Краткий курс по проектированию баз данных

Оглавление

Тема 1.1. История развития и причины появления СУБД

База данных (БД) представляет собой совокупность специальным образом организованных данных, хранимых в памяти вычислительной системы и отображающих состояние объектов и их взаимосвязей в рассматриваемой предметной области.

Система управления базой данных (СУБД) представляет собой совокупность программ, с помощью которых осуществляются управление структурой базы данных и контроль доступа к данным, хранящимся в ней. СУБД позволяет нескольким приложениям или пользователям осуществлять совместный доступ к данным.

Хорошая база данных не получается случайно, структура ее содержимого должна тщательно прорабатываться.

Правильно спроектированная БД облегчает управление данными и становится ценным поставщиком информации. Плохо спроектированная БД вероятнее всего станет накопителем избыточной информации, т. е. неоправданного дублирования данных. Избыточность, как правило, затрудняет выявление ошибок в данных.
База данных содержит избыточные данные, если одна и та же информация об одном и том же логическом объекте хранится в разных местах (логическим объектом, *сущностью* — entity — считается персона, местоположение или предмет, сведения о которых подлежат сбору и хранению). Например, избыточность данных имеет место, если номер телефона клиента хранится и в файле клиентов, и в файле коммивояжеров, и в файле счетов-фактур. Избыточность данных может стать источником несовместимости данных, когда трудно определить, какая информация является правильной, а какая нет (например, если телефон клиента был заменен новым только в файле счетов, а в других остался прежним). В отчетах, выполненных на основе такой информации, могут содержаться противоречивые сведения в зависимости от того, какая из версий данных была использована. Неконтролируемая избыточность данных — типичный признак плохо спроектированной БД.

Исторически сложилось так, что первые компьютерные приложения предназначались для решения организационных задач: обработка заказов и поставок, составление платежных ведомостей, разработка графиков выполнения работ и т. д. Такие приложения получали информацию из файлов, хранящихся на компьютере. Запросы следовали один за другим (сколько было продано, кем и кому?), а отчеты создавались с целью преобразования хранимых в компьютере данных к виду, удобному для руководства.

В недавнем прошлом руководители практически любого малого предприятия обрабатывали (а некоторые обрабатывают и по сей день) важную информацию с помощью картотек. Такие картотеки традиционно представляли собой коллекцию папок с документами, каждая из которых помечалась и хранилась в картотечном ящике. Аналогичную структуру имела картотека библиотек и до сих пор имеет картотека Публичной библиотеки (электронный каталог в ей уже появился, но в нем есть не все).

До тех пор пока объем данных был относительно мал, и руководителю организации требовались всего лишь несколько отчетов, такие картотеки прекрасно исполняли роль хранилища данных. Однако по мере роста предприятия и повышения требований к отчетности обработка информации с помощью картотек становилась все более трудоемкой. Фактически поиск и использование данных среди возрастающего количества картотечных ящиков становились настолько долгими и обременительными, что возможность извлечения сколько-либо полезной информации постепенно сводилась к нулю.

Перевод картотеки в соответствующую файловую структуру компьютера был достаточно сложной технической задачей (мы-то уже привыкли к современному дружественному пользовательскому интерфейсу компьютера и позабыли, какими враждебными и непонятными казались компьютеры пользователям в недалеком прошлом!). По этой причине возникла потребность в профессионалах особого рода — специалистах по обработке данных (Data Processing specialist, DP), которых необходимо было нанять или "взрастить" из имеющихся сотрудников. DP-специалист умел строить необходимые структуры файлов, зачастую разрабатывая и программное обеспечение, помогающее управлять данными в таких структурах, а также писал прикладные программы, которые автоматически создавали необходимые отчеты на основе данных файла. В то время было порождено огромное число компьютеризованных систем файлов. Изначально компьютерные файлы в файловой структуре были очень похожи на документы картотеки. И представляли собой отдельные текстовые, или бинарные файлы, по структуре напоминающие таблицу, в каждом из которых хранились данные в виде так называемых записей (Record) (Логически связанный набор из одного или более полей, описывающих персону, местоположение или предмет.)

До сих пор в языках программирования и структурах файлов (CSV –разделители запятые) можно встретить отголоски этого.

На основе содержимого таких файлов специалист писал программы, создававшие необходимые отчеты для коммерческого отдела. Со временем писались дополнительные программы, которые создавали новые отчеты. Со временем к одним файлам добавлялись новые, с другой структурой, иногда связанной по данным , иногда нет. По мере роста

числа файлов небольшая файловая структуравсе разрасталась и разрасталась. Каждый файл в этой системе принадлежал сотруднику или отделу, по заказу которого он был создан. Также на тот момент времени невозможно было при написании программ абстрагироваться от особенностей физической записи файлов на диск.

По мере расширения системы файлов усложняется и системное администрирование. Каждый файл должен иметь собственную систему управления, состоящую из программ, дающих клиентам возможность выполнять следующие действия:

• создание структуры файла;

• добавление данных в файл;

• удаление данных из файла;

• изменение данных в файле;

• вывод содержимого файла.

Даже простая система, состоящая из 20 файлов, потребует 5 x 20 = 100 управляющих программ.

Не менее сложным было и изменение структуры существующего файла (оно предполагало создание нового файла с нужной структурой и старыми данными, изменение всех программ, которые с ним работали и удаление старого файла). Проблемы были и с согласованием данных разных файлов.

Достоинства файловых систем: возможность автоматизированной обработки информации
Недостатки файловых систем: сложность изменения структуры файлов, сложность защиты информации, избыточность данных и связанные с ней аномалии обновления, добавления и удаления информации, сложность совместного использования данных

Таким образом, вскоре файловые системы перестали удовлетворять нужды предприятий. И в 1950-х и 1960-х, появились первые СУБД (иерархические).

СУБД различаются по многим параметрам, включая назначение, количество потенциальных пользователей и т.д , но один из основных –модель данных.

Модель данных — это абстрактное, самодостаточное, логическое определение объектов, операторов и прочих элементов, в совокупности составляющих *абстрактную машину доступа к данным,* с которой взаимодействует пользователь. Упомянутые объекты позволяют моделировать *структуру* данных, а операторы — *поведение* данных.

Иерархические СУБД, в основном, использовались в 60-х-70 годах

В 1969 на конференции CODASYL принимается стандарт по структуре сетевых баз данных.

В следующем году выходит статья Эдгара Кодда с описанием реляционной модели данных.

Первые **реляционные** продукты начали появляться в конце 1970-х и начале 1980-х годов. На данный момент книги преобладающее большинство СУБД были реляционными (по меньшей мере, они поддерживали язык SQL) и предназначались для работы практически на любой существующей программной и аппаратной компьютерной платформе.
С тех пор и до настоящего момента бОльшая часть СУБД является реляционными или реляционными с расширенными возможностями.
первые объектные системы были выпущены в конце 1980-х и начале 1990-х годов, а **объектно-реляционные** системы были созданы в конце 1990-х годов. Большинство объектно-реляционных СУБД основываются на расширенных оригинальных реляционных продуктах.

Реляционные базы данных имеют много преимуществ, но они не идеальны. Уже с первых лет они приносили много проблем. Для разработчиков приложений основной проблемой была потеря соответствия (impedance mismatch): различие между реляционной моделью и структурами данных, находящимися в памяти. Реляционная модель данных представляет их в виде таблиц и строк, или точнее, отношений и кортежей. В реляционной модели кортежем называется множество пар имя-значение, а отношением - множество кортежей. (Реляционное определение кортежа немного отличается от их определения в математике и многих языках программирования, имеющих соответствующие типы данных, в которых кортежи представляют собой последовательности значений.) Все операции в языке SQL получают и возвращают отношения, что приводит к математически элегантной реляционной алгебре. Использование отношений обеспечивает определенную элегантность и простоту, но одновременно накладывает некоторые ограничения. В частности, значения в реляционном кортеже должны быть простыми - они не должны содержать никаких структур, например вложенных записей или списков. Это ограничение не относится к структурам данных, находящихся в памяти, которые могут быть намного сложнее отношений. В результате, если вы захотите использовать сложную структуру данных, находящихся в памяти, то должны будете преобразовать реляционное представление для хранения на диске. В итоге возникает потеря соответствия - два разных представления, требующих трансляции (рис. 1.1). Потеря соответствия - основной источник неприятностей для разработчиков приложений. В 1990-х годах многие считали, что это в конце концов приведет к замене реляционных баз данных базами, реплицирующими структуры данных, хранящихся в памяти, на диск. Это десятилетие было отмечено ростом объектно-ориентированных языков программирования, а за ними появились объектно-ориентированные базы данных. Ожидалось, что они будут доминировать в области разработки программного обеспечения в новом тысячелетии. Однако в то время как объектно-ориентированные языки заняли лидирующие позиции в программировании, объектно-ориентированные базы данных исчезли во мраке.

Появление множества библиотек для объектно-реляционного отображения, таких как Hibernate и iBATIS, реализующих широко известные шаблоны отображения [Fowler PoEAA], смягчило, но не устранило проблему потери соответствия. Библиотеки для объектно-реляционного отображения взяли на себя большую часть "грязной" работы, но

сами по себе могут стать проблемой, если разработчики будут упрямо игнорировать потерю эффективности работы баз данных и обработки запросов.

