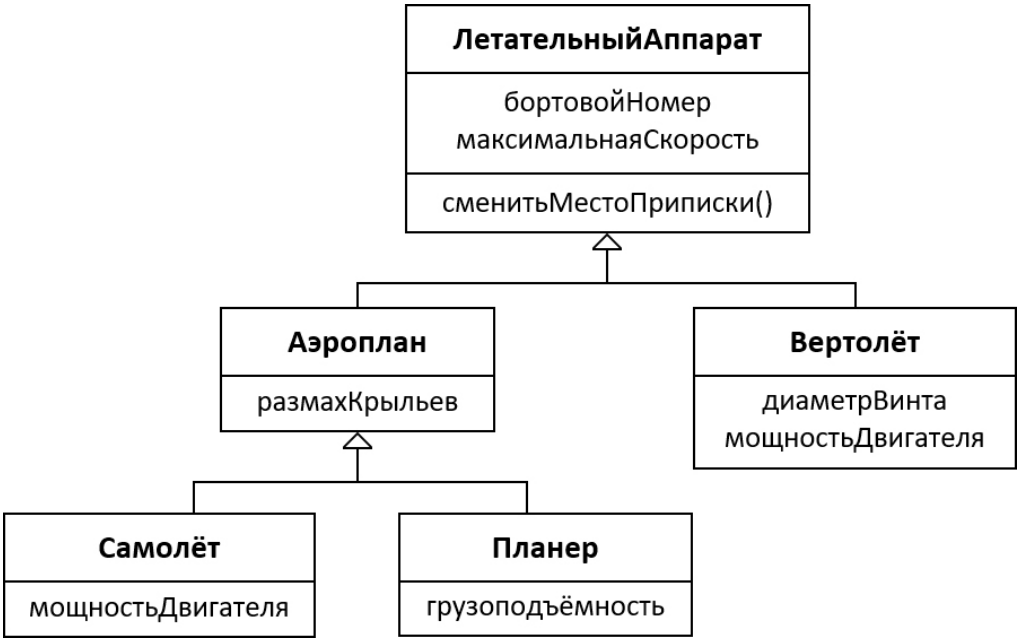


Рис. 71: Пример диаграммы с одиночным наследованием

На рис. 71 показан пример иерархии одиночного наследования: у каждого подкласса имеется только один суперкласс. В отличие от механизма наследования типов

сущностей ER-модели здесь отсутствует класс ПРОЧИЕ, т.е. в классе Летательный-Аппарат могут присутствовать «собственные» объекты, не относящиеся ни к классу Аэроплан, ни к классу Вертолёт.

Обычно при разработке одиночного наследования достаточно, но в диаграммах классов допускается и множественное наследование, когда один подкласс определяется на основе нескольких суперклассов.

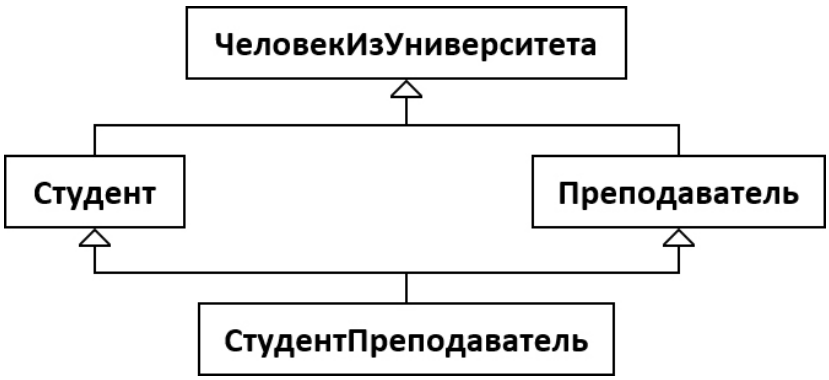


Рис. 72: Пример диаграммы с множественным наследованием

На рис. 72 классы Студент и Преподаватель порождены из одного суперкласса ЧеловекИзУниверситета. Бывают случаи, когда студенты начинают преподавать, т.е. логически один и тот же объект класса ЧеловекИзУниверситета принадлежит сразу двум классам: и Студенту и Преподавателю. Тогда мы можем определить класс СтудентПреподаватель путем множественного наследования от суперклассов Студент и Преподаватель.

Заметим, что в наследовании UML, в отличие от ER-модели, могут быть общие экземпляры у подтипов одного типа сущности. В данном случае множественное наследование возможно именно потому, что в классы Студент и Преподаватель входят разные объекты, которым соответствует один и тот же объект суперкласса.

Следует также 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 со СтудентПреподавателями, у которых и комната в общежитии, и служебный кабинет, и называются эти атрибуты одинаково?

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 атрибут только от одного из суперклассов.

3) Автоматически переименовать атрибуты (например, номерКомнатыСтудента и номерКомнатыПреподавателя) и унаследовать оба.

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.
- Третье решение заставляет использовать длинные имена атрибутов и операций, которые могут стать недопустимо длинными, если процесс множественного наследования будет продолжаться от полученного подкласса.

Поэтому, предлагается для решения проблемы одноименных атрибутов при множественном наследовании переименовать атрибуты, но не автоматически, а вручную. Такой подход применяется в реляционной модели данных.

Стоит отметить, что сложность реализации множественного наследования в РБД очень высокая, поэтому нужно очень осторожно использовать наследование классов вообще и стараться избегать множественного наследования.

Связь-ассоциация

Ассоциация - это структурная связь, показывающая, что объекты одного класса некоторым образом связаны с объектами другого или того же самого класса. В ассоциации могут связываться два класса, и тогда она называется бинарной. Допускается создание ассоциаций, связывающих сразу n классов (они называются n-арными ассоциациями). Ранее был рассмотрен пример с рыбалкой (стр. 137), показывающий, что бинарных связей достаточно. Поэтому мы ограничимся обсуждением только бинарных ассоциаций. С понятием ассоциации связаны четыре важных понятия: имя, роль, кратность и агрегация.

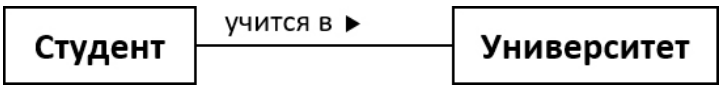


Рис. 73: Пример именованной ассоциации

Графически ассоциация изображается в виде линии. 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.

На рис. 73 показан пример именованной ассоциации. Треугольник показывает, что именованная ассоциация должна читаться как «Студент учится в Университете».

Другой способ именования ассоциации - это подпись роли каждого класса, участвующего в этой связи. Роль класса обозначает роль класса в данной связи.

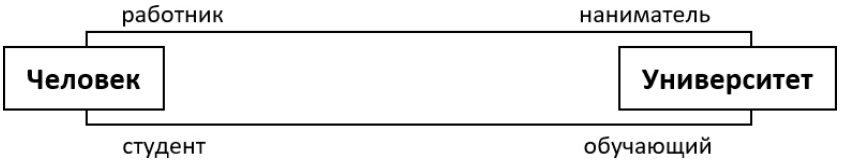


Рис. 74: Две ассоциации с указанными ролями

Роль задается именем, помещаемым рядом с линией ассоциации ближе к данному классу. На рис. 74 показаны две ассоциации между классами Человек и Университет, в которых эти классы играют разные роли.

В общем случае, можно задавать и имена ролей, и имя самой связи. 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.

Кратность (multiplicity) роли ассоциации - это характеристика, указывающая, сколько объектов класса с данной ролью должно участвовать в каждом экземпляре ассоциации. Кратность роли ассоциации задаётся указанием конкретного числа или диапазона рядом с концом связи.

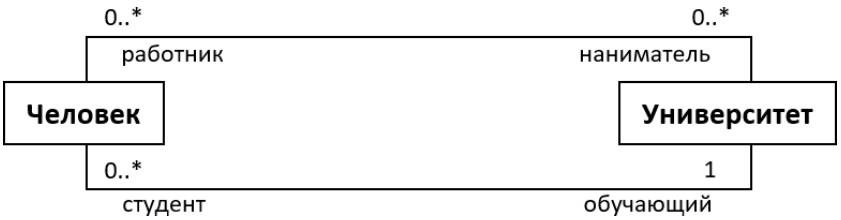


Рис. 75: Ассоциации с указанными кратностями

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.

Например, на рис. 75 на конце связи «студент» диапазон 0..* означает что в данной ассоциации может участвовать произвольное число людей, в том числе 0. Значение «1» на конце «обучающий» говорит о том, что для каждого экземпляра ассоциации должен быть ровно один университет в роли обучающий.

Аналогично указание диапазона «1..*» говорит о том, что в каждом экземпляре ассоциации должен участвовать хотя бы один объект (верхняя граница не задана). В более сложных 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.

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».

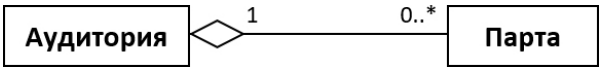


Рис. 76: Пример агрегатной ассоциации

Объектами класса Аудитория являются студенческие аудитории, в которых проходят занятия. В аудитории могут быть установлены парты. Поэтому в некотором смысле класс Парта является «частью» класса Аудитория. Мы умышленно сделали роль класса Парта в этой ассоциации необязательной, поскольку могут существовать аудитории без парт (например, класс для занятий танцами) и некоторые парты могут находиться на складе.

Обратите внимание, что, хотя аудитории, не оснащенные партами, как правило, непригодны для занятий, объекты классов Аудитория и Парта существуют независимо. Если некоторая аудитория ликвидируется, то находящиеся в ней парты не уничтожаются, а переносятся на склад.

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 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».

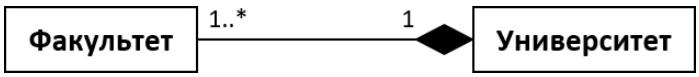


Рис. 77: Пример композитной агрегатной ассоциации

На рис. 77 приведён пример композиции. Любой факультет является частью одного университета, и ликвидация университета приводит к ликвидации всех существующих в нем факультетов.

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 (рис. 73) между двумя классами 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 (рис. 78).

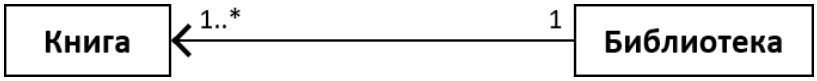


Рис. 78: Пример диаграммы классов с однонаправленной навигацией

В библиотеке должно содержаться некоторое количество книг, и каждая книга должна принадлежать некоторой библиотеке. С точки зрения библиотеки разумно иметь возможность найти книгу в библиотеке, т.е. произвести навигацию от объекта-библиотеки к связанным с ним объектам-книгам. Однако вряд ли потребуется по данному экземпляру книги узнать, в какой библиотеке она находится.

Ограничения целостности и язык OCL

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.

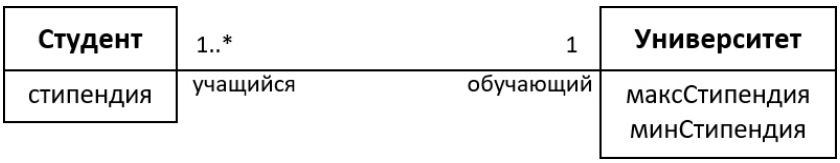


Рис. 79: Хотим «стипендию» между «максСтипендия» и «минСтипендия»

На рис 79 показаны классы Студент и Университет. Пусть мы хотим, чтобы значение атрибута «стипендия» класса Студент принадлежало отрезку `[минСтипендия; максСтипендия]`. Другими словами, объект класса Студент может входить в экземпляр ассоциации с объектом класса Университет только при условии, что размер стипендии данного студента находится в диапазоне, допустимом в данном университете.

Язык OCL (Object Constraints Language) позволяет описывать пободного рода ограничения. На языке OCL пример с рис. 79 записывается следующим образом:

```
context Студент inv:
self.стипендия >= self.обучающий.минСтипендия and
self.стипендия <= self.обучающий.максСтипендия
```

Хотя язык OCL формально считается частью UML, но он специфицирован в отдельном документе, в котором вводятся собственные понятия и определения. Из языка UML в OCL заимствованы следующие понятия:

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

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. Заметим, что хотя в UML допускаются n-арные связи, в OCL речь идет только про бинарные.

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 повторные элементы-дубликаты
- последовательность (sequence)
- упорядоченная коллекция, которая может содержать элементы-дубликаты

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
- ограничений, представляющих собой инварианты классов

С точки зрения определения ограничений целостности БД более важны средства определения инвариантов классов.

Понятие инварианта класса

Под инвариантом класса в OCL понимается условие, которому должны удовлетворять все объекты данного класса. Если говорить более точно, инвариант класса – это логическое выражение, при вычислении которого для любого объекта данного класса в течение всего времени существования этого объекта получается булевское значение true.

При определении инварианта требуется указать имя класса и выражение, определяющее инвариант указанного класса. Синтаксически это выглядит следующим образом: `context <class_name> inv: <OCL-выражение>`, где:

- <class-name> - имя класса, для которого определяется инвариант
- inv – ключевое слово, говорящее о том, что определяется именно инвариант, а не ограничение другого вида
- context – ключевое слово, которое говорит о том, что контекстом следующего после двоеточия OCL-выражения являются объекты класса <class-name>

OCL является типизированным языком, поэтому у каждого выражения имеется некоторый тип. OCL-выражение в инварианте класса должно быть логического типа. В общем случае OCL-выражение в определении инварианта основывается на композиции операций. В спецификации языка эти операции условно разделены на следующие группы:

- операции над предопределенными типами данных
- операции над объектами
- операции над коллекциями (множествами, мультимножествами и последовательностями)

Операции над предопределенными типами данных

В OCL в число предопределённых типов входят Boolean, Integer, Real и String. На рис. 80 перечислены стандартные операции над предопределёнными типами данных.

Тип данных	Операции
Boolean	and, or, xor, not, implies, if-then-else
Integer	*, +, -, /, abs(), операции сравнения
Real	*, +, -, /, floor(), операции сравнения
String	concat(), size(), substring()

Рис. 80: Предопределённые скалярные типы в OCL

Пояснения некоторых операций:

- xor (исключающее или) принимает два параметра булевского типа и возвращает true, если значением только одного параметра является true, иначе false.
- implies (импликация) принимает два параметра булевского типа и возвращает true, если значением первого параметра является false, или если значениями обоих параметров является true, иначе операция выдаёт false.
- floor вырабатывает наибольшее значение целого типа, меньшее или равное значению параметра операции. Например, floor(6.5)=6; floor(-5.5)=-6.
- concat конкатенирует две строки (начало второй строки «склеивается» с концом первой).
- size выдает целое значение, равное длине строки-аргумента.
- substring выдает подстроку строки-аргумента с заданными начальной позицией и длиной.

Операции над объектами

Одной из трёх операций над объектами является операция получения значения атрибута объекта, которая возвращает текущее значение соответствующего атрибута. Если атрибут типизирован именем некоторого класса, то результатом вызова операции является некоторый объект этого класса (объектный идентификатор), к которому также применимы операции над объектами. Если атрибута не существует, то возникает ошибка типа. Обозначение:

```
<объект>.<атрибут>
```

Операция перехода по экземпляру связи-ассоциации возвращает коллекцию всех объектов, которые ассоциированы с данным объектом через указываемый экземпляр ассоциации. Эта коллекция идентифицируется именем роли, противоположной по отношению к данному объекту. Обозначение:

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

В OCL также имеется операция вызова операции класса, но она несущественна при проектирования РБД.

Операции над коллекциями

В OCL поддерживается обширный набор операций над значениями коллекционных типов данных (множествами, мультимножествами, последовательностями), но мы рассмотрим только важные при проектировании БД.

Синтаксически вызовы операций над коллекциями записываются в стрелочной нотации. Общий синтаксис применения операции к коллекции выглядит следующим образом: `<коллекция>` → `<имя операции> (<список фактических параметров>)`.

Операция select

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.

Операция collect

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 некоторый экземпляр ассоциации. В этом случае выражение над элементом исходной коллекции основывается на операции перехода по экземпляру ассоциации.

Операции exists, forAll, count

В OCL определены три одноименных операции exists над множеством, мультимножеством и последовательностью; дополнительным параметром этих операций 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 exist 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 count применяется к коллекции и выдает число содержащихся в ней элементов.

Операции union, intersect, symmetricDifference

Параметрами двуместных операций union (объединение), intersect (пересечение), symmetricDifference (симметричное вычитание) 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 операции union, определенной над двумя множествами, является множество, т.е. в этом случае возможные дубликаты должны быть исключены.

Примеры инвариантов

Рассмотрим примеры инвариантов на основе диаграммы классов на рис. 81. Обратим внимание, в OCL ключевое слово «self» обозначает текущий объект класса-контекста инварианта.

Рис. 81: Диаграмма классов для примеров инвариантов

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

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

Данный инвариант накладывает ограничение на значения атрибута «возраст» класса Служащий. При проверке данного условия будут перебираться существующие объекты класса Служащий, и для каждого объекта будет проверяться, что значения атрибута «возраст» находятся в пределах заданного диапазона. Ограничение удовлетворяется, если условное выражение принимает значение true для каждого объекта класса.

Пример. Определить ограничение «в отделах с номерами больше 5 должны работать сотрудники старше 30 лет»:

```
context Отдел inv:
self.номер <= 5 or
self.служащий → select (возраст <= 30) → count () = 0
```

Условное выражение инварианта будет вычисляться для каждого объекта класса Отдел. Подвыражение после операции or (в последней строке) вычисляется слева направо. Сначала вычисляется подвыражение `self.служащий`, значением которого является множество объектов, соответствующих служащим, которые работают в

текущем отделе. Далее к этому множеству применяется операция `select (возраст > 30)`, в результате которой вырабатывается множество объектов, соответствующих служащим текущего отдела, возраст которых превышает 30 лет.

Лекция 20

Значением операции count () является число объектов в этом множестве. Все выражение принимает значение true, если последняя операция сравнения «=0» вырабатывает значение true, т.е. если в текущем отделе нет сотрудников младше 31 года. Ограничение в целом удовлетворяется только в том случае, если значением условия инварианта является true для каждого отдела.

Тот же инвариант можно сформулировать в контексте класса Служащий:

```
context Служащий inv:
self.возраст > 30 or self.отдел.номер <= 5
```

Здесь следует обратить внимание на подвыражение `self.отдел.номер <= 5`. Поскольку отдел – это имя роли ассоциации, значением подвыражения `self.отдел` является коллекция (множество). Но кратность роли отдел равна единице, т.е. каждому объекту служащего соответствует в точности один объект отдела. Поэтому в OCL допускается сокращенная запись операции `self.отдел.номер`, значением которой является номер отдела текущего служащего.

