Базы данных. Вводный курс

Сергей Кузнецов

Содержание

Лекция 1. Эволюция устройств внешней памяти и программных систем управления данными

1.1. Введение

В этой вводной лекции мы, прежде всего, обсудим предпосылки появления в компьютерах устройств внешней памяти, а также обоснуем принципиальную важность для организации информационных систем дисковых устройств с подвижными магнитными головками. Далее будут рассмотрены особенности организации и основное функциональное назначение одного из ключевых компонентов современных операционных систем – систем управления файлами. Наконец, в разделе 1.4. Потребности информационных систем мы покажем, почему возможностей файловых систем недостаточно для создания информационных программных систем. Будет продемонстрировано, что естественные требования информационных систем к средствам управления данными во внешней памяти приводят к необходимости наличия систем управления базами данных (СУБД). В ходе этого анализа будут определены основные черты, которыми должны обладать СУБД.

1.2. Устройства внешней памяти

В самом широком смысле *информационная система* представляет собой программный

комплекс, функции которого состоят в поддержке надежного хранения информации в памяти компьютера, выполнении специфических для данного приложения преобразований информации и/или вычислений, предоставлении пользователям удобного и легко осваиваемого интерфейса. Обычно объемы данных, с которыми приходится иметь дело таким системам, достаточно велики, а сами данные обладают достаточно сложной структурой. Классическими примерами информационных систем являются банковские системы, системы резервирования авиационных или железнодорожных билетов, мест в гостиницах и т. д.

О надежном и долговременном хранении информации можно говорить только при наличии запоминающих устройств, сохраняющих информацию после выключения электропитания. Оперативная (основная) память этим свойством обычно не обладает. В первые десятилетия развития вычислительной техники использовались два вида устройств внешней памяти: магнитные ленты и магнитные барабаны. При этом емкость магнитных лент была достаточно велика, но по своей природе они обеспечивали последовательный доступ к данным. Емкость магнитной ленты пропорциональна ее длине. Чтобы получить доступ к требуемой порции данных, нужно в среднем перемотать половину ее длины. Но чисто механическую операцию перемотки нельзя выполнить очень быстро. Поэтому быстрый произвольный доступ к данным на магнитной ленте, очевидно, невозможен.

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

Указанные ограничения не очень существенны для систем численных расчетов. Обсудим более подробно, какие реальные потребности возникают у разработчиков систем численных расчетов. Прежде всего, для получения требуемых результатов серьезные вычислительные программы должны проработать достаточно долгое время (недели, месяцы и даже, может быть, годы). Наличие гарантий надежности со стороны производителей аппаратных компьютерных средств не избавляет программистов от необходимости использования программного сохранения частичных результатов вычислений, чтобы при возникновении непредвиденных сбоев аппаратуры можно было продолжить выполнение расчетов с некоторой контрольной точки. Для сохранения промежуточных результатов идеально подходят магнитные ленты: при выполнении процедуры установки контрольной точки данные последовательно сбрасываются на ленту, а при необходимости перезапуска от сохраненной контрольной точки данные также последовательно с ленты считываются.

Вторая традиционная потребность численных программистов – максимально большой объем оперативной памяти. Большая оперативная память требуется, во-первых, для того, чтобы обеспечить программе быстрый доступ к большому количеству обрабатываемых данных. Во-вторых, сложные вычислительные программы сами могут иметь большой объем. Поскольку объем реально доступной в ЭВМ оперативной памяти всегда являлся недостаточным для удовлетворения текущих потребностей вычислений, требовалась быстрая внешняя память для организации оверлеев и/или виртуальной памяти. Мы не будем здесь вдаваться в детали организации этих механизмов программного расширения оперативной памяти, но заметим, что для этого идеально подходили магнитные барабаны. Они обеспечивают быстрый доступ к внешней памяти, а для расширения оперативной памяти одной программы (сложные вычислительные программы, как правило, выполняются на компьютере в одиночку)

большой объем внешней памяти не требуется.

Далее заметим, что, даже если программа должна обработать (или произвести) большой объем информации, при программировании можно продумать расположение этой информации во внешней памяти, чтобы программа работала как можно быстрее. Развитая поддержка работы с внешней памятью со стороны общесистемных программных средств не обязательна, а иногда и вредна, поскольку приводит к дополнительным накладным расходам аппаратных ресурсов.

Однако для информационных систем, в которых объем постоянно хранимых данных определяется спецификой бизнес-приложения, а потребность в текущих данных определяется пользователем приложения, одних только магнитных барабанов и лент недостаточно. Емкость магнитного барабана просто не позволяет долговременно хранить данные большого объема. Что же касается лент, то представьте себе состояние человека, который, стоя у билетной кассы, должен дождаться полной перемотки магнитной ленты. Естественным требованиям к таким системам является обеспечение высокой средней скорости выполнения операций при наличии больших объемов данных.

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

Магнитные диски представляют собой пакеты магнитных пластин (поверхностей), между которыми на одном рычаге двигается пакет магнитных головок (рис. 1.1). Шаг движения пакета головок является дискретным, и каждому положению пакета головок логически соответствует цилиндр пакета магнитных дисков. На каждой поверхности цилиндр «высекает» дорожку, так что каждая поверхность содержит число дорожек, равное числу цилиндров. При разметке магнитного диска (специальном действии, предшествующем использованию диска) каждая дорожка размечается на одно и то же количество блоков; таким образом, предельная емкость каждого блока составляет одно и то же число байтов. Для задания обмена с магнитным диском на уровне аппаратуры нужно указать номер цилиндра, номер поверхности, номер блока на соответствующей дорожке и число байтов, которое нужно записать или прочитать от начала этого блока.

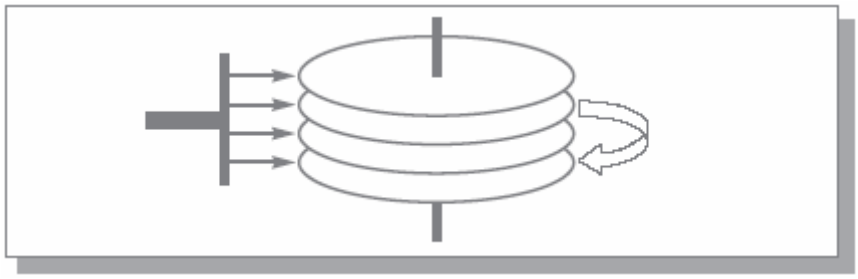


Рис. 1.1. Грубая схема дискового устройства памяти с подвижными головками

При выполнении обмена с диском аппаратура выполняет три основных действия: подвод головок к нужному цилиндру (обозначим время выполнения этого действия как $t_{пг}$), поиск на дорожке нужного блока (время выполнения – $t_{пб}$) и собственно обмен с этим блоком

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

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

Лекция 2. Понятие модели данных. Обзор разновидностей моделей данных

2.1. Введение

Историю технологии БД принято отсчитывать с начала 1960-х гг., когда появились первые попытки создания специальных программных средств управления базами данных. За прошедшие десятилетия возникали и использовались различные подходы к организации баз данных. Для описания и сравнения некоторых из них мы воспользуемся понятием *модели данных*, предложенным в 1969 г. Эдгаром Коддом [2.1]. Кодд ввел это понятие для описания конкретного *реляционного подхода* к организации БД. Соответственно, он говорил о *реляционной модели данных*, различным теоретическим и реализационным аспектам которой в основном посвящен этот курс. Однако понятие модели данных оказалось удобным не только для описания реляционного подхода и сравнения реализаций реляционных СУБД, но и для реализационно-независимого представления и сопоставления других подходов к организации баз данных.

2.2. Модель данных

В модели данных описывается некоторый набор родовых понятий и признаков, которыми должны обладать все конкретные СУБД и управляемые ими базы данных, если они основываются на этой модели. Наличие модели данных позволяет сравнивать конкретные реализации, используя один общий язык.

Хотя понятие модели данных было введено Коддом, наиболее распространенная трактовка модели данных, по-видимому, принадлежит Кристоферу Дейту, который воспроизводит ее (с различными уточнениями) применительно к реляционным БД практически во всех своих книгах (см., например, [1.3]). Согласно Дейту реляционная модель состоит из трех частей, описывающих разные аспекты реляционного подхода: структурной части, манипуляционной части и целостной части.

В структурной части модели данных фиксируются основные логические структуры данных, которые могут применяться на уровне пользователя при организации БД, соответствующих данной модели. Например, в модели данных SQL основным видом структур базы данных являются таблицы, а в объектной модели данных – объекты ранее определенных типов.

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

Наконец, в целостной части модели данных (которая явно выделяется не во всех известных моделях) специфицируются механизмы ограничений целостности, которые обязательно должны поддерживаться во всех реализациях СУБД, соответствующих данной модели. Например, в целостной части реляционной модели данных категорически требуется поддержка ограничения первичного ключа в любой переменной отношения, а аналогичное требование к таблицам в модели данных SQL отсутствует.

В этой лекции мы применим понятие модели данных для обзора как подходов, предшествовавших появлению реляционных баз данных, так и подходов, которые возникли позже. Мы не будем касаться особенностей каких-либо конкретных систем; это привело бы к изложению многих технических деталей, которые, хотя и интересны, но находятся несколько в стороне от основной цели курса.

2.3. Ранние модели данных

Начнем с рассмотрения общих подходов к организации трех типов ранних систем, а именно, систем, основанных на инвертированных списках, иерархических и сетевых систем управления базами данных. В целом ранние системы можно охарактеризовать следующим образом[1)]:

- Эти системы активно использовались в течение многих лет, задолго до появления работоспособных реляционных СУБД. На самом деле некоторые из ранних систем используются даже в наше время, накоплены громадные базы данных, и одной из актуальных проблем информационных систем является использование этих систем совместно с современными.
- Все ранние системы не основывались на каких-либо абстрактных моделях. Как мы упоминали, понятие модели данных фактически вошло в обиход специалистов в области БД только вместе с реляционным подходом. Абстрактные представления ранних систем появились позже на основе анализа и выявления общих признаков у различных конкретных систем.

- В ранних системах доступ к БД производился на уровне записей. Пользователи этих систем осуществляли явную навигацию в БД, используя языки программирования, расширенные функциями СУБД. Интерактивный доступ к БД поддерживался только путем создания соответствующих прикладных программ с собственным интерфейсом.
- Можно считать, что уровень средств ранних СУБД соотносится с уровнем файловых систем примерно так же, как уровень языка Cobol соотносится с уровнем языков ассемблера. Заметим, что при таком взгляде уровень реляционных систем соответствует уровню языков Ада или APL.
- Навигационная природа ранних систем и доступ к данным на уровне записей заставляли пользователей самих производить всю оптимизацию доступа к БД, без какой-либо поддержки системы.
- После появления реляционных систем большинство ранних систем было оснащено «реляционными» интерфейсами. Однако в большинстве случаев это не сделало их по-настоящему реляционными системами, поскольку оставалась возможность манипулировать данными в естественном для них режиме.

2.3.1. Модель данных инвертированных таблиц

К числу наиболее известных и типичных представителей систем, в основе которых лежит эта модель данных, относятся СУБД Datacom/DB, выведенная на рынок в конце 1960-х гг. компанией Applied Data Research, Inc. (ADR) и принадлежащая в настоящее время компании Computer Associates, и Adabas (ADAptable DAtabase System), которая была разработана компанией Software AG в 1971 г. и до сих пор является ее основным продуктом.

Организация доступа к данным на основе инвертированных таблиц используется практически во всех современных реляционных СУБД, но в этих системах пользователи не имеют непосредственного доступа к инвертированным таблицам (индексам). Кстати, когда мы будем рассматривать внутренние интерфейсы реляционных СУБД, можно будет увидеть, что они очень близки к пользовательским интерфейсам систем, основанных на инвертированных таблицах.

Структуры данных

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

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

Манипулирование данными

Поддерживаются два класса операций:

1. Операции, устанавливающие адрес записи и разбиваемые на два подкласса:
    - прямые поисковые операторы (например, установить адрес первой записи

таблицы по некоторому пути доступа);
    - операторы, устанавливающие адрес записи при указании относительной позиции от предыдущей записи по некоторому пути доступа.
2. Операции над адресуемыми записями.

Вот типичный набор операций:

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

Ограничения целостности

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

2.3.2. Иерархическая модель данных

Типичным представителем (наиболее известным и распространенным) является СУБД IMS (Information Management System) компании IBM. Первая версия системы появилась в 1968 г.

Иерархические структуры данных

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

На рис. 2.1 показан пример типа дерева (схемы иерархической БД). Здесь тип записи `Отдел` является предком для типов записи `Руководитель` и `Служащие`, а `Руководитель` и `Служащие` – потомки типа записи `Отдел`. Смысл полей типов записей в основном должен быть понятен по их именам. Поле `Рук_Отдел` типа записи `Руководитель` содержит номер

отдела, в котором работает служащий, являющийся данным руководителем (предполагается, что он работает не обязательно в том же отделе, которым руководит). Между типами записи поддерживаются связи (правильнее сказать, *типы* связей, поскольку реальные связи появляются в экземплярах типа дерева).

Рис. 2.1. Пример типа дерева

База данных с такой схемой могла бы выглядеть так, как показано на рис. 2.2 (мы показываем один экземпляр дерева).

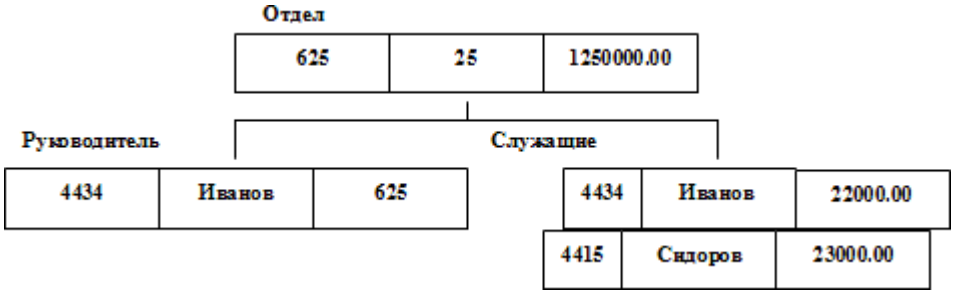


Рис. 2.2. Пример иерархической базы данных

Все экземпляры данного типа потомка с общим экземпляром типа предка называются близнецами. Для иерархической базы данных определяется полный порядок обхода дерева: сверху-вниз, слева-направо. Заметим, что в терминологии IMS вместо термина *запись* использовался термин *сегмент*, а под *записью базы данных* понималось все дерево сегментов.

Манипулирование данными

Примерами типичных операций манипулирования иерархически организованными данными могут быть следующие:

- 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;
- вставить новую запись в указанную позицию;
- удалить текущую запись.

Ограничения целостности

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

записями без связи «предок-потомок», не обеспечивается. Примером такой «внешней» ссылки является содержимое поля `Рук_Отдел` в экземпляре типа записи `Руководитель`.

2.3.3. 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. Архитектура системы основана на предложениях Data Base Task Group (DBTG) организации CODASYL (COnference on DAta SYstems Languages), которая отвечала за определение 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. В настоящее время IDMS принадлежит компании Computer Associates.

Сетевые структуры данных

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

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

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

- каждый экземпляр типа записи *P* является предком только в одном экземпляре типа связи *L*;
- каждый экземпляр типа записи *C* является потомком не более чем в одном экземпляре типа связи *L*.

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

- тип записи потомка в одном типе связи *L1* может быть типом записи предка в другом типе связи *L2* (как в иерархии);
- данный тип записи *P* может быть типом записи предка в любом числе типов связи;
- данный тип записи *P* может быть типом записи потомка в любом числе типов связи;
- может существовать любое число типов связи с одним и тем же типом записи предка и одним и тем же типом записи потомка; и если *L1* и *L2* - два типа связи с одним и тем же типом записи предка *P* и одним и тем же типом записи потомка *C*, то правила, по которым образуется родство, в разных связях могут различаться;
- типы записи *X* и *Y* могут быть предком и потомком в одной связи и потомком и предком – в другой;
- предок и потомок могут быть одного типа записи.

На рис. 2.3 показан простой пример схемы сетевой БД. На этом рисунке показаны три типа записи: Отдел, Служащие и Руководитель и три типа связи: Состоит из служащих, Имеет руководителя и Является служащим**.** В типе связи Состоит из служащих типом записи-предком является Отдел, а типом записи-потомком – Служащие (экземпляр этого типа связи связывает экземпляр типа записи Отдел со многими экземплярами типа записи Служащие, соответствующими всем служащим данного отдела). В типе связи Имеет руководителя типом записи-предком является Отдел, а типом записи-потомком – Руководитель (экземпляр этого типа связи связывает экземпляр типа записи Отдел с одним экземпляром типа записи Руководитель, соответствующим руководителю данного отдела). Наконец, в типе связи Является служащим типом записи-предком является Руководитель, а типом записи-потомком – Служащие (экземпляр этого типа связи связывает экземпляр типа записи Руководитель с одним экземпляром типа записи Служащие, соответствующим тому служащему, которым является данный руководитель).

Рис. 2.3. Пример схемы сетевой базы данных

Манипулирование данными

Вот примерный набор операций манипулирования данными:

- найти конкретную запись в наборе однотипных записей (например, служащего с именем Иванов);
- перейти от предка к первому потомку по некоторой связи (например, к первому служащему отдела 625);
- перейти к следующему потомку в некоторой связи (например, от Иванова к Сидорову);
- перейти от потомка к предку по некоторой связи (например, найти отдел, в котором работает Сидоров);
- создать новую запись;
- уничтожить запись;
- модифицировать запись;
- включить в связь;
- исключить из связи;
- переставить в другую связь и т.д.

Ограничения целостности

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

1 Заметим, что перечисляемые ниже характеристики в полной мере относятся и к другим не реляционным подходам к организации баз данных, которые возникли до появления реляционного подхода или почти одновременно с ним. В частности, подобными свойствами обладают системы, основанные на подходах MUMPS (наиболее известной в России является реализация этого подхода в СУБД Cache компании Intersystems) и Pick (этот подход реализован во многих СУБД, в частности, в СУБД UniVerse и UniData семейства U2 компании IBM).

Лекция 3. Введение в реляционную модель данных

3.1. Введение

В этом курсе, главным образом, обсуждаются различные аспекты реляционных баз данных. Принято считать, что реляционный подход к организации баз данных был заложен в конце 1960-х гг. Эдгаром Коддом [2.1]. В последние десятилетия этот подход является наиболее распространенным (с оговоркой, что в называемых в обиходе реляционными системах баз данных, основанных на языке SQL, в действительности нарушаются некоторые важные принципы классического реляционного подхода). Достоинствами реляционного подхода принято считать следующие свойства: реляционный подход основывается на небольшом числе интуитивно понятных абстракций, на основе которых возможно простое моделирование наиболее распространенных предметных областей; эти абстракции могут быть точно и формально определены; теоретическим базисом реляционного подхода к организации баз данных служит простой и мощный математический аппарат теории множеств и математической логики; реляционный подход обеспечивает возможность ненавигационного манипулирования данными без необходимости знания конкретной физической организации баз данных во внешней памяти. Компьютерный мир далеко не сразу признал реляционные системы. В 70-е года прошлого века, когда уже были получены почти все основные теоретические результаты и даже существовали первые прототипы реляционных СУБД, многие авторитетные специалисты отрицали возможность добиться эффективной реализации таких систем. Однако преимущества реляционного подхода и развитие методов и алгоритмов организации и управления реляционными базами данных привели к тому, что к концу 80-х годов реляционные системы заняли на мировом рынке СУБД доминирующее положение. В этой лекции на сравнительно неформальном уровне вводятся основные понятия реляционных баз данных, а также определяется сущность реляционной модели данных. Основной целью лекции является демонстрация простоты и возможности интуитивной интерпретации этих понятий. В следующих лекциях будут приводиться более формальные определения, на которых основана теория реляционных баз данных.

3.2. Основные понятия реляционных баз данных

Выделим следующие основные понятия реляционных баз данных: тип данных, домен, атрибут, кортеж, отношение, первичный ключ.

Для начала покажем смысл этих понятий на примере отношения СЛУЖАЩИЕ, содержащего информацию о служащих некоторого предприятия (рис. 3.1).

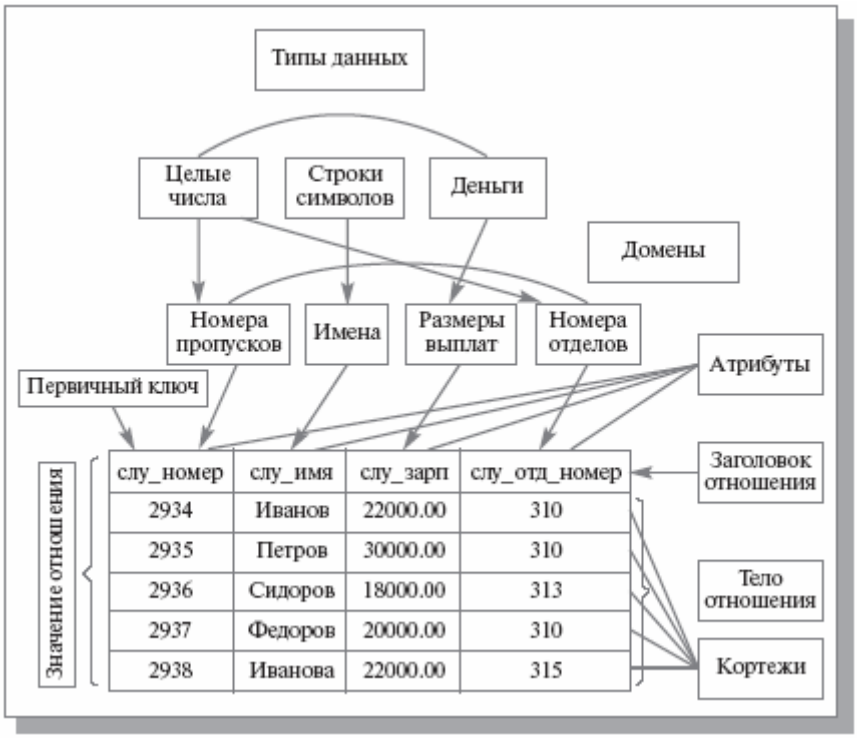


Рис. 3.1. Соотношение основных понятий реляционного подхода

3.2.1. Тип данных

Значения данных, хранимые в реляционной базе данных, являются типизированными, т. е. известен тип каждого хранимого значения. Понятие типа данных в реляционной модели данных полностью соответствует понятию типа данных в языках программирования. Напомним, что традиционное (нестрогое) определение *типа данных* состоит из трех основных компонентов: определение множества значений данного типа; определение набора операций, применимых к значениям типа; определение способа внешнего представления значений типа (литералов).

Обычно в современных реляционных базах данных допускается хранение символьных, числовых данных (точных и приблизительных), специализированных числовых данных (таких, как «деньги»), а также специальных «темпоральных» данных (дата, время, временной интервал). Кроме того, в реляционных системах поддерживается возможность определения пользователями собственных типов данных (более подробно мы обсудим это в лекции 23).

В примере на рис. 3.1 мы имеем дело с данными трех типов: строки символов, целые числа и «деньги».

3.2.2. Домен

Понятие домена более специфично для баз данных, хотя и имеются аналогии с подтипами в некоторых языках программирования (более того, в своем *«Третьем манифесте»* [1.5], [2.8], [3.3] Кристофер Дейт и Хью Дарвен вообще ликвидируют различие между доменом и типом данных). В общем виде домен определяется путем задания некоторого базового типа данных, к которому относятся элементы домена, и произвольного логического выражения, применяемого к элементу этого типа данных (*ограничения домена*). Элемент данных является элементом домена в том и только в том случае, если вычисление этого логического

выражения дает результат *истина* (для логических значений мы будем попеременно использовать обозначения *истина* и *ложь* или *true* и *false*). С каждым доменом связывается имя, уникальное среди имен всех доменов соответствующей базы данных.

Наиболее правильной интуитивной трактовкой понятия *домена* является его восприятие как допустимого потенциального, ограниченного подмножества значений данного типа. Например, домен ИМЕНА в нашем примере определен на базовом типе символьных строк, но в число его значений могут входить только те строки, которые могут представлять имена (в частности, для возможности представления русских имен такие строки не могут начинаться с мягкого или твердого знака и не могут быть длиннее, например, 20 символов). Если некоторый атрибут отношения определяется на некотором домене (как, например, на рис. 3.1 атрибут СЛУ_ИМЯ определяется на домене ИМЕНА), то в дальнейшем ограничение домена играет роль ограничения целостности, накладываемого на значения этого атрибута.

Следует отметить также семантическую нагрузку понятия домена: данные считаются сравнимыми только в том случае, когда они относятся к одному домену. В нашем примере значения доменов НОМЕРА ПРОПУСКОВ и НОМЕРА ОТДЕЛОВ относятся к типу целых чисел, но не являются сравнимыми (допускать их сравнение было бы бессмысленно).

3.2.3. Заголовок отношения, кортеж, тело отношения, значение отношения, переменная отношения

Понятие отношения является наиболее фундаментальным в реляционном подходе к организации баз данных, поскольку n-арное отношение является единственной родовой структурой данных, хранящихся в реляционной базе данных. Это отражено и в общем названии подхода – термин *реляционный (relational)* происходит от *relation (отношение)*. Однако сам термин отношение является исключительно неточным, поскольку, говоря про любые сохраняемые данные, мы должны иметь в виду *тип* этих данных, *значения* этого типа и *переменные*, в которых сохраняются значения. Соответственно, для уточнения термина отношение выделяются понятия *заголовка* отношения, *значения* отношения и *переменной* отношения. Кроме того, нам потребуется вспомогательное понятие кортежа.

Итак, *заголовком* (или *схемой*) *отношения* r (Hr) называется конечное множество упорядоченных пар вида <A, T>, где A называется именем атрибута, а T обозначает имя некоторого базового типа или ранее определенного домена. По определению требуется, чтобы все имена атрибутов в заголовке отношения были различны. В примере на рис. 3.1 заголовком отношения СЛУЖАЩИЕ является множество пар {<слу_номер, номера_пропусков>, <слу_имя, имена>, <слу_зарп, размеры_выплат>, <слу_отд_номер, номера_отделов>}.

Если все атрибуты заголовка отношения определены на разных доменах, то, чтобы не плодить лишних имен, разумно использовать для именования атрибутов имена соответствующих доменов (не забывая, конечно, о том, что это всего лишь удобный способ именования, который не устраняет различия между понятиями домена и атрибута).

Кортежем tr, соответствующим заголовку Hr, называется множество упорядоченных триплетов вида <A, T, v>, по одному такому триплету для каждого атрибута в Hr. Третий элемент – v – триплета <A, T, v> должен являться допустимым значением типа данных или домена T. Заголовку отношения СЛУЖАЩИЕ соответствуют, например, следующие кортежи: {<слу_номер, номера_пропусков, 2934>, <слу_имя, имена,

Иванов>, <слу_зарп, размеры_выплат, 22.000>, <слу_отд_номер, номера_отделов, 310>}, {<слу_номер, номера_пропусков, 2940>, <слу_имя, имена, Кузнецов>, <слу_зарп, размеры_выплат, 35.000>, <слу_отд_номер, номера_отделов, 320>}.

Телом Br отношения r называется произвольное множество кортежей tr. Одно из возможных тел отношения СЛУЖАЩИЕ показано на рис. 3.1. Заметим, что в общем случае, как это демонстрируют, в частности, рис. 3.1 и пример предыдущего абзаца, могут существовать такие кортежи tr, которые соответствуют Hr, но не входят в Br.

Значением Vr отношения r называется пара множеств Hr и Br. Одно из допустимых значений отношения СЛУЖАЩИЕ показано на рис. 3.1.

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

Здесь стоит подчеркнуть, что любая принятая на практике операция обновления базы данных – INSERT (вставка кортежа в переменную отношения), DELETE (удаление кортежа из значения-отношения переменой отношения) и UPDATE (модификация кортежа значения-отношения переменной отношения) – с модельной точки зрения является операцией присваивания переменной отношения некоторого нового значения-отношения. Это совсем не означает, что перечисленные операции должны выполняться именно таким образом в СУБД: главное, чтобы результат операций соответствовал этой модельной семантике.

Заметим, что в дальнейшем в тех случаях, когда точный смысл термина понятен из контекста, мы будем использовать термин отношение как в смысле значение отношения, так и в смысле переменная отношения.

По определению, *степенью*, или *«арностью»*, заголовка отношения, кортежа, соответствующего этому заголовку, тела отношения, значения отношения и переменной отношения является мощность заголовка отношения. Например, степень отношения СЛУЖАЩИЕ равна четырем, т. е. оно является 4-арным (*кватернарным*).

При приведенных определениях разумно считать *схемой реляционной базы данных* набор пар <имя_VARr, Hr>, включающий имена и заголовки всех переменных отношения, которые определены в базе данных. *Реляционная база данных* – это набор пар <VARr, Hr> (конечно, каждая переменная отношения в любой момент времени содержит некоторое значение-отношение, в частности, пустое).