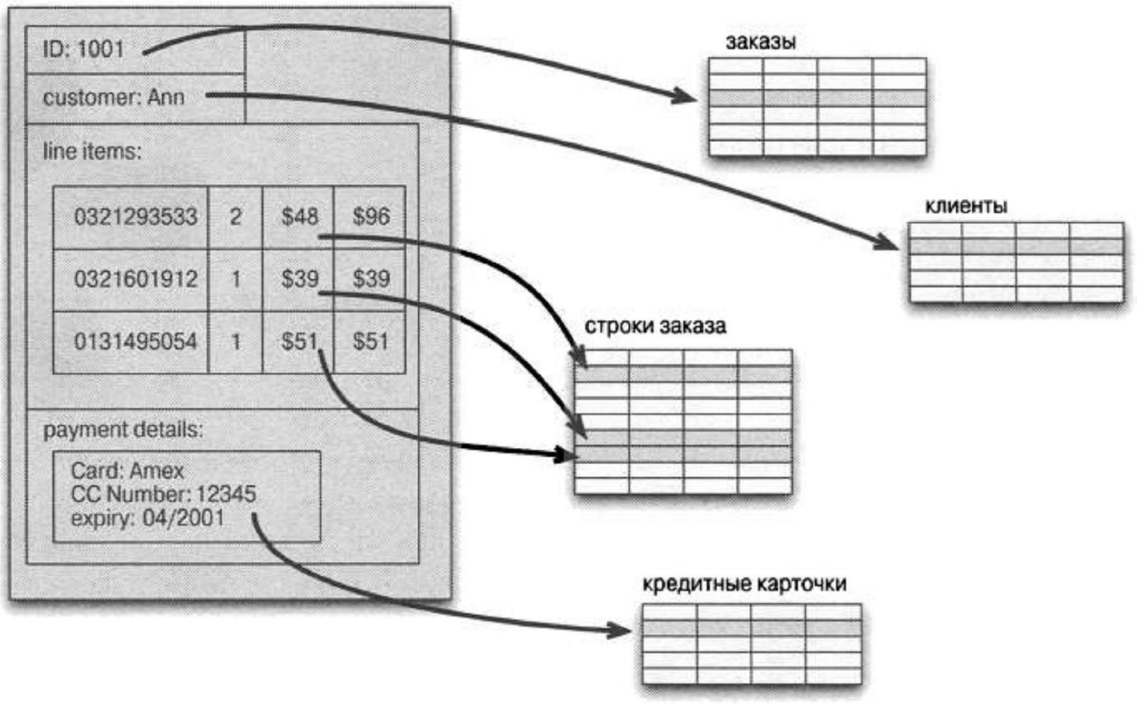


Рис. 1.1. Заказ, который в пользовательском интерфейсе выглядит как единая агрегированная структура, в реляционной базе данных разделяется на множество строк из многих таблиц

Реляционные базы данных продолжают доминировать в области промышленной обработки данных в 2000-х годах, но на протяжении первого десятилетия в стенах их неприступной крепости стали появляться трещины.

Веб-сайты стали внимательно следить за деятельностью клиентов и структурой. Появились крупные совокупности данных: связи, социальные сети, активность в блогах, картографические данные. Рост объема данных сопровождался ростом количества пользователей - крупнейшие сайты достигли громадных размеров и обслуживали огромное количество клиентов. Для того чтобы справиться с возрастающим объемом данных и трафика, потребовались более крупные вычислительные ресурсы. Существовали две возможности масштабирования: вертикальное и горизонтальное. ВертиКllАьное масштабирование (scaling up} подразумевает укрупнение компьютеров, увеличение количества процессоров, а также дисковой и оперативной памяти. Но более крупные машины становятся все более и более дорогими, не говоря уже о том, что существует физический предел их укрупнения. Альтернативой было использование большого количества небольших машин, объединенных в кластер. Кластер маленьких машин может использовать обычное аппаратное обеспечение и в результате удешевить масштабирование. Кроме того, кластер более надежен - отдельные машины могут выйти из строя, но весь кластер продолжит функционирование, несмотря на эти сбои, обеспечивая высокую надежность. Когда произошел сдвиг в сторону кластеров, возникла новая проблема - реляционные базы данных не предназначены для работы на кластерах. Кластерные реляционныебазы данных, та.кие ка.к Oracle RAC или Microsoft SQL Server,

основаны на концепции общей дисковой подсистемы. Они используют кластерную файловую систему, выполняющую запись данных в легко доступную дисковую подсистему, но это значит, что дисковая подсистема по-прежнему является единственным источником уязвимости кластера. Реляционные базы данных также могут работать на разных серверах с разными наборами данных, эффективно выполняя фрагментацию базы данных (sharding) (см. раздел 4.2, "Фрагментация"). Хотя это позволяет разделить рабочую нагрузку, вся процедура фрагментации должна контролироваться приложением, которое должно следить за тем, ка.кой сервер базы данных и за ка.кой порцией данных обращается. Кроме того, утрачивается возможность управления запросами, целостностью данных, транзакциями и согласованностью между частями кластера.

Вехи истории развития СУБД:

Год	Событие
1950-х и 1960-х	Появление и развитие иерархических СУБД
1969	Разработка Conference on Data Systems Languages (CODASYL) — постоянно действующая конференция по языкам обработки данных) стандарта сетевой модели данных
1969 конец/начало 1970	Опубликование Э.Ф. Коддом статьи «A Relational Model of Data for Large Shared Data Banks» , которая считается первой работой по реляционной модели данных. Но теория пока не начала использоваться: Внешняя концептуальная простота реляционной модели достигается за счет ресурсов компьютера, а компьютеры в то время не обладали достаточной мощностью для реализации реляционной модели.
1971	Для разработки стандартов для баз данных, созвана конференция Conference on Data Systems Languages (CODASYL), уже существовавшая к тому времени.
11 июня 2009 года.	Рождение термина "NoSQL" в современном смысле/ Он появился на конференции в Сан-Франциско, организованной Йоханом Оскарссоном (Johan Oskarsson), разработчиком программного обеспечения, живущим в Лондоне.

Тема 1.2. Сетевая, иерархическая и реляционная модели данных.

Есть и другое определение для модели данных:

Модель данных– это абстрактное, независимое, логическое определение структур данных, операторов над данными и прочего, что в совокупности составляет *абстрактную систему*, с которой взаимодействует пользователь.

2.1. Иерархическая модель

В иерархической модели связи между данными можно описать с помощью упорядоченного графа (или дерева). Упрощенно представление связей между данными в иерархической модели показано на рис. 2.1.

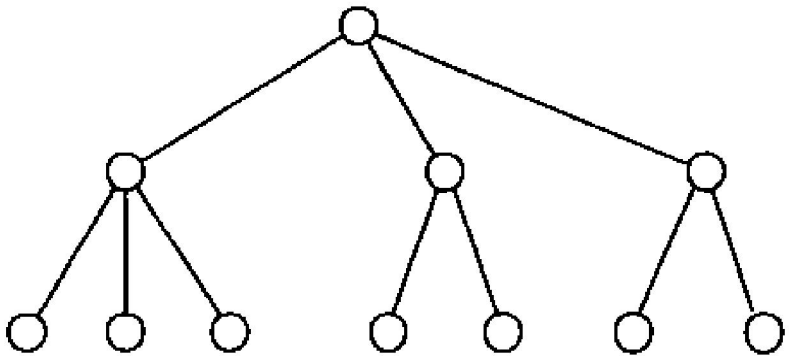


Рис. 2.1. Представление связей в иерархической модели

Для описания структуры (схемы) иерархической БД на некотором языке программирования используется тип данных «дерево». Тип «дерево» схож с типами данных «структура» языков программирования ПЛ/1 и С и «запись» языка Паскаль. В них допускается вложенность типов, каждый из которых находится на некотором уровне. Тип «дерево» является составным. Он включает в себя подтипы («поддеревья»), каждый из которых, в свою очередь, является типом «дерево». Каждый из типов «дерево» состоит из одного «корневого» типа и упорядоченного набора (возможно, пустого) подчиненных типов. Каждый из элементарных типов, включенных в тип «дерево», является простым или составным типом «запись». Простая «запись» состоит из одного типа, например числового, а составная «запись» объединяет некоторую совокупность типов, например, целое, строку символов и указатель (ссылку). Пример типа «дерево» как совокупности типов показан на рис. 2.2.

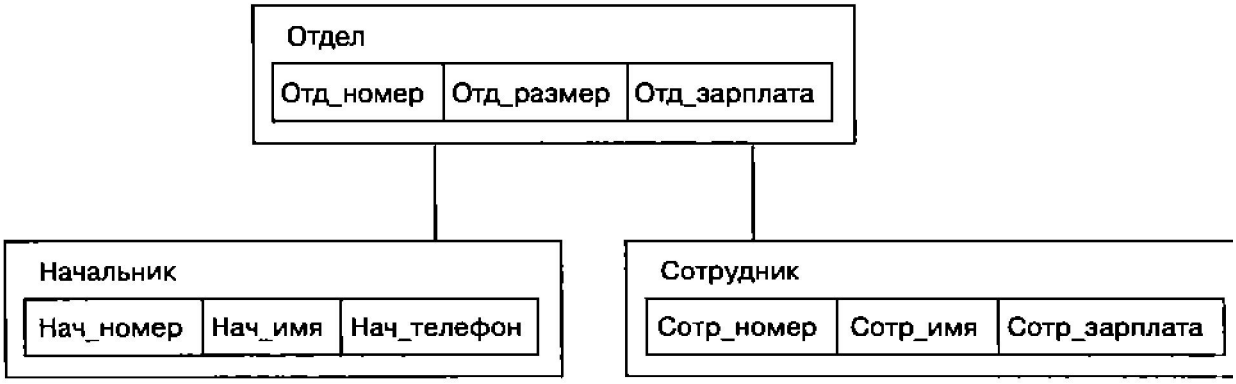


Рис. 2.2. Пример типа «дерево»

Корневым называется тип, который имеет подчиненные типы и сам не является подтипом. *Подчиненный* тип (подтип) является *потомком* по отношению к типу, который выступает для него в роли предка (родителя). Потомки одного и того же типа являются *близнецами* по отношению друг к другу. В целом тип «дерево» представляет собой иерархически организованный набор типов «запись». Иерархическая БД представляет собой упорядоченную совокупность экземпляров данных типа «дерево» (деревьев), содержащих экземпляры типа «запись» (записи). Часто отношения родства между типами переносят на отношения между самими записями. Поля записей хранят собственно числовые или символьные значения, составляющие основное содержание БД. Обход всех элементов

иерархической БД обычно производится сверху вниз и слева направо. В иерархических СУБД может использоваться терминология, отличающаяся от приведенной. Так, в системе IMS понятию «запись» соответствует термин «сегмент», а под «записью БД» понимается вся совокупность записей, относящаяся к одному экземпляру типа «дерево». Данные в базе с приведенной схемой (рис. 2.2) могут выглядеть, например, как показано на рис. 2.3.