Пример. Определить ограничение, что у каждого отдела должен иметься 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
```

Здесь «должность» является атрибутом класса Служащий, а атрибуты с именем «годОснования» имеются и у класса Отдел, и у класса Компания. Обратим внимание, что следующие два выражения эквиваленты:

- `self.служащий → exists (должность = ‘‘manager’’)`
- `self.служащий → select (должность = ‘‘manager’’) → count () > 1`

Пример. Определить ограничение максимально возможного количества сотрудников компании тысячей.

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

Здесь полезно обратить внимание на использование операции `collect`. Проследим за вычислением условного выражения. В результате выполнения операции `self.отдел` будет получено множество объектов, соответствующих всем отделам компании. При выполнении операции `collect (служащие)` для каждого объекта-отдела через экземпляр ассоциации будет образовано множество объектов-служащих данного отдела. А в результате самой операции `collect` будет получено множество всех служащих компании.

Плюсы и минусы использования языка OCL при проектировании РБД

Язык OCL позволяет формально и однозначно (без двусмысленностей, свойственных естественным языкам) определять ограничения целостности БД в терминах её 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.

На момент чтения курса никто не доказал и не опроверг утверждение, что на языке OCL можно выразить любое ограничение целостности, которое можно определить средствами SQL, или 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.

Получение схемы РБД из диаграммы классов UML

Если не обращать внимания на различия в терминологии, то здесь 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.

1) Прежде чем определять в классах операции, следует подумать, что 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.

2) Следует помнить, 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.

3) В спецификации UML говорится, 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, т.е. двунаправленные связи предпочтительнее.

Заключение

Нельзя сказать, что проектирование баз данных на основе семантических моделей в любом случае ускоряет или упрощает процесс проектирования. Все зависит от сложности предметной области, квалификации проектировщика и какие есть вспомогательные программные средства.

На раннем этапе проектирования до привязки к конкретной РСУБД проектировщик может обнаружить и исправить логические недочеты проекта, руководствуясь наглядным графическим представлением концептуальной схемы. Окончательный (или промежуточный) вид концептуальной схемы, полученной непосредственно перед переходом к формированию реляционной схемы, должен стать частью документации целевой реляционной БД. Наличие этой документации очень полезно для сопровождения и для изменения схемы БД в связи с изменившимися требованиями.

При использовании CASE-средств концептуальное моделирование БД может стать частью всего процесса проектирования целевой информационной системы, что должно способствовать правильной структуризации процесса, эффективности и повышению качества проекта в целом.

В контексте проектирования реляционных БД структурные методы проектирования, основанные на использовании ER-диаграмм, и OO-методы, основанные на использовании языка UML, различаются, главным образом, лишь терминологией. ER-модель концептуально проще UML, в ней меньше понятий, терминов, вариантов применения. И это понятно, поскольку разные варианты ER-моделей разрабатывались именно для поддержки проектирования реляционных БД, и ER-модели почти не содержат возможностей, выходящих за пределы реальных потребностей проектировщика реляционной БД.

Язык UML более универсальный, и не ограничивается только возможностями диаграмм классов. Поскольку UML может использоваться для унифицированного OO-моделирования всего чего угодно, в этом языке содержится масса различных понятий, терминов и вариантов использования, избыточных с точки зрения проектирования реляционных БД. Если вычленить из общего механизма диаграмм классов то, что действительно требуется для проектирования реляционных БД, то мы получим в точности ER-диаграммы с другой нотацией и терминологией. Поэтому выбор конкретной концептуальной модели – это вопрос вкуса и сложившихся обстоятельств. Например, если в вашей компании принято использовать UML при проектировании, то предпочтительнее и при проектировании БД использовать диаграммы классов. Тогда будет единая унифицированная документация для разных частей приложения.

Структуры данных в SQL-ориентированной СУБД

В большинстве случаев те методы, которые используются в современных СУБД, основаны на идеях, которые были заложены в экспериментальном проекте System R, хотя в любой развитой СУБД имеются собственные приёмы, которые обсуждаться в курсе не будут. Автор курса рекомендует к ознакомлению сайт по этой СУБД, который ведёт Paul McJones - один из участников System R (mcjones.org/System_R/). Для большего понимания данной главы автор также рекомендует к прочтению статью:

- С.Д. Кузнецов «Воссоединение SQL в 1995 г.: люди, проекты, политика» под ред. Пола МакДжонса (ссылка на статью)

Общие принципы организации данных во внешней памяти

Обсудим основные подходы к организации данных во внешней памяти, принятые в современных SQL-ориентированных СУБД. SQL-ориентированные СУБД обладают рядом особенностей, влияющих на организацию внешней памяти. Наиболее важными являются следующие особенности:

1) Наличие двух уровней системы:
   - уровня непосредственного управления данными во внешней памяти, а также обычно управления буферами оперативной памяти, управления транзакциями и журнализацией изменений БД
   - языкового уровня уровня, реализующего язык SQL
2) Поддержка таблиц-каталогов. Базы данных SQL являются самоописываемыми, т.е. все метаданные, описывающие структуру таблиц, ограничения целостности и другие объекты базы данных, являются частью БД и хранятся в отдельных таблицах-каталогах, которые ничем не отличаются от обычной таблицы БД. Т.е. вся обработка SQL в подсистеме верхнего уровня СУБД (например, проверка ограничений целостности) происходит в контексте таблиц-каталогов.
3) Регулярность структур данных. Поскольку основным объектом модели данных SQL является плоская таблица, основной набор объектов внешней памяти может иметь очень простую регулярную структуру. При этом эта структура должна поддерживать возможности эффективного выполнения операций SQL языкового уровня, как и простые операции фильтрации, так и трудоёмкое соединение нескольких таблиц. Для обеспечения такой эффективности во внешней памяти должны поддерживаться дополнительные «управляющие» структуры – индексы.
4) Наконец, для выполнения требования надежного хранения баз данных необходимо поддерживать избыточность хранения данных, что обычно реализуется в виде журнала изменений базы данных.

Соответственно, возникают следующие разновидности объектов во внешней памяти базы данных:

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

Хранение таблиц

Существуют два принципиальных подхода к физическому хранению таблиц. Наиболее распространенным является построчное хранение таблиц, при котором единицей физического хранения является строка таблицы. Требуется, чтобы строка целиком хранилась в одном блоке внешней памяти (нельзя половину строки хранить в одном блоке, а другую половину - в другом). Естественно, это обеспечивает быстрый доступ к целой строке (за один обмен с внешней памятью), но при этом во внешней памяти дублируются общие значения разных строк одной таблицы, и могут потребоваться лишние обмены с внешней памятью, если нужна часть строки (т.е. когда при каком-то `SELECT` выбираются на все столбцы, а какое-то маленькое их подмножество; но считываться строка всё равно будет целиком).

Альтернативным (менее распространенным) подходом является хранение таблицы по столбцам, т.е. единицей хранения является столбец таблицы с исключенными дубликатами. Естественно, при такой организации суммарно в среднем тратится меньше внешней памяти, поскольку дубликаты значений не хранятся, а также за один обмен с внешней памятью считывается больше полезной информации.

Дополнительным преимуществом хранения таблиц по столбцам является возможность использования значений столбца таблицы для оптимизации выполнения операций соединения. Мы не будем останавливаться причинах ускорения операций соединения, но поверьте на слово, что при хранении таблиц по столбцам уменьшается мощность промежуточных результатов при выполнении операции соединения. Также можно быстро считать значения агрегатных функций (таких как минимальное значение, максимальное значение, среднее, count distinct и других). Но при этом требуются существенные дополнительные действия для сборки целой строки (или его части). Для этого надо держать дополнительные структуры данных, которые позволяют собрать строку из столбцов.

Поскольку более распространено хранение по строкам, рассмотрим более подробно этот способ хранения таблиц. Для простоты будем считать, что в каждой странице данных хранятся строки только одной таблицы. Страница данных - это блок данных (во внешней памяти), в котором хранятся строки каких-то таблиц. Типовая структура страницы данных показа на рис. 82.

Каждая строка обладает уникальным идентификатором `tid=<N,i>`, где N - номер страницы, i - индекс, показывающий смещение от начала блока до описателя i-й строки таблицы. В описателе строки хранится ссылка на начало строки в этой же странице.

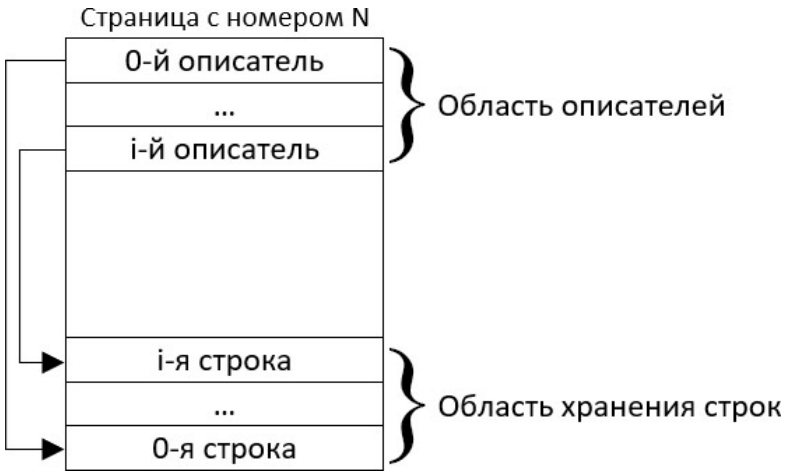


Рис. 82: Типовая структура страницы данных

Аналогично тому, как размер таблицы открытых файлов в одном процессе в ос UNIX ограничен, можно было бы ограничить каким-то числом количество строк в странице (например, строго 15). Но такое ограничение неосмысленно, т.к. в одной и той же таблицы могут быть строки переменной длины (например, со столбцом типа varchar). Т.е. нужно, чтобы количество строк в одной странице данных было динамическим, т.е. чтобы области описателей и область хранения строк были динамическими. Эта динамичность реализуется с помощью двух указателей (рис. 83): область описателей растёт сверху вниз, а область хранения строк таблицы снизу вверх.

Рис. 83: Стрелки указывают направление расширения областей

Обычно каждая строка хранится целиком в одной странице, из чего следует, что максимальная длина строки любой таблицы ограничена размерами страницы. Возникает вопрос: как хранить «длинные» данные, которые, в принципе, не помещаются в одной странице (например, аудио- и видеоданные, большие тексты)?

Наиболее простым решением является хранение таких данных в отдельных (вне базы данных) файлах с заменой «длинного» данного в строке на имя соответствующего файла. В некоторых системах такие данные хранились внутри базы данных в отдельном наборе страниц внешней памяти, связанном физическими ссылками. Оба эти решения ставят под вопрос как надёжность хранения данных, так и сильно ограничивают работу с длинными данными: как, например, удалить несколько байт из середины 2-мегабайтной строки?

Когда-то в проекте Exodus был предложен метод, который позволял длинные данные хранить на основе B-деревьев, о которых речь пойдёт далее. На сегодняшний день используется подход, зародившийся в компании Microsoft в начале 2000-х гг. Они решили поддерживать внутреннюю файловую систему внутри SQL сервера (прям там же, где хранится БД). И сегодня такие внутренние ФС используются для хранения blob (двоичной строки произвольной длины) и clob (символьной строки произвольной длины). Это решение не идеальное, но тем не менее изменение маленькой подстроки будет затрагивать только изменение блока, а не всего файла, и обеспечивается надёжность за счёт журнализации.

Рассмотрим ещё один момент в поддержке строк переменной длины. Допустим, в нашей строке используется столбец с типом varchar (символьная строка переменной длины с ограничением сверху). Типа varchar отличается от типа char тем, что первый займёт места в памяти ровно столько, сколько символов в строке, тип char, с другой стороны, всегда занимает выделенное место, даже если по факту использовано меньшее количество символов. Тогда может возникнуть ситуация, когда (i+1)-я строка (после операции изменения, например) настолько расширяется, что она целиком уже не помещается на данной странице (рис. 84). В этом случае в (i+1)-м описателе хранится новый идентификатор `tid=<M,k>`, где M - номер новой страницы.

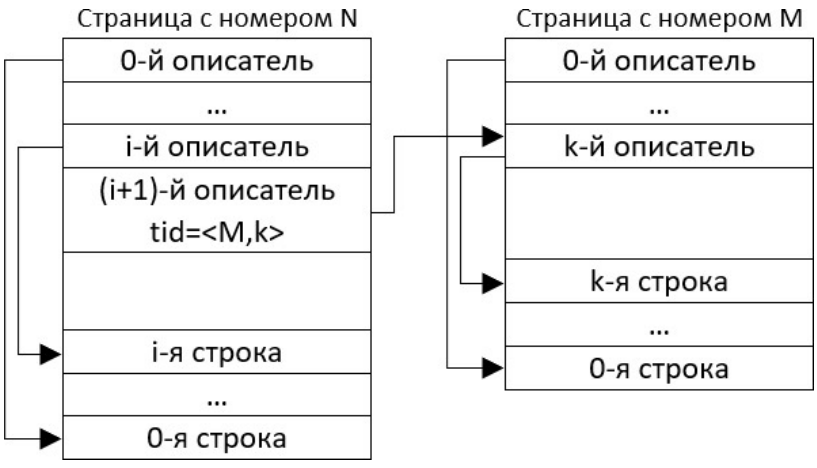


Рис. 84: (i+1)-й описатель указывает на k-й описатель страницы M

Если в какой-то момент времени места опять не будет хватать, то нужно освободить место на странице с номером M и поменять tid в странице N на некоторую новую страницу. Т.е. более чем один уровень косвенности не используется.

Как правило, в одной странице данных хранятся строки только одной таблицы. Однако существуют варианты с возможностью хранения в одной странице строк нескольких таблиц. Это повышает эффективность некоторых операций: например, если мы разрешаем хранить строки отделов рядом со строками служащих, которые работают в этих отделах, то немного быстрее выполняется операция соединения. Однако хранение строк нескольких таблице требует некоторой дополнительной служебной информации (при каждой строке надо хранить информацию к какой таблице эта строка относится).

При добавлении нового столбца в таблицу не происходит физической реорганизации таблицы. Достаточно лишь в описателе строки добавить новый столбец и расширять строки только при занесении информации в новое поле. Поскольку таблицы могут содержать неопределенные значения, необходима соответствующая поддержка на уровне хранения. Обычно для этого используют шкалу (boolean) при каждой строке таблицы: например, 0 обозначает, что столбец в строке существует и хранит данные, а 1 - что в данной строке данный столбец хранит неопределённое значение.

Не тривиальна проблема распределения памяти в страницах данных. Представим, что какая-то транзакция выполняет массовую операцию `DELETE`, т.е. некоторый блок на диске становится пустым. Спрашивается что с таким блоком делать? Если его обозначить свободным во время работы данной транзакции, то другая транзакция (например, которая вставляет строки) может использовать этот блок. Тогда первая транзакция, которая удаляет строки, может по каким-то причинам откатиться (ROLLBACK), и надо восстановить все удалённые строки причём с теми же самыми идентификаторами, чего сделать уже не получится.

Распространенным способом повышения эффективности СУБД является кластеризация таблицы по значениям одного или нескольких столбцов. Кластеризация - это указание, что в одном блоке внешней памяти надо хранить те строки таблицы, у которых значения столбца кластеризации одинаковые или близкие. Если таблица кластеризована, то некоторые запросы выполняются на несколько порядков быстрее, чем если бы строки были расположены произвольным образом. Однако кластеризацию поддерживать всегда невозможно, любая кластеризованная таблица по мере её роста деградирует, т.е. свойства кластеризации ухудшаются со временем. И рано или поздно приходится делать физическую реструктуризацию кластеризованной таблицы.

Полезной для оптимизации соединений является совместная кластеризация нескольких таблиц. Например, нам известно, что часто выполняется естественное соединение отделов со служащими. Тогда очень полезно кластеризовать совместно эти таблицы по столбцу «номер отдела», и если нужно получить служащих отдела номер 653, то это делается за одно чтение внешнего блока, в котором будут и строка отдела, и строки всех служащих, которые в данном отделе работают.

С целью использования возможностей распараллеливания обменов с внешней памятью иногда применяют схему декластеризованного хранения таблиц. В этом случае, наоборот, строки с общим столбцом декластеризации размещают на разных дисковых устройствах, с которыми можно работать параллельно. Например, если у нас много-много служащих, то их можно раскидать на несколько блоков на несколько дисков и прочитать всех служащих за один обмен с каждым диском.

Что же касается хранения таблицы по столбцам, то основная идея состоит в совместном хранении всех значений одного (или нескольких) столбцов. Для каждой строки таблицы хранится строка той же степени, которая состоит из ссылок на места расположения соответствующих значений столбцов. Хранение таблицы по столбцам хоть и повышает операцию с некоторыми агрегатными функциями

Когда-то в 1970-е гг. идея вертикального partitioning была рождена в компании IBM. Они создавали проект вслед за System R под названием System R*, который был, вероятно, первой распределённой СУБД. Они предлагали возможность хранить в разных узлах распределённой сети, в которой работает СУБД, не целиком таблицы, а их разделы. Они предлагали делать разбиение как по горизонтали (о чём сказано двумя абзацами выше), так и по вертикали, при которой таблица хранится по проекциям. По факту, хранение таблицы по столбцам - это хранение таблицы по проекциям.

Хранение таблицы по строкам наиболее полезно в аналитических системах (OLAP системах), в которых одна таблица может содержать тысячу столбцов. Причём при реальной работе требуется лишь небольшое число столбцов. Так что считывать строку целиком в таких системах очень дорого.

В конце 1990-х гг. в компании Microsoft была серия работ, посвященной динамической самонастройки СУБД. В этих работах использовались методы интеллектуального анализа данных для определения того, не требуется ли создать внутри SQL сервера индекс для какой-то таблицы (на основе текущих запросов к БД). Более конкретно, просматривались журналы запросов, оценивалось как бы прошла работа, если бы были некоторые индексы и принималось решение о создании индексов на каких-то столбцах.

Компромисс между хранением таблиц по строкам или по столбцам предлагается в системе Peloton, которая разрабатывается Joy Arulraj, Andy Pavlo и Prashanth Menon в Carnegie Mellon University. Ключевая особенность их СУБД в том, что она самоуправляемая (self-driven), т.е. система сама подстраивается под изменение рабочей нагрузки. Их СУБД сама балансирует между хранением по строкам, хранением по столбцам, хранением части таблицы в виде строк и части в виде столбцов в зависимости от рабочей нагрузки. Стоит отметить, что их СУБД не дисковая, а in-memory (т.е. вся БД хранится в оперативной памяти). С их работой можно ознакомиться по ссылке:

- J. Arulraj, A. Pavlo, and P. Menon, "Bridging the Archipelago Between Row-Stores and Column-Stores for Hybrid Workloads,"in Proceedings of the 2016 International Conference on Management of Data, 2016, pp. 583-598 (ссылка)

Подводя итоги, хранение таблиц по строкам предпочтительно для транзакционных систем, а хранение по столбцам - для аналитических (OLAP) систем. Существуют так же гибридные варианты, когда СУБД пытаются оптимизировать как для работы аналитической, так и транзакционной (HLAP системы), однако автор курса считает, что HLAP системы будут проигрывать при чисто аналитической работе OLAP системам, и при чисто транзакционной работе транзакционным системам.

Индексы

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

Поскольку при выполнении многих операций уровня SQL требуется сортировка строк таблиц в соответствии со значениями некоторых столбцов, полезным свойством индекса является обеспечение последовательного просмотра строк таблицы в заданном диапазоне значений ключа в порядке возрастания или убывания значений ключа. Наконец, одним из способов оптимизации выполнения эквисоединения таблиц (наиболее распространенная из числа дорогостоящих операций) является организация так называемых мультииндексов для нескольких таблиц, обладающих общими атрибутами. Любой из этих атрибутов (или их набор) может выступать в качестве ключа мультииндекса. Значению ключа сопоставляется набор строк всех связанных мультииндексом таблиц, значения выделенных атрибутов которых совпадают со значением ключа.

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

B+-деревья

Наиболее популярным методом организации индексов в базах данных является использование техники B+ деревьев. Техника B- и B+ деревьев была предложена

в начале 1970-х гг. Рудольфом Байером (Rudolf Bayer) и Эдом Маккрейтом (Ed McCreight):

- Rudolf Bayer. R. Bayer, E. McCreight. Organization and Maintenance of Large Ordered Indexes. Acta Informatica, Vol. 1, Fasc. 3, 1972 pp. 173-189 (ссылка)

Структура B+-дерева