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

3.2.4. Первичный ключ и интуитивная интерпретация реляционных понятий

По определению, *первичным ключом* переменной отношения является такое подмножество[5)]

S множества атрибутов ее заголовка, что в любое время значение первичного ключа (составное, если в состав первичного ключа входит более одного атрибута) в любом кортеже тела отношения отличается от значения первичного ключа в любом другом кортеже тела этого отношения, а никакое собственное подмножество[6] S этим свойством не обладает. В следующем разделе мы покажем, что существование первичного ключа у любого значения отношения является следствием одного из фундаментальных свойств отношений, а именно того свойства, что тело отношения является множеством кортежей.

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

Конечно, это достаточно грубая терминология, поскольку у обычных таблиц и строки, и столбцы упорядочены, тогда как атрибуты и кортежи отношений являются элементами неупорядоченных множеств. Тем не менее, когда мы перейдем к рассмотрению практических вопросов организации реляционных баз данных и средств управления, то будем использовать эту «житейскую» терминологию. Подобной терминологии придерживаются в большинстве коммерческих реляционных СУБД. Иногда также используются термины *файл* как аналог таблицы, *запись* как аналог строки и *поле* как аналог столбца. Напомню, что этой терминологией мы пользовались в лекции 1.

---

5 В вырожденном случае, когда заголовок переменной отношения является пустым множеством, первичный ключ этой переменной отношения состоит из пустого подмножества заголовка. Легко проверить, что этот случай не противоречит общему определению.

6 Напомним, что S′ является собственным подмножеством множества S в том и только в том случае, когда S′ входит в S, но не совпадает с S (это обозначается как $S' \subset S$).

Лекция 4. Базисные средства манипулирования реляционными данными: реляционная алгебра Кодда

4.1. Введение

В предыдущей лекции упоминались три составляющих реляционной модели данных. Две из них – *структурную* и *целостную* части – мы рассмотрели более или менее подробно, а *манипуляционной части* реляционной модели данных посвящается эта и следующие две лекции. Мы уделяем данной теме такое большое внимание, поскольку понимание формальных механизмов манипулирования реляционными данными исключительно важно для понимания технологии баз данных вообще. В этой лекции после небольшого введения будет рассмотрен вариант реляционной алгебры Кодда [2.1], предложенный Кристофером Дейтом около 15 лет тому назад. Мне этот вариант алгебры кажется наиболее понятным, хотя предлагаемый набор операций несколько избыточен. В следующей лекции мы обсудим новый «минимальный» вариант алгебры, предложенный Дейтом и Дарвеном в конце 1990-х гг. Возможно, новая алгебра не очень практична, но зато красива и элегантна. После этого в лекции 6 мы перейдем к реляционному исчислению, достаточно подробно рассмотрим один

из вариантов реляционного исчисления кортежей и кратко обсудим особенности исчисления доменов.

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

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

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

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

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

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

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

Для экономии времени и места мы не будем вводить какие-либо строгие синтаксические конструкции, а в основном ограничимся рассмотрением материала на содержательном уровне.

4.2. Обзор реляционной алгебры Кодда

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

Существует много подходов к определению реляционной алгебры, которые различаются наборами операций и способами их интерпретации, но, в принципе, являются более или менее равносильными. В данном разделе мы опишем немного расширенный начальный вариант алгебры, который был предложен Коддом (будем называть ее «алгеброй Кодда»). В этом варианте набор основных алгебраических операций состоит из восьми операций, которые делятся на два класса – теоретико-множественные операции и специальные реляционные операции. В состав теоретико-множественных операций входят операции:

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

Специальные реляционные операции включают:

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

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

4.2.1. Общая интерпретация реляционных операций

Если не вдаваться в некоторые тонкости, которые мы рассмотрим в следующих разделах, то почти для всех операций предложенного выше набора имеется очевидная и простая интерпретация.

- При выполнении операции *объединения* (`UNION`) двух отношений с одинаковыми заголовками производится отношение, включающее все кортежи, которые входят хотя бы в одно из отношений-операндов.
- Операция *пересечения* (`INTERSECT`) двух отношений с одинаковыми заголовками производит отношение, включающее все кортежи, которые входят в оба отношения-операнда.
- Отношение, являющееся *разностью* (`MINUS`) двух отношений с одинаковыми заголовками, включает все кортежи, входящие в отношение-первый операнд, такие,

что ни один из них не входит в отношение, которое является вторым операндом.

- При выполнении *декартова произведения* (`TIMES`) двух отношений, пересечение заголовков которых пусто, производится отношение, кортежи которого производятся путем объединения кортежей первого и второго операндов.
- Результатом *ограничения* (`WHERE`) отношения по некоторому условию является отношение, включающее кортежи отношения-операнда, удовлетворяющее этому условию.
- При выполнении *проекции* (`PROJECT`) отношения на заданное подмножество множества его атрибутов производится отношение, кортежи которого являются соответствующими подмножествами кортежей отношения-операнда.
- При *соединении* (`JOIN`) двух отношений по некоторому условию образуется результирующее отношение, кортежи которого производятся путем объединения кортежей первого и второго отношений и удовлетворяют этому условию.
- У операции *реляционного деления* (`DIVIDE BY`) два операнда – бинарное и унарное отношения. 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 *переименования* (`RENAME`) производит отношение, тело которого совпадает с телом операнда, но имена атрибутов изменены.
- Операция *присваивания* (`:=`) позволяет сохранить результат вычисления реляционного выражения в существующем отношении БД.

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

```
RENAME ≥ WHERE = PROJECT ≥ TIMES = JOIN = INTERSECT = DIVIDE BY ≥
UNION = MINUS
```

В другой форме приоритеты операций показаны на рис. 4.1. Вычисление выражения производится слева направо с учетом приоритетов операций и скобок.

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

Рис. 4.1. Таблица приоритетов операций традиционной реляционной алгебры

4.2.2. Замкнутость реляционной алгебры и операция переименования

Как мы отмечали в предыдущей лекции, каждое значение-отношение характеризуется заголовком (или схемой) и телом (или множеством кортежей). Поэтому, если нам действительно нужна алгебра, операции которой замкнуты относительно понятия отношения, то каждая операция должна производить отношение в полном смысле, т. е. оно должно обладать и телом, и заголовком. Только в этом случае можно будет строить вложенные выражения.

Заголовок отношения представляет собой множество пар <имя-атрибута, имя-домена>. Если посмотреть на общий обзор реляционных операций, приведенный в предыдущем подразделе, то видно, что домены атрибутов результирующего отношения однозначно определяются доменами отношений-операндов. Однако с именами атрибутов результата не всегда все так просто.

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

Аналогичные проблемы могут возникать и в случаях других двуместных операций. Для разрешения проблем в число операций реляционной алгебры вводится операция переименования. Ее следует применять в том случае, когда возникает конфликт именования атрибутов в отношениях-операндах одной реляционной операции. Тогда к одному из операндов сначала применяется операция переименования, а затем основная операция выполняется уже без всяких проблем. Более строго мы определим операцию переименования в следующей лекции, а пока лишь заметим, что результатом этой операции является отношение, совпадающее во всем с отношением-операндом, кроме того, что имя указанного атрибута изменено на заданное имя.

В дальнейшем изложении мы будем предполагать применение операции переименования во всех конфликтных ситуациях. Заметим, кстати, что невозможность применения некоторых

операций к произвольным парам значений отношений без предварительного переименования атрибутов отношений операндов означает, что «алгебра» Кодда не является алгеброй отношений в математическом смысле. Описываемая в следующей главе Алгебра А такими недостатками не обладает: результатом применения любой операции к любым отношениям является некоторое отношение.

4.3. Особенности теоретико-множественных операций реляционной алгебры

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

4.3.1. Операции объединения, пересечения, взятия разности. Совместимость по объединению

Начнем с операции объединения отношений (все, что будет сказано по поводу объединения, верно и для операций пересечения и взятия разности отношений). Смысл операции объединения в реляционной алгебре в целом остается теоретико-множественным. Еще раз напомним (см. рис. 3.4), что в теории множеств:

- результатом объединения двух множеств A{*a*} и B{*b*} является такое множество C{*c*}, что для каждого *c* либо существует такой элемент *a*, принадлежащий множеству A, что *c=a*, либо существует такой элемент *b*, принадлежащий множеству B, что *c=b*;
- пересечением множеств A и B является такое множество C{*c*}, что для любого *c* существуют такие элементы *a*, принадлежащий множеству A, и *b*, принадлежащий множеству B, что *c=a=b*;
- разностью множеств A и B является такое множество C{*c*}, что для любого *c* существует такой элемент *a*, принадлежащий множеству A, что *c=a*, и не существует такой элемент *b*, принадлежащий B, что *c=b*.

Но если в теории множеств операция объединения осмысленна для любых двух множеств-операндов, то в случае реляционной алгебры результатом операции объединения должно являться отношение. Если в реляционной алгебре допустить возможность теоретико-множественного объединения двух произвольных отношений (с разными заголовками), то, конечно, результатом операции будет множество, но множество разнотипных кортежей, т. е. не отношение. Если исходить из требования замкнутости реляционной алгебры относительно понятия отношения, то такая операция объединения является бессмысленной.

Эти соображения подводят к понятию *совместимости отношений по объединению*: два отношения совместимы по объединению в том и только в том случае, когда обладают одинаковыми заголовками. В развернутой форме это означает, что в заголовках обоих отношений содержится один и тот же набор имен атрибутов, и одноименные атрибуты определены на одном и том же домене (эта развернутая формулировка, вообще говоря, является излишней, но она пригодится нам в следующем абзаце).

Если два отношения совместимы по объединению, то при обычном выполнении над ними операций объединения, пересечения и взятия разности результатом операции является

отношение с корректно определенным заголовком, совпадающим с заголовком каждого из отношений-операндов. Напомним, что если два отношения «почти» совместимы по объединению, т. е. совместимы во всем, кроме имен атрибутов, то до выполнения операции типа объединения эти отношения можно сделать полностью совместимыми по объединению путем применения операции переименования.

Для иллюстрации операций объединения, пересечения и взятия разности предположим, что в базе данных имеются два отношения СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 и СЛУЖАЩИЕ_В_ПРОЕКТЕ_2 с одинаковыми схемами {СЛУ_НОМЕР, СЛУ_ИМЯ, СЛУ_ЗАРП, СЛУ_ОТД_НОМЕР} (имена доменов опущены по причине очевидности). Каждое из отношений содержит данные о служащих, участвующих в соответствующем проекте. На рис. 4.2 показано примерное наполнение каждого из двух отношений (некоторые служащие участвуют в обоих проектах).

СЛУЖАЩИЕ_В_ПРОЕКТЕ_1

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

СЛУЖАЩИЕ_В_ПРОЕКТЕ_2

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	СЛУ_ОТД_НОМЕР
2934	Иванов	22000.00	310
2935	Петров	30000.00	310
2939	Сидоренко	18000.00	313
2940	Федоренко	20000.00	310
2941	Иваненко	22000.00	315

Рис. 4.2. Примерное наполнение отношений СЛУЖАЩИЕ _В_ПРОЕКТЕ_1 и СЛУЖАЩИЕ _В_ПРОЕКТЕ_2

Тогда выполнение операции СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 UNION СЛУЖАЩИЕ_В_ПРОЕКТЕ_2 позволит получить информацию обо всех служащих, участвующих в обоих проектах. Выполнение операции СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 INTERSECT СЛУЖАЩИЕ_В_ПРОЕКТЕ_2 позволит получить данные о служащих, которые одновременно участвуют в двух проектах. Наконец, операция СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 MINUS СЛУЖАЩИЕ_В_ПРОЕКТЕ_2 выработает отношение, содержащее кортежи служащих, которые участвуют только в первом проекте. Результаты этих операций показаны на рис. 4.3.

СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 UNION **СЛУЖАЩИЕ_В_ПРОЕКТЕ_2**

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	СЛУ_ОТД_НОМЕР
2934	Иванов	22000.00	310
2935	Петров	30000.00	310
2939	Сидоренко	18000.00	313
2940	Федоренко	20000.00	310
2941	Иваненко	22000.00	315
2936	Сидоров	18000.00	313
2937	Федоров	20000.00	310
2938	Иванова	22000.00	315

СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 INTERSECT **СЛУЖАЩИЕ_В_ПРОЕКТЕ_2**

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	СЛУ_ОТД_НОМЕР
2934	Иванов	22000.00	310
2935	Петров	30000.00	310

СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 MINUS **СЛУЖАЩИЕ_В_ПРОЕКТЕ_2**

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	СЛУ_ОТД_НОМЕР
2936	Сидоров	18000.00	313
2937	Федоров	20000.00	310
2938	Иванова	22000.00	315

Рис. 4.3. Результаты выполнения операций UNION, INTERSECT и MINUS

Заметим, что включение в состав операций реляционной алгебры трех операций объединения, пересечения и взятия разности является, очевидно, избыточным, поскольку, например, операция пересечения выражается через операцию взятия разности[14)]. Тем не менее Кодд в свое время решил включить все три операции, исходя из интуитивных потребностей далекого от математики потенциального пользователя системы реляционных БД.

4.3.2. Операция расширенного декартова произведения и совместимость отношений относительно этой операции

Другие проблемы связаны с операцией взятия декартова произведения двух отношений. В теории множеств декартово произведение может быть получено для любых двух множеств, и элементами результирующего множества являются пары, составленные из элементов первого и второго множеств. Если говорить более точно, декартовым произведением множеств A{*a*} и B{*b*} является такое множество пар C{<c_1, c_2>}, что для каждого элемента <c_1, c_2> множества C существуют такой элемент *a* множества A, что c_1=*a*, и такой элемент *b* множества B, что c_2=*b*.

Поскольку отношения являются множествами, для любых двух отношений возможно получение прямого произведения. Но результат не будет отношением! Элементами результата будут не кортежи, а пары кортежей.

Поэтому в реляционной алгебре используется специализированная форма операции взятия декартова произведения – расширенное декартово произведение отношений. При взятии

расширенного декартова произведения двух отношений элементом результирующего отношения является кортеж, который представляет собой объединение одного кортежа первого отношения и одного кортежа второго отношения.

Приведем более точное определение *операции расширенного декартова произведения*. Пусть имеются два отношения R1{a_1, a_2, …, a_n} и R2{b_1, b_2, …, b_m}. Тогда результатом операции R1 TIMES R2 является отношение R{a_1, a_2, …, a_n, b_1, b_2, …, b_m}, тело которого является множеством кортежей вида {r_{a1}, r_{a2}, …, r_{an}, r_{b1}, r_{b2}, …, r_{bm}} таких, что {r_{a1}, r_{a2}, …, r_{an}} входит в тело R1, а {r_{b1}, r_{b2}, …, r_{bm}} входит в тело R2.

Но теперь возникает вторая проблема – как получить корректно сформированный заголовок отношения-результата? Поскольку схема результирующего отношения является объединением схем отношений-операндов, то очевидной проблемой может быть именование атрибутов результирующего отношения, если отношения-операнды обладают одноименными атрибутами.

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

Для наглядности предположим, что в придачу к введенным ранее отношениям СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 и СЛУЖАЩИЕ_В_ПРОЕКТЕ_2 в базе данных содержится еще и отношение ПРОЕКТЫ со схемой {ПРОЕКТ_НАЗВ, ПРОЕКТ_РУК} (имена доменов снова опущены) и телом, показанным на рис. 4.4. На этом же рисунке показан результат операции СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 TIMES ПРОЕКТЫ.

ПРОЕКТЫ

ПРОЕКТ_НАЗВ	ПРОЕКТ_РУК
ПРОЕКТ 1	Иванов
ПРОЕКТ 2	Иваненко

СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 TIMES ПРОЕКТЫ

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	СЛУ_ОТД_НОМЕР	ПРОЕКТ_НАЗВ	ПРОЕКТ_РУК
2934	Иванов	22000.00	310	ПРОЕКТ 1	Иванов
2935	Петров	30000.00	310	ПРОЕКТ 1	Иванов
2936	Сидоров	18000.00	313	ПРОЕКТ 1	Иванов
2937	Федоров	20000.00	310	ПРОЕКТ 1	Иванов
2938	Иванова	22000.00	315	ПРОЕКТ 1	Иванов
2934	Иванов	22000.00	310	ПРОЕКТ 2	Иваненко
2935	Петров	30000.00	310	ПРОЕКТ 2	Иваненко
2936	Сидоров	18000.00	313	ПРОЕКТ 2	Иваненко
2937	Федоров	20000.00	310	ПРОЕКТ 2	Иваненко
2938	Иванова	22000.00	315	ПРОЕКТ 2	Иваненко

Рис. 4.4. Отношение ПРОЕКТЫ и результат операции СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 TIMES ПРОЕКТЫ

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

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

---

14 Легко убедиться, что `A INTERSECT B = A MINUS (A MINUS B) = B MINUS (B MINUS A)`.

Лекция 5. Базисные средства манипулирования реляционными данными: алгебра A Дейта и Дарвена

5.1. Введение

В этой лекции мы обсудим новый «минимальный» вариант алгебры, предложенный несколько лет тому назад Дейтом и Дарвеном [1.5]. Как уже отмечалось в предыдущей лекции, возможно, новая алгебра не очень практична, но зато красива и элегантна.

Обсуждавшаяся в предыдущей лекции алгебра Кодда в большей степени базируется на теории множеств. 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 предложенной Крисом Дейтом и Хью Дарвеном Алгебры A 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 и т. д. Путем комбинирования базовых операций выражаются операции переименования атрибутов, соединения общего вида, взятия разности отношений. Алгебра A позволяет лучше осознать логические основы реляционной модели,

хотя, безусловно, является в меньшей степени ориентированной на практическое применение, чем алгебра Кодда[16]. Даже сами авторы Алгебры А, Дейт и Дарвен, в своем учебном языке *Tutorial D* [1.5] используют не Алгебру А напрямую, а некоторое ее надмножество, в большей степени напоминающее алгебру Кодда.

5.2. Базовые операции Алгебры А

Материал этой лекции излагается на несколько более формальном уровне, чем в предыдущих лекциях. Используемые понятия определяются, по существу, так же, как и в лекции 3, но для удобства и обеспечения точности изложения мы повторим определения.

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

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

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

Определим несколько основных операций (как будет показано далее, некоторые из них избыточны). Каждое из последующих определений состоит из: формальной спецификации ограничений (если они имеются), применимых к операндам соответствующей операции; формальной спецификации заголовка результата этой операции; формальной спецификации тела этого результата и неформального обсуждения формальных спецификаций.

Во всех формальных спецификациях exists обозначает *квантор существования*; exists t*r* означает «существует такой t*r*, что». Символ «$\in$» означает принадлежность одного множества другому; t*r*$\in$B*r* означает, что элемент t*r* принадлежит множеству B*r*. Выражение t*r*$\notin$B*r* означает, что элемент t*r* не принадлежит множеству B*r*. Операции minus и union являются традиционными теоретико-множественными операциями взятия разности и объединения множеств.

Поскольку некоторые базовые операции Алгебры А имеют названия обычных логических операций, чтобы избежать путаницы, имена реляционных операций берутся в угловые скобки: <NOT>, <AND>, <OR> и т. д. В исходный базовый набор операций входят операции реляционного дополнения <NOT>, удаления атрибута <REMOVE>, переименования атрибута <RENAME>, реляционной конъюнкции <AND> и реляционной дизъюнкции <OR>.

5.2.1. Операция реляционного дополнения

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

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

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

Видимо, следует пояснить, почему реляционный аналог операции логического отрицания называется здесь операцией реляционного дополнения. Во-первых, термин «дополнение» полностью соответствует сути операции <NOT>: тело результата операции <NOT> *r* является дополнением B*r* до полного множества кортежей, соответствующих H*r*. Во-вторых, это не противоречит природе булевской операции NOT: у булевского типа имеются всего два значения – true и false, и NOT true = false, а NOT false = true. (Кстати, обратите внимание, что операцию NOT в трехзначной логике (см. лекцию 1) уже нельзя считать операцией дополнения.)

Чтобы привести пример использования операции <NOT>, предположим, что в состав домена ДОПУСТИМЫЕ_НОМЕРА_ПРОЕКТОВ, на котором определен атрибут ПРО_НОМ отношения НОМЕРА_ПРОЕКТОВ с рис. 5.1 слева, входит всего пять значений {1, 2, 3, 4, 5}. Тогда результат операции <NOT> НОМЕРА_ПРОЕКТОВ будет таким, как показано на рис. 5.1 справа.

5.2.2. Операция удаления атрибута

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

НОМЕРА_ПРОЕКТОВ

ПРО_НОМ
1
2

<NOT> НОМЕРА_ПРОЕКТОВ

ПРО_НОМ
3
4
5

Рис. 5.1. Результат операции <NOT> НОМЕРА_ПРОЕКТОВ

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

Операция <REMOVE> производит отношение *s*, формируемое путем удаления указанного

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

Примером операции REMOVE (конечно же, очень похожим на пример использования операции PROJECT из предыдущей лекции) является СЛУЖАЩИЕ REMOVE ПРО_НОМ (получить данные о служащих, участвующих в проектах). Результат этой операции над отношением СЛУЖАЩИЕ, тело которого приведено в верхней части рис. 5.2, показан на рис. 5.2 внизу.

СЛУЖАЩИЕ

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

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

Рис. 5.2. Результат операции СЛУЖАЩИЕ REMOVE ПРО_НОМ

5.2.3. Операция переименования

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

- H*s* = (H*r* minus {<A, T>}) union {<B, T>}, т. е. в схеме результата B заменяет A;
- B*s* = {t*s* : exists t*r* exists v (t*r* $\in$ B*r* and v $\in$ T and <A, T, v>

$\in$ tr and ts = (tr minus {<A, T, v>}) union {<B, T, v>})}, т. е. в кортежах тела результата имя значений атрибута A меняется на B.

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

По причине очевидности пример использования этой операции мы приводить не будем.

5.2.4. Операция реляционной конъюнкции

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

- H*s* = Hr_1 union Hr_2, т. е. заголовок результата получается путем объединения заголовков отношений-операндов, как в операциях TIMES и JOIN из предыдущей лекции;
- B*s* = { *ts* : exists tr_1 exists tr_2 ((t$r_1$$\in$B$r_1$ and t$r_2$$\in$B$r_2$) and *ts* = t$r_1$ union tr_2)}; обратите внимание на то, что кортеж результата определяется как *объединение кортежей операндов*; поэтому:
  - если схемы отношений-операндов имеют непустое пересечение, то операция <AND> работает как естественное соединение;
  - если пересечение схем операндов пусто, то <AND> работает как расширенное декартово произведение;
  - если схемы отношений полностью совпадают, то результатом операции является пересечение двух отношений-операндов.

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

Для иллюстрации воспользуемся примерными отношениями, показанными на рис. 5.3, которые мы уже использовали в примерах предыдущей лекции.

СЛУЖАЩИЕ_В_ПРОЕКТЕ_1

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

СЛУЖАЩИЕ_В_ПРОЕКТЕ_2

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	СЛУ_ОТД_НОМЕР
2934	Иванов	22000.00	310
2935	Петров	30000.00	310
2939	Сидоренко	18000.00	313
2940	Федоренко	20000.00	310
2941	Иваненко	22000.00	315

ПРОЕКТЫ

ПРО_НОМ	ПРОЕКТ_РУК
1	Иванов
2	Иваненко

СЛУЖАЩИЕ

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

Рис. 5.3. Примерные отношения для иллюстрации операции <AND>

На рис. 5.4(a) у отношений СЛУЖАЩИЕ и ПРОЕКТЫ имеется общий атрибут ПРО_НОМ. Поэтому операция <AND> работает как операция естественного соединения. На рис. 5.4(b) пересечение заголовков отношений СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 и ПРОЕКТЫ пусто, и поэтому в результате реляционной конъюнкции производится расширенное декартово произведение этих отношений. Наконец, на рис. 5.4(c) схемы отношений СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 и СЛУЖАЩИЕ_В_ПРОЕКТЕ_2 совпадают, и телом операции <AND> является пересечение тел отношений-операндов.

(a) Результат операции **СЛУЖАЩИЕ** <AND> **ПРОЕКТЫ**

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

(b) Результат операции **СЛУЖАЩИЕ_В_ПРОЕКТЕ_1** <AND> **ПРОЕКТЫ**

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	СЛУ_ОТД_НОМЕР	ПРО_НОМ	ПРОЕКТ_РУК
2934	Иванов	22000.00	310	1	Иванов
2935	Петров	30000.00	310	1	Иванов
2936	Сидоров	18000.00	313	1	Иванов
2937	Федоров	20000.00	310	1	Иванов
2938	Иванова	22000.00	315	1	Иванов
2934	Иванов	22000.00	310	2	Иваненко
2935	Петров	30000.00	310	2	Иваненко
2936	Сидоров	18000.00	313	2	Иваненко
2937	Федоров	20000.00	310	2	Иваненко
2938	Иванова	22000.00	315	2	Иваненко

(c) Результат операции **СЛУЖАЩИЕ_В_ПРОЕКТЕ_1** <AND> **СЛУЖАЩИЕ_В_ПРОЕКТЕ_2**

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	СЛУ_ОТД_НОМЕР
2934	Иванов	22000.00	310
2935	Петров	30000.00	310

Рис. 5.4. Иллюстрации операции реляционной конъюнкции

5.2.5. Операция реляционной дизъюнкции

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

- Hs = Hr_1 union Hr_2 (из схемы результата удаляются атрибуты-дубликаты);
- Bs = { ts : exists tr_1 exists tr_2 (($tr_1 \in$ Br_1 or $tr_2 \in$ Br_2) and ts = tr_1 union tr_2)}; очевидно, что при этом:
  - если у операндов нет общих атрибутов, то в тело результирующего отношения входят все такие кортежи ts, которые являются объединением кортежей tr_1 и

tr_2, соответствующих заголовкам отношений-операндов, и хотя бы один из этих кортежей принадлежит телу одного из операндов;

- если у операндов имеются общие атрибуты, то в тело результирующего отношения входят все такие кортежи ts, которые являются объединением кортежей tr_1 и tr_2, соответствующих заголовкам отношений-операндов, если хотя бы один из этих кортежей принадлежит телу одного из операндов, и значения общих атрибутов tr_1 и tr_2 совпадают;
- если же схемы отношений-операндов совпадают, то тело отношения-результата является объединением тел операндов.

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

Предположим, у нас имеются отношения ПРОЕКТЫ_1 {ПРОЕКТ_НАЗВ, ПРОЕКТ_РУК} и НОМЕРА_ПРОЕКТОВ {ПРО_НОМ} (рис. 5.5). Предположим также, что домен атрибута ПРОЕКТ_НАЗВ включает значения ПРОЕКТ_1, ПРОЕКТ_2, ПРОЕКТ_3, домен атрибута ПРОЕКТ_РУК ограничен значениями Иванов, Иваненко, а доменом атрибута ПРО_НОМ является множество {1, 2, 3}. Результат операции ПРОЕКТЫ <OR> НОМЕРА_ПРОЕКТОВ показан на рис. 5.5.

Как показано на рис. 5.5, операция <OR> при наличии операндов с несовпадающими схемами производит результат, гораздо более мощный, чем результат операции взятия расширенного декартова произведения из лекции 4, и еще менее осмысленный с практической точки зрения.

Для иллюстрации операции <OR> над операндами, схемы которых имеют непустое пересечение, воспользуемся отношением ПРОЕКТЫ_2 {ПРО_НОМ, ПРОЕКТ_РУК} (рис. 5.6) и унарным отношением НОМЕРА_ПРОЕКТОВ, схема и тело которого показаны на рис. 5.5. Будем предполагать, что множества значений доменов атрибутов такие же, как в предыдущем примере. Результат операции ПРОЕКТЫ_2 <OR> НОМЕРА_ПРОЕКТОВ показан на рис. 5.6.

Как уже отмечалось, при совпадении схем отношений-операндов результатом выполнения над ними операции <OR> является объединение отношений. Это непосредственно следует из спецификации операции. Если этот факт кажется неочевидным, еще раз внимательно посмотрите на спецификацию. Иллюстрирующий пример мы приводить не будем.

ПРОЕКТЫ_1

ПРОЕКТ_НАЗВ	ПРОЕКТ_РУК
ПРОЕКТ 1	Иванов
ПРОЕКТ 2	Иваненко

НОМЕРА_ПРОЕКТОВ

ПРО_НОМ
1
2

Результат операции **ПРОЕКТЫ** <OR> **НОМЕРА_ПРОЕКТОВ**

ПРОЕКТ_НАЗВ	ПРОЕКТ_РУК	ПРО_НОМ
ПРОЕКТ 1	Иванов	1
ПРОЕКТ 2	Иванов	1
ПРОЕКТ 3	Иванов	1
ПРОЕКТ 1	Иваненко	1
ПРОЕКТ 2	Иваненко	1
ПРОЕКТ 3	Иваненко	1
ПРОЕКТ 1	Иванов	2
ПРОЕКТ 2	Иванов	2
ПРОЕКТ 3	Иванов	2
ПРОЕКТ 1	Иваненко	2
ПРОЕКТ 2	Иваненко	2
ПРОЕКТ 3	Иваненко	2
ПРОЕКТ 1	Иванов	3
ПРОЕКТ 2	Иваненко	3

Рис. 5.5. Результат операции <OR> над операндами без общих атрибутов

ПРОЕКТЫ_2

ПРО_НОМ	ПРОЕКТ_РУК
1	Иванов
2	Иваненко

ПРОЕКТЫ_2 <OR> **НОМЕРА_ПРОЕКТОВ**

ПРО_НОМ	ПРОЕКТ_РУК
1	Иванов
2	Иваненко
2	Иванов
1	Иваненко

Рис. 5.6. Результат операции <OR> над операндами, схемы которых частично пересекаются

---

16 Нельзя не упомянуть еще и о том, что «алгебра» Кодда в действительности не является алгеброй отношений в математическом смысле, поскольку ее операции применимы не ко всем отношениям. В отличие от этого Алгебра А – это «настоящая» алгебра, в которой отсутствуют какие-либо ограничения на операнды операций.