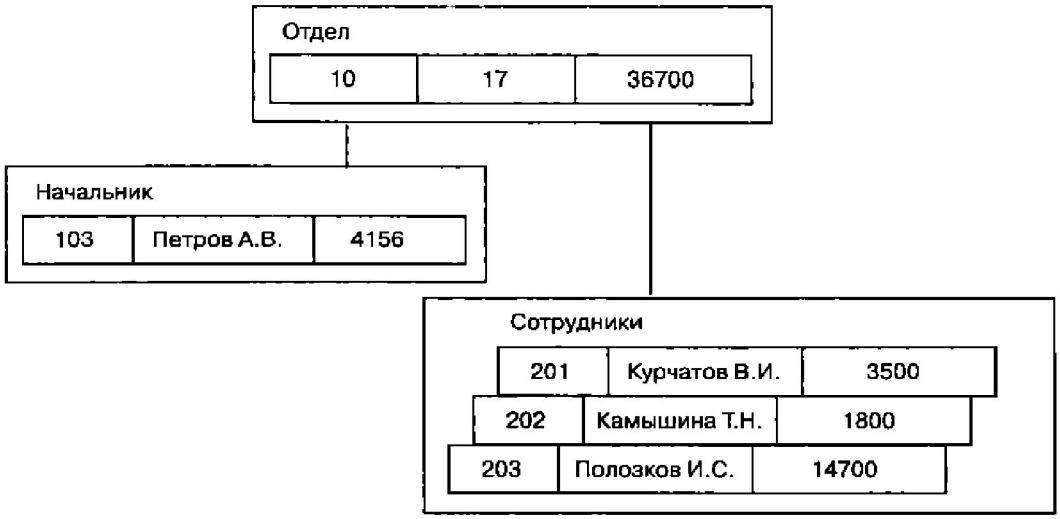


Рис. 2.3. Данные в иерархической базе

Для организации *физического* размещения иерархических данных в памяти ЭВМ могут использоваться следующие группы методов:

• представление линейным списком с последовательным распределением памяти (адресная арифметика, левосписковые структуры);

• представление связными линейными списками (методы, использующие указатели и справочники).

К основным операциям манипулирования иерархически организованными данными относятся следующие:

•поиск указанного экземпляра БД (например, дерева со значением 10 в поле Отд_номер);

• переход от одного дерева к другому;

• переход от одной записи к другой внутри дерева (например, к следующей записи типа Сотрудники);

• вставка новой записи в указанную позицию;

• удаление текущей записи и т. д.

В соответствии с определением типа «дерево», можно заключить, что между предками и потомками автоматически поддерживается контроль целостности связей. Основное правило контроля целостности формулируется следующим образом: потомок не может существовать без родителя, а у некоторых родителей может не быть потомков. Механизмы поддержания целостности связей между записями различных деревьев отсутствуют.

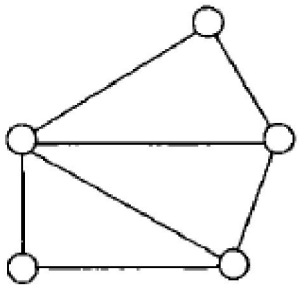
К *достоинствам* иерархической модели данных относятся эффективное использование памяти ЭВМ и неплохие показатели времени выполнения основных операций над данными. Иерархическая модель данных удобна для работы с иерархически упорядоченной информацией.

Недостатком иерархической модели является ее громоздкость для обработки информации с достаточно сложными логическими связями, а также сложность понимания для обычного пользователя.

На иерархической модели данных основано сравнительно ограниченное количество СУБД, в числе которых можно назвать зарубежные системы IMS, PC/Focus, Team-Up и Data Edge, а также отечественные системы Ока, ИНЭС
и МИРИС.

2.2. Сетевая модель

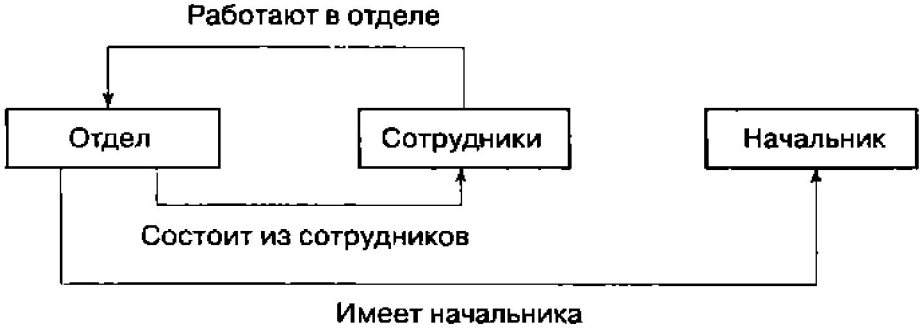
Сетевая модель данных позволяет отображать разнообразные взаимосвязи элементов данных в виде произвольного графа, обобщая тем самым иерархическую модель данных



(рис. 2.4).

Рис. 2.4. Представление связей в сетевой модели

Наиболее полно концепция сетевых БД впервые была изложена в Предложениях группы КОДАСИЛ (CODASYL). Для описания схемы сетевой БД используется две группы типов: «запись» и «связь». Тип «связь» определяется для двух типов «запись»: предка и потомка. Переменные типа «связь» являются экземплярами связей. Сетевая БД состоит из набора записей и набора соответствующих связей. На формирование связи особых ограничений не накладывается. Если в иерархических структурах запись-потомок могла иметь только одну запись-предка, то в сетевой модели данных запись-потомок может иметь произвольное число записей-предков (сводных родителей).
Пример схемы простейшей сетевой БД показан на рис. 2.5.



Типы связей здесь обозначены надписями на соединяющих типы записей линиях. В различных СУБД сетевого типа для обозначения одинаковых по сути понятий зачастую используются различные термины. Например, такие как элементы и агрегаты данных, записи, наборы, области и т. д. Физическое размещение данных в базах сетевого типа может быть организовано практически теми же методами, что и в иерархических базах данных. К числу важнейших операций манипулирования данными баз сетевого типа можно отнести следующие:
• поиск записи в БД;

Рис. 2.5. Пример схемы сетевой БД
• переход от предка к первому потомку; .

• переход от потомка к предку;
• создание новой записи;
• удаление текущей записи;
• обновление текущей записи;
• включение записи в связь;
• исключение записи из связи;
• изменение связей и т. д.

Достоинством сетевой модели данных является возможность эффективной реализации по показателям затрат памяти и оперативности.В сравнении с иерархической моделью сетевая модель предоставляет большие возможности в смысле допустимости образования произвольных связей.

Недостатком сетевой модели данных является высокая сложность и жесткость схемы БД, построенной на ее основе, а также сложность для понимания и выполнения обработки информации в БД обычным пользователем. Кроме того,в сетевой модели данных ослаблен контроль целостности связей вследствие допустимости установления произвольных связей между записями. Системы на основе сетевой модели не получили широкого распространения на практике. Наиболее известными сетевыми СУБД являются следующие: IDMS, db Vistalll, СЕТЬ, СЕТОР и КОМПАС

2.3. Реляционная модель

Реляционная модель данных предложена сотрудником фирмы IBM Эдгаром Коддом и основывается на понятии отношение (relation).

Отношение представляет собой множество элементов, называемых кортежами. Подробно теоретическая основа реляционной модели данных рассматривается в следующем разделе. Наглядной формой представления отношения является привычная для человеческого восприятия двумерная таблица.

лицы имеет одинаковую структуру и состоит из полей. Строкам таблицы соответствуют кортежи, а столбцам — атрибуты отношения. С помощью одной таблицы удобно описывать простейший вид связей между данными, а именно деление одного объекта (явления, сущности, системы и проч.), информация о котором хранится в таблице, на множество подобъектов, каждому из которых соответствует строка или запись таблицы. При этом каждый из подобъектов имеет одинаковую структуру или свойства, описываемые соответствующими значениями полей записей. Например, таблица может содержать сведения о группе обучаемых, о каждом из которых известны следующие характеристики: фамилия, имя и отчество, пол, возраст и образование. Поскольку в рамках одной таблицы не удается описать более сложные логические структуры данных из предметной области, применяют *связывание* таблиц.

Физическое размещение данных в реляционных базах на внешних носителях легко осуществляется с помощью обычных файлов.

Достоинство реляционной модели данных заключается в простоте, понятности и удобстве физической реализации на ЭВМ. Именно простота и понятность для пользователя явились основной причиной их широкого использования. Проблемы же эффективности обработки данных этого типа оказались технически вполне разрешимыми.

Основными *недостатками* реляционной модели являются следующие: отсутствие стандартных средств идентификации отдельных записей и сложность описания иерархических и сетевых связей.

Примерами зарубежных реляционных СУБД для ПЭВМ являются следующие: dBase!II Plus и dBase IY (фирма Ashton-Tate), DB2 (IBM), R:BASE (Microrim), FoxPro ранних версий и FoxBase (Fox Software), Paradox и dBASE for Windows (Borland), FoxPro более поздних версий, Visual FoxPro и Access (Microsoft), Clarion (Clarion Software), Ingres (ASK Computer Systems) и

Oracle (Oracle).

