Малахов Е.В.

Базы данных

Конспект лекций

Кафедра
информационных технологий

Последняя редакция: 09.2018

Содержание

Понятие данных. Обработка данных. Информационная система

Для информационной поддержки управленческих решений в разных областях науки и техники создаются разные информационные системы.

На сегодняшний день не существует единого и общепринятого определения информационной системы. Так „Словарь по кибернетике“ утверждает, что „*информационные системы* — системы обработки данных о любой предметной области со средствами накопления, хранения, обновления, поиска и выдачи данных…“ [15].Однако это определение не охватывает ряд аспектов обработки и передачи данных и не отражает место предметной области, для которой создаётся система.

Более корректным и полным представляется определение М.Р. Когаловского: „Автоматизированной информационной системой называется комплекс, который содержит вычислительное и коммуникационное устройства, программное обеспечение, лингвистические средства и информационные ресурсы, а также системный персонал, обеспечивающий поддержку динамической информационной модели некоторой части реального мира для удовлетворения информационных нужд пользователей“ [10].

Мы считаем, что и это определения следовало бы дополнить, сделав акцент на том, что **информационная система** — это совокупность связанных в единое целое системой каналов передачи информации баз данных, информационных хранилищ, баз знаний, а также информационных технологий, которые поддерживают процессы обработки, анализа и передачи информации различного уровня интеграции. При этом обмен информацией осуществляется как между объектами и подобластями выделенной предметной области, так и между данной предметной областью и остальной частью окружающего её мира.

В зависимости от объёма решаемых задач, используемых технических средств и организации функционирования информационные системы делятся на группы (классы) [15](рис. 1).

В определении информационной системы используются два понятия: данные и предметная область (подобласть).

Данные — цифровые и графические сведения об объектах окружающего мира [14].

Под термином ***обработка данных*** мы будем понимать последовательность действий, необходимых для выполнения некоторой задачи. Различают *числовую* и *нечисловую* обработки.

Для числовой обработки характерен большой объем вычислений, состоящих из ряда итераций (например, решение различных нелинейных уравнений, операций с матрицами и векторами и т.п.) с обязательным сохранением высокой точности результатов.

При нечисловой обработке не нужна высокая точность и большой объем вычислений. Однако имеется очень большой объем обрабатываемых данных и, кроме того, при нечисловой обработке требуется выполнение таких специфических операций, как поиск конкретных данных и их сортировка.

В само понятие „данные“ как объекты при числовой и нечисловой обработках вкладывается разное содержание.

Так при числовой обработке мы манипулируем переменными, векторами, матрицами, константами и др. При этом нас не интересует их содержимое. Например, при выполнении какой-нибудь арифметической операции или операции ввода/вывода необходимо знать адреса переменных (их имена), и не важно, что находится в этих переменных.

При нечисловой обработке объектами являются файлы, записи, поля, сетки, отношение и др. В этом случае нас больше интересует информация, которая содержится в конкретной записи, чем местоположение этой записи в файле или оперативной памяти.

Рис. 1. Классификация информационных систем

Термин **база данных (БД)** обозначает совокупность структурированных данных, предназначенных для общего использования при решении массовых проблем, существующих или сформулированных в определённой предметной области.

Оперируя какой-нибудь информацией, можно рассматривать данные, относящиеся к одной или общие для множества организаций или сфер деятельности человека.

Реальный мир, который должен быть отображён в БД, называют **предметной областью (ПрО)** [8].

Любая часть реального мира является результатом взаимодействия определённой части физического мира и интеллектуального мира, который возник в выделенной части физического мира [18].

Физическим миром будем называть Вселенную. Часть физического мира — это некоторая его *k*-мерная область *D*, заданная в виде множества *k*- мерных точек в *m*-мерном пространстве, которое описывает нашу Вселенную. Область *D* задаётся системой *k*-мерных отношений. То есть это множество всех *k*-мерных точек соответствующего пространства, которые удовлетворяют заданной системе математических соотношений.

Интеллектуальным миром будем называть биологическую систему, которая достигла такого уровня развития в некоторой физической части мира, при котором она в состоянии изменить часть физического мира, в котором она существует и продолжает развиваться, влияя на законы его эволюции.

Кроме того, в процессе развития интеллектуального мира на некотором этапе возникает систематическая потребность во все более совершенных формах анализа, обработки и организации информации и знаний, не связанных непосредственно с умственными процессами, но которые позволяют под управлением человека принимать желательные для него формы этих процессов.

Часть информационного пространства, которая имеет необходимую степень независимости, самостоятельности и способности к трансформации в необходимые формы и передачи их согласно человеческим законам, но, в то же время, независимую от человека, будем называть виртуальным миром.

Для уточнения понятия виртуальный мир или виртуальная реальность, или виртуальность (от лат. *virtus* — потенциальный, возможный, доблесть, энергия, сила, а также мнимый, воображаемый; лат. *realis* — вещественный, действительный, существующий) отметим два определения, приведенных в [6].

„Виртуальная реальность в постклассической науке — понятие, с помощью которого описывается совокупность объектов следующего (относительно нижележащей, порождающей их реальности) уровня. Эти объекты онтологично равноправны с порождающей их „константной“ реальностью и автономны; при этом их существование полностью определено перманентным процессом их воспроизведения порождающей реальностью...

Виртуальная реальность — технически конструируемая с помощью компьютерных средств интерактивная среда порождения и оперирования объектами, подобными реальным или воображаемым, на основе их трёхмерного графического представления, симуляции их физических свойств (объем, движение и т.п.), моделирования их способности влияния и самостоятельного присутствия в пространстве“.

Фактически, **виртуальный мир** — это воображаемый мир, для которого определены конкретные физические и математические законы и который, при соответствующих условиях, может быть с помощью технических средств сделан доступным для восприятия органами чувств человека. То есть виртуальным миром является „физический“ мир, созданный в пределах интеллектуального мира.

В каждом с миров: физическом, интеллектуальном, виртуальном, в свою очередь, можно выделить предметную область. Предметная область реального мира является продуктом взаимодействия соответствующих предметных областей каждого из миров.

В [21] определено, что предметная область или домен (от англ. *domain* — область) — чётко очерченный реальный, гипотетический или абстрактный мир, населённый взаимозависимым набором объектов, которые ведут себя согласно характерным для предметной области правилам и линиям поведения.

Построение любой информационной системы начинается с создания математической, информационной, онтологической и т.п. модели ПрО, которая может быть реализована в виде баз и хранилищ данных.

Поэтому для того, чтобы некоторую предметную область представить в БД, нужно выделить существенные понятия, необходимые пользователю, и связи между ними.

(Например, книги в библиотеке: читателю нужны автор и название, а не количество рисунков или качество бумаги).

Классификацию понятий предметной области, конкретное наполнение которых будет выполнено в БД, называют *логическим проектированием* данных или баз данных. С математической точки зрения можно сказать, что **логическое проектирование — это абстрагирование информации о предметной области.**

В связи с этим необходимо определить следующее:

— *физические данные* — данные, хранящиеся в памяти компьютера (дисковой или оперативной).

— *логическое представление данных* соответствует пользовательскому представлению физических данных.

Например: в файле БД хранятся 2 множества символьных строк — это физические данные. Вариант логического представления: авторы и названия книг.

Структуры данных

При использовании компьютера для сохранности и обработки данных необходимо хорошо знать тип и структуру данных, и определить способ их представления.

Программу, предназначенную для достижения некоторой цели, можно рассматривать как результат объединения структур данных и алгоритма.

Чаще всего существуют различные алгоритмы решения одной и той же задачи, которые зависят от способа представления и упорядоченности данных. Реализовать структуру данных необходимо так, чтобы обеспечить эффективность их обработки большим количеством алгоритмов.

Как правило, структурирование данных требует точного определения типа каждого элемента, входящего в структуру.

Математические принципы концепции типа, которые положены в основу языков программирования высокого уровня следующие [1]:

1. Любой тип данных определяет множество значений, к которым может относиться некоторая константа, которая может принимать любая переменная или выражение и которое может формироваться операцией или функцией.
2. Тип любой величины, обозначаемой константой, переменной или выражением, может быть выведен по её виду или из её описания (без необходимости каких-либо вычислений).
3. Каждая операция или функция требует аргументов определённого типа и даёт результат также фиксированного типа.

В вычислительной технике существует ряд так называемых **простейших стандартных типов** данных: целый, логический, символьный и т.п.

Кроме того, существует возможность определения новых типов путём *перечисления* множества их значений. Поэтому эти типы ещё зовут **перечисляемыми**. Они также являются простейшими, но, конечно, не стандартными. Например:

цвет {синий, красный, зелёный}

Структуры данных иначе называют *составными типами*, так как они представляют собой совокупности компонентов, относящихся к определённым ранее типам.

Если все компоненты относятся к одному типу, то такой тип называют *базовым*.

Очевидно, наиболее широко известная структура данных — одномерный или многомерный **массив**. Эта структура представляет собой отображение некоторого конечного множества данных на множество данных другого типа. Здесь областью определения является множество данных типа *индекс*, а областью значений — *множество элементов массива*. Массив состоит из элементов одного базового типа, поэтому структура массива *однородна*.

Наиболее общий метод получения составных типов состоит в объединении элементов разных типов. Причём сами элементы могут быть составными.

Пусть у нас есть следующие элементы:

$$a_i \in A_1, a_j \in A_2.$$

Если рассмотреть множество вида:

$$\{(a_i, a_j) | a_i \in A_1, a_j \in A_2\},$$

то мы получим составной тип, который называется **прямым** *(декартовым)* **произведением**. Каждый элемент такой структуры носит название *кортеж*.

$A_1 = \{1,3,8,9\}, A_2 = \{2,3\}$

$A_1 \times A_2 = \{(1,2),(3,2),(8,2),(9,2),(1,3),(3,3),(8,3),(9,3)\}.$

Ещё одним важным типом структурированных данных являются **последовательности**. Все элементы последовательности имеют один и тот же тип. Структура последовательности очень похожа на структуру массива. Существенное отличие заключается в том, что в массиве количество элементов фиксируется в его описании, а количество элементов последовательности (длина) конечно, но не фиксировано. Типичные примеры последовательного типа данных — *файл* и *стек*.

Последовательным файлом называется последовательность, для которой определены следующие операции:

1. Формирование пустой последовательности;
2. Выборка начального элемента последовательности;
3. Добавление элемента в конец последовательности,

Стек — последовательность, над которой возможное выполнение таких операций как:

1. Создание пустой последовательности.
2. Получение первого элемента последовательности.
3. Добавление элемента в начало последовательности.

Системы баз данных

Для создания, хранения и использования данных были разработаны специальные программные средства — *системы управления базами данных* (СУБД), где БД, как мы уже говорили, это структурированная совокупность данных, хранимая в вычислительной системе.

Совокупность БД и СУБД представляют собой *банк данных* (БнД).

Банк данных (БнД) является современной формой организации хранения и доступа к информации. Существует множество определений банка данных.

Например: „*Банк данных — это система специальным образом организованных данных (баз данных), программных, технических, языковых, организационно-методических средств, предназначенных для обеспечения централизованного накопления и коллективного многоцелевого использования данных*“ [Общеотраслевые руководящие материалы по созданию банков данных. - М.: ГКНТ, 1982].

Нельзя сказать, что термин „банк данных“ является общепризнанным. В некоторой англоязычной литературе в последнее время используется термин „система баз данных“ (Database System), который по своему смыслу близок к приведённому понятию банка данных (система баз данных включает базу данных, систему управления базами данных, соответствующее оборудование и персонал). Согласно семантике украинского и российского языков понятие „система баз данных“ воспринимается уже, чем его истинное обозначение [5]. Итак, слово „банк“ в этом смысле является лучшим, потому что „банк“ обычно обозначает не только то, что хранится в нем, но и всю инфраструктуру. Поэтому нельзя отождествлять понятие „ база данных“ и „банк данных“.

СУБД — набор программных средств, позволяющих [14]:

а) обеспечить пользователей языковыми средствами определения и манипуляции данными (выборка, обновление, удаление). Такими средствами является язык определения данных (ЯОД) и язык манипулирования данными (ЯМД). „Язык данных“, обозначающий или один, или оба эти языка, может быть включён в универсальный язык программирования (C, Pascal, C++, Delphi, C#, Java, ...) и называется *тогда подъязыком* данных, или быть автономным и называемым *языком запросов*;

б) обеспечить поддержку моделей данных пользователя.

При логическом проектировании априорно определённые рамки, которые используются для абстрагирования предметной области, называют моделью данных. То есть, модель данных — это средство для определения логического представления физических данных, относящихся к некоторому приложению. Модель данных позволяет представить разные связи или свойства объектов реального мира, определить способы получения новых понятий на основе имеющихся и т.п.;

в) обеспечить программу, реализующую функции ЯОД и ЯМД, в которых возможное определение, создание и манипуляция данными.

г) обеспечить защиту и целостность данных. Имеется в виду защита от несанкционированного доступа или несанкционированных действий над данными со стороны пользователей в многопользовательском и/или сетевом режимах.

Целостность имеет смысл при работе в тех же режимах. То есть, если один из пользователей внёс изменения в общие данные, то эти изменения должны быть доступны другим пользователям или получены ими. В противном случае возникают коллизии, типа знакомых ситуаций продажи железнодорожных билетов на одно и тоже место.

Архитектура системы БД

Графическое представление архитектуры СБД [1] имеет следующий вид (рис. 2).

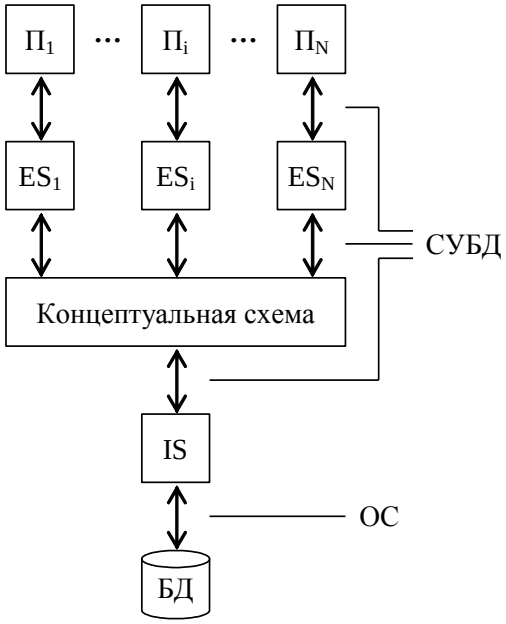


Рис. 2. Архитектура системы баз данных

Описание предметной области в терминах некоторой модели данных называют **концептуальной схемой**. Другими словами концептуальная схема — представление *всего* содержимого базы данных, которое описано с помощью *концептуального языка* определения данных, плюс средства безопасности и правила обеспечения целостности. Мы подчёркиваем именно *концептуального* ЯОД, потому что его определения должны относиться только к содержанию информации. То есть, в концептуальной схеме *не должно быть* упоминания о представлении хранимого файла, последовательности хранимых записей, индексирования и других подробностей хранения.

Такой схемой может быть, например, представление ПрО в виде совокупности связанных таблиц, описывающих объекты и связи между ними.

Отображение концептуальной схемы на физический уровень называют **внутренней моделью** (схемой), которая описывает различные типы хранимых записей, индексы, физическую последовательность хранимых записей и т.п.

На этом уровне оговаривается, хранится ли БД в виде совокупности файлов, как в локальных *dBase*-подобных системах, или в виде таблиц базы данных на сервере баз данных в клиент-серверных системах (типа *Oracle*, *Informix* и т.п.).

Однако эта схема не затрагивает реальную физическую сторону хранения данных — дисков, дорожек, секторов и т.п. Это забота ОС (например, единственный диск или *RAID*-массив).

И, наконец, конкретного пользователя может интересовать не вся ПрО, описанная концептуальной схемой, а лишь некоторая её часть. Представление части концептуальной схемы в терминах ПрО носит название **внешней схемы** (модели) данных, привязанной к конкретному приложению. Таким образом, внешняя схема — это содержимое БД, каким его видит определённый пользователь. Иначе говоря, для пользователя — внешнее представление и есть БД. Именно на этом уровне создаётся (реализуется) *пользовательский интерфейс*, который с одной стороны (со стороны пользователя) поддерживает терминологию предметной области, а с другой стороны — принимает участие в отображении информации на концептуальный уровень.

Здесь необходимо ещё раз упомянуть о языке данных. Точнее, о подъязыке. Дело в том, что часть внешней схемы, которая соответствует пользовательскому интерфейсу, реализуется с помощью *универсального языка*, а та, что отвечает за взаимодействие с БД, точнее с концептуальной схемой, — с помощью *подъязыка данных*. Конечно, можно воспользоваться и языком запросов, однако это значительно сузит возможности организации пользовательского интерфейса. Очевидно, что при таком подходе к внешней схеме предпочтительнее, чтобы подъязык данных и универсальный язык были *формально неразличимы*. Такие неразличимые или трудноразличимые языки называются сильносвязанными. Примерами могут служить *dBase*-подобные языки и, соответственно,

СУБД *Clipper, Paradox, dBase* и т.п. Однако они или получили развитие для устаревшей ОС *MS-DOS*, или ориентированы на работу с локальными БД (максимум — на вычислительную сеть в режиме файл-сервера), или и то, и другое.

Другие универсальные языки (*C, Pascal, Basic* и т.п.) и встроенные в них подъязыки данных слабосвязанные, что не является препятствием для создания качественных приложений.

Например, существует язык данных, который может использоваться и как язык запросов, и как встроенный практически во все перечисленные универсальные языки. Это язык *SQL — Structured Query Language*, позволяющий работать с системами клиент/сервер, распределёнными и локальными БД. К нему мы вернёмся позже.

В заключение данного раздела отметим ещё один аспект архитектуры БД, а именно: **файловую архитектуру** и **архитектуру клиент/сервер**. Причём и та и другая могут быть реализованные как на локальной машине, так и в сети ЭВМ.

В случае **файловой** архитектуры (в вычислительной сети — файл-серверной) и определение, и манипулирование данными, и взаимодействие с пользователем (порождение запросов и вывод результатов) осуществляется пользовательским приложением, созданным с помощью сильносвязанных языков. В многопользовательских системах такого типа, то есть в архитектуре файл-сервер, пользователь, точнее программист должен самостоятельно решать задачи обеспечения защиты и целостности данных, упоминавшиеся ранее.

При реализации архитектуры **клиент/сервер** систему баз данных делят на 2 части: один или несколько *серверов* (машин) *БД* и множество *клиентов* (множество может быть единичным).

Сервер — это собственно СУБД, удовлетворяющая всем отмеченным ранее критериям. Соответственно, он поддерживает все основные функции СУБД на внутреннем, концептуальном и, частично, внешнем уровнях, а именно: определение данных, манипулирование ими, обеспечение защиты и целостности данных, обработку запросов и т.п.

Клиенты — это различные пользовательские и системные приложения, которые реализуют интерфейсы пользователей, генерацию запросов и отображение результатов. Такие приложения создаются чаще всего с помощью универсальных языков, обязательно включающих понимаемые сервером подъязыки данных.

Сегодня для создания таких приложений разработаны немало разных *CASE*-систем (*Computer Aided Software Engineering*) типа *Delphi, Microsoft Visual Suite, C++ Builder, CAVO* и т.п., включающих в себя в качестве подъязыка данных *SQL*.

И сервер, и клиент могут быть организованы на одном компьютере. В этом случае мы говорим о **локальной системе**.

Если же клиент и сервер функционируют на разных машинах, связанных вычислительной сетью, то такая структура носит название **распределённой обработки** в силу независимости или параллельности работы СУБД и приложения. Сегодня синонимом именно такой структуры и стал термин *„клиент/сервер“*. Необходимо отметить, что как клиентов, так и серверов может быть несколько. При этом несколько клиентов могут иметь одновременный доступ к какому-нибудь серверу. Однако обязательным для распределённой обработки является то, что в любой момент времени клиент может иметь доступ только к одному серверу.

Если же клиент может получать доступ к любому количеству серверов одновременно, то такая система называется **распределённой системой БД**. При этом серверы рассматриваются клиентом как один, и пользователь может не знать, на какой именно машине какая часть данных содержится. Это забота СУБД (таких как *Oracle, Sybase* или *PostgreSQL*).

Отметим, что в обоих случаях распределённых систем возможен вариант, когда на каждой станции вычислительной сети организован и клиент, и сервер. То есть когда одна и та же машина выступает в роли клиента для одних пользователей, и в роли сервера для других. Подобную структуру позволяет организовать, например, СУБД *SQLbase* 6 фирмы *Gupta*.

Моделирование предметной области

При моделировании предметной области мы обычно начинаем с определения понятий о конкретных объектах или явлениях и представления их в удобных (привычных) нам терминах [6]. То есть мы оговариваем *сущность* этих объектов и явлений. Например, сущностью является множество одноименных объектов, множество студентов, множество дисциплин, библиотека (множество книг). При этом различают **имя сущности** как множество или набор объектов, которое как понятие совпадает с приведенным термином *сущность*, и **экземпляр сущности** — конкретный элемент этого набора.

Каждая сущность владеет рядом основных *свойств*, которые характеризуют её. Например: свойствами сущности *Студент* являются фамилия, имя, отчество, номер группы, номер студенческого билета. Такие свойства получили название **атрибутов** сущностей. Таким образом, **сущность — это множество объектов, обладающих одинаковым набором атрибутов**, а формальное *описание экземпляра сущности* представляет собой множество элементов данных, соответствующих *конкретным значениям* его атрибутов.

Подмножество атрибутов, однозначно определяющих конкретный экземпляр сущности, называют **идентификатором** или **потенциальным ключом**. Если это подмножество единичное, то такой ключ называется *простым*. Если же в подмножество включается несколько атрибутов, то ключ — *составной*. Так, потенциальным ключом может быть и весь набор атрибутов сущности.

При этом необходимо учесть, что *потенциальный ключ должен иметь следующие свойства:*

1. Свойство **уникальности**. То есть не существует двух разных экземпляров сущности с одинаковым значением ключа.
2. Свойство **неизбыточности**. То есть удаление любого атрибута из составного ключа приводит к нарушению первого свойства.

С помощью потенциальных ключей выполняется поиск и сортировка экземпляров сущностей.

Потенциальных ключей может быть несколько. Например: *специальность (номер специальности, название специальности, название специализации, ...)*.

В этом случае для идентификации экземпляра один из них выбирается в качестве **первичного ключа** или **привилегированного идентификатора (ID)**. Остальные будут **альтернативными** ключами.

Не путайте потенциальные ключи и *индексные*, несмотря на то, что потенциальные могут быть, а первичные наверняка будут индексными. Дело в том, что индексные ключи могут быть *неуникальными*, и некоторые СУБД позволяют делать выборку множества экземпляров по такому ключу. Например, *отсортировать студентов факультета по значению суммарного рейтинга сессии*.

Между экземплярами сущностей могут устанавливаться некоторые соответствия или **отображения одного множества экземпляров (одной сущности) на другое или другие**. Такое соответствие называется **связью**. На практике каждая связь несёт некоторую смысловую нагрузку, задаваемую пользователем.

Связи могут быть *бинарными* — между двумя элементами, *тернарными* — между тремя элементами, …, *n-арными*, когда в связи принимают участие *n* сущностей. Нас, в первую очередь, будут интересовать *бинарные связи*. Кроме того, различают *рефлексивные* связи между экземплярами одной и той же сущности, *транзитивные* (опосредованные) связи и др.

По *типу* связи разделяют следующим образом (представим *все возможные типы* бинарных связей в виде таблицы — они нам потребуются при проектировании БД):

Безусловные (БУ)	1:1 **R1**	1:N **R2**	M:N **R3**
Условные (У)	$1:1_y$ **R4**	$1_y:N$ **R5** $1:N_y$ **R6**	$M:N_y$ **R7**
Биусловные (2У)	$1_y:1_y$ **R8**	$1_в:N_y$ **R9**	$M_y:N_y$ **R10**

Сущности, принимающие участие в связи, соответственно *1-, N- и M-связные*.

R1. Связь 1:1 (или отображение 1 к 1) — каждому экземпляру сущности *A* соответствует в точности 1 экземпляр сущности *B* (рис. 3):

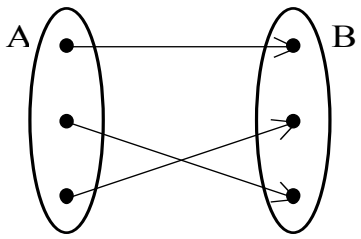


Рис. 3. Теоретико-множественное представление связи 1:1

Например: „кафедра — заведующий кафедры“.

Для *графического представления* связей между конкретными сущностями используются 2 подхода: *диаграммы „сущность — связь“* (или *ER*-диаграммы) и *информационные модели* (*ИМ*). ER-диаграмма выглядит так (рис. 4):

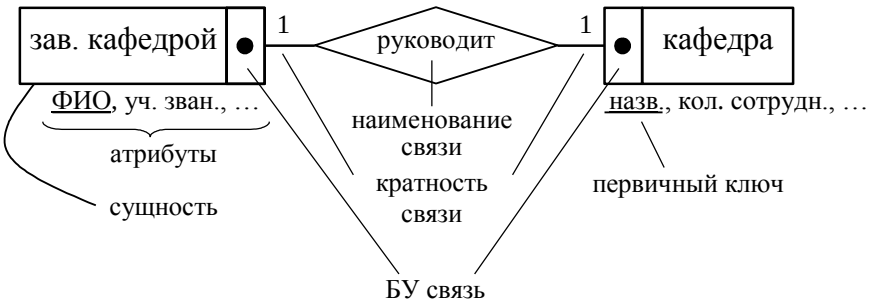


Рис. 4. ER-диаграмма примера связи 1:1

На ИМ сущности представляются в виде прямоугольников, которые содержат и название сущности, и атрибуты (рис. 5):

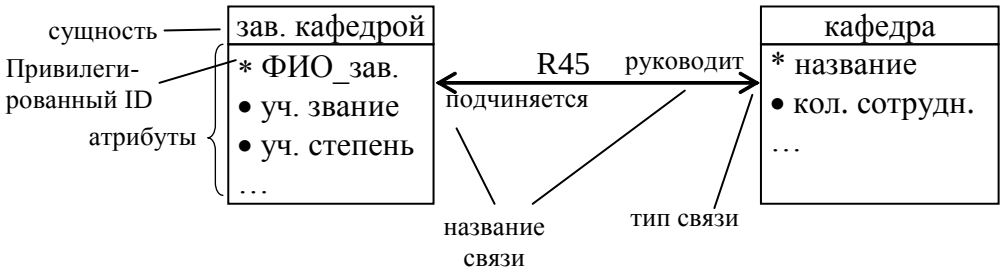


Рис. 5. Информационная модель примера связи 1:1

Де-факто существует ещё одна весьма распространённая нотация для представления связей между сущностями — это информационная модель, предложенная фирмой *Microsoft* и используемая в СУБД *Access* этой фирмы (рис. 6):

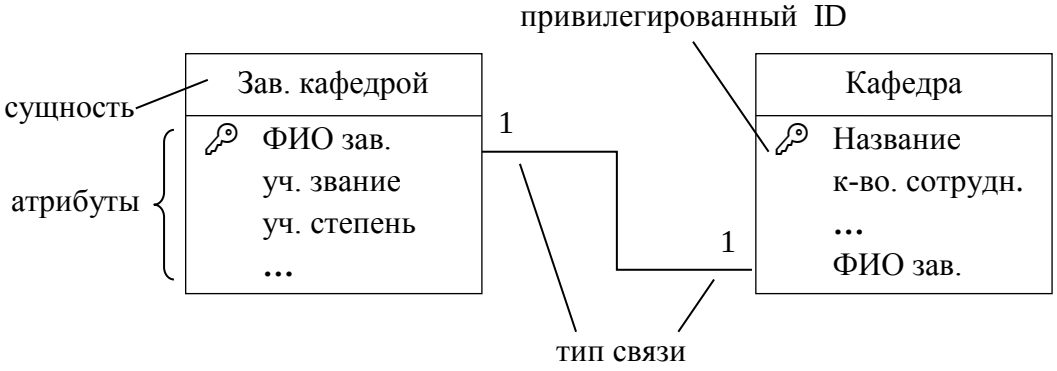


Рис. 6. Информационная модель MS Access примера связи 1:1

При моделировании предметной области и проектировании БД можно пользоваться любым из этих представлений. Более наглядно их различия будут показаны на этапе проектирования.

R2. Связь 1:N — каждому экземпляру сущности *A* соответствует от 1 до *N* экземпляров сущности *B* (рис. 7):

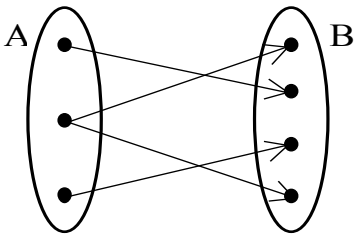


Рис. 7. Теоретико-множественное представление связи 1:N

Например: „порт — корабль“ (рис. 8)

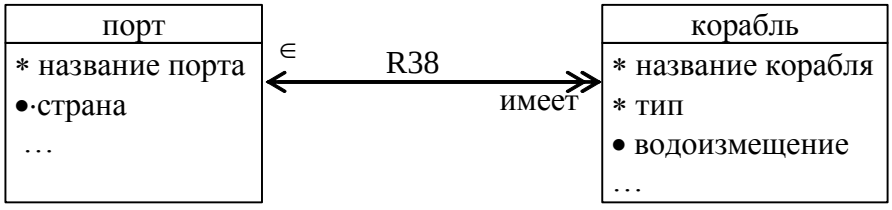


Рис. 8. Информационная модель примера связи 1:N

R3. Связь M:N — каждому экземпляру сущности *A* соответствует от 1 до *N* экземпляров сущности *B* и наоборот, каждый экземпляр сущности *B* соответствует от 1 до *M* экземплярам сущности *A* (рис. 9).

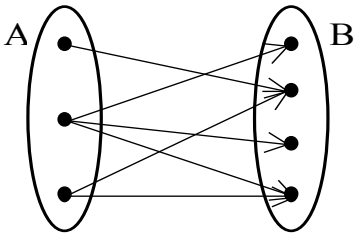


Рис. 9. Теоретико-множественное представление связи M:N

Например: „преподаватель — дисциплина“ (рис. 10).

Рис. 10. Информационная модель примера связи M:N

Условность в связях возникает тогда, когда некоторые экземпляры одной или обеих сущностей *не принимают участие* в связи. Например (рис. 11):

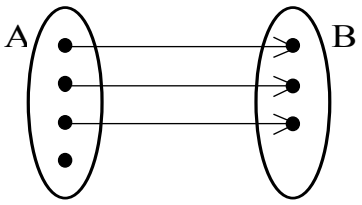


Рис. 11. Теоретико-множественное представление связи 1y:1

R4. Связь 1:1y: „студент — дипломный проект“ (рис. 12).

Рис. 12. ER-диаграмма примера связи 1y:1

R8. Связь 1y:1y: „Дисплей — ЭЛТ“ (рис. 13)

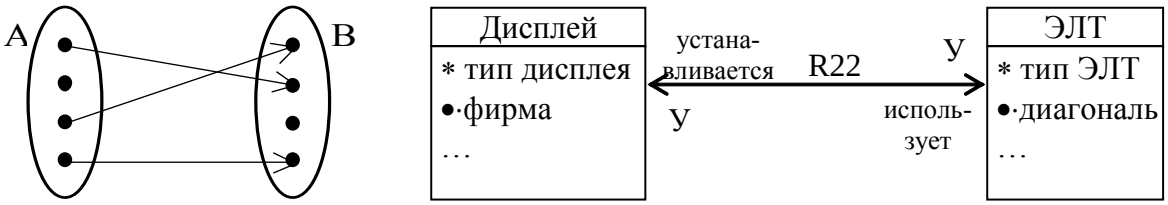


Рис. 13. Теоретико-множественное представление и информационная модель примера связи 1y:1y

R9. Связь 1y:Ny: „Кафедра — Лаборатория“ (рис. 14).

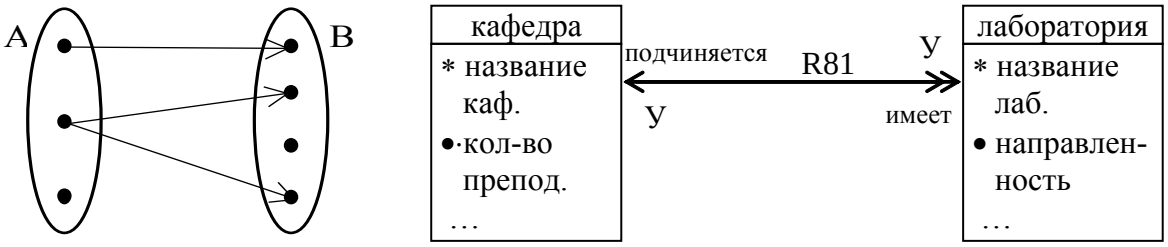


Рис. 14. Теоретико-множественное представление и информационная модель примера связи 1у:Ny

Необходимо отметить, что существует ещё один особый тип связи: *связь супертип-подтип*. Примеры такой связи представлены на рис. 15, рис. 16:

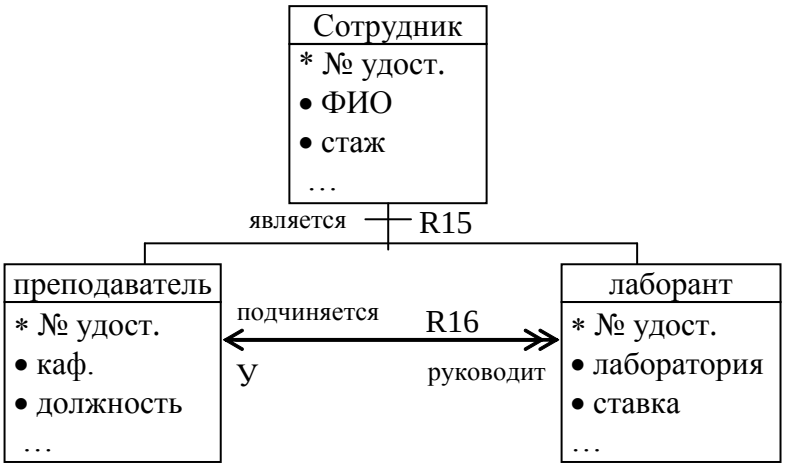


Рис. 15. Информационная модель примера связи супертип-подтип

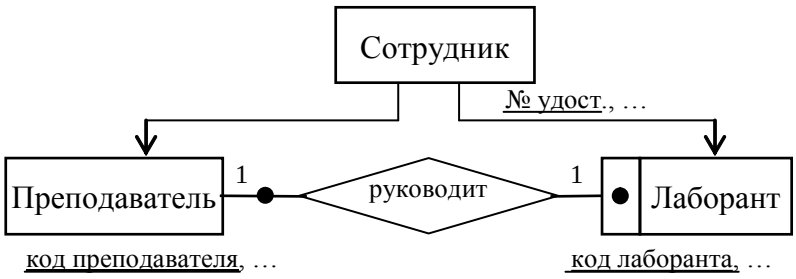


Рис. 16. ER-диаграмма примера связи супертип-подтип

Модели данных

Модель данных включает 3 составляющих [18]:

1. Структуру данных для представления точки зрения пользователя на БД.
2. Список операций, допустимых над данной структурой.

   Эти два пункта — основа языка данных конкретной модели.
3. Средства обеспечения защиты и целостности данных. С этой целью в модель вводятся некоторые ограничения. Например:
   а) каждое поддерево иерархической структуры данных должно иметь корневой (начальный) узел, и нельзя хранить порождённые узлы без родительского;
   б) каждая запись файла реляционной базы данных должна быть уникальной.

Существуют три *основных* модели данных:

1. Сетевые.
2. Иерархические.
3. Реляционные.

Первые две называют ещё *дореляционными* по времени их появления, хотя они той или другой мерой используются и сегодня. Сейчас появился ряд так называемых *постреляционных* моделей. В частности, объектно-ориентированная модель. Больше того, на сегодня создан ряд продуктов, например, *Oracle*, начиная с версии 8, в которых используются все элементы объектно-ориентированного подхода, хотя теория этой модели, как модели данных, в полной мере ещё не проработана. В частности, нет соответствующей математики для манипулирования элементами данных.

В литературе [4, 8, 9, 14, 15] можно встретить упоминание и о других *постреляционных* моделях: *дедуктивной, экспертной, многомерной, семантической* и др. Однако они представляют собой модификацию (в той или иной мере) или комбинацию трёх основных.

Коротко остановимся на *дореляционных* и нескольких *постреляционных* моделях и детально рассмотрим реляционную, как наиболее используемую.

Сетевая модель данных

Эта модель реализует *графовую форму* представления данных, при которой допускаются *произвольные связи* между типами сущностей: *вершины* графа — тип сущности, *дуги* — типы связей между сущностями.

Терминология, используемая при обработке данных в такой модели, была введена рабочей группой по базам данных ассоциации CODASYL (ассоциации по языкам обработки данных).

Этой группой были введены следующие термины (рис. 17):

— *схема*, описывающая структуру БД;
— *элемент данных* — наименьшая поименованная единица данных;
— *агрегат* — наименьшая совокупность элементов данных внутри записи, которую можно рассматривать как единое целое;
— соответственно, *запись* — наименьшая совокупность элементов и агрегатов данных:

Студент		
ФИО	Специальность	№ группы

— *набор* — служит для отображения связей между сущностями. Это наименьшая совокупность записей, которая отображает двухуровневую иерархическую структуру. При этом один тип записи объявляется *владельцем набора*, а другие *членами набора*.

Схема сетевой модели CODASYL состоит из множества типов записей, которые могут быть владельцами или членами типов наборов, определённых в этой схеме. Каждый тип набора определяет (представляет собой) связь между типами записи-члена и записи-владельца следующим образом:

1. Он определяет отображение 1:N между типами записей набора.
2. Каждый экземпляр набора содержит 1 экземпляр записи-владельца и произвольное число записей членов.

— *БД* — поименованная совокупность разных типов наборов, которая содержит связь между записями, представленными экземплярами наборов.

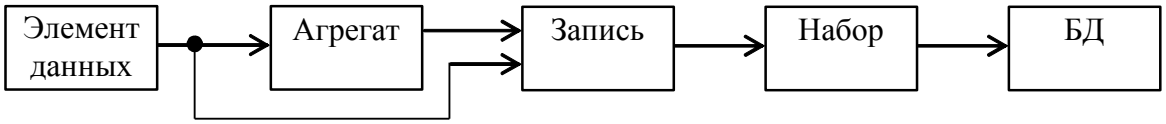


Рис. 17. Диаграмма зависимости между элементами модели CODASYL

Сетевая модель данных может быть описана в виде графа с *n*-арными связями. Например (рис. 18):

Рис. 18. Пример связи M:N в сетевой модели

Этот граф описывает связь M:N: множество студентов — множество дисциплин. Однако набор в сетевой модели CODASYL допускает только связи 1:N. Преобразование в этой модели выполняется путём введения *дополнительной записи* „специальность“ (рис. 19).

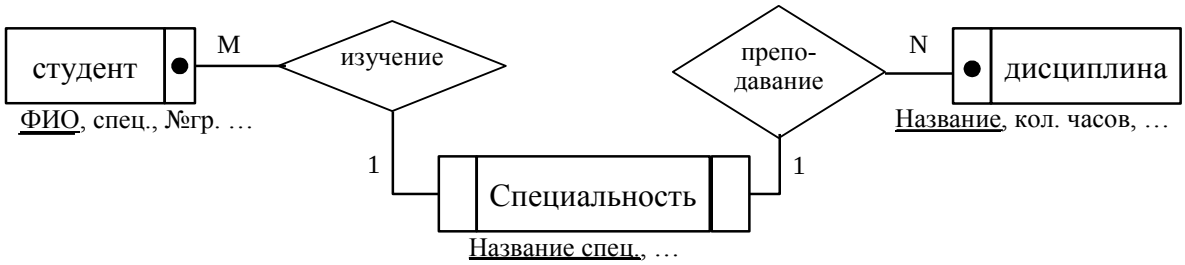


Рис. 19. Приведение связи M:N в соответствие к требованиям сетевой модели CODASYL

Другой вариант представления модели данных — описание её языком данных CODASYL. Так ЯОД CODASYL для сетевой модели содержит 4 пункта (статьи):

1. Статья *схемы*:
   SCHEMA NAME IS *имя_БД*
2. Одна или несколько статей *областей памяти*:
   AREA NAME IS *имя_области*
3. Статья *записи*:
   SET NAME IS *имя_записи* (например, „студент“)
   ФИО TYPE IS CHARACTER 30
   № гр. TYPE IS FIXED 3
4. Статья *набора*:
   SET NAME IS *имя_набора* (обучение)
   OWNER IS *имя_владельца* (специальность)
   MEMBER IS *имя_члена* (студент)

   SET NAME IS *имя_набора* (преподавание)
   OWNER IS *имя_владельца* (специальность)
   MEMBER IS *имя_члена* (дисциплина)

ЯМД содержит операторы операций над описанными элементами. Например:

FIND *имя_записи* RECORD USING *ключ*.

Иерархическая модель данных

Иерархическая модель также основывается на графовой форме представления данных. Однако здесь допустима только *древовидная структура* связей. Соответственно, графическая диаграмма схемы БД в иерархической модели называется деревом определения.

На внутреннем уровне такая БД представляется файлом, объединяющем экземпляры записей.

Простейший пример приложения иерархической модели данных — любая организация (рис. 20):

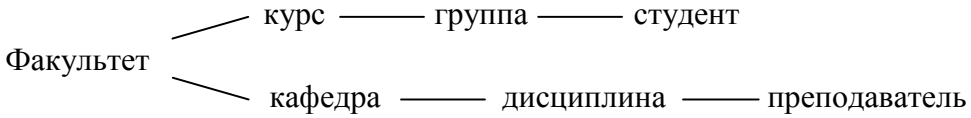


Рис. 20. Пример иерархии сущностей

Понятию „запись“ в сетевой модели эквивалентен термин *сегмент* в иерархической — наименьшая совокупность элементов данных.

Примером реализации этой модели может служить СУБД ***IMS***, созданная фирмой *IBM* в рамках проекта ***Apollo***.

Эта СУБД интересна тем, что на внутреннем уровне в ней реализованы 4 *способа хранения* (и, соответственно, *метода доступа*) данных, в которых осуществлялась попытка совместного хранения экземпляра *корневого сегмента* и экземпляров *всех* его сегментов-*потомков*:

HSAM — иерархический последовательный метод доступа (*Hierarchical Serial Access Method*). База данных сортируется по значению ключа каждого корневого сегмента. Сегменты хранятся в файле последовательно: прямом иерархическом порядке обхода дерева (*сверху — вниз, слева — направо*).

HISAM — иерархический индексно-последовательный метод доступа (*Hierarchical Index-Serial Access Method*) отличается тем, что в нем обеспечивается индексный доступ к корневым сегментам.

HDAM — иерархический прямой метод доступа (*Hierarchical Direct Access Method*). Экземпляры каждого типа записи размещаются в отдельном файле с прямой адресацией так, что можно непосредственно обратиться к любой записи. Иерархия обеспечивается указателями в сегментах.

HIDAM — иерархический индексно-прямой метод доступа (*Hierarchical Index-Direct Access Method*) — попытка объединить преимущества методов HISAM и HDAM. К корневым сегментам обеспечивается и индексный, и прямой доступ, а каждый из них содержит указатели на сегменты-потомки.

В отличие от сетевой модели для реализации отображения *M*:*N* здесь выполняется дублирование записей (рис. 21):

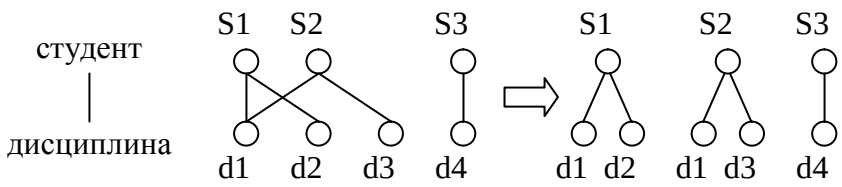


Рис. 21. Приведение связей M:N в соответствие к требованиям иерархической модели

Реляционная модель данных

Эта модель использует ещё одну форму представления БД — *табличную*, где данные оформлены в виде множества *двумерных таблиц*, каждая из которых описывает тот или иной *тип сущности* и её *свойства*. При этом *столбец* таблицы соответствует *одному* свойству или атрибуту сущности, а *строка — экземпляру сущности* с конкретными значениями атрибутов.

Например: сущность „дисциплина“

Название	Количество часов	Семестр	Экзамен
Физика	45	1	ЕСТЬ (T)
Физика	45	2	ЕСТЬ (T)
ВТ и программирование	60	3	НЕТ (F)
Ассемблер	30	3	НЕТ (F)
Ассемблер	30	4	ЕСТЬ (T)

Множество *всех возможных* значений какого-нибудь атрибута носит название *домен*. Например: атрибуту „Название“ сущности „Дисциплина“ соответствует домен, содержащий названия *всех дисциплин*, преподаваемых в ВУЗе; в домен, соответствующий количеству часов, входят *фиксированные значения 18, 36, 54, 72, …, 180*...; семестру соответствует домен с *натуральным рядом чисел 1-12*; а домен, из которого берутся значения для атрибута „Экзамен“ включает только *2 значения — ЕСТЬ/НЕТ* (True/False).

Подмножество значений домена, присутствующих в таблице(-ах) БД называется *активным доменом*.

Между сущностями и, соответственно, между таблицами, могут существовать или устанавливаться связи, получившие название отношений. Однако в этой модели данных есть особенность:

Под отношением понимается некоторое *подмножество декартова произведения* списка доменов.

Из этого определения следуют 2 вывода:

1. Отношения (связи) устанавливаются не просто между сущностями, а между их отдельными атрибутами или подмножествами атрибутов.
2. Сама таблица может рассматриваться как отношение, которое определяет набор связей, допустимых и/или имеющих смысл, между атрибутами. Отсюда название *реляционные* (*relation* – отношение) базы данных.

Пример: *декартово произведение* множеств $D1 = \{1, 2\}$ и $D2 = \{a, b, c\}$ даёт таблицу с двумя столбцами, которая и является *отношением*:

D1	D2
1	a
1	b
1	c
2	a
2	b
2	c

Строка такой таблицы называется *кортежем* отношения.

Итак, на разных уровнях архитектуры СБД в реляционной модели используются следующие термины:

Концептуальная схема	Внешняя схема	Внутренняя схема
Отношение	Таблица	Файл или таблица
Кортеж	Строка	Запись
Атрибут	Столбец	Поле

Вектор имён атрибутов называют *схемой отношения*: $REL(A_1, A_2, \ldots, A_n)$. Количество этих атрибутов или количество доменов называют *степенью отношения*, а количество кортежей — *кардинальным* числом или *мощностью*. Степень отношения обычно не меняется после создания отношения, а мощность будет колебаться по мере добавления или удаления кортежей.

Вскоре после своего появления идея (и теория) реляционных баз данных стала популярной среди разработчиков СУБД. Однако сделать реляционную СУБД оказалось непросто. Сложилась неоднозначная ситуация, когда после некоторых усовершенствований одни фирмы стали называть свои разработки реляционными (иногда просто прибавляя ‘/R’ к имени своей СУБД), а другие отказываться от создания реляционных СУБД в силу сложности задачи. Для того чтобы внести ясность в оценку разработок одних фирм и более выразительно сформулировать цель, к которой разработчикам нужно стремиться, для других (или тех же самых) фирм, *Е. Кодд*, автор реляционного подхода, в конце 70-х гг. опубликовал свои 12 правил соответствия произвольной СУБД реляционной модели (позднее появились 12 правил *К. Дейта* оценки соответствия системы распределённой СУБД, а в начале 90-х — 12 правил *E.F. Codd & Associates* соответствия системы системам оперативной аналитической обработки данных, то есть *OLAP*-системам), дополнив основные понятия реляционных баз данных определениями, важными для практики. Ниже приводятся эти правила вместе с общим положением, которое разъясняет и дополняет их [15].

0. **Основное (фундаментальное) правило.** Система, которая рекламируется или провозглашается поставщиком как реляционная СУБД, должна управлять базами данных исключительно способами, которые соответствуют реляционной модели.
1. **Информационное правило.** Вся информация, хранимая в реляционной базе данных, должна быть явным образом, на логическому уровне, представлена единственным образом: в виде значений в *R*-таблицах.
2. **Правило гарантированного логического доступа.** К каждому существующему в реляционной базе атомарному значению должен быть гарантирован доступ с помощью указания имени *R*-таблицы, значения первичного ключа и имени столбца.
3. **Правило наличия значения (missing information).** В полностью реляционной СУБД должны быть специальные индикаторы (отличные от пустой символьной строки или строки с одних пробелов и отличные от нуля или какого-то другого числового значения) для выражения (на логическом уровне, систематически и независимо от типа данных) того факта, что значение отсутствует по меньшей мере по двум разным причинам: его действительно нет или оно неприменимо к данной позиции. СУБД должная не только отражать этот факт, но и распространять на такие индикаторы свои функции манипулирования данными вне зависимости от типа данных.
4. **Правило динамического диалогового реляционного каталога.** Описание базы данных выглядит логически как обычные данные, так что авторизованные пользователи и прикладные программы могут использовать для работы с этим описанием тот же реляционный язык, что и при работе с обычными данными.
5. **Правило полноты языка работы с данными.** Как бы много в СУБД ни поддерживалось языков и режимов работы с данными, должен быть, по крайней мере, один язык, выражаемый в виде командных строк в некотором удобном синтаксисе, который бы позволял формулировать:
   — определение данных;

— определение правил целостности;
— манипулирование данными (в диалоге и с программы);
— определение выведенных таблиц (в том числе возможности их модификации);
— определение правил авторизации;
— границы транзакций.

6. **Правило модификации таблиц-представлений.** В СУБД должен существовать корректный алгоритм, позволяющий автоматически для каждой таблицы-представления определять во время её создания, может ли она использоваться для вставки и удаления строк, какие из столбцов допускают модификацию, и заносящий полученную таким образом информацию в системный каталог.
7. **Правило множественности операций.** Возможность оперирования базовыми или выведенными таблицами распространяется целиком не только на выдачу информации с БД, но и на вставку, модификацию и удаление данных.
8. **Правило физической независимости.** Диалоговые операторы и прикладные программы на логическом уровне не должны страдать от каких-нибудь изменений во внутреннем хранении данных или в методах доступа СУБД.
9. **Правило логической независимости.** Диалоговые операторы и прикладные программы на логическом уровне не должны страдать от таких изменений в базовых таблицах, которые сохраняют информацию и теоретически допускают неизменность этих операторов и программ.
10. **Правило сохранения целостности.** Диалоговые операторы и прикладные программы не должны меняться при изменении правил целостности в БД (задаваемых языком работы с данными и хранящихся в системном каталоге).
11. **Правило независимости от распределённости.** Диалоговые операторы и прикладные программы на логическом уровне не должны страдать от осуществляемого физического разнесения данных (если сначала СУБД работала с нераспределёнными данными) или перераспределения (если СУБД действительно распределённая).
12. **Правило ненарушения реляционного языка.** Если в реляционной СУБД есть язык низкого уровня (для работы с отдельными строками), она не должна позволять нарушать или „обходить“ правила, сформулированные языком высокого уровня (множественной) и занесённые в системный каталог.

Важность правил Кодда в том, что, будучи сформулированными около 30 лет тому, они никем не отрицались, не дополнялись и до сих пор являются единственными правилами такого рода. Несмотря на то, что не все они равноценны, а некоторые носят „печать времени“ своего появления, эти правила на протяжении продолжительного периода задают определённую точку отсчёта для одних (разработчики) и критерий соответствия для других (разработчики и пользователи).

Реляционная алгебра

Над отношениями необходимо уметь выполнять операции, обеспечивающие построение наиболее эффективных и максимально корректных запросов к БД с целью определения области выборки данных, области обновления данных (то есть данных для вставки, изменения или удаления), правил и требований безопасности, устойчивости и целостности данных при множественном доступе к ним и т.п.

Теория реляционной модели является сегодня наиболее полно проработанной, потому что в ней математически чётко определены такие операции. Эти операции получили название операций *реляционной алгебры* [4].

В 1972 г. *Кодд* определил 8 операций реляционной алгебры, распределённых на 2 группы:

1. *Традиционные операции* над множествами, модифицированные с учётом того, что операндами являются отношения: декартово произведение, объединение, пересечение, разность.
2. *Специальные реляционные операции*: выборка, проекция, соединение и деление.

Пять с этих восьми операций — декартово произведение, проекция, объединение, разность и выборка — являются *базовыми* и составляют *минимальный набор*. Это означает, что через них могут быть определены любые другие, в том числе и остальные три операции реляционной алгебры. Эти три операции были введены поэтому, что они настолько часто используются, что имело смысл обеспечить их непосредственную поддержку.

Прежде, чем рассмотреть операции, отметим свойство, без которого набор математических операций не является алгеброй. Это **свойство замкнутости**, которое соответствует третьему принципу концепции типа (см. стр. 8) и утверждает, что результат любой операции имеет тот же тип, что и операнд(ы) этой операции. Соответственно, *реляционное свойство замкнутости* утверждает, что **результат каждой операции над отношением** (то есть реляционной операции) **также является отношением**.