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. Это очень важно запомнить и отличать от сбалансированности AVL-дерева (двоичных деревьев в оперативной памяти), в которых сбалансированность означает, что длина пути от корня дерева к любому его листу отличается не более чем на 1.

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).

Недостаток B-деревьев в том, что их трудно балансировать. В курсе этот вопрос не освещается, но в автор рекомендует книгу «Искусство программирования» Дональда Э. Кнута, где подробно описывается процесс балансировки B-дерева.

Байер и Маккрейт придумали B+-дерево, которое является модификацией B-дерева. Преимущество B+-дерева в том, что его достаточно просто балансировать (алгоритм балансировки рассмотрим далее). В B+-дереве внутренние и листовые страницы обычно имеют разную структуру. Типовая структура внутренний страницы страницы B+-дерева выглядит как упорядоченная последовательность ключей и ссылок на страницы более низкого уровня:

N_1 `ключ`$_1$ N_2 `ключ`$_2$ ... N_m `ключ`$_m$ N_{m+1}

N_i - это ссылки на узлы дерева более низкого уровня. Причём выполняются свойства упорядоченности:

- `ключ`$_1 \leq$ `ключ`$_2 \leq ... \leq$ `ключ`$_m$
- в странице N_i находятся ключи k такие, что ключ$_i \leq k \leq$ключ$_{i+1}$

Второе свойство означает, что в странице, которая зажата между `ключ`$_k$ и `ключ`$_{k+1}$ все ключи тоже будут в диапазоне `[ключ`$_k$`; ключ`$_{k+1}$`]`.

Типовая структура листовой страницы B+-дерева выглядит как упорядоченная последовательность ключей и списков tid (идентификаторов) строк:

`ключ`$_1$ `список`$_1$ `ключ`$_2$ `список`$_2$ ... `ключ`$_k$ `список`$_k$

В листовой странице ключи тоже упорядоченны: `ключ`$_1 \leq$ `ключ`$_2 \leq ... \leq$ `ключ`$_m$. `список`$_r$ - это упорядоченный список идентификаторов строк (tid), включающих значение `ключ`$_r$.

Поиск в B+-дереве – это прохождение от корня к листу в соответствии с заданным значением ключа. Т.е. сначала ищем в корне дерева первый ключ который будет больше или равен заданному, далее идём по ссылке слева от этого ключа. Процесс повторяется, пока мы не доберёмся до листовой вершины и найдём нужный список tid-ов слева от ключа, большего или равного заданному.

Заметим, что поскольку B+-деревья являются сильно ветвистыми и сбалансированными, для выполнения поиска по любому значению ключа потребуется одно и то же (из-за сбалансированности) и обычно небольшое (из-за ветвистости) число обменов с внешней памятью. Более точно, в сбалансированном дереве, где длины всех путей от корня к листу одни и те же, если во внутренней странице помещается n ключей, то при хранении m записей требуется дерево глубиной $log_n(m)$.

Операция вставки записи в B+-дерево

Основной «изюминкой» B+-деревьев является автоматическое поддержание свойства сбалансированности. Рассмотрим, как это делается при выполнении операций вставки (`INSERT`) и удаления (`DELETE`) записей (запись - это пара `<ключ, tid>`). При вставки новой записи выполняются следующие действия:

1) Поиск листовой страницы (производится поиск по ключу).
   - Если в B+-дереве не содержится ключ с заданным значением, то будет получен номер страницы, где нужно такой ключ содержать, и соответствующие координаты внутри страницы.
2) Помещение записи на место. Естественно, что вся работа производится в буферах оперативной памяти (обычно размер буфера берут вдвое больше размера страницы внешней памяти). Листовая страница, в которую требуется занести запись, считывается в буфер, и в нем выполняется операция вставки.
   - Если после выполнения вставки новой записи размер используемой части буфера не превосходит размера страницы, то на этом выполнение операции занесения записи заканчивается. Буфер может быть немедленно вытолкнут во внешнюю память, или временно сохранен в оперативной памяти в зависимости от политики управления буферами.
   - Если же возникло переполнение буфера (т.е. размер его используемой части превосходит размер страницы), то выполняется расщепление страницы (splitting). 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, при этом изменяются ссылки по списку листовых страниц.

     Мы получили новую страницу, но не сказали как до неё добраться от корня. 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 внутренняя страница тоже может переполниться, и её тоже нужно расщепить на две. В результате потребуется вставить ключ и ссылку на новую страницу в страницу выше по иерархии и так далее вплоть до корня. Предельным случаем является переполнение корневой страницы B+-дерева. В этом случае корневая страница тоже расщепляется на две, и заводится новая корневая страница дерева, т.е. его глубина увеличивается на единицу, но дерево остаётся сбалансированным.

Обратим внимание, что расщепление вплоть до корня происходит очень редко. Как правило, восстановление сбалансированности происходит путём изменения небольшого поддерева. На практике очень редко расщепление проходит на несколько уровней, а тем более до корня, за счёт чего B+-деревья оказываются эффективнее обычных B-деревьев.

Лекция 21

Операция удаления записи из B+-дерева

При удалении записи из B+-дерева выполняется поиск записи по ключу:

- Если запись не найдена, то ничего удалять не нужно.
- Если запись найдена, то мы попадаем в соответствующую листовую страницу, и происходит реальное удаление записи в буфере, в который прочитана соответствующая листовая страница.
  - Если после выполнения удаления суммарный размер занятой памяти в буфере и размер занятой памяти в листовой странице, являющейся левым или правым братом данной страницы, больше, чем размер страницы, то операция завершается.
  - Если же суммарная занятая память меньше одной страницы, то производится слияние (merging) с правым или левым братом, т.е. в буфере производится новый образ страницы, содержащей общую информацию из данной страницы и её левого или правого брата. Ставшая ненужной листовая страница заносится в список свободных страниц. Соответствующим образом корректируется список листовых страниц, чтобы там не присутствовала удалённая страница.

    Чтобы устранить возможность доступа от корня к освобожденной странице, нужно удалить соответствующее значение ключа и ссылку на освобожденную страницу из внутренней страницы – её предка. Опять же может получиться, что внутренняя страница будет занимать мало места и её нужно слить с её левым или правым братом 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.

Самое привлекательное в операциях вставки и удаления записей в B+-дерево, что проходы до корня очень редкие, т.е. как правило операции слияния (merging) и расщепления (splitting) страницы не происходят локально (не более, чем на два уровня). В этом основное отличие B+-деревьев от B-деревьев, в которых балансировка дерева почти всегда задействует всё дерево.

Приёмы повышения эффективности B+-дерева

Проблемой является то, что при выполнении операций модификации слишком часто могут возникать расщепления и слияния. Причиной этому слишком «жёсткие» условия расщепления (когда не помещается новый ключ в страницу) и слияния (когда сумма занятых данных в текущей странице и в правом или левом брате меньше чем размер страницы).

Чтобы добиться эффективного использования внешней памяти с минимизацией числа расщеплений и слияний эмпирически было выявлено, что нужно делать упреждающие слияния, т.е. слияния, когда суммарный размер данных в текущей странице и левом или правом брате не в точности меньше страницы, а, например, 70% страницы. Аналогично с упреждающими расщеплениями: расщепление страницы производить не при её точном переполнении, а несколько раньше, когда остаётся мало свободного места (сколько именно это «мало» можно узнать только на практике).

Другая операция, которая позволяет поддерживать балансировку дерева без расщепления и переливания - это переливание. Переливание заключается в том, что если какой-либо блок переполняется, а его левый или правый брат недозаполнены, то с ними происходит перераспределение. Причём никаких новых страниц не создаётся (как правило достаточно откорректировать какие-то ссылки). Если места нет ни у страницы, ни у братьев, то уже тогда происходит расщепление. Проведём житейскую аналогию всех трёх операций:

- Расщепление - это аналог покупки новой квартиры, когда в старой стало невозможно тесно.
- Слияние - это, наоборот, когда все переехали из обоих квартир, а оставшейся семье приходится только деньги платить за квартиру, которая никому не нужна (т.е. можно жить в одной квартире, а вторую освободить).
- Переливание - это более простой случай: живёт семья в нескольких квартирах, в какой-то квартире становится тесно. Если в других место есть, то выравнивается число жителей между квартирами, чтобы всем было удобно.

Возможны также такие приёмы как слияние «три в два» и расщепление «два в три»:

- Слияния «три в два» - порождение двух листовых страниц на основе содержимого трех соседних.
- Расщепление «два в три» - расщепляется текущая и левый или правый брат на три страницы.

Все эти подходы направлены на уменьшение числа слияний и расщеплений, а тем самым, уменьшение числа обращений к диску. Обратной стороной этих методов является менее экономное использование дисковой памяти.

Одна из возможностей индексирования состоит в том, что можно использовать не атомарные ключи (ключи, соответствующие одному столбцу таблицы), а составные ключи. Например, если требуется часто делать поиск по номеру отдела и размеру

заработной платы, то можно сделать составной индекс по двум столбцам. Тогда прямой поиск по такому индексу будет происходить очень просто. Понятно, что если у каждого типа атомарного столбца поддерживается упорядоченность, то для любого составного ключа упорядоченность определяется как лексикографическая.

Из-за лексикографического порядка получается, что в каждой странице дерева с очень большой вероятностью рядом будут находиться составные значения ключа, у которых точно такое же значение первого элемента ключа. Т.е. возникает группировка аналогичная `GROUP BY` в SQL: для каждой группы с одинаковым значением первого элемента составного ключа может быть подгруппа с одинаковым значением второго элемента и т.д. Естественно, возникает желание не хранить так много, а вынести общий префикс «за скобки», но эта экономия места вырождается в то, что поиск происходит дольше. Т.е. мы фактически сделали «упаковку», которую при каждом поиске нужно будет распаковывать. Здесь приходится решать что же важнее - эффективный поиск или экономия памяти (а она может быть очень большой).

Отметим одно важно отличие между уникальными (т.е. когда создаётся B-дерево для возможного ключа) и неуникальными B-деревьями. В случае уникального B-дерева оно берёт на себя очень важную функцию проверки дубликатов. Для определения дубликатов нужно просто найти нет ли записи с таким значением ключа в B-дереве, т.е. мы нагружаем B-дерево дополнительной функцией. Оказывается, что уникальные B-деревья выглядят наиболее естественно, чем те, у которых разрешены дубликаты. Рассмотрим что происходит, когда создаётся неуникальное B-дерево. Представим такой предельный случай: в таблице `СЛУЖАЩИЕ` создаётся неуникальный индекс по столбцу `ПОЛ`. Теперь у нас (примерно) половина при значении «мужской» и половина при значении «женский», и все tid строк будут содержать повторяющиеся значения ключа `мужской, мужской, мужской, ...` или `женский, женский, женский, ...`. Если их точно поровну, то может оказаться, что имеется корень с двумя потомками: в одном - «мужской», в другом - «женский», и все их поддеревья содержат только «мужской» или только «женский».

Как видно, неуникальные B-деревья могут вырождаться и не приносить особого улучшения при поиске. Понять когда B-дерево становится бессмысленно, видимо, можно только во время обработки запросов.

Множественный доступ к B+-дереву

Также очень нетривиально работать с B+-деревьями в режиме мультидоступа. Например, в двух транзакциях параллельно (concurrently) в одном и то же B-дерева выполняется операция вставки. Конечно, грубые решения очевидны, например, возможен монопольный захват B+-дерева (т.е. блокировать корневую страницу) на всё выполнение операции модификации. Однако B+-дерево сильно ветвистое и вероятность того, что реально в двух транзакциях операции вставки будут реально конфликтовать, очень маленькая. На практике расщепление происходит редко, а если и происходит, то как правило локально, т.е. до корня не распространяется. А

как узнать заранее насколько вверх поднимется расщепление B+-дерева (т.е. откуда действительно нужно делать блокировку)?

В GNU SQL Server был придумано следующее решение. Пусть происходит операция вставки tid в B-дерева.

1) Проигрывается операция вставки. Т.е. ничего нигде не меняется, но происходит поиск по дереву (без синхронизации) нужного места куда вставлять. Проигрывается операция расщепления. Смотрим куда расщепления распространились бы, если бы блокировки стояли и запоминаем номер соответствующей страницы.
2) От корня с блокировками ищем ключ, а именно: блокируем корень и затем спускаемся на следующую внутреннюю страницу и блокируем её, а с предыдущей (корневой страницы) снимаем блокировку. И т.д., пока не доберёмся до того узла, который мы заблокировали на первом этапе.
   - Если не добираемся, то начинаем играть заново 1й этап.
   - Если добрались, то значит, что дерево не изменилось на нашем пути. Проигрываем операцию снизу и смотрим куда теперь пройдут изменения. Если изменения не доходят выше заблокированного узла, то выполняем операцию. Если вдруг B-дерево успело изменить так, что изменения пойдут выше блокированного узла, то снимаем блокировки и начинаем заново с первого этапа.

На практике оказалось, что как правило ничего не меняется за время первого раунда и очень редко возникает второй раунд, третий не наблюдался никогда.

Существуют и более тонкие решения, рассмотрение которых выходит за пределы материала этой лекции:

- Goetz Graefe, A survey of B-tree locking techniques, ACM Transactions on Database Systems (TODS), v.35 n.3, p.1-26, July 2010 (ссылка)

Интерфейс RSS

Описываемый в данном курсе интерфейс RSS не соответствует в точности ни одной из публикаций, посвященных System R, а является скорее некоторой компиляцией, согласующейся с завершающими публикациями.

Интерфейс RSS также называют ядром СУБД System R - это именно та подсистема, к которой обращается языковая подсистема верхнего уровня, реализующая SQL, для выполнения операций изменений БД. Можно считать, что это развитая файловая система специализированная для потребностей СУБД. Т.е. она поддерживает индексные структуры данных, обращается непосредственно к внешней памяти, поддерживает буферизацию, управляет транзакциями, ведёт журнализацию операций. И причём даже сам способ работы напоминает работу с файловой системой.

Можно считать, что интерфейс RSS это некоторая абстрактная машина, на которой реализуется SQL.

На уровне RSS отсутствует именование объектов базы данных, которые используются на уровне SQL. Система избавляется от имён на стадии лексического анализа запросов. Вместо имен объектов используются их уникальные идентификаторы, являющиеся прямыми или косвенными адресами внутренних описателей объектов на внешней памяти для постоянных объектов или в оперативной памяти для временных объектов. Когда система видит идентификатор, то по его структуре видно идентификатором чего он является (таблицы, столбца и т.д.). Замена имен объектов базы данных на их идентификаторы производится компилятором SQL на основе информации, черпаемой им из системных таблиц-каталогов, которые содержат метаданные, описывающие таблицы базы данных. В таблице-каталоге помимо имени соответствующего объекта базы данных помимо всего прочего есть внутренний идентификатор этого объекта, который понимает RSS, но не понимает SQL.

Рассмотрим операции в интерфейсе RSS. Выделяют 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 столбца к таблице
- операции управления прохождением транзакций
- операция явной синхронизации

Операции из группы сканирования позволяют последовательно, в порядке, определяемом типом сканирования, прочитать строки таблицы или списка, удовлетворяющие требуемым условиям. Группа сканирования включает операции `OPEN`, `NEXT` и `CLOSE`, которые означают:

- начало сканирования
- требование чтения следующей строки, удовлетворяющей условиям
- конец сканирования

На уровне RSS не существуют никакие системные таблицы (и представляемые таблицы). Т.е. RSS работает только с реальными таблицами, в которых хранятся данные.

Прямое сканирование таблицы

Для каждой таблицы возможны два режима сканирования - прямое сканирование и сканирование через индекс.

Прямое сканирование - это последовательный просмотр строк таблицы в порядке, определённом системой. Фактически, последовательно читаются страницы внешней памяти, которые содержат строки таблицы, и последовательно (в порядке физического расположения) считываются строки таблицы.

Параметром операции `OPEN` является идентификатор таблицы, который точно говорит что это за таблица, т.е. включает ещё и идентификатор файла или сегмента, в котором эта таблица хранится. В System R допускалось размещение в одной странице данных строк нескольких таблиц, поэтому при прямом сканировании происходил последовательный просмотр всех страниц сегмента с выделением в них строк, входящих в данную таблицу.

Это был очень дорогой способ сканирования таблицы. Предположим, что нам требуется тысяча строк, а в сегменте хранится 10000 страниц данных. И мы читаем 10000 страниц ради этой тысячи строк.

Заметим, что если отказаться от хранения строк многих таблиц в одной странице данных, то ситуация становится гораздо более дешёвой. Т.к. можно связать страницы одной таблицы в список и читать не все страницы в сегменте, а только те, на которых теоретически могут находиться строки нужной таблицы. Худшим случаем при считывании 1000 строк будет считывание 1000 страниц данных (если в одной странице только один одна строка), но такого, конечно же, на практике не бывает.

Сканирование таблицы через индекс

При начале сканирования таблицы через индекс в число параметров операции `OPEN` входит идентификатор какого-либо индекса, определенного ранее на столбцах этой таблицы. Кроме того, можно указать диапазон сканирования - левую и правую границу ключа этого индекса. При открытии сканирования через индекс производится начальная установка указателя сканирования в позицию листа B-дерева индекса, соответствующую левой границе заданного диапазона. Т.е. мы спускаемся от корня до нужной листовой страницы и находим там первое вхождение требуемого ключа. Процесс сканирования состоит в последовательном продвижении по листовым вершинам индекса до достижения правой границы диапазона сканирования с выборкой идентификаторов строк и чтением соответствующих строк.

Может показаться, что сканирование таблицы через индекс гораздо дешевле, чем прямой просмотр, но это неверно. Т.к. для обычного индекса упорядоченность ведётся по значению ключа, а не по значению идентификатора строки (tid). Может казаться, что для одного значения ключа для каждого tid из списка тидов с этим значением ключа нам потребуется чтение страницы данных. В худшем случае придётся прочитать столько страниц, сколько идентификаторов строк мы встретим при просмотре заданного диапазона значений ключа.

Индексы - это единственное средство, которое позволяет резко сократить расходы на прямой доступ по ключу к нужной строке таблицы. Но обратной стороной медали является то, что поддержка поиска по диапазону очень условная, т.е. это

может оказаться даже дороже, чем прямой просмотр таблицы. Чем уже диапазон сканирования, тем эффективнее просмотр.

Если вспомним пример с двумя значениями ключа «мужской» и «женский», и будет указан диапазон сканирования от «женский» до «мужской», то это приведёт к тому, что будет просмотрено всё дерево (т.е. все списки идентификаторов строк) и это будет крайне неэффективно. Отметим, что данная проблема неэффективной работы индексов при широком диапазоне актуальна для обычных дисков, для дисков, работающих в 100 раз быстрее, чтение дополнительного блока памяти стоит недорого.

Список в RSS

Рассмотрим такую структуру данных в RSS как список. Список - это внутренняя структура данных, которую не видно в SQL, но она очень полезна при оптимизации запросов. Список - это последовательность страниц внешней памяти, которые хранят строки (или идентификаторы строк) таблиц. Список является временной структурой данных, которая может создаваться подсистемой верхнего уровня SQL при выполнении запроса. Список существует либо до явного уничтожения, либо пропадает при мягком сбое системы (мягкий сбой - если разом опустошится оперативная память).