Заметим, что последние версии реляционных СУБД имеют некоторые свойства объектно-ориентированных систем. Такие СУБД часто называют объектно-реляционными. Примером такой системы можно считать продукты Oracle 8.x. Системы предыдущих версий вплоть до Oracle 7.x считаются «чисто» реляционными.

2.4. Объектно-ориентированная модель

В объектно-ориентированной модели при представлении данных имеется возможность идентифицировать отдельные записи базы. Между записями базы данных и функциями их обработки устанавливаются взаимосвязи с помощью механизмов, подобных соответствующим средствам в объектно-ориентированных языках программирования. Стандартизованная объектно-ориентированной модель описана в рекомендациях стандарта ODMG-93 (Object Database Management Group — группа управления объектно-ориентированными базами данных). Реализовать в полном объеме рекомендации ODMG-93 пока не удается. Для иллюстрации ключевых идей рассмотрим несколько упрощенную модель объектно-ориентированной БД. Структура объектно-ориентированной БД графически представима в виде дерева, узлами которого являются объекты. Свойства объектов описываются некоторым стандартным типом (например, строковым — string) или типом, конструируемым пользователем (определяется как class). Значением свойства типа string является строка символов. Значение свойства типа class есть объект, являющийся экземпляром соответствующего класса. Каждый объект-экземпляр класса считается потомком объекта, в котором он определен как свойство. Объект-экземпляр класса принадлежит своему классу и имеет одного родителя. Родовые отношения в БД образуют связную иерархию объектов.

Пример логической структуры объектно-ориентированной БД библиотечного дела приведен на рис. 2.10.

Здесь объект типа БИБЛИОТЕКА является родительским для объектов экземпляров классов АБОНЕНТ, КАТАЛОГ и ВЫДАЧА. Различные объекты типа КНИГА могут иметь одного или разных родителей. Объекты типа КНИГА, имеющие одного и того же родителя, должны различаться по крайней мере инвентарным номером (уникален для каждого экземпляра книги), но имеют одинаковые значения свойств *isbn, удк, название* и *автор.*

Логическая структура объектно-ориентированной БД внешне похожа на структуру иерархической БД. Основное отличие между ними состоит в методах манипулирования данными.

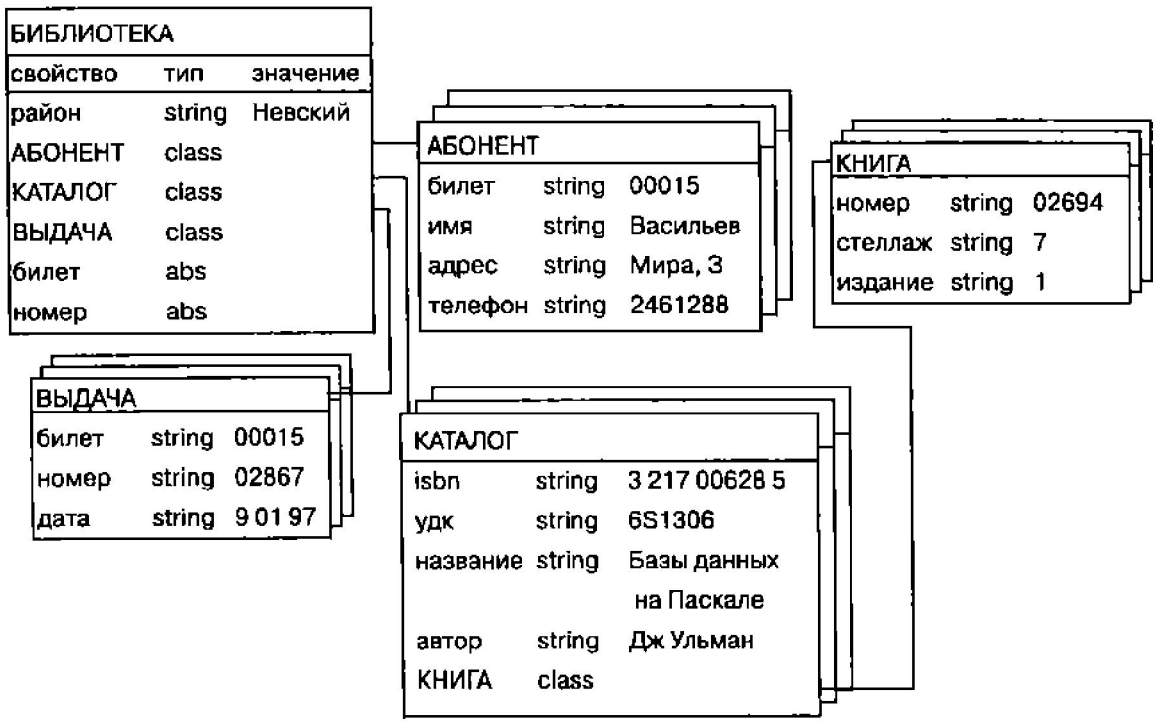


Рис. 2.10. Логическая структура БД библиотечного дела

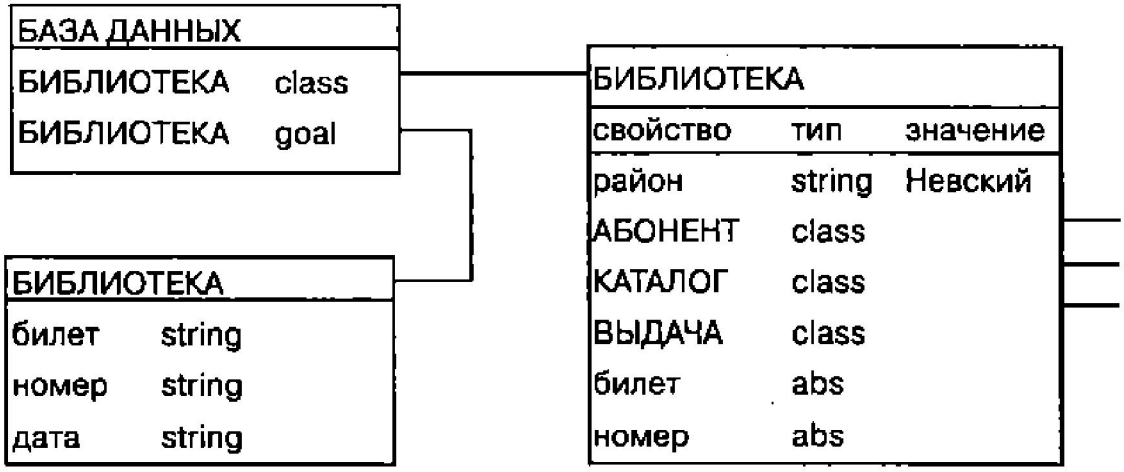
Для выполнения действий над данными в рассматриваемой модели БД применяются логические операции, усиленные объектно-ориентированными механизмами инкапсуляции, наследования и полиморфизма. Ограниченно могут применяться операции, подобные командам SQL (например, для создания БД). Создание и модификация БД сопровождается автоматическим формированием и последующей корректировкой индексов (индексных таблиц), содержащих информацию для быстрого поиска данных. Рассмотрим кратко понятия инкапсуляции, наследования и полиморфизма применительно к объектно-ориентированной модели БД.

Инкапсуляция ограничивает область видимости имени свойства пределами того объекта, в котором оно определено. Так, если в объект типа КАТАЛОГ добавить свойство, задающее телефон автора книги и имеющее название *телефон,* то мы получим одноименные свойства у объектов АБОНЕНТ и КАТАЛОГ. Смысл такого свойства будет определяться тем объектом, в который оно инкапсулировано.

Наследование, наоборот, распространяет область видимости свойства на всех потомков объекта. Так, всем объектам типа КНИГА, являющимся потомками объекта типа КАТАЛОГ, можно приписать свойства объекта-родителя: *isbn, удк, название* и *автор.* Если необходимо расширить действие механизма наследования на объекты, не являющиеся непосредственными родственниками (например, между двумя потомками одного родителя), то в их общем предке определяется абстрактное свойство типа abs. Так, определение абстрактных свойств *билет* и *номер* в объекте БИБЛИОТЕКА приводит к наследованию этих свойств всеми дочерними объектами АБОНЕНТ, КНИГА и ВЫДАЧА. Не случайно поэтому значения свойства *билет* классов АБОНЕНТ и ВЫДАЧА, показанных на рисунке, будут одинаковыми — 00015.

Полиморфизм в объектно-ориентированных языках программирования означает способность одного и того же программного кода работать с разнотипными данными. Другими словами, он означает допустимость в объектах разных типов иметь методы (процедуры или функции) с одинаковыми именами. Во время выполнения объектной программы одни и те же методы оперируют с разными объектами в зависимости от типа аргумента. Применительно к нашей объектно-ориентированной БД полиморфизм означает, что объекты класса КНИГА, имеющие разных родителей из класса КАТАЛОГ,

могут иметь разный набор свойств. Следовательно, программы работы с объектами класса КНИГА могут содержать полиморфный код.
Поиск в объектно-ориентированной БД состоит в выяснении сходства между объектом, задаваемым пользователем, и объектами, хранящимися в БД.
Определяемый пользователем объект, называемый объектом-целью (свойство объекта имеет тип *goal),* в общем случае может представлять собой подмножество всей хранимой в БД иерархии объектов. Объект-цель, а также результат выполнения запроса могут храниться в самой базе. Пример запроса о номерах читательских билетов и именах абонентов, получавших в библиотеке хотя бы одну книгу, показан на рис. 2.11.