Из этого свойства следует вывод, что **результат одной реляционной операции может использоваться в качестве входных данных для другой**, и существует возможность формировать *вложенные выражения*.

Итак:

1. *Декартово произведение*, определённое раньше, несколько модифицируется, так как благодаря свойству замкнутости, результат операции должен содержать кортежи, а не упорядоченные пары кортежей. Поэтому, **результатом декартового произведения или просто произведения отношений *R* и *S* будет отношение, состоящее из множества кортежей, образованных конкатенацией (сцеплением) каждого кортежа отношения *R* с каждым кортежем отношения *S***:

$$R \times S = \{ r.\, s \mid r \in R, s \in S \}$$

Пример

R

A	B	C
X	1	a
Y	2	a
Z	3	a
W	4	b
W	5	b
W	6	b

S

D	E
10	f
5	e
8	d
4	c
6	b
3	a

R×S

A	B	C	D	E
x	1	a	10	f
x	1	a	5	e
…	…	…	…	…
w	4	b	4	c
w	4	b	6	b
…	…	…	…	…
w	6	b	3	a

Обратите внимание, что схема результирующего отношения также образована конкатенацией схем сомножителей. Поэтому основным требованием этой операции является отсутствие в схемах исходных отношений атрибутов с одинаковыми *именами*.

Данная операция не очень важная на практике, потому что не даёт никакой дополнительной информации сравнительно с первоначальной. Однако она является одной из

базовых и лежит в основе очень широко используемой операции соединения. В *SQL* произведение не имеет соответствующих команд.

2. *Проекция*. Пусть r — кортеж из отношения R; $r[M]$ — часть этого кортежа, которая содержит только значение атрибутов, входящих в подмножество M схемы отношения. Тогда **проекцией R на M будет отношение, состоящее из кортежей значений тех атрибутов, которые входят в множество M**:

$$R[M] = \{r[M] \mid r \in R\}$$

На практике это означает удаление из отношения R атрибутов, которые не входят в множество M, со последующим исключением из полученного отношения одинаковых кортежей. Пример:

R[C]

C
a
b

R[A,C]

A	C
x	a
y	a
z	a
w	b
~~w~~	~~b~~
~~w~~	~~b~~

Эта операция также не имеет прямого аналога в *SQL*, однако её можно выполнить, например, с помощью простейшего варианта команды SELECT:

```
SELECT [DISTINCT] C FROM R;
SELECT [DISTINCT] A,C FROM R;
```

Эта команда по умолчанию не удаляет дублирующиеся кортежи. Для этого необходимо после SELECT самостоятельно добавить оператор DISTINCT.

3. *Объединение*. Это операция получения **отношения, состоящего из кортежей, принадлежащих либо отношению R, либо отношению S, либо обоим:**

$$R \cup S = \{r \mid r \in R \vee r \in S\}$$

Очевидно, что оба исходных отношения должны иметь *совместимые типы атрибутов*. Пример

$R[B,C] \cup S$

1	a
2	a
3	a
4	b
5	b
6	b
10	f
4	c

Из отношения S необходимо удалить 2 последние строки.

В данном примере не видно, какой будет схема результирующего отношения. Поэтому многие языки данных, в том числе, некоторые диалекты *SQL*, в частности, *PostgreSQL* выдвигают ещё более жёсткое требование — исходные отношения должны иметь *одинаковые схемы*. Тогда и результат будет иметь такую же схему.

В *SQL* объединение выполняет команда UNION:

```
SELECT B, C FROM R
      UNION
SELECT * FROM S;
```

4. *Разность*. Это операция получения **отношения, состоящее из кортежей отношения *R*, не входящих в отношение *S*:**

$$R - S = \{r \mid r \in R \wedge r \notin S\}$$

Требования к атрибутам здесь те же, что и для операции объединения. Пример:

$R[B,C] - S$

1	a
2	a
4	b
5	b

Прямого аналога в *стандартном SQL* нет, однако, большинство *диалектов* имеют команду MINUS или SUBTRACT:

```
SELECT B, C FROM R
      MINUS
SELECT D, E FROM S;
```

В ***PostgreSQL*** эту операцию реализует команда EXCEPT — *исключение*, имеющая аналогичный формат.

5. *Выборка (ограничение)*. Если проекция строит отображение значений одного атрибута на другие (одного столбца таблицы на другие или на самого себя), то выборка выполняет отображение кортежей, результатом которого является **отношение, содержащее подмножество всех кортежей отношения *R*, для которых выполняется (является истинным) некоторое логическое условие**:

$$G_F(R,c) = \{r \mid r \in R \wedge F(r[M],c), c = const \vee c \in S\}$$

F – это любая функция, чаще, операция сравнения. Поэтому сравниваемые значения должны быть определены в одном и том же домене, а *F* должна иметь смысл для этого домена. Например, нелепо сравнивать на *больше* (<) *или меньше* (>) значение атрибута „цвет“, хотя некоторые СУБД допускают и это (например, в зависимости от порядка следования при определении).

В *SQL* операция ограничения реализована в виде оператора WHERE.

1) $G_{=}(R[A],'z')$:

```
SELECT * FROM R
      WHERE A = 'z';
```

A	B	C
z	3	a

2) $G_{\neq}(R[B],S[D])$:

```
SELECT A, B, C FROM R, S
      WHERE R.B <> S.D;
```

A	B	C
x	1	a
y	2	a

Во втором примере приведен ещё один интересный элемент *SQL — префикс имени поля* в виде имени таблицы, отделённого точкой: ‘*R.*’ и ‘*S.*’. Префиксы нужны для уточнения имён полей, если они совпадают (одинаковы) в разных таблицах. Если имена атрибутов разные, префиксы можно опустить.

6. *Пересечение*. Результатом этой операции является **отношение, состоящее из кортежей, принадлежащих и отношению *R*, и отношению *S***:

$$R \cap S = \{r \mid r \in R \wedge r \in S\}$$

Требования к атрибутам совпадают с требованиями операций объединения и разности. Пример:

$R[B,C] \cap S$

3	a
6	b

В *SQL* пересечение выполняет команда INTERSECT:

```
SELECT B, C FROM R
    INTERSECT
SELECT * FROM S;
```

которая в *PostgreSQL* имеет наиболее жёсткие требования к схемам отношений

Кроме того, эту операцию можно выразить через пошаговое выполнение операции разности:

$$R \cap S = R - (R - S)$$

7. *Деление*. Эта операция имеет дело с отношениями, в которых схема одного $(S(M))$ является правильным подмножеством схемы другого $(R(N.M))$. Например $S(D,E)$ и $R(A,B,C)$, где $B \sim D$ и $C \sim E$. Результатом операции деления является **отношения, схема которого является схемой отношения *R* за исключением схемы отношения *S*, то есть $R[N]$, и которое составлено из таких кортежей отношение *R*, что для <u>всех</u> кортежей отношения *S* в отношении *R* есть кортежи с одинаковыми значениями атрибутов из множества *N***:

$$R/S = \{r[N] \mid \forall s \in S\ \exists r \in R | r[M] = s\},$$

где *M* — схема отношения *S*, *N.M* — схема отношения *R*.

То есть $r[N]$ — одинаковая часть кортежей отношения *R*, в которых другой частью являются все кортежи отношения *S* (каждый из них). Для иллюстрации примера добавим к отношению *R* ещё один кортеж:

R

A	B	C
x	1	a
y	2	a
z	3	a
w	4	b
w	5	b
w	6	b
w	3	a

тогда:

$R / G_{=}(S[D], 6 \vee 3)$

A
w

Обратите внимание, что в этом примере предварительно выполнена операция выборки по отношению *S*, благодаря которой получена такая таблица:

D	E
6	b
3	a

В отношении *R* есть только одно значение *w* атрибута *A*, который составляет подмножество *N* схемы отношения, образующее кортежи отношения *R* путём конкатенации с *каждым* из двух кортежей отношения-делителя (для сравнения: значение *z* — только с одним из этих кортежей).

В реальных диалектах SQL для выполнения этой операции нет соответствующей команды, хотя в стандарте она приведена как команда DIVIDE или DIVIDE BY:

R DIVIDE S;

Однако её реализация возможная путём комбинации операций декартового произведения, разности и проекции:

$$R/S = R[N] - ((R[N] \times S) - R)[N].$$

И наконец, рассмотрим ещё одну операцию, которая хоть и не относится к базовым, но используется настолько часто, что имеет своё отображение в большинстве версий *SQL*: операция *соединения*. Для представления этой операции сразу рассмотрим пример.

```
SELECT    R.A, R.B, R.C, S.D      FROM      R, S
      WHERE R.C = S.E;

SELECT A, B, C FROM R
            WHERE C = E AND B = D;
```

A	B	C
z	3	a
w	6	b

A	B	C	D
x	1	a	3
y	2	a	3
z	3	a	3
w	4	b	6
w	5	b	6
w	6	b	6

До появления стандарта *SQL*/92 инструкции такого вида были единственным вариантом построения соединения. Выполнение этой операции состоит из следующих шагов:

1. Построение декартового произведения отношений R и S:
   $R \times S$
2. Выполнение выборки тех кортежей произведения, в которых совпадают значения подмножеств M и N атрибутов (C и E — в первом примере, C и E, и B и D — во втором):
   $G_{=}((R \times S)[M],(R \times S)[N])$
3. Проекция на все атрибуты произведения за исключением одного из подмножеств M или N:
   $(G_{=}((R \times S)[M],(R \times S)[N]))[REL(R).(REL(S) - N)]$

Команда JOIN, добавленная в *SQL*/92, работает *только в том случае*, если схемы соединения отношений имеют одинаковые подмножества атрибутов (M = N). То есть, в отношении S вместо D, E будет записано B и C.

```
R JOIN S USING C;
```

Это соединение по атрибуту C.

Соединение по всем таким подмножествам носит название **естественного**:

```
R NATURAL JOIN S;
```

Однако в таком виде операция соединения не реализована ни в одной СУБД. Вместо неё используются её 4 расширения: внутреннее, а также левое, правое и полное внешнее соединения.

Приведенное выше определение полностью соответствует внутреннему соединению.

Любое внешнее соединение включает в себя кортежи внутреннего соединения, а также те кортежи одного или обоих отношений, для которых не выполнено условие второго шага алгоритма — выборки (то есть, нет „соответствующего“ кортежа в другом операнде этой операции).

Таким образом, множество кортежей результирующего отношения левого внешнего соединения определится, как объединение множеств кортежей результата внутреннего соединения и кортежей левого (первого) отношения R, дополненных, точнее, сцепленных с кортежами, имеющими пустые значения атрибутов второго отношения *S*:

$$\{r.s \mid G_{=}((R \times S)[M],(R \times S)[N])\} \cup$$

$$\cup\{r.e_S \mid r \in R \,\&\, R[M] - S[N], e_S - \text{пустой кортеж со схемой } S\}.$$

Аналогично, множество кортежей результирующего отношения правого внешнего соединения определится, как объединение множеств следующих кортежей:

$$\{r.s \mid G_{=}((R \times S)[M],(R \times S)[N])\} \cup$$

$$\cup\{e_R.s \mid s \in S \,\&\, S[N] - R[M], e_R - \text{пустой кортеж со схемой } R\}$$

Полное внешнее соединение определится, как объединение всех описанных выше множеств кортежей:

$$\{r.s \mid G_{=}((R \times S)[M],(R \times S)[N])\} \cup$$

$$\cup\{r.e_S \mid r \in R \,\&\, R[M] - S[N], e_S - \text{пустой кортеж со схемой } S\} \cup$$

$$\cup\{e_R.s \mid s \in S \,\&\, S[N] - R[M], e_R - \text{пустой кортеж со схемой } R\}$$

Примеры этих видов соединений будут рассмотрены в соответствующем разделе при описании их реализации средствами языка *SQL*.

Реляционная модель данных, несмотря на её преимущества, совсем не идеальна. В ряде случаев она не позволяет ясно (или вообще) отразить особенности предметной области: всего лишь одной из иллюстраций этому служит отсутствие прямых средств выражения иерархии. Поэтому постоянно ведутся поиски других моделей, также имеющих свои сильные и слабые стороны. В связи с этим, можно коротко упомянуть о нескольких из них [4, 9].

Объектная модель

Моделью данных, которая привлекает возрастающее внимание с конца 80-х гг., является объектная, или „объектно-ориентированная“ модель. Верней было бы говорить об „объектной модели“ и „объектно-ориентированных реализациях“ этой модели на ЭВМ. Оправданием термина „объектно-ориентированная модель“ может служить, во-первых, традиция и, во-вторых, отсутствие общепринятого определения объектной модели. Основными понятиями, с которыми оперирует эта модель, являются следующие:

— объекты, обладающие внутренней структурой и однозначно идентифицирующиеся уникальным внутрисистемным ключом;
— классы, являющиеся, по сути, типами объектов;
— операции над объектами одного или различных типов, названных „методами“;
— инкапсуляция структурного и функционального описания объектов, позволяющая разделять внутреннее и внешнее описания (в терминологии предшествующего объектному структурного программирования — „модульность“ объектов);
— наследование внешних свойств объектов на основании соотношения „класс-подкласс“.

К преимуществам объектно-ориентированной модели обычно относят:

— возможность для пользователя системы определять свои сколько угодно сложные типы данных (используя имеющийся синтаксис и свойства наследования и инкапсуляции);
— наличие наследования свойств объектов;
— повторное использование программного описания типов объектов при обращении к другим типам, ссылающимся на них.

К недостаткам объектно-ориентированной модели можно отнести:

— отсутствие строгих определений; разное понимания терминов и расхождения в

терминологии;
— как следствие эта модель не исследована настолько тщательно математически, как реляционная;
— отсутствие общеупотребимых стандартов, позволяющих связывать конкретные объектно-ориентированные системы с другими системами работы с данными.

Некоторые специалисты основным и главным отличием объектно-ориентированной модели от реляционной считают наличие уникального системного идентификатора. Это различие связано с одним интересным семантическим явлением. Дело в том, что в реляционной модели объект полностью описывается его атрибутами. Если человек в таблице представлен именем и номером телефона, то, что происходит после замены номера телефона в существующей строке? Идёт ли после этого речь о том же самом человеке или о другом? В реляционной модели нет средств получить ответ на этот вопрос; в объектно-ориентированной его даёт системный идентификатор, который не изменился. С другой стороны, мы можем „заменить“ в базе данных одного сотрудника на другого, сохранив все связи и атрибуты прежнего, но при этом системный идентификатор изменится. Ясно, что иметься в виду будет совсем другой человек.

Объектно-реляционная модель

Объектно-реляционная модель стала попыткой соединить преимущества объектно-ориентированной модели с преимуществами реляционной модели, в частности проработанностью её математики и распространённостью этой модели благодаря существующими реляционными СУБД.

Термин „объектно-реляционные базы данных“ стал известен разработчикам и пользователям СУБД после выпуска компанией *Informix* в конце 1996 г. своего нового продукта *Informix Universal Server* за счёт приобретения СУБД *Illustra* и одноименной компании. В свою очередь, компания *Illustra* была основана, преимущественно, с целью коммерциализации экспериментальной СУБД *Postgres*, ядро которой вошло и в СУБД *PostgreSQL*.

В стандарте *SQL*’99 объектная модель включает две основных компоненты:
— структурные, определённые пользователями типы данных (*User Defined Type — UDT*);
— типизированные таблицы (*Typed Table*).

Первый компонент позволяет определять новые типы данных, которые могут быть намного более сложными, чем встроенные типы данных *SQL*. При определении структурного *UDT* нужно специфицировать не только элементы данных, но и его поведение на основе интерфейса вызовов методов. Второй компонент позволяет определять таблицы, строки которых являются экземплярами (или значениями) *UDT*, с которым явным образом ассоциируется таблица. Также, начиная с *SQL-99*, поддерживается возможность использования типов данных, значение которых являются коллекциями значений других типов, например, массивы, списки и множества.

Из указанных выше понятий объектно-реляционных свойств в СУБД *PostgreSQL* реализовано *UDT* и массивы. Также поддерживается обработка *XML*-структур.

Дополнительно в СУБД *PostgreSQL* существует механизм создания объектно-реляционных связей между таблицами.

Объектно-ролевая модель

Ещё одной моделью данных, которая имеет конкретную реализацию (система

InfoModeller), является объектно-ролевая модель, предложенная ещё в начале 70-х гг., однако выведенная за рамки академических исследований совсем недавно коллективом фирмы *Asymetrix*. В отличие от реляционной модели в ней нет атрибутов, а основные понятия это объекты и роли, которые их описывают. Роли могут быть как „изолированные“, присущие исключительно какому-нибудь объекту, так и существующие как элемент какого-нибудь отношения между объектами. Модель, по словам авторов, служит для понятийного моделирования, которое отличает её от реляционной модели. Есть и другие отличия и интересные особенности: например, для неё, кроме графического языка, разработано подмножество естественного языка, который не допускает неоднозначностей, и, таким образом, пользователь (заказчик) не только общается с аналитиком на естественном языке, но и видит представленный тем же языком результат его работы по формализации задачи. Можно заметить, что многие пользователи, в отличие от аналитиков, с работой разбираются в рисунках, описывающих их деятельность, и схемах. Объектно-ролевая модель сейчас привлекает большое внимание специалистов, однако до промышленных масштабов её использования, сравнимых с двумя предыдущими, ей пока далеко.

Комбинированные модели данных. Переход от одной модели к другой

Теоретически упомянутые модели данных, а также большинство неупомянутых равносильны в том смысле, что все, что можно выразить в одной из них, можно выразить и в других. Расхождение, однако, составляет то, насколько удобно использовать ту или другую модель человеку-проектировщику для работы с реальными жизненными задачами, и то, насколько эффективно можно реализовать работу с конкретной моделью на ЭВМ (если это возможно вообще). Как уже было сказано, однозначно общеупотребительной модели сейчас нет (и, очевидно, не будет никогда) и, в результате, сосуществуют разные модели. Больше того, они сосуществуют взаимосвязано, или же попытки такого взаимосвязывания (вплоть до объединения) неустанно осуществляются.

Много дебатов, например, ведётся по вопросу, совместимы ли и, если да, то каким образом, реляционная и объектная модели. Существуют мнения, что они взаимоисключают, и что они взаимодополняют друг друга. Последнего мнения придерживается, например, такой авторитет в области теории баз данных, как К. Дейт [4]. Согласно Дейту синергия двух моделей могла бы („должна“ по утверждению автора) базироваться на формуле *область определения атрибута = класс объектов*. Другими словами, атрибутами в реляционных таблицах могут быть объекты произвольно заданной сложности. С точки зрения реляционной модели они остаются атомарными, а все возможности работы с ними, возникающие из наличия внутренней структуры, реализуются объектно-ориентированными методами. Существует выбор, какие свойства предметной области моделировать реляционными методами (то есть моделировать таблицами, связанными друг с другом ключами), а какие объектными, но это уже является проблемой разработчика базы данных: теория здесь лишь предоставляет возможности такого выбора. Предложение Дейта не противоречит ни реляционному, ни объектному подходу и выглядит теоретически обоснованным. Но одновременно оно противоречит другому подходу, который основывается на формуле *класс объектов = таблица*, когда с объектом связывается строка таблицы. Этот подход получил распространение в практике производителей объектно-ориентированных СУБД.

Другой аспект взаимной связи указанных двух моделей носит реализационный характер. Некоторые объектно-ориентированные системы сами реализованы на объектно-реляционных СУБД как на системах, получивших доминирующее распространение на рынке СУБД, и вследствие этого наиболее продвинутых как промышленные изделия. В таких системах определения, заданные в рамках объектного подхода, переводятся в реляционные

определения, или наоборот, объектно-ориентированные определения строятся как надстройки над реляционными системами.

Переход от объектно-ролевой модели к реляционной был заложен создателями в основу реализации первой. Объектно-ролевая модель согласно такой позиции рассматривается как понятийная, а реляционная модель — как реализационная. Трансформация определений „объектов-ролей“ в реляционные определения не просто возможна, но и изначально заложенная в *InfoModeller*, причём гарантируется высокое качество результата такого преобразования: полученные таблицы имеют так называемую „5-ту нормальную форму“, что считается более качественной в реляционном плане, чем обычно необходимая в реляционных СУБД „3-я нормальная форма“.

Для реализации многомерных структур данных в рамках существующих промышленных СУБД необходима была интеграция сетевой и реляционной моделей. В ней стандартными средствами реляционных СУБД и их языков организуются „ссылочные связи“ между таблицами таким образом, чтобы они образовывали ещё несколько измерений. На рис. 22 приведен пример трёхмерного куба, а на рис. 23 — его реализация с помощью таблицы ссылок.

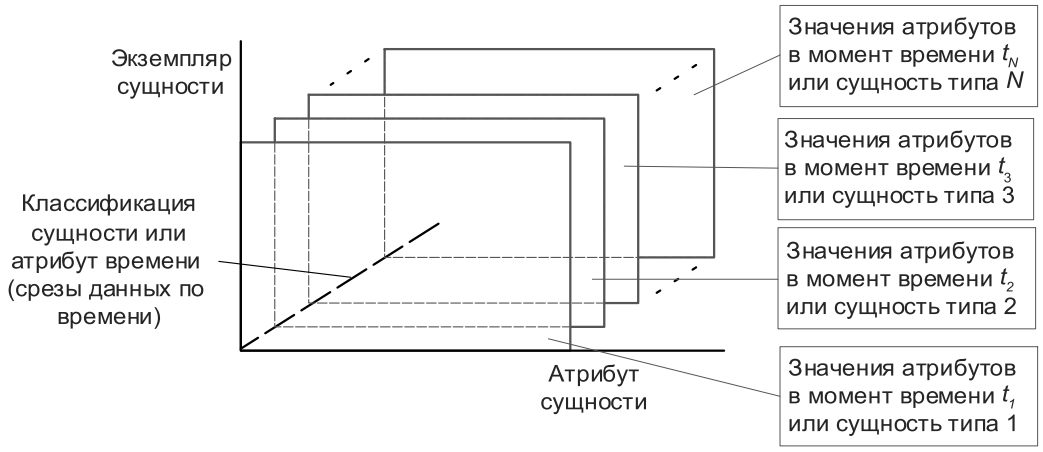


Рис. 22. Пример трёхмерного куба

Для манипулирования информацией, которая хранится в таких многомерных структурах, введены следующие основные операции:

- ***сечение*** — формирование подмножества куба (или гиперкуба), в котором значение одного или более измерений фиксированы (применение этой операции к трёхмерной структуре даст обычную реляционную таблицу) [10];
- ***вращение*** — изменение порядка представления измерений, обеспечивающее представление гиперкуба в более удобной для восприятия и анализа форме (изменение положения измерений для лучшей визуализации) [10];
- ***разбиение с поворотом*** даёт возможность проанализировать данные с разных точек зрения. Например, один срез содержит данные о студентах каждой специальности, а другой — данные о студентах каждого курса. Разбиение с поворотом часто выполняется вдоль оси времени с целью анализа тенденций и поиска закономерностей [9];
- ***консолидация (свёртка)*** содержит такие обобщающие операции, как суммирование значений (свёртка) или расчёты с использованием сложных вычислений, содержащих другие связанные данные (например, показатели по отдельным кафедрам могут быть „просуммированы“ до показателей факультетов, а они, в свою очередь, „свёрнуты“ к показателям университета) [9];

- ***детализация (операция спуска; нисходящий анализ)*** — операция, обратная консолидации, содержит отображения подробных сведений для рассмотренных консолидированных данных [9].

Очевидно, что для выполнения двух последних операций должна существовать иерархия значений измерений, то есть некоторая подчинённость одних значений другим, что легко описывается иерархической моделью.

Вся работа с гиперкубом, кроме рассмотренных прежде операций реляционной алгебры, сводится к различным его поворотам, группировкам и т.п.

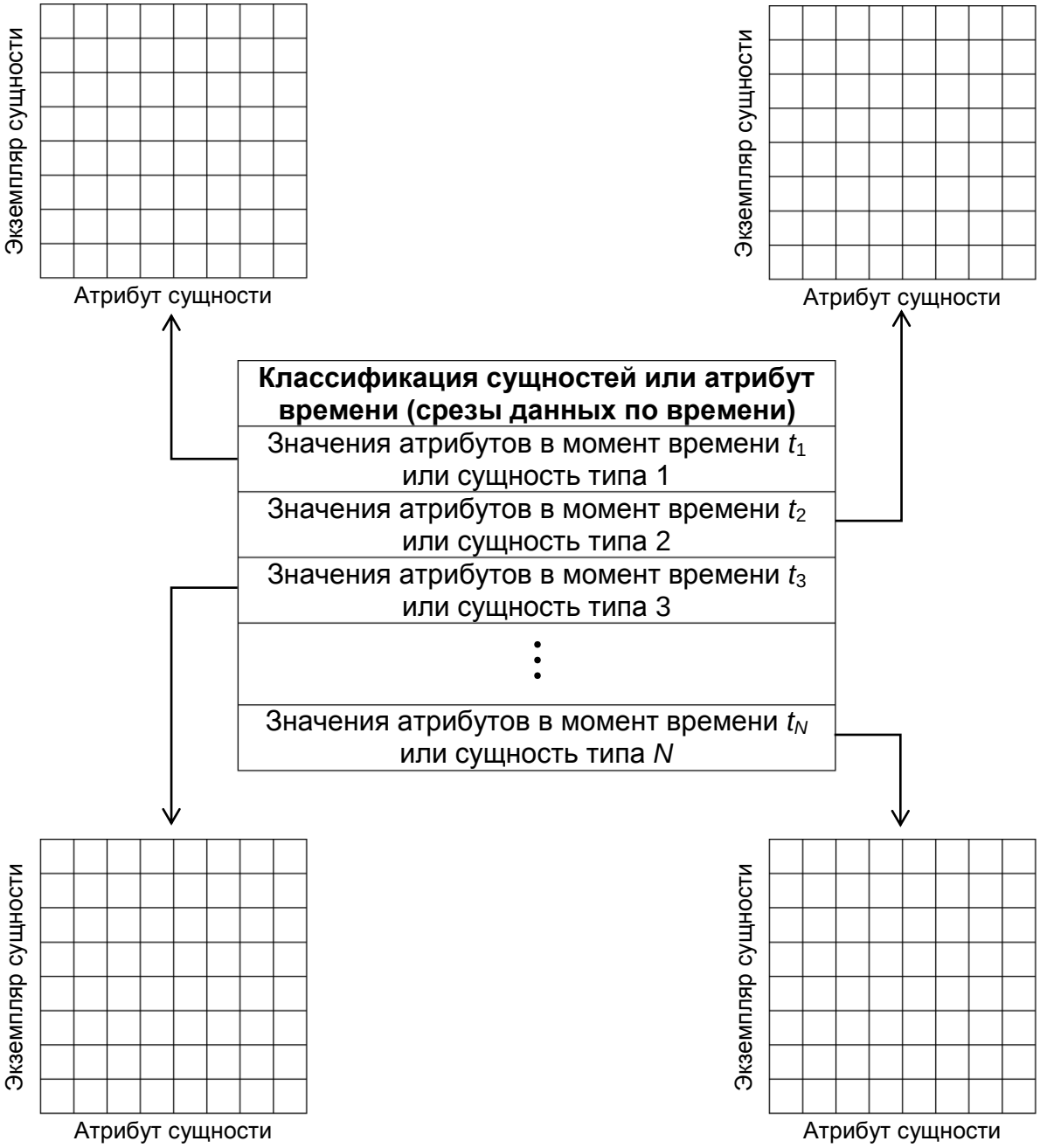


Рис. 23. Реализация трёхмерного куба с помощью сетевой модели и реляционных связей

Проектирование БД

Прежде всего, отметим наиболее важные цели проектирования БД [4].

1. Возможность хранения всех необходимых данных в БД.

Эта цель является очевидной и ещё раз подчёркивает необходимость построения концептуальной схемы БД, включающей все сущности или объекты ПрО, их атрибуты и определения отношений между ними.

2. Исключение избыточности данных.

Для понимания этой цели определим различие между дублированием данных и избыточным дублированием или избыточностью. Для этого рассмотрим отношение *R* из предыдущего раздела.

R

A	B	C
x	1	a
y	2	a
z	3	a
~~w~~	4	~~b~~
~~w~~	5	~~b~~
w	6	b

В этом отношении значения ‘a' и ‘b' атрибута ‘C' повторяются (дублируются). Однако если удалить значение ‘a', например, из первого кортежа, то восстановить этот кортеж мы уже не сможем. Если же удалить значение ‘b' из одного или даже из всех, кроме одного, кортежей, то его можно восстановить по значению ‘w' атрибута A, и наоборот. В первом случае мы имеем необходимое дублирование, во втором — избыточное.

Итак, если существует хотя бы одна проекция на *неединичное* подмножество схемы отношения, которое приводит к *уменьшению мощности* этого отношения, то такое отношение содержит *избыточные данные*.

Исключить избыточность можно, например, путём разделения исходного отношения на несколько (2 или больше):

A	B
x	1
y	2
z	3
w	4
w	5
w	6

A	C
x	a
y	a
z	a
w	b

В том случае, если указанная в определении проекция уменьшает мощность отношения не менее чем в два раза, то разбивка позволяет *сократить объем памяти*, необходимый для хранения информации.

Однако более важным является то, что устранение избыточности позволяет в значительной мере упростить процедуры добавления и удаления данных.

На решение этой проблемы ориентирована и третья цель проектирования:

3. Нормализация отношений.

Под *нормализацией* понимается разбивка (в общем случае, преобразование) отношения на два или более для приведения его в соответствие к требованиям так называемых **нормальных форм**.

Рассмотрим отношение *R*, представив его следующим образом:

R’

AB		C
A	B	a
x	1	
y	2	
z	3	
A	B	в
w	4	
w	5	
w	6	

С математической точки зрения отношения *R’* полностью корректно: оно состоит из двух кортежей, атрибут *AB* которых имеет значения, в свою очередь, равные некоторым отношениям. Значения же атрибута *C атомарны*, то есть неделимы.

Отношение, *все* значения которого атомарны, называется *нормализованным* или представленными в *первой нормальной форме*: 1НФ. В данном случае эти два понятия являются синонимами.

Итак, *R* и *R’* — это, по сути, одинаковые отношения, только *R* — нормализованное, а *R’* — ненормализованное.

Однако реляционная модель допускает только атомарные значения атрибутов. То есть, никакой домен не может содержать отношения. Поэтому БД, представленная в этой модели, всегда должна быть нормализована.

Кроме того, необходимо учитывать, что неограниченный рост числа отношений может привести к такому увеличению количества и сложности связей между отношениями, которое потребует значительного усложнения запросов к СБД, что, в свою очередь, снизит её эффективность. Чтобы этого избежать, необходимо найти компромисс в достижении третьей и четвертой целей:

4. Сведение количества хранимых в БД отношений к минимуму.

Нормализация отношений

Итак, для устранения избыточности данных, снижения вероятности возникновения проблем или *аномалий* при вставке, обновлении или удалении информации выполняется приведение отношения к нормальной форме, обеспечивающей решение этих задач.

Каждой нормальной форме соответствует некоторый определённый набор ограничений. Следовательно, отношение находится в некоторой нормальной форме, если удовлетворяет присущий ей набор ограничений.

Высшие нормальные формы полностью содержат в себе низшие. То есть, любая база данных *третьей* нормальной формы в то же время является базой данных *второй* и *первой* нормальных форм, но не наоборот.

Для достижения каждого уровня нормализации, улучшающей конструкцию базы данных, применяются определённые наборы правил. Причём, реализуется это путём разбивки отношения, находящегося в предыдущей нормальной форме, на два или более, удовлетворяющих следующую нормальную форму. Эта процедура носит название

декомпозиции, в основе которой лежит концепция *функциональных зависимостей* (ФЗ) между атрибутами отношения.

Пусть дано отношение *R*, а *X* и *Y* — правильные подмножества его схемы отношения. Говорят, что *Y функционально зависит от X* ($X \to Y$ [*читается: „X функционально определяет Y“ или „X стрелка Y“*]) тогда и только тогда, когда для каждого допустимого значения множества *X* существует в точности одно значение множества *Y*.

Иначе говоря, если два кортежа отношения R совпадают по значению *X*, то они совпадают по значению *Y*. Левую и праву части ФЗ называют **детерминантом** и **зависимой частью** соответственно.

Если выполняется обратное соотношение, то множества атрибутов являются **взаимозависимыми:** $X \leftrightarrow Y$. Пример:

R: $A \to C, B \to C, \{A, B\} \to C, B \to A, \{B, C\} \to A$

S: $D \leftrightarrow E$

Функциональная зависимость $X \to Y$ называется **полной**, если атрибут *Y* функционально зависит от множества *X* и не зависит от какого-либо его подмножества.

Функциональная зависимость $X \to Z$ называется **транзитивной**, если существует такой атрибут *Y*, что существуют функциональные зависимости $X \to Y$ и $Y \to Z$, и отсутствует функциональная зависимость $Z \to X$. (При отсутствии последнего требования мы имели бы „неинтересные“ транзитивные зависимости в каждом отношении, которое владеет несколькими ключами).

На сегодняшний день разработано 6 нормальных форм [4] — *1НФ - 6НФ*, которые накладывают на отношение возрастающие по жёсткости ограничения. В теории реляционных баз данных обычно говорится, что эти формы формируют такую последовательность:

— нормализация данных = первая нормальная форма (1НФ);
— вторая нормальная форма (2НФ);
— третья нормальная форма (3НФ);
— нормальная форма Бойса-Кодда (НФБК) как модификация предыдущей;
— четвертая нормальная форма (4НФ);
— пятая нормальная форма, или проекционно-соединительная нормальная форма (5НФ или ПСНФ);
— шестая нормальная форма, или доменно-ключевая нормальная форма (6НФ или ДКНФ).

Основные свойства нормальных форм:

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

Поскольку требование **первой нормальной формы** является базовым требованием классической реляционной модели данных, мы будем считать, что исходный набор отношений уже отвечает этому требованию (см. стр. 34).

Вторая нормальная форма

Отношение R находится во *второй нормальной форме* (2НФ) в том и только в том случае, когда находится в 1НФ, и каждый неключевой атрибут *полно зависит* от первичного ключа отношения R [4].

В этом определении предполагается, что в отношении существует единственный потенциальный ключ, который, соответственно, является первичным.

Если допустить наличие нескольких потенциальных ключей, то определение приобретёт следующий вид:

Отношение R находится во *второй нормальной форме* (2НФ) в том и только в том случае, когда оно находится в 1НФ, и каждый неключевой атрибут *полно зависит* от **каждого** потенциального ключа отношения R.

Рассмотрим пример отношения:

Научно-исследовательский центр				
№ сотрудника	**Название лаборатории**	**Название проекта**	**Местоположение (№ корпуса)**	**Научный руководитель лаборатории**
A	**B**	**C**	**D**	**E**
1	a	1	8	W
2	a	1	8	W
3	a	4	8	W
1	b	2	15	S
2	b	5	15	S
3	b	2	15	S
1	c	3	8	W
2	c	3	8	W

Первичный ключ этого отношения: $\{A, B\}$.

Некоторые из функциональных зависимостей: $\{A, B\} \to C; B \to E; B \to D; D \to E$.

Видно, что хотя первичным ключом является составной ключ $\{A, B\}$ (или *{№ сотрудника, Название лаборатории}*), атрибуты *E* (или *Научный Руководитель Лаборатории*) и *D* (или *Местоположение (№ корпуса)*) функционально зависят от части первичного ключа, атрибута *B*. В результате мы не сможем вставить в отношение **Научно-исследовательский центр** кортеж, описывающий какую-нибудь новую лабораторию, если не определены проекты, которые она будет выполнять, и сотрудники, привлечённые к их выполнению (первичный ключ не может содержать неопределённое значение). При удалении кортежа мы не только разрушаем связь определённого сотрудника с данным проектом, но теряем информацию о том, что он работает в определённом отделе. При переводе сотрудника в другой отдел мы будем вынуждены модифицировать все кортежи, описывающие этого сотрудника, или получим несогласованный результат. Такие досадные явления называются аномалиями схемы отношения или *коллизиями*. Именно они устраняются путём нормализации.

Можно выполнить следующую декомпозицию отношения **Научно-исследовательский центр** на два отношения — **Проекты** и **Штат**.

Проекты(*Название лаборатории, Научный Руководитель Лаборатории, Местоположение* (*№ корпуса*)).

Проекты		
Название лаборатории	**Местоположение (№ корпуса)**	**Научный руководитель лаборатории**
B	**D**	**E**
a	8	W
b	15	S
c	8	W

Функциональные зависимости этого отношения: $B \to E; B \to D; D \to E$.

Штат(*№ сотрудника*, *Название лаборатории*, *Название проекта*).

Штат		
№ сотрудника	**Название лаборатории**	**Название проекта**
A	**B**	**C**
1	a	1
2	a	1
3	a	4
1	b	2
2	b	5
3	b	2
1	c	3
2	c	3

С функциональными зависимостями: $\{A, B\} \to C$.

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

Третья нормальная форма

Отношение *R* находится в *третьей нормальной форме* (3НФ) в том и только в том случае, если находится в 2НФ, и *не имеет* неключевых атрибутов, *транзитивно зависимых* от какого-нибудь потенциального ключа отношения *R* [4].

Рассмотрим ещё раз отношения **Проекты**, которое находится в 2НФ. Заметим, что функциональная зависимость $B \to E$ есть транзитивной; она является следствием функциональных зависимостей $B \to D$ и $D \to E$. Другими словами, *Местоположение (№ корпуса)* на самом деле является характеристикой не лаборатории (*Названия лаборатории*), а *Научного Руководителя Лаборатории*, который там работает.

В результате мы не сможем занести в базу данных информацию, например о том, что научный руководитель *F* находится в корпусе № 2, до тех пор, пока в этом корпусе не появится лаборатория (первичный ключ не может содержать неопределённое значение). При удалении кортежа, который описывает научного руководителя лаборатории, мы лишимся информации о её местоположении. То есть в отношении **Проекты,** как и раньше, существуют аномалии. Их можно устранить путём дальнейшей нормализации.

Можно сделать декомпозицию отношения **Проекты** на два отношения — **Руководители** и **Лаборатории**:

Руководители(*Научный руководитель лаборатории*, *Местоположение* (*№ корпуса*))

Руководители	
Научный руководитель лаборатории	**Местоположение (№ корпуса)**
E	**D**
W	8
S	15

Функциональные зависимости: $D \to E$.

Лаборатории(*Название лаборатории*, *Местоположение* (*№ корпуса*))

Лаборатории	
Название лаборатории	**Местоположение (№ корпуса)**
B	**D**
a	8
b	15
c	8

Функциональные зависимости: $B \to D$.

Каждое с этих двух отношений находится в 3НФ и свободно от отмеченных коллизий.

Для подавляющего большинства систем довольно привести отношение до *3НФ* или её модификации — *НФ Бойса-Кодда* (НФБК), что обеспечивает *декомпозицию без потерь*.

Поэтому остановимся детальнее на этой форме.

Нормальная форма Бойса-Кодда

Под **разложением** или **декомпозицией без потерь** понимается такой способ декомпозиции отношения, при котором исходное отношение полностью и без избыточности *восстанавливается* путём *естественного соединения* полученных отношений.

Отношение находится в *НФБК* тогда и только тогда, когда *каждый* детерминант является потенциальным ключом.

Алгоритм декомпозиции без потерь, приводящий отношение к НФБК, следующий:

1. Определяется ФЗ, которая является причиной того, что отношение не находится в НФБК.
2. Выполняется проекция отношения на детерминант и зависимую часть найденной ФЗ.
3. Из схемы исходного отношения удаляются атрибуты, входящие в зависимую часть ФЗ.
4. Отношения, полученные в п. 2 и 3, и будут результатом декомпозиции. Эти отношения проверяются на соответствие НФБК, и, при необходимости, алгоритм повторяется.

Пример. Рассмотрим отношение *R*, ФЗ которого мы уже определили выше. Выпишем детерминанты и потенциальные ключи и определим, по каким элементами эти множества не совпадают. При этом для наглядности примера мы не будем обращать внимание на свойство неизбыточности потенциального ключа.

Детерминанты: $A, B, \{A, B\}, \{B, C\}$

Потенциальные ключи: $B, \{A, B\}, \{B, C\}$

Видно, что атрибут *A* является детерминантом, но не является потенциальным ключом. Ему соответствует ФЗ $A \to C$.

Выполним проекцию отношения R на атрибуты A и C: $R'' = R[A, C]$. Из отношения R удаляем атрибут C: $R' = R[A, B]$.

R’

A	B
x	1
y	2
z	3
w	4
w	5
w	6

R"

A	C
x	a
y	a
z	a
w	b

Для проверки того факта, что сделана декомпозиция без потерь, выполняется естественное соединение результирующих отношений, которое должно дать исходное.

В том случае, если ФЗ, соответствующих первому пункту алгоритма, несколько, то возникает проблема выбора ФЗ, зависимая часть которой или её подмножество является детерминантом другой ФЗ.

Одной из методик решения проблемы является поиск „*цепочек ФЗ*“ вида $A \to B \to C$ с последующим использованием для декомпозиции крайней правой ФЗ.

При использовании этой методики ФЗ удобно представить на *диаграмме ФЗ* (рис. 24).

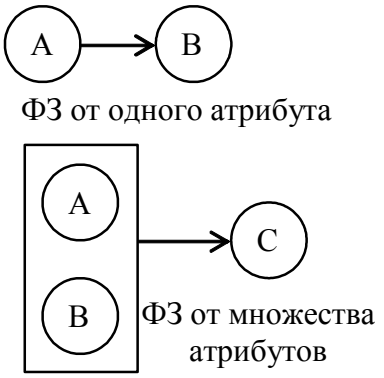


Рис. 24. Примеры диаграмм ФЗ

Покажем диаграмму ФЗ отношения *R* (рис. 25):

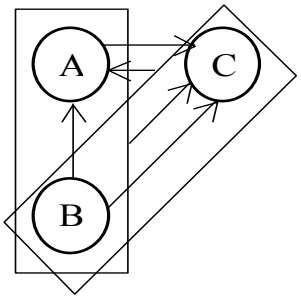


Рис. 25. Диаграмма ФЗ отношение R1

Выделить на этой диаграмме цепочку ФЗ довольно сложно. Однако эту диаграмму можно упростить, удалив из всего множества ФЗ *избыточные ФЗ*. То есть зависимости, не несущие информации, которая *не могла бы быть получена* из других ФЗ отношения.

Удалить избыточные ФЗ можно, используя два *правила вывода*. Пусть *W, X, Y, Z* — правильные подмножества схемы отношения R, а обозначение *XY* соответствует $X \bigcup Y$.

1. **Транзитивность**. Если $X \to Y$ и $Y \to Z$, то $X \to Z$, которая называется *транзитивной* зависимостью. Причём, **если все три зависимости входят в набор ФЗ отношения, то транзитивная зависимость является избыточной.**
2. Дополнение:
   а) если $X \to Y$, то $XZ \to Y$;
   б) если $X \to Y$, то $XZ \to YZ$.

   Причём, **дополненные зависимости в обоих случаях являются избыточными.**

Вернёмся к примеру. Согласно правилу *дополнения* исключаем из набора ФЗ отношения R зависимости $\{A,B\} \to C$ и $\{B,C\} \to A$. Получим новую диаграмму, на которой чётко видная *транзитивная* зависимость $B \to C$. Её мы также исключаем из рассмотрения (рис. 26).

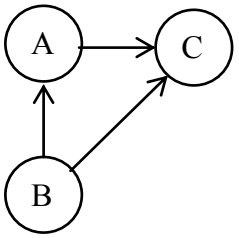


Рис. 26. Диаграмма ФЗ транзитивной зависимости

В результате получаем цепочку ФЗ $B \to A \to C$, из которой берём крайнюю правую зависимость для декомпозиции.

Приведенные правила являются двумя из трёх *правил* вывода или *аксиом Армстронга*.

3. Третье правило — **рефлексивность**: для любых множеств атрибутов X и Y: $XY \to X$ и $XY \to Y$.

Эти аксиомы и несколько следующих правил вывода позволяют упростить и уменьшить исходный набор ФЗ.

4. **Декомпозиция**. Если $X \to YZ$, то $X \to Y$ и $X \to Z$.
5. **Объединение**. Если $X \to Y$ и $X \to Z$, то $X \to YZ$.
6. **Композиция**. Если $X \to Y$ и $Z \to W$, то $XZ \to YW$.

Нарушения требований НФБК происходят крайне редко, поскольку это может случиться только тогда, когда:

— существуют два (или более) составных потенциальных ключа;
— эти потенциальные ключи перекрываются, то есть ими вместе используется, по крайней мере, один общий атрибут.

Хотя для большинства систем достаточно отношений в НФБК, иногда полезно продолжить процесс нормализации.

Четвертая нормальная форма

Рассмотрим пример. Пусть в научно-исследовательском центре из сотрудников лабораторий создали несколько рабочих групп, каждая из которых работает над одним или несколькими проектами и при этом использует ресурсы разных лабораторий.

Рабочие группы		
Рабочая группа	**№ проекта**	**Название лаборатории**
Y	**X**	**Z**
1	1	a
1	2	a
2	3	a
1	4	b
3	4	b
4	5	b
1	1	c
4	5	c

Единственным потенциальным ключом такого отношения является составной атрибут $\{X, Y, Z\}$, и нет никаких других детерминантов. Следовательно, отношения **Рабочие группы** находится в НФБК. Но при этом оно имеет недостатки: если, например, создаётся новая рабочая группа и присоединяется к каким-нибудь проектам, то необходимо вставить в отношение **Рабочие группы** столько кортежей, к скольким проектам она подключится, и ресурсы скольких лабораторий она будет использовать.

Для нормализации такого отношения в 1971 г. Р. Фейджином было введено понятие *многозначной зависимости*.

В отношении $R(X, Y, Z)$ существует **многозначная зависимость** $X \twoheadrightarrow Y$ в том и только в том случае, если множество значений *Y*, которое соответствует паре значений *X* и *Z*, зависит только от *X* и не зависит от *Z*.

Легко показать, что в общем случае в отношении $R(X, Y, Z)$ существует многозначная зависимость $X \twoheadrightarrow Y$ в том и только в том случае, когда существует многозначная зависимость $X \twoheadrightarrow Z$.

Таким образом, многозначные зависимости всегда создают связанные пары, и потому их по обыкновению представляют вместе в символическом виде: $X \twoheadrightarrow Y \mid Z$.

Теорема Фейджина. Отношение $R(X,Y,Z)$ будет равняться соединению его проекций $R[X,Y]$ и $R[X,Z]$ тогда и только тогда, когда для отношения R выполняется многозначная зависимость $X \twoheadrightarrow Y \mid Z$.

На основе изложений Фейджина была определена 4НФ.

Отношение *R* находится в *четвертой нормальной форме* (4НФ) тогда и только тогда, когда в случае существования *многозначной зависимости* $X \twoheadrightarrow Y$ все другие атрибуты отношения *R функционально* зависят от *X* [4].

Теперь можно выполнить декомпозицию отношения **Рабочие группы** на два, которые отвечают требованиям 4НФ и свободны от отмеченных аномалий.

Проекты	
Рабочая группа	**№ проекта**
Y	**X**
1	1
1	2
1	4
2	3
3	4
4	5

Ресурсы	
№ проекта	**Название лаборатории**
X	**Z**
1	a
1	c
2	a
3	a
4	b
5	b
5	c

Пятая нормальная форма

Во всех рассмотренных до этого момента случаях нормализация выполнялась декомпозицией одного отношения на два. Иногда это сделать не удаётся, однако же, иногда возможна декомпозиция на большее число отношений, каждое из которых имеет лучшие свойства.

Вернёмся к примеру из раздела о 4НФ — отношению **Рабочие группы**.

Первичным ключом этого отношения является полная совокупность его атрибутов, отсутствуют функциональные зависимости и, исходя из теоремы Фейджина, оно может быть декомпозовано без потерь на две его проекции, определяемыми многозначными зависимостями.

Однако в них могут существовать аномалии, когда в ПрО, например, какие-либо рабочие группы имеют или, наоборот, не имеют допуска к работе в той или иной лаборатории. Такие аномалии можно устранить путём декомпозиции отношения R на *три отношения* на основании понятия *зависимости соединения*.

Пусть *X, Y*, …, Z – некоторые подмножества *M* атрибутов отношения *R*, причём $X \bigcap Y \bigcap \ldots \bigcap Z = M$. Отношение *R* удовлетворяет **зависимости соединения** $*(X,Y,\ldots,Z)$

в том и только в том случае, когда R восстанавливается без потерь путём соединения своих проекций на X, Y, …, Z.

Зависимость соединения $*(X,Y,\ldots,Z)$ называется *тривиальной*, если хотя бы одно из подмножеств совпадает с M.

Соответственно, теорема Фейджина может быть сформулирована следующим образом.

Отношение $R(X,Y,Z)$ удовлетворяет *зависимости соединения* $*(XY,XZ)$ тогда и только тогда, когда оно удовлетворяет многозначной зависимости $X \twoheadrightarrow Y \mid Z$.

Соответственно, можно сделать вывод, что зависимость соединения является *обобщением* многозначной зависимости и позволяет привести отношение к 5НФ.

Отношение R находится в *пятой нормальной форме* (5НФ или проекционно-соединительной нормальной форме ПСНФ) в том и только в том случае, когда *каждая нетривиальная* зависимость соединения в R следует из *существования* некоторого *потенциального ключа* в R [4].

Для приведения к 5НФ отношение из примера разбивается на следующие:

Проекты(Рабочая группа, № проекта);

Группы(Рабочая группа, Название лаборатории);

Ресурсы(№ проекта, Название лаборатории).

Шестая нормальная форма

Шестая нормальная форма (6NF) — введена К. Дейтом [4] как обобщение пятой нормальной формы для хронологической базы данных.

Для такой БД максимально возможная декомпозиция позволяет бороться с избыточностью и упрощает поддержку целостности базы данных с точки зрения сохранения хронологии.

Отношение R находится в *шестой нормальной форме* (6НФ или доменно-ключевой нормальной форме ДКНФ) тогда и только тогда, когда оно находится в 5НФ и удовлетворяет *всем* нетривиальным зависимостям соединения.

Из определения следует, что отношение находится в 6НФ тогда и только тогда, когда оно неприводимо, то есть не может быть подвергнуто дальнейшей декомпозиции без потерь.

Рассмотрим упрощённый пример отношения **Научно-исследовательский центр**.

Научно-исследовательский центр			
Рабочая группа	**Название проекта**	**Научный руководитель проекта**	**Дата приказа**
A	**C**	**E**	**D**
1	2	S	10.10
1	2	W	24.10
1	5	W	23.11

Это отношение не находится в 6НФ и может быть подвергнуто декомпозиции на отношения **Проекты** и **Руководство**.

Проекты		
Рабочая группа	**Название проекта**	**Дата приказа**
A	**C**	**D**
1	2	10.10

Руководство		
Рабочая группа	**Научный руководитель проекта**	**Дата приказа**
A	**E**	**D**
1	S	10.10

1	5	23.11	1	W	24.10

Пятая и шестая нормальные формы — это последние нормальные формы, которые можно получить путём декомпозиции. Их условия весьма нетривиальны, и на практике 5НФ и ДКНФ практически не используются.

Замечание.

! В конце этого раздела необходимо заметить, что нормализация упрощает процессы добавления, обновления и удаления информации, позволяет избежать избыточных данных и уменьшить объем памяти, требуемой для хранения отношений. Однако благодаря увеличению количества таблиц и связей между ними любая нормализация усложняет процедуры поиска информации. Когда объем данных достигает нескольких гигабайт, это становится препятствием эффективного функционирования информационной системы. В этом случае разработчики соглашаются на так называемую процедуру ***денормализации*** тех отношений, по данным которых чаще всего выполняется поиск. Но каких-то обобщённых или стандартизированных процедур и алгоритмов денормализации отношений на сегодня не существует.

Формализация связей между отношениями

В предыдущем разделе рассмотрено проектирование СБД на уровне отдельных сущностей (отдельных отношений). Однако СБД, чаще всего, состоит с множества сущностей, между которыми существуют определённые связи. Поэтому естественным является вопрос, каким образом формализовать (определить) эти связи в СБД. Здесь также существует ряд правил, и для этого необходимо вспомнить, какие типы связей существуют. Но до этого определим ещё ряд понятий.

1. Иногда возникают ситуации, когда в **каком-нибудь кортеже отношения не существует значения** того или другого атрибута. Не „*значение не известно*“ пользователю, а именно „*значение не существует*“ (на данный момент или вообще) или „*не может существовать*“. Например: сущность *Принтер*:

Тип	Фирма	Тип чернил	Ширина ленты, мм
MP5350AI	Seikosha	—	24
Stylus Color 600	Epson	Quick Dry	—

При возникновении такой ситуации большинство реальных СУБД допускают использование так называемых ***NULL*- или *NIL*-значений.**

2. Пусть дано отношение *R*. Тогда **внешний ключ** *K'* в отношении *R'* — это подмножество схемы отношения *R'*, такое что:
   а) существует отношение *R"* с *потенциальным ключом K"*;
   б) для любого кортежа отношения *R'* значение *K'* является *NULL*-значением или *совпадает* со значением *K"* некоторого кортежа отношения *R"*. Отношение *R"* называется **родительским** или *главным*, *R'* — **детализированным** или *подчинённым*, а ключ *K"* — *родительским*.