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 строки из данного списка, и порядок сканирования совпадает с порядком занесения строк в список или порядком списка, если он упорядочен. В результате успешного выполнения операции открытия сканирования вырабатывается и возвращается идентификатор сканирования, который используется в качестве аргумента других операций этой группы.

Операции NEXT и CLOSE

Операция `NEXT` выполняет чтение следующей строки указанного сканирования, удовлетворяющего условию данной операции. Условие представляет собой дизъюнктивную нормальную форму простых условий, накладываемых на значения указанных столбцов таблицы. Простое условие – это условие вида `номер-поля op const`, где `op` – операция сравнения `<, <=, >, >=, =, !=`. Условие является параметром операции `NEXT`. Семантика операции `NEXT` следующая:

- Начиная с текущей позиции сканирования выбираются строки таблицы в порядке, определяемом типом сканирования, до тех пор, пока не встретится строка, значения столбцов которой удовлетворяют указанному условию. Эта строка и является результатом операции `NEXT`.
- Если при выборке строки достигается правая граница диапазона сканирования (правая граница значения ключа при сканировании через индекс или последняя

строка 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 `CLOSE` является идентификатор сканирования, и её выполнение приводит к тому, что этот идентификатор становится недействительным и, соответственно, уничтожаются служебные структуры памяти RSS, относящиеся к данному сканированию.

Операции создания и уничтожения объектов БД

Объекты в базе данных могут быть временными или постоянными. В число объектов входят таблицы, индексные структуры и списки.

- В число операций создания объектов БД входят создание таблиц (`CREATE TABLE`), списков (`CREATE LIST`) и индексов (`CREATE IMAGE`).
- В число операций уничтожения объектов БД входят уничтожение таблиц (`DROP TABLE`), списков (`DROP LIST`) и индексов (`DROP IMAGE`).

Входным параметром операций создания таблиц и списков является спецификатор структуры объекта, т.е. число столбцов и спецификаторы их типов. Кроме того, при спецификации столбцов таблицы указывается разрешение или запрещение наличия неопределенных значений столбцов в строках 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 оперативной 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.

Рассмотрим операцию `CREATE IMAGE`. Её входными параметрами являются:

- идентификатор таблицы, для которой создается индекс
- список номеров столбцов, значения которых составляют ключ индекса
- признаки упорядочения по возрастанию или убыванию для всех столбцов, составляющих ключ

- может быть указан признак уникальности индекса, т.е. запрет 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 её 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 может так оказаться, что мы почти построили индекс и вставляем последний ключ, и нарушается уникальность. В этом случае 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 её заполнения. Заметим, что создание кластеризованного индекса на непустой таблице требует полной физической реорганизации таблицы и часто СУБД запрещают его создание на непустой таблице.

Операции `DROP TABLE`, `DROP LIST` и `DROP IMAGE` могут быть выполнены в любой момент независимо от состояния объектов. Выполнение любой из этих операций 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. В случае с `CREATE IMAGE` всё более-менее просто: если нашёлся дубликат значения ключа, т.е. свойство уникальности не выполняется, то нужно просто освободить уже отведённые страницы данных под B-дерево.

Совсем другая ситуация с `DROP TABLE`. Рассмотрим как уничтожается таблица. Понятно, что её описатель нельзя уничтожить пока таблица реально содержит какие-то страницы данных (иначе ссылки на эти страницы будут безвозвратно потеряны, и они так и не будут высвобождены), т.е. надо удалить сначала данные, потом описатель таблицы. Допустим, мы хотим уничтожить таблицу из двух миллиардов строк, если мы будем удалять строки поштучно, то это займёт немалое количество времени и потребует журнал колоссального размера для обеспечения возможности отката при мягком сбое. Чтобы этого избежать, можно объявить операцию `DROP TABLE` нетранзакционной, т.е. она не журнализуется и нет возможности вернуть строки наполовину удалённой таблицы. Поэтому, хотя уничтожение непустых таблиц и не запрещено, нужно иметь в виду, что это очень дорогостоящая операция.

Операции модификации таблиц и списков

Группа операций модификации таблиц и списков включает операции:

- вставки строки в таблицу или список (`INSERT`)
- удаления строки из таблицы (`DELETE`)
- обновления строки в таблице (`UPDATE`)

Рассмотрим операцию вставки строки `INSERT`. Её параметрами являются идентификатор таблицы или списка и набор значений столбцов строки. Среди значений столбцов могут быть литеральные неопределенные значения NULL. Естественно, при выполнении вставки контролируется допустимость неопределенных значений в соответствующих столбцах.

- При вставке строки в кластеризованную таблицу, поиск места в сегменте под соответствующую строку происходит с использованием кластеризованного индекса, т.е. идёт поиск по B+-дереву кластеризованного индекса по ключу от корня, и система пытается вставить строку в ту страницу, где находятся строки с близкими значениями столбцов кластеризации.
- При занесении строки в некластеризованную таблицу место под строку выделяется в первой подходящей странице данных.
- Наконец, при вставке строки в список, она помещается в конец списка.

При занесении строки в таблицу производится коррекция всех индексов, определенных на этой таблице. Реально это выражается во вставке новой записи (пары `<ключ, tid>`) 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. Может оказаться, что есть на таблице 15 индексов и операция вставки в саму таблицу выполнена нормально, 14 индексов изменены нормально, а на пятнадцатом нарушается уникальность, и вся эта работа идёт коту под хвост. Поэтому сначала надо производить вставку в уникальные индексы. Из этого видно, что операция вставки строки тем более накладна, чем больше индексов определено для данной таблицы. Т.е. индексы хороши для операций выборки (чтения) данных, но вот операции изменения (и удаления) данных тем медленнее, чем индексов больше.

В результате успешного выполнения операции вставки строки в таблицу вырабатывается идентификатор новой строки, который выдается в качестве результата операции (не пользователю, а подсистеме верхнего уровня) и может быть в дальнейшем использован как прямой параметр операций удаления и модификации строк таблицы. При занесении строки в список значение идентификатора строки не вырабатывается:

- для списков допускается только последовательное сканирование, а новые строки добавляются в конец списка
- над ними нельзя определить индексов, и поэтому косвенная адресация элементов списков через их идентификаторы не требуется

Операции удаления и модификации строк допускаются только для таблиц (для списков нет этих операций). Естественно, что для выполнения этих операций необходимо идентифицировать соответствующую строку. В интерфейсе RSS допускаются два способа такой идентификации:

- С помощью идентификатора строки (явная адресация). Этот вариант возможен, т.к. идентификатор строки сообщается как ответный параметр операции занесения строки в постоянную таблицу.
- С использованием идентификатора открытого к этому времени сканирования. Имеется в виду строка, прочитанная с помощью последней операции `NEXT`.

Если при идентификации через сканирование выполняется операция `DELETE` (или `UPDATE`), которая задевает порядок сканирования (т.е. ведется сканирование по индексу и операция модификации меняет значение столбца в строке, входящее в состав ключа этого индекса), то текущая строка сканирования теряется. Другими словами, если представить операцию `SCAN` в виде стрелочки, то при удалении текущей строки она начинает показывать в дырку между двумя последовательными строками таблицы, которую мы через индекс просматриваем. И идентификатор, выданный операцией `NEXT` нельзя использовать для идентификации строки до выполнения следующей операции `NEXT`.

Лекция 22

У операции `DELETE` единственным параметром является идентификатор строки или идентификатор сканирования. У операции `UPDATE` помимо идентификатора должны быть номера и значения изменяемых столбцов строки. Среди значений могут находиться литеральные неопределенные значения NULL, если соответствующий столбец таблицы допускает хранение неопределенных значений.

При выполнении операции `DELETE` производится коррекция всех индексов, определенных на данной таблице. Операция `UPDATE` также может повлечь коррекцию индексов, если затрагивает столбцы, входящие в состав их ключей.

Операция построения списка BUILDLIST

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. Рассмотрим два случая в которых списки очень полезны:

- Списки очень полезны, если оптимизатор запросов SQL решил для операции эквисоединения использовать алгоритм sort match (стр. 96). Это хороший алгоритм, который позволяет эффективно пройти два предварительно отсортированных списка.
- Пусть нам нужно сделать запрос общего вида `SELECT ... UNION SELECT ...`. Одним из способов выполнения такого запроса является выполнение первого `SELECT`, затем выполнить второй запрос, после чего объединить результаты. Вообще говоря, для объединения результатов тоже необходима сортировка, иначе получится с дубликатами.

  Эффективный способ выполнения такой операции - это сделать два сортированных списка tid (идентификаторов строк) первого и второго `SELECT`, после чего объединить их алгоритмом sort match. Не только объединение, но и пересечение удобно делать через такие списки и т.д.

Однако посмотрим как много операций нужно произвести к интерфейсу RSS для построения списка:

1) открыть сканирование таблицы по которой хотим построить список
2) создать пустой список
3) идти по `SCAN` и выбирать строки до тех пор, пока не закончится сканирование
4) для каждой строки выполнить вставку в список, что мы строим

Для того, чтобы сократить накладные расходы для такого частого и предопределённого набора операций, существует макрооперация `BUILDLIST`, которая за одно

обращение к RSS, позволяет построить отсортированный список. Эта операция включает:

- сканирование заданной таблицы или списка
- создание нового списка, в который включаются указанные поля выбираемых строк
- сортировку построенного списка в соответствии со значениями указанных столбцов

Здесь важное место имеет сортировка. Понятно, что в общем случае сортировка внешняя (данные хранятся во внешней памяти, а не в оперативной). Обычно для внешней сортировки используются алгоритмы сортировки слияниями: весь набор данных разбивается на кусочки, чтобы каждый кусочек помещался в оперативной памяти. Далее тем или иным способом, храня единовременно в оперативной памяти лишь часть всей таблицы, строится отсортированный список. Сортировка работает тем быстрее, чем больше кусочек мы можем держать в оперативной памяти.

Параметрами операции `BUILDLIST` являются:

- набор параметров для открытия сканирования
- допускается любой способ сканирования
- список номеров столбцов, составляющих строки нового списка
- список номеров столбцов, по которым нужно производить сортировку

Как и в случае создания нового индекса, можно отдельно для каждого из этих столбцов указать требование к сортировке по возрастанию или убыванию значений данного столбца. Отдельным параметром операции `BUILDLIST` является признак, в соответствии со значением которого в новом списке допускаются или не допускаются строки-дубликаты.

Операция добавления столбца к существующей таблице

Операция RSS добавления столбца к существующей таблице `CHANGE` позволяет в динамике изменять схему таблицы. Параметрами операции `CHANGE` являются идентификатор существующей таблицы и спецификация нового столбца (его тип). Красота этой операции в том, что она очень дешёвая:

- изменяется только описатель данной таблицы в служебной таблице описателей таблиц
- до выполнения первой операции `UPDATE`, затрагивающей новый столбец таблицы, реально ни в одной строке таблицы память под новый столбец выделяться не будет

По умолчанию значения нового столбца во всех строках таблицы, в которые еще не производилось явное занесение значения, считаются неопределенными. Тем

самым, ни для одного столбца, динамически добавленного к существующей таблице, не может быть запрещено хранение неопределенных значений.

Операции управления прохождением транзакций

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`)
- 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, произведённых в данной транзакции, и восстановление состояния позиций сканирования. Откат может быть произведён до начала транзакции (в этом случае о восстановлении позиций сканирования говорить бессмысленно) или до установленной ранее в транзакции точки сохранения. На уровне RSS точка сохранения устанавливается с помощью операции `SAVE`. Ответным параметром операции `SAVE` является идентификатор точки сохранения (а прямых параметров, кроме идентификатора транзакции, она не требует). При выполнении этой операции в локальной памяти транзакции запоминаются:

- состояние сканов данной транзакции, открытых к моменту выполнения `SAVE`
- координаты последней записи об изменениях в базе данных в журнале, произведённой от имени данной транзакции

Этот идентификатор в дальнейшем может быть использован как аргумент операции `RESTORE`, при выполнении которой производится восстановление базы данных по журналу с использованием записей о её 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 её сканов. Откат к началу транзакции инициируется также вызовом операции `RESTORE`, но с указанием некоторого предопределенного идентификатора точки сохранения.

При выполнении своих транзакций пользователи (приложения) System R изолированы друг от друга, т.е. не ощущают того, что система функционирует в многопользовательском режиме. Это достигается за счет наличия в RSS механизма

неявной синхронизации (про который речь будет идти далее). До конца транзакции никакие изменения базы данных, произведённые в пределах этой транзакции, не могут быть использованы в других транзакциях. Попытка использования таких данных приводит к временным синхронизационным блокировкам этих транзакций.

При выполнении операции `END TRANSACTION` происходит "фиксация"изменений, произведённых в данной транзакции (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.

Операция явной синхронизации LOCK

Последняя операция интерфейса RSS – операция явной синхронизации `LOCK`. Эта операция позволяет установить явный синхронизационную блокировку указанной таблицы. Параметром операции является идентификатор таблицы. Выполнение операции LOCK гарантирует, что никакая другая транзакция до конца данной не сможет:

- изменить данную таблицу
- вставить в него новую строку
- удалить или модифицировать существующую строку, если установлена её блокировка в режиме чтения
- прочитать любую строку этой таблицы, если установлена монопольная блокировка таблицы

Т.к. подсистема нижнего уровня RSS сама берёт на себя обязанности по синхронизации, то зачем нужна операция явной блокировки? На самом деле, логически эта операция избыточна, т.е. если бы её не было, можно вполне реализовать SQL с использованием оставшейся части операций. Предварительно, до подробного обсуждения средств управления транзакциями заметим, что операция `LOCK` введена в интерфейс RSS для возможности оптимизации выполнения запросов.

Как видно из описания интерфейса RSS, этот интерфейс является построчным. Следовательно, и информация для синхронизации носит достаточно узкий характер. В то же время на уровне SQL имеется более полная информация. Например, если обрабатывается выражение SQL `DELETE FROM EMP`, то известно, что будут удалены все строки 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.

Хэширование

Альтернативным и достаточно популярным подходом к организации индексов является использование техники хэширования. Это очень обширная тема, которая заслуживает отдельного рассмотрения. Мы ограничимся здесь лишь несколькими замечаниями.

Общей идеей методов хэширования является применение к значению ключа, по которому нужно производить поиск, некоторой функции свертки (хэш-функции), вырабатывающей значение меньшего размера (с меньшим числом бит). Значение хэш-функции затем используется для доступа к требуемой записи.

В самом классическом случае свёртка ключа используется как адрес в таблице, содержащей ключи и записи. Основным требованием к хэш-функции является равномерное распределение значения свёртки. Т.е. мы пытаемся неравномерно поступающие значения ключа равномерно распределить в пространстве свёртки. Одним из распространенных видов «хороших» хэш-функций являются функции, выдающие остаток от деления значения ключа (воспринимаемого как целое число) на некоторое простое число.

Коллизии

Однако как бы ни была устроена хэш-функция, рано или поздно возникнет коллизия - ситуация, когда применительно к нескольким поступающим ключа образуется одно и то же значение свёртки. Другими словами коллизией называется ситуация, когда мы хотим вставить в хэш-таблицу значение свёртки и запись, а место уже занято.

Один из способов борьбы с коллизиями - это использование цепочек переполнения. При наличии коллизии в этот элемент хэш-таблицы добавляется ссылка на список, в который добавляются все остальные записи с таким же значением свёртки. Понятное дело, что длинные цепочки переполнения ломают всю суть хэширования, которая заключается в константном времени доступа к любой записи. Главным ограничением классического метода хэширования (когда делим на простое число) является фиксированный размер таблицы. Если таблица заполнена слишком сильно или переполнена, то возникнет слишком много цепочек переполнения, и главное преимущество хэширования будет утрачено. Если мы захотим расширить таблицу, т.е. взять простое число побольше, то потребуется заново обрабатывать все ключи.

Существует один частный случай хэширования, когда не возникает никаких коллизий и хэширование работает идеально - так называемое «совершенное хэширование». Им можно воспользоваться, когда набор ключей, которые будут храниться в таблице заранее известен. В этом случае по набору ключей подбирается хэш-функция, которая хэш-таблицу наименьшего размера, чтобы коллизий не было вообще. Ключи и записи распределяются по таблице в соответствии с этой хэш-функцией и никогда ничего не меняется. Идеальной ситуацией для применения совершенного хэширования служит лексический анализатор какого-то языка про-

граммирования. В языке программирования есть набор ключевых слов, свёртка которых заносится в хэш-таблицу. Когда отрабатывает лексический анализатор, то при считывании идентификатора, к нему применяется хэш-функция, и путём одного обращения к хэш-таблице определяется ключевое это слово или нет.

Идея доступа к данным на основе хэширования настолько привлекательна (потенциальная возможность за одно обращение к памяти получить требуемые данные), что от нее невозможно отказаться при работе с данными во внешней памяти. Исходная идея кажется очевидной: если при управлении данными на основе хэширования в оперативной памяти хэш-функция вырабатывает индекс элемента в хэш-таблице, то при обращении к внешней памяти необходимо генерировать номер блока дискового пространства, в котором находится запрашиваемый элемент данных. Т.е. с помощью хэширования за одно обращение к внешней памяти мы считаем блок, в котором находится требуемая нам запись.

В данном случае коллизии надо понимать по другому: коллизией называется ситуация, когда в страницу данных уже не влезает нужная запись. Причём стоит обратить внимание, что в случае хэширования в оперативной памяти, ещё одно считывание адреса в ОП не приводит к большим накладным расходам, а вот в случае коллизии при хэшировании во внешней памяти, требуется считывать ещё один блок, что намного дороже. Основные методы хэширования для поиска информации во внешней памяти направлены на решение именно этой задачи.

Расширяемое хэширование