Основным *достоинством* объектно-ориентированной модели данных в сравнении с реляционной является возможность отображения информации о сложных взаимосвязях объектов. Объектно-ориентированная модель данных позволяет идентифицировать отдельную запись базы данных и определять функции их обработки.
Недостатками объектно-ориентированной модели являются высокая понятийная сложность, неудобство обработки данных и низкая скорость выполнения запросов.
В 90-е годы существовали экспериментальные прототипы объектно-ориентированных систем управления базами данных. В настоящее время такие системы получили достаточно широкое распространение, в частности, к ним относятся следующие СУБД: G-Base (Grapael), GemStone (Servio-Logic совместно с OGI), Statice (Symbolics), ObjectStore (Object Design), Objectivity /DB (Objectivity), Versant (Versant Technologies), 0 2 (Ardent Software), ODBJupiter (научно-производственный центр «Интелтек Плюс»), а также Iris, Orion и Postgres.

NOSQL

Если база данных относится к категории NoSQL, значит, речь идет о нечетко определенном множестве баз данных с открытым кодом, в большинстве своем разработанных в начале XXI века и, как правило, не использующих язык запросов SQL.

К NoSQL моделям данных относят:

- Модель "ключ-значение"
- Документные базы данных
- Семейство столбцов
- Графовая модель

Нормализация и денормализация

Нормализация данных – это процесс приведения модели к виду, позволяющему получить в дальнейшем структуру базы данных, в которой устранена избыточность хранения и сведены к минимуму аномалии при добавлении, удалении, изменении данных. Процесс нормализации проводится поэтапно. На каждом из этапов на структуру базы накладывается некоторое ограничение и в структуре базы выправляется некоторый дефект. Про базу с соответствующими ограничениями говорят, что она находится в одной из **нормальных форм.**
Выделяют 6 нормальных форм:

Первая нормальная форма (1НФ)

Вторая нормальная форма (2НФ)

Третья нормальная форма (3НФ)

Нормальная форма Бойса-Кодда(НФБК)

Четвертая нормальная форма (4НФ)

Пятая нормальная форма (5НФ)

Каждая из нормальных форм включает требования предыдущей нормальной формы и накладывает свои ограничения/ требования к структуре базы.

Нормализация основывается на концепции **нормальных форм.** Говорят, что переменная отношения находится в определенной нормальной форме, если она удовлетворяет заданному набору условий. Например, переменная отношения находится во второй нормальной форме (или в 2НФ) тогда и только тогда, когда она находится в **1НФ** и удовлетворяет дополнительному условию,приведенному ниже.

На рисунке показано несколько нормальных форм, которые определены к настоящему времени. Первые три (1НФ, **2НФ** и ЗНФ) были описаны Коддом (Codd) . Как видно из рисунка, все нормализованные переменные отношения находятся в **1НФ,** некоторые переменные отношения в **1НФ** также находятся в 2НФ, и некоторые переменные отношения в **2НФ** также находятся в ЗНФ. Мотивом, которым руководствовался Кодд при введении дополнительных определений, было то, что вторая нормальная форма "более желательна" (в смысле, который будет разъяснен ниже), чем первая, а третья, в свою очередь, "более желательна", чем вторая. Таким образом, в общем случае при проектировании базы данных целесообразно использовать переменные отношения в третьей нормальной форме, а не в первой или второй.

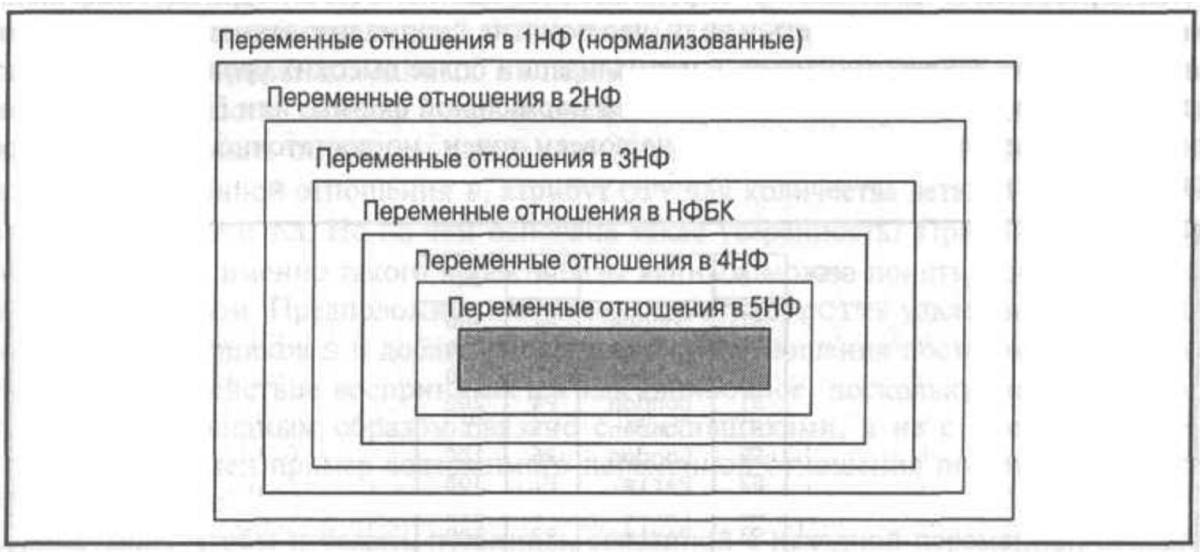


Рисунок. Уровни нормализации

Впоследствии Фейгином (Fagin) была определена новая, **четвертая нормальная форма (4НФ),** которая стала четвертой по счету, поскольку в момент ее создания нормальная форма Бойса-Кодда считалась третьей. Затем в Фейгин дал определение еще одной нормальной формы, которую назвал **проекционно-соединительной нормальной формой (ПСНФ);** ее называют также **петой нормальной формой** или **5НФ.** Как показано на рисунке, некоторые переменные отношения в НФБК находятся также в 4НФ, а некоторые переменные отношения в 4НФ находятся также в 5НФ.

Функциональная зависимость. Описывает связь между атрибутами отношения. Например если в отношении R, содержащем атрибуты A и B, атрибут B функционально зависит от атрибута A (что обозначается как A (B)), то каждое значение атрибута A связано только с одним значением атрибута B. (Причем атрибуты A и B могут состоять из одного или нескольких атрибутов.)

Функциональная зависимость является смысловым (или семантическим) свойством атрибутов отношения. Семантика отношения указывает, как его атрибуты

могут быть связаны друг с другом, а также определяет функциональные зависимости между атрибутами в виде *ограничений,* наложенных на некоторые атрибуты.

Рассмотрим отношение с атрибутами A и B, где атрибут B функционально зависит от атрибута A. Если нам известно значение атрибута A, то при рассмотрении отношения с такой зависимостью в любой момент времени во всех строках

этого отношения, содержащих указанное значение атрибута A, мы найдем одно

и то же значение атрибута B. Таким образом, если две строки имеют одно и то

же значение атрибута A, то они обязательно имеют одно и то же значение атрибута B. Однако для заданного значения атрибута B может существовать несколько различных значений атрибута A. Зависимость между атрибутами A и в можно

схематически представить в виде диаграммы, показанной на рисунке ниже

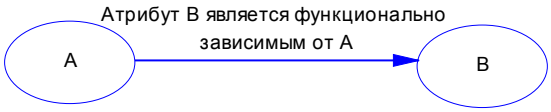


Рис. Диаграмма функциональной зависимости

Детерминант. Детерминантом функциональной зависимости называется атрибут или группа атрибутов, расположенная на диаграмме функциональной зависимости слева от стрелки.

При наличии функциональной зависимости атрибут или группа атрибутов, рас-

положенная на ее диаграмме слева от стрелки, называется *детерминантом*

(determinant). Например, на рис. 13.1 атрибут A является детерминантом атрибута

B. Способ выявления функциональной зависимости показан в следующем примере.

Функциональная зависимость называется *тривиальной,* если она остается справедливой при любых условиях. К тому же зависимость является тривиальной, если и только если в правой части выражения, определяющего зависимость, приведено подмножество (но необязательно собственное подмножество) множества., которое указано в левой части (детерминанте) выражения, как показано в следующем примере.

Множество всех функциональных зависимостей, которые могут быть выведены из заданного множества функциональных зависимостей X, называется *замыканием* X и записывается как X*. Для успешной работы, безусловно, необходимо определить ряд правил, позволяющих вычислить X* из X. Набор правил вывода, называемый *аксиомами Армстронга,* показывает способы вывода новых функциональных зависимостей из заданных . Предположим, что A, B и C — подмножества атрибутов отношения R. Аксиомы Армстронга приведены ниже.

1. Рефлексивность. Если B — подмножество A, то A—>B.

2. Дополнение. Если A->B, то A, C—>B, C.

3. Транзитивность. Если A—>B и B->C, то A->C.

4. Самоопределение. A->A.

5. Декомпозиция. Если A—>B, c, то A->B и A—>C.

6. Объединение. Если A->B и A->C, то A->B, C.

7. Композиция. Если A—>B и C->D, то A, C—>B, D.

Перед обсуждением первой нормальной формы целесообразно дать определение того состояния, которое ей предшествует.

Ненормализованная форма (ННФ). Таблица, содержащая одну или несколько

повторяющихся труп п дан ных.

Первая нормальная форма (1НФ). Отношение, в котором на пересечении каждой строки и каждого столбца содержится одно и только одно атомарное значение.

Обратите внимание! Значение может быть составным (ФИО) или многозначным (список номеров телефона). И то и другое не подходит 1 НФ

Полная функциональная зависимость. Если А и В – атрибуты отношения, то

атрибут в находится в полной функциональной зависимости от атрибута А, если атрибут в является функционально зависимым от А, но не зависит нb от одного собственного подмножества атрибута А.