В примере декомпозиции отношения *R* атрибут *A* в отношении *R"* является потенциальным и родительским ключом, а в отношении *R'* — внешним.

Замечание. В большинстве случаев при проектировании СБД *K"* является *первичным ключом* отношения *R"*.

Рекомендация. Необходимо, по возможности, избегать использования *NULL*-значений.

Например, путём разделения сущности на супертип и подтипы.

Теперь рассмотрим правила формализации некоторых типов связи [4]. Все отношения представлены в НФБК. Это *необязательное требование*, потому что:

1) проектирование с помощью ФЗ может быть выполнено и на заключительных этапах разработки СБД;
2) схема отношения может быть настолько большой (на практике более 20 атрибутов), что это существенным образом усложняет определение ФЗ и проектирование с их помощью.

Вторая ситуация изначально требует представления сущности в виде нескольких отношений.

1. *Связь 1:1*. Это единственный тип связи, который допускает **два варианта представления**. Если между двумя сущностями существует безусловная связь 1:1, то для их представления достаточно *одного отношения*. Причём **первичным ключом этого отношения может быть идентификатор любой из сущностей**. Если же по условиям задачи или системы необходимо каждую сущность представить в виде отдельного отношения, то **для формализации связей необходимо атрибуты первичного ключа одного из отношений добавить в схему другого отношения в качестве внешнего ключа.**

На *ER*-диаграмме это отразится просто добавлением множества атрибутов к списку атрибутов второй сущности.

На ИМ, и в этом её большая наглядность, внешний ключ будет представлен вспомогательными атрибутами *с обязательным указанием метки связи*. Пример *R1* (рис. 5,рис. 27):

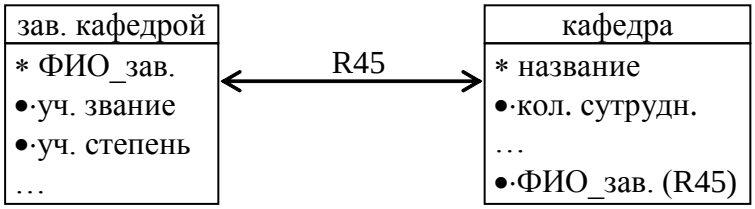


Рис. 27. Формализация связи 1:1

На ИМ *MS Access*, вместо метки связи используется её графическая интерпретация: сама линия связи соединяет соответствующие атрибуты внешнего и родительского ключей. Пример *R1* в нотации *MS Access* (рис. 5,рис. 28):

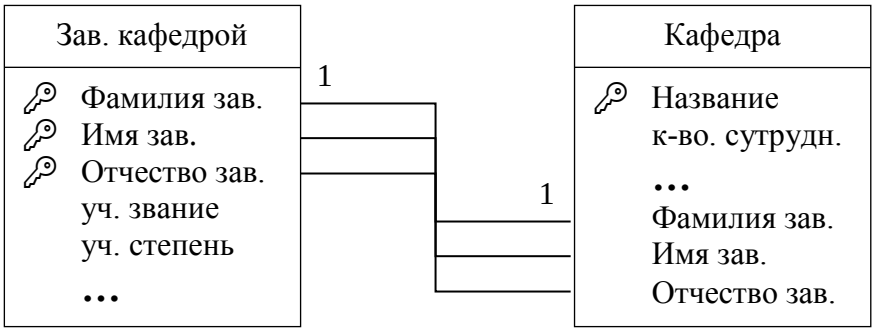


Рис. 28. Формализация связи 1:1 в нотации MS Access

2. *Связь 1:1у*. Если между сущностями установленная связь 1:1у, то
   а) каждая сущность представляется **отдельным отношением**. Причём **ID сущности становится первичным ключом соответствующего отношения.**
   б) ID сущности, соответствующей условности связи, добавляется как *вспомогательный*

атрибут (на ИМ ещё и с меткой связи) в другую сущность и, соответственно, в качестве внешнего ключа в схему её отношения.

Пример R4 (рис. 12,рис. 29):

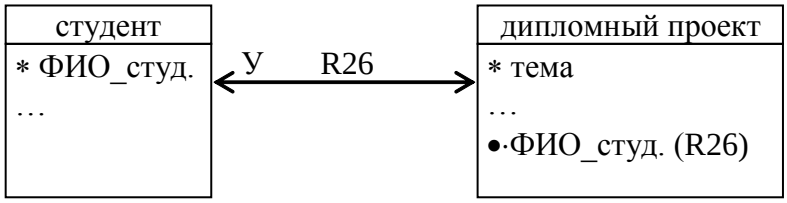


Рис. 29. Формализация связи 1y:1

Кстати, вспомогательный атрибут в этом случае будет одним из потенциальных ключей этого же отношения и может быть использован для поиска информации.

3. *Связь 1y:1y*. **Для формализации** связи этого типа **необходимо** использовать **3 отношения: по одному для каждой сущности и одно для связи**. На ИМ последнее отношение представляется *ассоциативным объектом (сущностью)*. Атрибутами этого отношения обязательно будут первичные ключи двух других. Пример *R8* (рис. 13,рис. 30):

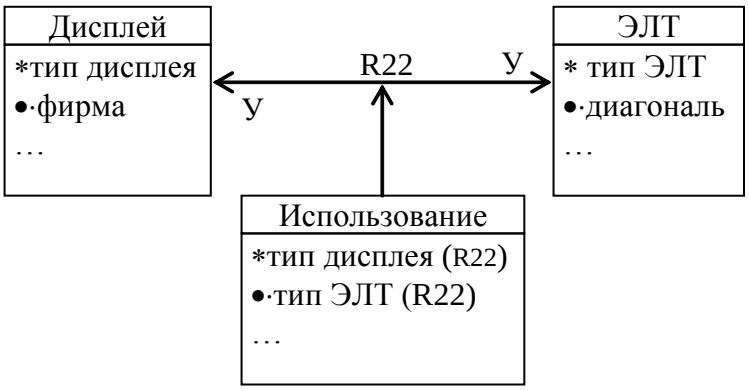


Рис. 30. Формализация связи 1y:1y

При этом типе связи оба множества атрибутов ассоциативного объекта являются потенциальными ключами. Причём, если один из них выбирается как первичный ключ этого отношения, то второй автоматически считается внешним. Кроме того, ассоциативный объект может иметь и множество собственных атрибутов.

4. *Связи 1:N и 1y:N*. Для формализации связей типа 1:*N не имеет значения,* является ли связь со стороны *односвязной* сущности условной или безусловной.

Каждая сущность, принимающая участие в связи типа 1:*N* или 1y:*N*, **представляется своим отношением**. При этом **первичный ключ отношения, соответствующего односвязной сущности, добавляется в качестве внешнего ключа в схему отношения *N*-связной сущности.**

Пример *R2* (рис. 8,рис. 31).

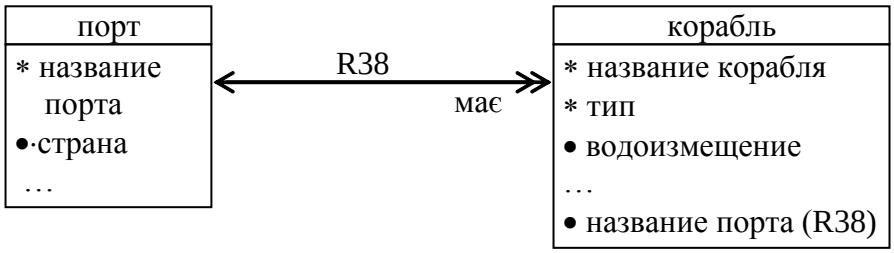


Рис. 31. Формализация связи 1:N, 1y:N

Ясно, что внешний ключ отношения *N*-связной сущности не может быть потенциальным ключом этой сущности.

5. *Связи 1:Ny и 1y:Ny* (рис. 14). Для задания связи 1:*N*, *условной со стороны N-связной сущности*, **необходимо** сформировать **по одному отношению для каждой сущности и одно отношение для связи, в схему отношения которого должны быть включены первичные ключи отношений сущностей.**

   Пример *R9* (рис. 32).

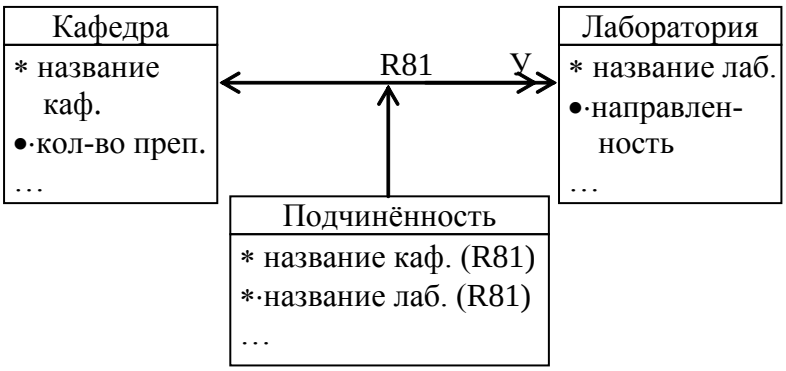


Рис. 32. Формализация связи 1:Ny и 1y:Ny

Потенциальными ключами отношения, соответствующего ассоциативной сущности, может быть ключ отношения *N*-связной сущности или множество, состоящее из ключей обоих отношений.

6. *Связь M:N любой условности* (БУ, У, 2У) также **требует трёх отношений для формализации: по одному для каждой сущности и одно для связи. Причём, последнее должно иметь в схеме отношения первичные ключи двух других.** Более того, если ассоциативный объект не имеет или не должен иметь собственного атрибута-идентификатора, то первичным ключом может быть только множество, содержащее оба внешних ключа. Если при этом отсутствуют и какие-либо другие собственные атрибуты этого отношения, то говорят, что оно *полностью является ключом*.

   Пример R3 (рис. 10,рис. 33).

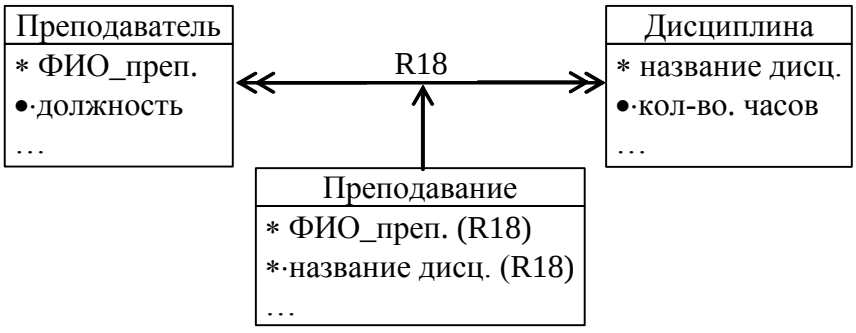


Рис. 33. Формализация связи M:N

7. При формализации *связей „супертип-подтип“* **отношение подтипов должны включать в качестве первичных ключей первичный ключ отношения супертипа**, как показано на *рассмотренном ранее примере* (рис. 15,рис. 16). Кроме того, между подтипами, в свою очередь, могут существовать связи, которые формализуются по уже рассмотренным правилам.

Иногда, для ускорения поиска информации, в отношение супертипа добавляется атрибут, значения которого говорят о том, какому подтипу соответствует этот экземпляр супертипа (что-то вроде метки).

8. Иногда для описания предметной области *недостаточно бинарных связей* между сущностями.

Для примера рассмотрим три сущности: *Преподаватель, Дисциплина, Специальность — Преподаватель* читает конкретную *Дисциплину* конкретной *Специальности*:

<table>
<tr><th>Преподаватель</th><th>Чтение</th></tr>
<tr><td>Малахов (1)</td><td>
<table>
<tr><th>Дисциплина</th><th>Специальность</th></tr>
<tr><td rowspan="2">Вычислительная техника (1)</td><td>АП (1)</td></tr>
<tr><td>ОС (2)</td></tr>
<tr><td>Ассемблер (2)</td><td>ОС (2)</td></tr>
</table>
</td></tr>
<tr><td>Юхименко (2)</td><td>
<table>
<tr><th>Дисциплина</th><th>Специальность</th></tr>
<tr><td>Вычислительная техника (1)</td><td>МТ (3)</td></tr>
<tr><td rowspan="2">Экономика (3)</td><td>ОП (4)</td></tr>
<tr><td>ОС (2)</td></tr>
</table>
</td></tr>
<tr><td>Бровков (3)</td><td>
<table>
<tr><th>Дисциплина</th><th>Специальность</th></tr>
<tr><td>Электроника и микроэлектроника (4)</td><td>АМ (5)</td></tr>
<tr><td>Вычислительная техника (1)</td><td>АП (1)</td></tr>
</table>
</td></tr>
</table>

Если бы множества атрибутов этих сущностей ограничились атрибутами приведенного отношения, то было бы достаточно нормализовать это отношение, используя ФЗ. Однако каждая сущность имеет порядка 5-10 атрибутов, и такая реализация очень проблематична.

Теоретико-множественная диаграмма связи между данными сущностями выглядит следующим образом (рис. 34):

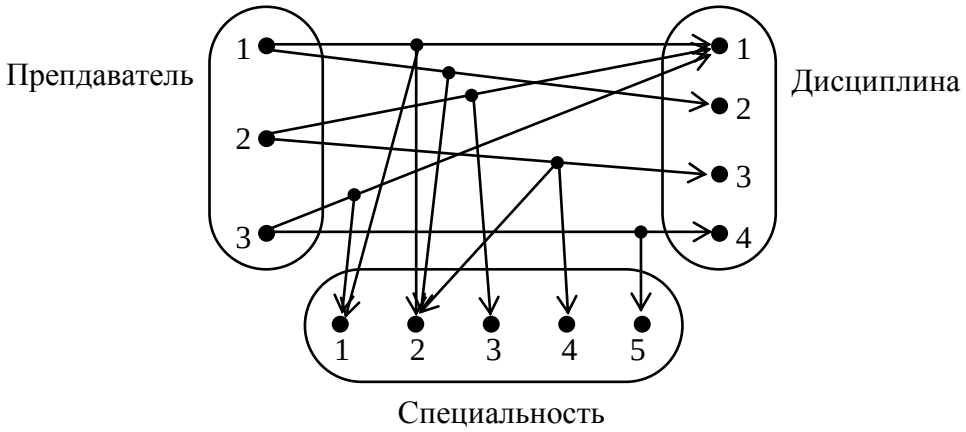


*Рис. 34. Теоретико-множественная диаграмма примера n-**арной** (тернарной) связи*

На диаграмме видно, что мы имеем дело с *тернарной* связью, причём каждая сущность является *N-связной*. Такая связь графически может быть показана изначально *только* на *ER*-диаграмме (рис. 35):

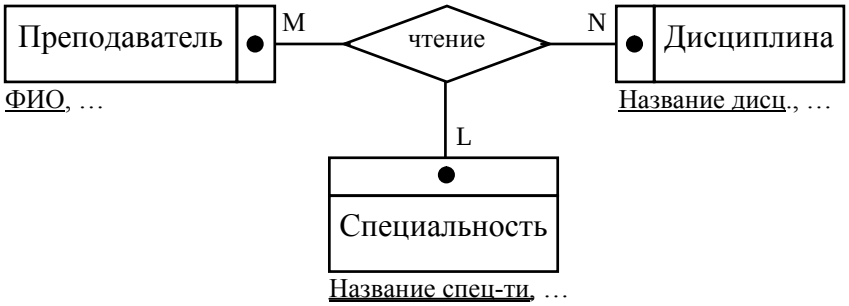


*Рис. 35. ER-диаграмма примера n-**арной** связи*

Для формализации *n*-арной связи **необходимо *n+1* отношение: по одному для каждой сущности, принимающей участие в связи, и одно для самой связи.** Причём **отношение для связи**, точнее, его схема **должна включать первичные ключи всех других *n* отношений.** Более того, множество всех этих ключей будет первичным ключом отношения связи.

Наконец следует добавить, что в нотации *MS Access* отсутствует возможность графического представления связей, формализация которых требует создания ассоциативной сущности. Поэтому при их формализации необходимо использовать подход, предложенный в сетевой модели данных (с. 19). Например, для связи *M:N* (рис. 36):

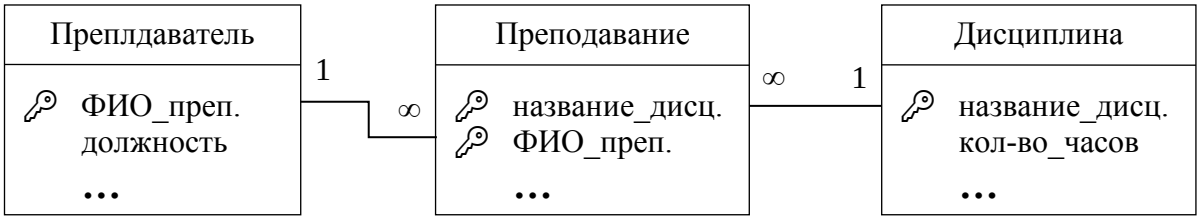


Рис. 36 Формализация связи M:N в нотации MS Access

Следует обратить внимание, что существенное *отличие* преобразования связи *M:N* в сетевой модели от приведенной формализации такой связи в нотации *MS Access* заключается *в обратной кратности* связей *1:N*.

Таким же образом, однако, *только после формализации*, может быть представлена и *n-арная* связь (рис. 37):

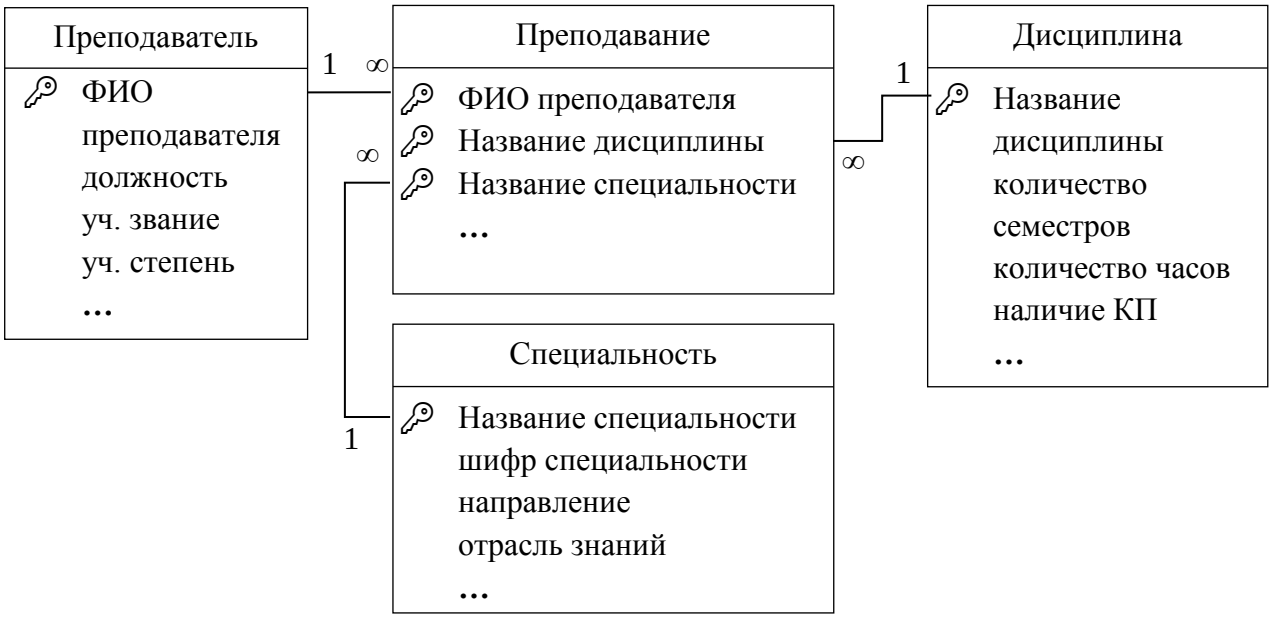


Рис. 37 Представление n-арной связи в нотации MS Access

На этом этапе мы завершаем рассмотрение теоретических аспектов проектирования БД и перейдём к вопросам практической реализации.

Демонстрационный пример

Задача.

П.1. Необходимо разработать информационную систему для регистрации авиабилетов пассажиров аэропорта, база данных которой должна содержать следующую информацию:

- информацию о всех зарегистрированных рейсах: номер рейса, пункт назначения, дата и время вылета и прибытия, количество мест, тип самолёта, авиакомпания;
- информацию о пассажирах аэропорта: фамилия, имя и отчество пассажира, его паспортные данные;
- информацию о сотрудниках (кассиров) аэропорта: фамилия, имя и отчество, контактная информация, паспортные данные, должность;
- информацию о классах салонов определённого рейса: название класса, стоимость, количество мест;
- информацию о билетах: номер билета, рейс, класс, пассажир, место, стоимость, операция (бронирование, покупка).

Эту информация первоначально расположена в следующих таблицах:

- Рейс (номер рейса, пункт назначения, дата, время вылета и прибытия, тип самолёта, авиакомпания);
- Класс (название, рейс, к которому относится класс, стоимость, количество мест);
- Пассажир (фамилия, имя, отчество, паспортные данные, пол);
- Авиакомпания (название, страна);
- Операция (покупка, бронирование);
- Сотрудник (фамилия, имя, отчество, должность, дата рождения, паспортные данные, адрес, телефон, идентификационный код);
- Билет (номер билета, пассажир, купивший данный билет, класс, операция,

сотрудник, продавший билет, дата продажи, место);

— другие таблицы по предложению студента.

П.2. Задайте связи между отношениями и формализуйте их.

П.3. Приведите отношения в соответствие нормальной форме Бойса-Кодда.

П.4. Начертите схему данных в нотации *MS Access*, информационную модель или *ER*-диаграмму.

Решение.

Определим отношения, которые будут составлять базу данных информационной системы. Каждое отношение состоит из объектов (строк), которые имеют одинаковые свойства (атрибуты). Рассмотрим, например, отношение *Рейс*. Каждый рейс, без исключения, осуществляет определённая авиакомпания, имеет свой пункт назначения, дату, время вылета и прибытия, заранее известно тип самолёта и номер рейса. Каждый пассажир должен сообщить свои паспортные данные, фамилию, имя, отчество и пол. Следовательно, данные обо всех пассажиров можно объединить в отношения с названием *Пассажир*. Аналогично рассматриваем другие объекты заданной ПрО и определяем следующие сущности: *Сотрудник*, *Билет*, *Класс*, *Авиакомпания*.

Для создания базы данных необходимо определить связи между полученными отношениями. Например, каждая авиакомпания обслуживает один или несколько рейсов, но один и тот же рейс не могут обслуживать несколько авиакомпаний. Между отношениями *Авиакомпания* и *Рейс* существует связь один-ко-многим. Каждый пассажир может покупать множество билетов на разные рейсы, но один и тот же билет может купить только один пассажир. Итак, имеем связь один-ко-многим между отношениями *Пассажир* и *Билет*.

Формализуем данные связи. В первом случае по правилу формализации необходимо первичный ключ отношения *Авиакомпания* добавить в отношение *Рейс* в качестве внешнего ключа. Это даст возможность определить, какая авиакомпания обслуживает определённый рейс. Во втором случае аналогично первичный ключ отношения *Пассажир* добавляем в отношение *Билет*.

Представим схему данных на рис. 38.

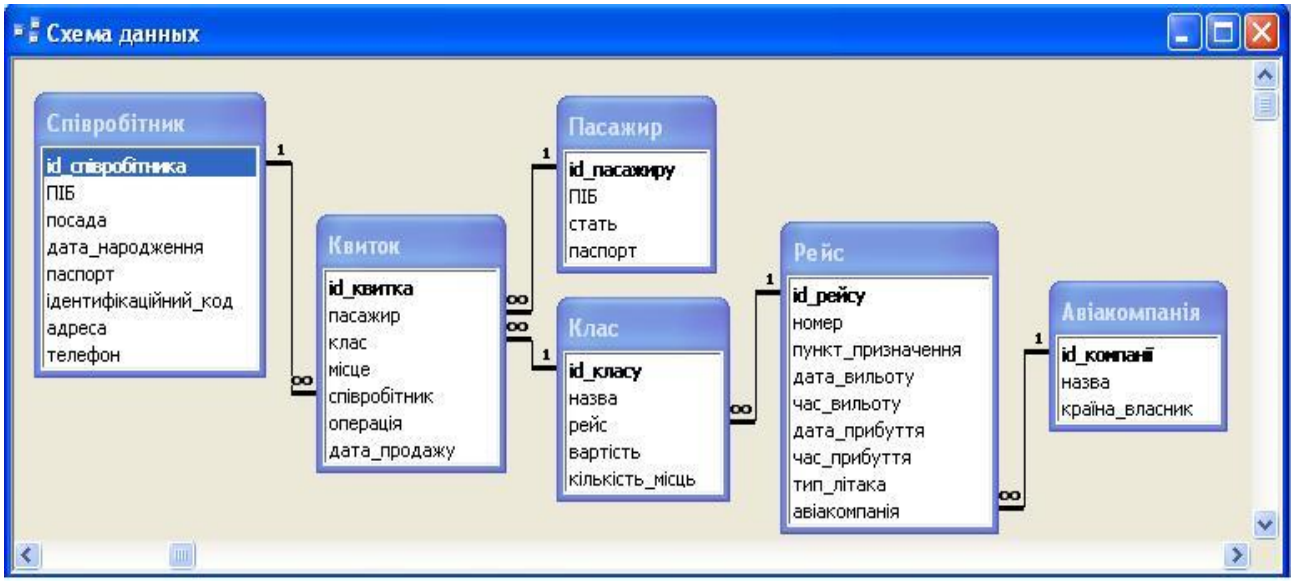


Рис. 38. Схема данных в нотации MS Access

Проверим схему данных на соответствие НФБК, согласно которой каждый атрибут отношения должен зависеть от потенциального ключа.

Рассмотрим на примере отношения *Рейс*. Покажем последовательный переход от одной нормальной формы к другой.

Данное отношение находится в первой нормальной форме, так как значения каждого атрибута не разделяются на несколько значений.

Данное отношение находится во второй нормальной форме, так как каждый неключевой атрибут функционально полно зависит от первичного ключа — *id_рейса*.

Данное отношение находится в третьей нормальной форме, так как каждый неключевой атрибут зависит только от первичного ключа *id_рейса* и не возникает информационной избыточности и аномалий.

Данное отношение находится в нормальной форме Бойса-Кодда, так как в нем отсутствуют функциональные зависимости атрибутов составного ключа от неключевых атрибутов. Это условие выполняется по умолчанию, так как в данном отношении ключ не является составным.

Аналогичным образом проверяются другие отношения.

SQL

Реализации моделей данных

Базовым требованием к реляционным СУБД является наличие мощного и в то же время простого языка, который позволяет выполнять все необходимые пользователям операции. В последние годы таким повсеместно принятым языком стал язык реляционных баз данных ***SQL*** *— Structured Query Language* (теперь все чаще название языка расшифровывается как *Standard Query Language*) [2, 3].

Программная реализация моделей данных, то есть построение СУБД или программных систем, наталкивается на большие сложности. Наиболее чётко это видно на примере реляционной модели, для которой ни единой СУБД, если придерживаться строго определений, не создано. Вместо этого существует целое семейство систем, которые в той или иной мере поддерживают язык работы с данными *SQL*. *SQL* первоначально появился как язык для реляционных СУБД, однако в меру своё практического развития он все больше и больше отклонялся от реляционности. Причиной тому являются как принципиальные проблемы реализации, так и организационно-рыночные факторы, которые не позволяют разным производителям СУБД договориться о „правильном“ стандарте.

Но при этом к преимуществам языка *SQL* все же таки относится наличие международных стандартов. Реально сейчас известно семь основных стандартов — ***SQL86***, *SQL89*, *SQL92* ***(SQL2)***, *SQL99* ***(SQL3)***, ***SQL2003***, *SQL2006*, *SQL2008*, каждый из которых имеет свою структуру и несколько уровней (реализации, соответствия).

Первый международный стандарт был принят в 1987 г., и соответствующая версия языка называется *SQL86*. Этот стандарт полностью поддерживается практически во всех современных коммерческих реляционных СУБД. Стандарт *SQL86* 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“*. Много важных аспектов языка согласно стандарту определяются в реализации.

Возможно, наиболее важными достижениями стандарта *SQL86* есть чёткая стандартизация синтаксиса и семантики операторов выборки и манипулирования данными и фиксация средств ограничения целостности БД, включающих возможности определения первичного и внешнего ключей отношений и так называемых проверочных ограничений целостности, которые позволяют сформулировать условие для каждой отдельной строки таблицы. Средства определения внешних ключей позволяют легко формулировать требования так называемой целостности БД по ссылкам. Формулировки ограничений целостности на основании понятия внешнего ключа просты и понятны.

Осознавая неполноту стандарта *SQL86*, на фоне завершения разработки этого стандарта специалисты разных фирм начали работу над стандартом *SQL*2. Эта работа также продолжалась несколько лет, было выпущено несколько проектов стандарта, пока, наконец, в марте 1992 г. не был выработан окончательный проект стандарта (после чего стандарт и соответствующий язык стали называть *SQL92*). Этот стандарт существенно более полон и охватывает практически все необходимые для реализации аспекты: манипулирование схемой БД, управление транзакциями и сессиями (сессия это последовательность транзакций, в рамках которой хранятся временные отношения), подключение к БД, динамический *SQL*. Наконец стандартизировано отношение-каталог БД, которое не связано с языком непосредственно, но очень сильно влияет на реализацию. Заметим, что в стандарте

представлены три уровня языка — базовый, промежуточный и полный. Необходимо, однако, отметить, что „чистых“ *SQL*-систем, поддерживающих какой-либо из уровней и только этот уровень какого-либо стандарта *SQL*, не существует. На практике все из существующих *SQL*-систем являются предложенными отдельными фирмами диалектами, которые, безусловно, девальвируют стандарты. В результате ввода производителями многочисленных „дополнительных возможностей“ (часто улучшающих эффективность работы с конкретной системой) перенести готовое приложение из одной СУБД, „удовлетворяющей“ по утверждению разработчика „стандарт *SQL*“, на другую СУБД, удовлетворяющей тот же стандарт, если и возможно, то нередко довольно тяжело. Только в последних выпусках СУБД ведущих производителей обеспечивается совместимость с полным вариантом языка. Поэтому все из существующих сегодня развитых „реляционных“ СУБД сертифицированы уполномоченными органами максимум на базовый уровень *SQL*92.

Наконец, одновременно с завершением работ по определению стандарта *SQL*92 была начатая разработка стандарта *SQL*3. Общей точкой зрения ведущих производителей СУБД является то, что будущие продукты, владея более развитыми возможностями, должны быть совместные с предыдущими выпусками. Хотя многие разработчики и пользователи реляционных СУБД осознают наличие многих непреодолимых недостатков языка *SQL*, от него теперь уже невозможно отказаться (как невозможно отказаться от использования языка С в процедурном программировании). Следовательно, нужен новый стандарт языка, обеспечивающий такие очевидно необходимые возможности, как определённые пользователями типы данных, более развитые средства определения таблиц, наличие полного механизма триггеров и т.п. Необходим именно стандарт, а не наличие развитых частных версий языка, поскольку это выгодно и производителям, и пользователям СУБД.

В 2003 году был принят стандарт *SQL2003*, получивший развитие в 2006-м и 2008-м годах. В них уже получили право на существование XML-данные и генераторы последовательностей.

Наконец, маркетинговая политика компаний-производителей СУБД (как и других компаний) заключается в том, чтобы убедить потенциальных покупателей в наличии уникальных возможностей своего продукта. Поэтому они не могут ждать принятия нового стандарта и вынуждены использовать частные расширения языка. В целом ситуация несколько парадоксальная: появление нового стандарта в основном зависит от специалистов ведущих компаний, которые являются членами комитета *SQL3*, а текущая деятельность компаний усложняет работу этого комитета.

Существующие *SQL*-ориентированные СУБД

Первой системой, которая реализовала поддержку придуманного для этой системы и самого первого варианта *SQL,* была *System R*, разработанная фирмой IBM в конце 70-х гг. *System R* не была коммерческим продуктом, которым стало её развитие — ряд СУБД *DB2,* существующая сейчас на большинстве машин этой фирмы. Первой же коммерческой *SQL*-системой стала СУБД *Oracle*, выпущенная во второй половине 80-х гг. фирмой, носящей теперь то же название. К середине 90-х гг. промышленно поставляемых *SQL-систем* стало довольно много, но если говорить о наиболее распространённых, то к ним относились *DB2, Oracle, Sybase, Informix* и *Interbase*. С рыночной точки зрения между этими (и теми, что их „настигают“) системами существует твёрдая конкуренция, имеющая положительные для пользователя стороны: фирмы-разработчики систем постоянно следят за достижениями

конкурентов и подхватывают один у другого новые технологические идеи, не допуская больших отставаний. Именно поэтому у ведущих компаний и наиболее распространённых на сегодня СУБД *Oracle, MS SQL* и *PostgreSQL* столь близки возможности и диалекты языка *SQL*: *PL/SQL, T-SQL* и *PLpgSQL*. С другой стороны, как отмечалось, привнесение рыночных факторов в разработку стандартного общеупотребимого *SQL* сказывается отрицательно как на сроках разработки стандарта, так и на его качестве.

Существующие СУБД других моделей данных

Хотя в количественном отношении объектно-ориентированные системы явным образом и значительно уступают реляционным, все-таки их разнообразие также велико. Они образуют группу объектно-ориентированных СУБД, которые предназначены для того, чтобы дать возможность пользователю создавать в рамках объектного подхода объекты предметной области и работать с ними, используя преимущества объектно-ориентированных средств и систем разработки. К таким СУБД относятся *ONTOS, Gemstore, UniSQL* и др.

Нельзя не заметить сходства *объектного подхода* с *сетевой моделью* (стр. 18) проектирования баз данных, распространённой в 70-х гг. (модель данных *CODASYL*), практически не встречаемой сейчас в новых системах. Многие из идей, тщательно разработанных в своё время в подходе к нормализации данных *CODASYL*, появляются в более поздних системах во второй раз, как это, например, стало с процедурами баз данных, относительно недавно появившимися в *SQL*-системах. А система *InfoModeller* была реализована специально созданной для этой цели фирмой, которая объединила авторов этой модели данных.

Достаточно интересной и перспективной представляется СУБД *Caché*, разработчики которой обеспечили поддержку и взаимозаменяемость как реляционной, так и объектной моделей данных.

И наконец, нельзя не упомянуть получившие, благодаря социальным сетям, в последнее время широкое распространение СУБД на основе ряда «нереляционных» моделей данных, называемых *NoSQL*-моделями, такими как:

- модель ключ-значение (*Redis, MemcacheDB*), используемых для быстрого сохранения базовых данных;
- распределённая модель (*Column-oriented*) – *Cassandra, HBase,* ориентированных на работу с *Big Data*;
- документо-ориентированные СУБД (*MongoDB, Couchbase*), предназначенные для хранения иерархических структур данных.
- графовые СУБД – *OrientDB, Neo4J,* которые, по сути, являются «наследниками» или развитием иерархической модели данных.

Средства разработки

Сегодня существует большое число программных систем, позволяющих непосредственно вводить и генерировать запросы на языке ***SQL***, а также отображать результаты выполнения запросов. Это такие продукты, как *Windows Interactive SQL* (*WISQL*) фирмы Borland, поставляемым в пакете *Delphi*; *FirstSQL* фирмы IBM; *Visual Data Manager* (*VDA*), что входят в пакет *Visual Basic* (*VB*) фирмы Microsoft и ряд других. Однако, такие системы не применимы для создания эффективных приложений, в первую очередь, по следующим причинам: *во-первых*, они отображают результаты только в виде стандартной

двумерной таблицы с именами атрибутов в её заголовке; но, во-вторых, *более существенным недостатком* является то, что пользователь должен знать сам язык данных для построения запросов. Тем не менее, такие системы незаменимые для разработчиков при отладке корректных и эффективных *SQL*-инструкций.

Ряд современных систем, предназначенных для построения приложений с использованием языков высокого уровня, поддерживают *SQL* как подъязык данных или позволяют интегрировать в себя продукты, которые поддерживают *SQL*. К ним относятся: *Delphi*, с языком *Object Pascal* в основе, *Visual Basic* (*VB*), *Oracle Developer/2000* и *Designer/2000, Informix Newera,* работающий в комплексе с *Visual C++,* и сам пакет *Visual C++*.

И наконец, весьма распространённым является сегодня *Web*-интерфейс информационных систем и их подсистем. Его использование в значительной мере содействует унификации приложений и облегчает масштабирование информационных систем, как в структурном, так и в территориальном аспектах.

Построение этого типа интерфейсов требует знаний и навыков использования языков *PHP* и *JavaScript*, которые также имеют в своём арсенале элементы доступа к серверам БД и генерирования *SQL*-запросов.

Мы не будем рассматривать какой-нибудь конкретный продукт (*CASE*-систему) потому что все они, кроме *CASE*-приложений систем *Oracle* и *Informix*, поддерживают единый или близкие стандарты *SQL*.

Так, например, VB5 содержит ряд объектов, позволяющих работать с БД. В частности, объекты *rdoQuery* и *rdoConnection,* которые своими методами *Execute* генерируют запрос или выполняют *SQL*-выражение. Причём, если в *Connection.Execute SQL*-инструкция записывается как параметр, то *Query.Execute* параметров не имеет, а инструкция задаётся в свойстве „*SQL*“ этого объекта.

Для выполнения примеров *SQL*-запросов, предоставленных в следующих разделах, и отладки запросов из контрольных задач рекомендуется, прежде всего, приложение *Interactive SQL* из системы *PgAdmin*, которой комплектуется СУБД *PostgreSQL*, или система *Windows Interactive SQL* из комплекта СУБД *Borland Interbase* или *Firebird*.

Все эти системы работают с любым сервером БД, однако дальше будут приведены примеры, которые отлаживались с локальной версией СУБД *Firebird* 2.0 или специфические для СУБД *PostgreSQL* и отлаживались с версией 9 этой СУБД.

Замечание относительно примеров.

! Предметной областью для демонстрации команд и операторов *SQL* избран учебный процесс в **Одесском национальном политехническом университете**. В частности, *номер студенческой группы* является буквенно-цифровым: *ФС-ГГП*, где ФС — буквенный шифр специальности, ГГ — год формирования группы (поступления в университет), П или ПП — порядковый номер группы (например, АМ-823 — третья группа специальности „Вычислительные машины“ 1982 года поступления). *Полный номер студенческого билета* состоит из номера группы и порядкового номера студента в группе (АМ-823/12). *Значениями оценок* являются баллы рейтинговой системы: 100-90 баллов (или ‘A’ в европейской системе ECTS) — „отлично“, 89-85 (или ‘B’) — „ очень хорошо“, 84-75 (или ‘C’) — „хорошо“, 74-70 (или ‘D’) — „удовлетворительно“, 69-60 (или ‘E’) — „достаточно“, меньше 60 — „неудовлетворительно“: 35-59 (или ‘FX’) — должен прослушать курс повторно, меньше 35 (или ‘F’) — должны пройти повторное обучение.

Итак, *Приложение А* содержит информационную модель (в объектной нотации и нотации UML) учебного примера базы данных, которая будет создаваться в следующих разделах.

В *Приложении Б* приведен сценарий команд создания таблиц учебной базы данных. Это полный вариант команд, которые будут постепенно сформированы на протяжении раздела „Язык определения данных (ЯОД, DDL) SQL. Часть 1“.

Приложение В содержит учебный пример таблиц с данными, к которым будут посылаться запросы для манипулирования этой информацией.

Кое-где будут встречаться дополнительные примеры, отличные от описанных выше. На них будет обращаться внимание отдельно.

Инсталляция и администрирование СУБД

СУБД PostgreSQL

Мощность СУБД *PostgreSQL* можно сравнить, пожалуй, лишь с мощностью СУБД *Oracle*. Однако, в отличие от последней (как и многих других СУБД), является системой с открытым кодом.

Как и *Oracle*, эта СУБД также имеет развитый диалект языка *SQL — PLpg/SQL*, поддерживает межсерверное взаимодействие, не имеет ограничений относительно масштабирования систем, достаточно нетребовательна к оборудованию, работает под управлением всех основных операционных систем и т.п. Поэтому именно она рекомендована для выполнения задач в рамках этого курса и в качестве основы как учебных, так и реальных или коммерческих информационных систем.

Язык определения данных (ЯОД, DDL) SQL. Часть 1

ЯОД SQL состоит из *операторов* и *команд*, ориентированных на работу с такими объектами БД, как *поля, домены, таблицы, индексы и представления*.

Для отображения атрибутов отношения БД необходимо определить *типы* их значений и привести их в соответствие типам, допустимым конкретной СУБД.

Типы данных

В языке *SQL* поддерживаются следующие типы данных:

***CHAR**[ACTER](n)* — текстовая строка фиксированной длины в *n* символов. Максимальное значение *n* = 254 символа. Константа такого типа задаётся в апострофах (‘). Причём значение ‘‘ позволяет включить в структуру и сам апостроф (‘).

VARCHAR — текстовая структура переменной длины. Её максимальный размер зависит от СУБД — 256 - 8000 символов.

Аналогичный ему тип ***LONG**[VARCHAR]* — размером до 16K или 2G, в зависимости от реализации. Разные СУБД организуют хранение данных этих двух типов или в самой таблице, или, чаще всего, в *отдельном файле (или таблице)*. А в таблице хранятся только *ссылки на смещение* конкретного значения.

***BIT**(n)* — битовая строка *n*-й длины. Предназначена для хранения двоичных значений как *единого целого* (коды программ, изображения и т.д.).

Сегодня большинство СУБД вместо этого типа данных используют тип ***BLOB*** — *Binary Large Object*, который хранится аналогично типу *VARCHAR*.

***DEC**[IMAL](n,m)* — десятичное число в *действительном виде*, то есть с явной десятичной точкой. *n* — общее количество разрядов числа, *m* — количество разрядов после точки. Причём, $m \le n$. Максимальное количество разрядов *p* зависит от СУБД, но *p* должно быть не меньше *n*.

***NUMERIC**(n,m)* — аналогично *DEC*, кроме того, что *p* может быть не больше *n*.

***INT**[EGER]* — десятичное целое. Его максимальный размер и, соответственно, значения зависят от СУБД.

SMALLINT — поддерживается не всеми СУБД. Аналогичный *INT*, но в два раза меньше.

***FLOAT**(n)* — десятичное число с плавающей точкой, представленное *в экспоненциальной* форме $x \times 10^y$, где *n* — количество разрядов числа *x*, максимальное значение которого, как и максимальное значение числа *y*, зависит от конкретной реализации.

REAL — совпадает с *FLOAT*, за исключением того, что значение *n* равняется *максимальному* и устанавливается в зависимости от реализации.

DOUBLE — в 2 раза больше *REAL*.

DATE
TIME } — названия говорят сами за себя. Характерным является то, что практически все СУБД позволяют устанавливать формат ввода и отображения данных этих типов, хотя каждая СУБД хранит такие данные в своём *внутреннем специфическом формате*, независимом от формата отображения.

Мы перечислили простейшие стандартные типы данных, которые поддерживаются всеми без исключения СУБД. Однако почти в каждом диалекте *SQL* существует 15-20 типов данных, специфических именно для конкретной СУБД.

В частности, благодаря механизму пользовательских типов данных, которые будут рассмотрен ниже, в СУБД ***PostgreSQL*** включены дополнительные типы данных, которые в большинстве являются неоднородными структурами или составными типами (см. раздел „Структуры данных“).

Домены

Домены в *SQL* предназначены для определения **элементарной спецификации пользовательского типа данных** с возможностью её одновременного использования в нескольких таблицах. Однако домен в *SQL не соответствует в полной мере реляционным доменам*, в первую очередь потому, что это не более чем *синтаксическое упрощение*: значения, которые входят в домен, *должны базироваться на указанных ранее типах*, и, в результате, не являются истинными типами, определёнными пользователем.

Для создания домена используются следующие команды:

```
CREATE DOMAIN    имя_домена      тип_поля
    [DEFAULT значение]
    [список ограничений];
```

Пример

```
CREATE DOMAIN    Color       CHAR(7)
    DEFAULT '???'
    [CONSTRAINT Valid_Colors]
    CHECK(VALUE IN('красный', 'жёлтый', 'зелёный', 'синий', '???'));
```

Тогда *тип_поля* при определении аргумента будет указываться как *Color*, а значение аргумента должно быть включено в перечисленное множество.

Изменить или удалить значение по умолчанию и список ограничений данного домена можно командой, аналогичной приведенной, ALTER DOMAIN.

Уничтожить существующий домен можно командой

```
DROP DOMAIN    домен    опция;
```

где *опция* может принимать значения:

RESTRICT — домен *не будет* уничтожен, если на него *есть хотя бы одна ссылка* в таблицах;

CASCADE — команда будет выполнена *не только для самого домена*, но и для таблиц с атрибутами, определёнными на него основе: тип таких полей будет переназначен на *базовый тип* домена.

Поддержка пользовательских типов данных в СУБД PostgreSQL

Создание новых типов

В разделе „Структуры данных“ (см. стр. 8) упомянуты такие простейшие типы данных, как перечисляемые. В СУБД ***PostgreSQL***, кроме доменов, реализована возможность создания *истинно пользовательских типов* данных (*User Defined Type — UDT*) с определением их настоящих реляционных доменов путём перечисления всех возможных значений создаваемого типа данных:

```
CREATE TYPE    Пользовательский_тип    AS ENUM (значение_1, значение_2, …);
```

где ENUM — означает перечисление каких-то констант без указания их каждого базового типа. Например:

```
CREATE TYPE color AS ENUM ('красный', 'оранжевый', 'жёлтый', 'зелёный', 'синий',
    'фиолетовый');
```

Перечисляемый тип данных — это множество упорядоченных элементов, поэтому для работы с ним предназначены операторы порядка или сравнения (>, <, >=, <=) и специальные функции:

Таблица 1 – Функции работы с перечисляемым типом

Функция	**Описание**	**Пример**
enum_first (anyenum)	Возвращает первый элемент множества	enum_first(null::rainbow) = red
enum_last (anyenum)	Возвращает последний элемент множества	enum_last(null::rainbow) = purple
enum_range (anyenum, anyenum)	Возвращает диапазон элементов, которые расположены между указанными двумя элементами множества	enum_range('orange'::rainbow, 'green'::rainbow) = {orange,yellow,green}
		enum_range(NULL, 'green'::rainbow) = {red,orange,yellow,green}
		enum_range('orange'::rainbow, NULL) = {orange,yellow,green,blue,purple}

Однако для определения значений пользовательского типа вместо ENUM может быть указано любой базовый тип данных или их комбинацию, как указано ниже.

Составные типы данных

Составные типы определяются командой:

CREATE TYPE *Название_типа* AS (*Название_атрибута Тип_атрибута* [, ...])

Например, для создания типа *Address*, включающего данные о городе, улице, доме и квартиру, можно выполнить команду:

```
CREATE TYPE  Address AS
    (City VARCHAR, Street VARCHAR, House SMALLINT, Flat SMALLINT);
```

Теперь в таблицу *Student* можно добавить новый атрибут *Address* соответствующего типа:

```
ALTER TABLE  Student  ADD COLUMN     Address ADDRESS;
```

Для ввода данных в атрибут составного типа используется два варианта:

'(*значение*_1 , *значение*_2 , …)'

или

ROW(*значение*_1 ,*значение*_2 , …),

где *значение*_1, *значение*_2, … — значение *атрибута*_1, *атрибута*_2 и т.д. описания составного типа.

В запросах на получение данных для доступа к атрибутам составного типа используются скобки, например:

```
SELECT (Address).City FROM Student;
```

В СУБД *PostgreSQL* создано множество составных типов:

— типы, которые автоматически устанавливаются в процессе инсталляции, например, геометрические, сетевые и битовые;

— типы, поддержку которых необходимо устанавливать отдельно, например, многомерные кубы и деревья.

Хранение в БД атрибутов составных типов данных эффективно лишь при наличии специальных алгоритмов их обработки. Следовательно, для всех составных типов созданы соответствующие функции, позволяющие значительно расширить возможности манипулирования данными этих типов.

В следующих таблицах представлены примеры составных типов данных, а также операторов и функций, которые можно использовать с ними.

Таблица 2 – Геометрические типы данных

Тип	**Размерность**	**Описание**	**Форма представления**
point	16 байт	Точка на плоскости	(x,y)
line	32 байта	Бесконечная линия (не полностью реализованная)	((x1,y1),(x2,y2))
lseg	32 байта	Ограниченный линейный сегмент (отрезок)	((x1,y1),(x2,y2))
box	32 байта	Прямоугольник	((x1,y1),(x2,y2))
path	16+16n байт	Замкнутый путь (тоже самое, что и многоугольник)	((x1,y1),...)
		Открытый путь (ломанная линия)	[(x1,y1),...]
polygon	40+16n байт	Многоугольник (тоже самое, что и замкнутый путь)	((x1,y1),...)
circle	24 байта	Круг	<(x,y), r> (*центр* и *радиус*)

Таблица 3 – Операторы, которые можно использовать с геометрическими типами данных

Оператор	**Описание**	**Пример**
+ , -	Преобразование	box '((0,0),(1,1))' + point '(2.0,0)'
* , /	Масштабирование / поворот	box '((0,0),(1,1))' * point '(2.0,0)'
#	Сечение прямоугольников	'((1,-1),(-1,1))' # '((1,1),(-1,-1))'
#	Количество точек на пути или в многоугольнике	# '((1,0),(0,1),(-1,0))'
@-@	Длина круга	@-@ path '((0,0),(1,0))'
@@	Центр	@@ circle '((0,0),10)'
##	Ближайшая точка к первому и второму операнду	point '(0,0)' ## lseg '((2,0),(0,2))'
<->	расстояние между	circle '((0,0),1)' <-> circle '((5,0),1)'
&&	Перекрытие ?	box '((0,0),(1,1))' && box '((0,0),(2,2))'

Оператор	Описание	Пример
<< , >>	Размещено строго слева или справа?	circle '((0,0),1)' << circle '((5,0),1)'
<<| , |>>	Размещено строго снизу или сверху?	box '((0,0),(3,3))' <<| box '((3,4),(5,5))'
< , >	Размещено снизу или сверху с возможным касанием?	circle '((0,0),1)' < circle '((0,5),1)'
?#	Пересекаются ?	lseg '((-1,0),(1,0))' ?# box '((-2,-2),(2,2))'
?- , ?|	Линия является горизонтальной или вертикальной ?	?- lseg '((-1,0),(1,0))'
?-| , ?||	Линии перпендикулярны или параллельны ?	lseg '((0,0),(0,1))' ?-| lseg '((0,0),(1,0))'
@>	Круг включает точку ?	circle '((0,0),2)' @> point '(1,1)'

Таблица 4 – Функции обработки геометрических типов данных

Функция	Описание	Пример
area(object)	Область	area(box '((0,0),(1,1))')
center(object)	Центр	center(box '((0,0),(1,2))')
diameter(circle), radius(circle)	Диаметр и радиус круга	diameter(circle '((0,0),2.0)')
height(box), width(box)	Вертикальный, горизонтальный размер прямоугольника	height(box '((0,0),(1,1))')
isclosed(path)	Путь замкнут?	isclosed(path '((0,0),(1,1),(2,0))')
length(object)	Длина	length(path '((-1,0),(1,0))')
npoints(path)	Количество точек пути	npoints(path '[(0,0),(1,1),(2,0)]')
npoints(polygon)	Количество точек многоугольника	npoints(polygon '((1,1),(0,0))')
pclose(path), popen(path)	Замкнуть путь / Разомкнуть путь	pclose(path '[(0,0),(1,1),(2,0)]')

Кроме того существует множество функций преобразования одних геометрических типов в другие, некоторые из которых представлены ниже.

Таблица 5 – Функции преобразования геометрических типов

Функция	Описание	Пример
box(circle)	Круг в прямоугольник	box(circle '((0,0),2.0)')
box(point, point)	Точки в прямоугольник	box(point '(0,0)', point '(1,1)')
box(polygon)	Многоугольник в прямоугольник	box(polygon '((0,0),(1,1),(2,0))')
circle(box)	Прямоугольник в круг	circle(box '((0,0),(1,1))')
circle(polygon)	Многоугольник в круг	circle(polygon '((0,0),(1,1),(2,0))')
point(box), point(circle), point(lseg), point(polygon)	Центр прямоугольника, круга, линии, многоугольника в точку	point(box '((-1,0),(1,0))')

Для работы с данными, которые описывают элементы вычислительных сетей, в частности сетевых адресов, в диалекте *PostgreSQL* были созданы соответствующие составные типы:

Таблица 6 – Типы данных сетевых адресов

Тип	Размерность	Описание	Пример
cidr	7 или 19 байт	IPv4 и IPv6 сети	192.168.100.128/25 10.1.2.3/32
inet	7 или 19 байт	IPv4 и IPv6 Host-узлы и сети	
macaddr	6 байт	MAC адреса	'08002b:010203' '08:00:2b:01:02:03'

К стандартному битовому типу BIT(*n*) для хранения битовой сроки длиной *точно в n бит* в *PostgreSQL* добавлен тип BITVARYING(n) для хранения битовой сроки *переменного максимального размера в n бит* и созданы операторы работы с битовым типом данных:

Таблица 7 – Операторы обработки битового типа данных

Оператор	Описание	Пример
||	Конкатенация	'10001' || '011' = 10001011
&	AND	'10001' & '01101' = 00001
|	OR	'10001' | '01101' = 11101
#	XOR	'10001' # '01101' = 11100
~	NOT	~ '10001' = 01110
<< , >>	Сдвиг влево, право	'10001' << 3 = 01000

Дополнительные структурные элементы

В СУБД ***PostgreSQL*** существует ряд дополнительных элементов БД, которые расширяют эффективность информационных систем и возможности для разработчиков.