Лекция 6. Базисные средства манипулирования реляционными данными: реляционное исчисление

6.1. Введение

Эта лекция завершает цикл лекций, посвященных манипуляционному аспекту реляционной модели данных. Материал лекции интересен и сам по себе, поскольку демонстрирует, насколько аппарат математической логики упрощает формулировку запросов к базам данных.

Предположим, что мы работаем с базой данных, которая состоит из отношений СЛУЖАЩИЕ {СЛУ_НОМ, СЛУ_ИМЯ, СЛУ_ЗАРП, ПРО_НОМ} и ПРОЕКТЫ {ПРО_НОМ, ПРОЕКТ_РУК, ПРО_ЗАРП} (в отношении ПРОЕКТЫ атрибут ПРОЕКТ_РУК содержит имена служащих, являющихся руководителями проектов, а атрибут ПРО_ЗАРП – среднее значение зарплаты, получаемой участниками проекта), и хотим узнать имена и номера служащих, которые являются руководителями проектов со средней заработной платой, превышающей 18000 руб.

Если бы для формулировки такого запроса использовалась реляционная алгебра, то мы получили бы, например, следующее алгебраическое выражение:

```
(СЛУЖАЩИЕ JOIN ПРОЕКТЫ WHERE (СЛУ_ИМЯ = ПРОЕКТ_РУК AND
ПРО_ЗАРП > 18000.00)) PROJECT (СЛУ_ИМЯ, СЛУ_НОМ)
```

Это выражение можно было бы прочитать, например, следующим образом:

- выполнить эквисоединение отношений СЛУЖАЩИЕ и ПРОЕКТЫ по условию СЛУ_ИМЯ = ПРОЕКТ_РУК;
- ограничить полученное отношение по условию ПРО_ЗАРП > 18000.00;
- спроецировать результат предыдущей операции на атрибут СЛУ_ИМЯ, СЛУ_НОМ.

Мы четко сформулировали последовательность шагов выполнения запроса, каждый из которых соответствует одной реляционной операции.

Если же сформулировать тот же запрос с использованием реляционного исчисления, которому посвящается эта лекция, то мы получили бы два определения переменных:

```
RANGE СЛУЖАЩИЙ IS СЛУЖАЩИЕ
```

и

```
RANGE ПРОЕКТ IS ПРОЕКТЫ
```

и выражение

```
СЛУЖАЩИЙ.СЛУ_ИМЯ, СЛУЖАЩИЙ.СЛУ_НОМ WHERE EXISTS (СЛУЖАЩИЙ.СЛУ_ИМЯ
= ПРОЕКТ.ПРОЕКТ_РУК AND ПРОЕКТ.ПРО_ЗАРП > 18000.00).
```

Это выражение можно было бы прочитать, например, следующим образом: выдать значения СЛУ_ИМЯ и СЛУ_НОМ для каждого кортежа служащих такого, что существует кортеж проектов со значением ПРОЕКТ_РУК, совпадающим со значением СЛУ_НОМ этого кортежа

служащих, и значением ПРО_ЗАРП, большим 18000.00.

Во второй формулировке мы указали лишь 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 всего лишь описывает свойства желаемого результата. Как мы указывали в начале лекции 4, на самом деле эти два механизма эквивалентны, и существуют не слишком сложные правила преобразования одного формализма в другой.

Реляционное исчисление является прикладной ветвью формального механизма исчисления предикатов первого порядка[25)]. В основе исчисления лежит понятие переменной с определенной для нее областью допустимых значений и понятие правильно построенной формулы, опирающейся на переменные, предикаты и кванторы.

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

Как и в лекциях, посвященных реляционной алгебре, в этой лекции нам не удастся избежать использования некоторого конкретного синтаксиса, который мы тем не менее формально определять не будем. Те или иные синтаксические конструкции будут вводиться по мере необходимости. В совокупности используемый синтаксис близок, но не полностью совпадает с синтаксисом языка баз данных QUEL, который долгое время являлся основным языком известной реляционной СУБД Ingres.

6.2. Исчисление кортежей

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

```
RANGE СЛУЖАЩИЙ IS СЛУЖАЩИЕ
```

Как уже говорилось, из этого определения следует, что в любой момент времени переменная СЛУЖАЩИЙ представляет некоторый кортеж отношения СЛУЖАЩИЕ. При использовании кортежных переменных в формулах можно ссылаться на значение атрибута переменной (это аналогично тому, как, например, при программировании на языке С можно сослаться на значение поля структурной переменной). Например, для того, чтобы сослаться на значение атрибута СЛУ_ИМЯ переменной СЛУЖАЩИЙ, нужно употребить конструкцию СЛУЖАЩИЙ.СЛУ_ИМЯ.

6.2.1. Правильно построенные формулы

Правильно построенная формула (Well-Formed Formula, WFF) служит для выражения условий, накладываемых на кортежные переменные.

Простые условия

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

```
СЛУЖАЩИЙ.СЛУ_НОМ = 2934
```
и

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

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

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

Более сложные варианты WFF строятся с помощью логических связок `NOT`, `AND`, `OR` и `IF ... THEN`[26)] с учетом обычных приоритетов операций (`NOT > AND > OR`) и возможности расстановки скобок. Так, если `form` – WFF, а `comp` – простое сравнение, то `NOT form`, `comp AND form`, `comp OR form` и `IF comp THEN form` являются WFF.

Для примеров воспользуемся отношениями `СЛУЖАЩИЕ`, `ПРОЕКТЫ` и `НОМЕРА_ПРОЕКТОВ` из предыдущей лекции (см. рис. 6.1).

СЛУЖАЩИЕ

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

ПРОЕКТЫ

ПРО_НОМ	ПРОЕКТ_РУК
1	Иванов
2	Иваненко

НОМЕРА_ПРОЕКТОВ

ПРО_НОМ
1
2

Рис. 6.1. Примерные значения отношений СЛУЖАЩИЕ, ПРОЕКТЫ и НОМЕРА_ПРОЕКТОВ

Правильно построенной является следующая формула:

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

Эта формула будет принимать значение `true` для следующих значений кортежной переменной СЛУЖАЩИЙ:

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

Конечно, нужно представлять себе какой-нибудь способ реализации системы, которая сможет по заданной WFF при существующем состоянии базы данных произвести такой

результат. И таким очевидным способом является следующий: в некотором порядке просмотреть область определения переменной и к каждому очередному кортежу применить условие. Результатом будет то множество кортежей, для которых при вычислении условия производится значение `true`. Очевидно, что результат эквивалентен выполнению алгебраической операции `СЛУЖАЩИЕ WHERE (NOT (СЛУЖАЩИЙ.СЛУ_ИМЯ = 'Иванов') OR (СЛУЖАЩИЙ.СЛУ_ЗАРП >= 22400.00 AND СЛУЖАЩИЙ.ПРО_НОМ = 1)` над отношением, тело которого представляет собой область определения кортежной переменной.

Пусть имеется следующее определение кортежной переменной `ПРОЕКТ`:

```
RANGE ПРОЕКТ IS ПРОЕКТЫ
```

Вот еще пример правильно построенной формулы:

```
СЛУЖАЩИЙ.СЛУ_ИМЯ = ПРОЕКТ.ПРОЕКТ_РУК
```

Эта формула будет принимать значение `true` для следующих пар значений кортежных переменных `СЛУЖАЩИЙ` и `ПРОЕКТ`:

СЛУЖАЩИЕ				**ПРОЕКТЫ**	
СЛУ_НОМЕР	**СЛУ_ИМЯ**	**СЛУ_ЗАРП**	**ПРО_НОМ**	**ПРО_НОМ**	**ПРОЕКТ_ РУК**
2934	Иванов	22400.00	1	1	Иванов
2941	Иваненко	22000.00	2	2	Иваненко
2934	Иванов	22400.00	2	1	Иванов

Очевидный способ реализации системы, которая по заданной WFF при существующем состоянии базы данных производит такой результат, заключается в следующем. В некотором порядке просматривать область определения (например) переменной `СЛУЖАЩИЙ`. Для каждого текущего кортежа из области определения переменной `СЛУЖАЩИЙ` просматривать область определения переменной `ПРОЕКТ`. Оставлять в области истинности те пары кортежей, для которых формула принимает значение `true`. Возможен и альтернативный подход: начать просмотр с области определения переменной `ПРОЕКТ`, и для каждого кортежа `ПРОЕКТ` просматривать область определения `СЛУЖАЩИЙ`.

Здесь нужно сделать несколько замечаний. Во-первых, если бы в данном случае формула была тождественно истинной (например, имела вид

```
(СЛУЖАЩИЙ.СЛУ_ИМЯ = СЛУЖАЩИЙ.СЛУ_ИМЯ)
AND (ПРОЕКТ.ПРОЕКТ_РУК = ПРОЕКТ.ПРОЕКТ_РУК))
```

то областью истинности этой формулы являлось бы декартово произведение (в строгом математическом смысле) тел отношений `СЛУЖАЩИЙ` и `ПРОЕКТ`. В реляционном исчислении кортежей, как и в реляционной алгебре, принято иметь дело с операцией расширенного декартова произведения, и поэтому считается, что в подобных случаях областью истинности WFF является отношение, заголовок которого представляет собой объединение заголовков отношений, на телах которых определены кортежные переменные, а кортежи являются объединением соответствующих кортежей из областей определения переменных. При этом

имя атрибута результирующего отношения уточняется именем соответствующей переменной. Поэтому правильнее было бы изображать область истинности формулы

```
СЛУЖАЩИЙ.СЛУ_ИМЯ = ПРОЕКТ.ПРОЕКТ_РУК
```

следующим образом:

СЛУЖАЩИЙ. СЛУ_НОМЕР	СЛУЖАЩИЙ. СЛУ_ИМЯ	СЛУЖАЩИЙ. СЛУ_ЗАРП	СЛУЖАЩИЙ. ПРО_НОМ	ПРОЕКТ. ПРО_НОМ	ПРОЕКТ. ПРОЕКТ_РУК
2934	Иванов	22400.00	1	1	Иванов
2941	Иваненко	22000.00	2	2	Иваненко
2934	Иванов	22400.00	2	1	Иванов

Во-вторых, как видно, показанное результирующее отношение в точности совпадает с результатом алгебраической операции `СЛУЖАЩИЕ JOIN ПРОЕКТЫ WHERE СЛУ_ИМЯ = ПРОЕКТ_РУК` с учетом особенности именования атрибутов результирующего отношения. Наконец, заметим, что описанный выше способ реализации, который приводит к получению области истинности рассмотренной формулы, в действительности является наиболее общим (и зачастую неоптимальным) способом выполнения операций соединения (он называется *методом вложенных циклов – nested loops join*).

Кванторы, свободные и связанные переменные

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

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

Пусть здесь и далее в этом разделе `СЛУ1` и `СЛУ2` представляют собой две кортежные переменные, определенные на отношении `СЛУЖАЩИЕ`. Тогда WFF

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

для текущего кортежа переменной СЛУ1 принимает значение true в том и только в том случае, если во всем отношении СЛУЖАЩИЕ найдется такой кортеж (ассоциированный с переменной СЛУ2), чтобы значение его атрибута СЛУ_ЗАРП удовлетворяло внутреннему условию сравнения. Легко видеть, что эта формула принимает значение true только для тех значений кортежной переменной СЛУ1, которые соответствуют служащим, не получающим минимальную зарплату. Соответствующее множество кортежей показано на рис. 6.2a (для тела отношения СЛУЖАЩИЕ из рис. 6.1).

(a) Область истинности WFF

EXISTS СЛУ2 (СЛУ1.СЛУ_ЗАРП > СЛУ2.СЛУ_ЗАРП)

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

(b) Область истинности WFF

FORALL СЛУ2 (СЛУ1.СЛУ_ЗАРП ≥ СЛУ2.СЛУ_ЗАРП)

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП	ПРО_НОМ
2935	Петров	29600.00	1
2935	Петров	29600.00	2

Рис. 6.2. Примеры правильно построенных формул с кванторами

Правильно построенная формула

```
FORALL СЛУ2 (СЛУ1.СЛУ_ЗАРП ≥ СЛУ2.СЛУ_ЗАРП)
```

для текущего кортежа переменной СЛУ1 принимает значение true в том и только в том случае, если для всех кортежей отношения СЛУЖАЩИЕ (связанных с переменной СЛУ2) значения атрибута СЛУ_ЗАРП удовлетворяют условию сравнения. Снова легко видеть, что формула принимает значение true только для тех значений кортежной переменной СЛУ1, которые соответствуют служащим, получающим максимальную зарплату[27]. Соответствующее множество кортежей показано на рис. 6.2b.

Очевидно, что показанные на рис. 6.2 отношения соответствуют условиям обеих формул. Но как в данном случае можно реализовать систему, которая по заданной формуле производит правильный результат? Наиболее очевидный способ интерпретации обеих обсуждавшихся выше формул следующий. В некотором порядке 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). Заметим, что здесь мы снова получаем два цикла, как и при интерпретации WFF с двумя свободными переменными. Но в данном случае во внешнем цикле обязательно просматривается область определения свободной переменной.

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

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

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

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

Легко заметить, что кванторы можно трактовать как булевские функции (функции, принимающие значения true или false) над множеством значений связанной кортежной переменной. С тем же успехом можно ввести в реляционное исчисление числовые функции над множествами, такие, как MIN (минимальное значение), MAX (максимальное значение), AVG (среднее значение) и т. д.

В этом случае можно было бы написать, например, WFF

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

в области истинности которой содержатся все кортежи отношения СЛУЖАЩИЕ, соответствующие тем служащим, которые получают заработную плату, превышающую минимальную зарплату служащих, участвующих в том же проекте. Понятно, что для получения результирующего отношения можно интерпретировать формулу таким же образом, как в обсуждавшемся выше случае наличия кванторов.

6.2.2. Целевые списки и выражения реляционного исчисления

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

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

- `var.attr`, где `var` – имя свободной переменной соответствующей WFF, а `attr` – имя атрибута отношения, на котором определена переменная `var`;
- `var`, что эквивалентно наличию подсписка $\texttt{var.attr}_1, \texttt{var.attr}_2, \ldots, \texttt{var.attr}_n$, где $\{\texttt{attr}_1, \texttt{attr}_2, \ldots, \texttt{attr}_n\}$ включает имена всех атрибутов определяющего отношения;
- `new_name = var.attr`; `new_name` – новое имя соответствующего атрибута результирующего отношения.

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

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

В качестве простого примера покажем выражение реляционного исчисления кортежей, результат которого совпадает с результатом операции `СЛУЖАЩИЕ DIVIDE BY НОМЕРА_ПРОЕКТОВ` (рис. 4.10 из лекции 4):

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

Конечно, результатом этого выражения является отношение

СЛУ_НОМЕР	СЛУ_ИМЯ	СЛУ_ЗАРП
2934	Иванов	22400.00
2935	Петров	29600.00

---

25 Это совсем не означает, что для понимания этой лекции требуется знание исчисления предикатов. Автор стремился к тому, чтобы материал лекции был в основном самодостаточным.

26 Через IF … THEN здесь обозначается одна и важных логических функций – импликация. По определению, IF a THEN b эквивалентно NOT a OR b. Хотя операция импликации является избыточной, она явно вводится в реляционное исчисление, поскольку часто требуется на практике для выражения условий.

27 Упражнение для читателей. Почему в первой формуле (с EXISTS) использовано условие СЛУ1.СЛУ_ЗАР > СЛУ2.СЛУ_ЗАРП, а второй формуле (с FORALL) – СЛУ1.СЛУ_ЗАР ≥ СЛУ2.СЛУ_ЗАРП?

Лекция 7. Элементы теории реляционных баз данных: функциональные зависимости и декомпозиция без потерь

7.1. Введение

Эта и две следующие лекции посвящены вопросам теории реляционных баз данных. Поскольку все направление реляционного подхода к организации баз данных является сугубо практическим, эта теория, главным образом, прагматическая. Основная проблема, на решение которой направлена теория реляционных баз данных, состоит в обнаружении полезных свойств некоторых схем баз данных и выработке способов построения таких схем. Принято кратко называть эту проблему проблемой проектирования реляционных баз данных.

Несмотря на свою практическую ориентированность, теория реляционных баз данных является самостоятельным научным направлением, в котором работали (и продолжают работать) многие известные исследователи, чьи имена будут встречаться в наших лекциях. Мы не планировали в данном курсе подробно описывать основные результаты в области теории реляционных баз данных. Наша цель состоит в том, чтобы обеспечить только определения и утверждения, необходимые для общего понимания процесса проектирования реляционных баз данных на основе нормализации.

Поскольку наиболее важные с практической точки зрения свойства реляционных баз данных базируются на понятии функциональной зависимости, мы выделили в отдельную лекцию краткое обсуждение соответствующих теоретических вопросов. Среди этих вопросов наибольший интерес представляют замыкания и покрытия множеств функциональных зависимостей, аксиомы Армстронга и теорема Хита о достаточном условии декомпозиции отношения без потерь. Понятия и утверждения данной лекции действительно нужны для усвоения материала лекции 8, но мы стремились еще и продемонстрировать читателям на несложных примерах, что собой представляет теория реляционных баз данных, каков уровень ее сложности и насколько она понятна интуитивно.

Заметим, что мы не выделяли в отдельные лекции теоретический материал, касающийся многозначных зависимостей и зависимостей соединения. Это было сделано по двум причинам. Во-первых, эти виды зависимостей реже встречаются при моделировании предметной области средствами баз данных. Поэтому мы сочли достаточным представить внутри лекции 9 только основы соответствующего теоретического материала. Во-вторых, хотя теория многозначных зависимостей и зависимостей соединения, по сути, не намного сложнее теории функциональных зависимостей, ее определения и утверждения слишком громоздки для данного курса.

Лекция 8. Проектирование реляционных баз данных на основе принципов нормализации: первые шаги нормализации

8.1. Введение

Эта лекция открывает серию из четырех лекций, посвященных проектированию реляционных баз данных. В данной лекции речь пойдет о нормализации схем отношений с учетом только функциональных зависимостей между атрибутами отношений. Эти «первые шаги» нормализации позволяют получить схему базы данных, в которых все переменные отношений находятся в нормальной форме Бойса-Кодда, обычно расцениваемой удовлетворительной для большей части приложений.

При проектировании базы данных решаются две основные проблемы.

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

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 (за исключением ограничений, задаваемых функциональными и многозначными зависимостями, а также зависимостями проекции/соединения). 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.

Так что в этой и следующей лекциях мы будем считать, что проблема проектирования реляционной базы данных состоит в обоснованном принятии решений о том, из каких отношений должна состоять БД и какие атрибуты должны быть у этих отношений.

В этой и следующей лекциях будет рассмотрен классический подход, при котором весь процесс проектирования базы данных осуществляется в терминах реляционной модели данных методом последовательных приближений к удовлетворительному набору схем отношений. Исходной точкой является представление предметной области в виде одного или нескольких отношений, и на каждом шаге проектирования производится некоторый набор

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

Каждой нормальной форме соответствует определенный набор ограничений, и отношение находится в некоторой нормальной форме, если удовлетворяет свойственному ей набору ограничений. Примером может служить ограничение первой нормальной формы – значения всех атрибутов отношения атомарны[31)]. Поскольку требование первой нормальной формы является базовым требованием классической реляционной модели данных, мы будем считать, что исходный набор отношений уже соответствует этому требованию.

В теории реляционных баз данных обычно выделяется следующая последовательность нормальных форм:

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

Основные свойства нормальных форм состоят в следующем:

- каждая следующая нормальная 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, т. е. декомпозиции отношения, находящегося в предыдущей нормальной форме, на два или более отношений, которые удовлетворяют требованиям следующей нормальной формы.

В этой лекции мы обсудим первые шаги процесса нормализации, в которых учитываются функциональные зависимости между атрибутами отношений. Хотя мы и называем эти шаги первыми, именно они имеют основную практическую важность, поскольку позволяют получить схему реляционной базы данных, в большинстве случаев удовлетворяющую потребности приложений.

8.2. Минимальные функциональные зависимости и вторая нормальная форма

Пусть имеется переменная отношения `СЛУЖАЩИЕ_ПРОЕКТЫ_ЗАДАНИЯ {СЛУ_НОМ, СЛУ_УРОВ, СЛУ_ЗАРП, ПРО_НОМ, СЛУ_ЗАДАН}`. Новые атрибуты `СЛУ_УРОВ` и `СЛУ_ЗАДАН` содержат, соответственно, данные о разряде служащего и о задании, которое выполняет служащий в данном проекте. Будем считать, что разряд служащего определяет размер его заработной платы и что каждый служащий может участвовать в нескольких проектах, но в каждом проекте он выполняет только одно задание. Тогда очевидно, что единственно возможным ключом отношения `СЛУЖАЩИЕ_ПРОЕКТЫ_ЗАДАНИЯ` является составной атрибут `{СЛУ_НОМ, ПРО_НОМ}`. Диаграмма минимального множества FD

показана на рис. 8.1, а возможное тело значения отношения – на рис. 8.2.

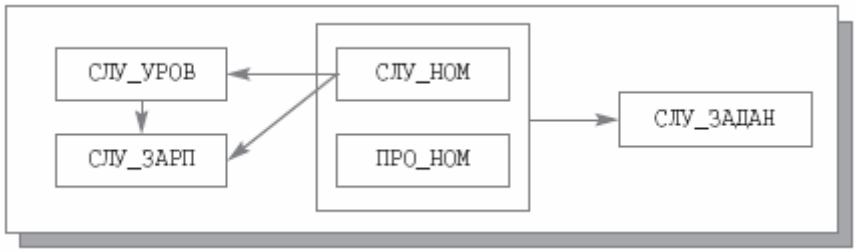


Рис. 8.1. Диаграмма множества FD отношения СЛУЖАЩИЕ_ПРОЕКТЫ_ЗАДАНИЯ

СЛУ_НОМ	СЛУ_УРОВ	СЛУ_ЗАРП	ПРО_НОМ	СЛУ_ЗАДАН
2934	2	22400.00	1	A
2935	3	29600.00	1	B
2936	1	20000.00	1	C
2937	1	20000.00	1	D
2934	2	22400.00	2	D
2935	3	29600.00	2	C
2936	1	20000.00	2	B
2937	1	20000.00	2	A

Рис. 8.2. Возможное значение переменной отношения СЛУЖАЩИЕ_ПРОЕКТЫ_ЗАДАНИЯ

8.2.1. Аномалии обновления, возникающие из-за наличия неминимальных функциональных зависимостей

Во множество FD отношения СЛУЖАЩИЕ_ПРОЕКТЫ_ЗАДАНИЯ входит много FD, в которых детерминантом является не возможный ключ отношения (соответствующие стрелки в диаграмме начинаются не с {СЛУ_НОМ, ПРО_НОМ}, т. е. некоторые функциональные зависимости атрибутов от возможного ключа не являются минимальными). Это приводит к так называемым аномалиям обновления. Под *аномалиями обновления* понимаются трудности, с которыми приходится сталкиваться при выполнении операций добавления кортежей в отношение (INSERT), удаления кортежей (DELETE) и модификации кортежей (UPDATE). Обсудим сначала аномалии обновления, вызываемые наличием FD СЛУ_НОМ⟶ СЛУ_УРОВ (эти аномалии связаны с избыточностью хранения значений атрибутов СЛУ_УРОВ и СЛУ_ЗАРП в каждом кортеже, описывающем задание служащего в некотором проекте).

- *Добавление кортежей.* Мы не можем дополнить отношение СЛУЖАЩИЕ_ПРОЕКТЫ_ЗАДАНИЯ данными о служащем, который в данное время еще не участвует ни в одном проекте (ПРО_НОМ является частью первичного ключа и не может содержать неопределенных значений). Между тем часто бывает, что сначала служащего принимают на работу, устанавливают его разряд и размер зарплаты, а лишь потом назначают для него проект.
- *Удаление кортежей.* Мы не можем сохранить в отношении СЛУЖАЩИЕ_ПРОЕКТЫ_ЗАДАНИЯ данные о служащем, завершившем участие в своем последнем проекте (по той причине, что значение атрибута ПРО_НОМ для этого служащего становится неопределенным). Между тем характерна ситуация, когда между проектами возникают перерывы, не приводящие к увольнению служащих.

- *Модификация кортежей.* Чтобы изменить разряд служащего, мы будем вынуждены модифицировать все кортежи с соответствующим значением атрибута СЛУ_НОМ. В противном случае будет нарушена естественная FD СЛУ_НОМ→СЛУ_УРОВ (у одного служащего имеется только один разряд).

8.2.2. Возможная декомпозиция

Для преодоления этих трудностей можно произвести декомпозицию переменной отношения СЛУЖАЩИЕ_ПРОЕКТЫ_ЗАДАНИЯ на две переменных отношений – СЛУЖ {СЛУ_НОМ, СЛУ_УРОВ, СЛУ_ЗАРП} и СЛУЖ_ПРО_ЗАДАН {СЛУ_НОМ, ПРО_НОМ, СЛУ_ЗАДАН}. На основании теоремы Хита эта декомпозиция является декомпозицией без потерь, поскольку в исходном отношении имелась FD {СЛУ_НОМ, ПРО_НОМ}→СЛУ_ЗАДАН. На рис. 8.3 показаны диаграммы множеств FD этих отношений, а на рис. 8.4 – их значения.

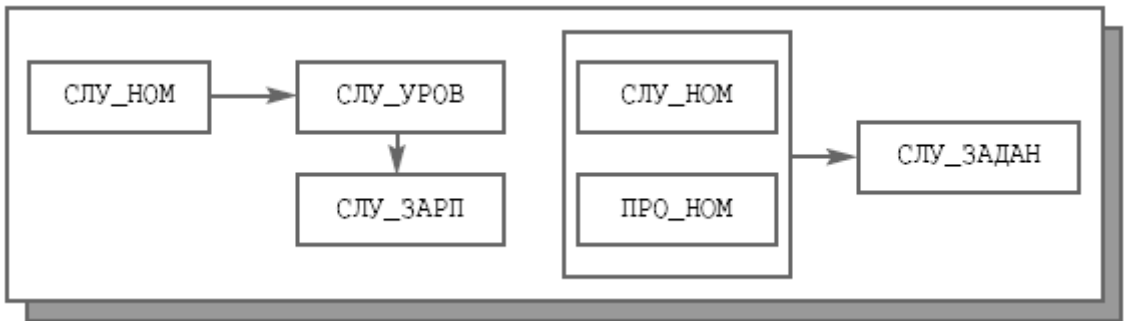


Рис. 8.3. Диаграммы FD в переменных отношений СЛУЖ и СЛУЖ_ПРО_ЗАДАН

Теперь мы можем легко справиться с операциями обновления.

- *Добавление кортежей.* Чтобы сохранить данные о принятом на работу служащем, который еще не участвует ни в каком проекте, достаточно добавить соответствующий кортеж в отношение СЛУЖ.
- *Удаление кортежей.* Если кто-то из служащих прекращает работу над проектом, достаточно удалить соответствующий кортеж из отношения СЛУЖ_ПРО_ЗАДАН. При увольнении служащего нужно удалить кортежи с соответствующим значением атрибута СЛУ_НОМ из отношений СЛУЖ и СЛУЖ_ПРО_ЗАДАН.
- *Модификация кортежей.* Если у служащего меняется разряд (и, следовательно, размер зарплаты), достаточно модифицировать один кортеж в отношении СЛУЖ.

Значение переменной отношения СЛУЖ

СЛУ_НОМ	СЛУ_УРОВ	СЛУ_ЗАРП
2934	2	22400.00
2935	3	29600.00
2936	1	20000.00
2937	1	20000.00

Значение переменной отношения 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
937	2	A

Рис. 8.4. Значения переменных отношений

8.2.3. Вторая нормальная форма

Как видно, на рис. 8.3 отсутствуют FD, не являющиеся *минимальными*. Наличие таких FD на рис. 8.1 вызывало аномалии обновления. Проблема заключалась в том, что атрибут СЛУЖ_УРОВ относился к сущности служащий, в то время как первичный ключ идентифицировал сущность задание_служащего_в_проекте.

Переменная отношения находится во *второй нормальной форме (2NF)* тогда и только тогда, когда она находится в первой нормальной форме, и каждый неключевой атрибут[32)] минимально функционально зависит от первичного ключа[33)].