Рассмотрим два наиболее используемых метода хэширования. Первый метод называется расширяемым хэшированием (extendable hashing). Он был разработан группой из IBM во главе с Рональдом Фейджином:

- Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, H. Raymond Strong. Extendible Hashing-A Fast Access Method for Dynamic Files. ACM Transactions on Database Systems, Vol. 4, No. 3, September 1979, pp. 315-344 (ссылка на статью)

В основе подхода расширяемого хэширования (Extendible Hashing) лежит принцип использования деревьев цифрового поиска в оперативной памяти. В оперативной памяти поддерживается справочник, организованный на основе бинарного дерева цифрового поиска, ключами которого являются значения хэш-функции, а в листовых вершинах хранятся номера страниц записей во внешней памяти. В этом случае любой поиск в дереве цифрового поиска является «успешным», т.е. ведет к некоторой странице внешней памяти. Входит ли в эту страницу искомая запись, обнаруживается уже после прочтения страницы в оперативную память.

Как отмечалось ранее, коллизию следует понимать как ситуацию переполнения страницы внешней памяти, когда значение хэш-функции указывает на этот блок, но места для включения записи в нем уже нет. В этом случае страница расщепляется на две, и дерево цифрового поиска переформируется соответствующим образом. При этом может потребоваться расширение самого справочника. Расширяемое

хэширование хорошо работает в условиях динамически изменяемого набора записей в хранимом файле, но требует наличия в оперативной памяти справочного дерева.

В области организации индексов на основе хэширования принято использовать термин «бакет» (bucket; корзина). Бакет обозначают страницу внешней памяти. Неважно каким образом устроена хэш-функция, она всегда возвращает строку бит.

Метод расширяемого хеширования заключается в том, что хеш-таблица представлена как каталог, а каждая ячейка будет указывать на бакет, который имеет определенную вместимость. Сама хеш-таблица будет иметь глобальную глубину G, а каждая из емкостей имеет локальную глубину l_i. Глобальная глубина G показывает сколько последних бит будут использоваться для того чтобы определить в какую емкость следует заносить значения. А из разницы локальной глубины и глобальной глубины можно понять сколько ячеек каталога ссылаются на емкость.

Рассмотрим сам алгоритм, когда поступил ключ k с некоторым значением свёртки $h(k)$:

1) По первым G битам свёртки $h(k)$ решаем в какой бакет отправить ключ.
2) Если в бакете есть свободное место, то просто помещаем туда ключ, если же нужный бакет переполнен, то смотрим на локальную глубину бакета:
   - Если локальная глубина меньше, чем глобальная глубина (на бакет несколько ссылок), то создаём новый бакет и заново прогоняем перераспределяем все ключи в текущем бакете с учётом новой длины бит. Не забыть увеличить глубину обоих бакетов на 1. Возвращаемся к шагу 1.
   - Если локальная глубина равна глобальной, то мы увеличиваем глобальную глубину на 1, удваивая при этом количество ячеек, количество указателей на бакеты, а также увеличиваем количество последних бит по которым мы распределяем значения.

Рассмотрим действие расширяемого хэширования на примере. Предположим, что есть есть три ключа k_1, k_2, k_3 со значениями хэш-функции:

- $h(k_1) = 100100$
- $h(k_2) = 010110$
- $h(k_3) = 110110$

Пусть в нашем примере в каждый бакет помещается одна запись. Сначала вставляются ключи k_1 и k_2. Мы видим, что значения свёртки $h(k_1)$ и $h(k_2)$ различаются старшим битом. Хэш-таблица после их вставки показана на рис. 85. На рисунке $G = 1$ - глобальная глубина, $l1 = 1$ и $l2 = 1$ - локальные глубины бакетов.

Если теперь вставить ключ k_3 со значением свёртки $h(k_3) = 110110$, то нам надо помещать его в один бакет с k_1, т.е. происходит переполнение. Локальная глубина $l2$ бакета равна глобальной глубине G, значит, согласно алгоритму:

- Надо увеличить хэш-таблицу вдвое, т.е. используется теперь $G = 2$ бита для индексации.

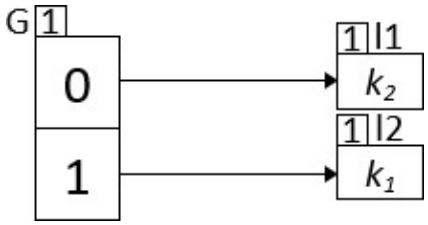


Рис. 85: Хэш-таблица после вставки k_1 и k_2

- Ключ k_1 надо заново прогнать через функцию хэширования, но теперь смотреть будем, не на один бит, а на два. $h(k_1) = 100100$ и $h(k_3) = 110110$ различаются в двух битах, значит обоим найдётся место и дальнейшее расширение не требуется.

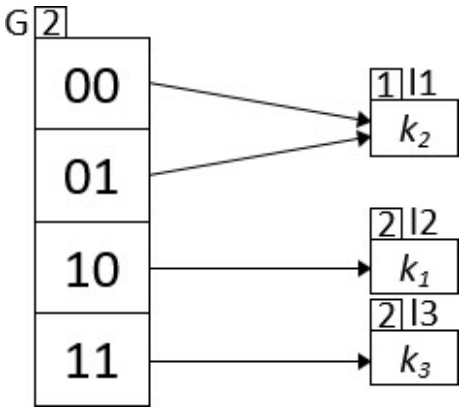


Рис. 86: Хэш-таблица после вставки k_3

На рис. 86 мы видим, что бакет2 (у которого подписана глубина $l2$) «расщепился» на два. В новый бакет как раз и попал ключ k_3. Однако заметим, что могло бы быть и наоборот: k_1 переместился бы в новый бакет, а k_3 пошёл бы в текущий. Также могла бы быть ситуация, что $h(k_1)$ совпадают не в одном, а в первых двух битах. В этом случае надо было бы ещё раз делать удвоение таблицы, и глобальная глубина стала бы равной трём, и все записи таблицы, начинающиеся с 0 также бы указывали на первый бакет, а остальные записи распределились бы между вторым и третим бакетом. Вернёмся к примеру.

Пусть нам надо вставить ключ k_4 со значением свёртки $h(k_4) = 010000$. По таблице мы приходим в первый бакет и мы видим, что локальная глубина $l1 = 1$, что меньше глобальной глубины $G = 2$. Согласно описанному выше алгоритму нужно создать новый бакет и перехэшировать ключ k_2, увеличить глубину обоих бакетов на 1 и заново пробовать вставить ключ k_4. На рис. 87 показана хэш-таблица после создания нового бакета и перехэширования k_2 (и увеличения глубины с 1 до 2).

Теперь пробуем вставить k_4 ещё раз. Опять же попадаем в первый бакет (где подписана глубина $l1$), который уже занят, причём локальная глубина равна глобальной. В этом случае, согласно алгоритму, мы удваиваем хэш-таблицу, новые страницы ссылаем на уже существующие и перехэшируем k_2, после чего снова пробовать

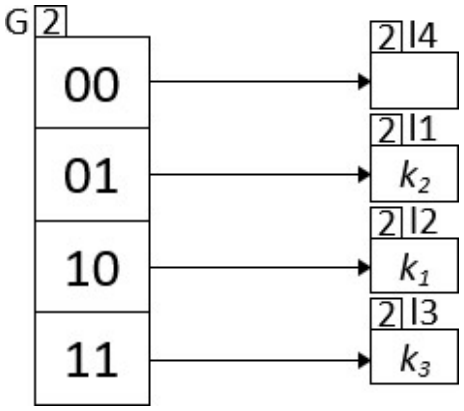


Рис. 87: Промежуточный результат вставки k_4

вставить k_4. После перехэширования k_2 переместится в новосозданный пятый бакет, а k_4 попадёт в первый. На рис. 88 показан результат вставки k_4.

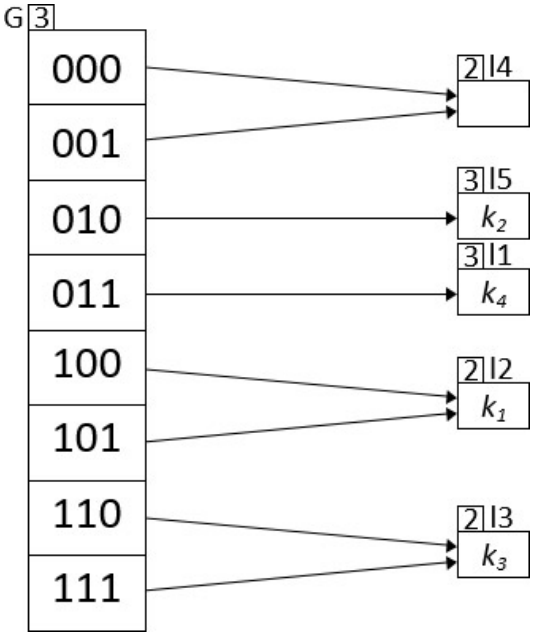


Рис. 88: Удвоение таблицы, перехэширование k_2 и вставка k_4

Линейное хэширование

Легендарная личность в области хэширования Витольд Литвин на год позже предложил метод линейного хэширования.

- Witold Litwin. Linear Hashing: A New Tool for File and Table Addressing. Proceedings pf the Sixth International Conference on Very Large Data Bases, October 1-3, 1980, pp. 212-223 (ссылка)

Идея линейного хэширования (Linear Hashing) состоит в том, чтобы можно было обойтись без поддержания справочника в оперативной памяти. Основой метода является то, что для адресации страницы внешней памяти всегда используются младшие биты значения хэш-функции. Если возникает потребность в расщеплении, то записи перераспределяются по страницам так, чтобы адресация осталась правильной.

В общем случае в схеме линейного хэширования имеются m исходных бакетов (от 0 до m-1). Хэш-функция $h_0(k) = f(k) \bmod m$, используется для отображения любого ключа в один из m бакетов. Мы не будем затрагивать конкретный вид $f(k)$, для нас самое главное, что по модулю m. Имеется также некоторый указатель p, ссылающийся на бакет, который нужно расщепить следующим, если он переполнен.

Рассмотрим алгоритм линейного хэширования на примере. Пусть количество бакетов $m = 4$, p - указатель на начало, хэш-функции $h_0(k) = k \bmod 4$ и $h_1(k) = k \bmod 8$. Далее станет понятно зачем нам две хэш-функции. Предположим, что между бакетами распределены ключи, как показано на рис. 89.

	бакет №	основные страницы			страницы переполнения
p ⟶	0	4	8	12	16
	1	1	5		
	2	6	10	22	
	3	15	3	7	19

Рис. 89: Пример заполненой таблицы для линейного хэширования

Пусть мы вставляем 11 в таблицу на рис. 89. Оно попадает в третий бакет, в котором места нет, поэтому:

- добавляем новый четвёртый бакет
- перехэшируем все элементы бакета p (нулевого) функцией $h_1(k) = k \bmod 8$
- перемещаем указатель p на следующий (первый)
- к третьему бакету добавляем страницу переполнения, куда и заносится 11

На рис. 90 показан результат всех этих операций. Четвёртый бакет находится под горизонтальной чертой, которая ограничивает бакеты первого раунда. По алгоритму, как только указатель p дойдёт до границы раунда (захочет перейти на 4), то он перейдёт на нулевой бакет, и будет установлена новая граница (она будет после 7го бакета). В первом раунде, когда происходит переполнение (оно может произойти в любом бакете), создаётся новый бакет, и элементы p-бакета перехэшируются функцией $h_1(k)$.

Пусть мы теперь вставляем ключи 18 и 26. Они пойдут во второй бакет, и возникнет переполнение, тогда:

- создастся новый пятый бакет

Рис. 90: Нулевой бакет расщепился, указатель сдвинулся на первый

- содержимое первого бакета перераспределится с использованием хэш-функции $h_1(k) = k \bmod 8$ между бакетами 1 и 5
- 18 попадёт в основную страницу бакета 2, 26 - в страницу переполнения

Результат показан на рис. 91. Как видно, в результате перехэширования ключ 5 из первого бакета перешёл в пятый. И при любом переполнении будет расщеплён второй бакет, в результате чего часть ключей перейдут в шестой бакет.

Рис. 91: Первый бакет расщепился, указатель сдвинулся на второй

Аналогично всё будет происходить вплоть до того, как произойдёт расщепление третьего бакета и начнётся новый раунд снова с $p = 0$, а граница второго раунда будет установлена после седьмого бакета. Только хэш-функции теперь будут не $h_0(k) = k \bmod 4$ и $h_1(k) = k \bmod 8$, а $h_1(k)$ и $h_2(k) = k \bmod 16$. Некоторое гипотетическое наполнение хэш-таблицы на момент начала следующего раунда показано на рис. 92.

Когда начнётся третий раунд, уже будут созданы бакеты от 0 до 15, указатель снова будет стоять на нулевом бакете, а функции будут $h_2(k) = k \bmod 16$ и $h_3(k) = k \bmod 32$. Аналогично всё будет происходить и далее.

Рис. 92: Начало второго раунда

Журнальная информация

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

Служебная информация

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

- Внутренние каталоги, описывающие физические свойства объектов базы данных, например, число столбцов таблицы, их размер и, возможно, типы данных; описатели индексов, определенных для данной таблицы и т.д.
- Описатели свободной и занятой памяти в страницах данных. Такая информация требуется для нахождения свободного места при занесении строки. Отдельно приходится решать задачу поиска свободного места в случаях некластеризован-

ных и кластеризованных таблиц. Для кластеризованных таблиц приходится дополнительно использовать кластеризованный индекс. Как уже отмечалось, нетривиальной является проблема освобождения страницы данных, в которой хранятся строки таблиц, в условиях мультидоступа.

- Связывание страниц одной таблицы. Если в одном файле внешней памяти могут располагаться страницы нескольких таблиц (обычно к этому стремятся), то нужно каким-то образом связать страницы одной таблицы. Тривиальный способ использования прямых ссылок между страницами часто приводит к затруднениями при синхронизации транзакций. Например, особенно трудно освобождать и заводить новые страницы таблицы. Поэтому стараются использовать косвенное связывание страниц с использованием служебных индексов. В частности, известен общий механизм для описания свободной памяти и связывания страниц на основе B-деревьев.

Механизм транзакций

Поддержка механизма транзакций – показатель уровня развитости СУБД. Корректное поддержание транзакций является основой обеспечения целостности баз данных, поэтому транзакции вполне уместны и в однопользовательских персональных СУБД. Кроме того, механизм транзакций составляет базис изолированности пользователей в многопользовательских системах. Эти два аспекта взаимосвязаны, что и будет показано далее.

ACID требования к транзакциям

В современных СУБД поддерживается понятие транзакции, характеризуемое аббревиатурой ACID:

- Atomicity (Атомарность)
- Consistency (Согласованность)
- Isolation (Изолированность)
- Durability (Долговечность)

В соответствии с этим понятием под транзакцией подразумевается последовательность операций над базой данных, обладающая следующими свойствами:

- Атомарность (Atomicity) 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. 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) означает, что транзакция может быть успешно завершена с фиксацией результатов своих операций только в том случае, когда 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. В классическом контексте классических баз данных термины согласованность и целостность эквивалентны. Единственным критерием согласованности данных является их удовлетворение ограничениям целостности. Другими словами, база данных находится в согласованном состоянии тогда и только тогда, когда она находится в целостном состоянии.

Почему так акцентируется внимание на классическом контексте и классических БД? Дело в том, что согласованность и целостность в распределённых СУБД типа NoSQL (Not only SQL) не эквивалентны. Основная особенность NoSQL заключается в том, что они работают в совершенно распределённой среде. Не с проста образование NoSQL связано с такими IT-гигантами, как Google, IBM, Amazon и т.д. Это компании, которым нужно, чтобы их клиенты могли работать с базами данных в любой точке Земли так, чтобы доступ был достаточно эффективным. Из-за этого база данных должна быть глобально распределена в мире, т.е. храниться не в одном ЦОД (центре обработки данных), а в нескольких.

С другой стороны, чтобы обеспечивать достаточно быстрый доступ локально в любой точке Земли, нужно поддерживать реплики - копии данных в разных ЦОД. Если не будет реплик, и нужные в Японии данные будут запрашиваться из базы данных в Англии, то это будет очень неэффективно. Понятное дело, что если в глобальной системе поддерживаются реплики, то очень дорого их поддерживать в одинаковом состоянии, идентичными. Но для того, чтобы сказать, что люди работают с базами данных надёжно и строго, хотя бы так, как обеспечивают транзакции ACID, необходимо, чтобы реплики были согласованы.

И тут возникает CAP-теорема о невозможность одновременного в распределённой системе обеспечения согласованности, доступности и возможность работать с данными, если сеть теряет связность. Т.е. нельзя, если сеть распалась на два несвязных фрагмента, обеспечить согласованность данных в этих двух фрагментах (нет физического доступа).

- Если хотим сохранить доступность данных, то мы должны пожертвовать согласованностью, т.е. P и A противоречат C.
- Если хотим обеспечить доступ к согласованным данным, понятно, что мы не должны допустить несвязность сети и т.д.

Здесь часто путают понятие согласованности. В транзакциях категории ACID под согласованностью мы понимаем, что база данных удовлетворяет некоторым ограничениям, которые установлены извне. Например, ограничение внешнего ключа

или то ограничение, что размер отдела, который записан в столбце численности служащих совпадает с реальным числом служащих, если их посчитать. Т.е. СУБД не знает об этих ограничениях, если ей не сказать явно; она не может ничего придумать по этому поводу. С другой стороны, в случае NoSQL, СУБД знает о том, что реплики должны быть одинаковыми, т.е. это физическая целостность реплик. Поэтому нельзя считать, что в распределённой системы нельзя обеспечить логическую целостность, нельзя обеспечить именно физическую согласованность.

Важно отметить, что именно из-за невозможности поддержания физической эквивалентности реплик, в распределённых системах нельзя обеспечить ACID транзакции, поэтому они опираются на другую модель - BASE транзакции. Требования BASE предполагают:

- базовую доступность (basic availability) - каждый запрос гарантированно завершается (успешно или безуспешно)
- гибкое состояние (soft state) - состояние системы может изменяться со временем, даже без ввода новых данных, для достижения согласованности данных
- согласованность в конечном счёте (eventual consistency) - данные могут быть некоторое время несогласованы, но приходят к согласованию через некоторое время

Лекция 23

Вернёмся к требованиям ACID к транзакция. Уже были рассмотрены свойства атомарности (Atomicity) и согласованности (Consistency), теперь рассмотрим изоляцию и долговечность:

- Изоляция (Isolation). Требуется, чтобы две одновременно (concurrently; параллельно или квазипараллельно) выполняемые транзакции никаким образом не действовали одна на другую. Результаты выполнения операций транзакции T_1 не должны быть видны никакой другой транзакции T_2 до тех пор, пока транзакция T_1 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.

Свойства ACID особенно важны для многопользовательских систем. Далее рассмотрим подробнее как поддерживаются свойства атомарности, согласованности, изолированности и долговечности транзакций, и какие для этого существуют алгоритмы.

Атомарность транзакций

В этом смысле под транзакцией понимается неделимая с точки зрения воздействия на базу данных 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 ACID транзакции является «Все или ничего»:

- при успешном завершении транзакции оператором `COMMIT` (это высокоуровневый аналог операции `END TRANSACTION` интерфейса RSS, см. стр. 198)
  - результаты гарантированно фиксируются во внешней памяти
  - смысл термина commit состоит в запросе «фиксации» результатов транзакции
- при завершении транзакции оператором `ROLLBACK` (это высокоуровневый аналог операции `RESTORE` интерфейса RSS)
  - результаты гарантированно отсутствуют во внешней памяти
  - смысл термина rollback состоит в запросе ликвидации результатов транзакции

Каким образом в СУБД поддерживаются индивидуальные откаты транзакций будет рассказано несколько позже.

Транзакции и целостность баз данных

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

Например, в базе данных `СЛУЖАЩИЕ-ОТДЕЛЫ` естественным ограничением целостности является совпадение значения атрибута `ОТД_РАЗМЕР` в строке таблицы `ОТДЕЛЫ`, описывающей данный отдел (например, отдел 625), с числом строк таблицы `СЛУЖАЩИЕ`, таких, что значение поля `СЛУ_ОТД_НОМЕР` равно 625. Как в этом случае принять на работу в отдел 625 нового сотрудника? Независимо от того, какая операция будет выполнена первой, вставка нового кортежа в таблице `СОТРУДНИКИ` или модификация существующего кортежа в отношении `ОТДЕЛЫ`, после выполнения любой из этих двух операций база данных окажется в не целостном состоянии. Обратим внимание, что с точки зрения структур данных всё в порядке. Здесь именно логическая несогласованность, т.е. база данных не удовлетворяет этому внешнему условию.

Для поддержки подобных ограничений целостности допускается их нарушение внутри транзакции с тем условием, чтобы к моменту завершения транзакции условия целостности были соблюдены. В системах, которые поддерживают ACID транзакции, каждая транзакция начинается при логически целостном состоянии базы данных и должна оставить это состояние целостным после своего завершения. Логика здесь простая: начальная проверка целостности БД проверяется при первоначальной загрузке БД, что гарантирует её целостность на момент выполнения некоторой транзакции T. Теперь эта транзакция фиксирует изменения операцией `COMMIT`:

- Если вдруг транзакция нарушает логическую целостность базы данных, то вместо фиксации происходит откат изменений (`ROLLBACK`) и база данных находится в целостном состоянии
- Если транзакция не нарушает целостность базы данных, то происходит фиксация, и база данных всё равно в целостном состоянии

Различаются два вида ограничений целостности (это точно специфицировано в стандарте SQL): немедленно проверяемые (immediate constraints) и откладываемые (deferred constraints).

- К немедленно проверяемым ограничениям целостности относятся такие ограничения, проверку которых бессмысленно или даже невозможно откладывать.

  Например, «возраст сотрудника не может превышать 150 лет». Понятно, что такое ограничение бессмысленно откладывать на потом и нужно проверять сразу же. Если пример с возрастом ещё можно отложить, то вот ограничение «зарплата сотрудника не может быть увеличена за одну операцию более чем

на 100000 рублей» вообще никак не проверить в конце транзакции, это надо делать сразу.

Немедленно проверяемые ограничения целостности соответствуют уровню отдельных операторов языкового уровня (уровня SQL) СУБД. При их нарушениях не производится откат транзакции, а лишь отвергается соответствующий оператор. Что делать с такой операцией решает само приложение.