К таким элементам относятся ***многомерные массивы*** (*array*), которые являются и дополнительным элементом, и дополнительным типом данных.

Общий синтаксис описания массива определяется двумя вариантами:

'{ *значение*_1 *символ значения*_2 *символ* ... }'

или

ARRAY[*значение*_1 *символ значения*_2 *символ* ...],

где *символ* — символ-разделитель.

Для разделения стандартных типов используется запятая ‘,’, исключая тип BOX, для которого используется символ ‘;’. Каждый элемент *значения* — это константа или подмассив. Например, представление массива в операциях ввода данных может быть таким:

'{10000, 10000, 10000, 10000}';

'{{"Meeting", "Lunch"}, {"Training", "Presentation"}}');

или таким:

ARRAY[10000, 10000, 10000, 10000],

ARRAY[['Meeting', 'Lunch'], ['Training', 'Presentation']]);

Эффективность использования массива определяется возможностью доступа к элементу по него индексу. Первый элемент массива имеет индекс 1, который указывается в квадратных скобках: [1]. Если по указанному индексу элемент отсутствует, возвращается NULL. При использовании многомерного массива за первым индексом перечисляются дополнительные индексы, например [1][2]. Также поддерживаются срезы данных, когда номера индексов определяются через интервал, например [1:3].

Существуют три типа модификации элементов массива:

полная модификация — все содержимое массива заменяет новыми данными, которые задано массивом-константой;

модификация среза — модифицируется лишь интервальное подмножество элементов;

модификация элемента — модифицируется отдельный элемент по индексу.

Примеры изменения элементов массива:

```
UPDATE Sal_emp SET Pay_by_Quarter = '{25000,25000,27000,27000}'
    WHERE Name = 'Carol';

UPDATE Sal_emp SET Pay_by_Quarter = ARRAY[25000,25000,27000,27000]
    WHERE Name = 'Carol';

UPDATE Sal_emp SET Pay_by_Quarter[1:2] = '{27000,27000}'
    WHERE Name = 'Carol';

UPDATE Sal_emp SET Pay_by_Quarter[4] = 15000
    WHERE Name = 'Bill';
```

Для обработки массивов СУБД *PostgreSQL* предлагает операторы и функции, представленные в следующих таблицах.

Таблица 8 – Операторы обработки массивов

Оператор	Описание	Пример использования
=	равно	ARRAY[1,2,3] = ARRAY[1,2,3] = true
<>	не равно	ARRAY[1,2,3] <> ARRAY[1,2,4] = true
<	меньше чем	ARRAY[1,2,3] < ARRAY[1,2,4] = true
>	больше чем	ARRAY[1,4,3] > ARRAY[1,2,4] = true
<=	меньше чем или равно	ARRAY[1,2,3] <= ARRAY[1,2,3] = true
>=	больше чем или равно	ARRAY[1,4,3] >= ARRAY[1,4,3] = true
@> <@	элементы одного массива содержатся во втором	ARRAY[1,4,3] @> ARRAY[3,1] = true ARRAY[2,7] <@ ARRAY[1,7,4,2,6] = true
&&	перекрытие	ARRAY[1,4,3] && ARRAY[2,1] = true
||	конкатенация массивов	ARRAY[1,2,3] || ARRAY[4,5,6] = {1,2,3,4,5,6} ARRAY[1,2,3] || ARRAY[[4,5,6],[7,8,9]] = {{1,2,3},{4,5,6},{7,8,9}}
||	Включение элемента в массив	3 || ARRAY[4,5,6] = {3,4,5,6}

Таблица 9 – Функции обработки массивов

Функция	Описание	Пример использования
array_append (anyarray, anyelement)	Добавить элемент к массиву	array_append(ARRAY[1,2], 3) = {1,2,3}
array_cat (anyarray, anyarray)	Объединить два массива	array_cat(ARRAY[1,2, 3], ARRAY[4,5])= {1,2,3,4,5}
array_ndims (anyarray)	Получить размерность массива	array_ndims(ARRAY[[1,2,3], [4,5,6]])=2
array_dims (anyarray)	Получить текстовое представление размерности массива	array_dims(ARRAY[[1,2,3], [4,5,6]]) = [1:2][1:3]

Функция	Описание	Пример использования
array_fill (anyelement, int[], [, int[]])	Получить массив с инициализацией элементов	array_fill(7, ARRAY[3], ARRAY[2]) = [2:4]={7,7,7}
array_length (anyarray, int)	Получить размерность массива указанного уровня	array_length(array[1,2, 3], 1) = 3
array_lower (anyarray, int)	Получить наименьшую размерность указанного уровня массива	array_lower('[0:2]= {1,2,3}'::int[], 1) = 0
array_prepend (anyelement, anyarray)	Добавить элемент в начало массива	array_prepend(1, ARRAY[2,3]) = {1,2,3}
array_to_string (anyarray, text)	Получить текстовую строку из элементов массива, используя указанный символ-разделитель	array_to_string(ARRAY[1, 2, 3], '~~') = 1~~2~~3
array_upper (anyarray, int)	Получить самую большую размерность указанного уровня массива	array_upper(ARRAY[1, 2,3,4], 1) = 4
string_to_array (text, text)	Получить массив из текстовой сроки, используя указанный символ-разделитель	String_to_array('xx~~yy~~zz', '~~') = {xx,yy,zz}
unnest(anyarray)	Преобразовать массив в строки таблицы	Unnest(ARRAY[1,2])

Одним из наиболее удобных среди дополнительных структур данных диалекта ***PostgreSQL*** является такой элемент БД, как ***последовательность*** (*sequence*). Этот элемент является ещё одним вариантом одноименного составного типа (см. раздел „Структуры данных“), который содержит упорядоченный числовой ряд с фиксированным различием между его элементами (фиксированным шагом). В *PostgreSQL* эта структура реализуется в виде однострочной таблицы, самыми важными полями которой являются *Имя*, *Шаг* (или *Приращение*), *Минимальное_значение*, *Максимальное_значение*, *Начальное_значение*, *Текущее_значение*.

Для оперирования последовательностями в этом диалекте *SQL* используется ряд команд и функций. Прежде всего, это генератор последовательности, который реализуется командой создания:

CREATE SEQUENCE *название*;

В этом простейшем варианте будет создана таблица с именем *название*, в единственный кортеж которой по умолчанию будут записаны следующие значения: в поле *Имени* — название, поля *Шаг*, *Минимальное* и *Начальное_значение* получат значение 1, а поле *Максимальное_значение* — значение $\left(2^{63}-1\right)$ или 9 223 372 036 854 775 807, что можно проверить командой

SELECT * FROM *название*;

Такой вариант последовательности чаще всего используется в качестве автоматического генератора привилегированного идентификатора сущности, который будет рассмотрен позже в демонстрационном примере. В некоторых других СУБД упрощённым аналогом такой последовательности является счётчик.

Если необходимо задать другие характеристики последовательности, то для этого используется более расширенный синтаксис команды CREATE SEQUENCE:

CREATE [TEMP[ORARY]] SEQUENCE *название*
[INCREMENT [BY] *приращение*]
[MINVALUE *минимальное_значение* | NO MINVALUE]
[MAXVALUE *максимальное_значение* | NO MAXVALUE]
[START [WITH] *начальное_значение*];

Необходимо иметь в виду, что при отрицательном значении приращения поля *Максимальное* и *Начальное_значение* по умолчанию получат значение (-1), а поле *Минимальное_значение* — $\left(-2^{63}+1\right)$.

Функция NEXTVAL(*название*) меняет текущее значение последовательности „*название*“ на значение приращения и возвращает новое значение в виде величины типа Integer. *Именно эта функция используется при автоматической генерации идентификаторов сущностей.*

Функция CURRVAL(*название*) возвращает значение, полученное функцией NEXTVAL последним для последовательности „*название*“ в текущей сессии. Если функция NEXTVAL никогда не вызвалась для этой последовательности в этой сессии, то будет сгенерировано сообщение об ошибке.

Функция LASTVAL возвращает значение, которое наиболее часто возвращалось функцией NEXTVAL в текущей сессии. Эта функция идентичная CURRVAL, за исключением того, что вместо использования имени последовательности как аргумента она извлекает значение последней последовательности, для которой была использована функция NEXTVAL в текущей сессии. Если функция NEXTVAL никогда не вызвалась в этой сессии, то будет сгенерировано сообщение об ошибке.

SETVAL(*название, новое_значение*) выполняет „сброс“ счётчика последовательности „*название*“. Это означает, что при следующем вызове функция NEXTVAL возвратит и/или сгенерирует и возвратит значение (*новое_значение* + 1), а функция CURRVAL — значение *новое_значение.*

Для удаления последовательности или нескольких последовательностей одновременно используется команда:

DROP SEQUENCE *название1* [,*название2, ...*];

Таблицы

Для определения таблиц ЯОД *SQL* имеет такие команды: CREATE TABLE — для создания таблицы, ALTER TABLE — для модификации таблицы и DROP TABLE — для её уничтожения.

Наиболее простой вариант команды создания таблицы:

CREATE TABLE *таблица*
(*поле* *тип_поля* [DEFAULT *значение*][,
поле *тип_поля* [DEFAULT *значение*][,…]]);

Операнды *таблица* и *поле* определяют имена соответствующих элементов. В качестве *типа_поля* указывается или имя домена, или один из базовых типов. Причём любой числовой тип может указываться без размера, который будет принят по умолчанию, а параметр DEFAULT *отменяет аналогичное значение* домена. Пример:

```
CREATE TABLE    Student
      (SecondName     CHAR(30),
       FirstName      CHAR(20),
       Patronymic     CHAR(30),
       SNum           SMALLINT,
       Spec           CHAR(2)      DEFAULT ‘ОП’,
       Gnam           INTEGER,
       Address        CHAR(30),
       Email          CHAR(30));
```

Когда создаётся таблица, можно определить **ограничения** на значения полей таблицы. В *SQL* существует два типа ограничений: на конкретные столбцы или просто *на столбцы* и на подмножество столбцов или *на таблицу*. Ограничения на тот или иной столбец указываются в определении соответствующего поля и имеют позиционное значение *только* в том смысле, что размещаются *после объявления типа*:

*поле тип_поля ограничения*_С

Например, если для столбца не указанное значение по умолчанию и при добавлении кортежа (строки) опущено значение соответствующего атрибута, то в ячейку таблицы будет введено значение NULL.

Если такая ситуация для данного поля недопустима, то в *ограничения* необходимо ввести оператор NOT NULL. Например:

```
…   SecondName      char(30)      NOT NULL,
    FirstName       char(20)      NOT NULL, …
```

Этот вариант ограничения не используется как ограничение на таблицу, хотя при отсутствии значения соответствующего поля *будет отклонена вся операция* обновления.

Другим вариантом является *ограничения по значению*. Для этого в определении поля используется оператор CHECK. Например, может выполняться проверка значения на соответствие диапазона. Например:

```
CREATE TABLE    Rating
      (Kod    INTEGER    CHECK(Kod > 0)    NOT NULL,
       DKod   INTEGER    CHECK(Kod > 0)    NOT NULL,
       Mark   INTEGER    DEFAULT 0         CHECK(Mark >= 0 AND Mark <= 100),
       MDate  DATE       NOT NULL);
```

Кроме того, проверка может выполняться и на принадлежность значения заданному множеству:

```
… Spec     char(2)     CHECK(Spec IN(‘АП’, ‘ОС’, ‘АМ’, ‘АС’, ‘ОМ’, ‘ОИ’)), …
```

Ограничение по значению распространяется на ограничения на таблицу. Ограничение на таблицу записываются после определения последнего поля и отделяются от него запятой:

```
CREATE TABLE    таблица
      (поле     тип_поля     [ограничение_С][,
       поле     тип_поля     [ограничение_С][,…]],
       ограничение_Т);
```

Например (изменим предыдущий пример).

```
CREATE TABLE    Rating
      (Kod          INTEGER          NOT NULL,
```

```
DKod      INTEGER      NOT NULL,
Mark      INTEGER      DEFAULT 0,
CHECK(Kod > 0 AND DKod > 0 AND (Mark BETWEEN 0 AND 100)) ,
MDate     DATE    NOT NULL);
```

Этот пример не имеет преимуществ перед предыдущим вариантом, а только демонстрирует возможности ограничения на таблицу.

Ограничения по значению отклоняют операцию обновления, если значения атрибутов кортежа не отвечают условиям ограничений.

Если мы уже затронули проверки значений и использовали в примерах ряд операторов, применяемых при проверке значений, сделаем отступление от определения таблиц и рассмотрим

Логические операторы

Результатом их действия будет значение „истина“ или „неправда“. К ним относятся операторы сравнения, булевы операторы и специальные операторы.

Операторы сравнения аналогичны тем, что применяются или существуют в большинстве языков программирования: =, >, <, >=, <=, <>.

Их использование также аналогично большинству языков.

Булевы операторы в *SQL* реализуют 3 операции булевой алгебры, образующих полный базис: AND, OR, NOT.

Специальных операторов в *SQL* семь: IN, BETWEEN, LIKE, IS NULL, EXISTS, ANY (SOME) и ALL. Однако три последние операторы используются ***только*** с *вложенными подзапросами*, поэтому будут рассмотрены в соответствующем параграфе (стр. 110) раздела „Язык манипулирования данными (ЯМД, DML) SQL“.

Оператор IN полностью определяет *некоторое множество значений*. Например:

```
… Spec    CHAR(2)     CHECK(Spec IN(‘АП’, ‘АМ’, ‘АС’, ‘ОМ’, ‘ОИ’, ‘ОС’)), ...
```

Аналогом этого была бы довольно сложная конструкция типа:

```
… Spec    CHAR(2)     CHECK(Spec = ‘АП’ OR Spec = ‘AM’ OR … OR Spec = ‘OC’), …
```

Оператор BETWEEN, как было понятно из примеров, вместе с оператором AND задаёт *диапазон значений*. Причём границы диапазона *входят* в число допустимых значений и могут быть как числами, так и строками *ASCII*-символов. В последнем случае при сравнении строк разной длины более короткая строка *дополняется* пробелами, имеющими наименьший *ASCII*-код среди символов алфавита. Поэтому инструкция вида

```
… SecondName    char(30)   CHECK(SecondName BETWEEN ‘A’ AND ‘МАЛЫЙ’), …
```

позволит ввести любую фамилию от А до М, точнее, до МАЛЫЙ, но не позволит, например, МАЛАХОВ. Если необходимо *исключить границы* диапазона из числа допустимых значений, можно воспользоваться инструкцией следующего вида:

```
… Mark        INTEGER     CHECK((Mark BETWEEN 0 AND 100)
                                  AND NOT Mark IN(0,100))…
```

Оператор LIKE применяемый *только к символьным полям* типа CHAR и VARCHAR и используется для наложения *шаблонов* на строки.

Для этого в операторе используются специальные символы-**шаблоны**: символ „подчёркивание“ (‘_’), заменяющий *один любой* символ, и символ „процент“ (‘%’), заменяющий символьную *строку произвольной длины*. Например, шаблона „*д_м*“ отвечают строки „*дам*“, „*дым*“, „*дом*“ и т.п., а шаблону „*%М%В*“ в поле фамилии

… SecondName char(30) CHECK(SecondName LIKE ‘%M%B’), …

соответствуют и **М**АЛАХО**В**, и ЛО**М**ОНОСО**В**, но не соответствует, например, **М**АЛИНО**В***СКИЙ*. Для того, чтобы в шаблоне оператора LIKE использовать и сами символы ‘_’ и ‘%’ необходимо любой символ, например слеш ‘/’, определить как *ESCAPE*-символ и предварять им каждый из управляющих, в том числе и самого себя:

… SpName char(60) CHECK(SpName LIKE ‘%/_%/%%//%’ ESCAPE ‘/’), …

В этом примере команда CHECK пропустит любые символьные строки, в которых в любом месте, но последовательно встречаются символы ‘_’, ‘%’ и ‘/’. Например:

„В спец_фонд выделить 15**%** дохода, но не более 1000 грн./месяц“

Оператор IS NULL используется в командах ЯМД для определения кортежей, в которых *отсутствуют значения* тех или иных атрибутов. Например:

… Patronymic IS NULL …

соответствует кортежам, в которых отсутствует отчество. Обратный ему оператор IS NOT NULL позволяет *отсеять отсутствующие значения*.

В диалекте *SQL* СУБД ***PostgreSQL***, согласно расширению, введенному в стандарте *SQL*’99, были добавлены

Регулярные выражения.

Регулярные выражения применяют для:

— поиска в строке подстроки, удовлетворяющей шаблону регулярного выражения;
— поиска и замены в строке подстроки, удовлетворяющей шаблону регулярного выражения;
— проверки на соответствие заданной строки шаблона;
— извлечение из строки подстроки, удовлетворяющей шаблону регулярного выражения.

Синтаксис регулярных выражений определён стандартом *POSIX* (*Portable Operating System Interface for Unix* — переносимый интерфейс операционных систем *Unix*). В стандарт *SQL*’99 была добавлена возможность использования регулярных выражений с помощью оператора SIMILAR TO как развитие оператора LIKE:

— *строка* SIMILAR TO *шаблон*;
— *строка* NOT SIMILAR TO *шаблон*;

Как и оператор LIKE, оператор SIMILAR TO возвращает истину, если содержимое всей строки соответствует содержимому шаблона.

Шаблон оператора SIMILAR TO кроме символов ‘%’, ‘_’ использует дополнительные символы:

— символ ‘|' обозначает альтернативы элементов;
— символ ‘?’ обозначает повторение предыдущего элемента 0 или 1 раз;
— символ ‘*’ обозначает повторение предыдущего элемента 0 или больше раз;
— символ ‘+’ обозначает повторение предыдущего элемента 1 или больше раз;
— символы ‘{*m*}’ обозначают повторение предыдущего элемента ровно *m* раз;
— символы ‘{*m*,}’ обозначают повторение предыдущего элемента *m* или больше раз;
— символы ‘{*m, n*}’ обозначают повторение предыдущего элемента от *m* до *n* раз;
— символы ‘()’ группируют элементы в один логический блок;
— символы ‘[ ]’ определяют класс символов через перечисление допустимых символов или с использованием символа диапазона ‘-’;

Если указанные символы необходимо использовать в шаблоне как обычные, перед ними используют символ ‘\’. Также нужно помнить, что в шаблоне все символы ‘\’ нужно также дублировать, то есть писать как ‘\\’.

Рассмотрим несколько примеров демонстрации возможностей оператора SIMILAR TO в сравнении с оператором LIKE.

Для поиска строк, в которых название улицы из адреса студента не содержит цифр, можно выполнить запрос:

```
SELECT * FROM Student WHERE (Address).Street NOT SIMILAR TO ‘%[0-9]+%’;
```

В шаблоне регулярного выражения определён новый класс символов как диапазон допустимых цифр, количество которых от 1 и больше. Если исключить оператор NOT, будут получены строки, в которых присутствует любое количество цифр. Шаблон ‘[0-9]+’ обеспечит получение строк, в которых все символы — цифры.

Для поиска строк, у которых адрес электронного почтового ящика студента содержит 3 или 4 латинские буквы, после которых следуют 5 цифр, можно выполнить запрос:

```
SELECT * FROM Student WHER Email SIMILAR TO ‘%[a-z]{3,4}[0-9]{5}%’
```

Для поиска строк, в которых содержимое почтовых сообщений содержит действительные числа, например 12.24 или .12, можно выполнить запрос:

```
SELECT * FROM Letters WHERE Content SIMILAR TO ‘%[0-9]?.[0-9]+%’
```

Для поиска строк, в которых содержимое почтовых сообщений содержит время в форматах типа „9:15 am“ или „12:25 pm“, используется запрос:

```
SELECT * FROM Letters
    WHERE Content SIMILAR TO ‘%(1[012]|[1-9]):[0-5][0-9]? +(am|pm)%’
```

Так как указанный формат имеет два варианта определения цифр перед символом ‘:’, в шаблоне используются символы альтернативы и группировки в один логический блок.

Для поиска строк, в которых содержимое почтовых сообщений содержит время в форматах типа „01:10“ или „23:25“, можно выполнить запрос:

```
SELECT * FROM Letters
    WHERE Content SIMILAR TO ‘%([01]?[0-9]|2[0-3]):[0-5][0-9]%’
```

Следующие два приклада иллюстрируют использование *ESCAPE*-символа для символов-шаблонов оператора SIMILAR TO. Так, для поиска строк, в которых содержимое почтовых сообщений содержит телефонный номер в формате (777) 777-77-77, можно выполнить запрос:

```
SELECT * FROM Letters
  WHERE Content SIMILAR TO ‘%\\([0-9]{3,5}\\)[ ]+[0-9]{3}\\-[0-9]{2}\\-[0-9]{2}%’
```

Если необходимо искать мобильный телефонный номер в формате +38 (код) 777-77-77, где код может принимать значение 050, 063, 067 или 093 можно использовать шаблон

```
‘%\\+38\\((050|063|067|093)\\) +[0-9]{3}\\-[0-9]{2}\\-[0-9]{2}%’
```

Для поиска строк, в которых содержимое почтовых сообщений содержит *E-mail* адрес, например opu@opu.ua, используется запрос:

```
SELECT * FROM Letters
    WHERE Content SIMILAR TO ‘%[a-z0-9._%-]+@[a-z0-9._%-]+\.[a-z]{2,4}%’
```

СУБД *PostgreSQL* также поддерживает регулярные выражения с помощью следующих операторов:

— ‘~' поиск по шаблону, чувствительному к регистру символов;
— ‘~ *’ поиск по шаблону, нечувствительному к регистру символов;
— ‘! ~' поиск по исключающему шаблону, нечувствительному к регистру символов;
— ‘! ~ *’ поиск по исключающему шаблону, нечувствительному к регистру символов.

Эти операторы полностью обеспечивают стандарт *POSIX*. В отличие от оператора SIMILAR TO, они возвращают истину, если шаблон регулярного выражения соответствует любой части строки, поэтому начальный и конечный символ ‘%’ в шаблоне использовать нет необходимости.

В POSIX регулярное выражение определяется как один или несколько разделов, которые отделены символом ‘|’, и устанавливает соответствие между любой частью строки и содержанием одного из разделов. Раздел включает нуль или большее количественных атомов или ограничений, которые могут быть сцеплены. Регулярное выражение устанавливает соответствие с первым и следующими подразделами, а пустой подраздел соответствует пустой строке. Количественный атом может сопровождаться одиночным количественным показателем. Без количественного показателя регулярное выражение ищет соответствие одному атому, а с количественным показателем – нескольким атомам.

Атом может принимать одну из форм:

— символы ‘(*re*)’, где ‘*re*’ – любое регулярное выражение;
— символ ‘.’ (*точка*), что соответствует любому символу;
— символы ‘[*chars*]’, где ‘*chars*’ – любая последовательность символов;
— символы ‘*kkk*’, где ‘*kkk*’ – *ASCII*-код;
— символы ‘*c*’, где ‘*c*’ – специальный символ. Например: ‘*d*’ – только цифры, ‘*s*’ – пробел, ‘*w*’ – буквы и символ ‘_’, ‘*D*’ – противоположность ‘*d*’, ‘*S*’ – противоположность ‘*s*’, ‘*W*’ – противоположность ‘*w*’, ‘*n*’ – символ перехода на красную строку, ‘*r*’ – символ возврата каретки, ‘*t*’ – символ табуляции;
— символ ‘*x*’ – любой одиночный символ;
— символы ‘*m*’, где ‘*m*’ – положительное целое для определения атома через его обратную ссылку на *m*-й атом в форме ‘(*re*)’, что чаще всего используется для определения подстрок, которые последовательно повторяются;
— символы ‘*re1*(?=*re2*)’ обеспечивают предварительный просмотр подстроки, соответствующей атому ‘*re2*’ и находится после подстроки, отвечающей атому ‘*re1*’;
— символы ‘*re1*(?!*re2*)’ обеспечивают предварительный просмотр подстроки, которая противоречит атому ‘*re2*’ и находится после подстроки, отвечающей атому ‘*re1*’.

Внутри символов квадратных скобок запрещено использовать символы ‘*D*’, ‘*S*’ и ‘*W*’.

К сожалению, символы ‘*w’* и ‘*D’* корректно работают с буквами кириллицы не во всех кодировках, поэтому для учёта букв кириллицы необходимо использовать символы [*chars*], например, символы [А-Яа-Я].

Количественные показатели, как и для оператора SIMILAR TO, могут быть представлены следующими формами: символ ‘?’, символ ‘*’, символ ‘+’, символы ‘{*m*}’, символы ‘{*m*,}’ и символы ‘{*m, n*}’.

Для поиска строк, в которых фамилия студента содержит две и более одинаковых латинских буквы или буквы кириллицы, используется запрос:

```
SELECT * FROM Student WHERE SecondName ~ '([ Za-Яа-Я]+)\\1';
```

При обработке регулярного выражения вся группа символов между круглыми скобками помечается как переменная $1. Когда встречается \1, это определяется как ссылка на значение переменной $1, то есть повтор буквы.

Для поиска строк, в которых содержимое почтовых сообщений содержит три и больше одинаковых повторяющихся слов, используется запрос:

SELECT * FROM Letters WHERE Content ~ '([Za-Яа-Я]+)(\\s+)\\1\\2\\1';

При обработке регулярного выражения вся группа символов между круглыми скобками помечается как переменная $1. Следующий пропуск определяется следующей группой в круглых скобках и помечается как переменная $2. Когда встречается \1, это определяется как ссылки на значение переменной $1, то есть повтор слова. Переменная $2 эквивалентная одному и более пропусков, а переменная $1 снова ссылается на первое слово, то есть третий повтор слова.

При установлении соответствия между шаблоном и строкой определяется максимальное количество символов строки, совпадающих с символами шаблона. Для определения максимального количества символов в строке необходимо к символам количественных показателей добавить символ ‘?’.

Ограничения регулярных выражений определяются символами:

— символ ‘^’ устанавливает соответствие между шаблоном и символами строки только в её начале;
— символ ‘$’ устанавливает соответствие между шаблоном и символами строки только в конце строки;
— символ ‘*A*’ устанавливает соответствие между шаблоном и символами строки только в её начале при наличии нескольких строк;
— символ ‘*m*’ устанавливает соответствие между шаблоном и символами только в начале слова;
— символ ‘*M*’ устанавливает соответствие между шаблоном и символами только в конце слова;
— символ ‘*y*’ устанавливает соответствие между шаблоном и символами только в начале или в конце слова;
— символ ‘*Y*’ устанавливает соответствие между шаблоном и символами, которые не являются начальными или конечными;
— символ ‘*Z*’ устанавливает соответствие между шаблоном и символами строки только в конце строки при наличии нескольких строк.

Для поиска строк, в которых имя студента содержит первый символ ‘A' или последний символ ‘c' используется запрос:

SELECT * FROM Student WHERE SecondName ~ ‘(A|c$)’;

Для извлечения подстроки, удовлетворяющей шаблону регулярного выражения, из строки используется функция SUBSTRING(*string from pattern*).

Если в шаблоне присутствуют *круглые скобки*, то возвращается подстрока по шаблону внутри скобок.

Например, в результате выполнения функции

SUBSTRING(‘ E-mail my@ukr.net ‘ FROM ‘[a-z0-9._%-]+@[a-z0-9._%-]+\.[a-z]{2,4}’);

будет выделена подстрока „*my@ukr.net*“.

Если же в шаблон включить круглые скобки, то функция

SUBSTRING(‘ E-mail my@ukr.net ‘ FROM ‘([a-z0-9._%-]+)@[a-z0-9._%-]+\.[a-z]{2,4}’)

выделит подстроку „*my*“.

Для извлечения из текста почтовых сообщений подстрок, содержащих две цифры, которые следуют за двумя любыми символами, используется запрос:

```
SELECT SUBSTRING(Content FROM ‘(..[0-9]{2})’)
    FROM Letters WHERE Content SIMILAR TO ‘%([0-9]{2})%’;
```

Для извлечения из содержимого почтовых сообщений подстрок, которые содержат время в форматах типа „9:15 am“ или „12:25 pm“, используется запрос:

```
SELECT  SUBSTRING (Content FROM ‘((1[012]|[1-9]):[0-5][0-9]? (am|pm))’)
    FROM Letters WHERE Content SIMILAR TO ‘%(1[012]|[1-9]):[0-5][0-9]? (am|pm)%’
```

Для извлечения из содержимого почтовых сообщений подстрок, которые содержат телефонные номера в формате (777) 777-777-77, используется запрос:

```
SELECT SUBSTRING(Content FROM  ‘(\\([0-9]{3,5}\\)[␣]+[0-9]{3}\\-[0-9]{2}\\-[0-9]{2})’)
    FROM Letters WHERE Content SIMILAR TO ‘%\\([0-9]{3,5}\\)[␣]+[0-9]{3}\\-[0-
    9]{2}\\-[0-9]{2}%’
```

Для поиска и замены в строке *source* подстроки, удовлетворяющей шаблону регулярного выражения *pattern* в строке, на подстроку *replacement* используется функция

REGEXP_REPLACE(*source*, *pattern*, *replacement* [, *flags*]).

В строке шаблона *pattern* допустимы все ранее описанные метасимволы.

Строка *replacement* может включать шаблон ‘*n*’, где ‘*n*’ принимает значение от 1 до 9 и определяет *n*-ое скобочное подвыражение заменяемого шаблона. Опция *flags* может принимать значение *g*, что указывает на замену всех совпадений, а не только первого.

Например, в результате выполнения функции

REGEXP_REPLACE(‘2+2=4’, ‘\\d’, ‘цифра’)

вместо первой двойки будет включена строка ‘цифра’: *цифра*+2=4

В то же время, в результате выполнения функции

REGEXP_REPLACE (‘2+2=4’, ‘\\d’, ‘цифра’, ‘g’)

все цифры будут заменены строкой ‘цифра’: *цифра+цифра=цифра*

При выполнении функции

regexp_replace(‘ d@9 dd’, ‘\\w@\\d', ‘XY’, ‘g')

будет выполнена замена подстроки *d@*9 на подстроку *XY*, в то время как при выполнении функции

regexp_replace(‘ d@9 dd’, ‘\\w(@)\\d', ‘X\\1Y', ‘g')

подстрока *d@*9 будет заменена на *X@Y* согласно элементу шаблона замены \1, который ссылается на первое скобочное подвыражение.

Дополнительные функции REGEXP_MATCHES, REGEXP_SPLIT_TO_TABLE и REGEXP_SPLIT_TO_ARRAY используют регулярные выражения для формирования из строки множества подстрок.

Функция REGEXP_MATCHES(*string*, *pattern* [, *flags*]) возвращает из строки *string* массив подстрок согласно шаблону *pattern*. Если шаблон содержит скобочные

подвыражения, функция возвращает массив, в котором *n*-й элемент определён как результат соответствия строки *n*-му скобочному выражению. Значение опции *flags* эквивалентны функции REGEXP_REPLACE.

Функция REGEXP_SPLIT_TO_TABLE(*string*, *pattern* [, *flags*]) возвращает из строки *string* множество подстрок согласно шаблону *pattern* в виде разделителя между подстроками. Функция REGEXP_SPLIT_TO_ARRAY имеет тот же синтаксис, но возвращает массив подстрок.

Выполнив запрос

SELECT REGEXP_MATCHES(‘Привет Мир!’, ‘(Привет) (Мир)’);

получаем массив подстрок {‘Привет’, ‘Мир’}.

Выполнив запрос

SELECT REGEXP_MATCHES(‘приветмирпрощаймир’, ‘(п[с]+)(с[п]+)’,’g');

получаем два массива подстрок {‘привет’, ‘мир’}, {‘прощай’, ‘мир’}.

Выполнив запрос

SELECT REGEXP_SPLIT_TO_ARRAY(‘Как прекрасен этот мир! Посмотри!’, ‘\\s+’);

получаем массив подстрок {‘Как, прекрасен, этот, мир!’, ‘Посмотри!’}

Выполнив запрос

SELECT ch FROM REGEXP_SPLIT_TO_TABLE(‘Как прекрасен этот мир!’, ‘\\s+’) AS ch;

получаем ответ в виде записей с подстроками в виде отдельных слов: ‘Как, прекрасен, этот, мир!’

Выполнив запрос

SELECT ch FROM REGEXP_SPLIT_TO_TABLE(‘Как прекрасен этот мир!’, ‘\\s*’) AS ch;

получаем ответ в виде записей, которые содержат подстроки из одного символа: ‘К’, ‘а’, ‘к’, ‘␣’, ‘п’, ‘р’, ‘е’, ‘к’, ‘р’, ‘а’, ‘с’, ‘е’, ‘н’, ‘␣’, ‘э’, ‘т’, ‘о’, ‘т’, ‘␣’, ‘м’, ‘и’, ‘р', ‘!’

Это все логические операторы *SQL*.

Возвращаемся к **определению таблиц**. Одним из вариантов ограничений, как на столбцы, так и на таблицу являются определения ключей.

Потенциальные ключи

Вспомним, что одним из свойств потенциального ключа является его *уникальность*. То есть в БД ***не может*** **существовать двух или более разных записей (строк) с *одинаковым* значением ключа**. Однако все СУБД, и это естественно, допускают в одной таблице (отношении) наличие множества одинаковых значений того или другого атрибута. Для того чтобы избежать или запретить дублирование значений атрибута, выбранного в качестве потенциального ключа, ещё на этапе построения БД используется следующее определение потенциального ключа:

UNIQUE [(*список_атрибутов*)]

Причём это определение может использоваться и как ограничение на столбец, и как ограничение на таблицу. Например, в таблице *Student*

… SNum SMALLINT UNIQUE, …

Здесь есть *две некорректности*. *Первая* заключается в том, что, как было отмечено ранее, потенциальный ключ не может содержать атрибутов, допускающих отсутствие значений. Следовательно, необходимо добавить NOT NULL. Но это не отменяет *вторую* некорректность: номер студента по списку уникален для данной группы, но может повторяться в другой группе и, тем более, на другой специальности. Поэтому номер зачётной книжки состоит из шифра специальности, номера группы и номера студента:

```
… SNum SMALLINT   NOT NULL,
  Spec CHAR(2)    NOT NULL   CHECK(Spec IN('ОИ', 'ОМ', 'ОП', 'ОС')),
  GNum INT        NOT NULL,
  UNIQUE(Spec, GNum, SNum));
```

Первичный ключ

Мы с вами отмечали, что *один и только один* из потенциальных ключей может использоваться в качестве первичного. Обсуждали мы это в теоретическом плане. Остаётся только сообщить об этом серверу БД. Для этого в *SQL* существует оператор PRIMARY KEY:

PRIMARY KEY [(*список_атрибутов*)]

Изменим предыдущий пример:

```
…  PRIMARY KEY (Spec, GNum, SNum));
```

Это пример использования первичного ключа как *ограничения на таблицу*. Иногда более целесообразно использовать *одноатрибутный* привилегированный идентификатор, то есть использовать первичный ключ как *ограничения на столбец*. Например, в таблицу Student можно добавить поле

```
…  Kod  INT    NOT NULL    PRIMARY KEY, …
```

которое обеспечивает сквозную нумерацию по университету или содержит, например, налоговый код (уникальность в пределах государства).

Внешние ключи

Понятие *внешних* и *родительских* ключей были введены для *формализации связей* между отношениями (таблицами). В *SQL* внешний ключ является ограничением, то есть его цель — ограничить данные, которые вводятся, так, чтобы он и его родительский ключ соответствовали принципам **ссылочной целостности**, сформулированным в определении внешнего ключа (см. стр. 44).

Как ограничения на таблицу *SQL* предоставляет следующий синтаксис для внешнего ключа:

FOREIGN KEY *список_атрибутов*
    REFERENCES *базовая_таблица* [*список_атрибутов*]

Например.

```
CREATE TABLE   Student
    (Kod          INT    NOT NULL   PRIMARY KEY    CHECK(Kod>0),
     SecondName   CHAR(30)    NOT NULL,
     FirstName    CHAR(20)    NOT NULL,
     Patronymic   CHAR(30),
     SNum         SMALLINT    NOT NULL,
```

```
Spec        CHAR(2)     NOT NULL   CHECK (Spec IN('ОИ','ОМ','ОС','ОП')),
GNum        INT    NOT NULL,
Address     CHAR(30),
Email       CHAR(30),
FOREIGN KEY (Spec, GNum)       REFERENCES Decanat (Spec, GNum));
```

Если во внешнем ключе присутствующий *только один* атрибут, то синтаксис будет немного другой. Создадим ещё одну таблицу для примера:

```
CREATE TABLE   Speciality
    (SpKod    CHAR(11)    NOT NULL   CHECK(SpKod LIKE '_ . _ _ _ _ _ _ _ _ _'),
     ScDirect CHAR(20)    NOT NULL,
     SpName   CHAR(40)    NOT NULL   UNIQUE,
     Spec     CHAR(2)     NOT NULL   UNIQUE
     CHECK(Spec IN('ОИ', …, 'ОС')));
```

Тогда в таблице Student можно изменить инструкцию

```
…   Spec   CHAR(2)    REFERENCES Speciality (Spec), …
```

Если родительским является первичный ключ, например, в таблице *Speciality* вместо *SpKod — Spec*, то в определении внешнего ключа можно обойтись *без имён атрибутов*:

```
…   Spec   CHAR(2)    REFERENCES Speciality, …
```

Теперь разберёмся, почему внешние ключи являются ограничениями. В частности, как они влияют на выполнение команд ввода (INSERT) и обновления (UPDATE).

Для детализированной таблицы, которая содержит внешний ключ, возможны два варианта изменений в зависимости от того, разрешены или нет для внешнего ключа *NULL*-значения. Если *NULL*- значения *запрещены*, то **добавлять в такую таблицу новый кортеж можно только в том случае, если в базовой таблице существует родительский ключ, *совпадающий* с внешним ключом нового кортежа**. Если же *NULL*-значения во внешнем ключе *разрешены*, то **ключ получает такое значение при отсутствии соответствующего ему родительского ключа**. В таком случае вводить новые кортежи можно всегда. Удалять кортежи из детализированной таблицы *также можно в любом* случае, независимо от значения внешнего ключа (DELETE).

Использование *NULL*-значений во внешних ключах для соблюдения принципа ссылочной целостности является не только желательным, но и необходимым в том случае, если *внешний ключ ссылается на первичный ключ **собственной** таблицы*: *SQL* допускает и такой вариант. Практически в любой литературе по языкам приводится следующий пример этой ситуации: информация о служащих некоторой фирмы сведена в таблицу, в которой один из атрибутов служащего содержит ссылку на него руководителя, в свою очередь, также являющегося служащим:

```
CREATE TABLE   Employees
    (EmpNo   INT          NOT NULL     PRIMARY KEY,
     Name    CHAR(30)     NOT NULL     UNIQUE,
     Manager INT          REFERENCES Employees);
```

Для базовых таблиц, *содержащих родительские* ключи, ситуация *с* добавлением и *с удалением или* модификацией *кортежей полностью* обратная: добавлять новые записи можно неограниченно, а удалять и модифицировать необходимо так, чтобы придерживаться

определения внешнего ключа. В частности, внешний ключ должен или содержать *NULL*-значение, или совпадать с родительским.

В *SQL* для изменения или удаления кортежей с родительским ключом существует три возможности:

1. Можно ограничить или запретить модификацию родительского ключа. Это означает, что его изменение ограничено в том смысле, что допустимо, если на родительский ключ нет ссылок из детализированных таблиц.
2. Можно при изменении родительского ключа автоматически изменять и внешние ключи, ссылающиеся на него. В этом случае говорят, что изменения выполняются каскадно.
3. Можно менять или удалять кортеж с родительским ключом, при этом значения соответствующих ему внешних ключей автоматически устанавливаются (*и здесь этот случай делится ещё на два*):
   — в NULL;
   — в значения по умолчанию.

Для реализации этих трёх случаев используются следующие операторы:

1. RESTRICTED (*ограничено*) или NO ACTION (в *SQL*\92);
2. CASCADE[S] (*каскадно*), (без S в *SQL*\92);
3. NULLS или SET NULL (*SQL*\92);
   SET DEFAULT.

Например, внесём изменения в таблицу Student.

```
CREATE TABLE    Student
      (Kod              INT     NOT NULL     PRIMARY KEY     CHECK(Kod>0),
       SecondName       CHAR(30)        NOT NULL,
       FirstName        CHAR(20)        NOT NULL,
       Patronymic       CHAR(30),
       SNum             SMALLINT        NOT NULL,
       Spec             CHAR(2)         REFERENCES Speciality (Spec)   NOT NULL,
       GNum             INT     NOT NULL,
       Address          CHAR(30),
       Email            CHAR(30),
       UNIQUE(Spec, GNum, SNum),
       FOREIGN KEY (Spec, GNum)     REFERENCES Decanat (Spec, GNum)
       ON UPDATE CASCADE
       ON DELETE CASCADE,
       UPDATE OF Speciality CASCADES,
       DELETE OF Speciality RESTRICTED);
```

В диалекте ***PostgreSQL*** ограничения на столбцы типа UPDATE OF, DELETE OF и т.п. отсутствуют. Однако их роль играют инструкции ON UPDATE или ON DELETE, которые указываются в качестве ограничений на соответствующие столбцы вместе с инструкцией REFERENCES.

И наконец, если вы являетесь *владельцем* или *создателем* таблицы (что это означает, мы рассмотрим чуть позже), то вы имеете возможность *модифицировать* и *удалять* таблицу.

Для модификации в *SQL* существует команда

```
ALTER TABLE     таблица     операция  объект[,
                таблица     операция  объект[, …]];
```

где *таблица* — имя таблицы, *операция* — ADD (*добавить*) или DROP (*удалить*), *объект* — столбец (при добавлении указываются все необходимые параметры) или ограничение типа UNIQUE, PRIMARY KEY, CHECK и т.д.

Например:

```
ALTER TABLE  Rating
      DROP   Mark,
      ADD    MarkECTS    CHAR(2)
                         CHECK(MarkECTS      IN('A', 'B', 'C', 'D', 'E', 'FX', 'F'))
                         DEFAULT 'F';
```

Соответственно, благодаря этой команде, можно:

— добавлять новые столбцы;
— определять для существующего столбца новое или заменить старое значение по умолчанию;
— удалять для существующего столбца значения по умолчанию;
— задавать новое ограничение целостности для таблицы;
— удалять существующее ограничение целостности для таблицы.

Относительно *добавления* и *удаления ограничений* необходимо обратить внимание на то, что для выполнения этих действий ограничения должны иметь имена (названия). Пример поименования ограничений приведен в п. „Домены“ этого раздела. Однако необходимо отметить, что все развитые СУБД, в том числе *PostgreSQL, всем* непоименованным пользователем ограничениям предоставляет *автоматические* имена, которые можно посмотреть средствами приложения *PgAdmin*.

Уничтожается таблица командой

```
DROP TABLE    таблица      опция;
```

Опциями может быть RESTRICT или CASCADE. Если при удалении таблицы устанавливается опция RESTRICT, а сама таблица содержит родительские ключи, на которые *существуют ссылки*, то команда будет *отвергнута*. Если же в такой ситуации установлена опция CASCADE, то команда будет выполнена, а все *ссылки* на неё будут *уничтожены* (заменены на NULL или DEFAULT).

Создание объектно-реляционных связей в PostgreSQL

В разделе „Объектно-реляционная модель“ отмечалось, что эта модель должна поддерживать основные элементы объектного подхода.

Так, в частности, в ***PostgreSQL*** поддерживается такой элемент, как **наследование**, благодаря механизму создания соответствующих объектно-реляционных связей. То есть таблица может наследовать некоторые атрибуты от одной или нескольких таблиц, что приводит к созданию отношений типа „*предок-потомок*“. В результате таблицы-потомки имеют те же атрибуты и ограничения, что и их таблицы-предки, а также дополняются собственными атрибутами.

При составлении запроса к таблице-предку можно требовать, чтобы запрос осуществил выборку или только из самой таблицы, или же просмотрел таблицу с её таблицами-потомками. При запросе к таблице-потомку в результат не включаются строки из таблицу-предка.

При создании таблицы-потомка на основе таблицы-предка используется оператор INHERITS:

CREATE TABLE *таблица-потомок* (…)
INHERITS (*таблица-предок* [*атрибут*_1, *атрибут*_2, …])

Рассмотрим **пример**. Пусть существует таблица *Student.*

Таблица *Master* с данными о студентах-дипломниках может наследовать все атрибуты таблицы *Student*, дополняя названием дипломной работы:

```
CREATE TABLE    Master
    (Diploma    VARCHAR)
     INHERITS (Student);
```

Индексы

Довольно тривиальной является задача, в которой необходимо найти и вывести содержимое кортежа по значению первичного ключа. Простота решения является следствием свойства уникальности первичного ключа. Однако записи в таблице не упорядочены согласно значению этого ключа (то есть атрибута или подмножества атрибутов).

При этом довольно часто возникает необходимость найти кортеж или группу кортежей с конкретным значением совсем другой подмножества атрибутов. Например:

```
SELECT  *    FROM    Speciality
    WHERE   ScDirect = ‘Компьютерные науки’;
```

Результат выполнения данного запроса будет получен *довольно быстро*, потому что первая же цифра значения атрибута *SpKod*, являющегося первичным ключом, соответствует научному направлению. Если бы это было не так, а впрочем, и в данном случае, такой поиск потребовал бы полного перебора строк таблицы *Speciality*.

Для ускорения решения этой задачи в *SQL* поддерживаются *индексы*.

Индекс — это **упорядоченный** (в алфавитном или числовому порядке) **список** содержимого столбцов или группы столбцов в таблице.

Управление индексами *существенным образом замедляет* выполнение операций обновления (INSERT, DELETE), и, кроме того, индекс занимает *дополнительное место в памяти*. Поэтому это средство *SQL* не было внесено в стандарт ANSI, хотя поддерживается всеми СУБД, и существовал ещё до введения этого стандарта. Синтаксис команды создания индекса выглядит следующим образом:

CREATE INDEX *индекс* ON *таблица* (*столбец*[, *столбец*[, …]]);

Если указано *более одного атрибута* для создания одного индекса, то данные упорядочиваются по значению *первого поля, внутри групп, которые* получились, осуществляется упорядочивание по *значению второго* поля, *внутри полученных* групп — по *значению* третьего и т.д.

Существенный нюанс: будучи однажды созданным, индекс является *невидимым* для пользователя. Сервер БД „сам решит“, когда есть смысл воспользоваться индексом, и сделает это *автоматически*.

Инструкция

CREATE UNIQUE INDEX *индекс* ON *таблица*(*поле*[, *поле*[, …]]);

фактически дублирует первичный ключ. При этом если таким образом индексируется

таблица с *уже существующими* кортежами, содержащими *одинаковые значения* атрибутов указанных в инструкции, то эта команда будет *отклонена*.

В зависимости от СУБД для таблицы может быть создано *до 250* индексов. Уничтожить индекс можно командой

DROP INDEX *индекс*;

которая *не меняет* содержимого полей.

Представления

Представление (VIEW) — объект, который **не содержит собственных данных. Это поименованная *производная виртуальная таблица*, которая *не может существовать самая по себе*, а определяется в терминах одной или нескольких поименованных таблиц** (*базовых таблиц* или других *представлений*).

В общем случае термин **производная таблица** обозначает таблицу, которая определяется в терминах других таблиц и, в конечном итоге, в терминах *базовых таблиц*, то есть является результатом выполнения каких-либо реляционных выражений над ними. **Базовая таблица** — это таблица, которая не является производной.

Различие же между базовой таблицей и представлением характеризуется следующим образом:

— базовые таблицы „*реально существуют*“ в том смысле, что они представляют данные, действительно хранимые в БД;

— представление же „*реально не существуют*“, а просто представляют разные схемы *просмотра* „реальных“ данных. Другими словами, представления подобный *окнам*, через которые просматривается информация, содержащаяся в базовых таблицах.

Более того, любые изменения в основной таблице будут *автоматически и немедленно* выданы через такое „окно“. И наоборот, изменения в представлении будут *автоматически и немедленно* применены к его базовой таблице.

В действительности же **представления — это запросы**, которые выполняются каждый раз, когда представление является объектом команды *SQL*.

Создание и уничтожение представления осуществляется командами CREATE VIEW и DROP VIEW.

Формат команды **уничтожения** довольно прост:

DROP VIEW *представление*;

Команда **создания** представления имеет формат:

CREATE VIEW *представление*[(*имена_столбцов*)] AS *запрос*;

Например:

```
CREATE VIEW    Rating_D    AS
      SELECT   Kod, Mark, MDate  FROM  Rating
      WHERE    DKod=1 AND Mark >= 30;
```

Это представление будет содержать коды, рейтинг и даты проведения модуля по первой дисциплине для тех студентов, у которых этот рейтинг не менее 30 баллов:

KOD	MARK	MDATE
1	82	2011-10-10
2	90	2011-10-10
3	46	2011-10-11
4	54	2011-10-10
5	86	2011-10-10

Значение фразы о том, что представление — это запрос, наиболее видно при обращении к представлениям, как к таблицам. Таким образом, команда

```
SELECT   *      FROM  Rating_D
      WHERE    Mark < 60;
```

на практике конвертируется и оптимизируется в команду:

```
SELECT   Kod, Mark, MDate      FROM      Rating
      WHERE    DKod=1 AND Mark >= 30 AND Mark < 60;
```

В этом довольно простом представлении в качестве имён полей использовались непосредственно имена полей таблицы, лежащей в основе представления. Иногда, как в приведенном примере, этого достаточно. Но иногда желательно дать новые имена столбцам представления. А иногда — это просто необходимо. Например, в тех случаях когда:

а) некоторые столбцы являются *выходными* и, следовательно, *не поименованы*. Такая ситуация часто возникает при *объединении* отношений с *разными именами* однотипных атрибутов;

б) два или более столбцов имеют *одинаковые имена* в таблицах, которые принимают участие в *соединении*.

Имена, которые станут именами столбцов представления, указываются в круглых скобках после него имени. Типы данных и размеры *автоматически выводятся* из полей запроса. Например:

```
CREATE VIEW  Rating_D(Student_Number, Discipline_Number,  Rating, Date_of_Mark)
      AS
         SELECT   Kod, DKod, Mark, MDate    FROM      Rating
               WHERE    Mark >= 30;
```

При этом пример с выборкой значений трансформируется в следующий:

```
SELECT   *      FROM      Rating_D
      WHERE    Rating < 60;
```

В некоторых системах интенсивность запросов на выборку данных из представлений настолько большая, что сводит на нет преимущества этого элемента ЯОД. В таких случаях, когда к тому же интенсивность обновления несравнимо меньше интенсивности выборки или периодичность модификации значительно больше выборки, целесообразным становится хранение данных, отображаемых через представление. Для решения указанной проблемы в некоторых СУБД (в частности, *Oracle* и *PostgreSQL*) введен дополнительный элемент ЯОД *SQL* — **материализованное представление**.

В СУБД *Oracle* простейший вариант синтаксиса команд его создания и удаления совпадает с синтаксисом соответствующих команд для обычного представления:

```
CREATE MATERIALIZED VIEW    представление[(имена_столбцов)]  AS      запрос;
```

В СУБД ***PostgreSQL*** версий до 9.2 включительно материализованное представление — это объединение таблицы и обычного (виртуального) представления:

CREATE TABLE *таблица* [(*имена_столбцов*)] AS *запрос*;

То есть в этих версиях *PostgreSQL* материализованное представление — это *реальная* таблица, но которая создаётся на основании запроса.

Начиная с версии 9.3 в *PostgreSQL* материализованные представления (MATERIALIZED VIEW) используют систему правил, аналогичную обычным представлениям (VIEW), но результаты их работы сохраняют в табличном объекте. Таким образом, основные различия между табличными объектами, созданными с помощью запросов:

```
CREATE TABLE  NewTable   AS
                         SELECT  *  FROM  SourceTable;
CREATE VIEW   SimpleView AS
              SELECT     *        FROM  SourceTable;
```

и

```
CREATE MATERIALIZED VIEW  MatView  AS
                                   SELECT  *  FROM  SourceTable
```

заключаются в следующем:

NewTable – обычная таблица, имеющая схему результирующей таблицы подзапроса (в данном примере - схему таблицы *SourceTable*) и заполненная сразу после создания данными из таблиц(-ы) подзапроса (*SourceTable*). Всё дальнейшее манипулирование данными в таблице *NewTable* никак не связано с таблицами подзапроса. Соответственно, обновление таблиц *NewTable* и *SourceTable* абсолютно независимо и не влияет друг на друга.

SimpleView – „обычное“ представление. Соответственно, изменения, вносимые в базовую таблицу *SourceTable,* будут отражены в виртуальной таблице *SimpleView* при очередном обращении к ней, а изменения, вносимые в представление *SimpleView*, будут немедленно внесены в базовую таблицу *SourceTable* при условии выполнения „ограничений обновления“ и при условии создания правил обновления.