Переменные отношений СЛУЖ и СЛУЖ_ПРО_ЗАДАН находятся в 2NF (все неключевые атрибуты отношений минимально зависят от первичных ключей СЛУ_НОМ и {СЛУ_НОМ, ПРО_НОМ} соответственно). Переменная отношения СЛУЖАЩИЕ_ПРОЕКТЫ_ЗАДАНИЯ не находится в 2NF (например, FD {СЛУ_НОМ, ПРО_НОМ}→СЛУ_УРОВ не является минимальной). Любая переменная отношения, находящаяся в 1NF, но не находящаяся в 2NF, может быть приведена к набору переменных отношений, находящихся в 2NF. В результате декомпозиции мы получаем набор проекций исходной переменной отношения, естественное соединение значений которых воспроизводит значение исходной переменной отношения (т. е. это декомпозиция без потерь). Для переменных отношений СЛУЖ и СЛУЖ_ПРО_ЗАДАН исходное отношение СЛУЖАЩИЕ_ПРОЕКТЫ_ЗАДАНИЯ воспроизводится их естественным соединением по общему атрибуту СЛУ_НОМ.

Заметим, что допустимое значение переменной отношения СЛУЖ может содержать кортежи, информационное наполнение которых выходит за пределы допустимых значений переменной отношения СЛУЖАЩИЕ_ПРОЕКТЫ_ЗАДАНИЯ. Например, в теле отношения СЛУЖ может находиться кортеж с данными о служащем с номером 2938, который еще не участвует ни в одном проекте. Наличие такого кортежа не влияет на результат естественного

соединения, тело которого все равно будет совпадать с телом допустимого значения переменной отношения СЛУЖАЩИЕ_ПРОЕКТЫ_ЗАДАНИЯ.

8.3. Нетранзитивные функциональные зависимости и третья нормальная форма

В произведенной декомпозиции переменной отношения СЛУЖАЩИЕ_ПРОЕКТЫ_ЗАДАНИЯ множество FD переменной отношения СЛУЖ_ПРО_ЗАДАН предельно просто – в единственной нетривиальной функциональной зависимости детерминантом является возможный ключ. При использовании этой переменной отношения какие-либо аномалии обновления не возникают. Однако переменная отношения СЛУЖ не является такой же совершенной.

8.3.1. Аномалии обновлений, возникающие из-за наличия транзитивных функциональных зависимостей

Функциональные зависимости переменной отношения СЛУЖ по-прежнему порождают некоторые аномалии обновления. Они вызываются наличием транзитивной FD СЛУ_НОМ⟶СЛУ_ЗАРП (через FD СЛУ_НОМ⟶СЛУ_УРОВ и СЛУ_УРОВ⟶СЛУ_ЗАРП). Эти аномалии связаны с избыточностью хранения значения атрибута СЛУ_ЗАРП в каждом кортеже, характеризующем служащих с одним и тем же разрядом.

- *Добавление кортежей.* Невозможно сохранить данные о новом разряде (и соответствующем ему размере зарплаты), пока не появится служащий с новым разрядом. (Первичный ключ не может содержать неопределенные значения.)
- *Удаление кортежей.* При увольнении последнего служащего с данным разрядом мы утратим информацию о наличии такого разряда и соответствующем размере зарплаты.
- *Модификация кортежей.* При изменении размера зарплаты, соответствующей некоторому разряду, мы будем вынуждены изменить значение атрибута СЛУ_ЗАРП в кортежах всех служащих, которым назначен этот разряд (иначе не будет выполняться FD СЛУ_УРОВ⟶СЛУ_ЗАРП).

8.3.2. Возможная декомпозиция

Для преодоления этих трудностей произведем декомпозицию переменной отношения СЛУЖ на две переменных отношений – СЛУЖ1 {СЛУ_НОМ, СЛУ_УРОВ} и УРОВ {СЛУ_УРОВ, СЛУ_ЗАРП}. По теореме Хита, это снова декомпозиция без потерь по причине наличия, например, FD СЛУ_НОМ⟶СЛУ_УРОВ. На рис. 8.5 показаны диаграммы FD этих переменных отношений, а на рис. 8.6 – их возможные значения.

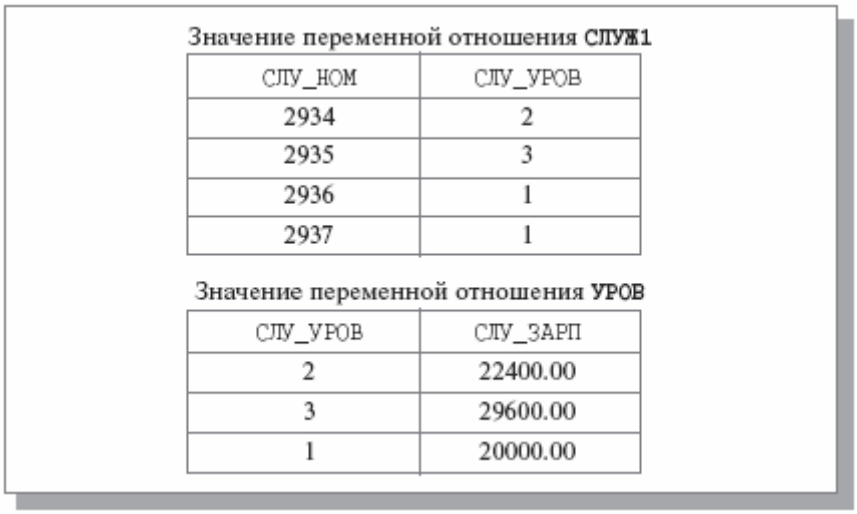
Рис. 8.5. Диаграммы FD в отношениях СЛУЖ1 и УРОВ

Как видно из рис. 8.6, это преобразование обратимо, т. е. любое допустимое значение

исходной переменной отношения СЛУЖ является естественным соединением значений отношений СЛУЖ1 и УРОВ. Также можно заметить, что мы избавились от трудностей при выполнении операций обновления.

- *Добавление кортежей.* Чтобы сохранить данные о новом разряде, достаточно добавить соответствующий кортеж к отношению УРОВ.
- *Удаление кортежей.* При увольнении последнего служащего, обладающего данным разрядом, удаляется соответствующий кортеж из отношения СЛУЖ1, и данные о разряде сохраняются в отношении УРОВ.
- *Модификация кортежей.* При изменении размера зарплаты, соответствующей некоторому разряду, изменяется значение атрибута СЛУ_ЗАРП ровно в одном кортеже отношения УРОВ.

8.3.3. Третья нормальная форма



Значение переменной отношения **СЛУЖ1**

СЛУ_НОМ	СЛУ_УРОВ
2934	2
2935	3
2936	1
2937	1

Значение переменной отношения **УРОВ**

СЛУ_УРОВ	СЛУ_ЗАРП
2	22400.00
3	29600.00
1	20000.00

Рис. 8.6. Тела отношений СЛУЖ1 и УРОВ

Трудности, которые мы испытывали, были связаны с наличием транзитивной FD СЛУ_НОМ⟶ СЛУ_ЗАРП. Наличие этой FD на самом деле означало, что атрибут СЛУ_ЗАРП характеризовал не сущность служащий, а сущность разряд.

Переменная отношения находится в *третьей нормальной форме (3NF)* в том и только в том случае, когда она находится во второй нормальной форме, и каждый неключевой атрибут нетранзитивно[34)] функционально зависит от первичного ключа[35)].

Отношения СЛУЖ1 и УРОВ оба находятся в 3NF (все неключевые атрибуты нетранзитивно зависят от первичных ключей СЛУ_НОМ и СЛУ_УРОВ). Отношение СЛУЖ не находится в 3NF (FD СЛУ_НОМ⟶СЛУ_ЗАРП является транзитивной). Любое отношение, находящееся в 2NF, но не находящееся в 3NF, может быть приведено к набору отношений, находящихся в 3NF. Мы получаем набор проекций исходного отношения, естественное соединение которых воспроизводит исходное отношение (т. е. это декомпозиция без потерь). Для отношений СЛУЖ1 и УРОВ исходное отношение СЛУЖ воспроизводится их естественным соединением по общему атрибуту СЛУ_УРОВ.

Заметим, что допустимые значения отношения УРОВ могут содержать кортежи,

информационное наполнение которых выходит за пределы тела отношения СЛУЖ. Например, в теле отношения УРОВ может находиться кортеж с данными о разряде 4, который еще не присвоен ни одному служащему. Наличие такого кортежа не влияет на результат естественного соединения, который все равно будет являться допустимым значением отношения СЛУЖ.

8.3.4. Независимые проекции отношений. Теорема Риссанена

Обратите внимание, что для переменной отношения СЛУЖ {СЛУ_НОМ, СЛУ_УРОВ, СЛУ_ЗАРП}, кроме декомпозиции на отношения СЛУЖ1 {СЛУ_НОМ, СЛУ_УРОВ} и УРОВ {СЛУ_УРОВ, СЛУ_ЗАРП}, возможна и декомпозиция на отношения СЛУЖ1 {СЛУ_НОМ, СЛУ_УРОВ} и СЛУЖ_ЗАРП {СЛУ_НОМ, СЛУ_ЗАРП}[36]. Оба отношения, полученные путем второй декомпозиции, находятся в 3NF, и эта декомпозиция также является декомпозицией без потерь. Тем не менее вторая декомпозиция, в отличие от первой, не устраняет проблемы, связанные с обновлением отношения СЛУЖ. Например, по-прежнему невозможно сохранить данные о разряде, которым не обладает ни один служащий. Посмотрим, с чем это связано.

Отношения СЛУЖ1 и УРОВ могут обновляться независимо (являются независимыми проекциями), и при этом результат их естественного соединения всегда будет таким, как если бы обновлялось исходное отношение СЛУЖ. Это происходит потому, что FD отношения СЛУЖ трансформировались в индивидуальные ограничения первичного ключа отношений СЛУЖ1 и УРОВ. При второй декомпозиции 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 r_1 и r_2 отношения r являются *независимыми* тогда и только тогда, когда:

- каждая FD в отношении r логически следует[37] из FD в r_1 и r_2;
- общие атрибуты r_1 и r_2 образуют возможный ключ хотя бы для одного из этих отношений.

Мы не будем приводить доказательство этой теоремы, но продемонстрируем ее верность на примере двух показанных выше декомпозиций отношения СЛУЖ. В первой декомпозиции (на проекции СЛУЖ1 и УРОВ) общий атрибут СЛУ_УРОВ является возможным (и первичным) ключом отношения УРОВ, а единственная дополнительная FD отношения СЛУЖ (СЛУ_НОМ ⟶СЛУ_ЗАРП) логически следует из FD СЛУ_НОМ⟶СЛУ_УРОВ и СЛУ_УРОВ⟶СЛУ_ЗАРП, выполняемых для отношений СЛУЖ1 и УРОВ соответственно. Вторая декомпозиция удовлетворяет второму условию теоремы Риссанена (СЛУ_НОМ является первичным ключом в каждом из отношений СЛУЖ1 и СЛУ_ЗАРП), но FD СЛУ_УРОВ⟶СЛУ_ЗАРП не выводится из FD СЛУ_НОМ⟶СЛУ_УРОВ и СЛУ_НОМ⟶СЛУ_ЗАРП.

Атомарным отношением называется отношение, которое невозможно декомпозировать на

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

---

31 Напомним из лекции 3, что *атомарность* значения трактуется в том смысле, что значение типизировано, и с этим значением можно работать только с помощью операций соответствующего типа данных.

32 Неключевым атрибутом называется атрибут, не входящий ни в один возможный ключ.

33 В определении предполагается, что у отношения имеется только один возможный ключ.

34 Очевидно, что FD называется нетранзитивной тогда и только тогда, когда она не является транзитивной.

35 В этом определении опять предполагается, что у отношения имеется только один возможный ключ.

36 Теоретически возможная третья декомпозиция отношения СЛУЖ на отношения СЛУЖ2 {СЛУ_НОМ, СЛУ_ЗАРП} и УРОВ {СЛУ_УРОВ, СЛУ_ЗАРП} не является декомпозицией без потерь. Чтобы убедиться в этом, рассмотрите случай, когда для двух разных разрядов сотрудников назначен один и тот же размер зарплаты. Покажите также, что для этой декомпозиции не выполняются условия теоремы Хита.

37 Т.е. выводится на основе аксиом Армстронга.

Лекция 9. Проектирование реляционных баз данных на основе принципов нормализации: дальнейшая нормализация

9.1. Введение

Функциональные зависимости, о которых мы говорили в предыдущих двух лекциях, и нормальные формы, основанные на учете «аномальных» функциональных зависимостей, являются естественными и легко понимаемыми, поскольку в их основе лежит понятие функционального отображения, интуитивно понятного даже людям, далеким от математики. Конечно, было бы замечательно, если бы ликвидация в ходе нормализации аномальных функциональных зависимостей гарантировала отсутствие аномалий обновления отношений.

К сожалению, эта гарантия в общем случае не обеспечивается. Иногда в переменных отношений требуется поддержка более сложных ограничений целостности, для выражения

которых понятие функции оказывается недостаточным. Класс зависимостей, опирающихся на понятие *функционала* – обобщение 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. Наличие в переменной отношения многозначных зависимостей, не являющихся функциональными зависимостями от возможного ключа, приводит к аномалиям обновления таких отношений. Фейджин показал, что в этом случае возможна декомпозиция данных отношений на две проекции, для которых подобные аномалии обновления не проявляются. Такие проекции находятся в четвертой нормальной форме.

Позже было установлено, что при наличии некоторых естественных ограничений, являющихся обобщением ограничений многозначных зависимостей, и в отношениях, которые находятся в четвертой нормальной форме, проявляются аномалии обновления. Более того, 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 существует 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 пятой нормальной форме, отсутствуют аномалии обновления, которые можно было бы устранить путем декомпозиции, и поэтому при достижении пятой нормальной формы процесс проектирования реляционной базы данных на основе нормализации естественным образом завершается.

9.2. Многозначные зависимости и четвертая нормальная форма

Чтобы перейти к вопросам дальнейшей нормализации, рассмотрим еще одну возможную (четвертую) интерпретацию переменной отношения `СЛУЖ_ПРО_ЗАДАН`. 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. Возможное значение четвертого варианта переменной отношения `СЛУЖ_ПРО_ЗАДАН` показано на рис. 9.1.

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. 9.1. Возможное значение переменной отношения СЛУЖ_ПРО_ЗАДАН (четвертый вариант)

9.2.1. Аномалии обновлений при наличии многозначных зависимостей и возможная декомпозиция

В новом варианте переменной отношения единственным возможным ключом является заголовок отношения `{СЛУ_НОМ, ПРО_НОМ, СЛУ_ЗАДАН}`. Кортеж `{сн, пн, сз}`

входит в тело отношения в том и только в том случае, когда служащий с номером сн выполняет в проекте пн задание сз. 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 переменной отношения СЛУЖ_ПРО_ЗАДАН должно выполняться следующее ограничение ($B_{СПЗ}$ обозначает тело отношения):

$$\text{IF } (\{сн, пн_1, сз_1\} \in B_{СПЗ} \text{ AND } \{сн, пн_2, сз_2\} \in B_{СПЗ})$$
$$\text{THEN } (\{сн, пн_1, сз_2\} \in B_{СПЗ} \text{ AND } \{сн, пн_2, сз_1\} \in B_{СПЗ})$$

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 значения 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.

Трудности, связанные с обновлением переменной отношения СЛУЖ_ПРО_ЗАДАН, решаются путем его декомпозиции на две переменных отношений: СЛУЖ_ПРО_НОМ {СЛУ_НОМ, ПРО_НОМ} и СЛУЖ_ЗАДАНИЕ {СЛУ_НОМ, СЛУ_ЗАДАН}. Значения этих переменных отношений, соответствующие значению переменной отношения СЛУЖ_ПРО_ЗАДАН с рис. 9.1, показаны на рис. 9.2.

Легко видеть, что декомпозиция, представленная на рис. 9.2, является декомпозицией без потерь и что эта декомпозиция решает перечисленные выше проблемы с обновлением переменной отношения СЛУЖ_ПРО_ЗАДАН.

Значение переменной отношения **СЛУЖ_ПРО_НОМ**

СЛУ_НОМ	ПРО_НОМ
2934	1
2934	2
....	....
2941	1

Значение переменной отношения **СЛУЖ_ЗАДАНИЕ**

СЛУ_НОМ	СЛУ_ЗАДАН
2934	A
2934	B
....	....
2941	A
2941	D

Рис. 9.2. Значения переменных отношений СЛУЖ_ПРО_НОМ и СЛУЖ_ЗАДАНИЕ

- *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.

9.2.2. Многозначные зависимости. Теорема Фейджина. Четвертая нормальная форма

Заметим, что последний вариант переменной 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 в случае четвертого варианта отношения СЛУЖ_ПРО_ЗАДАН мы имеем дело с новым видом зависимости, впервые обнаруженным Роном Фейджином в 1971 г. Фейджин назвал зависимости этого вида многозначными (multi-valued dependency – MVD). Как мы увидим немного позже, MVD является обобщением понятия FD.

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

```
(СЛУЖ_ПРО_НОМ WHERE (СЛУ_НОМ = сн AND СЛУ_ЗАДАН = сз)) PROJECT
{ПРО_НОМ}
```

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

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

Многозначные зависимости обладают интересным свойством «двойственности», которое демонстрирует следующая лемма.

Лемма Фейджина

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

Доказательство достаточности условия леммы. Пусть выполняется MVD A$\twoheadrightarrow$B. Пусть имеется некоторое удовлетворяющее этой зависимости значение *r* переменной отношения *R*,

a обозначает значение атрибута A в некотором кортеже тела B_r, a {b} – множество значений атрибута B, взятых из всех кортежей B_r, в которых значением атрибута A является a. Предположим, что для этого значения a MVD A⟶⟶C не выполняется. Это означает, что существуют такое допустимое значение c атрибута C и такое значение $b \in \{b\}$, что кортеж $\{a, b, c\} \notin B_r$. Но это противоречит наличию MVD A⟶⟶B. Следовательно, если выполняется MVD A⟶⟶B, то выполняется и MVD A⟶⟶C. Аналогично можно доказать необходимость условия леммы.

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

FD является частным случаем MVD, когда множество значений зависимого атрибута обязательно состоит из одного элемента. Таким образом, если выполняется FD A⟶B, то выполняется и MVD A⟶⟶B .[39)]

Мы видим, что отношения СЛУЖ_ПРО_НОМ и СЛУЖ_ЗАДАНИЕ не содержат MVD, отличных от FD, и именно в этом выигрывает декомпозиция из рис. 9.2. Правомочность этой декомпозиции доказывается приведенной ниже теоремой Фейджина, которая является уточнением и обобщением теоремы Хита.

Теорема Фейджина

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

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

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

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

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

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

Теорема Фейджина обеспечивает основу для декомпозиции отношений, удаляющей «аномальные» многозначные зависимости, с приведением отношений в четвертую нормальную форму.

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

В сущности, 4NF является BCNF, в которой многозначные зависимости вырождаются в функциональные (позволим себе на один момент отказаться от сокращений). Понятно, что отношение СЛУЖ_ПРО_ЗАДАН не находится в 4NF, поскольку детерминант MVD СЛУ_НОМ $\longrightarrow\!\!\!\!\!\rightarrow$ПРО_НОМ и СЛУ_НОМ$\longrightarrow\!\!\!\!\!\rightarrow$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.

---

39 Упражнение по ходу лекции. Пусть имеется отношение r с атрибутами A , B , C (в общем случае, составными), в котором существует FD A$\longrightarrow$B . Что в этом случае можно сказать про зависимость атрибутов A и C ?

Лекция 10. Проектирование реляционных баз данных с использованием семантических моделей: ER-диаграммы

10.1. Введение

Широкое распространение реляционных 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.

10.1.1. Ограниченность реляционной модели при проектировании баз данных

При использовании в проектировании ограниченность реляционной модели проявляется в следующих аспектах.

- Модель не обеспечивает достаточных средств для представления смысла данных. Семантика реальной предметной области должна независимым от модели способом представляться в голове проектировщика. В частности, это относится к отмечавшейся нами ранее проблеме представления ограничений целостности, выходящих за пределы ограничений первичного и внешнего ключа.
- Во многих прикладных областях трудно моделировать предметную область на основе плоских таблиц[40)]. В ряде случаев на самой начальной стадии проектирования дизайнеру приходится нелегко, поскольку от него требуется описать предметную область в виде одной (возможно, даже ненормализованной) таблицы.
- Хотя весь процесс проектирования происходит на основе учета зависимостей, реляционная модель не предоставляет какие-либо формализованные средства для представления этих зависимостей.
- Несмотря на то, что процесс проектирования начинается с выделения некоторых существенных для приложения объектов предметной области («сущностей») и выявления связей между этими сущностями, реляционная модель данных не предлагает какого-либо механизма для разделения сущностей и связей[41)].

10.1.2. Семантические модели данных

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

Прежде чем мы коротко рассмотрим особенности двух распространенных семантических моделей, остановимся на возможных областях их применения. Чаще всего на практике семантическое моделирование используется на первой стадии проектирования базы данных. При этом в терминах семантической модели производится концептуальная схема базы данных, которая затем вручную преобразуется к реляционной (или какой-либо другой) схеме. Этот процесс выполняется под управлением методик, в которых достаточно четко оговорены все этапы такого преобразования. 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.

Неоднократно приходилось и приходится наблюдать ситуации, в которых отсутствие такого рода документации существенно затрудняет внесение даже небольших изменений в схему существующей реляционной БД. Конечно, это относится к случаям, 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 ее последующим преобразованием в реляционную схему БД затруднительна в случае больших БД (содержащих несколько сотен таблиц). Причины, по всей видимости, не требуют пояснений.

История систем автоматизации проектирования баз данных (CASE-средств[42]) началась с автоматизации процесса рисования диаграмм, проверки их формальной корректности, обеспечения 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. 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 ее в состав системы проектирования баз данных?

Эта идея, естественно, показалась разумной производителям CASE-средств проектирования БД. Подавляющее большинство подобных систем, представленных на рынке, обеспечивает автоматизированное преобразование диаграммных концептуальных схем баз данных, представленных в той или иной семантической модели данных, в реляционные схемы, специфицированные чаще всего на языке SQL. У читателя 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 (ограничений целостности общего вида, триггеров и хранимых процедур и т. д.), которые невозможно сгенерировать автоматически на основе концептуальной схемы. Поэтому на завершающем этапе проектирования реляционной схемы снова требуется ручная работа проектировщика.

Еще раз обратите 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, 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 нам не известны 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.

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.

Наконец, третья возможность, которую следует упомянуть, хотя она еще не вышла (или только выходит, а может быть, так никогда и не выйдет) за пределы исследовательских и экспериментальных проектов, – это работа с базой данных в семантической модели, т. е. СУБД, основанные на семантических моделях данных. При этом снова рассматриваются два варианта: обеспечение пользовательского интерфейса на основе семантической модели данных с автоматическим отображением конструкций этого интерфейса в реляционную модель данных (это задача примерно того же уровня сложности, что и автоматическая компиляция концептуальной схемы базы данных в реляционную схему) и прямая реализация СУБД, основанная на какой-либо семантической модели данных. Многие авторитетные специалисты полагают, что ближе всего ко второму подходу объектно-ориентированные СУБД, чьи модели данных по многим параметрам близки к семантическим моделям (хотя в некоторых аспектах они более мощны, а в некоторых – более слабы).

---

40 Начиная с этой лекции, мы переходим к использованию терминов таблица, строка и столбец вместо строгих реляционных терминов отношение, атрибут и таблица, поскольку здесь под «реляционными» базами данных понимаются, главным образом, SQL-ориентированные базы данных, для которых эта упрощенная терминология более естественна.

41 Многие сторонники реляционного подхода считают отсутствие раздельного представления сущностей и связей преимуществом реляционной модели данных, мотивируя это тем, что зачастую то, что вчера считалось сущностью, сегодня разумнее принять за связь, и наоборот. Это, безусловно, верно с точки зрения поддержки и модификации существующих реляционных баз данных, но отнюдь не так с точки зрения проектирования базы данных.

42 Позволю себе одно терминологическое замечание, которое может показаться несколько наивным для специалистов в области инженерии программного обеспечения (software engineering), к числу которых я не принадлежу. Издавна существует отдельный класс программных систем, предназначенных для автоматизации проектирования новых продуктов в разных областях промышленности – автомобилестроении, аэрокосмической промышленности, электронной промышленности и т.д. Очевидно, что процесс проектирования автомобиля принципиально отличается от процесса проектирования микропроцессора, но, тем не менее, для обозначения любой Системы Автоматизации ПРоектирования используется собирательный термин САПР (CAD – Computer Aided Design). Это оправдывается тем, что разные подклассы САПР имеют гораздо больше общих черт, чем различий. Так вот, по моему мнению, система автоматизации проектирования БД по своему

назначению и строению в большей степени является системой класса САПР, чем системой класса CASE (Computer Aided Software Engineering). По всей видимости, средства автоматизированной поддержки проектирования баз данных стали в свое время называть CASE-средствами, поскольку они обычно включали не только инструменты для поддержки проектирования, но и инструменты, поддерживающие проектирование и разработку приложений баз данных. В последние годы такие инструменты все реже производятся в виде одного пакета, и сам термин «CASE-средство» почти вышел из употребления. Тем не менее, поскольку не появилось какое-либо другое собирательное название средств поддержки проектирвания баз данных, мы будем продолжать использовать именно этот термин.

Лекция 11. Проектирование реляционных баз данных с использованием семантических моделей: диаграммы классов языка UML

11.1. Введение

В этой лекции мы обсудим основные понятия диаграмм классов языка UML и возможности применения этой диаграммной модели для проектирования реляционных баз данных. Кроме того, в лекции будет кратко рассмотрен язык объектных ограничений OCL и приведены примеры формулировок на языке OCL ограничений целостности в терминах концептуальной схемы базы данных.

Языку объектно-ориентированного моделирования UML (Unified Modeling Language) посвящено великое множество книг, многие из которых переведены на русский язык (а некоторые и написаны российскими авторами). Язык UML разработан и развивается 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).

UML позволяет моделировать разные виды систем: чисто программные, чисто аппаратные, программно-аппаратные, смешанные, явно включающие деятельность людей и т. д. Даже если бы мы ограничились возможностями использования UML для проектирования программных информационных систем, это слишком далеко увело бы нас от основной темы курса.

Но, помимо прочего, язык UML активно применяется для проектирования реляционных БД. Для этого используется небольшая часть языка (диаграммы классов), да и то не в полном объеме. С точки зрения проектирования реляционных БД модельные возможности не слишком отличаются от возможностей ER-диаграмм. Но все же мы кратко опишем диаграммы классов UML, поскольку их использование при проектировании реляционных БД позволяет оставаться в общем контексте UML и применять другие виды диаграмм для проектирования приложений баз данных.

Лекция 12. Пример общей организации СУБД. Физическое представление реляционных баз данных во внешней памяти. Индексные структуры

12.1. Введение

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

После успешного завершения работ по созданию этой системы и получения экспериментальных результатов ее использования был разработан целый ряд коммерчески доступных реляционных систем, в том числе и на основе непосредственного развития System R. Исключительно важен опыт, приобретенный при разработке этой системы. Практически во всех более поздних реляционных СУБД в той или иной степени используются методы, примененные в System R. Поэтому лекции, посвященные внутренней организации SQL-ориентированных СУБД, во многом опираются на материалы статей, посвященных System R.

Организации СУБД System R посвящена обширная библиография [5.1–5.17] (здесь подобраны публикации, свободно доступные в Internet к моменту написания этого текста). Поскольку публикации появлялись по ходу практической реализации системы, каждая из них отражает состояние дел (идейное и практическое) именно на том этапе работы, когда была написана соответствующая статья. Некоторые идеи и представления, естественно, изменялись по ходу работы. Сравнительно законченное представление о системе в целом дают только заключительные публикации. Имеется обширный обзор литературы, посвященной System R и связанных с ней проектов [3.6].

Лекция 13. Методы управления транзакциями. Сихронизационные блокировки, временные метки и версии

13.1. Введение

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

13.2. Общее понятие транзакции и основные характеристики транзакций

Более точно, в современных СУБД поддерживается понятие транзакции, характеризуемое

аббревиатурой ACID (Atomicy, Consistency, Isolation и Durability). В соответствии с этим понятием под транзакцией разумеется последовательность операций над базой данных, обладающая следующими свойствами.

- *Атомарность (Atomicy)*. Это свойство означает, что результаты всех операций, успешно выполненных в пределах транзакции, должны быть отражены в состоянии базы данных, либо в состоянии базы данных не должно быть отражено действие ни одной операции (конечно, здесь речь идет об операциях, изменяющих состояние базы данных). Свойство атомарности, которое часто называют свойством “все или ничего”, позволяет относиться к транзакции, как к динамически образуемой составной операции над базой данных (в общем случае состав и порядок выполнения операций, выполняемых внутри транзакции, становится известным только на стадии выполнения).
- *Согласованность (Consistency).* В классическом смысле это свойство означает, что транзакция может быть успешно завершена с *фиксацией* результатов своих операций только в том случае, 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. (С точки зрения автора, в контексте баз данных термины *согласованность* и *целостность* эквивалентны. Единственным критерием согласованности данных является их удовлетворение ограничениям целостности, т.е. база данных находится в согласованном состоянии тогда и только тогда, когда она находится в целостном состоянии.)
- *Изоляция (Isolation).* Требуется, чтобы две одновременно (параллельно или квазипараллельно) выполняемые транзакции никоим образом не действовали одна на другую. Другими словами, результаты выполнения операций транзакции *T1* не должны быть видны никакой другой транзакции *T2* до тех пор, пока транзакция *T1* не завершится успешным образом.
- *Долговечность (Durability).* После успешного завершения транзакции все изменения, которые были внесены в состояние базы данных операциями этой транзакции, должны гарантированно сохраняться, даже в случае сбоев аппаратуры или программного обеспечения. Этому аспекту транзакционных систем посвящается лекция 14.