Функциональная зависимость А—>в является *полной* функциональной зависимостью, если удаление какого-либо атрибута из А приводит к утрате этой зависимости. Функциональная зависимость A->B называется *частичной,* если в А есть некий атрибут, при удалении которого эта зависимость сохраняется.

Вторая нормальная форма

Вторая нормальная форма применяется к отношениям с составными ключа-

ми, т.е. к таким отношениям, первичный ключ которых состоит из двух или не-

скольких атрибутов. Дело в том, что отношение с первичным ключом на основе

единственного атрибута всегда находится, по крайней мере, в форме 2НФ.

Вторая нормальная форма (2НФ). Отношение, которое находится в первой

г нормальной форме и каждый атрибут которого, не входящий в состав первичного ключа, характеризуется полной функциональной зависимостью от этого первичного ключа.

Или более строго

Вторая нормальная форма (2НФ). Отношение, находящееся в первой нормальной форме, в котором каждый атрибут, отличный от атрибута первичного ключа, является полностью функционально независимым от любого потенциального ключа.

Нормализация отношений 1НФ с приведением к форме 2НФ предусматривает устранение частичных зависимостей,

Третья нормальная форма

Транзитивная зависимость. Если для атрибутов A, B и C некоторого отношения существуют зависимости вида A—>B и B->C , это означает, что атрибут с транзитивно зависит от атрибута А через атрибут в (при условии, что атрибут А функционально не зависит ни от атрибута в, ни от атрибута с).

Третья **нормальная** форма (ЗНФ). Отношение, которое находится а первой и во второй нормальных формах и не имеет атрибутов, не входящих в первичный ключ атрибутов, которые находились бы в транзитивной функциональной зависимости от этого первичного ключа.

Или если более строго

третья нормальная форма (ЗНФ). Отношение, находящееся в первой и второй нормальной форме, в котором ни один атрибут, отличный от атрибута первичного ключа, не является транзитивно зависимым ни от одного потенциального ключа.

Нормализация отношений 2НФ с образованием отношений ЗНФ предусматривает устранение транзитивных зависимостей. Если в отношении существует транзитивная зависимость между атрибутами, то транзитивно зависимые атрибуты удаляются из него и помещаются в новое отношение вместе с копией их детерминанта.

Нормальная форма Бойса-Кодда (НФБК)

Отношения базы данных проектируются таким образом, чтобы можно было исключить в них присутствие частичных или транзитивных зависимостей, поскольку

эти зависимости приводят к появлению аномалий обновления.

Нормальная форма Бойса-Кодда (НФБК) основана на функциональных зависимостях, в которых учитываются все потенциальные ключи отношения. Тем не менее в форме НФБК предусмотрены более строгие ограничения по сравнению с общим определением формы ЗНФ

Нормальная форма Бойса-Кодда (НФБК). Отношение находится в НФБК тогда и только тогда, когда каждый его детерминант является потенциальным ключом.

Для проверки принадлежности отношения к НФБК необходимо найти все его детерминанты и убедиться в том, что они являются потенциальными ключами.

Напомним, что детерминантом является один атрибут или группа атрибутов, от которой полностью функционально зависит другой атрибут.

Различие между ЗНФ и НФБК заключается в том, что функциональная зависимость А—>В допускается в отношении ЗНФ, если атрибут В является первичным ключом, а атрибут А не обязательно является потенциальным ключом. Тогда как в отношении НФБК эта зависимость допускается *только* тогда, когда атрибут А является потенциальным ключом. Следовательно, нормальная форма

Бойса-Кодда является более строгой версией формы ЗНФ, поскольку каждое отношение НФБК является также отношением ЗНФ, но не всякое отношение ЗНФ

является отношением НФБК.

Пример

Отношение ClientInterview

clientNo	interview/Date	interviewTime	staffNo	roomNo
CR76	13-May-02	10:30	SG5	G101
CR56	13-May-02	12:00	SG5	G101
CR74	13-May-02	12:00	SG37	G102
CR56	l-Jul-02	10:30	SG5	G102

Рассматриваемое отношение ClientInterview имеет **три потенциальных ключа:**

(clientNo, interviewDate) , IstaffNo, interviewDate, interviewTime) и (roomNo, interviewDate, interviewTime). **Следовательно, отношение** ClientInterview **обладает** тремя составными **потенциальными ключами, которые перекрываются, поскольку в них совместно** используется один **общий атрибут —** interviewDate. В качестве **первичного** ключа данного отношения выбрана комбинация атрибутов (clientNo, interviewDate). Это отношение имеет следующую форму:

ClientInterview (clientNo, interviewDate, interviewTime, staffNo, roomNo)

Отношение ClientInterview содержит следующие **функциональные зависимости**

Обозначение	Зависимость	Описание
ФЗ1	clientNo, interviewDate -> interviewTime, staffNo, roomNo	Первичный ключ
ФЗ2	staffNo, interviewDate, interviewTime —> clientNo	Потенциальный ключ
ФЗ3	roomNo, interviewDate, interviewTime -> staffNo,clientNo	Потенциальный ключ
ФЗ4	staffNo, interviewDate —> roomNo	

Рассмотрим эти функциональные зависимости для определения нормальной формы отношения ClientInterview. Поскольку функциональные зависимости ФЗl, ФЗ2 и ФЗ3 являются потенциальными ключами этого отношения, то они не вызовут никаких проблем. Нам потребуется рассмотреть только функциональную зависимость staffNo, interviewDate —> roomNo (зависимость ФЗ4). Даже если комбинация атрибутов (staffNo, interviewDate} не является потенциальным ключом отношения ClientInterview, эта функциональная зависимость в ЗНФ допускается, поскольку атрибут roomNo является атрибутом первичного ключа и частью потенциального ключа (roomNo, interviewDate, interviewTime). Так как в этом отношении нет никаких частичных или транзитивных зависимостей от первичного ключа (clientNo, interviewDate} и допускается наличие функциональной зависимости ФЗ4, можно считать, что отношение ClientInterview находится в форме ЗНФ, однако это отношение не находится в форме НФБК (более строгий вариант формы ЗНФ), поскольку в нем присутствует детерминант (staffNo, interviewDate), который не является потенциальным ключом этого отношения. В форме НФБК требуется, чтобы все детерминанты отношения были его потенциальными ключами. Вследствие этого отношение ClientInterview может быть подвержено аномалиям обновления. Например, 13 мая 2002 года (значение 13-May-02) при изменении номера комнаты, выделенной сотруднику SG51 потребуется обновить значения в двух строках. Если при этом будет обновлена только одна строка, то база данных перейдет в противоречивое состояние.

Для преобразования отношения ClientInterview в форму НФБК необходимо устранить нарушающую это ограничение функциональную зависимость путем создания двух новых отношений — Interview и StaffRoom, — представленных в таблицах ниже. Отношения Interview и Staf fRoom имеют следующий вид:

Interview (clientNo, interviewDate, interviewTime, staffNo)

StaffRoom (staffNo, interviewDate, roomNo)

Отношение Interview в форме НФБК

clientNo	interviewDate	interviewTime	staffNo
CR76	13-May-02	10.30	SG5
CR56	13-May-02	12.00	SG5
CR74	13-May~02	12.00	SG37
CR56	l-Jul-02	10.30	SG5

Отношение Staff Room в форме НФБК

staffNo	**interviewDate**	**roomNo**
SG5	13-May-02	G101
SG37	13-May-02	G102
SG5	l-Jul-02	G102

Любое отношение, которое не находится в форме НФБК, можно преобразовать в отношения НФБК, как показано выше. Тем не менее преобразование отношений в форму НФБК не всегда приводит к желаемым результатам. Например, иногда после такой декомпозиции не сохраняется важная функциональная зависимость (поскольку детерминант и определяемые им атрибуты помещаются в разные отношения). Б этой ситуации трудно сохранить исходную функциональную зависимость отношения, и важное ограничение может быть утрачено. Если имеет место упомянутая ситуация, то лучше закончить процесс нормализации на этапе образования отношений ЗНФ, в которых все требуемые зависимости всегда сохраняются.

Обратите внимание на то, что в примере при создании двух новых отношений НФБК на основе исходного отношения ClientInterview "утрачивается" следующая функциональная зависимость: roomNo, interviewDate, interviewTime —> staffNo, clientNo (зависимость Ф33), поскольку детерминант этой зависимости больше не будет находиться в том же отношении, что и определяемые им атрибуты. Однако следует признать, что если не устранить функциональную зависимость staffNo, interviewDate —> roomNo (зависимость Ф34), то отношение ClientInterview будет обладать избыточностью данных. Решение о том, следует ли в процессе нормализации остановиться на форме ЗНФ или перейти к

форме НФБК, зависит от относительной избыточности данных, возникающей из-за наличия зависимости ФЗ4, а также от того, имеет ли значение "утрата" зависимости ФЗ3. Например, если всегда соблюдается условие, что сотрудники компании проводят в день только одно собеседование, то наличие зависимости ФЗ4 в отношении ClientInterview не вызывает избыточности и поэтому декомпозиция этого отношения на два отношения НФБК не требуется.

С другой стороны, если сотрудники компании проводят в день несколько собеседований, то наличие зависимости ФЗ4 в отношении ClientInterview вызывает избыточность и можно порекомендовать нормализацию этого отношения до формы НФБК. Однако следует также рассмотреть, насколько значимой является потеря зависимости ФЗ3. Иными словами, необходимо определить, несет ли зависимость ФЗ3 какую-либо важную информацию о собеседованиях с клиентами, которая должна быть представлена в одном из результирующих отношений. Ответ на этот вопрос поможет определить, следует ли сохранить все функциональные зависимости или устранить избыточность данных.