MatView – материализованное представление, т.е. „реальная“ таблица, получившая схему и первоначальные данные, аналогично примеру с *NewTable,* но при этом запрос, используемый для создания материализованного представления, хранится точно так же, как запрос обычного (как в примере с *SimpleView).* Однако, в отличие от обычной таблицы *NewTable,* данные в *MatView* нельзя в дальнейшем изменять напрямую, а в отличие от обычного представления *SimpleView,* изменения в базовой таблице *SourceTable* будут перенесены в материализованное *MatView* только после выполнения запроса:

REFRESH MATERIALIZED VIEW *MatView*;

До этого момента запросы, ссылающиеся на материализованное представление, будут отображать „устаревшие" данные.

В системных каталогах *PostgreSQL* информация о материализованном представлении хранится точно такая же, как для таблицы или обычного представления. Соответственно для анализатора материализованное представление – это такое же отношение, как таблица или обычное представление. Когда какой-либо запрос ссылается на материализованное представление, данные возвращаются непосредственно из материализованного

представления, как из таблицы. Правило, описанное при его создании, используется только для заполнения материализованного представления.

К представлениям мы ещё вернёмся при вводе и обновлении информации.

Позднее вернёмся и к DDL и рассмотрим такие элементы как *курсоры*, *триггеры*, *хранимые процедуры* и *пакеты*.

Демонстрационный пример

Запишите *SQL*-запрос для создания таблиц, спроектированных в задаче „Демонстрационный пример“ раздела „Формализация связей между отношениями“.

П.1. Обоснуйте заданные типы полей.

П.2. Запишите *SQL*-запросы для создания домена операций с билетами.

П.3. Задайте и обоснуйте ограничения значений всех полей. В частности:

— ограничения на поля создавайте с использованием специальных логических операторов;

— реализуйте связи между таблицами с помощью ограничения внешних ключей.

Решение.

Обратите внимание, что в этом примере *SQL*-запросы, которые созданы согласно задаче, включены в *SQL*-сценарий (*SQL-script*). Его структура будет рассмотрена в разд. „Сценарии“. Сейчас отметим только, что это текстовый файл с *SQL*-запросами, которые будут *последовательно выполнены*. Поэтому их позиция очень важна именно при создании БД: сначала прописываются команды создания пользовательских типов данных, далее — доменов, потом — базовых таблиц с родительскими ключами и лишь за ними — команды создания детализированных таблиц с внешними ключами.

Итак, *пример сценария для построения таблиц*:

```
CONNECT  c:\work\SQL\lectures  USER MEV  PASSWORD <password>;
CREATE SEQUENCE s_aircompany;
CREATE SEQUENCE s_voyage;
CREATE SEQUENCE s_class;
CREATE SEQUENCE s_passenger;
CREATE SEQUENCE s_employee;
CREATE SEQUENCE s_ticket;

/* Domain: operation, Owner: MEV */
CREATE DOMAIN operation CHAR(7) DEFAULT ‘покупка’
                                CHECK (VALUE IN (‘покупка’, ‘бронь’));

/* Table: aircompany, Owner: MEV */
CREATE TABLE aircompany
      (id_aircompany INT PRIMARY KEY DEFAULT NEXTVAL(‘s_aircompany’),
       name CHAR (50),
       country CHAR (30));
```

```
/* Table: voyage, Owner: MEV */

CREATE TABLE voyage
       (id_voyage INT PRIMARY KEY DEFAULT NEXTVAL('s_voyage'),
        number CHAR(20),
        destination CHAR (30),
        date_departure DATE,
        time_departure TIME,
        date_arrival DATE,
        time_arrival TIME,
        type_aircraft CHAR(30),
        aircompany INT REFERENCES aircompany(id_aircompany));

/* Table: class, Owner: MEV */

CREATE TABLE class
       (id_class INT PRIMARY KEY DEFAULT NEXTVAL('s_class'),
        name CHAR (30),
        voyage INT REFERENCES voyage(id_voyage),
        price DECIMAL(50,2),
        quantity_place INT);

/* Table: passenger, Owner: MEV */

CREATE TABLE passenger
       (id_passenger INT PRIMARY KEY DEFAULT NEXTVAL('s_passenger'),
        full_name CHAR (30),
        sex CHAR (1) CHECK(sex IN('м','ж')),
        passport CHAR (30));

/* Table: employee, Owner: MEV */

CREATE TABLE employee
       (id_employee INT PRIMARY KEY DEFAULT NEXTVAL('s_employee'),
        full_name CHAR(30),
        position CHAR(30),
        date_birth DATE,
        passport CHAR(10),
        identification_code FLOAT,
        adress CHAR(55),
        telefon CHAR(15));

/* Table: ticket, Owner: MEV */

CREATE TABLE ticket
       (id_ticket INT PRIMARY KEY DEFAULT NEXTVAL('s_ticket'),
        passenger INT REFERENCES passenger(id_passenger),
        class INT REFERENCES class(id_class),
        place INT,
        employee INT REFERENCES employee(id_employee),
        operation operation);

EXIT;
```

Язык манипулирования данными (ЯМД, DML) SQL

Основные команды ЯМД SQL

ЯМД *SQL* реализует или позволяет реализовать практически все основные операции реляционной алгебры. Основные операторы ЯМД — это SELECT, INSERT, UPDATE и DELETE. Первый называют **оператором выборки**, а другие — **операторами обновления**. *Сначала рассмотрим простейшие варианты обновления, а потом выборку и в конце вернёмся к обновления*.

Все *новые кортежи* в *SQL* **вводятся** в таблицы с помощью команды INSERT:

INSERT INTO *таблица* VALUES(*значение*, *значение*, …);

В таком варианте команды количество значений должно соответствовать количеству атрибутов таблицы. Значения при этом имеют *позиционное значение* и не могут содержать *выражения*. Например:

INSERT INTO Rating VALUES(111, 5, 85, ‘19.11.2011’);

Если же необходимо „пропустить“ значение какого-нибудь поля при вводе, это можно реализовать *двумя способами*:

— вставить *NULL*-значение в соответствующую позицию:

```
INSERT INTO Student
    VALUES(111, ‘Малахов’, ‘Евгений’, NULL, 12, ‘АМ’, 823, NULL, NULL);
```

Это возможно только в том случае, если в данном поле *допустимы* такие значения;

— *явным образом указать имена* столбцов, в которые будут введены новые значения. Пропущенным атрибутам будут даны или *NULL*-значения, если они допустимы, или значения по умолчанию, если они существуют (заданы). Пример:

INSERT INTO Rating(MDate, Kod, DKod) VALUES (‘21.11.2011’, 222, 3);

Строки из таблицы можно **исключить** с помощью команды

DELETE FROM *таблица* [*условие*];

Если *не* указанное *условие*, то команда выполняет „очистку“ таблицы:

DELETE FROM Rating;

Использование условия позволяет удалять *несколько кортежей* таблицы:

DELETE FROM Student WHERE GNum < 941;

Если в условии указанное имя *первичного ключа*, то это обеспечит удаление *одного единственного* кортежа.

! Обратите внимание на оператор WHERE. Из этого и ряда других примеров, приведенных раньше, видно, что этот оператор фактически реализует в *SQL* реляционную операцию **выборки (селекции)**.

Очевидным являются вопросы: или можно удалять „*текущую строку*“? Можно, но для этого потребуется ещё один элемент ЯОД *SQL* — **курсор**, который мы рассмотрим чуть позже.

Следующая команда ЯОД *SQL* позволяет **изменять** значения **уже существующих** полей таблицы:

UPDATE *таблица* SET *поле = значение*[, *поле = значение*[, …]] [*условие*];

Это простейший формат команды, Причём ключевое слово SET обеспечивает реализацию реляционной операции **проекции** на подмножество схемы отношения, представленного таблицей. Например:

```
UPDATE  Rating  SET  Mark = 30;
```

Эта команда обеспечивает запись значения 30 в поле *Mark* **всех кортежей** таблицы. Более того, в отличие от инструкции VALUES команды INSERT, в инструкции SET *допустимо использование скалярных выражений*. Например:

```
UPDATE  Rating  SET  Mark = Mark +(70 - Mark)*0.2;
```

Для модификации неединичного подмножества столбцов их необходимо перечислить в выражении SET:

```
UPDATE  Rating  SET  Mark = NULL, DKod = 2;
```

Этот пример демонстрирует и возможность „очистки“ поля, то есть изменения его значения на *NULL без использования* каких-нибудь *дополнительных средств*.

И, наконец, операция **выборки** из **проекции** позволяет изменить значения полей лишь части кортежей таблицы. Для этого в команду UPDATE добавляется *условие*. Например:

```
UPDATE  Student  SET  patronymic = 'Валериевич', GNum = 842
    WHERE        Kod = 111;
```

Следующая команда, которую мы рассмотрим и которую уже использовали в ряде примеров — это SELECT (*выборка*). Эту команду можно, пожалуй, назвать **основной в ЯМД** *SQL*, потому что она используется не только сама по себе, но и позволяет в значительной мере расширить другие команды манипулирования данными. И, кроме того, на **её основе строится** практически половина элементов *SQL* и, соответственно, запросов **ЯОД** *SQL*.

Несмотря на название этой команды, **она не является**, в чистом виде, **реализацией** реляционной операции **выборки**. Её простейший формат выполняет операцию **проекции:**

```
SELECT  список_столбцов  FROM  таблица;
```

Например:

```
SELECT  Kod, DKod, Mark  FROM  Rating;
```

Если необходимо вывести содержимое *всех полей* таблицы, то в команде вместо *список_столбцов* указывается шаблон *. Например:

```
SELECT  *  FROM  Student;
```

При этом необходимо учитывать, что столбцы будут выведены в том порядке, в каком они физически хранятся в таблице или перечислены в представлении.

Если же столбцы при выводе, например в интерактивном режиме, *необходимо переставить*, то их опять-таки *перечисляют* в команде.

Необходимо отметить, что по умолчанию команда SELECT операцию **проекции** реализует *не полностью*, не до конца. Дело в том, что *проекция* на подмножество схемы отношения подразумевает **исключение** из реляционной таблицы **совпадающих кортежей.** Команда же SELECT, например:

```
SELECT  SecondName  FROM  Student;
```

выведет фамилии *всех студентов* (из всех кортежей) в том числе и *повторяющиеся*. Это

пример, в котором, как и во всех предыдущих, по умолчанию используется *параметр* ALL. В общем виде формат команды из данного примера следующий:

SELECT ALL *список_полей* FROM *таблица*;

Для того чтобы *исключить одинаковые кортежи* из результата, то есть полностью выполнить операцию проекции, необходимо ключевое слово ALL заменить на DISTINCT. Например:

SELECT DISTINCT SecondName, FirstName FROM Student;

Ещё раз обращаем внимание, что с помощью инструкции DISTINCT из результата проекции удаляются кортежи, которые *полностью* совпадают. Соответственно в последнем примере повторяемыми будут считаться строки, в которых совпадают *и имя, и фамилия*.

Следующее *расширение* команды SELECT *аналогично* команде обновления (UPDATE) — это **выборка** и **проекция**. И так же, как в UPDATE, для этого используется оператор WHERE *условие*. Например:

SELECT * FROM Rating WHERE DKod = 2 AND Mark >= 60;

со следующим результатом:

KOD	DKOD	MARK	MDATE
2	2	76	12.10.2011
3	2	85	12.10.2011
4	2	85	12.10.2011

Причём, в *условии* команд SELECT, UPDATE и DELETE могут использоваться как *операторы сравнения*, так и *булевы операторы*, что видно из последнего примера. Кроме того, в *условии* этих команд, так же, как в *ограничениях* таблиц и доменов, могут использоваться *специальные операторы* LIKE, BETWEEN, IN и плюс ещё оператор IS NULL, упоминавшийся ранее. Приведём несколько примеров:

```
SELECT * FROM Student
    WHERE Spec IN('ОИ', 'ОС');
```

Запрос выведет все имеющиеся в таблице Student данные о студентах специальностей ОИ („Экономическая кибернетика“) и ОС („Прикладная математика“).

```
SELECT Kod, DKod, Mark, MDate FROM Rating
    WHERE DKod = 2 AND Mark BETWEEN 30 AND 60;
```

Запрос отобразит коды и рейтинг по конкретной (здесь — второй) дисциплине студентов, имеющих допуск к итоговому контролю по этой дисциплине, то есть имеют рейтинг в пределах 30 и 60 баллов,

```
SELECT * FROM Student
    WHERE SecondName LIKE '%M%B';
```

Запрос выберет студентов, фамилии которых заканчиваются буквой B, а в середине или в начале имеют букву M, и выведет все данные о них.

```
SELECT * FROM Student
    WHERE Patronymic IS NULL;
```

Запрос выберет и выведет все данные о студентах, не имеющих отчества.

Кроме условных операторов, в командах ЯМД, конкретно в SELECT, используются так называемые **агрегатные функции**, которые **возвращают некоторое *единственное* значение поля *для подмножества* кортежей таблицы**.

Агрегатные функции

В стандарте *SQL* определено 5 таких функций, хотя любая реализация языка содержит в 4 - 6 раз больше функций. Особенностью агрегатных функций является то, что, используя имена полей как аргументы, сами они указываются в команде SELECT *вместо* или *вместе* с полями. Это функции AVG (*average*) и SUM (*summa*), которые используются *только для числовых полей*, и функции MAX, MIN и COUNT, которые могут применяться и *для числовых, и для символьных атрибутов*. Так функции AVG и SUM позволяют вычислить соответственно среднее значение и сумму значений конкретного атрибута во **всех кортежах** таблицы. Например:

SELECT AVG(Mark) FROM Rating [WHERE DKod = 8];

Этот запрос возвращает средний рейтинг, подсчитанный, если условие отсутствует, по всем кортежам таблицы или, при наличия условия, только по кортежам, которые соответствуют восьмой дисциплине.

Названия функций MAX и MIN говорят сами за себя. Функция же COUNT подсчитывает *количество значений* в столбце или *количество строк* в таблице.

Например:

SELECT COUNT(*) FROM Student;

Использование ‘*’ как атрибута обеспечит подсчёт количества *всех* строк в таблице, *включая повторяемые и NULL*-евые (которых, по идее, быть не должно).

Для подсчёта количества не-*NULL-евых* значений какого-либо атрибута используется формат:

SELECT COUNT([ALL] *поле*) FROM *таблица*;

Ключевое слово ALL используется *по умолчанию*. Например:

SELECT COUNT(Patronymic) FROM Student;

Кстати, все остальные агрегатные функции в любом случае *игнорируют NULL*-значение.

Если же необходимо подсчитать количество *разных* значений какого-либо поля и *исключить* при этом *NULL*-значения, то в аргументах функции необходимо использовать оператор DISTINCT. Например:

SELECT COUNT(DISTINCT Patronymic) FROM Student;

Кстати, оператор DISTINCT может быть указан *в любой* агрегатной функции, но в MAX и MIN он *лишний*, а в SUM и AVG — не имеет смысла, так как повлияет на результат.

Ещё одна особенность агрегатных функций — возможность использования *скалярных выражений* в качестве их аргументов, в которых, правда, не должны быть самых агрегатных функций. Например:

```
SELECT AVG(LongevityInc + DegreeInc + TitleInc + SpecialInc)
    FROM SalaryIncrements;
```

В некоторых расширениях *SQL*, в частности *PostgreSQL* последнее ограничение снято.

Скалярные выражения могут быть аргументами и самой команды SELECT, также как и команды UPDATE. В этом случае в них могут входить и поля, и числовые константы, и агрегатные функции.

Например:

```
SELECT  (MAX(LongevityInc)+MAX(DegreeInc)+MAX(TitleInc)+MAX(SpecialInc))/4
    FROM    SalaryIncrements;
```

Символьные константы в выражениях использоваться *не могут*. Но зато их можно непосредственно включить в выходную таблицу результата выборки *в интерактивном режиме*.

Например:

```
SELECT  Kod, DKod, Mark, 'на 14.10.11'    FROM    Rating;
```

Результат — все строки таблицы с дополнительным полем:

Kod	DKod	Mark	
1	1	82	на 14.10.11
1	2	10	на 14.10.11
…	…	…	…
8	1	0	на 14.10.11

Или, например:

```
SELECT  AVG(Mark), 'к 10-го недели'    FROM    Rating;
```

Из определения агрегатных функций видно, что они *не могут* использоваться в одном SELECT-запросе вместе с полями, потому что возвращают *одно единственное значение*. Например (***неправильно***):

```
SELECT  Kod, MAX(Mark)    FROM    Rating;
```

Однако на практике может возникнуть такая задача: необходимо *получить среднее значение Mark в пределах определённой дисциплины*.

Решить эту задачу поможет оператор GROUP BY, который **обеспечивает выделение подгруппы с конкретным значением атрибута или подмножества атрибутов** и применение агрегатных функций к каждой подгруппе. Например:

```
SELECT  DKod, AVG (Mark)    FROM    Rating
    GROUP BY    DKod;
```

Результат может быть следующий:

DKod	
1	60
2	64
3	30
4	97
5	65

Теперь расширим этот пример. Предположим, что необходимо из отношения (таблицы), полученной в предыдущем примере, выполнить **выборку** кортежей, для которых *среднее значение* больше чем, например, 75. Мы уже знаем, что операция выборки реализуется с помощью оператора WHERE. Однако *результирующая таблица* этого примера строится *на основе групп* с использованием агрегатной функции, поэтому здесь существует ряд ограничений.

Для решения подобных задач, где в условии запроса используется агрегатная функция или поле, на которое выполняется проекция, в *SQL* был введен ещё один оператор, реализующий операцию **выборки** — HAVING.

В результате запрос, соответствующий поставленной задаче будет выглядеть следующим образом:

```
SELECT  DKod, AVG(Mark)  FROM  Rating
    GROUP BY  DKod
    HAVING  AVG(Mark) > 75;
```

с таким результатом:

DKod	
4	97

В выражении HAVING можно использовать и просто имена полей. Единственное условие при этом — такое поле или подмножество полей должны иметь одно и то же значение в пределах группы.

Например:

```
SELECT  DKod, AVG(Mark)  FROM  Rating
    GROUP BY  DKod
    HAVING  DKod <> 4;
```

для исключения из результата дисциплины с кодом 4.

Или, например:

```
… HAVING  NOT  DKod  IN(3, 4);
```

для исключения из результата дисциплин с кодом 3 и 4:

DKod	
1	60
2	64
5	65

Таким образом, оператор HAVING аналогичен WHERE за исключением того, что строки отбираются не просто по значениям столбцов, а *строятся* из значений столбцов указанных в GROUP BY и значений агрегатных функций, вычисленных для каждой группы, созданной GROUP BY.

Если же необходимо добавить в запрос с GROUP BY условие, содержащее поле, которое не используется для группировки, то такое условие, как и раньше, задаётся с помощью WHERE и даже одновременно с HAVING. Например:

```
SELECT  Kod, AVG(Mark)  FROM  Rating
    WHERE  DKod = 5
    GROUP BY  Kod
    HAVING  NOT  Kod  IN(3, 4);
```

Этот запрос исключит из подсчёта среднего рейтинга кортежи, которые касаются студентов, имеющих код 3 или 4, и подсчитает его для пятой дисциплины.

Следующий момент, который мы рассмотрим, связан с ***сортировкой значений,*** полученных *в результате* запроса SELECT. Дело в том, что кортежи в таблице хранятся *в порядке их поступления* (ввода). Соответственно при выводе (по умолчанию) этот порядок будет сохранен. Исключение составляют проиндексированные поля. Очевидно, более удобным для использования является вывод множества кортежей таблицы, *отсортированных* согласно значениям множества символьных и/или числовых полей *в прямом или обратном порядке*. Для обеспечения такой возможности в *SQL* введен оператор ORDER BY. Например:

```
SELECT  *  FROM  Student  ORDER BY SecondName;
```

Однако в этом примере кортежи, имеющие одинаковое значение фамилии, могут оказаться неупорядоченными по имени, отчеству и т.п., что также неудобно. Исправим этот пример:

```
SELECT  *  FROM  Student  ORDER BY SecondName, FirstName, Patronymic;
```

Для сортировки кортежей в обратном алфавитном порядке значений или атрибута по убыванию используется ключевое слово DESC. Например:

SELECT DKod, Mark FROM Rating ORDER BY DKod, Mark DESC;

(по *DKod — прямой,* по *Mark — обратный* порядок).

Причём поле, по которому выполняется сортировка, *не обязательно* должно присутствовать в подмножестве, на которое выполняется проекция.

Это особенно актуально во встроенном режиме, когда результаты разных операций проекции того же самого отношения выводятся в разные окна. Например:

1) SELECT DKod FROM Rating ORDER BY DKod;

2) SELECT Kod, Mark FROM Rating ORDER BY DKod, Mark DESC;

Аналогичные подходы могут использоваться и для таблиц, полученных в результате группировки. Например:

SELECT DKod, AVG(Mark) FROM Rating GROUP BY DKod ORDER BY DKod;

Однако отсортировать подобную таблицу по результатам агрегатной операции не представляется возможным, потому что такое поле не поименовано и не существует ни в базовой таблице, ни в представлении. На помощь здесь приходит внутренняя (автоматическая) нумерация выходных столбцов согласно порядку перечисления в команде SELECT. Например:

SELECT DKod, AVG(Mark) FROM Rating GROUP BY DKod ORDER BY 2 DESC;

Запрос даст один из полученных выше результатов, но в другом порядке:

DKod	
4	97
5	65
2	64
1	60
3	30

К сожалению, это единственный способ обратиться к таким столбцам, несмотря на то, что их можно поименовать, задав им псевдонимы или, используя уже принятый термин, **алиасы**. Например:

```
SELECT DKod Discipline, AVG(Mark) Averrage_Rating
    FROM Rating GROUP BY DKod;
```

Ранее мы отметили, что любой диалект *SQL* содержит намного больше агрегатных функций или подобных агрегатным. Так в ***PostgreSQL*** были введенные ***аналитические*** или ***оконные*** функции, которые кроме задач агрегирования, выполняют ещё и часть операций над многомерными структурами.

Аналитические функции

С помощью аналитических функций *SQL*-запросы оформляются проще и выполняются быстрее по сравнению с использованием чистого языка *SQL* в следующих случаях:

— при подсчёте промежуточной суммы, когда необходимо показать суммарную зарплату сотрудников отдела построчно, чтобы в каждой строке выдавалась сумма зарплат всех сотрудников вплоть до указанного;

— при подсчёте процентов в группе, когда необходимо показать, какой процент от общей зарплаты по отделу составляет зарплата каждого сотрудника;

— при запросе первых *N*, когда необходимо найти *N* сотрудников с самыми большими зарплатами или *N* товаров, которые пользуются самым большим спросом по регионам;

— при подсчёте скользящего среднего, когда необходимо получить среднее значение по текущим и предыдущим *N* строкам.
— при выполнении ранжирующих запросов, когда необходимо показать относительный ранг зарплаты сотрудника среди других сотрудников того же отдела.

Аналитические функции имеют следующий синтаксис:

имя_функции (аргумент, аргумент, …) OVER (описание_среза_данных)

Значение *имени_функции* определяется стандартными функциями агрегации (SUM, COUNT, MIN, MAX, AVG) и специальными функциями.

Описание_среза_данных определяется конструкциями: конструкция *фрагментации*, конструкция *упорядочивания*, конструкция *окна*.

Конструкция фрагментации определяется ключевым словом PARTITION BY и логически разбивает результирующее множество на группы по критериям, задаваемым выражениями секционирования. Аналитические функции применяются к каждой группе независимо и для каждой новой группы они сбрасываются. Если не указать конструкцию фрагментации, все результирующее множество считается одной группой. Каждая аналитическая функция в запросе может иметь уникальную конструкцию фрагментации. Синтаксис конструкции фрагментации аналогичен синтаксису конструкции GROUP BY в обычных *SQL*-запросах:

PARTITION BY *выражение* [, *выражение* [, *выражение*]]

Рассмотрим два запроса:

```
SELECT DeptKod, Job, SUM(Salary) sum_sal FROM Lecturer GROUP BY DeptKod, Job;

SELECT SecondName, DeptKod, Job,
    SUM(Salary) OVER (PARTITION BY DeptKod, Job) sum_sal FROM Lecturer;
```

Первый запрос вернёт сумму зарплаты преподавателей, сгруппированных по кафедрам и должностям. Второй запрос вернёт список преподавателей, для каждого из которых определяется сумма зарплаты преподавателей, имеющих ту же самую кафедру и должность.

Конструкция упорядочивания определяется ключевым словом ORDER BY и задаёт критерий сортировки данных в каждом фрагменте, как и в каждой группе. Конструкция ORDER BY в аналитических функциях имеет следующий синтаксис:

ORDER BY *выражение* [ASC | DESC] [NULLS FIRST | NULLS LAST]

Необходимо учитывать, что строки будут упорядочены только в пределах фрагментов (групп). Конструкции NULLS FIRST и NULLS LAST указывают, где при упорядочивании должны быть значения *NULL* — в начале или в конце.

Рассмотрим два запроса:

```
SELECT SecondName, DeptKod, Job, HireDate,
    SUM(Salary) OVER (PARTITION BY DeptKod, Job) sum_sal FROM Lecturer;

SELECT SecondName, DeptKod, Job, HireDate, SUM(Salary)
    OVER (PARTITION BY DeptKod, Job ORDER BY HireDate) sum_sal FROM Lecturer;
```

В запросе без конструкции упорядочивания для каждой строки (то есть по преподавателю) выводится общая сумма зарплат по всей группе преподавателей, сгруппированных по кафедре и должности, то есть определяются затраты университета на зарплату преподавателей по указанной группе на текущий момент.

В запросе с конструкцией упорядочивания преподаватели отсортированы по дате зачисления на работу, и для каждой строки выводится сумма зарплат преподавателя, описанного этой (текущей) строкой, и работающих в том же подразделении и на той же должности преподавателей, описанных предыдущими строками, то есть определяются затраты университета на зарплату преподавателей по указанной группе не на текущий момент, как в предыдущем запросе, а с учётом дат их приёма на работу:

SECONDNAME	DEPTKOD	JOB	HIREDATE	SUM_SAL
Погорецкая	2	Доц.	1980-09-01	1000
Чугунов	2	Доц.	1985-09-01	2250
Малахов	2	Зав. каф.	1990-09-01	1500
Юхименко	3	Проф.	1982-09-01	1300
Востров	4	Доц.	1982-09-01	1100

Чтобы получить список преподавателей с их именами, датой зачисления, зарплатой, где в каждой строке будет определены текущие затраты университета на зарплату преподавателей, можно выполнить запрос:

```
SELECT SecondName, DeptKod, Job, HireDate,
    SUM(Salary) OVER (ORDER BY HireDate) sum_sal FROM Lecturer;
```

Проанализировав результат запроса, можно определить, что без конструкции фрагментации группой считаются все строки, в каждой строке по преподавателю выводится сумма его зарплаты и зарплат преподавателей, которые зачислены раньше, а для преподавателей с одинаковыми значениями даты зачисления зарплаты суммируются в рамках этой условной группы.

Конструкция окна используется для оконных функций, то есть когда в секции упорядочивания указано ORDER BY, и может иметь один из синтаксисов:

[RANGE | ROWS] *начало_окна*

[RANGE | ROWS] BETWEEN *начало_окна* AND *конец_окна*

Описание начала и конца окна может определяться как:

UNBOUNDED PRECEDING | *значение* PRECEDING
    CURRENT ROW
UNBOUNDED FOLLOWING | *значение* FOLLOWING

Окно может быть создано по диапазону значений данных с использованием ключевого слова RANGE или по сдвигу относительно текущей строки с использованием ключевого слова ROWS.

Конструкция окна позволяет задать окно данных в пределах группы, которое перемещается или жёстко привязано (набор, интервал) и с которым будет работать аналитическая функция. Например, конструкция диапазона RANGE UNBOUNDED PRECEDING определяет применение аналитической функции к каждой строке данной группы от первого до текущего. Стандартным является жёстко привязанное окно, которое начинается с первой строки группы и продолжается до текущего.

Ключевое слово UNBOUNDED снимает фиксированную границу окна.

Ключевые слова PRECEDING и FOLLOWING задают верхнюю и нижнюю границы агрегирования, то есть интервал строк, „окно“ для агрегирования.

Возможны такие варианты окон:

— если нижняя граница окна фиксирована, то есть совпадает с первой строкой упорядоченной некоторым образом группы строк, а верхняя граница ползёт, то есть совпадает с текущей строкой в этой группе, то получаем нарастающий итог (кумулятивный агрегат), когда размер окна меняется, расширяясь в одну сторону, и само окно движется за счёт расширения;

— если нижняя и верхняя границы фиксированы относительно текущей строки в этой группе, например, первая строка до текущей и вторая строка после текущей, то получаем скользящий агрегат, когда размеры окна фиксированы или никуда не расширяются, а именно окно движется или скользит.

Рассмотрим следующий пример запроса:

```
SELECT SecondName, DeptKod, Job, SUM(Salary)
        OVER (PARTITION BY DeptKod, Job ORDER BY HireDate
                ROWS BETWEEN UNBOUNDED PRECEDING
                                        AND CURRENT ROW) sum_sal
    FROM Lecturer;
```

Этот запрос имеет несколько особенностей:

— в пределах каждой группы по кафедре и должности (PARTITION BY) преподаватели упорядочиваются по времени начала работы (ORDER BY);

— для каждого сотрудника в группе (каждой записи в группе) подсчитывается сумма зарплаты его и его предшественников в группе (ROWS BETWEEN).

Следующий запрос определяет для каждого сотрудника накопительные затраты университета после его зачисления на работу:

```
SELECT SecondName, HireDate, Salary,
   SUM(Salary) OVER(ORDER BY HireDate
      ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) rows_sal,
   SUM(Salary) OVER(ORDER BY HireDate
      RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) range_sal
   FROM Lecturer;
```

Особенностью этого запроса является разные результаты, возвращаемые функциями RANGE и ROWS. Если несколько преподавателей зачислены на работу одновременно, сумма „по диапазону“ (RANGE) будет отличаться от суммы „по строкам“ (ROWS), потому что RANGE не отличает строки этих преподавателей и суммирует их зарплаты (то есть даёт общую сумму по подгруппе), в то время, как ROWS суммирует затраты для каждой строки отдельно.

SECONDNAME	HIREDATE	SALARY	ROWS_SAL	RANGE_SAL
Погорецкая	1980-09-01	1000	1000	1000
Востров	1982-09-01	1100	2100	3400
Юхименко	1982-09-01	1300	3400	3400
Чугунов	1985-09-01	1250	4650	4650
Малахов	1990-09-01	1500	6150	6150

В следующем запросе преподаватели сгруппированы по кафедрам, а внутри каждой группы упорядочены по возрастанию зарплаты. Для каждой строки полученной таблицы подсчитывается средняя зарплата трёх преподавателей: текущего (то есть самой строки) и двух предыдущих в пределах одной группы (кафедры):

```
SELECT DeptKod,SecondName, Job, Salary,
        ROUND(AVG(Salary) OVER(PARTITION BY DeptKod ORDER BY Salary DESC
        ROWS BETWEEN 2 PRECEDING AND CURRENT ROW )) AS Avg_3
  FROM Lecturer;
```

Результат этого запроса выглядит следующим образом:

DEPTKOD	SECONDNAME	JOB	SALARY	AVG_3
2	Малахов	Зав.каф	1500	1500
2	Чугунов	Доц.	1250	1375
2	Погорецкая	Доц.	1000	1250
3	Юхименко	Проф.	1300	1300
4	Востров	Доц.	1100	1100

Здесь можно увидеть, что у первого и двух последних строк таблицы нет никаких предыдущих строк *в пределах их групп*, поэтому их зарплата — это единственный элемент, который принимает участие в подсчёте средней зарплаты. Для второй строки уже существует один предшественник, и функция AVG подсчитывается для двух значений. И только для третьей строки первой группы (кафедра 2) окно принимает максимальный размер, и средняя зарплата вычисляется для текущего (третьего) и двух предыдущих преподавателей этой кафедры.

Ниже представлены несколько функций, предназначенных для работы непосредственно с окнами.

Функция ROW_NUMBER() возвращает номер строки в окне для:

— создания нумерации, которая отличается от порядка сортировки строк результирующего набора;

— создания „несквозной“ нумерации, то есть выделения группы строк из общего множества и нумерования их отдельно для каждой группы;

— использования одновременно нескольких способов нумерации, поскольку, фактически, нумерация не зависит от сортировки строк.

Чтобы получить преподавателей с порядковым номером следования зарплаты по возрастанию и убыванию, можно выполнить запрос:

```
SELECT   SecondName, Salary,
         ROW_NUMBER() OVER (ORDER BY Salary DESC) AS Salbacknumber,
         ROW_NUMBER () OVER (ORDER BY Salary) AS Salnumber
  FROM Lecturer;
```

с таким результатом:

SECONDNAME	SALARY	SALBACKNUMBER	SALNUMBER
Погорецкая	1000	5	1
Востров	1100	4	2
Чугунов	1250	3	3
Юхименко	1300	2	4
Малахов	1500	1	5

Чтобы получить пять преподавателей с самой большой зарплатой, можно выполнить запрос:

```
SELECT  SecondName, Salary FROM
        (SELECT     SecondName, Salary,
                    ROW_NUMBER() OVER(ORDER BY Salary DESC) AS R
          FROM Lecturer) AS T WHERE R <= 5;
```

Чтобы получить преподавателей с порядковым номером строки по возрастанию их зарплаты, которое определяется отдельно в каждой группе по кафедре и должности, можно выполнить запрос:

```
SELECT    ROW_NUMBER()
              OVER (PARTITION BY DeptKod, Job ORDER BY Salary) AS Num,
          DeptKod, Job, SecondName, Salary
    FROM Lecturer;
```

NUM	DEPTKOD	JOB	SECONDNAME	SALARY
1	2	Доц.	Погорецкая	1000
2	2	Доц.	Чугунов	1250
1	2	Зав. каф.	Малахов	1500
1	3	Проф.	Юхименко	1300
1	4	Доц.	Востров	1100

Получить *по каждой кафедре* двух преподавателей с самой большой зарплатой позволит следующий запрос:

```
SELECT    DeptKod, Job, SecondName, Salary FROM
          (SELECT   DeptKod, Job, SecondName, Salary, ROW_NUMBER()
                    OVER(PARTITION BY DeptKod ORDER BY Salary DESC) AS R
            FROM Lecturer) AS T
          WHERE R <= 2;
```

Функция RANK() возвращает позицию каждой строки в секции результирующего набора, которая вычисляется как единица плюс количество рангов, находящихся до этой строки.

Функция DENSE_RANK() возвращает позицию каждой строки в секции результирующего набора без промежутков в ранжировании, которая вычисляется как количество разных значений рангов, предшествующих строке, увеличенное на единицу, при этом ранг увеличивается при каждом изменении значений выражений, входящих в конструкцию ORDER BY, а строки с одинаковыми значениями получают тот же ранг.

В следующем запросе приведены примеры использования функций:

```
SELECT    ROW_NUMBER () OVER (ORDER BY Salary) AS Salnumber,
          RANK() OVER (ORDER BY Salary) AS Salrank,
          DENSE_RANK() OVER (ORDER BY Salary) AS Saldenserank,
          DeptKod, Job, SecondName, Salary
    FROM Lecturer;
```

Чтобы выбрать преподавателей с низкой зарплатой, уровень которой находится на четвёртом месте, можно выполнить запрос:

```
SELECT    SecondName, Salary FROM
    (SELECT SecondName, Salary, DENSE_RANK() OVER(ORDER BY Salary) AS R
          FROM Lecturer) T
    WHERE R = 4;
```

Функция PERCENT_RANK() возвращает относительную позицию строки, которая вычисляется по формуле $(rank-1)/(total\_rows-1)$, где *total_rows* — общее количество строк в таблице.

Функция CUME_DIST() возвращает относительную позицию строки в группе, которая вычисляется по формуле $(row\_num)/(total\_rows)$, где *row_num* — количество строк, предшествующих текущей строке. Например, в группе из трёх строк возвращаются значения 1/3, 2/3 и 3/3.

Функция NTILE(*количество_групп*) возвращает номер группы, которой принадлежит строка. При этом строки распределяются в упорядоченные группы заданной мощности.

Чтобы выбрать преподавателей с их распределением на пять групп, можно выполнить запрос:

```
SELECT    RANK() OVER (ORDER BY Salary) AS Salrank,
          NTILE(5) OVER (ORDER BY Salary) AS Salntile, SecondName, Salary
    FROM Lecturer;
```

Функция LAG(*атрибут*, [*сдвиг*], [*значение_по_умолчанию*]) возвращает значение атрибута строки, предшествующей текущей строке в группе со сдвигом в оборотную сторону (по умолчанию *сдвиг* = 1 – *ссылка на предыдущую строку*). При этом возвращается *значение_по_умолчанию*, если индекс выходит за пределы окна, а также для первой строки группы.

Функция LEAD(*атрибут*, [*сдвиг*], [*значение_по_умолчанию*]) возвращает значение атрибута строки, следующего за текущей строкой в группе со сдвигом в прямом направлении (по умолчанию *сдвиг* = 1 – *ссылка на следующую строку*). При этом возвращается *значение_по_умолчанию*, если индекс выходит за пределы окна, а также для последней строки группы.

Например, следующий запрос для каждого сотрудника возвращает его зарплату, предыдущую по размеру зарплату и разность между ними.

```
SELECT    SecondName, Job, Salary,
          LAG(CAST(Salary AS INTEGER), 1, 0)
               OVER (ORDER BY DeptKod, Salary) AS sal_prev,
          Salary - LAG(CAST(Salary AS INTEGER), 1, 0)
               OVER (ORDER BY DeptKod, Salary) AS sal_diff
    FROM Lecturer;
```

Так как в реализации *PostgreSQL* функции LAG, LEAD работают с типом данных INTEGER, в запросе использован оператор преобразования типа CAST.

Если необходимо для каждого сотрудника сравнить его текущую зарплату, предыдущую по размеру зарплату и определить разность между ними, можно выполнить следующий запрос:

```
SELECT    SecondName, Job, Salary,
          LEAD(CAST(Salary AS INTEGER), 1, 0) OVER
               (ORDER BY DeptKod, Salary) AS sal_next,
          LEAD(CAST(Salary AS INTEGER), 1, 0)
               OVER (ORDER BY DeptKod, Salary) - Salary  AS sal_diff
    FROM Lecturer;
```

Синтаксис CAST совместим со стандартом SQL; синтаксис с :: является историческим для PostgreSQL.

Функция FIRST_VALUE(*атрибут*) возвращает первое значение атрибута строки в группе;

LAST_VALUE(*атрибут*) —последнее значение атрибута строки в группе;

NTH_VALUE(*атрибут*, *сдвиг*) — первое значение атрибута строки в группе с указанным сдвигом.

В следующем запросе в каждой строке к информации о сотруднике добавляется фамилия сотрудника с максимальной и минимальной зарплатой в группе сотрудников одной кафедры.

```
SELECT SecondName, DeptKod, Job, Salary,
        FIRST_VALUE(SecondName)
            OVER (PARTITION BY DeptKod ORDER BY Salary) AS sal_min,
        LAST_VALUE(SecondName)
            OVER (PARTITION BY DeptKod ORDER BY Salary) AS sal_max
    FROM Lecturer;
```

В дальнейшем (см. разделы „Подзапросы“ и „Иерархические (рекурсивные) запросы PostgreSQL“) будут рассмотрены более сложные запросы, использующие аналитические функции.

Реализация операций соединения

В отличие от имён полей в *представлении*, обратиться к исходным столбцам по их алиасу невозможно. Этим, кстати, **алиасы столбцов** отличаются от **алиасов таблиц**. Алиас таблицы удобно использовать, например, в том случае, когда *имя таблицы* довольно длинное, но оно необходимо для *уточнения имени поля*. Из всех предыдущих примеров это неочевидно, потому что у них мы использовали единственную таблицу в каждом запросе. Для иллюстрации обращения к множеству таблиц рассмотрим реализацию ещё одной реляционной операции — **соединения.**

Задача: нужно вывести текущий рейтинг всех студентов по всем дисциплинах из таблицы *Rating*. Однако пользователя не интересует их *код* — ему необходимы *фамилии и имена*, которые описаны в таблице *Student*.

Для решения этой задачи необходимо выполнить **проекцию** на некоторое *подмножество схемы отношения*, полученного в результате **соединения** двух таблиц. При выполнении запроса (одного) к нескольким таблицам и понадобится уточнение имён атрибутов. Это уточнение осуществляется путём добавления к имени поля *префикса* в виде имени таблицы, отделённого точкой. Например: *Student.Patronymic, Rating.Mark,* Причём этот префикс добавляется к **каждому** атрибуту из проецируемого подмножества. Именно здесь и можно воспользоваться алиасами таблиц. Например (*решение задачи*):

```
SELECT    S.SecondName, S.FirstName, R.DKod, R.Mark
          FROM      Student S, Rating R
    WHERE    S.Kod = R.Kod    ORDER BY S.SecondName, R.DKod;
```

Правда, необходимо отметить, что в вопросе применения префиксов этот пример не абсолютно наглядный. И вот почему: если среди атрибутов, на которые соединение выполняет проекцию, нет одинаковых имён, то префиксы можно опустить.

```
SELECT    SecondName, FirstName, DKod, Mark
          FROM    Student S   INNER JOIN   Rating R    ON   S.Kod = R.Kod
          ORDER BY SecondName, DKod;
```

Здесь приведена конструкция (*INNER JOIN*), несколько отличная от рассмотренной в разделе „Реляционная алгебра“. Это связано с тем, что в любых диалектах *SQL* команда соединения значительно расширена. Во всех СУБД вместо „чистой“ команды JOIN используются команды:

INNER JOIN — ***внутреннее*** *соединение*

и

LEFT | RIGHT | FULL OUTER JOIN — *левое*, *правое* или *полное* ***внешнее*** *соединения*.

Для понимания разницы между этими типами соединений рассмотрим пример, в котором несколько нарушим правила формализации биусловной связи 1:1 и схему данных, приведенную в Приложениях А-В, а именно: в таблицу *DWExec* добавим кортежи, содержащие идентификаторы студентов, которые *не имеют* тем дипломных проектов. Благодаря этому в поле *DpKod* появятся значения *NULL*, а таблица *DWExec* примет следующий, не совсем корректный вид:

DWExec

SKod	DpKod
1	9
2	1
3	3
4	2
5	5
6	10
7	NULL
8	NULL

Теперь решим задачу формирования списка руководителей дипломных проектов с отображением студентов, которыми они руководят, и идентификаторов тем дипломных проектов, выполняемых этими студентами. Для решения этой задачи выполним внутреннее соединение двух приведенных таблиц:

```
SELECT    LKod, SKod, L.DpKod
      FROM      Dwlist L      INNER JOIN    DWExec E   ON    L.DpKod = E.DpKod;
```

Получим следующий результат:

LKod	SKod	DpKod
1	1	9
4	2	1
4	3	3
4	4	2
5	5	5
1	6	10

Как можно увидеть, в результирующей таблице отсутствуют преподаватели, которые не руководят дипломными проектами, и студенты, которые ещё не получили тем дипломных проектов. Для того чтобы добавить всю такую информацию надо выполнить внешнее соединение (точнее, полное внешнее соединение):

```
SELECT    LKod, L.DpKod, SKod, E.DpKod
      FROM      Dwlist L      FULL OUTER JOIN      DWExec E
                                                   ON       L.DpKod = E.DpKod;
```

Результирующая таблица будет следующая:

LKod	DpKod	SKod	DpKod
4	1	2	1
4	2	4	2
4	3	3	3
5	5	5	5

LKod	DpKod	SKod	DpKod
1	9	1	9
1	10	6	10
5	4	NULL	NULL
2	6	NULL	NULL
2	7	NULL	NULL
3	8	NULL	NULL
NULL	NULL	7	NULL
NULL	NULL	8	NULL

Из этой таблицы видно, что „дополнительные“ кортежи дополняются *NULL*-значениями в соответствующих полях, полученных из другой таблицы.

Но такой результат может быть полезным на этапе назначения студентам тем дипломных проектов.

Ведь после этого заведующему кафедрой, кроме „связи“ между руководителями дипломных проектов и студентами, нужно знать только, кто из преподавателей имеет дипломников, и какие темы ещё не выданы. Для решения этой более ограниченной задачи необходимо выполнить левое внешнее соединение, которое добавит к результату внутреннего соединения кортежи первой (расположенной до оператора JOIN) таблицы, которые „не имеют пары“:

```
SELECT    LKod, L.DpKod, SKod, E.DpKod
      FROM      Dwlist L      LEFT OUTER JOIN      DWExec E
                                                   ON      L.DpKod = E.DpKod;
```

Результат запроса следующий:

LKod	DpKod	SKod	DpKod
4	1	2	1
4	2	4	2
4	3	3	3
5	5	5	5
1	9	1	9
1	10	6	10
5	4	NULL	NULL
2	6	NULL	NULL
2	7	NULL	NULL
3	8	NULL	NULL

Деканату же наоборот нужно дополнительно знать только, кто из дипломников ещё не нагружен. Тут поможет правое внешнее соединение, которое добавит к результату внутреннего соединения „лишние“ кортежи второй таблицы, находящейся после оператора JOIN:

```
SELECT    LKod, L.DpKod, SKod, E.DpKod
      FROM      Dwlist L      RIGHT OUTER JOIN    DWExec E
                                                   ON      L.DpKod = E.DpKod;
```

Результирующая таблица будет следующая:

LKod	DpKod	SKod	DpKod
4	1	2	1
4	2	4	2
4	3	3	3
5	5	5	5
1	9	1	9
1	10	6	10
NULL	NULL	7	NULL
NULL	NULL	8	NULL

Запрос, с которого было начато знакомство с вариантами соединений, выполняет внутреннее соединение. Поэтому он не отобразит ни студентов, которые не имеют кода, ни оценки, которые выставлены неизвестно кому. Понятно, что ни одного из подобных кортежей в приведенных таблицах быть не может.

В одном запросе можно выполнять соединение и большего, чем два, числа таблиц. Например, добавим к запросу, который решает задачу отображения фамилий студентов и их рейтинга, название специальности:

```
SELECT    SpName, SecondName, FirstName, Mark
      FROM      Student S, Rating R, Speciality SP
      WHERE     S.Kod = R.Kod AND S.Spec = SP.Spec
      ORDER BY      S.Spec, SecondName;
```

И, наконец, соединение можно выполнить над одной таблицей (*соединение таблицы самой с собой*). Префиксы и алиасы таблицы для этого просто необходимы. Например, **задача**: из каждого подмножества преподавателей, имеющих одинаковую суммарную надбавку, необходимо выбрать тех, у кого надбавка за стаж превышает 100 грн. (см. Приложение В).

Сначала выполним следующий запрос:

```
SELECT    LKod,
          LongevityInc + DegreeInc + TitleInc + SpecialInc     Summary_Increment,
          LongevityInc
      FROM      SalaryIncrements
      ORDER BY    2 DESC, LKod;
```

который даст такой результат:

LKOD	SUMMARY_INCREMENT	LONGEVITYINC
1	1845	300
5	1534	260
2	990	220
4	990	200
3	876	250

Выборку преподавателей с одинаковой суммарной надбавкой можно выполнить следующим образом:

```
SELECT    SI2.LKod, SI2.LongevityInc + SI2.DegreeInc + SI2.TitleInc + SI2.SpecialInc
          Summary_Increment, SI2.LongevityInc
      FROM      SalaryIncrements SI1, SalaryIncrements SI2
```

```
WHERE   SI2.LongevityInc + SI2.DegreeInc + SI2.TitleInc + SI2.SpecialInc =
        SI1.LongevityInc + SI1.DegreeInc + SI1.TitleInc + SI1.SpecialInc
ORDER BY 2 DESC, SI2.LKod;
```

Данная инструкция фактически выполняет **соединение** *двух копий таблицы SalaryIncrements*. Остаётся добавить только к выражению WHERE *вторую часть условия*:

```
…   AND SI2. LongevityInc > 100   …
```

Однако в этом примере результирующая таблица будет включать и единичные подгруппы, потому что суммарная надбавка каждого преподавателя равняется суммарной надбавке этого же преподавателя из второй копии таблицы:

LKOD	SUMMARY_INCREMENT	LONGEVITYINC
1	1845	300
5	1534	260
2	990	220
2	990	220
4	990	200
4	990	200
3	876	250

Распознать такие кортежи можно *по совпадению первичных ключей* и, соответственно, исключить их. Для этого в выражение WHERE добавим *ещё одно условие*:

```
…   AND SI2.LKod <> SI1.LKod   …
```

Это условие обеспечит удаление из результата не только единичных подгрупп, но и лишних кортежей из подгрупп, которые содержат по две строки, потому что одна из них также является соединением кортежа самого с собой. Но если в подгруппе было три и более строки, то их число уменьшается на единицу. Соответственно остаётся по *два и больше одинаковых кортежа*. Избавиться от них можно с помощью оператора DISTINCT:

```
SELECT DISTINCT   …
```

И, наконец, *над результатом соединения копий одной таблицы* можно выполнить *соединение с другим отношением*. Например, заменив *LKod* на фамилию и имя из таблицы *Lecturer*. Окончательный вариант запроса будет выглядеть следующим образом:

```
SELECT DISTINCT   SecondName, FirstName,
                  SI2.LongevityInc + SI2.DegreeInc + SI2.TitleInc + SI2.SpecialInc
                  Summary_Increment, SI2.LongevityInc
    FROM      SalaryIncrements SI1, SalaryIncrements SI2, Lecturer L
    WHERE     SI2.LongevityInc + SI2.DegreeInc + SI2.TitleInc + SI2.SpecialInc =
              SI1.LongevityInc + SI1.DegreeInc + SI1.TitleInc + SI1.SpecialInc
              AND SI2. LongevityInc > 100
              AND SI2.LKod <> SI1.LKod
              AND L.Kod = SI2.LKod
    ORDER BY 3 DESC, SI2.LKod;
```

В результате получим:

SECOND_NAME	FIRST_NAME	SUMMARY_INCREMENT	LONGEVITYINC
Погорецкая	Валентина	990	200
Востров	Георгий	990	220

До этого запроса необходимо сделать замечание о том, что версии СУБД *PostgreSQL* до 8.3 требуют для определения алиаса столбца использования оператора AS.

Однако СУБД *PostgreSQL* даже последних версий на такой запрос возвратит ошибку из-за того, что, эта СУБД, как и некоторые другие, *при использовании оператора DISTINCT* разрешает упорядочивать результирующие таблицы *только по тем полям, на которые выполнена проекция*. Поэтому или надо избавиться от оператора DISTINCT, или добавить в проекцию поле *LKod* из второй копии таблицы *SalaryIncrements*:

```
SELECT DISTINCT  SecondName, FirstName, SI2.LKod,
                 SI2.LongevityInc + SI2.DegreeInc + SI2.TitleInc + SI2.SpecialInc
                 AS Summary_Increment, …
```

Как компенсацию за этот небольшой недостаток СУБД ***PostgreSQL*** имеет ещё более расширенные возможности относительно реализации операции соединения.

В частности, в этом диалекте соединённая таблица — это таблица, полученная из двух других *не только реальных, но и также соединённых таблиц* согласно правилам определённого типа соединения. Здесь, кроме внутреннего и внешнего соединения, существует ещё и *перекрёстное*:

… R CROSS JOIN S …

Этот тип соединения, на самом деле, это ещё одна реализация *декартового произведения*. Таким образом, выражение FROM *R* CROSS JOIN *S* эквивалентно выражению FROM *R, S* или выражению FROM *R* INNER JOIN *S* ON TRUE (см. ниже).

Ограниченные соединения в *PostgreSQL*:

… R {[INNER] | {LEFT | RIGHT | FULL} [OUTER]} JOIN S ON *булево выражение* …

… R {[INNER] | {LEFT | RIGHT | FULL} [OUTER]} JOIN S USING *список столбцов* …

… R NATURAL {[INNER] | {LEFT | RIGHT | FULL} [OUTER]} JOIN S …

Операторы INNER и OUTER необязательны во всех формах: INNER используется по умолчанию, а LEFT, RIGHT и FULL подразумевают внешнее соединение.

Условие соединения указывается в операторе ON или USING, или, неявно, оператором NATURAL. *Условие соединения* определяет, какие строки двух исходных таблиц „соответствуют“ друг другу.

ON — наиболее общий вид условия соединения: оно принимает значение логического выражения того же типа, что используется в предложении WHERE. Пары строк с *R* и *S* соединяются, если ON *выражение* является для них истинным.

USING — сокращённая нотация ON: она содержит список разделённых запятыми имён столбцов, которые в соединяемых таблицах должны быть одинаковыми, и формирует условие соединения путём сравнения каждой из этих пар столбцов. Кроме того, результатом JOIN USING будет один столбец для каждой из сравниваемых пар входных столбцов плюс все остальные столбцы каждой таблицы. Таким образом,

USING (*a*, *b*, *c*)

эквивалентно

ON (*R.a* = *S.a* AND *R.b* = *S.b* AND *R.c* = *S.c*)

за исключением того, что если используется ON, в результате будет по два столбца *a*, *b* и *c*, в то время как с USING останется только один из каждой пары.

NATURAL — короткая форма USING: она создаёт список USING, состоящий из тех и только тех имён столбцов, которые присутствуют в обеих исходных таблицах. Как и в USING, эти столбцы отображаются в выходной таблице только один раз.