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

13.2.1. Атомарность транзакций

В этом смысле под транзакцией понимается неделимая с точки зрения воздействия на БД последовательность операторов манипулирования данными (чтения, удаления, вставки, модификации), такая, что либо результаты всех операторов, входящих в транзакцию, отображаются в состоянии базы данных, либо воздействие всех этих операторов полностью отсутствует.

Лозунгом транзакции является «Все или ничего»: при завершении транзакции оператором `COMMIT` (высокоуровневый аналог операции `END TRANSACTION` в интерфейсе RSS, см. лекцию 12) результаты гарантированно фиксируются во внешней памяти (смысл термина

commit состоит в запросе «фиксации» результатов транзакции); при завершении транзакции оператором ROLLBACK (высокоуровневый аналог операции RESTORE в интерфейсе RSS, см. лекцию 12) результаты гарантированно отсутствуют во внешней памяти (смысл термина *rollback* состоит в запросе ликвидации результатов транзакции).

Каким образом в СУБД поддерживаются индивидуальные откаты транзакций, описывается в лекции 14.

13.2.2. Транзакции и целостность баз данных

Понятие транзакции имеет непосредственную связь с понятием целостности базы данных. Очень часто база данных может обладать такими ограничениями целостности, которые просто невозможно не нарушить, выполняя только один оператор изменения базы данных. Например, в базе данных СЛУЖАЩИЕ−ОТДЕЛЫ (см. лекцию 1) естественным ограничением целостности является совпадение значения атрибута ОТД_РАЗМЕР в кортеже таблицы ОТДЕЛЫ, описывающей данный отдел (например, отдел 625), с числом кортежей 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 операции база данных окажется в нецелостном состоянии.

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

Более точно, различаются два вида ограничений целостности: немедленно проверяемые и откладываемые. К немедленно проверяемым ограничениям целостности относятся такие ограничения, проверку которых бессмысленно или даже невозможно откладывать. Примером ограничения, проверку которого откладывать бессмысленно, являются ограничения домена (например, возраст сотрудника не может превышать 150 лет). Более сложным ограничением, проверку которого невозможно отложить, является следующее: зарплата сотрудника не может быть увеличена за одну операцию более чем на 100000 рублей. Немедленно проверяемые ограничения целостности соответствуют уровню отдельных операторов языкового уровня СУБД. При их нарушениях не производится откат транзакции, а лишь отвергается соответствующий оператор.

Откладываемые ограничения целостности – это ограничения на базу данных, а не на какие-либо отдельные операции. По умолчанию такие ограничения проверяются при конце транзакции, и их нарушение вызывает автоматическую замену оператора COMMIT на оператор ROLLBACK. Однако в некоторых системах поддерживается специальный оператор насильственной проверки ограничений целостности внутри транзакции. Если после выполнения такого оператора обнаруживается, что условия целостности не выполнены, пользователь может сам выполнить оператор ROLLBACK с откатом транзакции до ее начала

или до установленной ранее точки сохранения или постараться устранить причины нецелостного состояния базы данных внутри транзакции (видимо, это осмысленно только при использовании интерактивного режима работы).

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

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

Во-первых, если при выполнении транзакции не устанавливать точки сохранения и не проверять периодически соответствие текущего состояния базы данных (с точки зрения данной транзакции) ограничениям целостности, то долговременно выполняемая транзакция вполне вероятно может быть «откачена» системой при выполнении завершающего оператора COMMIT. Конечно, это означает непроизводительный расход системных ресурсов и времени пользователей. Во-вторых, чем длиннее транзакция, модифицирующая состояние базы данных, тем потенциально больше ограничений целостности придется проверять при ее завершении и тем накладнее становится оператор COMMIT.

Простое и элегантное решение этой проблемы предлагается в [1.5]. Авторы предлагают отказаться от откладываемых ограничений целостности базы данных, а вместо этого ввести составные операторы изменения базы данных (нечто наподобие блоков BEGIN … END, поддерживаемых в языках программирования). После выполнения каждого такого блока (или отдельного оператора изменения базы данных, используемого без операторов начала и конца блока) база данных должна находится в целостном состоянии. Если составной оператор нарушает ограничение целостности, то он целиком отвергается, и вырабатывается соответствующий код ошибки. Транзакция в этом случае не откатывается. Понятно, что при использовании такого подхода при выполнении оператора COMMIT не требуется проверять ограничения целостности, и каждая зафиксированная транзакция будет оставлять базу данных в целостном состоянии.

Интересно, что для реализации описанного подхода не требуются какие-либо новые механизмы, кроме точек сохранения транзакции, насильственной проверки ограничений целостности и частичных откатов транзакций, а отмеченные ранее проблемы снимаются. К сожалению, насколько известно автору данной книги, этот подход на практике пока не применяется.

13.2.3. Изолированность транзакций

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

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

При соблюдении обязательного требования поддержки целостности базы данных возможно наличие нескольких уровней изолированности транзакций. Заметим, что впервые эти уровни изолированности транзакций были установлены и описаны участниками проекта System R.

Отсутствие потерянных изменений (первый уровень изолированности)

Рассмотрим сценарий совместного выполнения двух транзакций, показанный на рис. 13.1. В момент времени t_1 транзакция T_1 изменяет объект базы данных o (выполняет операцию W(o)). До завершения транзакции T_1 в момент времени $t_2 > t_1$ транзакция T_2 также изменяет объект o. В момент времени $t_3 > t_2$ транзакция T_2 завершается оператором ROLLBACK (например, по причине нарушения ограничений целостности).

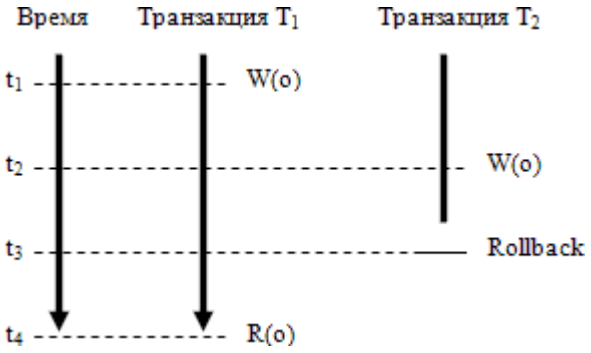


Рис. 13.1. Потерянные изменения

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

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

Отсутствие чтения «грязных» данных (второй уровень изолированности)

Рассмотрим сценарий совместного выполнения транзакций T_1 и T_2, показанный на рис. 13.2. В момент времени t_1 транзакция T_1 изменяет объект базы данных o (выполняет операцию

W(o)). В момент времени $t_2 > t_1$ транзакция T_2 читает объект o (выполняет операцию R(o)). Поскольку транзакция T_1 еще не завершена, транзакция T_2 видит несогласованные *«грязные» данные*. В частности, в момент времени $t_3 > t_2$ транзакция T_1 может завершиться откатом (например, по причине нарушения ограничений целостности).

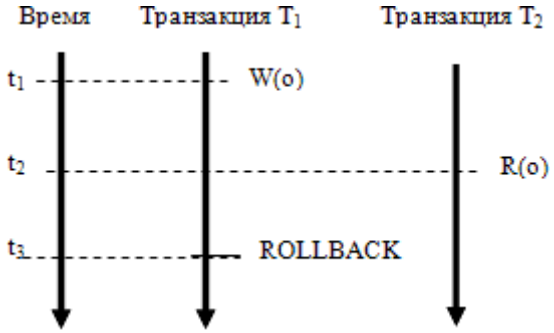


Рис. 13.2. «Грязные» чтения

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

Отсутствие неповторяющихся чтений (третий уровень изоляции)

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

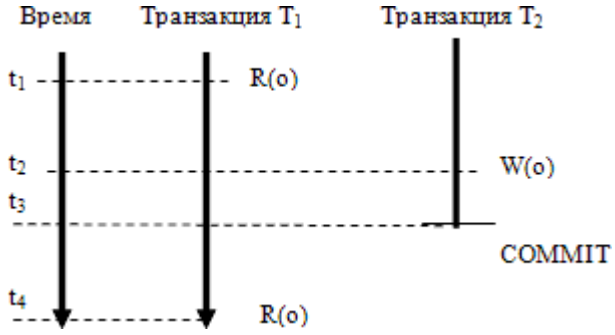


Рис. 13.3. Неповторяющиеся чтения

Чтобы избежать *неповторяющихся чтений*, до завершения транзакции T_1 никакая другая транзакция не должна изменять объект o (для этого достаточно заблокировать доступ по записи к объекту o до завершения транзакции T_1). Часто это является максимальным требованием к средствам обеспечения изолированности транзакций, хотя, как будет видно

немного позже, отсутствие неповторяющихся чтений еще не гарантирует реальной изолированности пользователей.

Заметим, что существует возможность обеспечения разных уровней изолированности для разных транзакций, выполняющихся в одной системе баз данных (кстати, соответствующие операторы были предусмотрены уже в стандарте SQL:1992). Как уже отмечалось, для корректного соблюдения ограничений целостности достаточен первый уровень. Существует ряд приложений, которым хватает первого уровня изолированности (например, прикладные или системные статистические утилиты, для которых некорректность индивидуальных данных несущественна). При этом удается существенно сократить накладные расходы СУБД и повысить общую эффективность.

Проблема фантомов

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

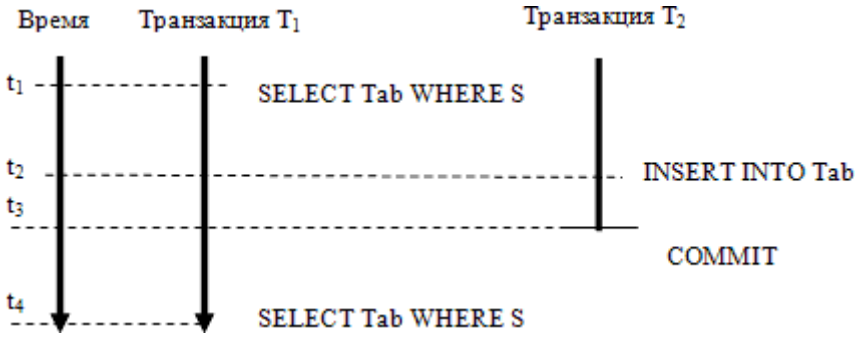


Рис. 13.4. Проблема фантомов

В момент времени t_1 транзакция T_1 выполняет оператор выборки кортежей таблицы `Tab` с условием выборки `S` (т.е. выбирается часть кортежей таблицы `Tab`, удовлетворяющих условию `S`). До завершения транзакции T_1 в момент времени $t_2 > t_1$ транзакция T_2 вставляет в таблицу `Tab` новый кортеж `r`, удовлетворяющий условию `S`, и успешно завершается. В момент времени $t_3 > t_2$ транзакция T_1 повторно выполняет тот же оператор выборки, и в результате появляется кортеж, который отсутствовал при первом выполнении оператора.

Конечно, такая ситуация противоречит идее изолированности транзакций и может возникнуть даже на третьем уровне изолированности транзакций. Чтобы избежать появления кортежей-фантомов, требуется более высокий «логический» уровень изоляции транзакций. Идеи требуемого механизма (предикатные синхронизационные блокировки) появились также еще во время выполнения проекта System R, но в большинстве систем не реализованы.

13.2.4. Сериализация транзакций

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

Пусть в системе одновременно выполняется некоторое множество транзакций $S = \{T_1,$

$T_2, \ldots, T_n\}$. План (способ) выполнения набора транзакций S (в котором, вообще говоря, чередуются или реально параллельно выполняются операции разных транзакций) называется сериальным, если результат совместного выполнения транзакций эквивалентен результату некоторого последовательного выполнения этих же транзакций $(T_{i1}, T_{i2}, \ldots, T_{in})$.

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

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

Между транзакциями T_1 и T_2 могут существовать следующие виды конфликтов:

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

Практические методы сериализации транзакций основываются на учете этих конфликтов.

Лекция 14. Средства журнализации и восстановления баз данных

14.1. Введение

Одним из основных требований к развитым СУБД является надежность хранения баз данных. Это требование предполагает, в частности, возможность восстановления согласованного состояния базы данных после любого рода аппаратных и программных сбоев. Очевидно, что для выполнения восстановлений необходима некоторая дополнительная информация. В подавляющем большинстве современных реляционных СУБД такая избыточная дополнительная информация поддерживается в виде журнала изменений базы данных.

Итак, общей целью журнализации изменений баз данных является обеспечение возможности

восстановления согласованного состояния базы данных после любого сбоя. Поскольку основой поддержания целостного состояния базы данных является механизм транзакций, журнализация и восстановление тесно связаны с понятием транзакции. Общими принципами восстановления являются следующие:

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

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

Возможны следующие ситуации, при которых требуется производить восстановление состояния базы данных:

- Индивидуальный откат транзакции. Тривиальной ситуацией отката транзакции является ее явное завершение оператором `ROLLBACK`. Возможны также ситуации, когда откат транзакции инициируется системой. Примерами могут быть возникновение исключительной ситуации в прикладной программе (например, деление на ноль) или выбор транзакции в качестве жертвы при разрушении синхронизационного тупика. Для восстановления согласованного состояния базы данных при индивидуальном откате транзакции нужно устранить последствия операторов модификации базы данных, которые выполнялись в этой транзакции.
- Восстановление после внезапной потери содержимого оперативной памяти (мягкий сбой). Такая ситуация может возникнуть при аварийном выключении электрического питания, при возникновении неустранимого сбоя процессора (например, срабатывании контроля основной памяти) и т.д. Ситуация характеризуется потерей той части базы данных, которая к моменту сбоя содержалась в буферах оперативной памяти СУБД.
- Восстановление после поломки основного внешнего носителя базы данных (жесткий сбой). Эта ситуация при достаточно высокой надежности современных устройств внешней памяти может возникать сравнительно редко, но, тем не менее, СУБД должна быть в состоянии восстановить базу данных даже и в этом случае. Основой восстановления является архивная копия и журнал изменений базы данных.

Во всех трех случаях основой восстановления является хранение избыточных данных. Эти избыточные данные хранятся в журнале, содержащем последовательность записей об изменении базы данных.

Возможны два основных варианта ведения журнальной информации. В первом варианте для каждой транзакции поддерживается отдельный локальный журнал изменений базы данных этой транзакцией. Эти локальные журналы используются для индивидуальных откатов транзакций и могут поддерживаться в основной (правильнее сказать, в виртуальной) памяти СУБД. Кроме того, поддерживается общий журнал изменений базы данных, используемый для восстановления состояния базы данных после мягких и жестких сбоев.

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

который используется и при выполнении индивидуальных откатов. Здесь мы рассматриваем именно этот вариант.

В этой лекции сначала мы проанализируем особенности подсистемы СУБД, управляющей буферами основной памяти, и связь механизмов буферизации и журнализации. Затем на содержательном уровне без технических деталей обсуждим общие принципы журнализации изменений и восстановления целостного состояния базы данных после сбоев, опираясь, в основном на методы, применявшиеся в System R и ее ранних предшественниках.

14.2. Буферизация блоков базы данных в основной памяти и ее связь с журнализацией

Журнализация операций изменения базы данных[66)] тесно связана не только с управлением транзакциями, но и с буферизацией блоков базы данных в основной памяти. По причинам объективно существующей разницы в скорости работы процессоров и основной памяти и устройств внешней памяти (эта разница в скорости существовала, существует, и будет существовать всегда) буферизация блоков базы данных в основной памяти является единственным реальным способом достижения приемлемой эффективности СУБД. Без поддержки буферизации базы данных СУБД работала бы со скоростью магнитных дисков, т.е. на несколько порядков медленнее, чем если бы обработка данных происходила в основной памяти.

Если бы каждая запись об изменении базы данных, которая должна поступить в журнал при выполнении любой операции обновления базы данных, реально немедленно перемещалась бы во внешнюю память, это 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: при нормальной работе буфер выталкивается во внешнюю память журнала только при полном заполнении записями. Более точно, для буферизации записей журнала обычно используются два буфера. После полного заполнения первый буфер выталкивается на магнитный диск, и пока совершается этот обмен, журнальные записи размещаются во втором буфере. К моменту конца обмена заполняется второй буфер, он выталкивается во внешнюю память, а журнальные записи снова размещаются в первом буфере и т.д.

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

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

14.2.1. 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 основной памяти, и нет свободных страниц основной памяти, в соответствии с некоторым критерием выбирается некоторая занятая страница основной памяти, освобождается (т.е. изымается из виртуальной памяти какого-то процесса и, может быть, копируется на диск) и подключается к виртуальной памяти запросившего процесса с предварительным считыванием с диска нужных данных.

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

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

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

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

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

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

14.2.2. Физическая синхронизация

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

Следует заметить, что, вообще говоря, координацию параллельного доступа к страницам буферного пула не обеспечивает логическая синхронизация, используемая для сериализации транзакций (см. лекцию 13). Например, предположим, что в двух параллельно выполняемых транзакциях одновременно выполняются операции модификации кортежей, у одного из которых tid = (*n*, 1), а у другого tid = (*n*, 2). Если в СУБД используются блокировки на уровне кортежей, то система допустит параллельное выполнение этих двух операций, и они будут одновременно изменять страницу, содержащую копию блока базы данных с номером *n*. При выполнении обеих операций может потребоваться перемещение кортежей внутри этого блока, и понятно, что в результате ничего хорошего, скорее всего, не получится. Аналогично, логическая синхронизация может легко допустить параллельное выполнение нескольких операций, требующих обновления одного и того же индекса. Некоординированное параллельное обновление B-дерева с большой вероятностью приводит к разрушению его структуры.

Поэтому при выполнении операций уровня RSS необходимо поддерживать дополнительную «физическую» синхронизацию, в которой единицами блокировки служат страницы буферного пула (или блоки) базы данных. В пределах операции перед чтением из страницы буферного пула (блока базы данных) требуется запросить у подсистемы управления буферным пулом блокировку соответствующей страницы (блока) в режиме S, а перед записью в страницу (в блок) – ее блокировку в режиме X. Совместимость блокировок обычная, такая же, как в табл. 13.1.

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

При обнаружении ошибки операции нужно ликвидировать все ее следы в базе данных и выдать соответствующий код ошибки на уровень RDS. Проще всего сделать это, произведя

обратные изменения всех страниц (блоков базы данных), которые были изменены при прямом выполнении операции. Но для этого требуется, чтобы все страницы (блоки базы данных), заблокированные при выполнении операции, оставались заблокированными до конца этой операции.

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

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

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

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

14.2.3. Протокол упреждающей записи в журнал и его связь с буферизацией

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

Заметим, что эта проблема не возникает при индивидуальных откатах транзакций, поскольку в этих случаях содержимое основной памяти не утрачено, и при восстановлении можно пользоваться содержимым как буфера журнала, так и буферных страниц базы данных. Но если произошел мягкий сбой, и содержимое буферов утрачено, то для проведения восстановления базы данных необходимо иметь некоторое согласованное состояние журнала и базы данных во внешней памяти.

Основным принципом согласованной политики выталкивания буфера журнала и буферных страниц базы данных является то, что запись об изменении объекта базы данных должна оказаться во внешней памяти журнала раньше, чем измененный объект окажется во внешней

памяти базы данных. Соответствующий протокол журнализации (и управления буферизацией) называется WAL (Write Ahead Log, «пиши сначала в журнал») и состоит в том, что если требуется вытолкнуть во внешнюю память буферную страницу, содержащую измененный объект базы данных, то перед этим нужно гарантировать выталкивание во внешнюю память журнала буферной страницы журнала, содержащей запись об изменении этого объекта.

При следовании протоколу WAL, если во внешней памяти базы данных находится некоторый объект базы данных, по отношению к которому выполнена операция модификации, то во внешней памяти журнала обязательно находится запись, соответствующая этой операции. Обратное неверно, т.е. если во внешней памяти журнала содержится запись о некоторой операции изменения объекта базы данных, то сам измененный объект может отсутствовать во внешней памяти базы данных.

Дополнительное условие на выталкивание буферов накладывается тем требованием, что каждая успешно завершенная транзакция должна быть реально зафиксирована во внешней памяти. Какой бы сбой не произошел, система должна быть в состоянии восстановить состояние базы данных, содержащее результаты всех транзакций, зафиксированных до момента сбоя.

Самым простым решением было бы выталкивание буфера журнала, за которым следует массовое выталкивание буферов страниц базы данных, изменявшихся данной транзакцией. Довольно часто так и делают, но это вызывает существенные накладные расходы при выполнении операции фиксации транзакции.

Оказывается, что минимальным требованием, гарантирующим возможность восстановления последнего согласованного состояния базы данных, является выталкивание при фиксации транзакции во внешнюю память журнала всех записей об изменении базы данных этой транзакцией. При этом последней записью в журнал, производимой от имени данной транзакции, является специальная запись о конце транзакции.

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.

14.3. Индивидуальный откат транзакции

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

Итак, индивидуальный откат транзакции (еще раз подчеркнем, что это возможно только для

незавершенных транзакций) выполняется следующим образом:

- Выбирается очередная журнальная запись из списка данной транзакции.
- Выполняется противоположная по смыслу операция: вместо операции `INSERT` выполняется соответствующая операция `DELETE`, вместо операции `DELETE` выполняется `INSERT`, и вместо прямой операции `UPDATE` – обратная операция `UPDATE`, восстанавливающая предыдущее состояние объекта базы данных.
- Любая из этих обратных операций также журнализуется. Собственно для индивидуального отката это не нужно, но при выполнении индивидуального отката транзакции может произойти мягкий сбой, при восстановлении после которого потребуется откатить транзакции, для которых не полностью выполнен индивидуальный откат.
- При успешном завершении отката в журнал заносится запись о конце транзакции. С точки зрения журнала такая транзакция является зафиксированной.

Лекция 15. Общее введение в SQL, типы данных и средства определения доменов

15.1. Введение

Оставшаяся часть этого курса посвящается языку реляционных баз данных SQL [3.9], [4.1–4.3]. В курсе о реляционных базах данных невозможно обойтись без материала, который относится к этому языку. Это связано совсем не с тем, что язык представляет собой особое достижение в области реляционных БД. Напротив, многие черты SQL, начиная с самых первых его вариантов, противоречили принципам реляционной модели данных, заложенным Эдгаром Коддом [2.1]. С другой стороны, спецификация языка SQL, по своей сути, является завершенной спецификацией модели данных, которая сегодня играет роль суррогата реляционной модели. Если бы мы попытались обойтись в этом курсе без обсуждения языка SQL, курс был бы полностью оторван от жизни. Сегодня SQL является lingua franca в мире баз данных. Интерфейсы, основанные на SQL, поддерживаются почти во всех используемых СУБД, далеко не все из которых первоначально разрабатывались как реляционные системы. И похоже, что эта ситуация при жизни нынешнего поколения радикальным образом не изменится. Кроме того, язык сам по себе достаточно интересен. В нем нашел отражение многолетний практический опыт многих людей, и он впитал в себя многие положительные (и отрицательные) черты других языков и подходов (не только языков баз данных и не только реляционного подхода). В данной лекции после небольшой исторической справки и краткого введения в структуру языка SQL будут рассмотрены типы данных, допустимые в языке SQL и в SQL-ориентированных базах данных, а также языковые средства определения, изменения определения и отмены определения доменов.

В начале лекции мы представим небольшой исторический обзор SQL. Язык уже далеко не молод. В 2004 г. сообщество баз данных отметило его 30-летний юбилей. Поэтому, чтобы правильно понимать и трактовать современные варианты SQL, нужно знать историю языка хотя бы в общих чертах.

15.1.1. Краткая история языка SQL

Язык SQL, предназначенный для взаимодействия с базами данных, появился в середине 70-х гг. (первые публикации датируются 1974 г.) и был разработан в компании IBM в рамках проекта экспериментальной реляционной СУБД System R. Исходное название языка SEQUEL (Structured English Query Language) только частично отражало суть этого языка. Конечно, язык был ориентирован главным образом на удобную и понятную пользователям формулировку запросов к реляционным БД. Но, в действительности, он почти с самого начала являлся полным языком БД, обеспечивающим помимо средств формулирования запросов и манипулирования БД следующие возможности:

- средства определения и манипулирования схемой БД;
- средства определения ограничений целостности и триггеров;
- средства определения представлений БД;
- средства определения структур физического уровня, поддерживающих эффективное выполнение запросов;
- средства авторизации доступа к отношениям и их полям[67)];
- средства определения точек сохранения транзакции и выполнения фиксации и откатов транзакций.

В языке отсутствовали средства явной синхронизации доступа к объектам БД со стороны параллельно выполняемых транзакций: с самого начала предполагалось, что необходимую синхронизацию неявно выполняет СУБД.

В настоящее время язык SQL реализован во всех коммерческих реляционных СУБД и почти во всех СУБД, которые изначально основывались не на реляционном подходе. Все компании-производители провозглашают соответствие своей реализации стандарту SQL, и на самом деле реализованные диалекты SQL очень близки. Этого удалось добиться не сразу.

Наиболее близки к System R были две системы компании IBM – SQL/DS и DB2[68)]. Разработчики обеих систем использовали опыт проекта System R, а СУБД SQL/DS напрямую основывалась на программном коде System R. Отсюда предельная близость диалектов SQL, реализованных в этих системах, к SQL System R. Из SQL System R были удалены только те части, которые были недостаточно проработаны (например, точки сохранения) или реализация которых вызывала слишком большие технические трудности (например, ограничения целостности и триггеры). Можно назвать этот путь к коммерческой реализации SQL движением сверху вниз.

Другой подход применялся в таких системах, как Oracle, Informix и Sybase. Несмотря на различие в способах разработки систем, реализация SQL везде происходила «снизу вверх». В первых выпущенных на рынок версиях этих систем использовалось ограниченное подмножество SQL System R. В частности, в первой известной нам реализации SQL в СУБД Oracle в операторах выборки не допускалось использование вложенных подзапросов и отсутствовала возможность формулировки запросов с соединениями нескольких отношений.

Тем не менее, несмотря на эти ограничения и на очень слабую, на первых порах, эффективность СУБД, ориентация компаний на поддержку разных аппаратных платформ и заинтересованность пользователей в переходе к реляционным системам позволили компаниям добиться коммерческого успеха и приступить к совершенствованию своих реализаций. В текущих версиях Oracle, Informix, Sybase и Microsoft SQL Server поддерживаются достаточно мощные диалекты SQL, хотя реализация иногда вызывает сомнения.[69)]

Особенностью большинства современных коммерческих СУБД, затрудняющей сравнение существующих диалектов SQL, является отсутствие единообразного описания языка. Обычно описание разбросано по разным руководствам и перемешано с описанием специфических для данной системы языковых средств, не имеющих прямого отношения к SQL. Тем не менее, можно сказать, что базовый набор операторов SQL, включающий операторы определения схемы БД, выборки и манипулирования данными, авторизации доступа к данным, поддержки встраивания SQL в языки программирования и операторы динамического SQL, в коммерческих реализациях устоялся и более или менее соответствует стандарту.

Деятельность по стандартизации языка SQL началась практически одновременно с появлением его первых коммерческих реализаций. В 1982 г. комитету по базам данных Американского национального института стандартов (ANSI) было поручено разработать спецификацию стандартного языка реляционных баз данных. Первый документ из числа имеющихся у автора проектов стандарта датирован октябрем 1985 г. и является уже не первым проектом стандарта ANSI. Стандарт был принят ANSI в 1986 г., а в 1987 г. одобрен Международной организацией по стандартизации (ISO). Этот стандарт принято называть *SQL/86*.

Понятно, что в качестве основы стандарта нельзя было использовать SQL System R. Во-первых, этот вариант языка не был должным образом технически проработан. Во-вторых, его слишком сложно было бы реализовать (кто знает, как бы сложилась судьба SQL, если бы все идеи проекта System R были реализованы полностью). Поэтому за основу был взят диалект языка SQL, сложившийся в IBM к началу 1980-х гг. В сущности, этот диалект представлял собой технически проработанное подмножество SQL System R.

К 1989 г. стандарт SQL/86 был несколько расширен, и был подготовлен и принят следующий стандарт, получивший название ANSI/ISO *SQL/89*. Анализ доступных документов показывает, что процесс стандартизации SQL происходил очень сложно с использованием не только научных доводов. В результате SQL/89 во многих частях имеет чрезвычайно общий характер и допускает очень широкое толкование. В этом стандарте полностью отсутствуют такие важные разделы, как манипулирование схемой БД и динамический SQL. Многие важные аспекты языка в соответствии со стандартом определяются в реализации.[70)]

Возможно, наиболее важными достижениями стандарта SQL/89 являются четкая стандартизация синтаксиса и семантики операторов выборки данных и манипулирования данными и фиксация средств ограничения целостности БД. Были специфицированы средства определения первичного и внешних ключей отношений и так называемых проверочных ограничений целостности, которые представляют собой подмножество немедленно проверяемых ограничений целостности SQL System R. Средства определения внешних ключей позволяют легко формулировать требования так называемой ссылочной целостности БД. Это распространенное в реляционных БД требование можно было сформулировать и на основе общего механизма ограничений целостности SQL System R, но формулировка на основе понятия внешнего ключа более проста и понятна.