- 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 её начала или до установленной ранее точки сохранения или постараться устранить причины несогласованного 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.

Описанный механизм поддержки целостности баз данных обеспечивает требуемое свойство транзакций: никакая транзакция не может быть зафиксирована, если её 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 ограничениям целостности (с точки зрения данной транзакции), то долговременно выполняемая транзакция вполне вероятно может быть «откачена» системой при выполнении завершающего оператора `COMMIT`. 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 её завершении и тем дороже становится оператор `COMMIT`.

Простое и элегантное решение этой проблемы предлагают Дейт и Дарвен (в третьем манифесте). Авторы предлагают отказаться от откладываемых ограничений целостности базы данных, а вместо этого ввести составные операторы изменения базы данных (внутри транзакции): 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 (не транзакция целиком, а только составной блок), и вырабатывается соответствующий код ошибки. Транзакция в этом случае не откатывается. При использовании такого подхода при выполнении оператора `COMMIT` не требуется проверять ограничения целостности, и каждая зафиксированная транзакция будет оставлять базу данных в целостном состоянии. 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. К сожалению, этот подход на практике пока не применяется.

Изолированность транзакций

В многопользовательских системах с одной базой данных одновременно может работать несколько пользователей или прикладных программ. Предельной задачей системы является обеспечение изолированности пользователей (приложений), 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.

Рассмотрим несколько сценариев, в которых нарушается свойство изолированности транзакций.

Потерянные изменения

Предположим, что в какой-то момент времени $t1$ транзакции T_1 и T_2 уже выполняются, и транзакция T_1 изменяет объект базы данных o, т.е. выполняет операцию $W(o)$ (`UPDATE` или `DELETE`). До завершения транзакции T_1 в момент времени $t2 > t1$ транзакция T_2 также изменяет объект o. Предположим, что в момент времени $t3 > t2$ до завершения транзакции T_1, транзакция T_2 завершается оператором `ROLLBACK` (например, по причине нарушения ограничений целостности). На рис. 93 показан описанный сценарий.

Тогда получается, что транзакция T_2 откатит не только свои изменения, но и те, что сделала T_1! Если в момент времени $t4 > t3$ транзакция T_1 снова считывает объект o, т.е. выполняет операцию чтения $R(o)$, то она не видит своих изменений, которые были выполнены в момент времени $t1$. В частности, из-за этого может не удастся фиксация этой транзакции, что, возможно, повлечет потерю изменений у еще одной транзакции и т.д.

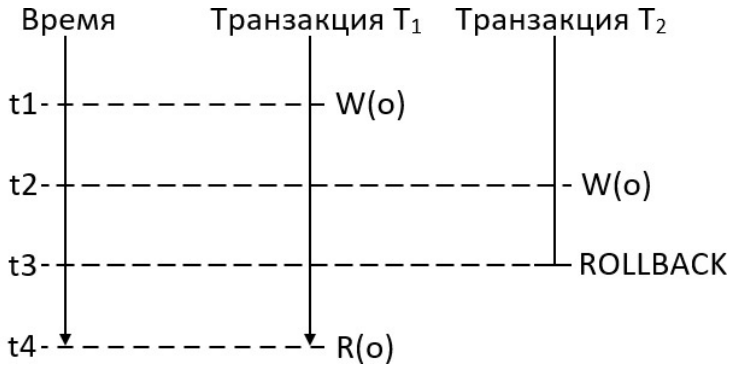


Рис. 93: Транзакция T_1 не видит свои изменения объекта o

Такая ситуация называется ситуацией потерянных изменений (lost updates). 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 изменённый транзакцией T_1 объект o. В частности, достаточно заблокировать доступ по изменению объекта o до завершения транзакции T_1 (и чаще всего так и поступают). Отсутствие потерянных изменений является минимальным требованием к СУБД при обеспечении изолированности одновременно выполняемых транзакций.

Отсутствие чтения «грязных» данных

Пусть также имеются две транзакции T_1 и T_2. В момент времени $t1$ транзакция T_1 изменяет объект базы данных o (выполняет операцию $W(o)$). В момент времени $t2 > t1$ транзакция T_2 читает объект o (выполняет операцию $R(o)$). Транзакция T_1 еще не завершена, поэтому транзакция T_2 видит несогласованные «грязные» данные, которых в базе данных не было. Пусть в момент времени $t3 > t2$ транзакция T_1 завершается откатом (например, по причине нарушения ограничений целостности), тогда эти грязные данные ещё и не зафиксируются. На рис. 94 показан описанный сценарий.

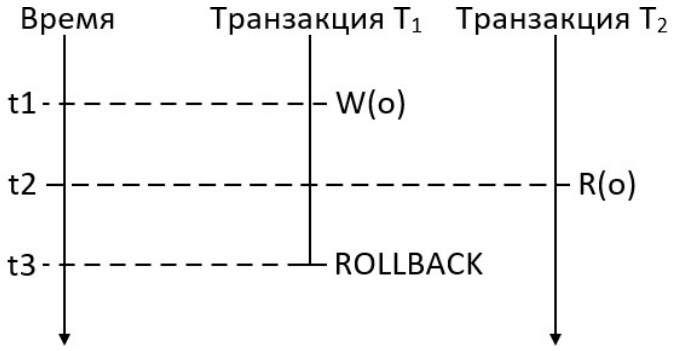


Рис. 94: Транзакция T_2 читает «грязные» данные

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

Пусть в момент времени $t1$ транзакция T_1 читает объект базы данных o (выполняет операцию $R(o)$). До завершения транзакции T_1 в момент времени $t2 > t1$ транзакция T_2 изменяет объект o (выполняет операцию $W(o)$) и успешно завершается оператором `COMMIT`. В момент времени $t3 > t2$ транзакция T_1 повторно читает объект o и видит его изменённое состояние. На рис. 95 показан описанный сценарий.

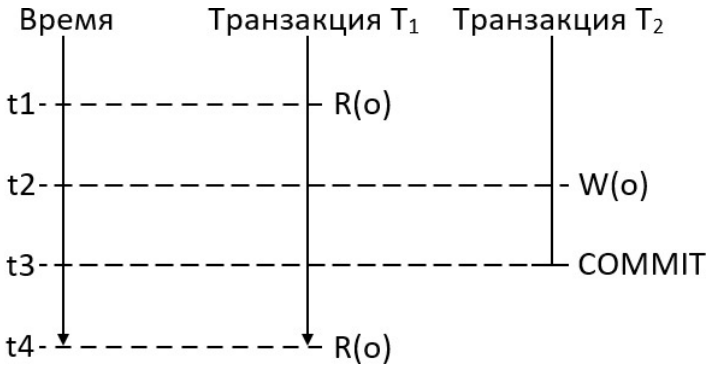


Рис. 95: Транзакция T_1 при повторном чтении получает иной результат

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 отсутствие неповторяющихся чтений еще не гарантирует реальной изолированности пользователей.

Проблема фантомов

К более тонким проблемам изолированности транзакций относится так называемая проблема строк-«фантомов», приводящая к ситуациям, которые также противоречат изолированности пользователей. Пусть в момент времени $t1$ транзакция T_1 выполняет оператор выборки строк таблицы `Tab` с условием выборки `S`, т.е. выбирается часть строк таблицы `Tab`, удовлетворяющих условию `S`.

До завершения транзакции T_1 в момент времени $t2 > t1$ транзакция T_2 вставляет в таблицу `Tab` новую строку `r`, удовлетворяющую условию `S`, и успешно завершается.

В момент времени $t3 > t2$ транзакция T_1 повторно выполняет тот же оператор выборки, и в результате появляется строка, которая отсутствовала при первом выполнении оператора. На рис. 96 показан описанный сценарий.

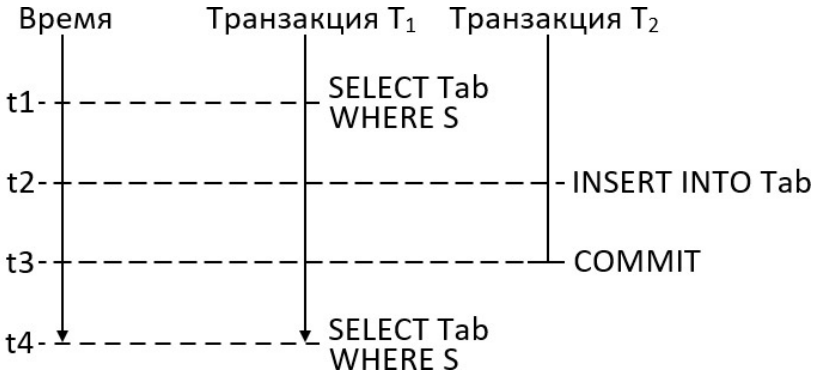


Рис. 96: Транзакция T_1 при повторном `SELECT` получает иной результат

Конечно, такая ситуация противоречит идее изолированности транзакций. Чтобы избежать появления строк-фантомов, требуется более высокий «логический» уровень изоляции транзакций. Идеи требуемого механизма, которым оказались предикатные синхронизационные блокировки, появились еще во время выполнения проекта System R, но в большинстве систем в полном объёме не реализованы.

Сериализация транзакций

Для выполнения требования изолированности транзакций, в СУБД должны использоваться какие-либо методы регулирования совместного выполнения транзакций. Пусть в системе одновременно выполняется некоторое множество транзакций $S = \{T_1, T_2, \ldots, T_n\}$. План (способ) выполнения набора транзакций S, в котором, вообще говоря, чередуются или реально параллельно выполняются операции разных транзакций $T_1, T_2, \ldots, T_n$, называется *сериальным*, если результат совместного выполнения транзакций эквивалентен результату некоторого последовательного выполнения этих же транзакций $(T_{i1}, T_{i2}, \ldots, T_{in})$. Обратите внимание на круглые скобочки, которые обозначают упорядоченность.

Сериализация транзакций – это механизм одновременного выполнения транзакций $T_1, T_2, \ldots, T_n$ 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. Потому что тривиальным решением является любое последовательное выполнение заданного набора транзакций. Но существуют и такие ситуации, в которых можно выполнять операторы разных транзакций в любом порядке с сохранением свойства сериальности

(например, 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 (конфликт Write/Write) – транзакция T_2 пытается изменять объект, изменённый не закончившейся транзакцией T_1. Может привести к ситуации потерянных изменений.
- R/W (конфликт Read/Write)– транзакция T_2 пытается изменять объект, прочитанный не закончившейся транзакцией T_1. Может привести к возникновению ситуации неповторяющихся чтений.
- W/R (конфликт Write/Read) – транзакция T_2 пытается читать объект, изменённый не закончившейся транзакцией T_1. Может привести к возникновению ситуации «грязного» чтения.

Заметим, что нет конфликта Read/Read. Только читающие транзакции не конфликтуют. Практические методы сериализации транзакций основываются на учете этих конфликтов.

Методы сериализации транзакций

Существуют два базовых подхода к сериализации транзакций:

- основанный на синхронизационных захватах объектов базы данных
- основанный на использовании временных меток

Филипп Бернштейн, один из самых лучших специалистов в мире на текущий момент в области управления транзакциями, высказал утверждение, которое никто ещё ни доказал и не опроверг, что все известные методы сериализации транзакцией являются комбинацией этих двух базовых подходов.

Суть обоих подходов состоит в обнаружении конфликтов транзакций и их устранении. Далее рассмотрим эти подходы сравнительно подробно. Кроме того, кратко обсудим возможности сериализации транзакций на основе поддержки версий объектов базы данных. Такой подход направлен на ускорение выполнения «только читающих» транзакций, т.е. транзакций, в которых не выполняются операции изменения базы данных.

Чем пессимисты отличаются от оптимистов? Пессимист считает, что если что-то может произойти (как правило нехорошее), то оно произойдёт. Оптимист считает, что если что-то может произойти, то оно может и не произойти (Чего раньше времени париться, пока всё хорошо?).

Для каждого из подходов имеются две разновидности: пессимистическая и оптимистическая.

- При применении пессимистических методов, ориентированных на ситуации, когда конфликты возникают часто, конфликты распознаются и разрешаются

немедленно при их возникновении. Зачастую для разрешения конфликта надо принудительно откатить какую-то транзакцию.

- Оптимистические методы хорошо работают, когда конфликты редки. В этом случае любая транзакция, изменяющая базу данных выполняется без синхронизации вообще, но все её результаты в базу данных не пишутся, а сохраняются в её локальной памяти (до операции `COMMIT`). Когда отрабатывает операция `COMMIT` система проигрывает транзакцию заново и смотрит не было ли конфликтов при выполнении рабочей фазы транзакции. Если конфликтов не было, то все её изменения реально записываются в БД. Если конфликты обнаруживаются, то транзакция откатывается.

Практически для любого пессимистического метода есть её оптимистический вариант. Как правило, оптимистические методы технически более сложные, там больше вещей, которые не касающихся сути, а чисто технических. Поэтому далее мы ограничимся рассмотрением более распространенных пессимистических разновидностей методов сериализации транзакций. Пессимистические методы сравнительно просто трансформируются в свои оптимистические варианты.

Лектор рекомендует к ознакомлению статьи по сериализации транзакций:

- Перевод: С.Д. Кузнецов «Ричард Хардинг, Дана Ван Акен, Эндрю Павло, Майкл Стоунбрейкер. Оценка протоколов управления распределенными транзакциями» (ссылка)
- Перевод: С.Д. Кузнецов «Сияньяо Ю, Джордж Безерра, Эндрю Павло, Сринивас Девадас, Майкл Стоунбрейкер. Вглядываясь в бездну: оценка схем управления параллелизмом на процессоре с тысячью ядер» (ссылка)

Синхронизационные блокировки

Это наиболее распространенный в централизованных СУБД (включая системы, основанные на архитектуре «клиент-сервер») 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 в базовом варианте 2PL являются следующие:

- S (Shared) - совместный режим, означающий совместную (по чтению) блокировку объекта. Такая блокировка требуется для операции чтения объекта.
- X (eXclusive) - монопольный режим, означающий монопольную (по записи) блокировку объекта. Требуется 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. 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). Блокировка объекта одной транзакцией по чтению не совместима с блокировкой другой транзакцией того же объекта по записи:

- никакая транзакция не может изменять объект, читаемый некоторой транзакцией (кроме самой этой транзакции)
- никакой транзакции нельзя читать объект, изменяемый некоторой транзакцией (кроме самой этой транзакции)

Заметим, что эти правила совместимости в точности соответствуют разновидностям конфликтов транзакций.

Рис. 97: Правила совместимости режимов блокировок S и X

На рис. 97 приведены правила совместимости захватов одного объекта разными транзакциями. В первом столбце приведены возможные состояния объекта с точки зрения синхронизационных захватов. Минус в первом столбце "-" означает, что объект не блокирован. 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.

Красный крестик, т.е. 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 конфликта R/R не существует.

Для обеспечения сериализации транзакций синхронизационные блокировки объектов, произведённые по инициативе транзакции, можно снимать только при её 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.

Возникает вопрос: «Что следует считать объектом для синхронизационного захвата?» Самый наигрубейший вариант - блокировать всю базу данных. 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 (строка) – 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.

Но при использовании для блокировок крупных объектов возрастает вероятность конфликтов транзакций и, тем самым, уменьшается допускаемая степень чередования операций транзакций 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.

Во многих SQL-ориентированных СУБД разработчики 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, т.к. это блокировка физического объекта. Использование страничных блокировок в двухфазном протоколе иногда вызывает очень неприятные синхронизационные проблемы, усложняющие организа-

цию СУБД. Эти проблемы связаны с тем, что страницы приходится блокировать на двух разных уровнях:

- на уровне управления буферами страниц в оперативной памяти
- на уровне выполнения логических операций

В результате могут возникнуть очень неприятные синхронизационные тупики между логическим и физическим уровнями. Несколько позднее мы более подробно опишем данную проблему.

В большинстве современных систем используются покортежные синхронизационные блокировки. 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.

Гранулированные синхронизационные блокировки

Подобные рассуждения привели к разработке механизма гранулированных синхронизационных блокировок. 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. Что же собой представляют эти режимы блокировок?

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`.

Текущее состояние объекта \ Хотим заблокировать	X	S	IX	IS	SIX
-	✓	✓	✓	✓	✓
X	✗	✗	✗	✗	✗
S	✗	✓	✗	✓	✗
IX	✗	✗	✓	✓	✗
IS	✗	✓	✓	✓	✓
SIX	✗	✗	✗	✓	✗

Рис. 98: Правила совместимости режимов блокировок S, X, IS, IX и SIX

На рис. 98 изображена таблица совместимости блокировок S, X, IS, IX и SIX. Поясним правила совместимости. Для атомарных объектов разумны только блокировки в режимах S и X, для которых правила совместимости остаются такими же, как прежде (на рисунке они соответствуют верхнему левому прямоугольнику).

Пусть 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.

Блокировака 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
- 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
- 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 синхронизационные 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 (декларативная операция), 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 совершенно 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 передавалась дополнительная информация - диапазон сканирования. Диапазон ограничивает множество кортежей, среди которых не должны возникать фантомы. Опираясь на наличие этой информации, предлагалось ввести в систему блокировок System R элементы предикатных блокировок. 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 таблицей совместимости или если указанные диапазоны значений ключей не пересекаются.

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 вставки или 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. Однако описанное решение проблемы фантомов далеко от идеального.

Лекция 24

Во-первых, это работает только когда сканирование делается через индекс. На самом деле, как показывает практика использования SQL сервера, больше всего сканирований происходит без использования индексов. Если индекса нет, то единственный способ гарантировать отсутствие фантомов - блокировать таблицу целиком при начале работы.

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 op значение`, где `op` $\in \{=, >, <\}$. В типичных СУБД в интерфейсе подсистемы управления памятью 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 случае сканирования через индекс оно конъюнктивно соединяется с условием, задаваемым диапазоном сканирования.

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). И при операциях вставки или удаления имеются неявные условия - это конъюнктивные логические выражения, состоящие из простых предикатов вида `имя_поля = значение` для всех полей таблицы, а также операциями обновления кортежей. И при выполнении операции `UPDATE` имеются аналогичные неявные условия для всех изменяемых 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 $a_1, a_2, ..., a_n$, а $m_1, m_2, ..., m_n$ – множества допустимых значений полей $a_1, a_2, ..., a_n$ соответственно (естественно, все m_i 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 ($k \leq n$).

Достаточно очевидно следующее утверждение: Пусть имеются два простых условия scond1 и scond2. Пусть транзакция T_1 запрашивает блокировку scond1, а транзакция T_2 – scond2 в режимах, которые были бы несовместимы, если бы

scond1 и scond2 являлись не условиями, а объектами базы данных (т.е. комбинации S-X, X-S, X-X). Эти блокировки совместимы в том и только в том случае, когда прямоугольники, соответствующие scond1 и scond2, не пересекаются.

Действительно, 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.

В каких бы режимах не требовала транзакция $T1$ блокировки условия $(0 < a < 5) \wedge (b = 5)$, а транзакция T_2 - блокировки условия $(0 < a < 6) \wedge (0 < b < 4)$, эти блокировки всегда будут совместимы. На рис. 99 отрезок сверху изображает условие $(0 < a < 5) \wedge (b = 5)$, а синий прямоугольник - условие $(0 < a < 6) \wedge (0 < b < 4)$. Как видно, эти множества не пересекаются.

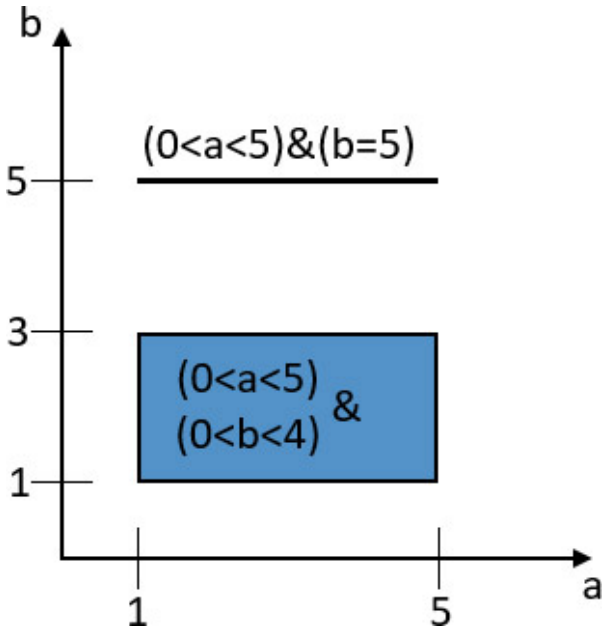


Рис. 99: Множества кортежей с указанными условиями не конфликтуют

При поддержке такой системы блокировок простых условий можно обойтись без гранулированных блокировок. В частности, чтобы гарантированно заблокировать таблицу целиком, достаточно заблокировать условие:

$$\texttt{(min(}m_1\texttt{) < имя\_поля}_1\texttt{ < max(}m_1\texttt{))} \wedge$$
$$\texttt{(min(}m_2\texttt{) < имя\_поля}_2\texttt{ < max(}m_2\texttt{))} \wedge$$
$$\ldots$$
$$\texttt{(min(}m_n\texttt{) < имя\_поля}_n\texttt{ < max(}m_n\texttt{))}$$

Чтобы заблокировать базу данных, достаточно заблокировать условие, являющееся конъюнкцией условий блокировки всех таблиц этой базы данных. 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.

Синхронизационные тупики, их распознавание и разрушение

Независимо от вида протокола 2PL, всем методам блокировочных протоколов свойствена возможность возникновения тупиков (deadlocks) между транзакциями. На рис. 100 показан простой сценарий возникновения синхронизационного тупика между транзакциями T_1 и T_2. Такой ориентированный граф называется графом ожидания транзакций. Циклы в таком графе обозначает синхронизационный тупик. Далее поговорим о них более подробно.

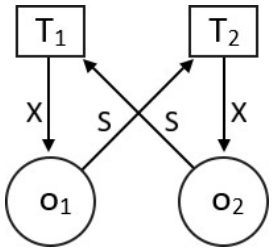


Рис. 100: Цикл обозначает синхронизационный тупик

Транзакции T_1 и T_2 устанавливают монопольные блокировки (в режиме X) объектов o_1 и o_2 соответственно. После этого T_1 требует совместную блокировку объекта o_2, а T_2 – совместную блокировку объекта o_1. Ни одно из этих требований блокировки не может быть удовлетворено, следовательно, ни одна из транзакций не может продолжаться: 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.

Обнаружение тупиковых ситуаций

Основой обнаружения тупиковых ситуаций является построение или постоянное поддержание графа ожидания транзакций. Граф ожидания транзакций – это двудольный ориентированный граф, вершины которого соответствуют либо транзакциям, либо объектам блокировок. Вершины, соответствующие транзакциям, будем изображать прямоугольниками, а вершины, соответствующие объектам блокировок, будем изображать 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.

В системе существует тупиковая ситуация в том и только в том случае, когда в графе ожидания транзакций имеется хотя бы один цикл. Для распознавания тупико-

вых ситуаций периодически производится построение графа ожидания транзакций, и в этом графе ищутся циклы.