И наконец, очень важной особенностью диалекта *PostgreSQL* является возможность создания „многоэтажных“ конструкций из операторов JOIN. То есть соединения любого типа могут быть объединены в цепочку или вложены друг в друга: одна или обе таблицы *R* и *S* могут быть результирующими таблицами операции соединения (*JOIN-выражениями*). Для контроля порядка соединений можно использовать круглые скобки вокруг JOIN-выражений. При отсутствии скобок, вложенное множество операторов JOIN рассматривается слева направо.

Следующий пример ранжирует студентов по их рейтингу. Для этого используются ***оконные функции в соединении***. Набор строк разбивается на фрагменты с учётом значения столбца *DKod* и сортируется по столбцу *Mark*. При этом ключевое слово ORDER BY в предложении OVER упорядочивает результат RANK, а ORDER BY в инструкции SELECT упорядочивает результирующий набор:

```
SELECT    S.SecondName, D.DName, R.Mark,
          RANK()    OVER (PARTITION BY  R.DKod
               ORDER BY    R.Mark DESC) AS "RANK"
     FROM   Student S   INNER JOIN    Rating R  ON S.Kod = R.Kod
                        INNER JOIN    Discipline D     ON R.DKod = D.DKod
     ORDER BY           S.SecondName;
```

Результат этого запроса будет следующий:

SecondName	DName	Mark	RANK
Арсирий	Исследование операций	20	4
Арсирий	Сетевые технологии	76	3
Арсирий	Технологии проектирования БД	90	1
Блажко	Экономика	97	1
Бровков	Технологии проектирования БД	82	3
Бровков	Сетевые технологии	10	4
Бровков	Исследование операций	75	2
Журан	Технологии проектирования БД	86	2
Любченко	Сетевые технологии	85	1
Любченко	Технологии проектирования БД	46	5
Любченко	Исследование операций	100	1
Микулинская	Экономическая кибернетика	30	1
Филатова	Сетевые технологии	85	1
Филатова	Технологии проектирования БД	54	4
Филатова	Исследование операций	65	3
Филиппова	Технологии проектирования БД	0	6

В следующем примере возвращается значение DENSE_RANK рейтинга студентов по разным дисциплинам, при этом аргумент ORDER BY в предложении OVER упорядочивает результат DENSE_RANK, а ORDER BY в инструкции SELECT упорядочивает результирующий набор:

```
SELECT    S.SecondName, D.DName, R.Mark,
          DENSE_RANK()    OVER (PARTITION BY  R.DKod
               ORDER BY    R.Mark DESC) AS "DENSE_RANK"
```

```
FROM   Student S   INNER JOIN   Rating R  ON S.Kod = R.Kod
                   INNER JOIN   Discipline D     ON R.DKod = D.DKod
ORDER BY S.SecondName;
```

Этот пример является, на первый взгляд, не совсем выразительным из-за того, что результирующая таблица почти не отличается от предыдущей кроме некоторых значений столбца DENSE_RANK:

SecondName	DName	Mark	DENSE_RANK
Арсирий	Исследование операций	20	4
Арсирий	Сетевые технологии	76	2
Арсирий	Технологии проектирования БД	90	1
Блажко	Экономика	97	1
Бровков	Технологии проектирования БД	82	3
Бровков	Сетевые технологии	10	3
Бровков	Исследование операций	75	2
Журан	Технологии проектирования БД	86	2
Любченко	Сетевые технологии	85	1
Любченко	Технологии проектирования БД	46	5
Любченко	Исследование операций	100	1
Микулинская	Экономическая кибернетика	30	1
Филатова	Сетевые технологии	85	1
Филатова	Технологии проектирования БД	54	4
Филатова	Исследование операций	65	3
Филиппова	Технологии проектирования БД	0	6

В следующем примере строки делятся на три группы

```
SELECT    S.SecondName, D.DName, R.Mark,
          NTILE(3)  OVER (ORDER BY    R.Mark DESC) AS "GR_RANK"
     FROM   Student S   INNER JOIN   Rating R  ON S.Kod = R.Kod
                        INNER JOIN   Discipline D     ON R.DKod = D.DKod;
```

в результате чего получим такую таблицу:

SecondName	DName	Mark	GR_RANK
Любченко	Исследование операций	100	1
Блажко	Экономика	97	1
Арсирий	Технологии проектирования БД	90	1
Журан	Технологии проектирования БД	86	1
Любченко	Сетевые технологии	85	1
Филатова	Сетевые технологии	85	1
Бровков	Технологии проектирования БД	82	2
Арсирий	Сетевые технологии	76	2
Бровков	Исследование операций	75	2
Филатова	Исследование операций	65	2
Филатова	Технологии проектирования БД	54	2
Любченко	Технологии проектирования БД	46	3
Микулинская	Экономическая кибернетика	30	3
Арсирий	Исследование операций	20	3
Бровков	Сетевые технологии	10	3
Филиппова	Технологии проектирования БД	0	3

Изменяя предыдущий пример запроса, добавим аргумент PARTITION BY. Строки сначала делятся по атрибуту *DKod*, а потом делятся на три группы для каждого значения *Mark*. При этом ORDER BY в предложении OVER упорядочивает результат NTILE, а ORDER BY в инструкции SELECT — результирующий набор:

```
SELECT    S.SecondName, D.DName, R.Mark,
          NTILE(3)  OVER (PARTITION BY      R.DKod
                ORDER BY   R.Mark DESC) AS "GR_RANK" FROM
     Student S  INNER JOIN     Rating R    ON S.Kod = R.Kod
                INNER JOIN     Discipline D   ON R.DKod = D.DKod;
```

Это, в свою очередь, создаст следующие изменения в результирующей таблице:

SecondName	DName	Mark	GR_RANK
Арсирий	Технологии проектирования БД	90	1
Журан	Технологии проектирования БД	86	1
Бровков	Технологии проектирования БД	82	2
Филатова	Технологии проектирования БД	54	2
Любченко	Технологии проектирования БД	46	3
Филиппова	Технологии проектирования БД	0	3
Любченко	Сетевые технологии	85	1
Филатова	Сетевые технологии	85	1
Арсирий	Сетевые технологии	76	2
Бровков	Сетевые технологии	10	3
Микулинская	Экономическая кибернетика	30	1
Блажко	Экономика	97	1
Любченко	Исследование операций	100	1
Бровков	Исследование операций	75	1
Филатова	Исследование операций	65	2
Арсирий	Исследование операций	20	3

Демонстрационный пример

Запишите *SQL*-запросы для манипулирования данными из таблиц, созданных в задаче „Демонстрационный пример“ раздела „Язык определения данных (ЯОД, DDL) SQL. Часть 1“.

П.1. Составьте *SQL*-запросы для ввода новой информации в таблицы базы данных. При этом определите и отследите последовательность заполнения таблиц для сохранения ссылочной целостности.

Решение.

Согласно схеме данных таблицы *aircompany*, *passenger* и *employee* содержат родительские ключи для таблиц *voyage* и *ticket* и не имеют внешних ключей. Поэтому они создавались первыми, и в них в первую очередь должны вноситься новые кортежи. Таблица *voyage* содержит родительский ключ для таблицы *class*, и в неё новый кортеж будет внесён следующим. Потом — в таблицу *class* с родительским ключом для таблицы *ticket*, которая и создана последней, и новые данные в неё можно будет внести только при наличии соответствующих значений в таблицах *class, passenger* и *employee*.

```
INSERT INTO    aircompany
       VALUES (NEXTVAL('s_aircompany'), 'Lufthansa', 'Германия');
```

Для этого примера можно дальше заполнить полностью каждую из таблиц по отмеченной очереди, а можно, как в реальной системе: в соответствующую таблицу(-цы) внести кортеж(и) с родительским(и) ключом(-ами), а потом — все кортежи с зависимыми от него(-их) внешними ключами.

```
INSERT INTO    voyage
      VALUES (NEXTVAL('s_voyage'), 'RK 4578', 'Одесса – Нью-Йорк',
                  '2012-08-09', '12:30:00', '2012-08-10', '03:40:00', 'Boeing 747', 1);

INSERT INTO    class      VALUES (NEXTVAL('s_class'), 'бизнес', 1, 1000.00, 50);

INSERT INTO    passenger
      VALUES (NEXTVAL('s_passenger'), 'Чугунов А.А.', 'м', 'КН 123456');

INSERT INTO    employee
      VALUES (NEXTVAL('s_employee'), 'Арсирий Е.А.', 'кассир', '1966-03-06',
                  'КМ 555555', 56788765349876, ' г. Одесса, ул. Голубая, бы. 6, кв. 35', 2345678);

INSERT INTO    ticket     VALUES (NEXTVAL('s_ticket'), 4, 6, 10, 2, 'покупка');
```

Проконтролировать введенные кортежи можно с помощью соответствующих команд

SELECT * FROM *таблица*;

Например.

SELECT * FROM aircompany;

id_aircompany	name	country
1	Lufthansa	Германия
2	Аеросвіт	Украина
3	Трансаэро	Украина
4	Дніпроавіа	Украина
5	Turkish Airlines	Турция

SELECT * FROM class;

id_class	Name	voyage	price	quantity_place
1	Бизнес	1	1000.00	50
2	Эконом улучшенный	1	800.00	50
3	Эконом	1	500.00	100
4	Бизнес	2	1200.00	50
5	Эконом улучшенный	2	600.00	50
6	Эконом	2	300.00	100
7	бизнес	3	1500.00	50
8	Эконом улучшенный	3	1400.00	50
9	эконом	3	1100.00	100
10	бизнес	4	2000.00	50
11	эконом	4	1800.00	50
12	эконом	4	1500.00	100

SELECT * FROM passenger;

id_passenger	full_name	sex	passport
1	Малахов Е.В.	м	КМ 123456
2	Чугунов А.А.	м	КН 123456
3	Алёхин А.Б.	м	КМ 654321
4	Филиппова С.В.	ж	КЕ 123456
5	Журан Е.А.	ж	КМ 234567
6	Лингур Л.М.	ж	КМ 345678
7	Андриенко В.М.	ж	КМ 456789

SELECT * FROM voyage;

id_ voyage	number	destination	date_ departure	time_ departure	date_ arrival	time_ arrival	type_ aircraft	aircom-pany
1	RK 4578	Одесса – Нью-Йорк	2012-08-09	12:30:00	2012-08-10	03:40:00	Boeing 747	1
2	RK 3467	Одесса – Лондон	2009-06-11	18:40:00	2009-06-12	02:50:00	A310	1
3	RK 3478	Одесса – Бухарест	2009-03-19	17:40:00	2009-03-19	17:20:00	АН 158	2
4	RK 2398	Одесса – Пекин	2009-05-05	12:10:00	2009-05-06	15:50:00	Boeing 747	2
5	RK 3490	Одесса – Рим	2009-06-12	18:45:00	2009-06-13	16:20:00	ИЛ 76	2

SELECT * FROM employee;

id_ employee	full_name	posi-tion	date_birth	passport	identification_ code	adress	telefon
1	Шевченко Г.В.	Кассир	1956-10-04	КМ 777777	23458765349876	г. Одесса, ул. Незнайкіна, бы. 56, кв. 45	1234567
2	Арсирий Е.А.	Кассир	1966-03-06	КМ 555555	56788765349876	г. Одесса, ул. Голубая, бы. 6, кв. 35	2345678
3	Погорецкая В.Я.	Кассир	1974-09-12	КМ 444111	23458456349876	г. Одесса, пл. Розовая, бы. 2, кв. 32	3456789
4	Ивченко И.Ю.	Кассир	1984-05-10	КМ 222333	85474776534987	г. Одесса, ул. Белая, б. 23, кв. 4	7654321

SELECT * FROM ticket;

id_ticket	passenger	class	place	employee	operation
1	4	6	10	2	покупка
2	1	5	12	2	бронь
3	3	4	11	4	покупка
4	5	2	14	4	покупка
5	2	2	21	4	покупка
6	7	2	6	4	покупка
7	6	2	4	4	бронь

П.2. Выберите направления полётов определённого типа самолёта (по выбору студента).

Решение.

SELECT destination FROM voyage WHERE type_aircraft=‘Boeing 747’;

Результирующая таблица.

destination
Одесса – Нью Йорк
Одесса – Пекин

П.3. Создайте представление (*view*), содержащее информацию о направлениях рейсов, количество и общую сумму проданных билетов на каждое направление. Результат отсортируйте по направлению рейса.

Решение.

```
CREATE VIEW sale AS
    SELECT destination, COUNT(place), SUM(C.price) FROM ticket T, voyage V, class C
    WHERE T.class=C.id_class AND C.voyage=V.id_voyage
    GROUP BY destination ORDER BY destination;
```

Результирующая таблица.

destination	count	SUM
Одесса – Лондон	3	2100.00
Одесса – Нью Йорк	4	3200.00

П.4. Выберите из базы данных ФИО пассажиров и рейсы, на которые забронированы билеты.

Решение.

```
SELECT P.full_name, V.number FROM passenger P, voyage V, ticket T, class C
    WHERE P.id_passenger=T.passenger AND C.id_class=T.class AND
                                C.voyage=V.id_voyage AND T.operation='бронь';
```

Результирующая таблица.

full_name	number
Малахов Е.В.	RK 3467
Лингур Л.М.	RK 4578

П.5. Выберите из базы данных пункт назначения рейса, название класса и количество мест проданных на каждое из направлений. Данные отсортируйте по пунктам назначения.

Решение.

```
SELECT V.destination, C.name, COUNT (T.place) FROM ticket T, voyage V, class C
    WHERE T.class=C.id_class AND C.voyage=V.id_voyage AND T.class=C.id_class
    GROUP BY V.destination, C.name;
```

Результирующая таблица.

Destination	name	count
Одесса – Лондон	Эконом	1
Одесса – Лондон	Эконом улучшенный	1
Одесса – Лондон	Бизнес	1
Одесса – Нью-Йорк	Эконом улучшенный	4

Подзапросы

Задача: получить список студентов, которые в 1996 году учились на третьем курсе специальности „Экономическая кибернетика“. Шифр неизвестен, известно, что он есть и в таблице *Student*, и в таблице *Speciality*.

Вопрос: Какие шаги необходимо выполнить?

Ответ:

1) найти значение поля *Spec* в таблице *Speciality* и
2) использовать его для выборки информации из таблицы *Student* (см. Приложение А).

SQL позволяет объединить эти два запроса в один. Причём, если *второй запрос возвращает **единственное** значение*, то он включается в *операцию сравнения* выражения WHERE:

```
SELECT   SecondName, FirstName        FROM     Student
    WHERE   GNum  BETWEEN 941 AND 950
            AND   Spec = (SELECT   Spec    FROM    Speciality
                           WHERE   SpName = ‘Экономическая кибернетика’);
```

Аналогично запросам, в подзапросах можно использовать агрегатные функции. Например, **задача**: определить студентов, имеющих рейтинг по первой дисциплине выше среднего. **Решение**:

```
SELECT   Kod, Mark    FROM     Rating
    WHERE   DKod = 1 AND
            Mark >= (SELECT  AVG(Mark)   FROM  Rating  WHERE   DKod = 1);
```

Сначала в подзапросе вычисляется средний балл среди всех студентов, а далее это значение используется в условии запроса верхнего уровня для этой же таблицы.

Как было отмечено, в предыдущих примерах результатом выполнения подзапроса было единственное значение. Однако возможны ситуации, когда подзапрос возвращает множество значений атрибута. В этом случае на помощь приходит оператор IN (*единственный* со *специальных*, который допустим в подзапросах). Например, получить список студентов, как в первом примере, но тех, что учатся на всех специальностях направления „Экономика“:

```
… AND  Kod   IN(SELECT    Kod    FROM    Student
                  WHERE  Spec   IN(SELECT   Spec FROM  Speciality
                                    WHERE  ScDirect = 'Экономика'));
```

Вложенность такого рода подзапросов может быть сколь угодно большой. Например, **задача**: модифицировать второй пример так, чтобы вместо кодов выводились фамилия, имя студента и название дисциплины. **Решение**:

```
SELECT   SecondName, FirstName, DName, Mark
  FROM     Student S, Rating R, Discipline D
  WHERE   S.Kod = R.Kod AND R.DKod=D.DKod AND R. DKod = 1 AND R.Kod IN
           (SELECT  Kod  FROM  Rating  WHERE   DKod = 1 AND Mark >=
                    (SELECT  AVG(Mark)  FROM     Rating
                                        WHERE   DKod = 1));
```

Выполнение таких запросов осуществляется снизу вверх: сначала подзапрос нижнего уровня, его результат используется в подзапросе следующего уровня, а далее результат этого подзапроса — в базовом запросе.

Более сложным вариантом подзапросов являются так называемые **связанные подзапросы**. Рассмотрим **задачу**: вывести на экран полные номера групп, в которых есть студенты, имеющие, например, нулевой рейтинг по первой дисциплине.

Вариантом решения может быть следующий:

```
SELECT    Spec, GNum   FROM       Student S, Rating R
      WHERE     R.Kod = S.Kod AND DKod = 1 AND Mark = 0;
```

Однако в результате появятся одинаковые кортежи: сколько студентов с нулевым рейтингом — столько раз будет повторено название специальности. Это, во-первых. А во-вторых, выполняется соединение трёх таблиц, что является нетривиальной задачей для сервера.

Другим подходом к решению поставленной задачи может быть реализация алгоритма:

1. Получить кортеж таблицы *Student*.
2. Используя значение его поля *Kod*, выбрать из таблицы *Rating* все кортежи с таким же значением поля *Kod*.
3. Если среди выбранных кортежей есть кортежи с нулевым значением поля *Mark* и единичным значением поля *DKod*, то текущий кортеж таблицы *Student* включить в результирующее отношение.
4. Повторить алгоритм для всех других кортежей таблицы *Student*.

Этот алгоритм легко реализуется с помощью подзапросов (точнее связанных подзапросов) и его общий вид такой:

1. Выбрать строку из таблицы, указанной во внешнем запросе. Это, так называемая, **текущая строка-кандидат** (в смысле *на включение в результат запроса*).
2. Выполнить подзапрос. Причём в условии выборки кортежей из таблицы, указанной в подзапросе, используются значения строки-кандидата.
3. Проверить истинность условия выборки кортежей во внешнем запросе, используя результат выполнения подзапроса.
4. Если условие выборки внешнего запроса истинно, то строка-кандидат включается в результат этого запроса.

Для нашего примера такой запрос будет выглядеть следующим образом:

```
SELECT    Spec, GNum       FROM       Student S
      WHERE     0 IN
            (SELECT       Mark      FROM   Rating R
                  WHERE     R.Kod = S.Kod AND DKod = 1);
```

Связанность подзапросов отнюдь не ограничивает глубину их вложенности. Например, если бы были нужны не номера групп, а названия специальностей, на которых есть студенты, имеющие нулевой рейтинг по первой дисциплине, то запрос, ориентированный на решение поставленной задачи, выглядел бы таким образом:

```
SELECT    SpName     FROM       Speciality Sp
      WHERE     Spec IN
            (SELECT       Spec      FROM   Student S
                  WHERE     0 IN
                        (SELECT   Mark     FROM       Rating R
                              WHERE     R.Kod = S.Kod AND DKod = 1));
```

Префикс и, соответственно, алиас *R* не является обязательным, потому что *SQL* обращается сперва к таблице, указанной в подзапросе, и, если у неё такого поля нет, то — к таблице с внешнего подзапроса.

Подзапросы могут связывать таблицу и со своей копией. Например, **задача**: вывести список студентов, имеющих рейтинг по первой дисциплине выше среднего *по своей специальности*. **Решение**:

```
SELECT    SecondName, FirstName, Mark, S1.Spec
      FROM      Student S1, Rating R1
      WHERE     S1.Kod = R1.Kod AND DKod = 1 AND Mark >
            (SELECT      AVG(Mark)    FROM      Rating R2
                  WHERE     DKod = 1 AND Kod    IN
                        (SELECT      Kod      FROM      Student S2
                              WHERE     S2.Spec = S1.Spec));
```

Здесь в подзапросе самого нижнего уровня (связанном подзапросе) формируется множество кодов студентов, которые имеют тот же самый шифр специальности, что и студент, описанный текущей строкой-кандидатом. Дальше среди этого множества по таблице *Rating* подсчитывается средний балл. В базовом запросе этот результат сравнивается с рейтингом избранного студента снова из таблицы *Rating*. При истинности сравнения из таблицы *Student* выбирается его фамилия, имя и шифр специальности, а из таблицы *Rating* — его балл.

Для дальнейшего расширения запросов и предоставленных ими возможностей рассмотрим ещё ряд *специальных операторов условия*: EXISTS, ANY(SOME) и ALL. Мы их не рассмотрели вместе с другими условными операторами, потому что они могут использоваться только с подзапросами.

Оператор EXISTS предназначен для того, чтобы фиксировать **наличие** выходных данных в результате подзапроса, и в зависимости от этого возвращает значение „истина“ или „ложь“. Например:

```
SELECT    Kod      FROM      Rating
      WHERE     NOT (Mark < 60 OR Mark > 75)
                AND EXISTS
                      (SELECT     Kod     FROM      Rating
                            HAVING    MIN(Mark) >= 90
                            GROUP BY      Kod);
```

будет выведен список „троечников“ в том случае, если существуют „круглые отличники“.

Если удалить все условия, кроме EXISTS, то будет выведена вся таблица. Очевидно, что запрос в такой форме большей частью не имеет смысла. Более целесообразной является проверка условия существования для каждого кортежа отношения. Для этого необходимо воспользоваться связанными подзапросами. Например, для выборки преподавателей, имеющих одинаковую суммарную надбавку к заработной плате, как в одной из предыдущих задач, можно вместо соединения отношений воспользоваться подзапросом:

```
SELECT    LKod, SI1.LongevityInc + SI1.DegreeInc + SI1.TitleInc + SI1.SpecialInc
            Summary_Increment
      FROM      SalaryIncrements SI1
      WHERE     EXISTS
         (SELECT  *     FROM     SalaryIncrements SI2
            WHERE     SI2.LongevityInc + SI2.DegreeInc + SI2.TitleInc + SI2.SpecialInc =
                      SI1.LongevityInc + SI1.DegreeInc + SI1.TitleInc + SI1.SpecialInc
                           AND SI2.LKod <> SI1.LKod);
```

Впрочем, если в выходные данные добавить фамилию и имя преподавателя из таблицы *Lecturer*, то получится совместное использование соединения и подзапроса с EXISTS в одном запросе:

```
SELECT   SecondName, FirstName,
         LKod, SI1.LongevityInc + SI1.DegreeInc + SI1.TitleInc + SI1.SpecialInc
         Summary_Increment
    FROM     Lecturer L, SalaryIncrements SI1
    WHERE    EXISTS
      (SELECT  *   FROM    SalaryIncrements SI2
         WHERE   SI2.LongevityInc + SI2.DegreeInc + SI2.TitleInc + SI2.SpecialInc =
                        SI1.LongevityInc + SI1.DegreeInc + SI1.TitleInc +
         SI1.SpecialInc
                        AND SI2.LKod <> SI1.LKod)
              AND L.LKod = SI1.LKod;
```

Задача. Модифицировать пример вывода названий специальностей, на которых есть студенты с нулевым рейтингом по первой дисциплине.

Решение:

```
SELECT   SpName   FROM     Speciality Sp
    WHERE   EXISTS
            (SELECT   *   FROM     Student S
                 WHERE   S.Spec = Sp.Spec
                         AND EXISTS
                              (SELECT  *  FROM     Rating
                                   WHERE   Kod = S.Kod
                                           AND DKod = 1
                                           AND Mark = 0));
```

Фактически использование оператора EXISTS эквивалентно подсчёту числа строк в результате подзапроса и сравнению с 0 или 1: ‘> 0’ для EXISTS и ‘<1’ — для NOT EXISTS.

Ещё более простым будет решение этой задачи, если воспользоваться оператором ANY или SOME. Кстати, это один и тот же оператор, просто имеющий две мнемоники. Например:

```
SELECT   SpName   FROM  Speciality Sp
    WHERE   Spec = ANY
            (SELECT  Spec   FROM    Student
                 WHERE     Kod = ANY
                           (SELECT  Kod  FROM  Rating
                                WHERE   DKod = 1 AND Mark = 0));
```

Этот пример наглядно демонстрирует, что в отличие от EXISTS, оператор ANY *не требует связывания подзапросов*. Но, кроме этого, пример показателен и интересен двумя следующими моментами. Оператор ANY берет все кортежи, полученные в подзапросе, и ***для каждого*** из них сравнивает значение поля, указанного в подзапросе, *с единственным значением* поля из внешнего запроса. Единственным потому, что используется значение оператора **текущего** кортежа внешнего запроса. Если ***хотя бы одно*** значение из подзапроса удовлетворяет условию проверки, то ANY возвращает значение „истина“, а строка таблицы из внешнего запроса включается в его результат.

В **данном** примере условием является равенство, поэтому именно в **этом** случае действие ANY полностью совпадает с действием IN. В общем случае ANY отличается от IN тем, что может использоваться с любыми операторами сравнения, а IN соответствует только равенству (или неравенству).

Однако здесь необходимо быть внимательным, потому что ANY обозначает „*любое* значение из выбранных в подзапросе“. Поэтому условие „<ANY“ фактически будет соответствовать условию *меньше максимального* из выбранных, и наоборот, „>ANY“ соответствует условию *больше минимального* из выбранных.

Вторая особенность приведенного примера заключается в том, что замена в нем EXISTS на ANY вполне корректна. Связано это с тем, что в использованных таблицах нет *NULL*-значений. Если бы это было не так и необходимо было бы вывести *названия* специальностей, у студентов которых нет нулевого рейтинга, то ANY и EXISTS реагировали бы на это по-разному: оператор NOT EXISTS возвратит название специальности *независимо* от того, выполняется ли требование задачи или же просто в поле Spec какой-либо из таблиц стоит *NULL*-значение:

```
SELECT   SpName   FROM     Speciality Sp
     WHERE   NOT EXISTS
             (SELECT   *   FROM     Student S
                  WHERE   S.Spec = Sp.Spec …
```

Дело в том, что результатом EXISTS может быть только значения „*истина“* и „*ложь“*, а для операторов сравнения, в которых принимает участие ANY, для *NULL*-значений генерируется значение UNKNOWN, действующее также как FALSE.

Применение ещё одного оператора — ALL — означает, что условие внешнего запроса должно удовлетворять *каждый* кортеж из подзапроса. Соответственно, „>ALL“ или „<ALL“ будет означать *больше максимального* или *меньшее минимального* из значений, выбранных в подзапросе. „<>ALL“ — соответствует *отсутствию значения во множестве, сформированном подзапросом*. А „=ALL“ отслеживает случай, когда значения поля во *всех* кортежах подзапроса *равны*. **Например**: вывести список группы при условии, что все студенты группы имеют одинаковый рейтинг по первой дисциплине. **Решение:**

```
SELECT SecondName, FirstName, Spec, GNum, Mark FROM Student S, Rating R
     WHERE   S.Kod = R.Kod AND DKod = 1 AND Mark = ALL
          (SELECT     Mark     FROM Rating    WHERE DKod = 1 AND Kod IN
          (SELECT     Kod      FROM     Student
               WHERE   Spec = S.Spec AND GNum = S.GNum));
```

Запросы и подзапросы, основанные на команде SELECT, можно использовать и в других командах *SQL*. Например, для того, чтобы извлечь данные из одной таблицы и разместить их в другой можно воспользоваться инструкцией:

```
INSERT INTO   OI
          SELECT     *   FROM          Student
               WHERE   Spec = ‘ОИ’;
```

Запрос выберет всех студентов этой специальности и внесёт их в новую таблицу. *Если схемы отношений совпадают не полностью*, то можно воспользоваться *вместо* ‘*’ *проекцией*.

Аналогично можно сформировать таблицу, которая хранит значение среднего рейтинга каждого студента. Пример

```
INSERT INTO    AveRat1
            SELECT      Kod, AVG(Mark)      FROM        Rating
                GROUP BY    Kod;
```

Если вместо среднего рейтинга студента нужен средний рейтинг специальности, уже необходимо воспользоваться подзапросом. При этом подзапрос не должен ссылаться (в случае связанных подзапросов) на указанную в команде INSERT изменяемую таблицу. Например:

```
INSERT INTO    AveRat2
            SELECT    SpName, AVG(Mark)
                        FROM       Rating R, Speciality
                WHERE     Spec = ANY
                                (SELECT     Spec    FROM     Student
                                        WHERE       Kod = R.Kod)
                GROUP BY SpName;
```

В отличие от INSERT в командах DELETE и UPDATE можно ссылаться на таблицу, указанную в самом внешнем запросе, то есть в самих этих командах. Например:

```
DELETE FROM     Student S
      WHERE       0 =
            (SELECT     SUM(Mark)     FROM       Rating
                WHERE       Kod = S.Kod);
```

Запрос удаляет полных двоечников. Для этого в подзапросе для каждой строки-кандидата таблицы *Student* суммируются соответствующие кортежи таблицы *Rating*. Если студент имеет нулевые баллы по всем дисциплинам, то в базовом запросе информация о нем удаляется, **но** только из таблицы *Student* и только в таких СУБД, как *Firebird*, из-за того, что эти СУБД не поддерживают ограничений ON UPDATE и ON DELETE при создании таблиц. СУБД *PostgreSQL, Oracle* и прочие, которые такие ограничения поддерживают, вообще отвергнут этот запрос из-за того, что на *каждый* кортеж таблицы *Student* существуют ссылки из таблицы *Rating*. Как сохранить ссылочную целостность данных и этим же запросом удалить соответствующие данные из таблицы *Rating* будет рассмотрено в разделе „ЯОД (DDL) SQL. Часть 2.Триггеры“.

Большинство СУБД поддерживают и такую конструкцию:

```
DELETE FROM     Rating
      WHERE       DKod = 1 AND Mark <
            (SELECT     AVG(Mark)     FROM   Rating   WHERE    DKod = 1);
```

а некоторые и такую:

```
DELETE FROM     Rating
      WHERE    DKod = 1 AND Mark <
              (SELECT     AVG(Mark)   FROM       Rating       WHERE    DKod = 1)
      AND         MDate =
              (SELECT     MAX(MDate)     FROM       Rating
                    WHERE    DKod = 1
                    HAVING   MAX(MDate) <> MIN(MDate));
```

или, даже, такую:

```
DELETE FROM     RATING
      WHERE    DKod = 1 AND Mark <
            (SELECT    AVG(Mark)   FROM      Rating      WHERE    DKod = 1)
      AND        MDate =
            (SELECT    MAX(MDate)    FROM  Rating
                WHERE    DKod = 1 AND NOT MDate = ALL
                     (SELECT  MDate  FROM  Rating  WHERE    DKod = 1));
```

Два последних запроса, используя разные средства, решают задачу удаления кортежей, где в первом условии оценка по первой дисциплине ниже средней, а во втором — максимальна дата получения оценки по первой дисциплины, но эта дата не единственная (по этой же дисциплине). То есть если все студенты получили оценку в один день, то удалять их не надо.

Аналогично формируются подзапросы для команды обновления. **Например**, разделить каждую группу, в которой больше 31 человека на 2. Причём в одну из новых групп ввести студентов с рейтингом по второй дисциплины выше среднего.

```
UPDATE  Student     SET     GNum = GNum + 1
   WHERE    Kod IN
        (SELECT    Kod    FROM    Student S1
            WHERE  31 < (SELECT   COUNT(*)    FROM    Student S2
                              WHERE    S1.Spec = S2.Spec AND S1.GNum = S2.GNum)
                    AND Kod IN
                         (SELECT  Kod    FROM    Rating
                            WHERE    DKod = 1 AND Mark >
                                 (SELECT    AVG(Mark)        FROM    Rating
                                      WHERE    DKod = 1 AND Kod IN
                                           (SELECT    Kod  FROM    Student S3
                                                WHERE    S3.Spec = S1.Spec
                                                     AND S3.GNum =
                                                     S1.GNum))));
```

Замечание. Этот запрос будет корректен, если на *каждом* курсе существует по *одной* группе.

В этом запросе фактически две группы подзапросов (операторы условия, которые их содержат, выделены жирным шрифтом). Первая группа — это двухуровневый подзапрос, который в подзапросе нижнего уровня по таблице *Student* подсчитывает количество студентов в группе, полный номер которой совпадает со значением текущей строки-кандидата. Если это количество больше 31, значение кода студента добавляется в результат подзапроса верхнего уровня. Вторая группа — это трёхуровневый подзапрос. Третий (нижний) уровень по таблице *Student* формирует список кодов студентов группы, к которой принадлежит студент, описанный текущей строкой-кандидатом. Подзапрос следующего уровня высчитывает по таблице *Rating* средний балл по второй дисциплине среди отобранных студентов. Подзапрос верхнего уровня добавляет к своему результату код студента, если его рейтинг по второй дисциплине выше подсчитанного среднего. Если результаты обеих групп подзапросов совпадают, то есть оба выделенных условных оператора генерируют значение „истина“, базовый запрос выполняет инкремент номера группы (значение поля *GNum*) в текущей строке-кандидате, то есть переводит соответствующего студента в новую группу.

Диалект *SQL* СУБД ***PostgreSQL*** имеет ряд расширений для вложенных подзапросов, что даёт возможность повысить эффективность их применения.

Например, часто необходимо создавать запросы, использующие условия с атрибутами таблиц, которые прошли дополнительную обработку, например, с функциями агрегирования. Дополнительная обработка выполняется в виде вложенных запросов. Вложенные запросы можно создавать с использованием операторов EXISTS и IN, как было показано раньше, или в виде ссылки на временную таблицу. Ссылка на временную таблицу реализуются двумя способами: в FROM-предложении и с использованием оператора WITH.

Использование оператора WITH позволяет разбить сложный запрос на множество подзапросов в удобной для восприятия человеком форме.

Оператор WITH использует следующую конструкцию:

```
WITH имя_запроса [(список_атрибутов_из_запроса)] AS (описание_запроса)
    Описание_основного_запроса
```

Примером использования может быть решение представленной ранее задачи о выводе списка студентов, имеющих рейтинг выше среднего по своей специальности:

```
WITH    Avg_Mark AS
        (SELECT   S1.Spec, AVG(Mark) AS Avg_Mark
            FROM      Student S1, Rating R1
            WHERE     S1.Kod = R1.Kod
            GROUP BY      S1.Spec)
SELECT    SecondName, FirstName, Mark, S.Spec
    FROM      Student S, Rating R, Avg_Mark A
    WHERE     S.Kod = R.Kod AND S.Spec = A.Spec AND R.Mark > A.Avg_Mark;
```

Оператор WITH позволяет более эффективно ***использовать в подзапросах оконные функции***. Например, чтобы получить преподавателей, их зарплату, процент от максимальной зарплаты, процент от минимальной зарплаты на одной и той же кафедре, можно создать следующий запрос:

```
WITH t AS
    (SELECT   DeptKod, SecondName, Salary,
              FIRST_VALUE(Salary)     OVER(PARTITION BY DeptKod
                                ORDER BY Salary DESC
                                ROWS BETWEEN UNBOUNDED PRECEDING
                                AND UNBOUNDED FOLLOWING) AS Max_Sal,
              LAST_VALUE(Salary)      OVER (PARTITION BY DeptKod
                                ORDER BY Salary DESC
                                ROWS BETWEEN UNBOUNDED PRECEDING
                                AND UNBOUNDED FOLLOWING) AS Min_Sal
        FROM Lecturer)

SELECT    DeptKod, SecondName, Salary,
          Max_Sal, ROUND(100*Max_Sal/Salary) "Max_Sal_%",
          Min_Sal, ROUND(100*Min_Sal/Salary) "Min_Sal_%"
    FROM t ORDER BY DeptKod;
```

Объединение

Запросы и подзапросы, основанные на команде SELECT, используются при реализации ещё одной операции реляционной алгебры — **объединения**. Естественно, атрибуты, на которые выполняют проекцию запросы, принимающие участие в объединении, должны удовлетворять требованию этой операции, то есть быть совместимыми по типам. Например, вывести на экран коды студентов и дисциплин, по которым они имеют максимальный и минимальный рейтинги:

```
SELECT    Kod, DKod, Mark, 'Максимальный␣рейтинг'   FROM   Rating
      WHERE    Mark = (SELECT    MAX(Mark)   FROM   Rating)
UNION
SELECT    Kod, DKod, Mark, 'Минимальный␣␣рейтинг' FROM   Rating
      WHERE    Mark = (SELECT    MIN(Mark)    FROM   Rating);
```

Два пробела (символ ‘␣’) добавлены для выравнивания длины константы, в противном случае результаты запросов будут несовместимы.

И, наконец, команда UNION и оператор WITH являются основой ещё одного расширения диалекта ***PostgreSQL*** — это

Иерархические (рекурсивные) запросы **PostgreSQL**

Добавление к оператору WITH необязательного оператора RECURSIVE позволяет запросу обращаться к собственным результирующим данным, обеспечивая их рекурсивную обработку.

Алгоритм рекурсивного запроса должен включать две части: первая часть — это *ядро*, которое обычно возвращает одну строку с исходной точкой иерархии или части иерархии, а вторая часть — *рекурсивная*, которая будет связываться с временной таблицей, объявленной в операторе WITH. Обе части объединяются оператором UNION или UNION ALL. Строго говоря, процесс формирования ответа на запрос скорее итерационный, чем рекурсивный, но понятие RECURSIVE избрано комитетом со стандартизации *SQL*.

Рассмотрим пример запроса, который демонстрирует подсчёт суммы чисел от 1 до 100:

```
WITH RECURSIVE    t(n)    AS    (SELECT       1
                                      UNION ALL
                                   SELECT      n+1   FROM   t    WHERE     n < 100)
      SELECT sum(n) FROM t;
```

В этом примере рабочая таблица содержит лишь одну строку на каждом шаге.

Чтобы получить первые 10 чисел Фибоначчи, можно выполнить рекурсивный запрос:

```
WITH RECURSIVE    t(n, fn, "fn-1", "fn-2") AS
            (SELECT   1, 1, 1, 0
                  UNION
             SELECT   n+1, "fn-1"+"fn-2", "fn-1"+"fn-2", "fn-1"
                  FROM   t    WHERE     n <= 10)
      SELECT    n, fn  FROM   t;
```

Запрос получения названий подразделений Университета (институтов, кафедр) с учётом их иерархии представлен следующим образом:

```
WITH RECURSIVE    Rec(DeptKod, PKod, DeptName)   AS
      (SELECT   DeptKod, PKod, DeptName    FROM     Department
            WHERE    PKod     IS NULL
         UNION
      SELECT    D.DeptKod, D.PKod, D.DeptName
            FROM     Department D   INNER JOIN    Rec R
                                    ON    (D.PKod = R.DeptKod))
SELECT DeptKod, PKod, DeptName FROM Rec;
```

В этом примере просмотр иерархии подразделений начинается с подразделения, в котором отсутствует вышестоящее подразделение, то есть атрибут *PKod* не определён.

Для вывода на экран названий подразделений в удобной форме с иерархической табуляцией предыдущий запрос модифицирован следующим образом:

```
WITH RECURSIVE    Rec(DeptKod, PKod, DeptName, Path, Level) AS
   (SELECT DeptKod, PKod, cast(DeptName AS VARCHAR) AS
               DeptName, cast(DeptKod AS VARCHAR) AS Path,1
         FROM Department WHERE PKod IS NULL
      UNION
   SELECT D.DeptKod, D.PKod, lpad(' ', 3*level) || D.DeptName,
               Path || cast(D.DeptKod AS VARCHAR), Level+1
         FROM Department D INNER JOIN Rec R ON (D.PKod = R.DeptKod))
   SELECT DeptKod, DeptName FROM Rec ORDER BY Path;
```

Благодаря разбивке запроса на части с помощью оператора WITH при реализации реляционных операций соединения (JOIN) и объединения (UNION), СУБД *PostgreSQL* позволяет строить очень сложные запросы, которые возвращают аналитическую информацию с помощью соответствующих оконных функций (см. раздел „Аналитические функции“).

Например, чтобы получить среднюю зарплату преподавателей за 12 месяцев всех лет и определить её изменения в процентах по отношению к предыдущему и последующему годам, можно выполнить запрос:

```
WITH
      m_y_sal AS (SELECT      EXTRACT(Month       FROM       HireDate) m,
            EXTRACT (Year     FROM   HireDate) AS y, Salary     FROM   Lecturer),
      m_y_sal_sum AS (SELECT       m, y, SUM(Salary) Salary     FROM   m_y_sal
                                                                GROUP BY    m, y),
      month_list AS (WITH       RECURSIVE ml(m) AS
                                    (SELECT   1
                                          UNION ALL
                                     SELECT   m+1  FROM   ml     WHERE     m <= 12)
                                SELECT m FROM ml),
      year_list AS (SELECT DISTINCT      EXTRACT (Year     FROM   HireDate) y
                              FROM       Lecturer),
      month_year_list AS (SELECT        y, m       FROM       month_list, year_list),
```

```
m_y_sal_sum_add AS (SELECT   l.y, l.m, coalesce(s.Salary, 0) Salary
                    FROM month_year_list l   LEFT JOIN   m_y_sal_sum s
                                             ON (l.m = s.m and l.y = s.y) ),
m_y_sal_sum2 AS (SELECT   m, y, SUM(Salary)
                          OVER (ORDER BY   y, m) Sum_Sal
                 FROM m_y_sal_sum_add),
m_y_sal_sum_l AS (SELECT  m, y, Sum_Sal,
                          LAG(Sum_Sal,1,Sum_Sal)
                                OVER (ORDER BY y, m) Prior_Year,
                          LEAD(Sum_Sal, 1, Sum_Sal)
                                OVER (ORDER BY y, m) Next_Year
                 FROM     m_y_sal_sum2   WHERE   m = 12)
SELECT   m, y, sum_sal, prior_year, ROUND(100*prior_year/sum_sal) "p_y_%",
         next_year, ROUND(100*next_year/sum_sal) "n_y_%"
  FROM   m_y_sal_sum_l;
```

m	y	sum_sal	prior_year	p_y_%	next_year	n_y_%
12	1980	1000	1000	100	3400	340
12	1982	3400	1000	29	4650	137
12	1985	4650	3400	73	6150	132
12	1990	6150	4650	76	6150	100

В примере запрос логически разбит на восемь подзапросов, которые используются как виртуальные таблицы в отдельных подзапросах:

m_y_sal — подзапрос сохраняет зарплату сотрудников по месяцам и годам;

m_y_sal_sum — подзапрос сохраняет сумму зарплат сотрудников по месяцам и годам;

month_list — подзапрос сохраняет множество всех номеров месяцев через рекурсивный вызов;

year_list — подзапрос сохраняет множество лет приёма сотрудников на работу;

month_year_list — подзапрос сохраняет множество месяцев по всем годам приёма сотрудников на работу.

Поддержка механизма объектно-реляционных связей в PostgreSQL

Объектно-реляционные связи между таблицами, создание которых было рассмотрено в соответствующем параграфе (см. стр. 78), накладывают определённые ограничения на манипулирование данными в таких таблицах.

В частности, внесение данных в таблицы-потомки не выполняет автоматическое внесение в таблицу-предок, но они логически видимы в ней. Так в результате выполнения запроса

```
INSERT INTO   Master
   VALUES(2, 'Микулинская', 'Мария', 'Геннадиевна', 1, 'ОИ', 1, NULL, NULL, 'Тема12');
```

строка по студенту будет видна и в таблице *Student*.

Для управления видимостью строк в таблице-предке, которые были созданы в таблице-потомке, используется оператор ONLY, предшествующем имени таблицы.

Например, для получения содержимого только таблицы-предка *Student* без затрагивания таблицы-потомка *Master* используется запрос:

```
SELECT * FROM   ONLY   Student;
```

Например, для обновления строк только таблицы *Student* без затрагивания строк таблицы *Master* используется запрос:

```
UPDATE ONLY   Student   SET SecondName = 'Глава'   WHERE   kod = 1;
```

А для обновления строк таблицы *Student* и строк таблицы *Master* используется стандартный запрос:

```
UPDATE   Student   SET SecondName = 'Глава' WHERE   Kod = 1;
```

Демонстрационный пример

Запишите *SQL*-запросы **с использованием подзапросов** для манипулирования данными из таблиц, созданных в задаче „Демонстрационный пример“ раздела „Язык определения данных (ЯОД, DDL) SQL. Часть 1“.

П.1. Выберите из базы данных направления рейсов и классы, на которые ещё не продано ни одного билета.

Решение.

```
SELECT DISTINCT V.destination, C.name FROM voyage V, class C
      WHERE C.voyage=V.id_voyage AND C.id_class NOT IN
                                                (SELECT class FROM ticket);
```

Результирующая таблица.

Destination	**name**
Одесса – Бухарест	бизнес
Одесса – Бухарест	эконом
Одесса – Бухарест	эконом улучшенный
Одесса – Нью-Йорк	бизнес
Одесса – Нью-Йорк	эконом
Одесса – Пекин	бизнес
Одесса – Пекин	эконом

П.2. Выберите из базы данных пункт назначения рейса, дату вылета и название класса салона, на которой проданы все билеты.

Решение.

```
SELECT   V.destination, V.date_departure, C.name     FROM   voyage V, class C
      WHERE V.id_voyage=C.voyage AND C.quantity_place =
            (SELECT COUNT(class) FROM ticket T
                  WHERE C.id_class=T.class GROUP BY class);
```

Результирующая таблица.

destination	**date_departure**	**name**
Одесса – Нью-Йорк	2012-08-09	эконом улучшенный

П.3. Выберите из базы данных направление рейса, класс и количество билетов, которые остались в продаже. Результат отсортируйте по направлению рейса.

Решение.

```
WITH count_ticket AS
    (SELECT class, COUNT(id_ticket) AS kolvo FROM ticket GROUP BY class),
        count_place AS
            (SELECT c.id_class, c.name, v.destination, quantity_place
                FROM class c, voyage v
                WHERE c.voyage=v.id_voyage)
    SELECT c.destination, c.name, quantity_place-kolvo AS  tail
        FROM count_place c, count_ticket K
        WHERE c.id_class=k.class ORDER BY c.destination;
```

Результирующая таблица.

destination	name	tail
Одесса – Лондон	бизнес	49
Одесса – Лондон	эконом улучшенный	49
Одесса – Лондон	эконом	99
Одесса – Нью-Йорк	эконом улучшенный	46

ЯОД (DDL) SQL. Часть 2

Теперь вернёмся к ЯОД *SQL* и рассмотрим, во-первых, ряд аспектов, связанных с влиянием команд ЯМД, а во-вторых, ещё несколько элементов ЯОД, основанных на команде SELECT и, в свою очередь, используемых в командах ЯМД.

Модифицируемые представления

Для начала рассмотрим обновление и ввод информации с помощью представлений. Мы отмечали, что любые изменения, выполненные над данными в представлении, будут немедленно отражены в базовой таблице этого представления. При этом *необходимыми условиями* модификации и вывода значений через представления являются следующие:

1. Представление *не должно содержать* функций агрегирования.
2. Представление *не должно использовать* операторы GROUP BY и HAVING в своём определении.
3. Для обеспечения возможности ввода значений представление *должно включать* все поля базовой таблицы, для которых *определённое* ограничение NOT NULL.

Например. Представление

```
CREATE VIEW     AveRat   AS
      SELECT     Kod, AVG(Mark)     FROM    Rating
            GROUP BY   Kod;
```

не является обновляемым. Почему? Для ответа см. условия модификации.

А в представлении

```
CREATE VIEW     FIO     AS
      SELECT     Kod, SecondName, FirstName, Patronymic     FROM     Student;
```

можно модифицировать ФИО, но *нельзя вводить новые* кортежи, потому что ряд полей имеют ограничение на *NULL*-значение и не имеют значений по умолчанию.

Представление же

```
CREATE VIEW     Badrat     AS
      SELECT   Kod, DKod, Mark, MDate    FROM     Rating
            WHERE    Mark = 0;
```

является обновляемым. То есть с помощью его в базовую таблицу можно и вводить, и модифицировать значение. Однако здесь возможна и такая ситуация: команда

```
INSERT INTO     Badrat     VALUES(15,5,45, '25.12.2011');
```

будет принята как корректная, и добавит кортеж в таблицу Rating. Однако же *увидеть* результат этой операции, ни, тем более, его *удалить* пользователь *не сможет*, потому что новое значение не отвечает условию выборки. Избежать такой ситуации можно путём добавления в определение представления инструкции:

```
…     WITH CHECK OPTION;
```

Эта инструкция работает по принципу „*все или ничего*“ и блокирует все команды обновления, результаты которых не могут быть отображены в представлении. Соответственно, команда INSERT из нашего примера будет *отклонена*.

Хранимые процедуры (ХП)

Хранимой процедурой называется **скомпилированная программа** произвольной длины, написанная на языке *SQL*, которая **хранится в БД** вместе с другими объектами.

Хранимые процедуры позволяют сократить количество сообщений или транзакций между клиентом и сервером. Кроме того, использование хранимых процедур имеет ещё ряд преимуществ:

1. **Универсальность**:
   — одна *хранимая процедура* может использоваться совместно многими клиентами, созданными с помощью различных языков программирования.
2. **Эффективность**:
   — уменьшение нагрузки на канал связи между клиентом и узлом-сервером, потому что обработка записей выполняется непосредственно на сервере;
   — уменьшение нагрузки на узел-сервер, потому что отдельные этапы оптимизации кода процедуры могут осуществляться не во время выполнения процедуры, а во время её регистрации в СУБД.
3. **Безопасность:**
   — пользователю можно предоставить возможность вызвать хранимую процедуру, но не управлять данными, вызываемыми этой процедурой;
   — такие процедуры могут использоваться для сокрытия от пользователя многих особенностей *конкретного строения* системы или БД, что обеспечивает большую степень независимости и целостности данных.

Существует 2 типа хранимых процедур: **процедуры выбора** и **выполняемы процедуры**. Отличаются они тем, что процедуры *выбора **обязательно** должны возвращать значение* в вызывающую программу, а *выполняемые — **могут*** и ***не** возвращать* значение.

Хранимые процедуры создаются с помощью команды

CREATE PROCEDURE *процедура*

и состоят из **заголовка** процедуры и **тела** процедуры.

Заголовок процедуры содержит:

— **имя** *хранимой процедуры*, которое должно быть *уникальным* среди имён процедур и таблиц в БД;
— **необязательный** список **входных параметров** и их типов данных, значения которых процедура получает от вызывающей программы;
— **оператор** RETURNS со списком **выходных параметров** и их типов данных, в том случае, если процедура возвращает значение источнику вызова.

Благодаря этому *хранимые процедуры* могут использоваться как табличные выражения в команде SELECT (вместо отношения).

Тело процедуры состоит из:

— необязательного списка **локальных переменных** с указанием типов данных;
— **блока инструкций**, вложенного между операторами BEGIN и END (не во всех СУБД). Большинство СУБД и стандарт *SQL* поддерживают *многоуровневое вложение* блоков.

Язык **хранимых процедур и триггеров** *SQL* содержит:

- команды ЯМД *SQL*: INSERT, UPDATE, DELETE и SELECT;
- операторы и выражения *SQL*;
- расширения *SQL*, включающие инструкции присваивания, управления потоками, регистрации событий и обработки ошибок, а именно:

— *переменная = выражение* — операция присваивания значения выражения локальной переменной, входному или выходному параметру;
— INTO *список переменных* — присваивание переменным или параметрам результата работы команды SELECT;
— SUSPEND — возвращает выходные параметры вызывающей программе и приостанавливает хранимую процедуру до тех пор, пока вызывающая программа не потребует следующий кортеж. *Не рекомендуется использовать в выполняемых процедурах*;
— FOR *SELECT-выражение* DO *составной_оператор* —
составной_оператор — инструкция или блок инструкций, который будет выполнен для *каждой строки*, возвращённой *SELECT-выражением*, где *SELECT-выражение* — обычная команда SELECT, за исключением того, что в конце команды должен *обязательно* присутствовать оператор INTO;
— IF(*условие*) THEN *составной_оператор* [ELSE *составной_оператор*]
— WHILE(*условие*) DO *составной_оператор* — цикл с предпроверкой;
— EXCEPTION *исключение* — генерирует поименованную исключительную ситуацию *исключение* — определённую пользователем ошибку, которая может быть обработана оператором WHEN;
— WHEN{*ошибка*[, *ошибка*[, …]] | ANY} DO *составной_оператор* — обработка ошибок, указанных в списке, или любых ошибок, если указано ANY. Эта инструкция, если присутствует, указывается *в конце хранимой процедуры* непосредственно перед END;
— EXIT — переход на конец *хранимой процедуры* (последний END), например, для прерывания цикла;
— EXECUTE PROCEDURE *процедура* [*параметр*[, *параметр*[, …]]]
[RETURNING_VALUES [*параметр*[, *параметр*[, …]]] — выполняет хранимую процедуру с именем *процедура*, входными параметрами, перечисленными после имени, и которая возвращает значение в параметрах, перечисленных после RETURNING_VALUES.

Входными и выходными параметрами должны быть переменные, определённые *внутри текущей* процедуры. Благодаря этой инструкции возможны *вложенные вызовы* процедур и *рекурсии*.

— /* *комментарий* */ — для комментариев.