Осознавая неполноту стандарта SQL, на фоне завершения разработки этого стандарта специалисты различных компаний начали работу над стандартом SQL2. Эта работа также длилась несколько лет, было выпущено множество проектов стандарта, пока наконец в марте 1992 г. не был принят окончательный проект стандарта (SQL/92). Этот стандарт существенно полнее стандарта SQL/89 и охватывает практически все аспекты, необходимые для реализации приложений: манипулирование схемой БД, управление транзакциями (появились точки сохранения) и сессиями (сессия – это последовательность транзакций, в пределах

которой сохраняются временные отношения), подключения к БД, динамический SQL. Наконец, были стандартизованы отношения-каталоги БД, что вообще-то не связано непосредственно с языком, но очень сильно влияет на реализацию.[71)]

В 1995 г. стандарт был дополнен спецификацией интерфейса уровня вызова (Call-Level Interface – *SQL/CLI*). SQL/CLI представляет собой набор спецификаций интерфейсов процедур, вызовы которых позволяют выполнять динамически задаваемые операторы SQL. По сути дела, SQL/CLI представляет собой альтернативу динамическому SQL. Интерфейсы процедур определены для всех основных языков программирования: C, Ada, Pascal, PL/1 и т. д. Следует заметить, что стандарт SQL/CLI послужил основой для создания повсеместно распространенных сегодня интерфейсов ODBC (Open Database Connectivity) и JDBC (Java Database Connectivity).

В 1996 г. к стандарту SQL/92 был добавлен еще один компонент – *SQL/PSM* (Persistent Stored Modules). Основная цель этой спецификации состоит в том, чтобы стандартизировать способы определения и использования хранимых процедур, т. е. специальным образом оформленных программ, включающих операторы SQL, которые сохраняются в базе данных, могут вызываться приложениями и выполняются внутри СУБД.

Незадолго до завершения работ по определению стандарта SQL2 была начата разработка стандарта SQL3. Первоначально планировалось завершить проект в 1995 г. и включить в язык некоторые объектные возможности: определяемые пользователями типы данных, поддержку триггеров, поддержку темпоральных свойств данных и т. д. Реально работу над новым стандартом удалось частично завершить только в 1999 г., и по этой причине (а также в связи с проблемой 2000 года) стандарт получил название SQL:1999.

Приведем краткую характеристику текущего состояния стандарта SQL:1999 и перспектив его развития. Прежде всего, заметим, что каждый новый вариант стандарта языка SQL был существенно объемнее предыдущих версий. Так, если стандарт SQL/89 занимал около 600 страниц, то объем SQL/92 составлял на 300 с лишним страниц больше. Самые первые проекты SQL3 занимали около 1500 страниц. Это вполне естественно, потому что язык усложняется, а его спецификации становятся более детальными и точными. Но разработчики SQL3 пришли к выводу, что при таких объемах стандарта вероятность его принятия и последующей успешной поддержки заметно уменьшается. Поэтому было принято решение разбить стандарт на относительно независимые части, которые можно было бы разрабатывать и поддерживать по отдельности.

В 1999 г. были приняты пять первых частей стандарта SQL:1999. Первая часть (SQL/Framework) посвящена описанию концептуальной структуры стандарта. В этой части приводится развернутая аннотация следующих четырех частей и формулируются требования к реализациям, претендующим на соответствие стандарту.

Вторая часть SQL:1999 (SQL/Foundation) образует базис стандарта. Вводится система типов языка, формулируются правила определения функциональных зависимостей и возможных ключей, определяются синтаксис и семантика основных операторов SQL:

- операторов определения и манипулирования схемой базы данных;
- операторов манипулирования данными;
- операторов управления транзакциями;
- операторов управления подключениями к базе данных и т. д.

Третью часть занимает уточненная по сравнению с SQL/92 спецификация SQL/CLI. В

четвертой части специфицируется SQL/PSM – синтаксис и семантика языка определения хранимых процедур. Наконец, в пятой части – SQL/Bindings – определяются правила связывания SQL для стандартных версий языков программирования FORTRAN, COBOL, PL/1, Pascal, Ada, C и MUMPS.

В стандарт SQL:1999 должны были войти еще несколько частей. Среди них спецификации следующих средств:

- управление распределенными транзакциями (SQL/Transaction);
- поддержка темпоральных свойств данных (SQL/Temporal);
- управление внешними данными (SQL/MED);
- связывание с объектно-ориентированными языками программирования (SQL/OLB);
- поддержка оперативной аналитической обработки (SQL/OLAP).

В конце 2003 г. был принят и опубликован новый вариант международного *стандарта SQL: 2003*. Многие специалисты считали, что в варианте стандарта, следующем за SQL:1999, будут всего лишь исправлены неточности SQL:1999. Но на самом деле, в SQL:2003 специфицирован ряд новых и важных свойств, часть из которых мы затронем в этом курсе.

Претерпела некоторые изменения общая организация стандарта. Стандарт SQL:2003 состоит из следующих частей:

- 9075-1, SQL/Framework;
- 9075-2, SQL/Foundation;
- 9075-3, SQL/CLI;
- 9075-4, SQL/PSM;
- 9075-9, SQL/MED;
- 9075-10, SQL/OLB;
- 9075-11, SQL/Schemata;
- 9075-13, SQL/JRT;
- 9075-14, SQL/XML.

Части 1-4 и 9-10 с необходимыми изменениями остались такими же, как и в SQL:1999. Часть 5 (SQL/Bindings) перестала существовать; соответствующие спецификации включены в часть 2. Раздел части 2 SQL:1999, посвященный информационной схеме, выделен в отдельную часть 11. Появились две новые части – 13 и 14. Часть 13 полностью называется «SQL Routines and Types Using the Java Programming Language» («Использование подпрограмм и типов SQL в языке программирования Java»). Появление такой части стандарта оправдано повышенным вниманием к языку Java со стороны ведущих производителей SQL-ориентированных СУБД. Наконец, последняя часть SQL:2003 посвящена спецификациям языковых средств, позволяющих работать с XML-документами в среде SQL.

На мой взгляд, текущее состояние процесса стандартизации языка SQL отражает текущее состояние технологии SQL-ориентированных баз данных. Ведущие поставщики соответствующих СУБД (сегодня это компании IBM, Oracle и Microsoft) стараются максимально быстро реагировать на потребности и конъюнктуру рынка и расширяют свои продукты все новыми и новыми возможностями. Очевидна потребность в стандартизации соответствующих языковых средств, но процесс стандартизации явно не поспевает за происходящими изменениями.

15.1.2. Структура языка SQL

В данной лекции мы начинаем систематически описывать базовые механизмы языка SQL. Чтобы пояснить, о какой части языка пойдет речь в этой и следующих лекциях, обратимся к рис. 15.1.

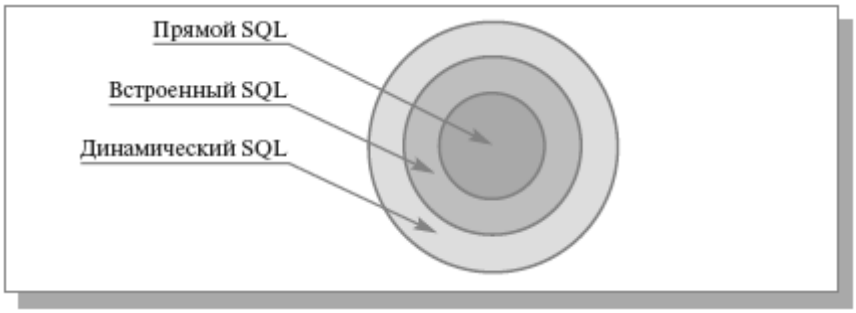


Рис. 15.1. Один из способов разделения языка SQL на уровни

Язык SQL, соответствующий последним стандартам SQL:2003, SQL:1999 (и даже SQL/92), это очень богатый и сложный язык, все возможности которого трудно сразу осознать и тем более понять. Поэтому приходится разбивать язык на уровни, или слои, такие, что каждый уровень языка включает все конструкции, входящие в более низкие уровни. В стандарте определяется несколько способов разбиения языка на уровни. В одной из классификаций язык разбивается на *базовый (entry)*, *промежуточный (intermediate)* и *полный (full)* уровни.

Эта классификация ориентирована, прежде всего, на производителей СУБД, в которых поддерживается SQL. Реализация базового уровня языка является обязательным условием хотя бы какого-то соответствия стандарту. Реализация промежуточного уровня желательна, и обычно именно такой уровень языка поддерживается ведущими компаниями-производителями SQL-ориентированных СУБД. Наконец, полный уровень языка является целью, к достижению которой следует стремиться. В данной классификации критерием отнесения той или иной возможности языка к некоторому уровню является оцениваемая создателями стандарта SQL (большая часть которых является сотрудниками ведущих компаний, производящих SQL-ориентированные СУБД) техническая сложность реализации этой возможности. Конечно, такая классификация важна и для программистов приложений баз данных, но только для того, чтобы оценить реальные возможности конкретной СУБД. Для понимания языка SQL это разбиение на уровни несущественно.

Другая классификация показана на рис. 15.1. Среди всех конструкций языка SQL можно выделить такие конструкции, которые можно было использовать при *прямом (direct)* взаимодействии конечного пользователя с СУБД (например, в интерактивном режиме). В некотором смысле этот уровень также является базовым, поскольку соответствующие средства языка в наибольшей степени отражают его ориентированность на работу с мультимножествами. На следующем уровне, уровне *встраиваемого (embedded)* SQL, язык расширяется конструкциями, позволяющими использовать возможности прямого SQL в программах, написанных на традиционных языках программирования. Наконец, на уровне *динамического (dynamic)* SQL во встраиваемый SQL добавляются конструкции, позволяющие приложениям обращаться к СУБД с конструкциями прямого SQL, которые динамически образуются во время выполнения программы.

Нам кажется, что вторая классификация является более полезной для читателя, постигающего основы языка SQL. По нашему мнению, дополнительные возможности, присутствующие во встраиваемом и в динамическом SQL, не слишком сильно влияют на модельное представление языка. Конечно, возможности встраиваемого и динамического

SQL необходимо хорошо знать разработчикам приложений SQL-ориентированных баз данных. Но поскольку задачей этого курса не является обучение использованию языка SQL при программировании приложений баз данных, мы не будем затрагивать эти темы. Обратимся к прямому SQL, причем не в полном объеме стандартов SQL:2003 и SQL:1999 (этого не позволяет сделать объем курса). Обсудим только наиболее важные аспекты.

В этой лекции обсуждаются основные аспекты системы типов данных языка SQL и средства определения доменов.

Замечание: Лекции, посвященные языку SQL, опираются, главным образом, на стандарт SQL:1999. В тех случаях, когда будут упоминаться дополнительные возможности, специфицированные в наиболее свежей версии стандарта – SQL:2003, мы будем явно на это указывать. Поэтому здесь мы используем терминологию стандарта (*таблицы, строки, столбцы* и т. д.).[72)]

---

67 В этом абзаце применяется терминология, которая использовалась в публикациях, посвященных System R.

68 Как это ни странно, компания IBM, имевшая уникальный и положительный опыт реализации экспериментальной реляционной СУБД System R, не стала первой компанией, выпустившей на рынок коммерческую реляционную СУБД. Компанию IBM опередила на два года незадолго до того образованная компания Oracle, выпустившая свою первую систему в 1979 г. Современные эксперты по разному объясняют причины этой «заторможенности» IBM, но, по всей видимости, основная причина кроется в традиционном консерватизме руководства компании.

69 Например, одной из выигрышных черт SQL System R являлось то, что в одной транзакции разрешалось комбинировать все возможные операторы SQL. Поскольку технически это обеспечить достаточно трудно, почти во всех сегодняшних SQL-ориентированных СУБД имеются ограничения на состав операторов, которые можно выполнять в одной транзакции.

70 Это практически обесценивает стандарт с точки зрения программистов приложений баз данных, поскольку не дает возможности создавать приложения, не привязанные к особенностям конкретных СУБД.

71 Среди прочих достижений System R нельзя не отметить то, что в базах данных, управляемых этой СУБД, хранились как данные, так и метаданные – описатели отношений, их полей, представлений, ограничения целостности и т.д. Для хранения метаданных использовались специальные служебные отношения, которые стали называть отношениями-каталогами. Из отношений-каталогов можно было выбрать данные с помощью обычных средств языка SQL. Конечно, организация служебных данных – это вопрос реализации SQL, но этот вопрос непосредственно касается потенциальных пользователей SQL-ориентированных СУБД, и поэтому стандартизация представления пользователю отношений-каталогов (в стандарте SQL, информационной схемы базы данных) является исключительно важным делом.

72 К сожалению, приходится использовать термин строка в двух смыслах: строка таблицы (table row) и символьная или битовая строка (character or bit string). Постараемся обеспечить правильное понимание смысла термина в контексте его использования.

Лекция 16. Средства определения базовых таблиц и ограничений целостности

16.1. Введение

Как мы уже отмечали ранее, к спецификации языка SQL можно относиться как к спецификации некоторой модели данных, в определенных аспектах близкой к реляционной модели. Мы стремимся к тому, чтобы порядок лекций, посвященных языку SQL, способствовал правильному пониманию именно этой модели, а не технических тонкостей языка. Предыдущая лекция посвящалась тому, что (т. е. данные каких типов) может храниться в SQL-ориентированной базе данных.

Теперь следует понять, где хранятся эти данные. Как и в реляционной модели данных, в модели SQL поддерживается единственная родовая структура данных, называемая в данном случае *базовой таблицей*. В первом из двух основных разделов лекции обсуждаются средства языка SQL, предназначенные для определения, изменения определения и отмены определения базовых таблиц.

Понятие базовой таблицы родственно понятию отношения: можно считать, что базовая таблица обладает заголовком[94)], в котором содержатся различаемые имена столбцов и их типы данных (заголовок базовой таблицы является множеством и представляет собой близкий аналог заголовка отношения), и телом, включающим строки, которые соответствуют заголовку таблицы (казалось бы, здесь мы имеем аналоги тела отношения и кортежей). Но коренное отличие базовой таблицы от истинного отношения состоит в том, что тело таблицы не обязательно является множеством. Среди строк тела таблицы могут встречаться дубликаты, и в общем случае тело базовой таблицы SQL представляет собой мультимножество строк.

Забегая вперед (см. лекцию 19), следует заметить, что порождаемые таблицы SQL, которые формируются при выполнении запросов к SQL-ориентированной базе данных, еще более отдаляют SQL от реляционной модели. В таких таблицах может отсутствовать и правильно сформированный заголовок (могут иметься одноименные столбцы).

Почему же, понимая принципиальные отклонения языка SQL от реляционной модели данных, мы включили эти две темы в один курс и, более того, иногда неформально называем SQL языком реляционных баз данных? Тому есть несколько причин.

- Во-первых, используя язык SQL, можно не нарушать предписаний реляционной модели, и тогда к «правильно построенной» SQL-ориентированной базе данных применимы все фундаментальные результаты теории реляционных баз данных, включая принципы проектирования на основе нормализации.
- Во-вторых, полностью отвергая родство языка SQL с реляционной моделью данных, мы выступали бы против установившихся исторических традиций. Этот язык возник около 30 лет тому назад во время реализации в компании IBM проекта по созданию экспериментальной СУБД System R, основной целью которого являлось обоснование практической реализации реляционного подхода к организации баз данных. Так что исторически SQL базировался на реляционной модели данных (возможно, не совсем верно понятой и/или воплощенной).
- Наконец, по нашему мнению, в области информационной технологии любой практически используемый инструмент не может быть полностью свободен от

компромиссов. Идеологически чистые решения возможны только в научно-экспериментальной работе. «Великий и ужасный» язык SQL – это порождение ряда компромиссов между теорией, практикой и маркетинговой деятельностью. Этот язык является настолько реляционным, насколько это понадобилось потребителям коммерческих СУБД, прямо или косвенно финансировавшим разработку языка.

В операторе SQL `CREATE TABLE` специфицируются не только столбцы таблицы, но и ограничения целостности, которым должны удовлетворять данные, хранящиеся в базовой таблице. Эти ограничения являются частным случаем ограничений базы данных целиком, для определения которых, а также изменения и ликвидации определений имеются специальные операторы. Обсуждению этих средств посвящен второй основной раздел этой лекции.

---

94 В SQL не используются термины заголовок и тело таблицы. Здесь мы временно пользуемся этой терминологией только для целей сопоставления модели SQL с реляционной моделью данных.

Лекция 17. Общая характеристика оператора SELECT и организация списка ссылок на таблицы в разделе FROM

17.1. Введение

В этой и следующих трех лекциях рассматривается важнейший оператор языка SQL – оператор `SELECT`, предназначенный для выборки данных из SQL-ориентированной базы данных. Этот оператор имеет довольно сложную и развитую структуру, но, по нашему мнению, его необходимо знать любому специалисту, так или иначе связанному с использованием баз данных; поэтому в нашем курсе ему уделяется так много внимания. Первая лекция носит подготовительный характер. В ней мы рассматриваем виды скалярных выражений, используемые, прежде всего, в конструкциях оператора `SELECT`, обсуждаем базовую семантику выполнения этого оператора и анализируем принципы и разновидности указания таблиц, из которых производится выборка данных.

Несмотря на то, что язык SQL является полным языком баз данных, включающим множество разнообразных средств определения схемы, ограничения и поддержки целостности базы данных, поддержки администрирования, заполнения и модификации таблиц базы данных, поддержки разработки приложений и т. д., для подавляющего большинства пользователей этот язык остается языком запросов, т. е. языком, позволяющим формулировать произвольно сложные и точные декларативные запросы к базе данных.

Как отмечалось в конце предыдущей лекции, структура стандарта языка SQL фактически не позволяет описать одну часть языка (в частности, средства запросов) в отрыве от других частей. Тем не менее, полагая, что средства выборки данных составляют наиболее интересную и практически значимую часть языка, мы выделили для их рассмотрения несколько отдельных лекций.

Напомним, что в этом курсе мы ограничиваемся базовым подмножеством SQL:1999 и SQL:2003 («прямым SQL») и даже это подмножество описываем не в полном объеме стандарта.

Кроме того, в данной лекции мы не будем точно придерживаться порядка введения понятий и синтаксических конструкций, принятого в стандарте языка. Мы начнем с некоторой общей картины, дающей представление об операторе выборки, а затем будем постепенно уточнять ее.

17.2. Скалярные выражения

Скалярное выражение[115)] – это выражение, вырабатывающее результат некоторого типа, специфицированного в стандарте. Скалярные выражения являются основой языка SQL, поскольку, хотя это реляционный язык, все условия, элементы списков выборки и т. д. базируются именно на скалярных выражениях. В SQL:1999 имеется несколько разновидностей скалярных выражений. К числу наиболее важных разновидностей относятся численные выражения; выражения со значениями-строками символов; выражения со значениями даты-времени; выражения со значениями-временными интервалами; булевские выражения. Мы не будем слишком глубоко вникать в тонкости, но тем не менее приведем некоторые базовые спецификации и пояснения.

Прежде чем перейти к конкретным видам скалярных выражений, рассмотрим некоторые наиболее общие языковые конструкции, на которых эти выражения базируются.

17.2.1. Общие синтаксические правила построения скалярных выражений

В SQL:2003 имеются девять разновидностей выражений в соответствии с девятью категориями типов данных, значения которых вырабатываются при вычислении выражения

```
value_expression ::=
    numeric_value_expression
  | string_value_expression
  | datetime_value_expression
  | interval_value_expression
  | boolean_value_expression
  | array_value_expression
  | multiset_value_expression
  | row_value_expression
  | user_defined_value_expression
  | reference_value_expression
```

Как уже отмечалось в начале этого раздела, мы ограничимся обсуждением первых пяти разновидностей выражений. В основе построения этих видов выражений лежит первичное выражение, определяемое следующим синтаксическим правилом:

```
value_expression_primary ::=
    unsigned_value_specification
  | column_reference
  | set_function_specification
  | scalar_subquery
  | case_expression
  | (value_expression)
  | cast_specification
```

В пределах этого курса можно считать, что спецификация беззнакового значения (`unsigned_value_specification`) – это всегда литерал соответствующего типа или вызов *ниладической* функции (например, `CURRENT_USER`)[116)]. При вычислении выражения

V для строки таблицы каждая ссылка на столбец (column_reference) этой таблицы, непосредственно содержащаяся в V, рассматривается как ссылка на значение данного столбца в данной строке. Агрегатные функции (*функции над множествами* – set_function_specification) обсуждаются в лекции 19. Если первичное выражение является скалярным подзапросом (scalar_subquery, или подзапросом, результатом которого является таблица, состоящая из одной строки и одного столбца) и результат подзапроса пуст, то результат первичного выражения – неопределенное значение. (*Подзапросы* обсуждаются в следующей лекции, выражения с переключателем (case_expression) рассматриваются ниже в этом разделе.)

17.2.2. Численные выражения

Численное выражение – это выражение, значение которого относится к числовому типу данных. Вот формальный синтаксис численного выражения:

```
numeric_value_expression> ::= numeric_term
   | numeric_value_expression + term
   | numeric_value_expression – term
numeric_term ::= numeric_factor
   | numeric_term * numeric_factor
   | numeric_term / numeric_factor
numeric_factor ::= [ { + | – } ] numeric_primary
numeric_primary ::= value_expression_primary
   | numeric_value_function
```

Следует обратить внимание на то, что в численных выражениях SQL первичная составляющая (numeric_primary) является либо первичным выражением (см. выше), либо вызовом функции с численным значением (numeric_value_function). Из этого, в частности, следует, что в численные выражения могут входить выражения с переключателем и операции преобразования типов. Вызовы функций с численным значением определяются следующими синтаксическими правилами:

```
numeric_value_function ::=
   POSITION (character_value_expression
   IN character_value_expression)
   |{CHAR_LENGTH | CHARACTER_LENGTH }
   (string_value_expression)
   | OCTET_LENGTH (string_value_expression)
   | BIT_LENGTH (string_value_expression)
   | EXTRACT ({ datetime_field | time_zone field }
     FROM { datetime_value_expression
       | interval_value_expression })
   | CARDINALITY (array_value_expression
         | multiset_value_expression)
   | ABS (numeric_value_expression)
   | MOD (numeric_value_expression)
```

Мы достаточно подробно обсуждали функции определения позиции и длины по отношению к символьным и битовым строкам при рассмотрении соответствующих типов данных; здесь приводится только уточненный синтаксис их вызова. Функция EXTRACT извлечения поля из значений дата-время или интервал позволяет получить в виде точного числа с масштабом 0 значение любого поля (года, месяца, дня и т. д.). Какой конкретный тип точных чисел будет выбран – определяется в реализации. Функции ABS и MOD возвращают абсолютное значение числа и остаток от деления одного целого значения на другое соответственно.

17.2.3. Выражения, значениями которых являются символьные или битовые строки

Выражения символьных и битовых строк – это выражения, значениями которых являются символьные или битовые строки. Соответствующие конструкции определяются следующим синтаксисом:

```
string_value_expression ::= character_value_expression
   | bit_value_expression
character_value_expression ::= concatenation
   | character_factor
concatenation ::= character_value_expression || character_factor
character_factor ::= character_primary [ collate_clause ]
character_primary ::= value_expression_primary
   | string_value_function
bit_value_expression ::= bit_concatenation
   | bit_factor
bit_concatenation ::= bit_value_expression || bit_primary
bit_primary ::= value_expression_primary
   | string value function
```

Если не вдаваться в тонкости, смысл выражений символьных и битовых строк понятен из описания синтаксиса: единственная применимая для построения выражений операция – это конкатенация, производящая «склейку» строк-операндов. Более важно то, что первичной составляющей выражения над строками может быть как первичное скалярное выражение (см. выше), так и вызов функций, возвращающих строчные значения. Репертуар и синтаксис вызова таких функций определяются следующими правилами:

```
string_value_function ::= character_value_function
   | bit_value_function
character _value_function ::= SUBSTRING
   (character _value_expression
     FROM start_position
       [ FOR string_length ])
     | SUBSTRING (character _value_expression
       SIMILAR character _value_expression
         ESCAPE character_value_expression)
     | { UPPER | LOWER }
       (character_value_expression)
     | CONVERT (character_value_expression
       USING conversion_name)
     | TRANSLATE (character_value_expression)
       USING translation_name)
     | TRIM ([ {LEADING | TRAILING | BOTH} ]
       [ character_value_expression ]
       [ character_value_expression ])
     | OVERLAY (character_value_expression
       PLACING character_value_expression
         FROM start_position
           [ FOR string_length ])
bit _value_function ::= SUBSTRING (bit_value_expression
       FROM start_position
         [ FOR string_length ])
start_position ::= numeric_value_expression
string length ::= numeric_value_expression
```

Основные полезные функции – выделение подстроки (`SUBSTRING`) и замена малых букв на заглавные и наоборот (`UPPER` и `LOWER`) – мы упоминали при рассмотрении типов символьных и битовых строк. Обсуждение функции `SUBSTRING ... SIMILAR ... ESCAPE` отложим до следующей лекции. Как видно из описания синтаксиса функций,

возвращающих строчные значения, для символьных строк имеются еще четыре функции: CONVERT, TRANSLATE, TRIM и OVERLAY. По смыслу все они очень просты. Функция CONVERT меняет кодировку символов в заданной строке, причем набор символов не меняется. Способ задания правил перекодировки определяется в реализации. Функция TRANSLATE, наоборот, в соответствии с правилами трансляции «переводит» текстовую строку на другой язык (используя набор символов целевого алфавита). Кодировка не меняется. Функция TRIM «отсекает» последовательности указанного символа в начале, в конце или в конце и начале заданной строки. Наконец, функция OVERLAY заменяет указанную подстроку первого операнда строкой, заданной в качестве второго операнда.

17.2.4. Выражения даты-времени

К *выражениям даты-времени* мы относим выражения, вырабатывающие значения типа дата-время и интервал. Выражения даты-времени определяются следующими синтаксическими правилами:

```
datetime_value_expression ::=
   datetime_term
     | interval_value_expression + datetime term
     | datetime_value_expression + interval term
     | datetime value expression – interval term
datetime_term ::= datetime_primary
   [ AT { LOCAL | TIME ZONE interval_value_expression } ]
datetime_primary ::= value_expression_primary
     | datetime_value_function
```

Как видно из описания синтаксиса, сами выражения строятся очень просто – на основе обычных арифметических операций. Снова более интересны первичные составляющие – вызовы функций, возвращающих значение дата-время. Эти вызовы определяются следующим синтаксисом:

```
datetime_value_function ::= CURRENT_DATE
   | CURRENT_TIME [ (precision) ]
   | LOCALTIME [ (precision) ]
   | CURRENT_TIMESTAMP [ (precision) ]
   | LOCALTIMESTAMP [ (precision) ]
```

Видимо, приведенные синтаксические правила не нуждаются в комментариях: можно получить текущую дату, а также текущее время с желаемой точностью. Отличие функций LOCALTIME и LOCALTIMESTAMP от CURRENT_TIME и CURRENT_TIMESTAMP, соответственно, состоит в том, что первая пара функций не возвращает смещение локального времени от Гринвича.

Синтаксис выражений со значениями типа интервал определяется следующими правилами:

```
interval_value_expression ::=
     interval_term
   | interval_value_expression + interval term
   | interval_value_expression – interval term
   | (datetime value expression – datetime term)
     interval_qualifier
interval_term ::= interval_factor
   | interval_term * numeric_factor
   | interval_term / numeric_factor
   | numeric_term * interval_factor
```

```
interval_factor ::= [ { + | - } ]
     interval_primary [ <interval qualifier> ]
interval_primary ::= value_expression_primary
   | interval_value_function
```

Как видно из приведенных правил, выражения со значениями типа интервал устроены очень просто; почти вся содержательная информация была приведена при обсуждении соответствующего типа данных. Стоит только заметить, что квалификатор интервала указывается для того, чтобы явно специфицировать единицу измерения интервала. Поддерживается только одна функция ABS (абсолютное значение), аргументом которой является выражение со значением типа интервал.