Традиционной техникой, для которой существует множество разновидностей нахождения циклов в ориентированном графе является редукция графа. Рассмотрим редукцию графа на примере. В целях упрощения предполагается, что все блокировки являются монопольными (в режиме X), т.е. для каждой вершины-объекта имеется не более одной входящей дуги, т.е. только одна транзакция блокировку удерживает.

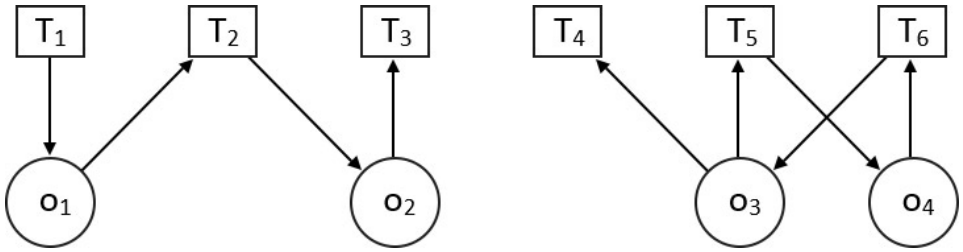


Рис. 101: Граф ожидания для примера поиска тупика

Предположим, что происходит ситуация, описанная на рис. 101:

- Объект o_1 блокирован транзакцией T_1, и его ожидает транзакция T_2.
- Объект o_2 блокирован транзакцией T_2, и его ожидает транзакция T_3.
- Объект o_3 блокирован транзакцией T_6, и его ожидают транзакции T_4 и T_5.
- Объект o_4 блокирован транзакцией T_5, и его ожидает транзакция T_6.

Алгоритм редукции графа ожидания:

1) Удаляются все дуги от транзакций к объектам такие, что данная транзакция не ожидает блокировки каких-либо объектов. Если транзакции не ожидают удовлетворения запроса блокировок, то они не могут создавать тупика.
2) Удаляются все дуги от объектов к транзакциям такие, что у транзакции нет заблокированных объектов. Если транзакции ожидают удовлетворения блокировок, но не удерживают заблокированных объектов, то они не могут создавать тупика.
3) Для объектов без входящих дуг, но с исходящими дугами, ориентация одной произвольной исходящей дуги изменяется на противоположную - это моделирует удовлетворение запроса блокировки.
4) Снова повторяются описанные действия с шага 1 до тех пор, пока не прекратится удаление дуг.

Если в результате работы этого алгоритма в графе останутся дуги, то они обязательно образуют цикл. Рассмотрим пошагово данный алгоритм для нашего примера:

1) Удаляем дугу от транзакции T_1, которая не ожидает блокировок каких-либо объектов.

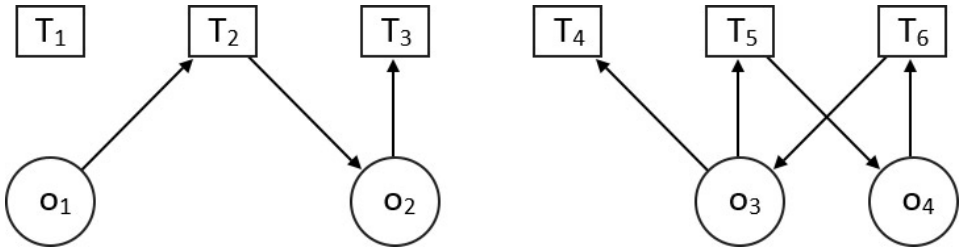


Рис. 102: Удалены дуги от транзакций, не ожидающих блокировок

2) Удаляем дуги к транзакциям T_3 и T_4, у которых нет блокировок.

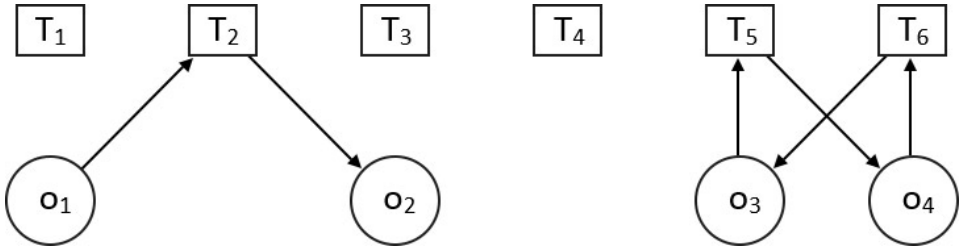


Рис. 103: Удалены дуги к транзакциям без заблокированных объектов

3) Для объекта o_1 без входящих дуг, но с исходящей дугой, её ориентация меняется на обратную.

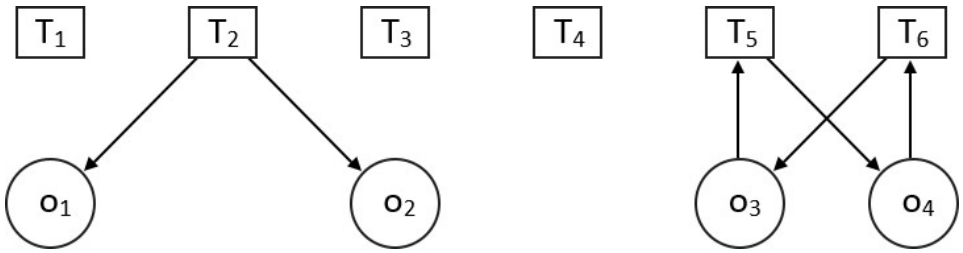


Рис. 104: T_2 блокирует объект o_1

На первом шаге второго раунда будут удалены дуги исходящие от T_2, после чего будет совершён переход на третий раунд, в котором граф не изменится, и алгоритм завершит свою работу.

Разрушение тупиков

Предположим теперь, что нам удалось найти цикл в графе ожидания транзакций. Что делать теперь? Нужно каким-то образом обеспечить возможность продолжения работы хотя бы для части транзакций, попавших в тупик. Разрушение тупика начинается с выбора в цикле 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 наибольшее 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 её 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 не успела израсходовать много системных ресурсов. Но, с другой стороны, такая транзакция не могла и накопить много блокировок, и поэтому её насильственное завершение вряд ли поможет устранить тупиковую ситуацию. Так стоит ли ею жертвовать? 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 её стоимости, в которую с разными весами входят время выполнения, число накопленных блокировок, приоритет и т.д. В качестве «жертвы» выбирается 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.

Метод 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 временных 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 временная метка $t(T)$, соответствующая времени начала выполнения транзакции T. При выполнении операции над объектом o транзакция T помечает его своими идентификатором, временной меткой и типом операции (чтение или изменение). Перед выполнением операции над объектом o транзакция T_2 выполняет следующие действия:

1) Проверяет, помечен ли объект o какой-либо транзакцией T_1. Если не помечен, то помечает этот объект своей временной меткой и типом операции и выполняет операцию.

2) Иначе (если o помечен T_1) транзакция T_2 проверяет, не завершилась ли транзакция T_1, пометившая этот объект. Если транзакция T_1 закончилась, то T_2 помечает объект o и выполняет свою операцию.

3) Иначе (если T_1 не завершилась) T_2 проверяет конфликтность операций. Если операции неконфликтны, то при объекте o запоминается идентификатор транзакции T_2, остается или проставляется временная метка с меньшим значением, и транзакция T_2 выполняет свою операцию.

4) Иначе (если операции транзакций T_2 и T_1 конфликтуют), то если $t(T_2) > t(T_1)$ (т.е. транзакция T_1 «моложе» T_2), то производится откат T_1 и всех других транзакций, идентификаторы которых сохранены при объекте o, и T_2 выполняет свою операцию.

5) Если же $t(T_2) \leq t(T_1)$ (T_1 «старше» T_2), то производится откат T_2, и T_2 получает новую временную метку и начинается заново.

К недостаткам метода ТО относятся потенциально более частые откаты транзакций, чем в случае использования синхронизационных захватов. Это связано с тем, что конфликтность транзакций определяется более грубо. Кроме того, в распределенных системах не очень просто вырабатывать глобальные временные метки с отношением полного порядка это отдельная большая наука. Но в распределенных системах эти недостатки окупаются тем, что не нужно распознавать тупики, а построение графа ожидания в распределенных системах стоит очень дорого.

Версионные методы

Основная идея версионных алгоритмов сериализации транзакций 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 временных меток. 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.

Версионный вариант метода 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 временных меток, в алгоритме MVTO порядок выполнения операций одновременно выполняемых транзакций задается порядком временных меток, которые получают транзакции во время старта.

Временные метки также используются для идентификации версий данных при чтении и модификации – каждая версия получает временную метку той транзакции, которая её записала. Алгоритм MVTO не только следит за порядком выполнения операций транзакций, но также отвечает за преобразование 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 объекта o.

При описании алгоритма будем использовать следующие обозначения. Как и раньше, временную метку, полученную транзакцией T_i в начале её 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:

1) 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 $t(T_k) \leq t(T_i)$.
2) Операция $W_i(o)$ обрабатывается следующим образом.
   - Если уже обработана операция $R_j(o_k)$, такая что $t(T_k) \leq t(T_i) < t(T_j)$, то операция $W_i(o)$ отменяется, а транзакция T_i откатывается.
   - В противном случае $W_i(o)$ преобразуется в $W_i(o_i)$.
3) Завершение (`COMMIT`) любой транзакции T откладывается до завершения всех транзакций, записавших версии объектов, которые прочитала T.

При откате любой транзакции уничтожаются все созданные ею версии объектов базы данных и откатываются все транзакции, прочитавшие хотя бы одну из этих версий («каскадные» откаты).

Основные преимущества алгоритма MVTO - это отсутствие задержек и откатов при выполнении операций чтения, а основной недостаток – возможность возникновения 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.

Версионный вариант протокола 2PL

В двухверсионном варианте протокола 2PL (Two-Version Two-Phase Locking Protocol, 2V2PL) поддерживается до двух версий объектов базы данных.

Текущей версией объекта базы данных будем называть версию, созданную зафиксированной транзакцией с наиболее поздним временем фиксации. Незафиксированной версией объекта БД – версию, созданную еще незавершившейся транзакцией. В соответствии с протоколом 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:

1) операция $R_i(o)$ немедленно выполняется над текущей версией объекта o
2) операция $W_i(o)$, приводящая к созданию новой версии объекта o, выполняется только после завершения (фиксации или отката) транзакции, создавшей незафиксированную версию объекта o
3) 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

Другими словами, есть объект, который все могут читать и есть его версия, которая созданная некоторой незафиксировавшейся транзакцией. В момент фиксации гарантировано незафиксированная версия становится текущей, и какая-то другая транзакция может создать новую незафиксированную версию.

Для реализации такого поведения используются специальные виды синхронизационных 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 (исключается проблема неповторяющихся чтений).
- 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.

	RL	WL	CL
RL	✓	✓	✗
WL	✓	✗	✗
CL	✗	✗	✗

Рис. 105: Таблица совместимости блокировок RL, WL, CL

На рис. 105 изображена таблица совместимости данных блокировок. 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 (т.к. в действительности блокируются разные версии этого объекта). И, конечно, как и во всех схемах сериализации транзакций на основе блокировок, здесь возможны синхронизационные тупики.

Версионно-блокировочный протокол сериализации транзакций для поддержки только читающих транзакций

Гибридный протокол, поддерживающий эффективное выполнение транзакций, не изменяющих состояние базы данных (Multiversion Protocol for Read-Only Transactions, ROMV). При применении этого протокола при образовании каждой транзакции явно указывается её 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 временной 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 ROMV выигрывает в принципиальном отсутствии откатов только читающих транзакций. Конечно, при работе изменяющих транзакций возможно возникновение синхронизационных тупиков и откатов, и здесь требуется использовать обычные методы распознавания и разрушения тупиков.

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.

Заключение

Были описаны основные принципы управления транзакциями в системах управления базами данных, различные методы, алгоритмы и протоколы, способствующие управления транзакциями в соответствии с принципами ACID. Следует заметить, что существует достаточно развитая теория управления транзакциями с собственными средствами формализации постановки задач и доказательства корректности алгоритмов. Для обеспечения более простого понимания сути материала все эти формализмы в курс не включены.

Были рассмотрены два основных подхода к сериализации транзакций – на основе синхронизационных блокировок и временных меток. У каждого из этих подходов имеются свои достоинства и недостатки, но на практике существенно больше распространен метод синхронизационных блокировок.

В заключение лекции были рассмотрены расширения этих подходов с применением версий объектов базы данных. Соответствующие алгоритмы и протоколы позволяют уменьшить число потенциальных конфликтов транзакций, но для их поддержки требуются дополнительные расходы внешней памяти и усложнение общей архитектуры СУБД.

Средства журнализации и восстановления баз данных

Если посмотреть с практической точки зрения, то одним из основных требований к СУБД является надежность хранения баз данных. Это связано с тем, что базы данных слишком дороги, чтобы их лишиться. Даже для транзакционных СУБД, если бизнес лишается базы данных на несколько часов, то он теряет деньги. Требование надёжного хранения 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 и означает, что восстанавливается последнее по времени согласованное состояние базы данных.

Возможны следующие ситуации, при которых требуется производить восстановление состояния базы данных:

- Индивидуальный откат транзакции. Тривиальной ситуацией отката транзакции является её 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 оперативной 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 (жёсткий 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.

Есть несколько способов ведения журнала.

- Один из подходов заключается в поддерживании отдельного локального журнала изменений базы данных для каждой транзакции. Эти локальные журналы используются для индивидуальных откатов транзакций и могут поддерживаться в оперативной (правильнее сказать, в виртуальной) памяти СУБД. Кроме того, поддерживается общий журнал изменений базы данных, используемый для восстановления состояния базы данных после мягких и жёстких сбоев. Данный подход позволяет быстро выполнять индивидуальные откаты транзакций, но приводит к дублированию информации в локальных и общем журналах.
- Поэтому чаще используется другой вариант – поддержка только общего журнала изменений базы данных, который используется и для индивидуальных откатов, и для восстановления после мягкого и жёстких сбоев. Далее мы рассмотрим именно этот вариант.

Буферизация блоков базы данных в оперативной памяти

Журнализация операций изменения базы данных тесно связана не только с управлением транзакциями, но и с буферизацией блоков базы данных в оперативной памяти. По причинам объективно существующей разницы в скорости работы процессоров и оперативной памяти и устройств внешней памяти. Сегодня разница в скорости работы оперативной памяти и внешней памяти, основанной на магнитных дисках составляет 10^4 раз. Эта разница в скорости существовала, существует, и будет существовать, поэтому буферизация блоков базы данных в оперативной 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 оперативной памяти.

Если бы каждая запись об изменении базы данных, которая должна поступить в журнал при выполнении любой операции обновления базы данных, реально бы немедленно перемещалась во внешнюю память, это привело 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, т.е. при нормальной работе СУБД буфер выталкивается во внешнюю память журнала только при полном заполнении записями.

Лекция 25

Смежной к журнализации и управлениями транзакциями является вопрос буферизации блоков базы данных в оперативной памяти (или кэширование в оперативной памяти). Кэширование - это единственный способ амортизации, сглаживания разницы в скоростях между скоростью доступа к оперативной памяти и данным на магнитных дисках. Без кэширования скорость работы СУБД была бы ограничена скоростью работы магнитных дисков, что в 10^4 медленнее скорости ОП.

Аналогично нужно буферизовать журнал в оперативной памяти, иначе любая операция изменения базы данных работала бы со скоростью магнитного диска. Можно было бы работать в режиме старт-стоп: заполнили буфер журнала, затем выталкиваем его во внешнюю память, на это время все операции тормозятся. В этом случае скорость работы с магнитным диском «навязывается» на каждую группу операций изменения.

Поэтому используется техника, которую также называют push-pull (тяни-толкай) или техникой двух полукарманов, которая, в частности, использовалась на БЭСМ-6 для вывода данных на перфокарты. Устройство вывода на перфокарты - это медленное устройство (надо пробивать дырочки в картонке), для запуска которого нужно область перфокарты предварительно подготовить в памяти компьютера. Для работы устройства втечение всего времени, после того, как на него поступал подготовленный образ перфокарты, в это время заполнялся другой буфер, и рассчитывалась скорость таким образом, чтобы пока первая перфокарта будет выдана, вторая будет готова к обработке.

Аналогично для буферизации записей журнала используются два буфера размером в блок внешней памяти. Во время выполнения операций при журнализации изменений заполняется один буфер, и когда он полон, запускается обмен с внешней памятью, и начинается заполняться второй буфер. Т.е. в хорошем случае всё время происходит запись на магнитные диски, и не тормозятся операции изменения. При необходимости можно сделать и больше двух буферов. Это зависит от конкретной скорости работы внешней памяти и насколько быстро заполняются буфера.

Здесь идет речь об использовании буферов, базы данных, журнала, располагающихся именно в физической оперативной памяти. Физическая оперативная память - это память, управление которой СУБД берёт на себя, т.е. это память которой не управляет операционная система. Если бы эти буфера располагались в виртуальной памяти, управляемой операционной системой, то руководствуясь своими собственными стратегиями управления оперативной памяти, в любой момент времени ОС может удалить буферную страницу из оперативной памяти, перенести её 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 оперативную память, выполнив совершенно не ожидаемый СУБД обмен с внешней памятью.

Стоит отметить, что операционная система - это самая умная программа, которая только бывает на компьютере, т.к. она работает в условиях, когда ей толком ничего не известно. Она всё время пытается подстроиться под то, что в данный момент выполняется на компьютере, но у неё нет каких-то априорных знаний касательно работы процессов. С другой стороны, такие априорные знания о работе СУБД есть, не удивительно, у самой СУБД. Поэтому управлять свои кэшем сама СУБД может более грамотно, чем операционная система.

Управление буферным пулом базы данных

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 оперативной памяти, и нет свободных страниц оперативной памяти, то в соответствии с некоторым критерием выбирается некоторая занятая страница оперативной памяти, которая 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
- 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, возможно, выталкивает во внешнюю память базы данных и освобождает её.

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 СУБД располагает большей информацией о страницах буферного пула, чем операционная система о страницах оперативной 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: высокоприоритетные страницы стареют (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.

Физическая синхронизация

Поскольку в СУБД может одновременно (concurrently) выполняться несколько транзакций, вполне реальна ситуация внутри ядра СУБД, 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 отсутствие синхронизации при параллельном изменении 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), где n- номер страницы внешней памяти. 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) – её блокировку в режиме X. Совместимость блокировок обычная для режимов S и 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) были изменены. 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.

Для того, чтобы оставить базу данных в согласованном (физически согласованном) состоянии, при обнаружении ошибки операции нужно ликвидировать все её следы в базе данных и выдать соответствующий код ошибки на уровень RDS (на уровень подсистемы SQL, на уровне конечного пользователя).

Пусть у нас выполняется `UPDATE` 50ти кортежей в таблице. Скажем, 49 выполнилось нормально, а 50-я завершилась с ошибкой. В этом случае нужно ликвидировать результаты 49ти, а по поводу 50-го `UPDATE` сказать, что была ошибка. А вот для одной маленькой операции проще всего ликвидировать её следы, проведя обратные изменения всех страниц 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.

Это условие достаточное, но 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:

- нахождение блока данных для вставки
- вставка кортежа в найденный блок
- обновление первого индекса
- ...
- обновление n-го индекса n (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.

Протокол упреждающей записи в журнал

Реальная ситуация является более сложной. Имеются два вида буферов – буфера журнала и буферный пул страниц оперативной 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 оперативную 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 её 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.

Эта проблема не возникает при индивидуальных откатах транзакций, поскольку в этих случаях содержимое оперативной 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 изменённый 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 изменённый объект базы данных, то перед этим нужно гарантировать выталкивание во внешнюю память журнала буферной страницы журнала, содержащей запись об изменении этого объекта.

Протокол WAL не является необходимым, но следование этому протоколу достаточно для восстановления базы данных после мягкого сбоя. Система рассчитывает на то, что если во внешней памяти базы данных находится некоторый объект базы данных, который изменялся какой-то транзакцией, то во внешней памяти журнала обязательно находится запись, соответствующая этой операции. Обратное неверно, т.е. может так оказаться, что в журнале есть запись об изменении объекта, а сам объект во внешей памяти находится не изменённым.