Например, *процедура выбора*, возвращающая информацию о каждой группе: численность, минимальный, средний и максимальный рейтинг по каждой дисциплине:

```
SET TERM    ;     ^
CREATE PROCEDURE      GroupInfo
      RETURNS    (Spec    CHAR(2), GNum      INT, DName   CHAR(30),
                  NumOfStud    INT, MinRat   DEC,
                  AveRat DEC, MaxRat         DEC)
AS
BEGIN
      FOR     SELECT    S.Spec, GNum, DName, COUNT(S.Kod),
                        MIN(Mark), AVG(Mark), MAX(Mark)
                  FROM      Student S, Rating R, Discipline D
                  WHERE     S.Kod = R.Kod   AND   D.DKod=R.DKod
                  GROUP BY      S.Spec, GNum, DName
                  INTO      :Spec, :GNum, :DName, NumOfStud,
                            :MinRat, :AveRat, :MaxRat
```

```
    DO
            SUSPEND;
END ^
SET TERM    ^    ;
```

Значение команды SET TERM будет разъяснено дальше в разделе „Сценарии“. Пока что имейте в виду необходимость использования этих команд до и после команды CREATE PROCEDURE.

Как мы уже отмечали, использовать такую хранимую процедуру можно как *табличное выражение* в команде SELECT:

```
SELECT    *    FROM    GroupInfo;
```

что даст следующий результат:

Spec	GNum	DName	NumOfStud	MinRat	AveRat	MaxRat
АС	954	Экономика	1	97	97	97
ОИ	943	Сетевые технологии	2	76	81	85
ОС	102	Исследование операций	1	100	100	100
ОМ	971	Сетевые технологии	1	10	10	10
ОС	102	Технологии проектирования БД	1	46	46	46
ОС	102	Сетевые технологии	1	85	85	85
ОМ	971	Технологии проектирования БД	1	82	82	82
ОИ	943	Исследование операций	2	20	43	65
ОМ	951	Технологии проектирования БД	1	0	0	0
ОИ	943	Технологии проектирования БД	2	54	72	90
ОМ	971	Исследование операций	1	75	75	75
ОП	971	Технологии проектирования БД	1	86	86	86
ОП	970	Экономическая кибернетика	1	30	30	30

Другой пример использования *хранимой процедуры*, которой можно воспользоваться в примере деления группы:

```
SELECT    Spec, MAX(GNum)    FROM    GroupInfo    GROUP BY    Spec;
```

Этот пример демонстрирует, что над таблицей, которая возвращается хранимой процедурой, можно выполнять *все* операции выборки *SQL*, в том числе, использующие *агрегатные функции* и *группировку*.

Как было отмечено выше, в процедуру можно передавать параметры. Например, процедура выбора, которая возвращает коды студентов конкретной группы:

```
SET TERM    ;    ^
CREATE PROCEDURE    StudGroup    (Spec CHAR(2), GNum INT)
    RETURNS    (Kod INT)
AS
BEGIN
    FOR    SELECT    Kod    FROM    Student
                WHERE    Spec = :Spec AND GNum = :GNum
                INTO    :Kod
    DO
            SUSPEND;
```

```
END ^
SET TERM ^ ;
```

Воспользоваться этой хранимой процедурой можно, задав параметры *в явном виде*. Например:

```
SELECT * FROM StudGroup('ОС', 941);
```

Не менее эффективным будет вызов этой *хранимой процедуры в подзапросе*. Например, для решения предыдущей задачи:

```
SELECT S.Spec, GNum, AVG(Mark) FROM Student S, Rating R
    WHERE R.Kod IN
        (SELECT Kod FROM StudGroup (S.Spec, S.GNum))
            GROUP BY S.Spec, GNum;
```

Выполняемые процедуры, как видно из названия, должны выполнять некоторые действия. Например, *хранимая процедура*, удаляющая всех выпускников из таблицы *Student* и их рейтинг из таблицы *Rating*:

```
SET TERM ; ^
CREATE PROCEDURE RemoveDiploma (GNum INT)
AS
DECLARE VARIABLE Kod INT;
BEGIN
    FOR SELECT Kod FROM Student
            WHERE GNum = :GNum
            INTO :Kod
    DO
        BEGIN
            DELETE FROM Rating WHERE Kod = :Kod;
        END
    DELETE FROM Student WHERE GNum = :GNum;
END ^
SET TERM ^ ;
```

Потому что эта *хранимая процедура не возвращает значения* и, больше того, *уничтожает при этом кортежи*, то использовать её в команде SELECT *нельзя*. Для вызова таких процедур используется команда расширения *SQL* — EXECUTE PROCEDURE *процедура*.

```
EXECUTE PROCEDURE RemoveDiploma (961);
```

Задача.

1. Изменить процедуру для удаления всех групп *данного выпуска*.
2. **Переместить** все значения о таких студентах в специальную таблицу.

Особенности построения хранимых процедур в СУБД PostgreSQL

Язык программирования PLpg/SQL СУБД PostgreSQL

Большинство современных СУБД для программирования хранимых процедур предлагает расширение стандарта *SQL*. СУБД *Oracle* была одной из первых промышленных

СУБД, которая внедрила язык программирования *PL/SQL* не только как язык программирования процедур, но и как язык программирования приложений БД. По своим возможностям *PL/SQL* можно сравнивать с большинством алгоритмических языков третьего поколения (*ANSI C*, *Pascal*, *Basic*). СУБД *PostgreSQL* при создании языка *PLpg/SQL* позаимствовала от СУБД *Oracle* большинство языковых конструкций *PL/SQL*.

Структура подпрограмм PLpg/SQL

В отличие от хранимых процедур стандартного *SQL*, приведенных выше, и, даже, в отличие от СУБД *Oracle*, которая содержит два типа подпрограмм — ***процедуры*** (аналогичные стандартным) и ***функции***, *PLpg/SQL* поддерживает только *пользовательские* ***функции***.

Для создания такой *функции* используется следующий синтаксис:

```
CREATE OR REPLACE FUNCTION Имя_функции
    (Имя_параметра Вид_передачи_значения Тип_параметра, …)
    RETURN Тип_возвращаемого _значения
    AS $$
        PLpg/SQL программа, которая должна завершаться оператором
        RETURN Возвращаемое_значение;
    $$ LANGUAGE plpgSQL;
```

Возможные два ***вида передачи значения*** в параметры функции:

IN — значение параметра передаётся в функцию (передача по значению) (по умолчанию)
OUT — значение параметра может быть возвращено из функции (передача по ссылке)
VARIADIC — произвольное количество параметров, передаваемых как однородный массив

Рассмотрим возможности языка *PLpg/SQL* для программирования хранимых процедур.

Аналогично стандартному *SQL* ***структура хранимой процедуры***, которая создаётся *PLpg/SQL*, содержит три блока: декларативный блок, исполняемый блок, блок обработки исключений — условий, которые вызывают ошибки или предупреждения об ошибках:

```
DECLARE
    // Объявление
    BEGIN
        // Выполнение команд обработки данных
    EXCEPTION
        // Обработка исключительных ситуаций
    END;
```

Исполняемый блок является обязательным, а два других блока могут отсутствовать.

! Важным отличием диалекта *PLpg/SQL* от стандарта является то, что **все** его функции принадлежат только к категории процедур выбора, а потому **всегда** должны возвращать значение. Вследствие этого для вызова функций в *PLpg/SQL* используется только команда SELECT.

Например, пользовательская функция, аналогичная рассмотренной в предыдущем параграфе процедуре выбора, возвращающей информацию о численности и успеваемости группы, будет выглядеть следующим образом:

```
CREATE OR REPLACE FUNCTION      GroupInfo (DName1 CHAR(30))
    RETURNS

    /* тип возвращаемого значения = TABLE */
    TABLE ("Шифр специальности" CHAR(2), "№ группы" INT,
            "Дисциплина" CHAR(30), "К-во студентов" BIGINT,
            "Минимальный рейтинг" INT, "Средний рейтинг" NUMERIC,
            "Максимальный рейтинг" INT)
    AS $$
    BEGIN

    /* возвращаемое значение – запрос */
        RETURN QUERY
            SELECT      S.Spec, GNum, DName, COUNT(S.Kod),
                        MIN(Mark), AVG(Mark), MAX(Mark)
                FROM    Student S, Rating R, Discipline D
                WHERE   S.Kod = R.Kod AND D.DKod = R.DKod  AND
                        D.DName = DName1
                GROUP BY    S.Spec, GNum, DName;
    END;

$$ LANGUAGE plpgsql;
```

При отладке для удобства частой модификации можно предварять команду создания функции командой её удаления при наличии:

```
DROP FUNCTION IF EXISTS      GroupInfo(DName1 CHAR(30));
```

Для вызова такой функции необходимо обратиться к ней, как к табличному выражению с помощью команды *SELECT*:

```
SELECT  *   FROM    GroupInfo('Исследование операций');
```

Переменные в PLpg/SQL

Переменная в *PLpg/SQL* может иметь любой тип данных, присущий стандартному *SQL*, такой как NUMBER, CHAR, DATE, или присущий диалекту *PLpg/SQL*, такой как BOOLEAN. Например, необходимо объявить переменную с именем *Part_No* так, чтобы она могла сохранять 4-хзначные числовые значения, и переменную с именем *In_Stock*, принимающую булево значение TRUE или FALSE. Объявляются переменные этого примера следующим образом:

```
…   Tax     NUMBER(4),
    Bonus   DEC(4, 2),
    Valid   BOOLEAN…
```

Кроме того, в *PLpg/SQL* существует возможность объявлять записи и таблицы, используя сложные типы данных этого диалекта: RECORD и TABLE.

Присвоение значений переменным в этом расширении ничем не отличается от стандарта *SQL*.

Во-первых, это оператор присваивания ‘:=‘. Например:

```
Tax := Price * Tax_Rate;

Bonus := Current_Salary * 0.10;

Valid := FALSE.
```

Во-вторых, это ввод в переменную значения из БД с помощью фразы INTO команды SELECT или FETCH. Например: вычислить 10% премии при выплате зарплаты сотрудника:

```
SELECT  Salary * 0.10  INTO  Bonus  FROM  Lecturer;
```

После этого значения переменной *Bonus* можно использовать в других вычислениях, или внести его в таблицу БД.

Переменные *PLpg/SQL* значительно расширяют возможности пользовательских функций, вплоть до „конструирования“ запросов внутри процедуры. Например, модификация запроса для вывода информации о группе демонстрирует „сборку“ запроса в виде текстовой строки с дальнейшим его выполнением:

```
CREATE OR REPLACE FUNCTION      GroupInfo (DName CHAR(30))
    RETURNS
    /* тип возвращаемого значения = TABLE */
    TABLE ("Шифр специальности" CHAR(2), "№ группы" INT,
            "Дисциплина" CHAR(30), "К-во студентов" BIGINT,
            "Минимальный рейтинг" INT, "Средний рейтинг" NUMERIC,
            "Максимальный рейтинг" INT)
    AS $$
    DECLARE
        str             VARCHAR;
        report_record   RECORD;
    BEGIN
        str := ' SELECT     S.Spec, GNum, DName, COUNT(S.Kod),
                            MIN(Mark), AVG(Mark), MAX(Mark)
                    FROM        Student S, Rating R, Discipline D
                    WHERE       S.Kod = R.Kod AND D.DKod=R.DKod  AND
                                D.DName = ' ' ' || DName || ' ' '
                    GROUP BY    S.Spec, GNum, DName ‘;
        /* выполнение запроса из строки str и возвращение ответа на запрос из
           процедуры */
        RETURN QUERY EXECUTE    str
    END;
$$ LANGUAGE plpgsql;
```

Атрибуты

Переменные в *PLpg/SQL* могут быть так называемыми „***атрибутами***“, то есть свойствами, позволяющими ссылаться на тип данных, который имеет один из столбцов таблиц БД, не повторяя его объявление. Синтаксис объявления переменной как атрибута следующий:

Атрибут%TYPE

Например, таблица *Books* содержит столбец с именем *Title*. Чтобы дать переменной *My_Title* тот же тип данных, что и у столбца *Title*, не зная точного определения этого столбца в БД, достаточно указать следующую инструкцию:

```
…My_Title   Books.Title%TYPE;…
```

Такое объявление переменной имеет два преимущества:

— нет необходимости знать точный тип данных столбца *Title*;

— если определение столбца *Title* в БД изменится, например, увеличится его длина, тип данных переменной *My_Title* изменится соответственно во время выполнения.

Также можно использовать другой вид атрибута с синтаксисом:

Атрибут%ROWTYPE

В PLpg/SQL для группировки данных используются записи. Запись состоит из нескольких столбцов, в которых могут храниться значения данных. ***Атрибут*** %ROWTYPE обозначает тип записи, которая представляет строку в таблице. Такая запись, то есть переменная, объявленная с ***атрибутом*** %ROWTYPE, может сохранять целую строку данных, полученную из таблицы или через курсор (который будет рассмотрен в следующем параграфе).

Столбцы в строке таблицы и соответствующие столбцы в записи имеют одинаковые имена и типы данных. В следующем примере объявляется запись с именем *Dept_Rec*:

```
…Dept_Rec    Dept%ROWTYPE;…
```

Для обращения к значениям столбцов записи используются уточнённые ссылки, как показывает следующая инструкция:

```
My_Deptno := Dept_Rec.Deptno;
```

Управляющие структуры

В *PLpg/SQL*, как и в других языках программирования, существуют ***команды передачи управления***, к которым принадлежат оператор *условного перехода* и операторы *циклов*.

В *PLpg/SQL* они называются ***управляющими структурами***: условного управления (IF-THEN-ELSE) и итеративного управления FOR-LOOP, WHILE-LOOP.

В качестве **примера** условного управления, рассмотрим программу, которая подсчитывает количество академических задолженностей студента, код которого равен 5, и в соответствии с допустимым количеством генерирует для студента предупреждающие сообщения. Оформим эту программу в виде функции *PLpg/SQL*:

```
CREATE FUNCTION     Предупреждение_if_else()
        RETURNS     INTEGER
        AS $$
        DECLARE
                min_positive_mark           CONSTANT     NUMERIC(2) := 60;
                max_negotive_mark_count     CONSTANT     NUMERIC(1) := 2;
                negotive_mark_count         NUMERIC(1);
                StudentKod                  INTEGER;
        BEGIN
                StudentKod := 5;
                SELECT    COUNT(R1.Kod)    INTO     negotive_mark_count
                        FROM       Rating R1
                        WHERE     Kod = StudentKod AND Mark < min_positive_mark;
                IF      negotive_mark_count > max_negotive_mark_count  THEN
                        INSERT INTO    Letters (kod, content)
                                VALUES(StudentKod,'Ваше количество задолженностей =' ||
                                        negotive_mark_count ||
                                        ', что превышает допустимое количество!');
```

```
        ELSE
            INSERT INTO    Letters (kod, content)
                VALUES(StudentKod,
                        'Вам необходимо ликвидировать задолженности в
                    количестве ' ||
                        negotive_mark_count);
        END IF;
        RETURN 1;
    END;$$
LANGUAGE 'plpgSQL';
```

Для обращения к этой функции необходимо выполнить следующий запрос:

```
SELECT    Предупреждение_if_else();
```

Как было уже отмечено, в *PostgreSQL* все функции должны что-либо возвращать. Из-за того, что от этой функции не ожидается никакого результата, создаём его искусственно – возвращаем значение 1 как символ того, что функция успешно выполнилась. То есть при вызове с помощью команды SELECT увидим в результате значения 1, а в таблице Letters соответствующие записи. Если у студента с кодом 5 вообще нет оценок или все оценки положительные, добавится запись с количеством задолженностей, равным 0.

Для цикла FOR-LOOP задаётся интервал целых чисел, а действия внутри цикла выполняются один раз для каждого целого в этом интервале. **Например**: следующая программа определяет количество академических задолженностей студентов с номерами от 1 до 7 и в соответствии с допустимым количеством генерирует для каждого студента сообщения:

```
CREATE FUNCTION     Предупреждение_for_loop()
    RETURNS    INTEGER
    AS $$
    DECLARE
        min_positive_mark          CONSTANT     NUMERIC(2) := 60;
        max_negotive_mark_count    CONSTANT     NUMERIC(1) := 2;
        negotive_mark_count        NUMERIC(1);
        StudentKod                 Student.Kod%TYPE;
        Name                       Student.FirstName%TYPE;
    BEGIN
        FOR     StudentKod    IN 1..7   LOOP
          SELECT    FirstName, COUNT(R.Kod)
                INTO      Name, negotive_mark_count
                FROM      Student S, Rating R
                WHERE     S.Kod = StudentKod AND
                          Mark < min_positive_mark AND S.Kod = R.Kod
                GROUP BY FirstName;
          IF    negotive_mark_count > max_negotive_mark_count      THEN
                INSERT INTO    Letters (kod, content)
                    VALUES(StudentKod, Name ||
                           '! Ваше количество задолженностей =' ||
                           negotive_mark_count ||
                           ', что превышает допустимое количество!');
          ELSE
                INSERT INTO    Letters (kod, content)
                    VALUES(StudentKod, Name ||
```

```
                                    '! Вам необходимо ликвидировать задолженности в
                                    количестве '|| negotive_mark_count);
            END IF;
        END LOOP;
        RETURN 1;
    END;$$
    LANGUAGE 'plpgSQL';
```

При вызове этой функции с помощью следующей инструкции:

```
SELECT    Предупреждение_for_loop();
```

увидим в её результате 1, а в таблице *Letters* — соответствующие записи. Если у студента вообще нет оценок или все они положительные, запись в таблице *Letters* добавится, но без сообщения (столбец *Content* будет пустой).

Обработка ошибок

В *PLpg/SQL* можно обрабатывать только внутренние определённые условия ошибок, которые называются исключениями. Когда возникает ошибка, обрабатывается исключение. Это значит, что нормальное выполнение прекращается, и управление передаётся в область обработки исключений или блока подпрограммы *PLpg/SQL*. Для обработки исключений создаются специальные программы — ***обработчики исключений*.**

Курсоры

При обновлении и удалении кортежей выборка строк из таблицы осуществлялась по первичному ключу или какому-то условию. То есть выбиралось некоторое множество (иногда единичное) кортежей из всей таблицы. Однако часто возникает необходимость построчного просмотра таблицы и обновления или удаления **текущего** кортежа. Для решения таких задач в *SQL* введен ещё один элемент ЯОД — **курсор.** Курсоры позволяют получить доступ к **отдельным строкам** таблицы.

В СУБД и, в частности, в *PostgreSQL* существует два вида курсоров: неявные и явные. *PostgreSQL* неявно объявляет курсор для любой инструкции манипулирования данными *SQL*, в том числе для запроса, возвращающего только одну строку. Более того, для выполнения предложений *SQL* и сохранения их результатов СУБД ***PostgreSQL*** использует „*персональные области*“ *SQL*. Курсор этого диалекта *SQL* позволяет обращаться к „персональной области“ по имени и получать из неё информацию.

Существует 4 основных команды *SQL*, связанных с курсорами: DECLARE *X* CURSOR, OPEN, FETCH, CLOSE.

Оператор DECLARE *X* CURSOR **определяет** курсор с именем *X* с помощью оператора SELECT. Например:

```
DECLARE    cStudent    CURSOR
    FOR    SELECT    *    FROM    Student;
```

Причём в отличие от представлений, в этой инструкции *не формируется отношение* для курсора. DECLARE CURSOR — это чисто *декларативное* (определяющее) выражение. *Запрос* для курсора *выполняется* только при **открытии** курсора командой OPEN *курсор*. Например:

```
OPEN        cStudent;
```

Однако и эта инструкция только лишь *формирует отношение* в результате запроса, но *не отображает* (не возвращает) этот результат. Это отношение называется **активным множеством.**

Для выборки *единственного* кортежа из результирующего множества используется команда FETCH. Например:

```
FETCH      cStudent;
```

Последующее выполнение этой команды будет возвращать по одному очередные (следующие) строки активного множества.

Стандартный *SQL* и практически все СУБД поддерживают только **однонаправленные** курсоры. То есть с помощью команды FETCH *нельзя* выбрать *предыдущую* строку активного множества. Для этого необходимо *закрыть* курсор и *открыть* его снова.

Диалект СУБД ***PostgreSQL*** поддерживает двунаправленные курсоры. Для этого в команду FETCH добавляется опция *направление*:

NEXT — следующая строка (по умолчанию),

PRIOR, FIRST, LAST — предыдущая, первая или последняя строка,

ABSOLUTE *шаг*, RELATIVE *шаг* — переход на определённый абсолютный или относительный *шаг* в любом направлении.

Например:

```
FETCH LAST FROM cStudent INTO x;
FETCH RELATIVE -2 FROM cStudent INTO x;
```

где *x* — некоторая локальная переменная или параметр, в который записывается результат.

К тому же в этом диалекте введена команда MOVE, которая передвигает курсор в любом направлении, однако без получения каких-либо данных. То есть она работает так же, как команда FETCH, за исключением того, что только передвигает курсор, не возвращая строку, к которой выполнено перемещение.

Для решения поставленной выше задачи удаления или обновления текущей строки в *SQL* существует два расширения:

1. **Модифицируемый курсор,** в объявлении которого указывается операнд

   ```
   FOR UPDATE OF      список_атрибутов;
   ```

   Для оптимизации и ускорения работы сервера БД *список_атрибутов* должен содержать имена только тех полей, которые действительно предполагается обновлять. Например:

   ```
   DECLARE   cRating    CURSOR
         FOR SELECT   *   FROM   Rating
         FOR UPDATE OF   Mark;
   ```

2. Второе расширение — это так называемые **current**-формы команд UPDATE и DELETE, которые можно применять к *модифицируемому курсору*, указанному в условии выборки этих команд с помощью оператора CURRENT OF *курсор*. Например:

   ```
   UPDATE   Rating   SET      Mark = 75
         WHERE      CURRENT OF   cRating;
   ```

   или

   ```
   DELETE FROM    Rating
         WHERE   CURRENT OF   cRating;
   ```

В большинстве случаев, требующих явного курсора, СУБД ***PostgreSQL*** позволяет использовать курсорные циклы FOR вместо операторов OPEN, FETCH и CLOSE, что упрощает программирование. Курсорный цикл FOR неявно объявляет индекс своего цикла как запись %ROWTYPE, открывает курсор, итеративно извлекает строки данных из активного множества в столбцах записи, и закрывает курсор после того, как все строки обработаны. В следующем примере курсорный цикл FOR неявно объявляет запись *Emp_Rec*, как принадлежащий типу *cEmp*%ROWTYPE:

```
DECLARE    Salary_Total   INTEGER;
        cEmp    CURSOR      IS
           SELECT    SecondName, Salary, HireDate, DeptKod    FROM   Lecturer;
                BEGIN
                     FOR     Emp_Rec   IN     cEmp     LOOP
                             Salary_Total := Salary_Total + Emp_Rec.Salary;
                     END LOOP;
                END;
```

Как показывает этот пример, для обращения к отдельным полям записи используются уточнённые ссылки.

Если объявляется курсор, который получает, скажем, фамилию, зарплату, дату приёма и должность сотрудника, то *PLpg/SQL* позволяет создать запись, содержащую такую же информацию. Это делается также с помощью атрибута %ROWTYPE. Предположим, что записано следующее объявление курсора и записи:

```
DECLARE cEmp  CURSOR      IS
      SELECT      SecondName, Salary, HireDate, Job    FROM     Lecturer;

Emp_rec    Lecturer%ROWTYPE;
```

При выполнении инструкции

```
FETCH     cEmp    INTO    Emp_rec;
```

значение столбца *SecondName* таблицы *Lecturer* будет присвоено атрибуту *SecondName* записи *Emp_rec*, значение атрибута *Salary* будет присвоено атрибуту *Salary*.

Закрытие курсора командой CLOSE приводит к *освобождению* использованных им системных ресурсов. Например:

```
CLOSE      cRating;
```

В заключение необходимо отметить такое: *большинство* версий *SQL не поддерживает* генерацию команд курсоров (конечно, кроме создания и удаления) в чистом виде. То есть, *нельзя вводить и отлаживать* эти команды практически во всех *интерактивных SQL-приложениях*. Эти команды и курсоры полностью предназначены для использования в рамках хранимых процедур и **триггеров** — ещё одного элемента языка данных *SQL*.

Триггеры

Триггеры имеют определённое сходство (аналогичны) хранимым процедурам, но в то же время отличаются от них.

Триггеры — это ***подпрограммы***, написанные на *SQL*, которые **выполняются тогда, когда происходит определённое *событие*, направленное к *конкретной* таблице.**

Таким событием может быть выполнение некоторой команды обновления ЯМД *SQL*: вставки, обновления или удаления данных. Соответственно **каждый триггер ассоциируется с *конкретной* операцией и *конкретной* таблицей**. Такое соответствие устанавливается при

создании триггера командой CREATE TRIGGER. Формат этой команды в разных СУБД несколько отличается. Поэтому сначала рассмотрим построение триггеров в стандартном *SQL*, а потом — особенности работы с триггерами с использованием диалекта *PostgreSQL*.

Триггеры могут использоваться в следующих областях функционирования приложений:

— реализация контроля за всеми действиями пользователей БД;
— обеспечение целостности содержимого БД;
— реализация сложных правил работы программ;
— обеспечение сложных правил безопасности данных;
— автоматическое создание значений в полях таблиц БД.

В *качестве примера обеспечения целостности* рассмотрим задачу удаления кортежей, относящихся к выпускникам ВУЗа, из двух таблиц: *Student* и *Rating*. Эта задача решалась путём вызова выполняемой процедуры, потому что было необходимо удалить *сначала* кортежи из *детализированной* таблицы, а *потом* — из *базовой*. Решение этой задачи с помощью триггеров будет выглядеть следующим образом:

```
CREATE TRIGGER DelRating FOR Student
ACTIVE BEFORE DELETE
AS BEGIN
        DELETE FROM Rating R
            WHERE R.Kod = OLD.Kod;
    END
```

! Обратите внимание, что этот триггер работает с таблицей *Rating*, однако „привязан“ к таблице *Student* и направленному к ней событию DELETE.

В результате решение поставленной задачи сведётся к единственному (как и при *хранимой процедуре*) запросу:

```
DELETE FROM Student WHERE GNum = 941;
```

Задача. Создать триггер для *полного* решения задачи удаления сплошных двоечников из раздела „Подзапросы“.

В команде CREATE TRIGGER добавился ещё ряд инструкций, характерных *только* для *языка хранимых процедур и триггеров*:

Параметр	**Описание**
ACTIVE	необязательный оператор, который используется по умолчанию (только в *Interbase*), предназначен *для активации* триггера. С помощью инструкции ALTER TRIGGER INACTIVE в *Interbase* можно „*отключить*“ триггер, не уничтожая его.
BEFORE	говорит о том, что триггер должен быть активирован *перед* выполнением команды, с которым связан триггер (INSERT, UPDATE или DELETE). Противоположная ей инструкция — AFTER (активация *после* выполнения операции).
OLD	**контекстная переменная**, применяемая *для ссылки на значение атрибута*, которое *было* в кортеже ***перед*** выполнением операции UPDATE или DELETE. Соответственно, контекстная переменная NEW позволяет сослаться на *новое значение столбца* ***при*** выполнении (или ***после*** выполнения) операции UPDATE или INSERT.

Например:

```
CREATE TRIGGER  AddNewIncrements  FOR  Lecturer
AFTER INSERT
AS  BEGIN
        INSERT INTO  SalaryIncrements (LKod)  VALUES(NEW.Kod);
    END
```

Этот триггер позволяет при выполнении команды

```
INSERT INTO  Lecturer  VALUES(20, 'Ааа', 'Ббб', 'Ввв', 'Доц.', 6, NULL, NULL);
```

автоматически добавить кортеж в таблицу *SalaryIncrements* со значением поля *LKod* совпадающим с введённым значением поля *Kod* и нулевым значением всех надбавок (значение по умолчанию).

В СУБД ***PostgreSQL*** триггер состоит из двух частей: *триггерное событие* и *действия триггера*.

Триггерное событие описывается в виде:

```
CREATE OR REPLACE TRIGGER Имя_триггера Тип_события Событие
    ON Имя_таблицы Область_применения;
```

Параметр *Тип_события* (BEFORE или AFTER) и параметр *Имя_таблицы* (операции модификации над которой перехватываются триггером) ничем не отличаются от используемых в стандартном *SQL*, как и параметр *Событие*, определяющий тип *SQL*-операции. Но в этом диалекте при операции UPDATE можно указывать ещё и имя атрибута таблицы, при воздействии на который срабатывает триггер.

Основной особенностью триггерного события *PLpg/SQL* является параметр *Область_применения*, который может принимать следующие значения:

FOR EACH ROW	триггер срабатывает столько раз, сколько операция модификации (UPDATE) воздействует на записи таблицы (если при операции модификации не была затронута ни одна запись, то триггер не сработает ни разу);
FOR STATEMENT	триггер срабатывает лишь один раз вместе с самой операцией модификации, даже если операция модификации не затронула ни единой записи таблицы.

При использовании множественных событий, когда триггер может перехватывать сразу несколько типов событий (INSERT, UPDATE, DELETE) в подпрограмму *PLpg/SQL* необходимо включить системные логические переменные INSERTING, UPDATING, DELETING, которые имеют тип данных BOOLEAN (TRUE, FALSE) и возвращают TRUE, если триггер сработал при операции INSERT, UPDATE, DELETE соответственно.

Действия триггера — это *хранимая процедура* в виде функции (FUNCTION), которая выполняется при срабатывании триггера и возвращает значение типа TRIGGER. Если процедура спроектирована с использованием языка *PLpg/SQL*, то, кроме стандартных *контекстных переменных* NEW. и OLD., можно также использовать следующие *системные переменные*:

TG_NAME	имя выполняемого триггера;
TG_WHEN	значение параметра *Тип_события* из триггерного события;
TG_LEVEL	значение параметра *Область_применения* из триггерного события;
TG_OP	тип операции модификации перехваченной триггером;

TG_RELNAME имя таблицы, над которой выполняется операция модификации;
TG_NARGS количество параметров *хранимой процедуры*, которая входит в действия триггера;
TG_ARGV[] массив значений параметров *хранимой процедуры* (индекс начинается с 0).

Триггер может обеспечивать сохранение в специальных таблицах следующей информации: имена пользователей, которые вносят изменения, время изменений, типы операций, выполняемых пользователем, содержание операций модификации.

Приведём пример. Пусть в БД существует таблица *Employer* со структурой:

```
CREATE TABLE   Employer
      (Emp_Num   INTEGER,
       Name      CHAR(40),
       Job       CHAR(20),
       Tax       NUMERIC(4,2));
```

Необходимо отслеживать изменения над таблицей *Employer* путём сохранения их в таблицу *Audit_Employer*, структура которой может иметь вид:

```
CREATE TABLE   Audit_Employer
      (Emp_Num     INTEGER,
       Name        CHAR(40),
       Job         CHAR(20),
       Tax         NUMERIC(4,2),
       Oper_type   CHAR(1),
       User_Name   CHAR(20),
       Change_time TIMESTAMP);
```

Для этого создадим триггер, который перехватывает все операции модификации над таблицей *Employer* и вносит их в таблицу *Audit_Employer*.

Описание триггерного события и действий триггера представлено ниже.

```
CREATE OR REPLACE FUNCTION Audit_Employer_F() RETURNS TRIGGER
    AS $$
        BEGIN
            IF (Tg_op ='INSERT' OR Tg_op ='UPDATE') THEN
                INSERT INTO Audit_Employer VALUES
                    (NEW.Name, NEW.Job, NEW.Tax, SUBSTR(Tg_op,1,1),
                        CURRENT_USER, CURRENT_TIME);
                RETURN NEW;
            ELSE
                INSERT INTO Audit_Employer VALUES
                    (OLD.Name, OLD.Job, OLD.Tax, SUBSTR(Tg_op,1,1),
                        CURRENT_USER, CURRENT_TIME);
                RETURN OLD;
            END IF;
        END;
    $$ LANGUAGE plpgSQL;

CREATE TRIGGER Audit_Employer
    AFTER INSERT OR UPDATE OR DELETE ON Employer
    FOR EACH ROW EXECUTE PROCEDURE Audit_Employer_F();
```

В представленном триггере применяются следующие системные функции:
CURRENT_USER — имя пользователя, выполняющего операции;

CURRENT_TIME — текущее время;

SUBSTR(*Строка, Начальный_номер, Количество_символов*) — это функция, которая извлекает из строки подстроку, начинающуюся с символа с номером (*Начальный_номер* + 1) и имеющую длину, равную значению параметра *Количество_символов*.

Триггеры, описываемые конструкциями диалекта *PLpg/SQL*, могут обеспечивать более ***сложные правила целостности***, чем это предусмотрено в стандарте.

Кроме случаев, когда необходимая целостность не обеспечивается простыми декларативными ограничениями целостности с использованием внешних ключей (FOREIGN KEY), триггеры *PLpg/SQL* используются, когда целостность связывает таблицы, расположенные на разных узлах распределённой БД, или реализуют более сложные правила целостности, чем ограничение по значению (CHECK-инструкция). В частности, триггер может выполнить *сложный контроль данных прежде, чем* позволить выполнить некоторые операции, перехваченные триггером.

Например: необходимо создать правило ограничения целостности, которое запрещает назначать работнику зарплату, которая не входит в допустимые границы.

В решении используется дополнительная таблица-справочник *Job_Classification* со следующей структурой:

```
CREATE TABLE    Job_Classification
      (Job          CHAR(20)    PRIMARY KEY,
       Min_Tax   NUMERIC(4,2),
       Max_Tax   NUMERIC(4,2));
```

Примером заполнения таблицы может быть следующее:

```
INSERT INTO    Job_Classification    VALUES ('JOB1',10,30);

INSERT INTO    Job_Classification    VALUES ('JOB2',20,60).
```

Описание триггера для решения поставленной задачи выглядит следующим образом:

```
CREATE OR REPLACE FUNCTION Tax_Check_F() RETURNS TRIGGER
      AS $$
      DECLARE
            Min_Tax Job_Classification.Min_Tax%TYPE;
            Max_Tax Job_Classification.Max_Tax%TYPE;
      BEGIN
      /* получить минимальный и максимальный оклад для новой должности
      служащего из таблицы Job_Classification в Min_Tax и Max_Tax */
            SELECT Min_Tax, Max_Tax INTO Min_Tax, Max_Tax
                  FROM Job_Classification WHERE Job = NEW.Job;
      /* если новый оклад служащего меньше или больше, чем ограничение по
      должностной классификации, генерируется исключительная ситуация, а операция
      отменяется. */
            IF (NEW.Tax < Min_Tax OR NEW.Tax > Max_Tax)
                  THEN RAISE EXCEPTION
                        'Оклад служащего % вне границ диапазона', NEW.Tax;
            END IF;
            RETURN NEW;
      END;
      $$ LANGUAGE plpgSQL;

CREATE TRIGGER Tax_Check BEFORE INSERT OR UPDATE ON Employer
      FOR EACH ROW EXECUTE PROCEDURE Tax_Check_F();
```

Для демонстрации использования триггера при ***обеспечении сложных правил безопасности*** приведём следующий пример. **Задача**: необходимо ограничить время работы с БД: сотрудники предприятия могут работать с БД только с 8 до 18 часов и не могут работать по выходным дням. **Решение:**

```
CREATE OR REPLACE FUNCTION Repmit_Changes_F() RETURNS TRIGGER
    AS $$
    BEGIN
    /* проверка для выходных дней */
        IF (TO_CHAR(CURRENT_DATE,'DY')= 'SAT' OR
                TO_CHAR(CURRENT_DATE,'DY')= 'SUN')
            THEN RAISE EXCEPTION 'В выходные работать нельзя';
        END IF;
    /* проверка по часы работы (с 8 до 18) */
        IF (TO_CHAR(CURRENT_DATE,'HH24') < 8 OR
                TO_CHAR(CURRENT_DATE,'HH24') > 18)
            THEN RAISE EXCEPTION 'Сейчас работать нельзя';
        END IF;
    END;
    $$ LANGUAGE plpgSQL;

CREATE TRIGGER Emp_Repmit_Changes
    BEFORE INSERT OR UPDATE OR DELETE ON Employer FOR Statement
    EXECUTE PROCEDURE Repmit_Changes_F();
```

Функция TO_CHAR(CURRENT_DATE,'HH24') извлекает из текущей даты текущее время. Функция TO_CHAR(CURRENT_DATE,'DY') извлекает из текущей даты день недели. Если в БД *PostgreSQL* используется русская локализация, то функция возвращает аббревиатуру названий дней недели на русском языке (ПНД, ВТР ..., СБТ). Для проверки значений можно выполнить запрос:

```
SELECT TO_CHAR(CURRENT_DATE,'DY') FROM EMPLOYER;
```

Функция TO_CHAR(CURRENT_DATE,'DD') извлекает из текущей даты значение дня.

Поддержка хранилищ данных средствами PLpg/SQL.

Основная цель создания хранилищ данных — ускорить или оптимизировать обработку больших объёмов информации. В *PLpg/SQL* для поддержки таких структур используются такие элементы ЯОД этого диалекта *SQL*, как триггеры и материализованы представления.

Как отмечалось ранее (см. п. „Представления“) материализованные представления отличаются от стандартных для *SQL* виртуальных представлений (или просто представлений) тем, что хранятся в БД в явном (физическом или материализованном) виде, тогда как виртуальное представление хранит только *SQL*-запрос, описывающее желательную форму представления БД для пользователя. Материализованное представление — это реальная таблица БД, сформированная и заполненная на основании *SQL*-запроса.

Для поддержки материализованных представлений необходимо пройти два этапа:

— создать таблицы на основании *SQL*-запроса;

— обеспечить согласование содержимого таблиц, входящих в *SQL*-запрос, и содержимого материализованного представления.

Рассмотрим **пример** создания материализованного представления, которое сохраняет результаты следующего *SQL*-запроса:

```
SELECT Job, COUNT(Job) FROM Lecturer GROUP BY Job;
```

Для создания таблицы материализованного представления с именем *Report* необходимо выполнить запрос:

```
CREATE TABLE Report AS
    SELECT Job, COUNT(Job) FROM Lecturer GROUP BY Job;
```

Ниже представлен триггер, обеспечивающий асинхронное согласование содержимого таблицы *Lecturer*:

```
CREATE OR REPLACE FUNCTION Report() RETURNS TRIGGER
    AS $$
    BEGIN
        IF Tg_op = 'INSERT' THEN
            SELECT COUNT(*) FROM Report WHERE Job = NEW.Job;
            IF FOUND THEN
                UPDATE Report SET Count = Count + 1
                    WHERE Job = NEW.Job;
            ELSE
                INSERT INTO Report VALUES(NEW.Job, 1);
            END IF;
        END IF;
        IF Tg_op = 'UPDATE' AND NEW.Job != OLD.Job THEN
            UPDATE Report SET Count = Count + 1
                WHERE Job = NEW.Job;
            UPDATE Report SET Count = Count - 1
                WHERE Job = OLD.Job;
        END IF;
        IF Tg_op = 'DELETE' THEN
            UPDATE Report SET Count = Count - 1
                WHERE Job = OLD.Job;
        END IF;
        IF Tg_op = 'INSERT' OR Tg_op = 'UPDATE' THEN
            RETURN NEW;
        ELSE RETURN OLD;
        END IF;
    END;
    $$ LANGUAGE plpgSQL;

CREATE TRIGGER Report AFTER INSERT OR UPDATE OR DELETE
    ON Lecturer FOR EACH ROW EXECUTE PROCEDURE Report();
```

Демонстрационный пример

Запишите *SQL*-запросы для манипулирования данными из таблиц, созданных в задаче „Демонстрационный пример“ разд. „Язык определения данных (ЯОД, DDL) SQL. Часть 1“.

П.1. Создайте триггер, который при добавлении нового рейса внесёт данные в связанные(-ую) таблицы(-у).

Решение.

```
CREATE FUNCTION voyage_class()
    RETURNS trigger
```

```
    AS    BEGIN
                INSERT INTO class (voyage) VALUES (new.id_voyage);
               RETURN new;
          END;'
LANGUAGE 'plpgSQL';

CREATE TRIGGER voyage_class
    AFTER INSERT ON voyage
    FOR EACH ROW
    EXECUTE PROCEDURE voyage_class();
```

П.2. Создайте триггер, для удаления уволенного сотрудника

Решение.

```
CREATE FUNCTION del_employee()
    RETURNS trigger
    AS    BEGIN
               UPDATE ticket SET employee=NULL
                WHERE employee=old.id_employee;
               RETURN old;
          END;'
LANGUAGE 'plpgSQL';

CREATE TRIGGER del_employee
    BEFORE DELETE ON employee
    FOR EACH ROW
    EXECUTE PROCEDURE del_employee();
```

П.3. Создайте функцию, которая будет при выдаче билета вносить данные в таблицы *Passenger* и *Ticket*, при этом проверять существует ли сотрудник, выдающий билет, и класс салона, в который выдают билет, при ошибке должны быть отображены соответствующие сообщения.

Решение.

```
CREATE OR REPLACE FUNCTION ticket_create -- название функции
          (full_name_pas CHAR (30), sex CHAR (1), passport CHAR, name_class CHAR
          (30), place INT, full_name_emp CHAR(30), operation CHAR(7))
    RETURNS INTEGER -- функция возвращает тип данных INTEGER
    AS $$
  -- Начало тела функции
  -- Секция объявления переменных.
    DECLARE
    id_passenger passenger.id_passenger%TYPE; -- тип данных ссылается на тип
          данных поля id_passenger таблицы "passenger"
    id_ticket ticket.id_ticket%TYPE;
    t_class class.id_class%TYPE;
    t_employee employee.id_employee%TYPE;
  -- Секция тела функции.
    BEGIN
      -- Получение нового значения кода пассажира из генератора "s_passenger""
       id_passenger := NEXTVAL('s_passenger');
         -- Добавление записи в таблицу
```

```
        INSERT INTO passenger VALUES (id_passenger,full_name_pas,sex,passport);
            -- Получение нового значения кода билета из генератора “s_ticket”"
        id_ticket := NEXTVAL('s_ticket');
            -- Проверка на правильность значения входного параметра класса
            -- из таблицы “class” через получение ответа на запрос.
            -- Результат запроса отправляется в переменную t_class (код класса).
        SELECT id_class INTO t_class FROM class WHERE name = name_class;
            -- Если ответ пустой, то значение класса некорректное.
        IF NOT FOUND THEN
            -- Вызов обработчика ошибки и вывод на экран сообщения.
            -- В строке с сообщением параметр % указывает на
            -- значение переменной после запятой.
                RAISE EXCEPTION 'Ошибка: Класса % не существует', name_class;
        END IF;

        SELECT id_employee INTO t_employee FROM employee WHERE full_name =
                full_name_emp;
            -- Если ответ пустой, то значение сотрудника некорректное.
        IF NOT FOUND THEN
            -- Вызов обработчика ошибки и вывод на экран сообщения.
            -- В строке с сообщением параметр % указывает на
            -- значение переменной после запятой.
                RAISE EXCEPTION
                        'Ошибка: Сотрудника % не существует', full_name_emp;
        END IF;

    -- Добавление записи к таблице
        INSERT INTO ticket
                VALUES (id_ticket, id_passenger, t_class, place, t_employee, operation);
            -- Вывод на экран сообщения об успешной выдаче билета
        RAISE NOTICE ' Билет выдан';
            -- Возврат из функции значения кода билета
        RETURN id_ticket;

    END;
-- Завершение тела функции
    $$ LANGUAGE plpgSQL; -- название модуля обработки языковых конструкций
            тела функции
```

Пример вызова функции выдачи билета:

```
SELECT Ticket_Create ('Иванов','м','КМ 897654','бизнес',12,'Арсирий Е.А.','покупка');
```

И, наконец, проверка результата:

```
SELECT * FROM Passenger;
```

Сценарии

Для того чтобы упорядочить вызов и облегчить создание последовательности **несвязанных** запросов, например, команд ЯОД *SQL* для создания таблиц, хранимых процедур, триггеров и т.п., их целесообразно включать в так называемые **сценарии** (*script*). Это файлы, имеющие стандартное расширение *SQL*, которые можно создавать и редактировать в любом текстовом редакторе.

Мы не будем рассматривать все особенности сценариев. Отметим только два нюанса:

1. Подавляющее большинство сценариев должно начинаться командой

   CONNECT имя/алиас БД USER имя_пользователя PASSWORD пароль;

   и заканчиваться командами

   COMMIT;

   и

   EXIT;

   Если создаются связанные таблицы или для таблиц создаются представления, индексы и пр., то командой COMMIT (сохранение результатов транзакции, то есть выполненных действий) должна заканчиваться каждая команда CREATE TABLE.

   Кстати, если в параметрах СУБД или интерактивного приложения типа *Windows Interactive SQL* установлена опция AUTOCOMMIT, то добавлять эту команду к сценарию не надо (даже, запрещено).

   Команда, которая выполняет действие, обратное COMMIT, то есть отменяет результаты транзакции — ROLLBACK („*откат*“ выполненных операций).
2. Вторая тонкость связана с созданием хранимых процедур с помощью сценариев (и не только). Дело в том, что *во всех без исключения СУБД* команды, из которых состоит *хранимая процедура*, должны *обязательно* заканчиваться символом ‘;’. Но при этом большинство СУБД требует, чтобы этим символом заканчивались и все запросы в сценарии. Поэтому, если не принять соответствующих мер, то эти СУБД будут стараться выполнять команды *хранимой процедуры* по мере её создания. Для того чтобы избежать такой ситуации, необходимо *каждую* или *блок команд CREATE PROCEDURE* предварять командой

   SET TERM[INATOR] *символ* ;

   где *символ* — любой малоиспользуемый символ, например, ‘^’, и заканчивать *каждую* команду создания хранимой процедуры этим **символом-заменителем** (подобно оператору END), а после неё или в самом конце сделать *обратное переназначение*:

   SET TERM ; *символ*

Язык управления данным SQL (ЯУД, DCL)

Управление доступом

Понятие управления доступом имеет смысл, в первую очередь, в многопользовательском режиме работы СУБД.

На уровне БД управление доступом заключается в создании и удалении пользователей путём назначения их имён, паролей и прав доступа к конкретным БД и является прерогативой администратора БД (DBA или SYSDBA). Каждый пользователь является владельцем таблиц, связанных с той или иной БД. Соответственно, владелец таблицы может управлять доступом к данным на уровне таблиц. Такое управление заключается в назначении **привилегий** другим пользователям, то есть определении того, может или нет конкретный пользователь выполнять ту или другую команду.

Привилегии *SQL* являются **объектными**. То есть пользователь имеет привилегию (право) выполнять *конкретную команду* для *конкретного объекта* БД: таблицы, представления, столбца и т.п.

Привилегии, назначенные владельцем таблицы, совпадают по мнемонике и по смыслу с командами ЯОД *SQL*: SELECT, INSERT, DELETE, UPDATE и REFERENCES.

Для предоставления того или другого доступа к таблице (таблицам) их владелец должен выполнить команду GRANT. Её формат:

GRANT *список_операций* ON *таблица* TO *список_пользователей*;

Например:

GRANT SELECT, INSERT ON Student TO User1, User2;

Из приведенного множества привилегий необходимо выделить UPDATE и REFERENCES, так как они, кроме общего формата команды GRANT, *позволяют ограничить привилегии множеством столбцов*. Например:

GRANT UPDATE(Mark) ON Rating TO Lecturer10;

для обновления полей.

Или, например:

GRANT REFERENCES (Spec) ON Speciality TO DeptHead;

Эта инструкция позволяет пользователю *DeptHead* создавать внешние ключи, которые будут ссылаться на поле *Spec* таблицы *Speciality* как на родительский ключ. Если в этом привилегии не указаны столбцы, то в качестве родительских могут использоваться любые поля этой таблицы.

Следующий момент. Если в *списке_операций* команды GRANT перечислены **все привилегии**, то есть какому-либо пользователю открыт **полный доступ** к таблице, то вместо такого списка в *SQL* можно воспользоваться инструкцией ALL PRIVILEGES. Например:

GRANT ALL [PRIVILEGES] ON Student TO DeptHead;

Аналогично, если **какую-нибудь** привилегию или список привилегий необходимо открыть **для каждого пользователя** системы, даже тех, которые только **будут созданы**, то вместо *списка_пользователей* достаточно указать оператор PUBLIC. Например:

GRANT SELECT ON Rating TO PUBLIC;

Инструкция в данном примере открывает всем пользователям доступ для чтения **всех**

данных таблицы *Rating*: таково действие привилегии SELECT.

Однако часто возникает необходимость ограничить просмотр таблицы конкретными полями. Так как указать **столбцы** можно *только в привилегиях* UPDATE и REFERENCES, в этом случае необходимо воспользоваться представлениями. Например, создать представление:

```
CREATE VIEW    StudEmail   AS
      SELECT    SecondName, FirstName, Email   FROM   Student;
```

и предоставить другому пользователю права на просмотр уже не таблицы *Student*, а представления *StudEmail*:

```
GRANT   SELECT   ON   StudEmail   TO   PUBLIC;
```

В отличие от применения привилегий к базовым таблицам, привилегии на представления позволяют ограничить доступ не только к столбцам, но и к кортежам. Например:

```
CREATE VIEW    OSStudents
AS  SELECT   *   FROM    Student
            WHERE   Spec = 'OC'
            WITH CHECK OPTION;

GRANT   UPDATE   ON   OSStudents   TO   DeptHead;
```

Инструкция WITH CHECK OPTION не позволяет изменять шифр специальности.

Ещё одна инструкция, внешне похожая на эту —— WITH GRANT OPTION — позволяет **транслировать привилегии** по „цепочке“ пользователей. Например, если в группе разработчиков таблицы создают одни пользователи, приложения – другие, а использовать данные будут третьи, то владельцы таблиц могут *передать полномочия по назначению привилегий* другим пользователям:

```
GRANT   UPDATE, SELECT   ON   OSStudents      TO   DeptHead
      WITH GRANT OPTION;
```

Тогда пользователь *DeptHead* может выполнить запрос:

```
GRANT   SELECT   ON   User1.OSStudents      TO   ViceDeptHead;
```

где *User*1 — владелец таблицы *Student* и представления *OSStudents*.

Отмена привилегий осуществляется командой REVOKE, имеющей формат абсолютно идентичный команде GRANT.

В ***PostgreSQL*** существует возможность управлять базами данных, доступом к ним и решать множество задач над ними, используя **командную** консоль *PostgreSQL*. Для вызова командной консоли предназначена команда *pSQL*, которая поддерживает ряд опций.

Например, для получения информации о текущих привилегиях, которые предоставлены ролям при организации доступа к таблицам, используется опция \dp *Имя_таблицы* или \z *Имя_таблицы*.

Вызов командной консоли для решения этой задачи для таблицы *Lecturer* выглядит следующим образом:

```
pSQL /z Lecturer
```

Опция \du [*Шаблон*] предоставляет список *всех* ролей базы данных или только ролей,

которые соответствуют шаблону.

Привилегии доступа содержатся в списке доступа, элементы которого описываются следующей структурой:

Имя_роли = *Список_прав_доступа* / *Имя_владельца_таблицы*

Список прав доступа может содержать следующие символы:

a, r, w, d, x, t — права доступа, соответственно, на *внесение* новой записи, *чтение* записей, *изменение* записей, *удаление* записей, *контроль* ссылок, *выполнение* триггеров.

Особенности управления доступом в СУБД PostgreSQL

Роли и управление ролями

В версиях СУБД *PostgreSQL* до 8.1 при управлении пользователями и группами пользователей использовались соответствующие команды создания (CREATE USER, CREATE GROUP) и изменения параметров пользователя или группы (ALTER USER и ALTER GROUP). В версиях СУБД *PostgreSQL*, начиная с 8.1, появился более гибкий механизм управления — **роли**.

Для ***создания роли*** используется команда CREATE ROLE.

CREATE ROLE *Имя_роли* [[WITH] *Опция* [*Опция*, ...]]

Опциями являются:

SUPERUSER | NOSUPERUSER	роль с правами администратора / без прав (по умолчанию — NOSUPERUSER)
CREATEDB | NOCREATEDB	роль может создавать свои БД / не может (по умолчанию — NOCREATEDB)
CREATEROLE | NOCREATEROLE	роль может создавать другие роли / не может
INHERIT | NOINHERIT	роль автоматически будет / не будет наследовать привилегии наследуемых ими ролей (по умолчанию — NOINHERIT)
LOGIN | NOLOGIN	роль может устанавливать соединение / не может (для группы) (по умолчанию — LOGIN)
CONNECTION LIMIT connlimit	количество одновременных подключений роли к СУБД (по умолчанию — нет ограничений)
[ENCRYPTED | UNENCRYPTED] PASSWORD *Пароль*	пароль для авторизованной установки соединения роли с СУБД, может храниться в зашифрованном виде или открыто (по умолчанию — ENCRYPTED)
VALID UNTIL *дата*	крайний срок действия роли (по

умолчанию — без срока действия)

Например, для создания роли *Director*, которая будет пользователем СУБД (имеет право подключаться к СУБД через процедуру авторизации), необходимо выполнить команду:

CREATE ROLE Director LOGIN;

Например, для создания роли *Director*, которая будет пользователем СУБД до 31-10-2007, необходимо выполнить команду:

CREATE ROLE Director LOGIN WITH PASSWORD ‘dbms’ VALID UNTIL ‘2007-10-31’;

Например, для создания роли *Programmers*, которая в дальнейшем будет соответствовать группе пользователей, необходимо выполнить команду

CREATE ROLE Programmers NOLOGIN;

Для **изменения параметров роли** используется команда ALTER ROLE:

ALTER ROLE имя_роли [[WITH] *Опция* [*Опция*, …]]

Опции этой команды совпадают с опциями команды CREATE ROLE.