Четвертая нормальная форма (4НФ)

Как было сказано выше, НФБК позволяет устранить любые аномалии, вы званные функциональными зависимостями. Однако в результате теоретических исследований был выявлен еще один тип зависимости — *многозначная зависимость* (Multi-Valued Dependency — MVD), которая при проектировании отношений также может вызвать проблемы, связанные с избыточностью данных .

Рассмотрим представленное в таблице отношение BranchStaffOwner, в котором содержатся имена сотрудников (sName) и владельцев недвижимости (oName) определенного отделения компании (branchNo). В этом случае предположим, что имя сотрудника (sName) однозначно идентифицирует каждого члена персонала компании, а имя владельца (oName) однозначно идентифицирует каждого владельца.

Отношение BranchStaffOwner

branchNo	sName	oName
вооз	Ann Beech	Carol Farrel
вооз	David Ford	Carol Farrel
вооз	Ann Beech	Tina Murphy
вооз	David Ford	Tina Murphy

В этом примере в отделении компании с номером 'ВОО31 работают сотрудники с именами 'Ann Beech' и 'David Ford1. Кроме того, в нем зарегистрированы владельцы

недвижимости с именами 'Carol Farrel' и 'Tina Murphy'. Однако если в данном отделении компании между сотрудниками и владельцами недвижимости нет никакой прямой связи, то необходимо создать строку для каждого сочетания данных о сотруднике и владельце для обеспечения гарантии, что отношение находится в непротиворечивом состоянии. Это требование отражает наличие в отношении BranchStaffOwner многозначной зависимости. Иначе говоря, в данном отношении существует многозначная зависимость, так как в

нем содержатся две независимые связи типа 1:*.

Многозначная зависимость . Представляет такую зависимость между атрибутами отношения (например, A, в и с), что каждое значение A представляет собой

множество значений для а и множество значений для с. Однако множества значений для в и с не зависят друг от друга.

Многозначная зависимость между атрибутами A, B и C некоторого отношения

дальше будет обозначаться следующим образом:

A -> B

A -> C

Например, определим многозначную зависимость, имеющую место в отношении BranchStaffOwner, представленном в таблице ниже, следующим образом;

branchNo —> sName

branchNo —> oName

Многозначная зависимость может быть дополнительно определена как *тривиальная* или *нетривиальная.* Например, многозначная зависимость A—>в некоторого отношения R определяется как тривиальная, если атрибут в является подмножеством атрибута A или A и B = R. И наоборот, многозначная зависимость определяется как нетривиальная, если ни то ни другое условие не выполняется. Тривиальная многозначная зависимость (МЗЗ) не накладывает никаких ограничений на данное отношение, а нетривиальная — накладывает.

Многозначная зависимость в представленном отношениBranchStaffOwner является нетривиальной, так как в этом отношении ни то ни другое условие не удовлетворяется. Следовательно, на отношение BranchStaffOwner по причине наличия нетривиальной МЗЗ накладывается ограничение, которое приводит к появлению повторяющихся строк, необходимых для

того, чтобы гарантировать непротиворечивость связи между атрибутами sName и oName. Например, если бы потребовалось зарегистрировать нового владельца недвижимости в отделении ВООЗ, то пришлось бы создать две новые строки, по одной для каждого сотрудника компании, чтобы обеспечить сохранение непротиворечивого состояния

указанного отношения. Это — пример аномалии обновления, вызванной наличием нетривиальной многозначной зависимости.

Хотя отношение BranchStaffOwner находится в форме НФБК, оно все еще остается плохо структурированным из-за избыточности данных, вызванной наличием нетривиальной МЗЗ. Очевидно, что целесообразнее было бы определить более строгую форму НФБК, которая исключила бы создание таких реляцион-

ных структур, как отношение BranchStaf fOwner.

Определение четвертой нормальной формы

Четвертая нормальная форма (4НФ) Отношение в нормальной форме Бойса-Кодда, которое не содержит нетривиальных многозначных зависимостей.

Четвертая нормальная форма (4НФ) является более строгой разновидностью нормальной формы Бойса-Кодда, поскольку в отношениях 4НФ нет нетривиальных МЗЗ и поэтому нет и избыточности данных

Нормализация отношения НФБК с получением отношений 4НФ заключается в устранении МЗЗ из отношения НФБК путем выделения в новое отношение одного или нескольких участвующих в МЗЗ атрибутов вместе с копией одного или нескольких детерминантов.

Например, отношение BranchStaf fOwner (см. табл. 13.22) не находится в форме 4НФ, поскольку в нем присутствует нетривиальная МЗЗ. Это отношение следует преобразовать в отношения BranchStaf f и BranchOwner, показанные в таблицах ниже.

Отношение BranchStaff в форме 4НФ

branchNo	sName
вооз	Ann Beech
вооз	David Ford

Отношение BranchOwner в форме 4НФ

branchNo	oName
вооз	Carol Farrel
вооз	Tina Murphy

Оба новых отношения находятся в форме 4НФ, поскольку отношение BranchStaff содержит тривиальную МЗЗ branchNo -> sName, а отношение BranchOwner —

тривиальную МЗЗ branchNo -> oName. Обратите внимание на то, что эти отношения в форме 4НФ не характеризуются избыточностью данных, а воз-

можность появления аномалий обновления исключена. Например, чтобы зарегистрировать в отделении ВООЗ нового владельца объекта недвижимости, достаточно ввести единственную строку в отношение BranchOvmer.

Пятая нормальная форма (5НФ)

При любой декомпозиции отношения на два других отношения полученные отношения обладают свойством соединения без потерь. Это означает, что полученные отношения можно снова соединить и получить прежнее отношение в исходном виде. Однако бывают случаи, когда требуется выполнить декомпозицию отношения более чем на два отношения. В таких (достаточно редких) случаях возникает необходимость учитывать зависимость соединения без потерь, которая устраняется с помощью пятой нормальной формы (5НФ). кратко рассмотрим сущность зависимости соединения без потерь и ее связь с пятой нормальной формой (5НФ),

Зависимость соединения без потерь

Зависимость соединения без потерь. Свойство декомпозиции, которое гарантирует отсутствие фиктивных строк при восстановлении первоначального отношения с помощью операции естественного соединения.

При разбиении отношений с помощью операции проекции используемый метод декомпозиции определяется совершенно точно. В частности, следует позаботиться о том, чтобы при обратном соединении полученных отношений можно было восстановить исходное отношение. Такая декомпозиция называется декомпозицией с *соединением без потерь* (а также *беспроигрышным* или *неаддитивным* соединением), поскольку при ее выполнении сохраняются все данные исходного отношения, а также исключается создание дополнительных фиктивных строк.

Например, отношения BranchStaff и BranchOwner выше, которые получены путем декомпозиции отношения BranchStaffOwner обладают свойством соединения без потерь. Иначе говоря, исходное отношение BranchStaf fOwner может быть реконструировано путем применения операции естественного соединения к отношениям BranchStaff и BranchOwner. В этом примере выполнена декомпозиция исходного отношения на два отношения, однако

бывают случаи, когда требуется выполнить декомпозицию без потерь с образова-

нием более двух отношений . Именно в таких случаях применимы понятия зависимости соединения без потерь и пятой нормальной формы (5НФ).

Определение пятой нормальной формы (5НФ)

Пятая нормальная форма (5НФ). Отношение без зависимостей соединения.

Пятая нормальная форма (5НФ), которая также называется *проективно-соединительной нормальной формой,* или ПСНФ (Project-Join Normal Form -PJNF), означает, что отношение в такой форме не имеет зависимостей соединения

Рассмотрим показанное на рисунке отношение PropertyItemSupplier.

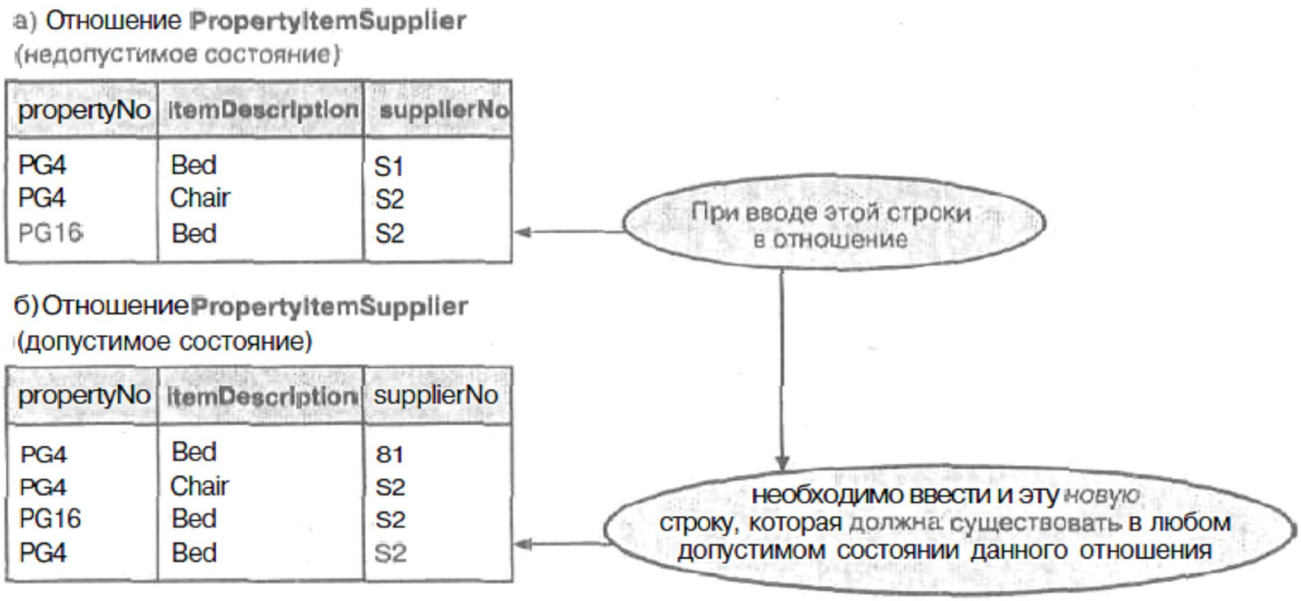


Рисунок *Пример зависимости соединения: а) недопустимое состояние отношения PropertyltemSupplier; б) допустимое состояние отношения PropertyltemSupplier*

Это отношение описывает объекты недвижимости (propertyNo), для которых требуются определенные предметы обстановки (ItemDescription), поставляемые поставщиками (supplierNo) на эти объекты недвижимости (propertyNo). Кроме того, если для какого-то объекта недвижимости (p) требуется некоторый предмет обстановки (i), некий поставщик (s) занимается поставкой таких предметов (i), и поставщик (s) уже выполнил поставку, по меньшей мере, одного предмета обстановки на этот объект недвижимости (p), то этому же поставщику (a) снова будет поручено поставить необходимый предмет обстановки (i) на объект недвижимости (p). В данном примере предположим, что описание предмета обстановки (ItemDescription) однозначно идентифицирует предмет обстановки соответствующего типа.