17.2.5. Булевские выражения

К *булевским выражениям* относятся выражения, вырабатывающие значения булевского типа (напомним, что булевский тип языка SQL содержит три логических значения – true, false и unknown). Булевские выражения определяются следующими синтаксическими правилами:

```
boolean_value_expression ::= boolean_term
   | boolean_value_expression OR boolean_term
boolean_term ::= boolean_factor
   | boolean_term AND boolean_factor
boolean_factor ::= [ NOT ] boolean_test
boolean_test ::= boolean_primary [ IS [ NOT ] truth_value ]
truth_value ::= TRUE | FALSE | UNKNOWN
boolean_primary ::= predicate
   | (boolean_value_expression)
   | value_expression_primary
```

Выражения вычисляются слева направо с учетом приоритетов операций (наиболее высокий приоритет имеет унарная операция NOT, следующим уровнем приоритета обладает «мультипликативная» операция конъюнкции AND, и самый низкий приоритет у «аддитивной» операции дизъюнкции OR) и круглых скобок. Операции IS и IS NOT определяются следующими таблицами истинности:

IS	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	FALSE
FALSE	FALSE	TRUE	FALSE
UNKNOWN	FALSE	FALSE	TRUE

IS NOT	TRUE	FALSE	UNKNOWN
TRUE	FALSE	TRUE	TRUE
FALSE	TRUE	FALSE	TRUE
UNKNOWN	TRUE	TRUE	FALSE

17.2.6. Выражения с переключателем

Выражения с переключателем в некотором смысле ортогональны рассмотренным выше видам выражений, поскольку разные выражения с переключателем могут вырабатывать значения разных типов в зависимости от типа данных элементов. Поскольку мы еще вообще не рассматривали этот вид выражений, обсудим их более подробно. Как обычно, начнем с

синтаксиса:

```
case_expression ::= case_abbreviation
   | case_specification

case_abbreviation ::= NULLIF (value_expression , value_expression)
   | COALESCE (value_expression_comma_list)
case specification ::= simple_case | searched_case

simple_case ::= CASE value_expression simple_when_clause_list
     [ ELSE value_expression ] END

searched_case ::= CASE searched_when_clause_list
     [ ELSE value_expression ] END
simple_when_clause ::= WHEN value_expression
     THEN value_expression
searched_when_clause ::= WHEN conditional_expression
     THEN value_expression
```

Наиболее общим видом выражения с переключателем является выражение с поисковым переключателем (`searched_case`). Правила вычисления выражений этого вида состоят в следующем. Вычисляется логическое выражение, указанное в первом разделе `WHEN` списка (`searched_when_clause_list`). Если значение этого логического выражения равняется *true*, то значением всего выражения с поисковым переключателем является значение выражения, указанного в первом разделе `WHEN` после ключевого слова `THEN`. Иначе аналогичные действия производятся для второго раздела `WHEN` и т. д. Если ни для одного раздела `WHEN` при вычислении логического выражения не было получено значение *true*, то значением всего выражения с поисковым переключателем является значение выражения, указанного в разделе `ELSE`. Типы всех выражений, значения которых могут являться результатом выражения с поисковым переключателем, должны быть совместимыми, и типом результата является «наименьший общий» тип набора типов выражений-кандидатов на выработку результата[117)]. Если в выражении отсутствует раздел `ELSE`, предполагается наличие раздела `ELSE NULL`.

В выражении с простым переключателем (`simple_case`) тип данных операнда переключателя (выражения, непосредственно следующего за ключевым словом `CASE`, назовем его `CO` – `Case Operand`) должен быть совместим с типом данных операнда каждого варианта (выражения, непосредственно следующего за ключевым словом `WHEN`; назовем `WO` – `When Operand`). Выражение с простым переключателем

```
CASE CO WHEN WO1 THEN result1
   WHEN WO2 THEN result2
   . . . . . .
   WHEN WOn THEN resultn
   ELSE result
END
```

эквивалентно выражению с поисковым переключателем

```
CASE WHEN CO = WO1 THEN result1
   WHEN CO = WO2 THEN result2
   . . . . . .
   WHEN CO = WOn THEN resultn
   ELSE result
END
```

Выражение NULLIF (V1, V2) эквивалентно следующему выражению с переключателем:

```
CASE WHEN V1 = V2 THEN NULL ELSE V1 END.
```

Выражение COALESCE (V1, V2) эквивалентно следующему выражению с переключателем:

```
CASE WHEN V1 IS NOT NULL THEN V1 ELSE V2 END.
```

Выражение COALESCE (V1, V2, . . . V*n*) для *n* ≥ 3 эквивалентно следующему выражению с переключателем:

```
CASE WHEN V1 IS NOT NULL THEN V1 ELSE COALESCE (V2,... n) END.
```

---

115 В стандарте языка SQL в качестве общего термина для обозначения таких выражений используется термин *value expression* . Однако в менее формальных публикациях обычно применяется более понятный термин *scalar expression* , для которого, вдобавок, существует адекватный русский эквивалент скалярное выражение. В этом курсе мы также предпочитаем использовать именно этот термин.

116 Другие варианты появляются во встраиваемом и динамическом SQL, а также расширении языка, предназначенного для написания кода хранимых процедур, триггеров, методов определяемых пользователями типов и т.д. В любом случае беззнаковое значение известно до начала компиляции любой содержащей его конструкции языка SQL.

117 Для набора типов T1, T2, …, Tn, будем называть тип T, если значения каждого из типов T1, T2, …, Tn неявно приводимы к типу T, и не существует типа T', такого, что значения типов T1, T2, …, Tn неявно приводимы к типу T', и значения типа T' неявно приводимы к типу T.

Лекция 18. Предикаты раздела WHERE оператора SELECT

18.1. Введение

В этой лекции мы продолжим рассматривать механизм выборки данных языка SQL – оператор SELECT. Лекция целиком посвящена видам условных выражений, которые могут содержаться в разделе WHERE оператора выборки. Определяются и иллюстрируются на примерах запросов все виды предикатов, специфицированных в стандарте SQL:1999.

Конструкции оператора SELECT языка SQL в значительной степени ортогональны. В частности, выбор способа указания ссылки на таблицы в разделе FROM никак не влияет на выбор варианта формирования условия выборки в разделе WHERE. Это полезное свойство языка позволяет нам абстрагироваться от обсуждавшегося в предыдущей лекции многообразия способов указания ссылки на таблицу и сосредоточиться на возможностях формирования запросов при использовании различных предикатов, допускаемых стандартом SQL:1999 в логических выражениях раздела WHERE.

В стандарте SQL:1999 специфицированы 12 разновидностей предикатов, причем некоторые из них в действительности представляют собой семейства (например, под общим названием предиката сравнения скрываются шесть видов предикатов). Набор допустимых предикатов в SQL явно избыточен, но тем не менее в языке SQL имеется явная тенденция расширения этого набора. В частности, в SQL:2003 в связи с введением генератора типов мультимножеств в дополнение ко всем разновидностям предикатов SQL:1999 появилось три новых вида предикатов: предикаты для проверки того, что заданное значение является элементом мультимножества (`MEMBER`); что одно мультимножество входит в другое мультимножество (`SUBMULTISET`) и что мультимножество не содержит дубликаты (`IS A SET`). В этом курсе мы не приводим подробного описания этих видов предикатов по нескольким причинам:

- введение конструктора типов мультимножеств в стандарте SQL:2003 не означает, что достигнута общая цель разработчиков стандарта SQL по обеспечению полного набора типов коллекций; по всей видимости, в будущих версиях стандарта появятся дополнительные конструкторы типов коллекций, и набор видов предикатов изменится;
- предикаты с мультимножествами трудно пояснять и иллюстрировать в отрыве от других объектно-реляционных средств языка SQL;
- включение подобного материала в данную лекцию заметно увеличило бы ее объем и затруднило понимание более традиционных конструкций.

В лекции содержится много примеров запросов с использованием различных видов предикатов. Для полного усвоения материала требуется тщательно проанализировать эти примеры.

Лекция 19. Группировка и условия раздела HAVING, порождаемые и соединенные таблицы

19.1. Введение

В предыдущих двух лекциях мы обсудили допускаемые в стандарте SQL виды ссылок на таблицы в разделе `FROM` оператора `SELECT` и подробно, с многочисленными примерами, рассмотрели возможные способы построения условных выражений раздела `WHERE`. Данную лекцию мы начинаем с анализа возможностей и целесообразности использования в запросах разделов `GROUP BY` и `HAVING`. Соответствующий раздел 19.2 «Агрегатные функции, группировка и условия раздела `HAVING`» формально похож на раздел 18.2 «Логические выражения раздела `WHERE`» лекции 18: обсуждаются виды предикатов, которые можно использовать в условных выражениях раздела `HAVING`, и приводятся иллюстрирующие примеры. Но в действительности мы преследуем большую цель: показать, что во многих случаях разделы `GROUP BY` и `HAVING` являются избыточными; запрос можно сформулировать более понятным образом без их использования. Применение разделов `GROUP BY` и `HAVING` оказывается действительно полезным, а иногда и необходимым, в тех случаях, когда в запросе присутствует несколько вызовов агрегатных функций на группах строк.

После обсуждения разделов `GROUP BY` и `HAVING` можно будет считать, что мы полностью рассмотрели базовые конструкции оператора выборки (раздел `ORDER BY` не заслуживает

дополнительного обсуждения). Поэтому в разделах 19.3. «Ссылки на порождаемые таблицы в разделе FROM» и 19.4. «Более сложные конструкции оператора выборки» мы возвращаемся к отложенным в лекции 17 темам порождаемых таблиц, соединенных таблиц и порождаемых таблиц с горизонтальной связью.

В обычных порождаемых таблицах SQL нет ничего особенного. По всей видимости, возможность указывать в разделе FROM выражения запросов, а не только ссылки на базовые или представляемые таблицы, была введена в SQL на основе следующих естественных соображений. Результатом вычисления выражения запросов в SQL является таблица. Следовательно, в любой конструкции языка, где может присутствовать ссылка на таблицу SQL, следует допустить присутствие выражения запросов. Одновременное наличие возможностей определения представляемых таблиц, указания именованного выражения запросов в разделе WITH и указания выражения запросов порождаемой таблицы непосредственно в списке раздела FROM, очевидно, является избыточным.

Соединенные таблицы появились еще в стандарте SQL/92, и внедрение в стандарт SQL этой возможности было действительно обоснованным. В соответствии с традиционной общей семантикой оператора SELECT в нем вообще не предусматривались явные средства для выражения потребности в соединении двух или более таблиц. Наличие возможности указывать несколько ссылок на таблицы в разделе FROM и спецификации произвольного логического выражения в разделе WHERE для ограничения расширенного декартова произведения этих таблиц позволяет выражать с помощью традиционных средств SQL соединение общего вида в смысле Кодда, и до поры до времени это считалось достаточным.

19.1.1. Внешние соединения

Но имеются два важных частных случая соединений, которые выражаются с помощью традиционных средств SQL излишне громоздко,- это естественные и внешние соединения. При наличии возможности определения внешних ключей таблицы кажется достаточно странной потребность всякий раз явно указывать в запросах условие естественного соединения. Например, во многих примерах запросов в лекции 18 присутствует условие соединения EMP.DEPT_NO = DEPT.DEPT_NO в тех случаях, когда в действительности нам требовался результат операции EMP NATURAL JOIN DEPT.

Внешние соединения были введены еще Эдгаром Коддом в 1979 г. [2.2]. В целом, основная идея этой разновидности операции соединения состояла в том, что, с одной стороны, результат операции обычного соединения двух отношений повышает информационный уровень данных, поскольку в результате операции мы имеем информационно связанные данные. Но, с другой стороны, в результирующем отношении мы теряем информацию об исходных объектах, которые оказались несвязанными и не вошли в результат соединения. Кодд придумал, как, используя неопределенные значения, определить обобщенную операцию, которая будет обладать достоинствами обычной операции соединения, не приводя к потере исходной информации. Вернее, он предложил три операции: левое внешнее соединение, правое внешнее соединение и полное (симметричное) внешнее соединение. Приведем их определения (в реляционных терминах данного курса).

Пусть имеются отношения r_1 и r_2, совместимые относительно операции взятия расширенного декартова произведения. Пусть *s* является результатом операции r_1 LEFT OUTER JOUN r_2 WHERE comp (левое внешнее соединение r_1 и r_1 по условию comp).

Тогда Hs = Hr$_1$ union Hr$_2$. Пусть tr$_1$ ∈ Br$_1$ и tr$_2$ ∈ Br$_2$. Тогда tr$_1$ union tr$_2$ ∈ Bs в том и только в том случае, когда comp (tr$_1$ union tr$_2$) = true. Если имеется кортеж tr$_1$ ∈ Br$_1$, для которого нет ни одного кортежа tr$_2$ ∈ r$_2$, такого, что comp (tr$_1$ union tr$_2$) = true, то tr$_1$ union tr$_{2null}$ ∈ Bs, где tr$_{2null}$ – кортеж, соответствующий Hr$_2$, все значения которого являются неопределенными[140)].

Пусть *s* является результатом операции r$_1$ RIGHT OUTER JOUN r$_2$ WHERE comp (правое внешнее соединение r$_1$ и r$_2$ по условию comp). Тогда Hs = Hr$_1$ union Hr$_2$. Пусть tr$_1$ ∈ Br$_1$ и tr$_2$ ∈ Br$_2$. Тогда tr$_1$ union tr$_2$ ∈ Bs в том и только в том случае, когда comp (tr$_1$ union tr$_2$) = true. Если имеется кортеж tr$_2$ ∈ Br$_2$, для которого нет ни одного такого кортежа tr$_1$ ∈ Br$_1$, что comp (tr$_1$ union tr$_2$) = true, то tr$_{1null}$ union tr$_2$ ∈ Bs, где tr$_{1null}$ – кортеж, соответствующий Hr$_1$, все значения которого являются неопределенными.

Наконец, пусть *s* является результатом операции r$_1$ FULL OUTER JOUN r$_2$ WHERE comp (полное внешнее соединение r$_1$ и r$_2$ по условию comp). Тогда Hs = Hr$_1$ union Hr$_2$. Пусть tr$_1$ ∈ Br$_1$ и tr$_2$ ∈ Br$_2$. Тогда tr$_1$ union tr$_2$ ∈ Bs в том и только в том случае, когда comp (tr$_1$ union tr$_2$) = true. Если имеется кортеж tr$_1$ ∈ Br$_1$, для которого нет ни одного кортежа tr$_2$ ∈ Br$_2$, такого, что comp (tr$_1$ union tr$_2$) = true, то tr$_1$ union tr$_{2null}$ ∈ Bs, где tr$_{2null}$ – кортеж, соответствующий Hr$_2$, все значения которого являются неопределенными. Если имеется кортеж tr$_2$ ∈ Br$_2$, для которого нет ни одного кортежа tr$_1$ ∈ Br$_1$, такого, что comp (tr$_1$ union tr$_2$) = true, то tr$_{1null}$ union tr$_2$ ∈ Bs, где tr$_{1null}$ – кортеж, соответствующий Hr$_1$, все значения которого являются неопределенными.

Понятно, что традиционными средствами SQL можно выразить все виды внешних соединений (например, с использованием переключателей), но такие запросы будут очень громоздкими. Компании-производители SQL-ориентированных СУБД пытались обеспечивать выразительные средства внешних соединений путем расширения системы обозначений для операций сравнения. Этот подход был не слишком удачным и не обеспечивал общего решения.

В стандарте языка SQL специфицирован отдельный специализированный подъязык для формирования выражений соединения таблиц. Такие выражения называются соединенными таблицами, и их можно использовать в качестве ссылок на таблицы в списке раздела FROM. Разработчики стандарта SQL не любят мельчить – в языке допускается 14 видов соединений:

- прямое соединение;
- внутреннее соединение по условию;
- внутреннее соединение по совпадению значений указанных одноименных столбцов;
- естественное внутреннее соединение;
- левое внешнее соединение по условию;
- правое внешнее соединение по условию;

- полное внешнее соединение по условию;
- левое внешнее соединение по совпадению значений указанных одноименных столбцов;
- правое внешнее соединение по совпадению значений указанных одноименных столбцов;
- полное внешнее соединение по совпадению значений указанных одноименных столбцов;
- естественное левое внешнее соединение;
- естественное правое внешнее соединение;
- естественное полное внешнее соединение;
- соединение объединением.

Во всех этих операциях нет ничего сложного, но их неформальное описание исключительно громоздко. Поэтому в разделе 19.4. «Более сложные конструкции оператора выборки» мы определяем операции на формальном уровне, а потом иллюстрируем их на примерах.

Наконец, последняя тема этой лекции относится к еще одному типу ссылок на таблицу, допускаемых в разделе `FROM`: порождаемым таблицам с горизонтальной связью. Фактически порождаемая таблица с горизонтальной связью представляет собой выражение запросов, в котором может присутствовать корреляция со строками таблиц, специфицированных в списке раздела `FROM` *слева* от данной порождаемой таблицы с горизонтальной связью. Наличие порождаемых таблиц с горизонтальной связью требует некоторого уточнения семантики выполнения раздела `FROM` оператора `SELECT`. По нашему мнению, это средство является полностью избыточным, хотя и не вредным, поскольку его реализация не должна вызывать затруднений и/или снижать эффективность системы.

---

140 Здесь мы прибегаем к компромиссу между реляционной терминологией и моделью данных SQL: конечно, в реляционной модели кортеж из неопределенных значений не может соответствовать заголовку отношения, поскольку `NULL` не является значением ни одного типа данных.

Лекция 20. Средства формулировки аналитических и рекурсивных запросов

20.1. Введение

Две темы, которым посвящается эта лекция, касаются сравнительно новых возможностей оператора `SELECT` языка SQL, впервые появившихся в стандарте SQL:1999 и открывающих возможность использования языка в приложениях, для которых ранее он не был приспособлен. Речь идет о возможностях аналитических и рекурсивных запросов. Эти темы логически не связаны, их объединяет лишь то, что соответствующие средства очень громоздки и не всегда легко понимаются. В данной краткой лекции мы не стремимся привести полное описание возможностей, специфицированных в стандарте SQL. Наша цель состоит лишь в том, чтобы в общих чертах описать подход SQL в указанных направлениях.

В аналитических приложениях обычно требуются не детальные данные, непосредственно хранящиеся в базе данных, а некоторые их обобщения, агрегаты. Например, аналитика

интересует не заработная плата конкретного человека в конкретное время, а изменение заработной платы некоторой категории людей в течение определенного промежутка времени. Если пользоваться терминологией SQL, то типичный запрос к базе данных со стороны аналитического приложения содержит раздел `GROUP BY` и вызовы агрегатных функций. Хотя в этом курсе мы почти не касаемся вопросов реализации SQL-ориентированных СУБД, из общих соображений должно быть понятно, что запросы с разделом `GROUP BY` в общем случае являются «трудными» для СУБД, поскольку для группирования таблицы, вообще говоря, требуется внешняя сортировка.

В системах баз данных, специально спроектированных в расчете на аналитические приложения, проблему обычно решают за счет явного избыточного хранения агрегированных данных (т.е. результатов вызовов агрегатных функций). Конечно, для этого требуется динамическая корректировка хранимых агрегатных значений при изменении детальных данных, но для таких специализированных баз данных это не слишком обременительно, поскольку аналитические базы данных обновляются сравнительно редко.

Однако далеко не каждое предприятие может позволить себе одновременно поддерживать оперативную базу данных для работы обычных приложений оперативной обработки транзакций (OLTP), таких, как бухгалтерские, кадровые и другие приложения, и аналитическую базу данных для приложений оперативной аналитической обработки (OLAP). Приходится выполнять аналитические приложения над детальными оперативными базами данных, и эти приложения обращаются к СУБД с многочисленными трудоемкими запросами с разделами `GROUP BY` и вызовами агрегатных функций.

Разработчики стандарта языка SQL старались одновременно решить две задачи: сократить число запросов, требуемых в аналитических приложениях, и добиться снижения стоимости запросов с разделом `GROUP BY`, обеспечивающих требуемые суммарные данные. В этой лекции мы обсудим наиболее важные, с нашей точки зрения, конструкции языка SQL, облегчающие формулировку, выполнение и использование результатов аналитических запросов: разделы `GROUP BY ROLLUP` и `GROUP BY CUBE` и новую `агрегатную функцию GROUPING`, позволяющую правильно трактовать результаты аналитических запросов при наличии неопределенных значений.

Традиционно язык SQL никогда не обладал возможностью формулировки рекурсивных запросов, где под *рекурсивным запросом* (упрощенно говоря) мы понимаем запрос к таблице, которая сама каким-либо образом изменяется при выполнении этого запроса. Напомню, что это заложено в базовую семантику оператора SQL: до выполнения раздела `WHERE` результат раздела `FROM` должен быть полностью вычислен.

Однако разработчикам приложений часто приходится решать задачи, для которых недостаточно традиционных средств формулировки запросов языка SQL: например, нахождение маршрута движения между двумя заданными географическими точками, определения общего набора комплектующих для сбора некоторого агрегата и т.д. Компании-производители SQL-ориентированных СУБД пытались удовлетворять такие потребности за счет частных решений, обладающих ограниченными рекурсивными свойствами, но до появления стандарта SQL:1999 общие стандартизованные средства отсутствовали.

Следует отметить и некоторое давление на SQL-сообщество со стороны сообщества логических систем баз данных. На основе языка логического программирования Prolog был разработан язык реляционных баз данных Datalog, обеспечивающий все необходимые средства для обычной работы с базами данных наряду с развитыми возможностями рекурсивных запросов. Требовался адекватный ответ со стороны разработчиков стандарта

SQL.

Компромиссное (не слишком красивое) решение для введения рекурсии в SQL было найдено на основе введения раздела `WITH` в выражение запроса. Только в этом разделе допускается как линейная, так и взаимная рекурсия между вводимыми порождаемыми таблицами. При этом только для линейной рекурсии обеспечиваются дополнительные возможности управления порядком вычисления рекурсивно определенной порождаемой таблицы и контроля отсутствия циклов. Следует заметить, что при чтении стандарта временами возникает впечатление, что его авторы сами не до конца еще осознали всех возможных последствий, к которым может привести использование введенных конструкций. Я думаю, что в следующих версиях стандарта следует ожидать уточнений и/или ограничений использования названных конструкций. В связи с этим в данной лекции мы ограничиваемся общими определениями рекурсивных конструкций языка SQL и обсуждением простого случая рекурсивного запроса.

Лекция 21. Средства манипулирования данными

21.1. Введение

Базы данных, по крайней мере, в приложениях категории OLTP, являются высоко динамичными объектами. В таких приложениях на две операции выборки данных в среднем приходится одна операция обновления содержимого базы данных (добавления новых данных, удаления или модификации существующих данных). Поэтому для пользователей и разработчиков OLTP-приложений средства *манипулирования данными* по важности находятся на втором месте после средств выборки данных.

В этой лекции мы обсудим средства манипулирования данными, входящие в *прямой* SQL. Заметим, что с практической точки зрения более важными являются средства манипулирования данными, выходящие за пределы прямого SQL и присутствующие во *встраиваемом* и *динамическом* SQL. Но, как мы неоднократно отмечали, в этом курсе мы не обсуждаем возможности использования SQL для создания приложений. По мнению автора, материал данной лекции полезен для общего понимания специфики операторов манипулирования данными, а расширения этих операторов, присутствующие во встраиваемом и динамическом SQL, в любом случае нужно изучать совместно с другими аспектами подобных уровней языка.

Лекция состоит из трех основных разделов. В разделе 21.2. Базовые средства манипулирования данными мы обсудим синтаксис и семантику операторов манипулирования данными, полагая, что они действуют над базовыми таблицами. В разделе 21.3. Представления, над которыми возможны операции обновления будет продемонстрировано, что в ряде случаев, специфицированных в стандарте языка SQL, операторы манипулирования данными можно применять к порождаемым таблицам и представлениям с однозначным отображением результатов действия этих операторов на соответствующие базовые таблицы. Раздел 21.4. Операции обновления баз данных и механизм триггеров посвящен механизму триггеров, которые, по существу, представляют собой «хранимые процедуры», автоматически вызываемые при возникновении соответствующих условий. Триггеры не обязательно связываются с действиями, производимыми при манипулировании данных, но, поскольку одно из основных функций этого механизма состоит в поддержании целостности баз данных, как правило, такая связь имеется. Поэтому мы включили обсуждение механизма триггеров в соответствии со

стандартом SQL именно в данную лекцию.

21.2. Базовые средства манипулирования данными

К базовым средствам манипулирования данными языка SQL относятся «поисковые» варианты операторов UPDATE и DELETE. Эти варианты называются поисковыми, потому что при задании соответствующей операции задается логическое условие, налагаемое на строки адресуемой оператором таблицы, которые должны быть подвергнуты модификации или удалению. Кроме того, в такую категорию языковых средств входит оператор INSERT, позволяющий добавлять строки в существующие таблицы. Логично начать изложение именно с оператора INSERT, поскольку, для того чтобы можно было что-либо модифицировать в таблицах или удалять из таблиц, нужно, чтобы в таблицах содержались какие-то строки.

21.2.1. Оператор INSERT для вставки строк в существующие таблицы

Общий синтаксис оператора *INSERT* выглядит следующим образом:

```
INSERT INTO table_name
     { [ (column_commalist) ] query_expression
     | DEFAULT VALUES
```

На вид синтаксические правила кажутся очень простыми, пока не вспомнишь, что обозначает синтаксическая категория query_expression (см. подраздел 17.2.1. «Общие синтаксические правила построения скалярных выражений» лекции 17). Даже если ограничиться простейшей составляющей этой конструкции (simple_table), то мы имеем следующие возможности:

```
simple_table ::= query_specification
     | table_value_constructor
     | TABLE table_name
```

Вставка всех строк указанной таблицы

Тем самым, стандарт допускает вставку в указанную таблицу всех строк некоторой другой таблицы (вариант table_name). Эта другая таблица может быть как базовой, так и представляемой. Естественно, что в последнем случае в определении представления не должны присутствовать ссылки на таблицу, в которую производится вставка. При использовании данного варианта оператора вставки число столбцов вставляемой таблицы должно совпадать с числом столбцов таблицы, в которую производится вставка, или с числом столбцов, указанных в списке column_commalist, если этот список задан. Типы данных соответствующих столбцов вставляемой таблицы и таблицы, в которую производится вставка, должны быть совместимыми. Если в операции задан список column_commalist и в нем содержатся не все имена столбцов таблицы, в которую производится вставка, то в оставшиеся столбцы во всех строках заносятся значения столбцов по умолчанию. Если для какого-либо из оставшихся столбцов значение по умолчанию не определено, при выполнении операции вставки фиксируется ошибка.

Чтобы привести пример этого варианта операции INSERT (пример 21.1), предположим, что в базе данных EMP-DEPT-PRO имеется еще одна промежуточная таблица EMP_TEMP, в

которой временно хранятся данные о служащих, проходящих испытательный срок. Пусть эта таблица имеет следующий заголовок:

EMP_TEMP:

EMP_NO : EMP_NO
EMP_NAME : VARCHAR
EMP_BDATE : DATE

В таблице EMP_TEMP хранятся не полные сведения о служащих, а именно те, которые требуются на время испытательного срока. Если выполнить операцию

```
INSERT INTO EMP (EMP_NO, EMP_NAME, EMP_BDATE) TABLE EMP_TEMP;
```

то в основной таблице EMP появятся строки, соответствующие служащим, проходившим испытательный срок. При этом в столбцах EMP_NO, EMP_NAME, EMP_BDATE этих строк будут содержаться данные, взятые из таблицы EMP_TEMP, а в столбцах EMP_SAL, DEPT_NO, PRO_NO будут находиться значения, определенные для данных столбцов по умолчанию. Конечно, поскольку столбец EMP_NO является первичным ключом таблицы EMP (по всей видимости, и таблицы EMP_TEMP), операция вставки будет успешно выполнена только в том случае, когда ограничение первичного ключа таблицы EMP не будет нарушено (конечно же, требуется выполнение и всех других ограничений целостности, определенных для таблицы EMP).

Вставка явно заданного набора строк

Теперь обратимся к варианту оператора INSERT, в котором набор вставляемых строк задается явно с использованием синтаксической конструкции table_value_constructor. Напомним синтаксические правила, определяющие эту конструкцию:

```
table_value_constructor ::=
    VALUES row_value_constructor_comma_list
row_value_constructor ::= row_value_constructor_element
    | [ ROW ] (row_value_constructor_element_comma_list)
    | row_subquery
row_value_constructor_element ::= value_expression
    | NULL | DEFAULT
```

Самый простой пример использования этого варианта оператора вставки состоит в занесении в таблицу EMP явно задаваемых данных о новом служащем (пример 21.2):

```
INSERT INTO EMP
   ROW (2445, 'Brown', '1985-04-08', 16500.00, 630, 772);
```

В этом примере явно заданы значения всех столбцов заносимой строки (как показывают синтаксические правила, ключевое слово ROW можно опустить). Возможен и такой вариант (пример 21.2.1):

```
INSERT INTO EMP
   ROW ( 2445, DEFAULT, NULL, DEFAULT, NULL, NULL);
```

В этом случае мы знаем о новом служащем очень мало, но уверены в том, что его имя и размер заработной платы должны быть назначены по умолчанию, а про дату рождения,

номер отдела и номер проекта ничего не известно. Обратите внимание, что выполнение подобной операции не нарушает ограничения целостности таблицы EMP.