Например, для изменения пароля пользователя необходимо выполнить команду:

ALTER ROLE postgres WITH PASSWORD ‘dbms2000’;

Для **управления наследованием ролей** так же, как в стандартном *SQL* для управления привилегиями, используются команды GRANT и REVOKE.

Команда GRANT позволяет одной роли наследовать привилегии другой роли и имеет синтаксис:

GRANT *Имя_роли_предоставляющей_привилегии*
TO *Имя_роли_получающей_привилегии*;

Например, для предоставления роли *Director* привилегий роли *Programmers* необходимо выполнить команду:

GRANT Programmers TO Director;

Команда REVOKE, которая позволяет с одной роли снять привилегии другой роли, имеет следующий синтаксис:

REVOKE *Имя_роли_предоставляющей_привилегии*
FROM *Имя_роли_получающей_привилегии*;

Например, для снятия с роли *Director* привилегий роли *Programmers* необходимо выполнить команду:

REVOKE Programmers FROM Director;

Если роль создана без наследования ролей по умолчанию (опция INHERIT), при работе с этой ролью необходимо принудительно устанавливать роли-наследники с помощью команды SET ROLE. Например:

SET ROLE *Имя_унаследованной_роли*;

для установления роли свойства автоматического наследования

Сброс ролей может выполняться двумя способами:

SET ROLE NONE;

или

RESET ROLE;

При работе с ролями можно использовать две системных переменных

SESSION_USER имя пользователя , открывшего сессию;
CURRENT_USER имя пользователя, работающего под заданной ролью.

Для получения значений этих переменных необходимо выполнить инструкцию:

SELECT SESSION_USER, CURRENT_USER;

После создания роли, которая может устанавливать соединение с СУБД, можно установить это соединение.

Например, для установки соединения с СУБД пользователю *Director*:

pSQL postgres -U Director

Управление схемами данных

В большинстве СУБД для логического разделения данных на отдельные именные пространства используется понятие **схем** (*schema*). Схема позволяет создавать объекты БД (таблицы, представления, функции, последовательности) *в отдельном пространстве имён*, доступ к которому осуществляется эквивалентно объектному представлению: *имя_схемы.имя_объекта*.

Соответственно, одно и то же имя объекта может без какого-либо конфликта использоваться в различных схемах. В отличие от баз данных, схемы жёстко не разделяются: пользователь может получить доступ к объектам в любой схеме в пределах базы данных, к которой он подключён, если у него есть соответствующие привилегии.

Целью использования механизма схем может быть:

- необходимость разрешить нескольким пользователям использовать одну БД, не смешивая их данные;
- логического распределения таблиц между группами в одной физической БД;
- ограничения доступа к таблицам со стороны разных пользователей на основе представлений;
- возможность более прозрачного управления объектами БД путём их организации в логические группы;
- устранение конфликтов в использовании имён или других объектов разными приложениями с помощью их размещения в отдельных схемах.

Необходимость использования механизма схем определяется задачами, решаемыми в рамках конкретной БД, и стратегией использования этой БД:

Задача/стратегия	Для каких случаев
Не создаются никакие схемы. Все роли неявно получают доступ к схеме *PUBLIC*.	1. Если в БД работает только один пользователь или группа скооперированных пользователей. 2. Позволяет смягчить переход из СУБД, где нет схем.
Создание схемы для каждой роли с именем, таким же как и у самой роли.	1. Если каждый пользователь имеет отдельную схему, то он по умолчанию получает доступ к своей схеме (по умолчанию путь поиска начинается с элемента *$user*). 2. Можно отобрать доступ к схеме public (или вообще удалить её), так что пользователи будут ограничены только своими схемами.

Задача/стратегия	Для каких случаев
Для установки разделяемых приложений их помещают в отдельные схемы.	1. Если таблицы должны использоваться любым пользователем. 2. В случае дополнительных функций от сторонних разработчиков и т.д. При этом необходимо предоставить соответствующие привилегии, чтобы разрешить доступ к ним другим пользователям.

Для **создания схемы данных** используется операция

CREATE SCHEMA *Имя_Схемы*;

Пример создания схемы *Director*:

CREATE SCHEMA Director;

Для **предоставления пользователю прав доступа к схеме** используется команда:

GRANT USAGE ON SCHEMA *название_ схемы* TO *название_пользователя*;

Пример предоставления пользователю прав доступа к схеме

GRANT USAGE ON SCHEMA Director TO Director;

Для **назначения пользователя владельцем схемы** используется команда:

ALTER SCHEMA *Имя_ схемы* OWNER TO *Имя_пользователя*;

Пример включения пользователя *Director* в схему *Director*:

ALTER SCHEMA Director OWNER TO Director;

Несмотря на принадлежность схемы конкретному пользователю, все его запросы будут выполняться к схеме *PUBLIC*. Поэтому при выполнении запросов к конкретной схеме необходимо в качестве префикса соответствующего объекта указывать её имя. Например:

SELECT * FROM Director.Lecturer;

После установки соединения с БД СУБД автоматически выполняет поиск всех таблиц из запросов в схеме *PUBLIC*. При этом необходимо помнить, что в стандарте *SQL* не существует концепции схемы *PUBLIC*. Поэтому, если есть необходимость максимального соответствия стандарту, то использовать схему *PUBLIC* нельзя (возможно даже следует удалить её).

Для **установки порядка доступа к схемам**, то есть порядка поиска таблиц в схемах, используется команда:

SET SEARCH_PATH TO *Имя_ схемы*;

Например:

SET SEARCH_PATH TO PUBLIC;

Когда для пользователя определены несколько схем, названия схем к именам таблиц в запросах добавляются в указанном порядке. Если таблица не существует в схеме с названием *имя_ схемы*1, СУБД будет использовать следующее название схемы — *имя_ схемы*2, пока таблица не будет найдена в схеме.

Пример установки порядка поиска:

SET SEARCH_PATH TO Director, PUBLIC;

Для того, чтобы пользователь после установки соединения с СУБД автоматически

определял порядок поиска в схемах, используется команда:

ALTER ROLE *Имя_пользователя* SET SEARCH_PATH = *Имя_ схемы*1 [, *Имя_ схемы*2,…];

Например, для того, чтобы пользователю *Director* автоматически устанавливался порядок доступа к схемам *Director, PUBLIC* можно выполнить команду:

ALTER ROLE Director SET SEARCH_PATH TO Director, PUBLIC;

Приведём пример достижения одной из целей использования схем, а именно: ограничения доступа к таблицам со стороны разных пользователей на основе представлений.

Пусть в БД существует таблица:

```
CREATE TABLE Persons
      (Person_Id    INTEGER     PRIMARY KEY,
       Name         VARCHAR(30),
       Sex          CHAR(1),
       Birthday     DATE);
```

в которую внесём тестовую информацию:

```
INSERT INTO Persons VALUES(1,'Иванов','M','01/01/2000');
INSERT INTO Persons VALUES(2,'Петров','M','01/01/1990');
```

Необходимо пользователю *Director,* когда он выполняет запрос

```
SELECT * FROM Persons;
```

запретить получать информацию о людях, которым на текущий момент ещё нет исполнилось 18 лет.

Для этого создадим представление:

```
CREATE OR REPLACE VIEW Director.Persons AS
      SELECT * FROM Persons
            WHERE EXTRACT(Year FROM (Age(Birthday))) >= 18;
```

Снимем с пользователя *Director* привилегии доступу к таблице *Persons* из схемы *PUBLIC*

```
REVOKE SELECT ON Persons FROM Director;
```

Установим привилегии доступу к представлению *Persons* из схемы *Director*

```
GRANT SELECT ON Director.Persons TO Director;
```

Теперь пользователь, выполняя запрос

```
SELECT * FROM Persons;
```

получит лишь записи о людях, которым на текущий момент уже исполнилось 18 лет.

Управление представлениями

В СУБД ***PostgreSQL*** представления не являются обновляемыми автоматически.

Например, выполним над представлением *Director.Persons* операцию:

```
UPDATE Director.Persons SET Name = 'Сидоров' WHERE Person_Id = 2;
```

В результате операции будет получена ошибка:

```
ERROR:  cannot update a view
HINT:  You need an unconditional ON UPDATE DO INSTEAD rule.
```

Ошибка указывает на необходимость создания правила типа

… ON UPDATE DO INSTEAD …

Но существует механизм правил, который позволяет определять операции INSERT, UPDATE, DELETE над представлением.

Синтаксис операции создания правила:

CREATE RULE *название_правила* AS ON *событие* TO *название_таблицы* [WHERE *условие*]
DO [ALSO | INSTEAD]
NOTHING | SQL_*операция* | (SQL_*операция*1; SQL_*операция*2 ...);

Типы *события*: INSERT, UPDATE, DELETE.

Синтаксис описания *условия* правила эквивалентен синтаксису выражения WHERE стандартного *SQL*.

Если тип *события* правила является INSERT, тогда *INSERT*-операция, обрабатываемая правилом, выполняется перед выполнением *SQL*-операций из правила. Это позволяет самому правилу видеть результат выполнения операции. Но для событий типа UPDATE или DELETE операция выполняется уже после выполнения *SQL*-операций по правилам.

Параметр ALSO указывает правилу, которое обрабатывает операцию, на необходимость до неё выполнять дополнительно ещё *SQL*-операции, которые могут быть описаны в правиле.

Параметр INSTEAD указывает правилу, которое обрабатывает операцию, выполнять вместо этой операции *SQL*-операции, которые могут быть описаны в правиле. Поэтому правила по своему назначению похожи на триггеры, но могут ещё и заменять реальные операции в БД благодаря параметру INSTEAD.

SQL-операция может включать стандартные операции:

INSERT INTO … VALUES;

UPDATE …;

DELETE FROM …;

INSERT INTO … SELECT …

Дополнительно, как и в триггере, в качестве значений могут быть использованы структуры с контекстными переменными: NEW.*атрибут*, OLD.*атрибут* — значение атрибута после выполнения операции, или перед выполнением операции, соответственно. Как и в триггере, переменную NEW. можно использовать при работе операций INSERT и UPDATE, а переменную OLD. — при работе операций UPDATE и DELETE.

Например, для того, что исключить ошибку запроса в начале параграфа при выполнении операции UPDATE над представлением можно создать следующее правило:

```
CREATE RULE Persons_Update_Director AS ON UPDATE TO Director.Persons
    DO INSTEAD    UPDATE    Persons SET
                                    Person_Id =  NEW.Person_Id,
                                    Name  = NEW.Name,
                                    Sex  = NEW.Sex,
                                    Birthday = NEW.Birthday
                                WHERE Person_Id =  OLD.Person_Id;
```

Снимем с пользователя *Director* привилегии доступа к таблице *Persons* из схемы *PUBLIC*

```
REVOKE UPDATE ON Persons FROM Director;
```

Установим привилегии доступа к представлению *Persons* из схемы *Director*

```
GRANT UPDATE ON Director.Persons TO Director;
```

Теперь пользователь *Director* может успешно выполнить запрос:

```
UPDATE Director.Persons SET Name = 'Сидоров' WHERE Person_Id = 2;
```

В описании правила в *WHERE*-предложении указывается условие определения записи реальной таблицы, связанной с записью представления. Поэтому пользователь *Director* не сможет успешно выполнить запрос:

```
UPDATE Director.Persons SET Name = 'Сидоров' WHERE Person_Id = 1;
```

Это связано с тем, что запись с *Person_Id* = 1 не является видимой в представлении.

Для того, чтобы пользователь мог выполнять над представлением операции INSERT и DELETE можно создать правила:

```
DROP RULE Persons_Insert_Director ON Director.Persons;

CREATE RULE Persons_Insert_Director AS ON INSERT TO Director.Persons
    DO INSTEAD INSERT INTO Persons
        SELECT NEW.Person_Id,NEW.Name, NEW.Sex, NEW.Birthday;

REVOKE INSERT ON Persons FROM Director;

GRANT INSERT ON Director.Persons TO Director;

CREATE RULE Persons_Delete_Director AS ON DELETE TO Director.Persons
    DO INSTEAD DELETE FROM Persons WHERE Person_Id =  OLD.Person_Id;

REVOKE DELETE ON Persons FROM Director;

GRANT DELETE ON Director.Persons TO Director;
```

Если необходимо, чтобы для всех пользователей, кроме пользователя *Director*, игнорировалась *INSERT*-операция над представлением *Director.Persons* можно создать следующее правило:

```
CREATE OR REPLACE RULE Table_Insert_Director_Persons
    AS ON INSERT TO Director.Persons
        WHERE CURRENT_USER != 'Director' DO INSTEAD NOTHING;
```

Управление транзакциями

Транзакция является последовательностью операций, выполненных как одна логическая единица работы. Запускаются транзакции приложением пользователя и содержат либо команды ЯМД, выполняющие единое согласованное изменение данных, либо одну команду ЯОД или ЯУД. Таким образом, транзакции представляют собой своего рода сеанс взаимодействия приложения с БД.

Транзакции начинаются с выполнения первой исполняемой команды *SQL* и заканчиваются либо сохранением изменений в базе данных, либо отказом от сохранения (откатом).

Фактически, сущность транзакции состоит в связывании нескольких шагов в одну операцию по принципу „все-или-ничего“. Внутренние промежуточные состояния между шагами не видны для других конкурирующих транзакций и, если во время выполнения транзакции случится ошибка, которая помешает транзакции, завершится, то в БД никаких изменений сделано не будет.

В стандарте *SQL* сохранение изменений и откат выполняются соответственно командами *COMMIT* и *ROLLBACK*. Но использовать их непосредственно при „обычной“ работе необходимости нет: транзакция начинается и заканчивается вместе с соответствующим запросом приложения.

Однако в различных ПрО есть задачи, для решения которых необходимо выполнение нескольких запросов. Причём результаты их работы либо должны быть сохранены все, либо, в случае каких-либо сбоев (связи или аппаратуры) не сохранены вообще.

В этом случае транзакцию необходимо объявлять и заканчивать явно с помощью соответствующих команд ЯУД.

Рассмотрим несколько примеров таких задач, приведенных в документации к СУБД *PostgreSQL*.

Пусть в БД содержатся балансы для нескольких клиентов и общие балансы для филиалов. Предположим, что необходимо перевести 100 д.е. от клиента с кодом 1111111111 клиенту с кодом 2222222222. Последовательность запросов, реализующих данную операцию, может выглядеть так:

```
UPDATE accounts SET balance = balance - 100.00
       WHERE ClientID = 1111111111;
UPDATE branches SET balance = balance - 100.00
       WHERE ClientID = (SELECT branch_name FROM accounts
                         WHERE ClientID = 1111111111);
UPDATE accounts SET balance = balance + 100.00
       WHERE ClientID = 2222222222;
UPDATE branches SET balance = balance + 100.00
       WHERE ClientID = (SELECT branch_name FROM accounts
                         WHERE ClientID = 2222222222);
```

Если не принять меры, то в результате какого-либо системного сбоя может получиться одна из ошибок целостности:

– у первого клиента будет вычтена данная сумма, но второй её не получит;
– второй клиент получит деньги, которые не будут вычтены у первого;
– баланс клиента будет уменьшен (увеличен), но баланс его филиала не изменится.

Для того чтобы избежать этих ошибок, достаточно все данные запросы объединить в одну транзакцию. В СУБД *PostgreSQL* транзакция начинается командой *BEGIN* и завершается командой *END* или *COMMIT*:

```
BEGIN
последовательность команд SQL
COMMIT
```

Но даже в пределах таких транзакций остаётся возможность управлять изменениями на ещё более детализированном уровне с помощью „точек сохранения“ (*savepoints*). Точки

сохранения позволяют выборочно отбрасывать части транзакции, в то же время, выполняя остаток транзакции.

После задания точки сохранения с помощью оператора *SAVEPOINT*, при необходимости можно откатить транзакцию до этой точки сохранения с помощью оператора *ROLLBACK*. Все изменения БД внутри транзакции между точкой сохранения и местом, откуда вызван откат, теряются, но изменения, которые были сделаны до точки сохранения, остаются.

После отката к точке сохранения, она продолжает оставаться заданной и, таким образом, можно делать к ней откат несколько раз. И наоборот, если появилась уверенность, что откат к определённой точке сохранения не понадобится снова, она может быть убрана, чтобы система могла освободить некоторые ресурсы. При этом необходимо учитывать, что откат к некоторой точке сохранения или её удаление, автоматически удаляет все точки сохранения, которые были заданы после неё.

Например, пусть в транзакции из предыдущей задачи после списания средств со счёта первого клиента и зачисления их на счёт второго обнаружилось, что эти средства должны были зачислить на счёт клиента с кодом 33333333333. Вместо выполнения целого ряда дополнительных запросов для исправления такой ситуации можно было бы воспользоваться точками сохранения:

```
BEGIN;
UPDATE  accounts   SET  balance = balance - 100.00
          WHERE    ClientID = 1111111111;
SAVEPOINT          savepoint_1;
UPDATE  accounts   SET  balance = balance + 100.00
          WHERE    ClientID = 2222222222;
ROLLBACK TO        savepoint_1;
UPDATE  accounts   SET  balance = balance + 100.00
          WHERE    ClientID = 33333333333;
COMMIT;
```

Другое важное свойство транзакционных СУБД состоит в строгой изоляции транзакций: несколько транзакций запускаются конкурентно, и каждая из них не видит тех „незавершённых“ изменений, которые производят другие транзакции. То есть, во время выполнения транзакций другие пользователи (приложения) могут пытаться прочитать и/или внести изменения в те же самые объекты БД и генерировать для этого свои транзакции, вызывая тем самым конфликт или коллизию доступа к БД.

Например, если одна транзакция занята сложением всех балансов филиалов, она не должна учитывать как денег снятых со счета одного клиента, так и денег пришедших на счёт другого. Таким образом, транзакции должны выполнять принцип „все-или-ничего“ не только в плане нерушимости тех изменений, которые они производят в БД, но и также в плане того, что они видят в момент работы. Обновления, которые вносит открытая транзакция, являются невидимыми для других транзакций пока данная транзакция не завершиться, после чего все внесённые ею изменения станут видимыми.

Это обеспечивается с помощью механизма так называемых *блокировок*.

Блокировка

Блокировки **обеспечивают целостность данных** в многопользовательских (параллельных) системах.

О проблемах, которые возникают в таких системах сказано выше: они связаны с *конфликтами* при обработке *нескольких параллельных* транзакций.

В *SQL* решение таких конфликтов основано на одном з методов управления параллелизмом — **методе блокировки**. Его идея очень проста: в случае, когда для выполнения некоторой транзакции необходимо, чтобы какой-нибудь объект (например, кортеж) не изменялся *непредвиденно и без ведома* этой транзакции, то такой объект **блокируется**, точнее **блокируется доступ** к нему **со стороны других транзакций**.

В *SQL* (в ЯУД) существует два типа блокировок: **монопольная** (**исключительная**) блокировка и **разделяемая**, которые иногда называют блокировками записи и чтения соответственно (***X-блокировка*** и ***S-блокировка***). Различие между этими типами определяется следующими правилами:

1. Если транзакция *A* блокирует кортеж ***p*** *X-блокировкой*, то **запрос** другой транзакции *B* с блокировкой этого кортежа **будет отклонён**. *X-блокировка* держится до конца транзакции.
2. Если транзакция *A* блокирует кортеж ***p*** *S-блокировкой*, то
   а) **запрос** транзакции *B* на *X-блокировку* этого же кортежа **будет отклонён**;
   б) **запрос** транзакции *B* на *S-блокировку* кортежа **будет принят**.

Протокол доступа к данным при использовании блокировок имеет следующий вид:

- Прежде чем прочитать объект, транзакция должна наложить на этот объект *S-блокировку*.
- Прежде чем обновить объект, транзакция должна наложить на этот объект *X-блокировку*. Если транзакция уже заблокировала объект *S-блокировкой* (для чтения), то перед обновлением объекта *S-блокировка* должна быть заменено *X-блокировкой*.
- Если блокировка объекта транзакцией *B* отвергается поэтому, что объект уже заблокирован транзакцией *A*, то транзакция *B* переходит в ***состояние ожидания***. Транзакция *B* будет находиться в состоянии ожидания до тех пор, пока транзакция *A* не снимет блокировки объекта.
- *X-блокировки*, наложенные транзакцией *A*, сохраняются до конца транзакции A.

В ***PostgreSQL*** реализована поддержка множества типов блокировок (*lockmode*), которые делятся на *явные* и *неявные*. *Явные блокировки* – это те, которые выполнены в запросе с помощью ключевого слова *LOCK* (уровень таблиц) и модификаторов запросов *FOR UPDATE* или *FOR SHARE* (уровень кортежей), то есть указаны пользователем. *Неявные блокировки* – это те, которые реализуются при выполнении различных запросов (*SELECT, UPDATE, INSERT, ALTER* и прочие). *PostgreSQL* поддерживает ещё отдельный вид блокировок, называемых *рекомендательными* (*ADVISORY LOCK*).

Блокировки *уровня таблицы* реализуются с помощью команды

LOCK TABLE *таблица*[, *таблица*[, ...]] [IN *блокировка* MODE] [NOWAIT]

Ниже приведен перечень различных типов блокировок таблиц с указанием конфликтов, возникающих в случае, если таблице уже назначен какой-либо тип блокировки другой транзакцией.

Требуемая блокировка таблицы	Текущая блокировка таблицы							
	ACCESS SHARE	**ROW SHARE**	**ROW EXCLUSIVE**	**SHARE UPDATE EXCLUSIVE**	**SHARE**	**SHARE ROW EXCLUSIVE**	**EXCLUSIVE**	**ACCESS EXCLUSIVE**
ACCESS SHARE								X
ROW SHARE							X	X
ROW EXCLUSIVE					X	X	X	X
SHARE UPDATE EXCLUSIVE				X	X	X	X	X
SHARE			X	X		X	X	X
SHARE ROW EXCLUSIVE			X	X	X	X	X	X
EXCLUSIVE		X	X	X	X	X	X	X
ACCESS EXCLUSIVE	X	X	X	X	X	X	X	X

С режимом *NOWAIT* транзакция не ожидает реализации конфликта с какой-либо блокировкой, а немедленно завершается.

Блокировки *уровня кортежа* не влияют на возможность чтения данных запросами других транзакций, а блокируют только возможность записи или блокировки текущей строки. Для реализации блокировки строки в *SELECT*-выражение необходимо добавить один из операторов *FOR KEY SHARE*, *FOR SHARE*, *FOR NO KEY UPDATE*, *FOR UPDATE* (перечислены по степени увеличения накладываемых ограничений доступа):

SELECT *список_полей* FROM *таблица* WHERE *условие* FOR UPDATE;

Наиболее используемый из них – *FOR UPDATE*. При назначении этого типа блокировки кортежи, возвращённые командой *SELECT*, будут заблокированы для модификации со стороны других транзакций до момента завершения текущей.

При использовании блокировок вводится такое *свойство транзакции* как **уровень изоляции**, определяющий **степень вмешательства других параллельных транзакций** в её работу. ***Изоляция*** — это свойство, которое определяет как/когда изменения, сделанные одной операцией, будут видны конкурентной операции.

В разных системах может быть от *двух до пяти уровней изоляции*. Стандарт *SQL* определяет четыре уровня изоляции:

READ UNCOMMITTED	Чтение объектов даже незавершённых транзакции;
READ COMMITTED	Чтение объектов только после завершения транзакции;
REPEATABLE READ	Повторяемое чтение;
SERIALIZABLE	Все транзакции с этим уровнем изоляции не выполняются последовательно. При сохранении результатов (*COMMIT*) такой транзакции проверяется наличие конфликта при изменении данных и в случае, если данные уже были изменены конкурентной транзакцией, текущая транзакция завершается ошибкой.

То есть приложение, выполняющее эту транзакцию, узнает об успешности только после её завершения. В случае неудачи необходимо будет повторять все действия и вычисления до тех пор, пока транзакция не совершится успешно, либо не будет принято решение об отказе от выполнения действия.

Этот уровень позволяет решить проблему „конкурентоспособности“ транзакций. То есть ситуацию, когда при параллельном доступе к объекту приоритет отдаётся той транзакции, которая раньше начала его модификацию. Проблемы такого вида хорошо известны в параллельном программировании и носят название *состязания* (*race conditions*).

Все СУБД обязательно поддерживают уровень **повторного чтения**, который гарантирует, что **в рамках этой транзакции все записи, принимающие участие в запросе, не изменяются**. Если быть более точным, то некоторые СУБД разделяют этот уровень на 2: **недопустимость** изменений и **невидимость** изменений.

Рассмотрим команду задания блокировки и уровня изоляции из **стандарта *SQL*** на примере СУБД *Interbase*.

```
SET TRANSACTION
    [READ WRITE | READ ONLY]
    [WAIT | NO WAIT]
    [[ISOLATION LEVEL]
        {SNAPSHOT [TABLE STABILITY] |
        READ COMMITTED
        [[NO] RECORD_VERSION]}];
```

Параметр	**Описание**
READ WRITE	Определяет, что транзакция может читать и записывать в таблицы (по умолчанию).
READ ONLY	Определяет, что транзакция может только читать таблицы.
WAIT	Определяет, что транзакция ожидает доступа (поступает в очередь), если она вызывает конфликт блокировки с другой транзакцией (по умолчанию).
NO WAIT	Определяет, что транзакция немедленно возвращает ошибку, если она сталкивается с конфликтом блокировки (без ожидания).
ISOLATION LEVEL	Определяет уровень изоляции для этой транзакции при попытке других параллельных транзакций обратиться к тем же самым таблицам.
SNAPSHOT	(по умолчанию) обеспечивает повторяемое чтение (*repeatable-read*) БД в момент старта транзакции. Изменения, сделанные другими параллельными транзакциями, невидимы.
SNAPSHOT TABLE STABILITY	обеспечивает повторяемое чтение БД, гарантируя, что транзакции не могут записывать в таблицы, но могут читать из них.

Параметр	Описание
READ COMMITTED	даёт возможность транзакции по умолчанию видеть самые последние завершённые (*committed*) изменения, сделанные другими параллельными транзакциями. Они могут также модифицировать строки до тех пор, пока не произойдёт конфликт модификации. Незавершённые (uncommitted) изменения, сделанные другими транзакциями, остаются невидимыми до выполнения команды COMMIT.
NO RECORD_VERSION	По умолчанию, читает только последнюю версию строки. Если определена опция WAIT блокировки, то транзакция ждёт, пока последняя версия строки не будет завершена (*committed*) или отменена (*rolled back*), и повторяет чтение.
RECORD_VERSION	Читает последнюю завершённую версию строки, даже если более новая незавершённая версия также размещена на диске.

В ***PostgreSQL***, начиная с версии 8.0, реализована поддержка всех 4-х уровней изоляции, определённых стандартом *SQL*: *READ UNCOMMITTED*, *READ COMMITTED* (используется по умолчанию), *REPEATABLE READ*, *SERIALIZABLE*. В последних версиях этой СУБД уровень изоляции рекомендуется задавать в команде *BEGIN*.

BEGIN ISOLATION LEVEL *уровень_изоляции*;

…

COMMIT;

Кроме того, для совместимости со старыми версиями, другими СУБД и стандартом *SQL* допускается использование команды *SET TRANSACTION*, имеющей практически аналогичный формат.

Сжатие базы данных

При удалении или обновлении записи ***только помечаются*** как удалённые, но физически не удаляются. По этой причине физическое дисковое пространство со временем иссякает, что приводит к существенному снижению производительности или полной остановке.

Для восстановления памяти, занятой „мёртвыми“ кортежами, в *SQL* существует команда сжатия БД VACUUM – операция сборки мусора и, опционально, анализа БД.

VACUUM [*опция*] [*имя_таблицы* [(*имя_поля* [, ...])]]

Без указания *имени таблицы* операция будет проводиться со всеми таблицами БД.

Опции:

– Без опции (*обычное сжатие*) команда удалит все „мёртвые“ записи таблиц(-ы), данные, оставшиеся в результате отмены транзакций, а также неиспользуемые временные данные.

Этот вариант сжатия просто восстанавливает пространство и делает его доступным для повторного использования. Команда в таком формате может работать параллельно с обычным чтением и записью в таблице, так как эксклюзивная блокировка при этом не устанавливается. Однако дополнительное пространство, в большинстве случаев, операционной системе не возвращается, а просто становится доступными для повторного использования в той же таблице.

- С опцией *ANALYZE* будет выполнен анализ данных во всех полях таблицы (или только в указанной колонке таблицы). Информация в дальнейшем будет использована оптимизатором запросов для построения более эффективного плана их выполнения.
- С опцией *FULL* (*полное сжатие*) команда переписывает все содержимое таблицы в новый файл на диске, без „мёртвого“ пространства, что позволяет неиспользуемое пространство вернуть операционной системе. Этот вариант сжатия гораздо медленнее и требует, во-первых, дополнительного дискового пространства, т.к. он не освобождает старую копию таблицы, пока операция записи новой копии не будет завершена, а во-вторых, эксклюзивной блокировки для каждой обрабатываемой таблицы. Т.е. при использовании опции *FULL* таблица будет недоступна до тех пор, пока команда *VACUUM* не закончит своё выполнение.
- Указание опции *FREEZE* эквивалентно выполнению команды *VACUUM* с параметром *vacuum_freeze_min_age* равным нулю в конфигурации. Это, так называемое, агрессивное „замораживание“ записей.

  Дело в том, что вместе с данными в базе хранится и номер транзакции, в рамках которой были созданы/изменены эти данные. Так реализуется версионность БД. По этому номеру проверяется, видны ли эти данные или нет. Но поскольку количество номеров транзакций ограничено, нужно что-то делать, когда номер транзакции переполняется. Для решения этой проблемы и предназначена опция FREEZE, которая изменяет все номера транзакций старше *vacuum_freeze_min_age* в файлах данных на специальное значение *FrozenXID*. Это значение всегда меньше любого номера транзакции, таким образом, когда после сжатия счётчик транзакций начнётся сначала, данные с номером транзакции *FrozenXID* всё равно будут видны.

Для автоматизации в *PostgreSQL* есть опция автоматического сжатия БД – *AUTOVACUUM*, настройки которого записаны в файле конфигурации БД *postgresql.conf*.

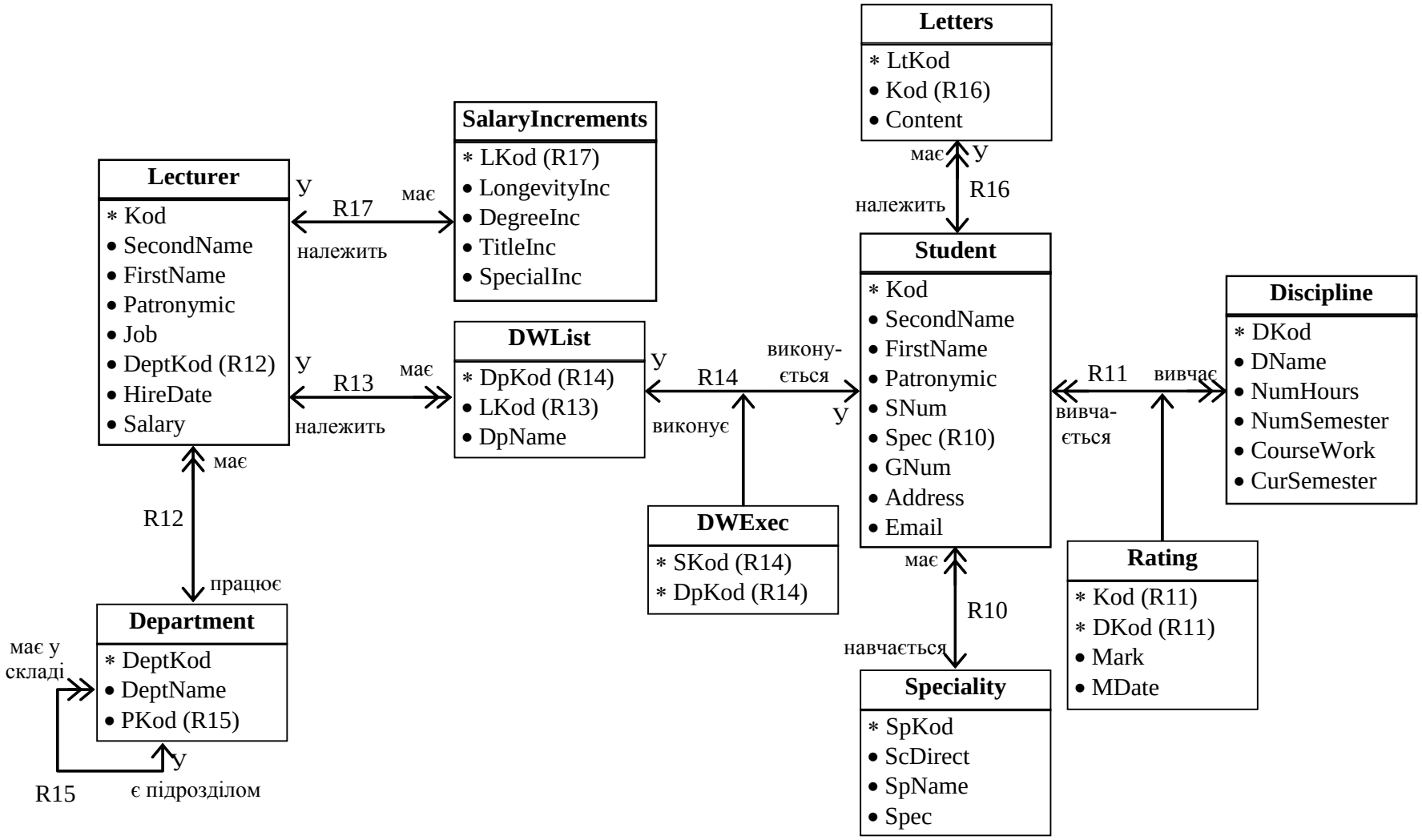
На запуск сжатия в автоматическом режиме влияют следующие параметры конфигурации:

Параметр	**Описание**
autovacuum = on	активность автоматического сжатия
log_autovacuum_min_duration = -1	журналирование автоматического сжатия: -1 – отключено; 0 – регистрирует все действия и их продолжительность; > 0 – регистрирует только запущенные действия, по крайней мере, указанное число миллисекунд
autovacuum_max_workers = 3	максимальное количество процессов автоматического сжатия
autovacuum_naptime = 1min	задержка между запусками сжатия
autovacuum_vacuum_threshold = 50	минимальное число изменённых или удалённых записей таблицы для того, чтобы было запущено сжатие
autovacuum_vacuum_scale_factor = 0.2	доля изменённых или удалённых записей таблицы для того, чтобы было запущено сжатие
autovacuum_freeze_max_age = 200000000	максимальный возраст таблицы (в транзакциях), при превышении которого будет запущено сжатие

Приложение А
Схема системы баз данных к разделу „SQL“*

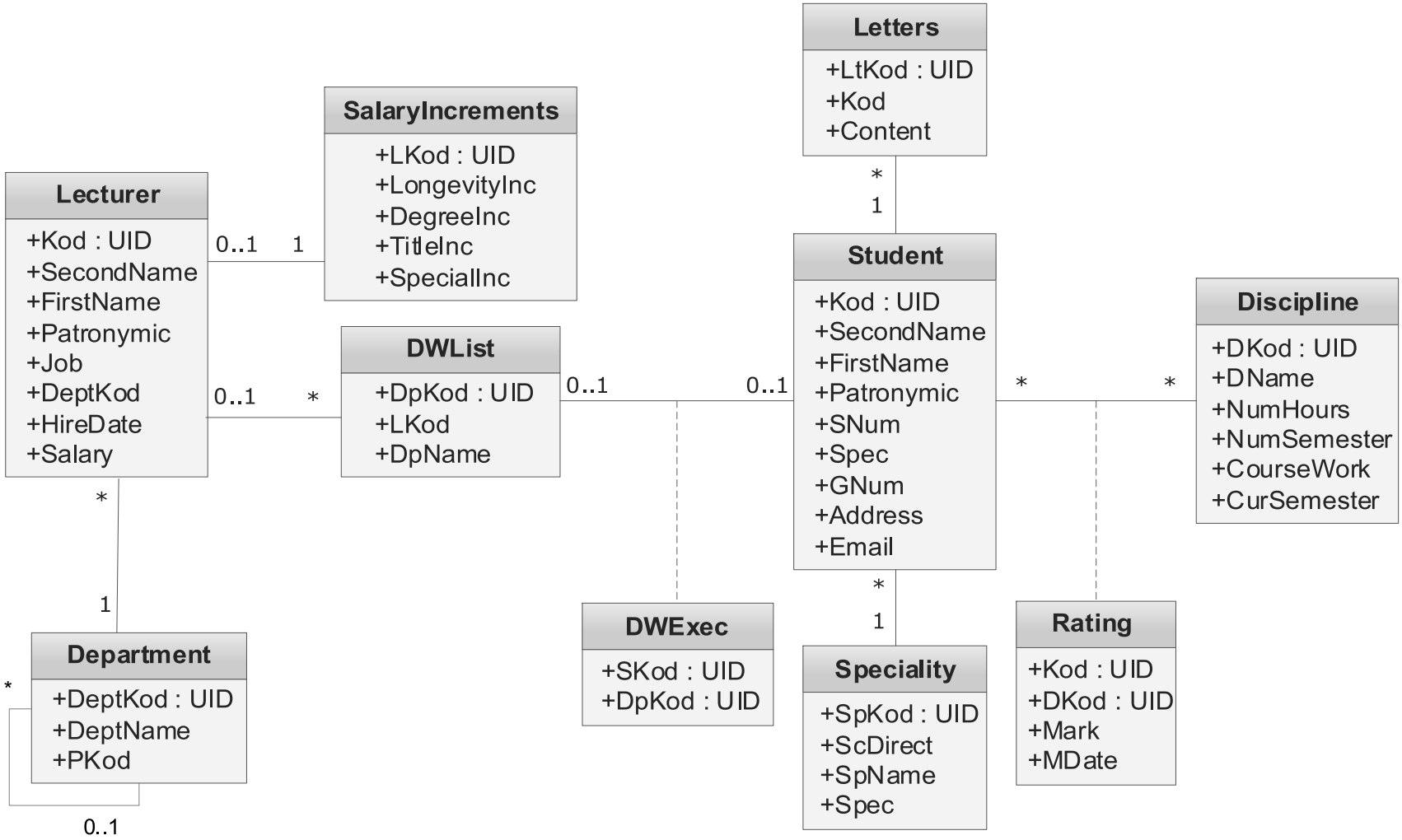
* Все диаграммы начерчены с помощью редактора ConceptDraw Pro ®

Объектно-реляционная нотация**



** Соответствует объектно-ориентированной нотации Шлеер, которая может быть полностью согласована с реляционной моделью данных

Нотация UML***



*** Нотация UML запрещает явную идентификацию экземпляров объектов, однако для согласования с реляционной моделью данных на схеме указан первичный ключ каждой сущности (UID – Unique IDentificator)

Приложение Б
Пример сценария для построения таблиц к разделу „SQL“

```
/* Потенциально возможное соединение с сервером базы данных */
    [CONNECT   c:\work\SQL\lectures      USER MEV      PASSWORD <password>;]

/* TYPE: ADDRESS, Owner: MEV (Город, Улица, Дом, Квартира) */
    CREATE TYPE      Address      AS
        (City          VARCHAR,
         Street        VARCHAR,
         House         SMALLINT,
         Flat          SMALLINT);

/* DOMAIN: POST, Owner: MEV (Должность) */
    CREATE DOMAIN    Post      CHAR(9)
        CONSTRAINT Valid_Posts
         CHECK(VALUE IN('Зав. каф.','Проф.','Доц.','Ст. преп.', 'Преп.','Ассист.'));

/* SEQUENCE: letters_kod, Owner: MEV */
    CREATE SEQUENCE letters_kod;

/* Table: DEPARTMENT, Owner: MEV (Код_Подразделения, Название_Подразделения,
Код_Подразделения_которому_Подчиняется) */
    CREATE TABLE     Department
        (DeptKod    INTEGER   NOT NULL   CHECK(DeptKod>0)  PRIMARY KEY,
         DeptName   CHAR(40)  NOT NULL   UNIQUE,
         PKod       INTEGER   REFERENCES Department(DeptKod));

/* Table: LECTURER, Owner: MEV (Код_Преподавателя, Фамилия, Имя, Отчество,
Должность, Код_Подразделения, Дата_Приема_на_Должность, Заработная_Плата) */
    CREATE TABLE     Lecturer
        (Kod           INTEGER   NOT NULL     CHECK(Kod>0)   PRIMARY KEY,
         SecondName    CHAR(30)  NOT NULL,
         FirstName     CHAR(20)  NOT NULL,
         Patronymic    CHAR(30)  NOT NULL,
         Job           Post      NOT NULL,
         DeptKod       INTEGER   REFERENCES Department(DeptKod),
         HireDate      DATE,
         Salary        NUMERIC(5));

/* Table: SALARYINCREMENTS, Owner: MEV (Код _Преподавателя, Надбавка_за_Стаж,
```

Надбавка_за_Научную_Степень, Надбавка_за_Ученое_Звание, Специальная_Надбавка)*/

```
CREATE TABLE    SalaryIncrements
    (LKod       INTEGER   NOT NULL    CHECK(LKod>0)
                PRIMARY KEY   REFERENCES Lecturer(Kod),
     LongevityInc  NUMERIC(5)    DEFAULT 0,
     DegreeInc     NUMERIC(5)    DEFAULT 0,
     TitleInc      NUMERIC(5)    DEFAULT 0,
     SpecialInc    NUMERIC(5)    DEFAULT 0);
```

/* Table: SPECIALITY, Owner: MEV (Код_Специальности, Название_Направления, Название_Специальности, Шифр_Специальности) */

```
CREATE TABLE    Speciality
    (SpKod      CHAR(10)    NOT NULL      PRIMARY KEY
                            CHECK(SpKod LIKE '_._____'),
     ScDirect   CHAR(25)    NOT NULL,
     SpName     CHAR(40)    NOT NULL      UNIQUE,
     Spec       CHAR(2)     NOT NULL      UNIQUE
                CHECK(Spec IN('ОИ','ОМ','ОС','ОП','АІ','АС','АК','АП')));
```

/* Table: STUDENT, Owner: MEV (Код_Студента, Фамилия, Имя, Отчество, Порядковый_Номер, Шифр_Специальности, Номер_ Группы, Адрес_Проживания, Адрес_Электронной_Почты) */

```
CREATE TABLE    Student
    (Kod        INTEGER     NOT NULL   CHECK(Kod>0)   PRIMARY KEY,
     SecondName CHAR(30)    NOT NULL,
     FirstName  CHAR(20)    NOT NULL,
     Patronymic CHAR(30),
     SNum       SMALLINT    NOT NULL,
     Spec       CHAR(2)     NOT NULL
                CHECK(Spec IN('ОИ','ОМ','ОС','ОП','АІ','АС','АК','АП'))
                REFERENCES Speciality (Spec),
     GNum       INTEGER     NOT NULL,
     Address    Address,
     Email      CHAR(30),
     UNIQUE(Spec, GNum, SNum));
```

/* Table: DWLIST, Owner: MEV (Код_Дипломной_Работы, Тема, Код_Преподавателя) */

```
CREATE TABLE    Dwlist
    (DpKod   INTEGER          NOT NULL
                              CHECK(DpKod>0)    PRIMARY KEY,
     DpName  VARCHAR(100)     UNIQUE,
     LKod    INTEGER          REFERENCES Lecturer(Kod));
```

```
/* Table: DWEXEC, Owner: MEV (Код_Студента, Код_Дипломной_Работы) */

    CREATE TABLE    DWExec
        (SKod       INTEGER     REFERENCES Student(Kod),
         DpKod      INTEGER     REFERENCES Dwlist (DpKod)       UNIQUE,
         PRIMARY KEY (SKod, DpKod));


/* Table: LETTERS, Owner: MEV (Код_Письма, Код_студента, Содержание_Переписки) */

    CREATE TABLE    Letters
        (Ltkod      INTEGER     NOT NULL    CHECK(Ltkod>0)      PRIMARY KEY
                                DEFAULT NEXTVAL('letters_kod'),
         Kod        INTEGER     REFERENCES Student (Kod),
         Content    VARCHAR);


/* Table: DISCIPLINE, Owner: MEV (Код_ Дисциплины, Название_ Дисциплины,
Количество_Часов, Количество_Семестров, Курсовая_Работа, Текущий_Семестр) */

    CREATE TABLE    Discipline
        (DKod           INTEGER     NOT NULL    CHECK(DKod>0)     PRIMARY KEY,
         DName          CHAR(30),
         Numhours       INTEGER,
         Numsemester    INTEGER,
         Coursework     BOOLEAN,
         Cursemester    INTEGER);


/* Table: RATING, Owner: MEV (Код_Студента, Код_ Дисциплины, Оценка, Дата_Модуля) */

    CREATE TABLE    Rating
        (Kod        INTEGER     NOT NULL        REFERENCES Student(Kod),
         DKod       INTEGER     REFERENCES Discipline(DKod),
         Mark       INTEGER     DEFAULT 0
                                CHECK (Mark BETWEEN 0 AND 100),
         MDate      DATE,
         PRIMARY KEY (Kod, DKod));


    EXIT;
```

Приложение В
Примеры таблиц к разделу „SQL“

SELECT * FROM Department;

DeptKod	DeptName	PKod
1	ИБЭИТ	
2	ЭКИТ	1
3	УАА	1
4	ИКС	
5	СПО	4
6	ИС	4

SELECT * FROM Lecturer;

Kod	SecondName	FirstName	Patronymic	Job	DeptKod	HireDate	Salary
1	Малахов	Евгений	Валериевич	Зав.каф	2	1990-09-01	1500
2	Востров	Георгий	Николаевич	Доц.	4	1982-09-01	1100
3	Чугунов	Анатолий	Анатолиевич	Доц.	2	1985-09-01	1250
4	Погорецкая	Валентина	Яковлевна	Доц.	2	1980-09-01	1000
5	Юхименко	Бируте	Ионовна	Проф.	3	1982-09-01	1300

SELECT * FROM SalaryIncrements;

LKod	LongevityInc	DegreeInc	TitleInc	SpecialInc
1	300	495	375	675
2	220	275	165	330
3	250	313	188	125
4	200	250	150	390
5	260	429	325	520

SELECT * FROM Speciality;

SpKod	ScDirect	SpName	Spec
8.04030101	Прикладная математика	Прикладная математика	ОС
8.03050201	Экономика	Экономическая кибернетика	ОИ
8.03060102	Менеджмент	Менеджмент	ОМ
8.03050401	Экономика	Экономика предприятий	ОП
8.05010202	Компьютерная инженерия	Специализированные компьютерные системы	АК
8.05010301	Программная инженерия	Программное обеспечение систем	АС

SELECT * FROM Student;

Kod	SecondName	FirstName	Patronymic	SNum	Spec	GNum	Address	Email
1	Бровков	Владимир	Георгиевич	3	ОМ	971	(" г. Одесса"," ул. Неизвестного",5,1)	bvgo12345@ukr.net
2	Арсирий	Елена	Александровна	17	ОИ	943	(" г. Одесса"," ул. Вторая",3,1)	arsiriy@mail.ru
3	Любченко	Вера	Викторовна	3	ОС	102	(" г. Одесса"," ул. 1-го мая",5,1)	lubchenko@te.net.ua
4	Филатова	Татьяна	Вячеславовна	15	ОИ	943	(" г. Одесса"," ул. Неизвестного",15,1)	filatova@mail.ru
5	Журан	Елена	Анатолиевна	4	ОП	971	(" г. Одесса"," ул. Обычная",45,4)	zhuran@ukr.net
6	Микулинская	Мария	Геннадиевна	5	ОП	970	(" г. Одесса"," ул. Розовая",67,8)	mmg12345@te.net.ua
7	Блажко	Александр	Анатолиевич	12	АС	954	(" г. Одесса"," ул. Местная",2,4)	blazko@ukr.net
8	Филиппова	Светлана	Валериевна	11	ОМ	951	(" г. Одесса"," ул. Крайняя",56,9)	filyppova@mail.ru

SELECT * FROM Dwlist;

DpKod	DpName	LKod
1	Тема 1	4
2	Тема 2	4
3	Тема 3	4
4	Тема 4	5
5	Тема 5	5
6	Тема 6	2
7	Тема 7	2
8	Тема 8	3
9	Тема 9	1
10	Тема 10	1

SELECT * FROM DWExec;

SKod	DpKod
1	9
2	1
3	3
4	2
5	5
6	10

SELECT * FROM Letters;

Ltkod	Kod	Content
1	2	В задаче №4 ответ 12.24
2	5	Отъезд на конференцию состоится в 8:00 am
3	3	Сбор возле актового зала в 17:00
4	1	Позвоните мне по телефону номер (777) 777-77-77
5	8	Мой контактный телефон +38 (050) 111-11-11
6	7	Дайте ответ по адресу opu@opu.ua

SELECT * FROM Discipline;

DKod	DName	Numhours	Numsemester	Coursework	Cursemester
1	Технологии проектирования БД	72	1	TRUE	7
2	Сетевые технологии	72	1	TRUE	7
3	Экономическая кибернетика	36	3	FALSE	1
4	Экономика	36	1	FALSE	5
5	Исследование операций	72	1	FALSE	7

SELECT * FROM Rating;

Kod	DKod	Mark	MDate
1	1	82	10-10-2011
1	2	10	12-10-2011
1	5	75	11-10-2011
2	1	90	10-10-2011
2	2	76	12-10-2011
2	5	20	11-10-2011
3	1	46	11-10-2011
3	2	85	12-10-2011
3	5	100	11-10-2011
4	1	54	10-10-2011
4	2	85	12-12-2011
4	5	65	11-10-2011
5	1	86	10-10-2011
6	3	30	14-10-2011
7	4	97	13-10-2011
8	1	0	10-10-2011

Список литературы

1. Вирт Н. Алгоритмы и структуры данных: Пер. с англ. – М.: Мир, 1989. – 360 с.
2. Грабер М. Введение в SQL: Пер. с англ. – М.: ЛОРИ, 1996 – 382 с.
3. Грофф Дж., Вайнберг П. SQL: Полное руководство: Пер. с англ. – 2-е изд., перераб. и доп. – К.: Издательская группа BHV, 2001. - 816 с.
4. Дейт К. Дж. Введение в системы баз данных: Пер. с англ. – 7-е изд. - М.: Издательский дом „Вильямс“, 2001. - 1072 с.Джексон Г. Проектирование реляционных баз данных для использования с микроЭВМ: Пер. с англ. – М.: Мир, 1991. - 252 с.
5. Диго С. М. Проектирование и использование баз данных: Учеб. - М.: Финансы и статистика, 1995. - 191 с.
6. Иванов А. Е. Виртуальная реальность // История философии. Энциклопедия. - Минск, 2002. - С. 183 – 186.
7. Иванов Ю. Н. Теория информационных объектов и СУБД. - М.: Наука, 1988. - 232 с.
8. Кармен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. – М.: МЦНМО, 2000. – 960 с.
9. Коннолли Т., Бегг К., Страчан А. Базы данных: проектирование, реализация и сопровождение. Теория и практика: Пер. с англ. – 2-е изд. – М.: Издательский дом „Вильямс“, 2001. - 1120 с.
10. Когаловский М. Р. Перспективные технологии информационных систем. - М.: ДМК Пресс, 2003. - 288 с.
11. Коренев В. В., Гареев А. Ф., Васютин С. В., Райх В. В. Базы данных. Интеллектуальная обработка информации. - М.: „Нолидж“, 2000.
12. Молинаро Э. SQL. Сборник рецептов. - Пер. с англ. – Спб: Символ-Плюс, 2009. - 672 с., ил.
13. Нагао М. и др. Структуры и базы данных: Пер. с японск. – М.: Мир, 1986. – 197 с.
14. Озкарахан Э. Машины баз данных и управление базами данных: Пер. с англ. – М.: Мир, 1989. - 696 с.
15. Романов В. П. Интеллектуальные информационные системы в экономике. - М.: Изд. Экзамен, 2003 г.
16. Словарь по кибернетике / Под ред. В.С. Михалевича. - 2-е изд. - К.: Гл. ред. УСЭ им. М. П. Бажана, 1989. - 751 с.
17. Спирли Э. Корпоративные хранилища данных. Планирование, разработка, реализация: Пер. с англ. – М.: Издательский дом „Вильямс“, 2001. - Т. 1. - 400 с.
18. Уорсли Дж., Дрейк Дж. PostgreSQL. Для профессионалов (+ CD). - Спб.: Питер, 2003. - 496 с: ил.
19. Цаленко М. Ш. Моделирование семантики у баз данных. - М.: Наука, 1989. - 287 с.
20. Цикритзис Д., Лоховски Ф. Модели данных: Пер. с англ. – М.: Финансы и статистика, 1985. - 344 с.
21. Шлеер С. Объектно-Ориентированный анализ: моделирование мира в состояниях. / С. Шлеер, С. Меллор – К.: Диалектика, 1993. - 216 с.
22. Blum R. PostgreSQL 8 for Windows. - Mcgraw-hill Companies, 2007. - 402 p.