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 (при протоколе WAL), 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. Представим, что маленькая транзакция меняет 7 блоков внешней памяти, и если при её фиксации будет выполняться 7 обменов с внешней памятью, то все преимущества от буферизации идут коту под хвост, т.е. `COMMIT` становится намного дольше, чем рабочая фаза транзакции.

Оказывается, что можно существенно сэкономить: 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, т.е. `COMMIT`.

Рассмотрим теперь, как можно выполнять операции восстановления базы данных в различных ситуациях, если в системе поддерживается общий для всех транзакций журнал с общей буферизацией записей, поддерживаемый в соответствии с протоколом WAL.

Индивидуальный откат транзакции

Для обеспечения возможности индивидуального отката транзакции в журнале все записи от данной транзакции связываются в обратный список (от последней записи по времени к первой записи этой транзакции). В начале списка находится хронологически последняя запись - это запись о последнем изменении базы данных произведённом данной транзакцией. В случае индивидуального отката транзакции,

хронологически последние записи могут быть еще не вытолкнуты во внешнюю память журнала и находиться в буфере оперативной памяти.

Для закончившихся транзакций, индивидуальные откаты уже невозможны. В этом случае началом списка является запись о конце транзакции, которая обязательно вытолкнута во внешнюю память журнала, т.е. весь список этой транзакций для зафиксированных транзакций находится во внешней памяти. Концом списка (т.е. последней, когда мы идём по обратному ходу) является первая запись об изменении базы данных, которая произведённа данной транзакцией. Обычно в каждой записи проставляется уникальный идентификатор транзакции, по которым можно восстановить прямой список записей об изменениях базы данных данной транзакцией.

Итак, индивидуальный откат транзакции (возможен только для незавершенных транзакций) выполняется следующим образом. Мы идём по списку записей в направлении обратном хронологии:

1) Выбирается очередная журнальная запись из списка записей данной транзакции.
2) Выполняется противоположная по смыслу операция:
   - вместо операции `INSERT` выполняется соответствующая операция `DELETE`
   - вместо операции `DELETE` выполняется `INSERT`
   - вместо прямой операции `UPDATE` – обратная операция `UPDATE`, восстанавливающая предыдущее состояние объекта базы данных

Эти обратные операции называются undo, и они тоже журнализуются. Эта журнализация нужна для того, чтобы при мягком сбое разобраться до какого момента мы восстановили транзакцию. Если полный откат транзакции выполнен успешно, то в журнал заносится запись о конце транзакции, и с точки зрения журнала (и базы данных) такая транзакция является зафиксированной.

Восстановление после мягкого сбоя

К числу основных проблем восстановления после мягкого сбоя относится то, что одна логическая операция (уровня RSS) изменения базы данных может изменять несколько физических блоков базы данных. Например, одна логическая операция может изменять блок данных и несколько блоков индексов.

Блоки базы данных буферизуются в оперативной памяти и выталкиваются независимо. Если происходит мягкий сбой, то может оказаться, что набор блоков внешней памяти базы данных несогласован, 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 исходном блоке в описатель кортежа записывается его новый tid, а в новом блоке размещается сам изменённый 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. они соответствуют либо состоянию объекта либо до его изменения, либо после его изменения. Главное, что не в процессе изменения.

Схема восстановления от точки физической согласованности

Будем считать, что мы умеем восстанавливать физически согласованное состояние базы данных. В журнале отмечаются моменты времени, в которые база данных находится в физически согласованном состоянии. Такие моменты времени называются точками физической согласованности базы данных (point of physical consistency, ppc).

Физическая согласованность означает, что во внешней памяти содержатся согласованные результаты операций, завершившихся до соответствующего момента времени, и отсутствуют результаты операций, которые не завершились.

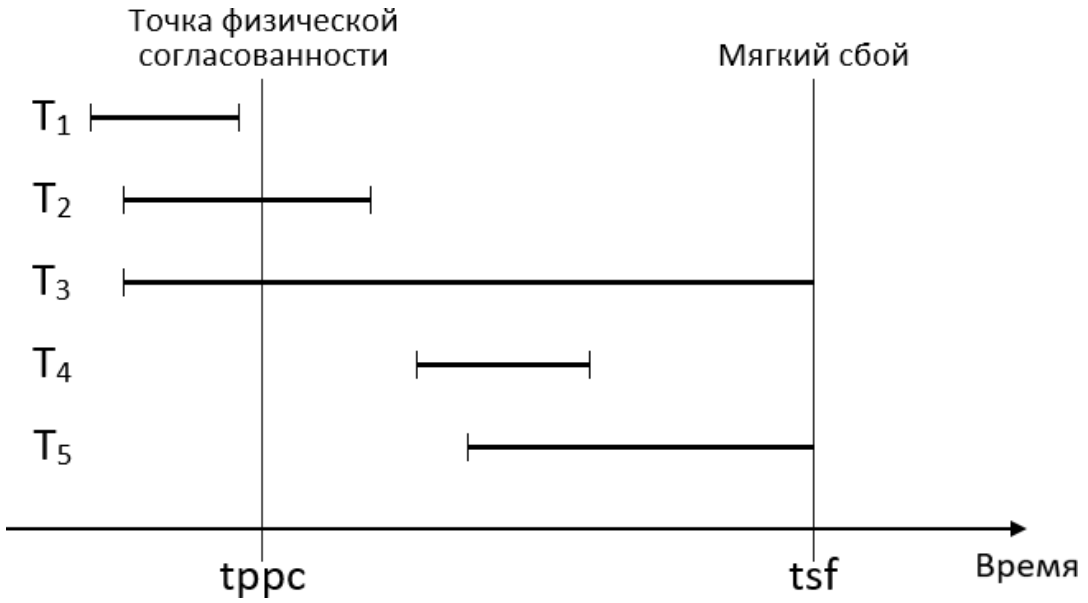


Рис. 106: Возможные сценарии выполнения транзакций при мягком сбое

На рис. 106 показаны все возможные состояния транзакций к моменту мягкого сбоя в момент времени tsf. Предположим, что мы умеем восстанавливать физическую согласованность к моменту времени tppc (далее обсудим как именно это сделать). Рассмотрим какие действия нужно выполнить для каждой из транзакций:

- Для транзакции T_1 никаких действий производить не требуется, т.к. она закончилась до момента tppc, и все её результаты гарантированно отражены во внешней памяти базы данных.
- Транзакция T_2 началась до установки ppc, а закончилась после, поэтому для неё 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 tsf, в журнале содержатся записи обо всех изменениях базы данных, произведённых 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 tsf, при восстановлении необходимо устранить все последствия её выполнения.
- Для транзакции T_4, которая успела начаться после момента tppc и закончиться до момента мягкого сбоя tsf, нужно произвести полное повторное выполнение операций в прямом направлении. Поскольку транзакция T_4 успешно завершилась до момента мягкого сбоя tsf, в журнале содержатся записи обо всех изменениях базы данных, произведённых этой транзакцией.
- Наконец, для транзакции T_5, начавшейся после момента tppc и не успевшей завершиться к моменту мягкого сбоя tsf, никаких действий предпринимать не требуется. Результаты операций этой транзакции полностью отсутствуют во внешней памяти базы данных.

Каким же образом можно обеспечить наличие точек физической согласованности базы данных? Для этого используются два основных подхода:

- подход, основанный на использовании теневого механизма
- подход, в котором применяется журнализация постраничных изменений базы данных

Теневой механизм

Изначально теневой механизм был предложен для поддержки целостности файлов при аварийном отключении питания компьютера. Файл представляется как

набор блоков внешней памяти с логической нумерацией от 1 до n, и каждому логическому блоку соответствует некоторый реальный блок на физическом устройстве. Понятно, что это отображение должно храниться в некоторой таблице. Когда работы с файлом не происходит, то данная таблица хранится в метаданных файла во внешней памяти и называется теневой таблицей. При открытии файла содержимое теневой таблицы копируется в оперативную память, и эта копия называется текущей таблицей.

Основная идея состоит в том, что при открытии файла, когда работающая с файлом программа изменяет блок этого файла, то во внешней памяти выделяется новый блок, в который записывается новое изменённое содержимое. При этом только текущая таблица отображения изменяет ссылку на новый блок, а теневая таблица во внешней памяти остаётся неизменной. При закрытии файла текущая таблица записывается на место теневой таблицы, и освобождаются ранее изменённые блоки.

Если во время работы с открытым файлом происходит мягкий сбой при котором теряется содержимое оперативной памяти, то для восстановления согласованного состояния файла, достаточно просто использовать теневую таблицу отображения.

Как можно применять теневой механизм в контексте баз данных? Вся база данных хранится в нескольких файлах: в System R они назывались сегментами, в Oracle они называются table space, но главное, что это аналоги блочных файлов. Периодически СУБД устанавливает точки физической согласованности базы данных для каждого файла, при этом:

1) Все логические операции завершаются, а новые не начинаются.
2) Выталкиваются все страницы буферного пула базы данных, содержимое которых отличается от содержимого соответствующих блоков внешней памяти.
3) Теневая таблица отображения файлов 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. Т.е. надо, чтобы либо она полностью заменилась, либо отсутствовала во внешней памяти.

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 утрачена теневая таблица отображения (поскольку частично её поменяем, а частично оставим её такой, какой она была) и безнадёжно потеряем текущую, т.к. она была в оперативной памяти, т.е. мы просто лишимся возможности восстановить 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. Назовём 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. После успешной записи текущей таблицы в область B флаг принимает 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.

При использовании теневого механизма восстановление хронологически последнего сохраненного физически согласованного состояния базы данных происходит мгновенно: СУБД устанавливает текущую таблицу отображения равной теневой. Все проблемы восстановления решаются, но за счет слишком большого перерасхода внешней памяти. В пределе может потребоваться вдвое больше внешней памяти, чем реально нужно для хранения базы данных.

Заметим, что чем больше времени между двумя контрольными точками, тем больше требуется внешней памяти. С другой стороны, нельзя выполнять операцию установки контрольный точки слишком часто, т.к. она вычислительно дорогая. Поэтому как правильно выбрать интервалы времени между контрольными точками - вопрос сложный.

Журнализация постраничных изменений

Возможен другой подход, при использовании которого наряду с логической журнализацией операций изменения базы данных производится ещё и журнализация постраничных изменений. Каждый раз когда в какой-то транзакции при выполнении какой-то операции нужно поменять страницу кэша (буферного пула), которая соответствует копии какого-то блока внешней памяти, формируется запись о постраничном изменении.

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 раньше, при выполнении логической операции обновления базы данных, вообще говоря, изменяется несколько блоков базы данных.

Для обеспечения возможности отката отдельной операции, нужно до конца операции монопольно блокировать все страницы буферного пула базы данных, содержащие копии изменяемых этой операцией блоков базы данных. Это может потребоваться, например, если обнаруживается нарушение свойства уникальности какого-либо индекса.

Допустим, что база данных в момент мягкого сбоя находилась в физически согласованно состоянии, но часть изменённых блоков осталась в кэше и во внешнюю память не попала, а другая часть попала. При восстановлении к точке физически согласованного состояния нужно убрать следы незавершённых изменений, а для этого надо уметь различать: для каждой записи в журнале блок базы данных находится в старом состоянии или в новом, которое соответствует этой записи.

Для того, чтобы с этим разобраться, в каждый блок внешней памяти и в каждую запись об изменении этого блока в журнале постраничных изменений записывается монотонно возрастающий номер записи. 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.

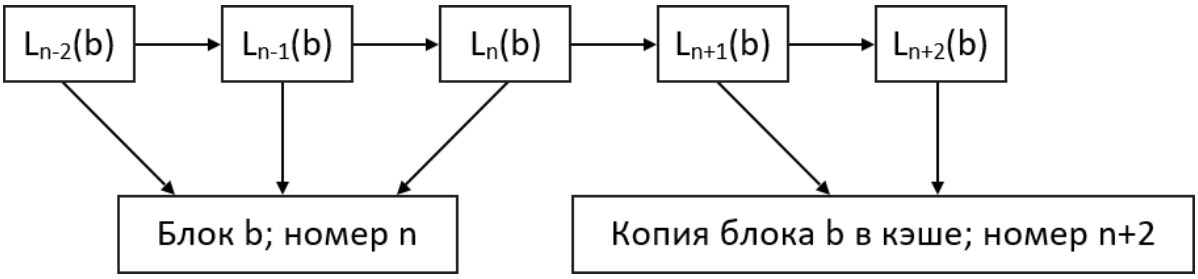


Рис. 107: Записи L_{n+1} и L_{n+2} не изменили блок во внешней памяти

На рис. 107 изображена цепочка записей в журнале об изменении блока b. В самом блоке b указан номер n, который означает, что в состоянии блока отражены результаты операций изменения блока, соответствующих журнальным записям с номерами $n-2$, $n-1$, n. Другими словами, записи с номерами $n-2$, $n-1$, n попали во внешнюю память, а с номерами $n+1$, $n+2$ не попали.

Изменения блока, произведённые операциями, которым соответствуют две хронологически последние журнальные записи $L_{n+1}(b)$ и $L_{n+2}(b)$, 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 $L_n(b), L_{n-1}(b)$ и $L_{n-2}(b)$.

Лекция 26

Мы поняли, что для восстановления согласованности базы данных нужно вести журнал изменений. Вопрос в том, что за записи в этом журнале? Нам нужно уметь делать undo - от текущего блока переходить к предыдущему образу блока. Когда-то разработчики System R пытались придумать язык, который описывал бы изменения блока в компактном виде, но такой язык придумать не удалось. Традиционно, в журнал записывается старое содержимое блока внешней памяти.

В ведении журнала постраничных изменений имеются два поднаправления. В первом поднаправлении поддерживается общий журнал логических и страничных операций. Можно сказать, что есть логический и физический журналы. В результате получаются два вида записей:

- В логическом журнале записи компактные - они содержат идентификатор транзакции, старый кортеж до изменения и новый кортеж после изменения.
- В физическом журнале записи содержат предыдущее содержимое блока и несут иной смысл.

Естественно, наличие двух видов записей, интерпретируемых абсолютно поразному, усложняет структуру журнала. Кроме того, записи о постраничных изменениях, актуальность которых носит локальный характер, существенно 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.

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 об-

разование 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.

В отличие от этого, 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:

1) Прекращают инициироваться новые логические операции.
2) После завершения всех выполняемых логических операций выталкиваются во внешнюю память все изменённые страницы буферного пула.
3) Формируется и выталкивается во внешнюю память логического журнала специальная запись о точке физически согласованного состояния.
4) В случае успешного предыдущего действия разрешается инициация новых логических операций, и физический журнал пишется заново.

Операция выталкивания записи о точки физической согласованности является атомарной:

- Если она успешно выполняется, то при следующем восстановлении после мягкого сбоя будет использоваться новая точка физически согласованного состояния.
- Иначе ситуация воспринимается как мягкий сбой, и логически согласованное состояние базы данных восстанавливается от предыдущей точки физически согласованного состояния (с оповещением администратора базы данных).

Если внимательно посмотреть на то, как работает теневой механизм для установки точки физически согласованного состояния, и как работает журнал постраничных изменений, то оказывается что они чрезвычайно похожи. И там, и там сохраняются лишние блоки во внешней памяти. При использовании теневого механизма нам нужен один лишний блок на каждый изменённый блок базы данных на целый интервал времени между двумя точками физической согласованности. В случае же журнала постраничных изменений, нам требуется столько лишних блоков сколько было операций изменений этого блока. Т.е. в общем случае это более дорогой способ. Поэтому похоже на то, что теневой механизм более пригоден для целей восстановления физической согласованности базы данных.

Восстановление базы данных после жёсткого сбоя

Обсудим что делать в случае, когда потерян носитель базы данных. Понятно, что для восстановления последнего согласованного состояния базы данных после жёсткого сбоя логического журнала изменений базы данных явно недостаточно (не к чему его применять). Самый простой способ восстановления основывается на использовании логического журнала и последней архивной копии базы данных. В этом случае восстановление начинается с обратного копирования базы данных из архивной копии на исправный носитель. 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 во внешней памяти базы данных будут присутствовать результаты всех транзакций, которые успешно зафиксировались и будут отсутствовать результаты транзакций, которые не успели завершиться к моменту жёсткого сбоя. Таким образом будет получено последнее до момента жёсткого сбоя логически согласованное состояние базы данных.

Вопрос, а как выполнять эти операции? Самый прямолинейный способ - это проигрывать все транзакции в рабочем режиме системы, т.е. для каждой новой транзакции в журнале образовывать новую транзакцию в СУБД и в ней выполнять нужные операции со всей необходимой синхронизацией, т.е. точно так же как в рабочем режиме. Однако, если придерживаться некоторой дисциплины выполнения логических операций над базой данных, то при восстановлении базы данных после жёсткого сбоя можно просто последовательно повторно выполнять операции по журналу, не обращая внимание на то, в каких транзакциях они выполнялись до жёсткого 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

Если работать именно по такой дисциплине, то последовательность записей в логическом журнале будет в точности соответствовать тому сериальному плану, по которому выполнялись транзакции в рабочем режиме.

На самом деле, поскольку жёсткий 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 жёсткого сбоя – это достаточно длительный процесс.

Несмотря на то, что к ведению журнала предъявляются особые требования по части надежности, в принципе, возможна и его утрата. Тогда единственным способом восстановления базы данных является возврат к последней архивной копии. Конечно,

в этом случае не удастся получить последнее согласованное состояние базы данных, но это лучше, чем ничего.

Последний вопрос, который в данном курсе следует обсудить, касается того как производить архивные копии базы данных и журнала. Современные СУБД умеют делать архивирование в режиме online, не нарушая работу пользователей, но мы не будем заострять на этом внимание, т.к. это дело техники. Традиционное архивирование базы данных происходит в offline, когда база данных не используется (например, в ночное время).

Интересно, что вместо архивирования самой базы данных, можно архивировать журнал, который весит заметно меньше. Для создания архивной копии журнала нужно:

- запретить создание новых транзакций и дождаться завершения всех транзакций
- вытолкнуть все страницы буферного пула во внешнюю память
- скопировать журнал

Тогда для полного восстановления базы данных после жёсткого сбоя достаточно иметь архивную копию базы данных, последовательность архивных копий журнала 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.

Эта оптимизация состоит в том, что архивированный логический журнал можно существенно сжать:

- 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.

На рис. 108 показан процесс сжатия последовательности журнальных записей, соответствующих последовательности операций над кортежем, у которого tid = k и имеются четыре целочисленных поля. Слева сверху показан исходный журнал, который содержит одну запись вставки кортежа и три записи обновления. Операция `UPDATE (tid=k, (1,3), (6,30))` обозначает, что первый и третий столбцы кортежа меняют значения на 6 и 30 соответственно. Логично, что можно сразу вставлять кортеж с такими значениями, что и происходит после первого шага процесса сжатия. Аналогичными рассуждениями мы приходим к тому, что можно сразу вставить результирующий кортеж. Очевидно, что если хронологически последней в последовательности является запись, соответствующая операции `DELETE`, то после сжатия последовательность станет пустой.

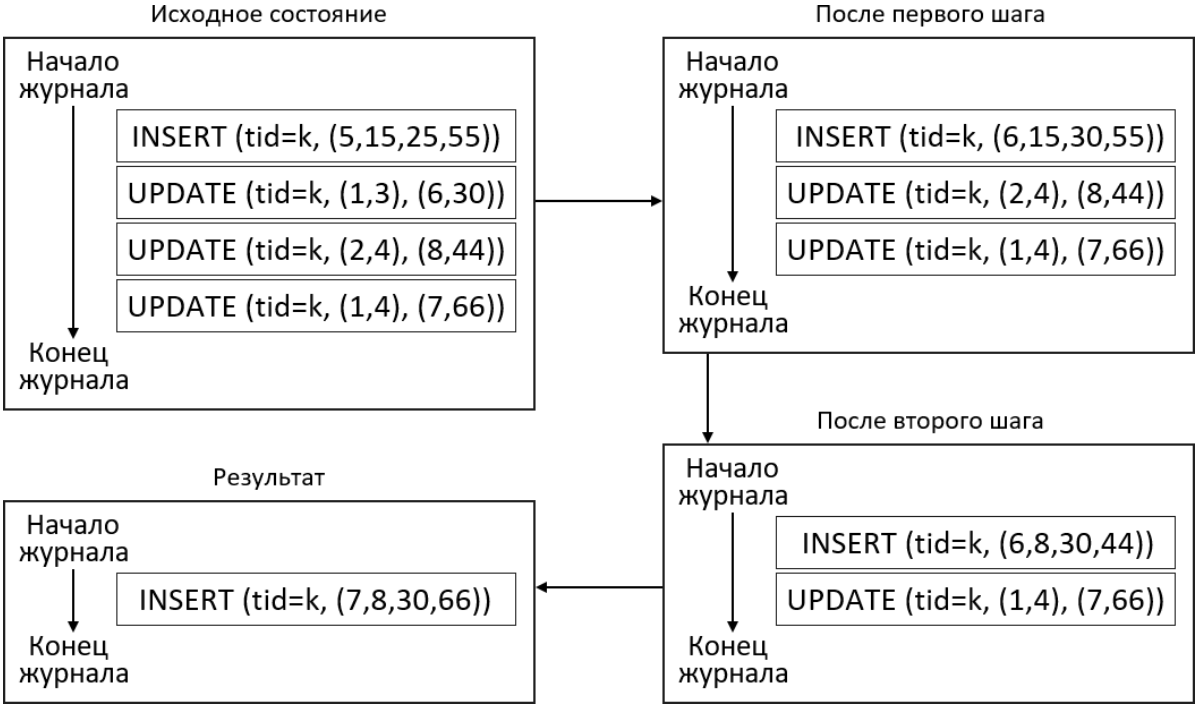


Рис. 108: Процесс сжатия последовательности журнальных записей

Заметим, что точно таким же образом можно совместно сжать два хронологически последовательных полных или сжатых журнала. Таким образом, для восстановления базы данных после жёсткого сбоя можно воспользоваться:

- исходной архивной копией
- одним сжатым архивным журналом
- последним логическим журналом

Могут возникнуть сомнения относительно сложности и продолжительности процесса сжатия журнала. Но здесь следует заметить, что эта работа не должна выполняться на сервере базы данных, такое сжатие может выполняться на отдельном компьютере в режиме offline. Кроме того, если в какой-то момент происходит жёсткий сбой и имеются архивная копия базы данных, сжатый архивный журнал (начиная от этой копии) и набор еще не сжатых архивных журналов, то этого уже достаточно для восстановления, так что сроки завершения процесса полного сжатия не являются критическими.

Заключение

Были рассмотрены основные принципы и алгоритмы подсистем СУБД, предназначенных для управления буферами оперативной памяти, журнализации и восстановления базы данных после различных сбоев. Изложение велось без технических деталей, таких как возможные структуры данных журналов.

ФАКУЛЬТЕТ
ВЫЧИСЛИТЕЛЬНОЙ
МАТЕМАТИКИ И
КИБЕРНЕТИКИ
МГУ ИМЕНИ
М.В. ЛОМОНОСОВА