Если обладать полной информацией об определении таблицы EMP, то формулировку операции примера 21.2.1 можно переписать короче следующим эквивалентным образом (пример 21.2.2):

```
INSERT INTO EMP (EMP_NO) 2445;
```

Вспомним теперь, что одной из разновидностей value_expression_primary является scalar_subquery (см. раздел 17.2 «Скалярные выражения» лекции 17). Это означает, что в список элементов конструктора строки могут входить скалярные запросы, т. е. запросы, результат выполнения которых состоит из единственной строки, включающей единственный столбец. Поэтому допустима, например, такая операция вставки (пример 21.3):

```
INSERT INTO EMP VALUES
           ROW (2445, (SELECT EMP_NAME
                       FROM EMP
                       WHERE EMP_NO = 2555),
                    '1985-04-08',
                    SELECT EMP_SAL
                       FROM EMP
                       WHERE EMP_NO = 2555),
                    NULL, NULL ),
           ROW (2446, (SELECT EMP_NAME
                       FROM EMP
                       WHERE EMP_NO = 2556),
                    '1978-05-09',
                    (SELECT EMP_SAL
                       FROM EMP
                       WHERE EMP_NO = 2556),
                    NULL, NULL );
```

После выполнения этой операции в таблице EMP появятся две новые строки для служащих с уникальными идентификаторами 2445 и 2446, причем первому из них будет присвоено имя и размер заработной платы служащего с уникальным идентификатором 2555, а второму – аналогичные данные о служащем с уникальным идентификатором 2556.

Вставка строк результата запроса

Наконец, обсудим вариант оператора вставки, когда набор вставляемых строк определяется через спецификацию запроса. Предположим, например, что требуется сохранить в отдельной таблице DEPT_SUMMARY сведения о числе служащих каждого отдела, их максимальной, минимальной и суммарной заработной плате. Пусть таблица DEPT_SUMMARY уже создана и имеет следующийзаголовок[152)]:

DEPT_SUMMARY:

DEPT_NO : DEPT_NO
DEPT_EMP_NO : INTEGER
DEPT_MAX_SAL : SALARY
DEPT_MIN_SAL : SALARY
DEPT_TOTAL_SAL : SALARY

Тогда заполнить таблицу можно с помощью следующей операции вставки (пример 21.4):

```
INSERT INTO DEPT_SUMMARY
   (SELECT DEPT_NO, COUNT(*), MAX (EMP_SAL),
        MIN (EMP_SAL), SUM (EMP_SAL)
     FROM EMP
     GROUP BY DEPT_NO);
```

21.2.2. Оператор UPDATE для модификации существующих строк в существующих таблицах

Общий синтаксис оператора `UPDATE` выглядит следующим образом:

```
UPDATE table_name SET update_assignment_commalist
    WHERE conditional_expression
update_assignment ::= column_name =
    { value_expression | DEFAULT | NULL }
```

Семантика оператора модификации существующих строк определяется следующим образом:

1. для всех строк таблицы с именем `table_name` вычисляется булевское выражение `conditional_expression`. Строки, для которых значением этого булевского выражения является `true`, считаются подлежащими модификации (обозначим множество таких строк через T_m);
2. каждая строка `s` (`s` T_m) подвергается модификации таким образом, что значение каждого столбца этой строки, указанного в списке `update_assignment_commalist`, заменяется значением, указанным в правой части соответствующего элемента спискамодификации[153)]. Значения столбцов строки `s`, не указанные в списке модификации, остаются неизменными.

Приведем примеры операций модификации таблиц.

Пример 21.5. Перевести всех служащих, выполняющих проект с номером 772, в отдел 632 и повысить им заработную плату на 1000 руб.

```
UPDATE EMP SET DEPT_NO = 632, EMP_SAL = EMP_SAL + 1000.00
   WHERE PRO_NO = 772;
```

При выполнении данной операции на первом шаге в таблице `EMP` будут найдены все строки, относящиеся к служащим, которые участвуют в проекте с номером `772`. На втором шаге во всех этих строках значение столбца `DEPT_NO` будет изменено на `632`, а к значению столбца `EMP_SAL` будет прибавлено `1000.00`.

Пример 21.6. Для всех служащих, работающих в отделах, заработная плата менеджеров которых превышает 30000 руб., установить размер заработной платы, на 1000 руб. превышающий средний размер заработной платы соответствующего отдела, а номера проектов, в которых участвуют эти служащие, сделать неопределенными.

```
UPDATE EMP SET EMP_SAL = (SELECT AVG (EMP1_SAL)
     FROM EMP EMP1
     WHERE EMP.DEPT_NO = EMP1.DEPT_NO)
         + 1000.00, PRO_NO = NULL
     WHERE (SELECT EMP1.EMP_SAL
          FROM EMP EMP1, DEPT
          WHERE EMP.DEPT_NO = DEPT.DEPT_NO
```

```
            AND DEPT_MNG = EMP1.EMP_NO AND) > 30000.00;
```

Конечно, если вам больше нравится другой стиль, то запрос, фигурирующий в разделе WHERE, можно переформулировать с использованием вложенного подзапроса (пример 21.6.1).

```
UPDATE EMP SET EMP_SAL = (SELECT AVG (EMP1_SAL)
     FROM EMP EMP1
     WHERE EMP.DEPT_NO = EMP1.DEPT_NO)
         + 1000.00, PRO_NO = NULL
     WHERE DEPT.NO IN (SELECT DEPT.DEPT_NO
          FROM EMP, DEPT
          WHERE DEPT_MNG = EMP_NO
          AND EMP_SAL > 30000.00);
```

Эти примеры позволяют понять, насколько богаты возможности оператора UPDATE. В разделе WHERE может содержаться любое условие, допускаемое в операторе выборки, а в элементах списка раздела SET может присутствовать любой вид value_expression, в том числе любой запрос, вырабатывающий одиночное значение (скалярный подзапрос).

21.2.3. Оператор DELETE для удаления строк в существующих таблицах

Общий синтаксис оператора DELETE выглядит следующим образом:

```
DELETE FROM table_name
     WHERE conditional_expression
```

В некотором смысле оператор DELETE является частным случаем оператора UPDATE (или, наоборот, действие оператора UPDATE представляет собой комбинацию действий операторов DELETE и INSERT).

Семантика оператора модификации существующих строк определяется следующим образом:

1. для всех строк таблицы с именем table_name вычисляется булевское выражение conditional_expression. Строки, для которых значением этого булевского выражения является true, считаются подлежащими удалению (обозначим множество таких строк через T_d);
2. каждая строка s ($s \ T_d$) удаляется из указанной таблицы.

С целью иллюстрации приведем два примера операции удаления строк.

Пример 21.7. Удалить из таблицы EMP все строки, относящиеся к служащим, которые участвуют в проекте с номером 772.

```
DELETE FROM EMP WHERE PRO_NO = 772;
```

Пример 21.8. Удалить из таблицы EMP все строки, относящиеся к служащим, размер заработной платы которых превышает размер заработной платы менеджеров их отделов.

```
DELETE FROM EMP WHERE EMP_SAL >
     (SELECT EMP1.EMP_SAL
          FROM EMP EMP1, DEPT
```

```
WHERE EMP.DEPT_NO = DEPT.DEPT_NO
   AND DEPT.DEPT.MNG = EMP1.EMP_NO);
```

Как и в операторе `UPDATE`, в разделе `WHERE` оператора `DELETE` можно использовать любой вид булевского выражения, допустимого в операторе выборки. Поэтому возможности оператора удаления строк ограничены лишь фантазией пользователя.

---

152 Мы не будем приводить полное определение таблицы, включающее требуемые ограничения целостности.

153 Если в правой части элемента модификации присутствует `value_expression`, в котором содержится запрос, то в случае использования в этом запросе имен столбцов модифицируемой таблицы под значениями этих столбцов понимается значение до модификации.

Лекция 22. Средства языка SQL для обеспечения авторизации доступа к данным, управления транзакциями, сессиями и подключениями

22.1. Введение

В этой лекции мы обсудим средства языка, которые касаются скорее администраторов баз данных, нежели конечных пользователей или программистов приложений. Но надо сказать, что любой квалифицированный пользователь SQL-ориентированной базы данных должен иметь представление об административных средствах SQL (тем более что средства управления транзакциями во многом затрагивают и его интересы).

Данная лекция включает материал, в меньшей степени концептуально связанный, чем это было в предыдущих лекциях курса, посвященных языку SQL. В первом из основных разделов лекции 22.2. Поддержка авторизации доступа к данным в языке SQL мы обсудим базовые идеи авторизации доступа к данным, заложенные в основу языка SQL. Метод *авторизации доступа*, используемый в SQL, относится к *мандатным (mandatory) видам защиты данных*. При этом подходе с каждым зарегистрированным в системе пользователем (*субъектом*) и каждым защищаемым объектом системы связывается *мандат*, определяющий действия, которые может выполнять данный субъект над данным объектом. В отличие от такого подхода, при применении *дискреционного (discretionary) метода ограничения доступа* с каждым из объектов системы связывается одна или несколько *категорий пользователей*, каждой из которых позволяются или запрещаются некоторые действия над объектом.

Следующий раздел 22.3. Управление транзакциями в SQL посвящен фундаментальному в области баз данных (и не только) понятию *транзакции* – последовательности операций над базой данных (в общем случае включающей операции обновления базы данных), которая воспринимается системой как одна неделимая операция. При классическом подходе к управлению транзакциями следуют принципу `ACID (Atomicy, Consistency, Isolation, Durability)`. Этому принципу следовали и разработчики языка SQL. Однако понятие транзакции выходит далеко за пределы SQL; механизмы управления

транзакциями составляют отдельную и большую исследовательскую область. В данной лекции мы не будем углубляться в технические детали управления транзакциями (этому посвящалась Лекция 13) и ограничимся возможностями, заложенными в язык SQL.

Наконец, в последнем основном разделе лекции 22.4. Подключения и сессии мы обсудим средства языка SQL, предназначенные для управления *сессиями* и *подключениями* пользовательских приложений. Подавляющее большинство реализаций языка SQL основывается на архитектурной модели *клиент-сервер*. Приложения обычно выполняются на *клиентской* аппаратуре, отделенной (по крайней мере, логически) от *серверной* аппаратуры, на которой работает собственно СУБД. Чтобы получить доступ к базе данных, приложение должно *подключиться* к серверу и образовать *сессию* в этом подключении. У приложения может одновременно существовать несколько подключений к разным серверам баз данных, но не более одной сессии в каждом подключении.

Лекция 23. Объектные расширения

23.1. Введение

В последней лекции этого курса мы кратко изложим суть объектных расширений, которые включены в стандарт SQL:1999. Лекция основана не на официальном тексте стандарта (он очень формален и скучен), а на книге Джима Мелтона [4.2], которая, по сути, является неформальным описанием семантики (rationale) соответствующей части языка. В указанной книге объектным расширениям языка SQL посвящено более 200 страниц. Естественно, наше изложение будет гораздо более кратким.

Как отмечалось в лекциях 12 и 15, язык SQL появился в середине 1970-х гг. при выполнении экспериментального проекта реляционной СУБД System R. Проект выполнялся в компании IBM, и это вполне естественно, потому что именно сотрудник IBM Эдгар Кодд предложил миру идею реляционных баз данных. От System R исходит большинство традиционных средств стандарта SQL:1999 (и SQL:2003), которые мы обсуждали в восьми предыдущих лекциях. Однако в этой лекции речь пойдет о возможностях современных вариантов SQL, которые не имеют отношения к System R (за исключением некоторых экспериментов по представлению сложных объектов средствами SQL) и, вообще говоря, к реляционной модели данных, а именно — о так называемых объектно-реляционных расширениях языка. Чтобы у читателей не возникло впечатление, что объектные расширения появились в языке SQL внезапно, благодаря какому-то озарению разработчиков языка, мы начнем эту лекцию с небольшого (и крайне субъективного) очерка истории объектно-реляционного подхода к организации систем баз данных.

23.1.1. Истоки и краткая история объектно-реляционных баз данных

Пальму первенства в области объектно-реляционных систем управления базами данных (ОРСУБД) оспаривают два весьма известных специалиста в области технологии баз данных – Майкл Стоунбрейкер (Michael Stonebraker) и Вон Ким (Won Kim).

Первые ОРСУБД

Майкл Стоунбрейкер начал работать в области баз данных в начале 1970-х гг. прошлого века

в университете Беркли. Его первым всемирно известным проектом была реляционная СУБД Ingres, которая существует и используется до сих пор в двух ипостасях – как свободно распространяемая система (университетская Ingres; код поддерживается в Беркли) и как коммерческая СУБД, принадлежащая компании Computer Associates. В исходном варианте СУБД Ingres отсутствовала поддержка языка SQL (поддерживался собственный язык запросов QUEL), но система уже обладала некоторыми уникальными чертами, которые, с небольшой натяжкой, можно было бы назвать объектными (например, в СУБД Ingres допускалось определение пользовательских процедур, выполняемых на стороне сервера). Кроме того, в проекте Ingres очень большое внимание уделялось управлению правилами.

В 1980-е гг. Майкл Стоунбрейкер возглавлял проект Postgres (вариант этой системы под названием PostgreSQL в настоящее время является весьма популярным свободно доступным продуктом). В Postgres были реализованы многие интересные средства: поддерживалась темпоральная модель хранения и доступа к данным, и в связи с этим был полностью пересмотрен механизм журнализации изменений, откатов транзакций и восстановления БД после сбоев; обеспечивался мощный механизм ограничений целостности; поддерживались ненормализованные отношения (работа в этом направлении началась еще в среде Ingres), хотя и довольно странным способом: в поле отношения мог храниться динамически выполняемый запрос к БД.

Одно свойство системы Postgres сближало ее со свойствами объектно-ориентированных СУБД (ООСУБД). В Postgres допускалось хранение в полях отношений данных абстрактных, определяемых пользователями типов. Это обеспечивало возможность внедрения поведенческого аспекта в БД, т. е. решало ту же задачу, что и ООСУБД, хотя, конечно, семантические возможности модели данных Postgres были существенно слабее, чем у объектно-ориентированных моделей данных. Основная разница состояла в том, что в Postgres не предполагалось наличие языка программирования, одинаково понимаемого как внешней системой программирования, так и системой управления базами данных. Как и в Ingres, в исходном варианте Postgres не поддерживался язык SQL (имелся собственный язык запросов Postquel). Кстати, во времена Postgres Майкл Стоунбрейкер не использовал термин объектно-реляционная система, предпочитая называть свою СУБД системой следующего поколения.

В начале 1990-х гг. Стоунбрейкер создал компанию Illustra, основной целью которой был выпуск коммерческого варианта СУБД Postgres, получившего название Illustra. В этой системе поддерживались основные идеи Postgres, но уже присутствовала и поддержка языка SQL. В конце 1995 г. компания Illustra была поглощена компанией Informix, и это привело к выпуску в 1996 г. СУБД Informix Universal Server (см. ниже).

Имя Вона Кима стало широко известно во второй половине 1970-х гг., когда он примкнул к участию в экспериментальном проекте компании IBM System R. Наиболее известная ранняя работа доктора Кима была посвящена преобразованию SQL-запросов с целью превращения запросов с вложенными подзапросами в запросы с соединениями.

В 1980-е гг. Вон Ким работал в компании MCC, где успешно выполнил реализацию серии прототипов ООСУБД Orion. В этих прототипах были опробованы многие идеи объектно-ориентированных СУБД. Одной из интересных особенностей проекта было то, что в качестве основного языка программирования использовался объектный вариант известного функционального языка Lisp.

В конце 80-х гг. д-р Ким основал компанию UniSQL, выпустившую в 1991 г. первую версию продукта UniSQL, который Вон Ким стал называть объектно-реляционной системой. Трудно

оценивать коммерческий успех этой СУБД. В настоящее время она принадлежит Корейской национальной телекоммуникационной компании и, по всей видимости, продолжает использоваться. Поскольку UniSQL была первой СУБД, официально называемой объектно-реляционной системой, приведем ее краткое описание.

UniSQL обеспечивала возможность построения так называемых федеративных систем баз данных. При этом обеспечивалось единое представление данных, которые могли храниться либо в базе данных, непосредственно управляемой UniSQL, либо в какой-либо из реляционных баз данных, управляемой СУБД Oracle, Informix, Sybase и т. д., либо в какой-либо дореляционной базе данных. Сервер UniSQL обеспечивал интегрированный доступ к данным, управляемым разными СУБД. Одна из возможных конфигураций использования системы показана на рис. 23.1.

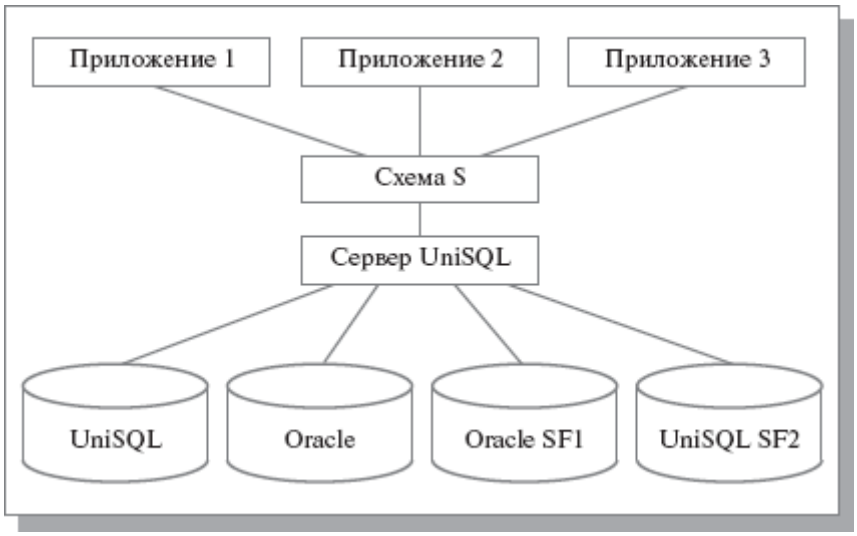


Рис. 23.1. Возможная конфигурация системы UniSQL

Как показывает рис. 23.1, сервер UniSQL позволяет представлениям работать через «глобальную» схему базы данных S, полученную из двух «фрагментарных» схем баз данных, которые управляются непосредственно UniSQL и СУБД Oracle[190)].

Разработчики UniSQL полагали, что построение полнофункциональной СУБД, основанной на принципиально новой модели данных, крайне проблематично. Был выбран подход к расширению реляционной модели, выражающийся в следующих четырех принципах:

- значениями атрибутов отношений могут быть не только литеральные значения, но и объекты;
- значения атрибутов отношений не обязательно являются атомарными;
- при построении таблиц (классов) может использоваться механизм наследования;
- классы включают операции.

В созданной компанией системе поддерживалось расширение стандарта SQL – SQL/X, одновременно включающее и объектно-ориентированные, и реляционные возможности. В одном языке поддерживались возможности и определения данных, и манипулирования ими. В качестве языковых средств программирования приложений поддерживались языки C++ и Smalltalk.

Внедрение объектных расширений в основные РСУБД

В конце 1989 г. группа известных специалистов в области языков программирования баз данных опубликовала документ под названием Манифест систем объектно-ориентированных баз данных [2.6] (для краткости будем называть его Первым манифестом). Основным поводом к написанию и публикации этого материала было то, что к тому времени существовал ряд систем управления базами данных, которые, по большому счету, объединяла только общая привязанность к объектно-ориентированным языкам программирования. В Первом манифесте была предпринята попытка дать определение системам объектно-ориентированных баз данных. Авторы стремились привести описание основных свойств и характеристик, которыми должна обладать система, претендующая на то, чтобы называться системой объектно-ориентированных баз данных. Первый манифест был написан академическими исследователями; почти все они являлись и являются профессорами различных университетов. Конечно, это нашло свое отражение в стиле Первого манифеста – очень мягком и умеренно рекомендательном (хотя по своему духу предложения этого манифеста были весьма радикальными).

Через год после публикации Первого манифеста вышел в свет Манифест систем баз данных третьего поколения [2.7], инициатором которого, очевидно, был Майкл Стоунбрейкер (хотя у документа формально имелось много авторов). Мы говорим об этом с уверенностью, поскольку в этом манифесте повсюду видны идеи Стоунбрейкера, использованные им в проектах Ingres и Postgres.

В некотором роде Манифест систем баз данных следующего поколения (для краткости мы будем называть его Вторым манифестом) стал ответом миру объектно-ориентированных баз данных со стороны мира SQL-ориентированных баз данных. Если Первый манифест был хотя и немного путанным, но все-таки носил научный характер, то Второй манифест является в большей степени инженерно-публицистическим документом. Второй манифест можно расценивать как реакцию индустрии СУБД на неприятные для нее измышления науки.

Второй манифест (или, вернее, работы, приведшие к его появлению) имел важные последствия. В 1995 г. компания Informix (ныне входящая в состав IBM) купила компанию Майкла Стоунбрейкера Illustra[191)], и Стоунбрейкер стал техническим директором Informix. В начале 1996 г. компания Informix объявила о выпуске принципиально нового продукта Informix Universal Server, в котором, как утверждалось, лучшие черты Informix Online Server сочетались с развитыми объектными чертами, присущими Illustra.

К выпуску Informix Universal Server очень ревниво отнеслась компания Oracle, которая немедленно заявила, что у нее готов собственный объектно-реляционный продукт, по всем параметрам превосходящий систему компании Informix. Эта система, получившая название Oracle8, была выпущена в конце лета 1996 г.

Годом позже к группе производителей объектно-реляционных СУБД (ОРСУБД) примкнула компания IBM, выпустившая продукт DB2 Universal Database. Как выяснилось позже, все наиболее важные свойства этого продукта были реализованы еще в 1995 г. в СУБД DB2 for Common Servers. Просто компания IBM предпочла до поры не афишировать свои расширения.

Первые пару лет вокруг объектно-реляционных СУБД стоял большой шум. Позже выяснилось, что маркетинговые ожидания компаний гигантов оказались преувеличенными. (В частности, это было одной из основных причин падения компании Informix.) Сегодня

объектные расширения SQL-ориентированных СУБД предлагаются пользователям лишь в качестве дополнительных, хотя и важных возможностей.

Объектные расширения языка SQL были зафиксированы в стандарте SQL:1999 [3.9]. В той или иной мере эти расширения поддерживаются во всех трех перечисленных выше продуктах. В настоящее время ближе всех к стандарту находятся СУБД компании Oracle и DB2 компании IBM.

23.1.2. Объектная модель SQL

Объектные расширения SQL:1999 базируются на некоторой объектной модели, хотя эта модель в явном виде в стандарте не специфицируется. Объектная модель SQL[192)] не является тождественной объектным моделям какого-либо объектно-ориентированного языка программирования или какой-либо объектно-ориентрованной системы баз данных. Однако при определении объектной модели SQL участники процесса стандартизации тщательно проанализировали ряд других языков и систем с целью выяснения достоинств и недостатков их объектных моделей. По мнению авторов стандарта SQL:1999, выработанная ими объектная модель похожа на объектную модель языка Java[193)], но при этом адаптирована к природе языка SQL как языка СУБД с наличием стабильно хранимых метаданных и данных.

Объектная модель SQL:1999 включает два основных отличительных компонента – структурные, определяемые пользователями типы данных (User Defined Type – UDT) и типизированные таблицы (Typed Table). Первый компонент позволяет определять новые типы данных, которые могут быть гораздо более сложными, чем встроенные типы данных языка SQL. При определении структурного UDT требуется специфицировать не только содержащиеся в нем элементы данных, но и семантику типа данных, т. е. его поведение на основе интерфейса вызовов методов. Второй компонент – типизированные таблицы – позволяет определять таблицы, строки которых являются экземплярами (или значениями) UDT, с которым явно ассоциируется таблица. Во многих отношениях строка типизированной таблицы похожа на объект класса в объектно-ориентированной системе.

В стандарте SQL:1999 определены два пакета объектных свойств – минимальный (PKG006) и полный (PKG007), которым должны удовлетворять SQL-ориентированные ОРСУБД, претендующие на соответствие стандарту. Ниже будут перечислены свойства, включенные в каждый из пакетов, но смысл этих свойств будет понятен только после прочтения остальных разделов.

Пакет PKG006 включает всего пять свойств:

- свойство S023 («Basic structured types») – возможность определять UDT и их методы с ограниченными возможностями;
- свойство S041 («Basic reference types») – возможность определять и использовать ссылки на экземпляры UDT, входящие в типизированные таблицы;
- свойство S051 («Create table of type») – возможность создания типизированных таблиц;
- свойство S151 («Type predicate») – возможность определения точного типа (в иерархии типов) экземпляра UDT;
- свойство T041 («Basic LOB data type support») – возможность определения LOB-типов в смысле SQL (с необязательной поддержкой операций, кроме операций сохранения и полной выборки).

Пакет PKG007 содержит девять дополнительных свойств:

- свойство S024 («Enhanced structured types») добавляет к свойству S023 ряд развитых возможностей, в число которых входят возможности кодирования методов на языках, отличных от SQL, сравнения экземпляров UDT и передача экземпляров UDT в качестве параметров различных процедур;
- свойство S043 («Enhanced reference types») расширяет свойство S041 возможностями определения ссылок с областью действия, автоматической проверки законности ссылок и т. д.;
- свойство S071 («SQL-paths in function and type name resolution») позволяет использовать путевые выражения SQL (SQL-path) в алгоритме разрешения типа;
- свойство S081 («Subtables») расширяет возможности свойства S051, допуская организацию иерархии таблиц, аналогичной иерархии типов соответствующих UDT;
- свойство S111 («ONLY in query expressions») обеспечивает возможность выборки только экземпляров указанного типа, без экземпляров любого из его подтипов;
- свойство S161 («Subtype treatment») позволяет информировать среду SQL о том, что некоторый экземпляр UDT в действительности является экземпляром указанного подтипа;
- свойство S211 («User-defined cast functions») разрешает определять подпрограммы, преобразующие экземпляры UDT к другим типам;
- свойство S231 («Structured type locators») способствует доступу к экземплярам UDT из прикладных программ;
- свойство S023 («Transform functions») позволяет определять подпрограммы, преобразующие значения UDT в значения предопределенных типов данных, и наоборот.

23.1.3. Цели лекции

Следует заметить, что в этой, безусловно, перегруженной материалом лекции мы преследуем две основные цели. Первая цель состоит в том, чтобы показать читателям, что в средствах определения структурных типов SQL используются, по сути, все базовые возможности определения объектов базы данных и выборки данных, которые обсуждались в предыдущих лекциях. Более того, определенные пользователями типы в SQL являются объектами первого класса; UDT можно использовать в любой конструкции языка, в которой допускается применение предопределенного или конструируемого типа данных. Очень важно отдавать себе отчет в том, что наличие возможности определять пользовательские типы не делает язык SQL менее реляционным или более объектным. Эта возможность «всего лишь» фантастически повышает выразительную мощность языка.

Второй целью является демонстрация того, как на основе базовых механизмов языка удалось ввести дополнительные конструкции, которые действительно вплотную приближают SQL к объектному миру. Здесь, конечно, основную роль играет связка UDT и механизма типизированных таблиц, которые играют в SQL своеобразную совмещенную роль классов и коллекций объектов.

Может оказаться, что материал этой лекции покажется сложным, поскольку для его усвоения не помешало бы иметь предварительную подготовку в области полнотиповых языков программирования, объектно-ориентированных языков и систем баз данных и т. д. Хочется надеяться, что возникновение трудностей при изучении лекции не отпугнет читателей от этой темы, а напротив, послужит стимулом к изучению дополнительной литературы.

Возможна и другая опасная ситуация. Краткость и некоторая формальность изложения может создать ложное впечатление тривиальности объектных расширений SQL. В этом случае могу посоветовать перечитать предыдущие лекции курса, относящиеся к SQL, считая, что везде, где используются предопределенные или конструируемые типы, применяются некоторые UDT, а в тех случаях, где имеются таблицы, связываемые естественным соединением, используются типизированные таблицы. Думаю, это позволит оценить мощь новых возможностей SQL.

Введя этот необходимый контекст, перейдем к описанию соответствующих механизмов SQL:1999.

---

190 Вопросы интеграции данных выходят за пределы тематики этого курса. Однако следует сделать два замечания. Во-первых, проблематика обеспечения доступа к разнородным данным через некоторую глобальную, или концептуальную схему интересует сообщество баз данных в течение нескольких десятков лет. Существовали многочисленные попытки обеспечить интеграцию баз данных, представленных во всех возможных моделях (сетевой, иерархической, реляционной, объектно-ориентированной). С точки зрения теории решение проблемы возможно, но на практике это приводит к очень сложным с технической точки зрения реализациям, обладающим крайне низкой производительностью. Во-вторых, в MCC в 1980-е годы был создан весьма успешный прототип системы, интегрирующей SQL-ориентированные базы данных. Должно быть понятно, что такая интеграция существенно проще в техническом смысле, поскольку глобальная и фрагментарные схемы представлены в близких понятиях. Похоже, что проект UniSQL в большой степени базировался и на этой работе.

191 Компания Illustra была создана Стоунбрейкером для коммерциализации разработанной под его руководством свободно доступной СУБД Postgres.

192 Конечно, это не модель данных в смысле Кодда.

193 Далеко не факт, что ориентация на язык Java была правильным решением. По мнению автора данного курса, причиной являются отнюдь не уникальные достоинства языка Java (обсуждение этого языка не является задачей автора), а то, что во время разработки стандарта SQL:1999 язык Java был особенно моден. Помимо прочего, заметим, что для языка Java (насколько известно автору) никогда не определялась формальная объектная модель.