Чтобы определить, какого рода ограничение распространяется на отношение PropertyltemSupplier (рис а), рассмотрим следующее утверждение.

Если для объекта недвижимости PG4 требуется предмет обстановки 'Bed'(Кровать) (согласно данным в строке 1), поставками на объект недвижимости PG4 занимается поставщик S2 (согласно данным в строке 2), поставщик S2 занимается поставками предметов обстановки 'Bed1

(согласно данным в строке 3), То поставщик S2 должен выполнить поставку предмета обстановки 'Bed1 на объект недвижимости PG4 Этот пример наглядно иллюстрирует циклический характер ограничения, которое распространяется на отношение PropertyltemSupplier. Если это ограничение соблюдается, то строка (PG4, Bed, S2) должна существовать во всех допустимых состояниях отношения PropertyltemSupplier (рис ***б).*** **Это** — пример одной из аномалий обновления, и такое отношение называется содержащим *зависимость соединения* (Join Dependency — JD).

Зависимость соединения. Представляет собой одну из разновидностей зависимости. Например, если рассматривается отношение R с подмножествами атрибутов R, обозначенными *как* A, B, ...; Z, то отношение R удовлетворяет зависимость соединения, если и только если каждое допустимое значение R равно соединению его проекций на атрибуты A, в, ..., 2. ,

Итак, отношение PropertyltemSupplier содержит зависимость соединения и поэтому не находится в пятой нормальной форме (5НФ); Для удаления этой зависимости соединения необходимо выполнить декомпозицию отношения PropertyltemSupplier на три отношения 5НФ, а именно Propertyltem **(Ri),**ItemSupplier (R2) и PropertySupplier (R3), как показано в таблицах ниже

В таком случае можно утверждать, что отношение PropertyltemSupplier в форме (A, B, C) удовлетворяет зависимости соединения JD (R1(A, B), R2 (B, C), R3(A, O).

Отношение Propertyltem в форме 5НФ

propertyNo	ItemDescription
PG4	Bed
PG4	Chair
PG16	Bed

Отношение ItemSupplier в форме 5НФ

ItemDescription	supplierNo
Bed	S1
Chair	S2
Bed	S2

Отношение PropertySupplier в форме 5НФ

propertyNo	supplierNo

PG4	S1
PG4	S2
PG16	S2

Следует отметить, что естественное соединение любых *двух* из этих трех отношений приведет к появлению фиктивных строк, но естественное соединение *всех*

трех отношений приведет к восстановлению отношения PropertyltemSupplier в исходное состояние.

Общее определение денормализации

Напомним, что нормализация *переменной отношения R означает ее замену множеством* таких проекций Rl, R2, ..., Rn, что результатом обратного соединения проекций Rl, R2, ..., Rn обязательно будет значение R. Конечной целью нормализации является *сокращение степени избыточности данных* за счет приведения проекций Rl, R2, ..., Rn к максимально высокому уровню нормализации.

Теперь можно перейти к определению понятия денормализации. Пусть Rl, R2, Rn является множеством переменных отношения. Тогда **денормализацией** этих переменных отношения называется такая замена переменных отношения их соединением R, что для всех возможных значений i (где i = 1, ..., n) выполнение проекции R по атрибутам Ri обязательно снова приводит к созданию значений Ri. Конечной целью денормализации является *увеличение степени избыточности данных* за счет приведения переменной отношения R к более низкому уровню нормализации по сравнению с исходными переменными отношения Rl, R2, ..., Rn. Точнее, преследуется цель сократить количество соединений, которые потребуется выполнять в приложении на этапе прогона, поскольку (в действительности) некоторые из этих соединений уже выполнены заранее в составе работ по проектированию базы данных.

Некоторые проблемы денормализации

Использование понятия денормализации связано с некоторыми вполне очевидными проблемами. Первая из них заключается в том, что начиная денормализацию, трудно сказать, когда ее следует прекратить. В случае выполнения нормализации существуют ясные логические критерии ее продолжения до тех пор, пока не будет достигнута самая высокая из возможных нормальных форм. Следует ли при выполнении денормализации стремиться к тому, чтобы достичь самой низкой из возможных нормальных форм? Безусловно, нет, поэтому не существует никаких *логических* критериев точного определения момента прекращения этого процесса. Иначе говоря, в случае

денормализации прежний подход (применявшийся при нормализации), созданный на основании строго научной и логичной теории, заменяется чисто прагматическим и субъективным подходом.

Второе очевидное затруднение связано с проблемами избыточности и аномалиями обновления, которые возникают из-за того, что приходится иметь дело с не полностью нормализованными переменными отношения. Эти проблемы достаточно подробно обсуждались выше. Но менее очевидной является проблема *выборки* данных, т.е. денормализация может существенно усложнить выполнение некоторых запросов.

Третья, и самая главная, проблема формулируется следующим образом. (Это относится к "правильной" денормализации, т.е. к денормализации, которая выполняется только на физическом уровне, а также к тому типу денормализации, которую иногда приходится осуществлять в современных продуктах SQL.) Когда речь идет о том, что денормализация "способствует достижению высокой производительности", фактически подразумевается, что она способствует достижению высокой *производительности некоторых конкретных приложений.* Любая выбранная физическая структура, которая прекрасно подходит для одних приложений с точки зрения их производительности, может оказаться совершенно непригодной для других.

Обоснование необходимости введения контролируемой избыточности

Цель. Определение необходимости ввода контролируемой избыточности за счет ослабления условий нормализации для повышения производительности системы. Нормализация — это процедура определения того, какие атрибуты связаны в отношении. Одна из основных задач при разработке реляционной базы — объединение в одном отношении тех атрибутов, между которыми существует функциональная зависимость. Результатом нормализации является логическая модель базы данных — структурно цельная система с минимальной избыточностью. Но в некоторых случаях оказывается, что нормализованная модель не обеспечивает максимальной производительности при обработке данных. Следовательно, при некоторых обстоятельствах может оказаться необходимым ради повышения производительности пожертвовать частью той выгоды, которую обеспечивает модель с полной нормализацией. К денормализации следует прибегать лишь тогда, когда нормализованная база не удовлетворяет требованиям, предъявляемым к производительности системы. Мы не призываем к полному отказу от нормализации в логической модели базы данных: нормализация позволяет однозначно зафиксировать назначение каждого атрибута в реляционной базе, а это может оказаться решающим фактором при создании эффективно работающей системы. Кроме того, необходимо учесть и следующие особенности:
• денормализация может усложнить физическую реализацию системы;
• денормализация часто приводит к снижению гибкости;
• денормализация может ускорить чтение данных, но при этом замедлить обновление записей.
Формально *денормализацию* можно определить как модификацию реляционной модели, при которой степень нормализации модифицированного отношения становится ниже, чем степень нормализации, по меньшей мере, одного из исходных отношений. Термин "денормализация" будет также применяться в случае, когда два отношения объединяются

в одно и полученное отношение остается нормализованным, однако содержит больше пустых значений, чем исходные отношения. Некоторые авторы определяют денормализацию как *модификацию реляционной модели с учетом требований эксплуатации.*

Существует следующее эмпирическое правило: если производительность системы не удовлетворяет поставленным требованиям и проектируемое отношение имеет низкую скорость обновления при большой частоте запросов, денормализация реляционной модели может оказаться оправданной. Основная информация, касающаяся данного этапа проектирования, содержится в таблице соответствия транзакций и отношений, которая создается на этапе анализа транзакций.

На этом этапе строится таблица соответствия отношений (таблиц) и наиболее часто выполняемых транзакций (или наиболее критичных) .

Транзакция — это логическая единица работы; она начинается с выполнения операции BEGIN TRANSACTION и заканчивается операцией COMMIT (выполнение всех действий транзакции) или ROLLBACK(откат всех действий транзакции).

В данном случае транзакция рассматривается действие или последовательность действий, выполняемых одним и тем же пользователем (или прикладной программой), которые получают: доступ к базе данных или изменяют ее содержимое.

Таблица имеет вид:

Транзакция отношение	Транзакция 1				Транзакция 2				Транзакция…				Транзакция N			
	I	R	U	D	I	R	U	D	I	R	U	D	I	R	U	D
Отношение 1	X											X				
Отношение 2						X										
Отношение …			X											X		
Отношение M										X					X	

Где I — вставка; R — чтение (выборка); U — обновление; D — удаление,
а X- применение операции

Эта таблица наглядно показывает, к каким отношениям обращаются транзакции, выполняемые в рассматриваемой базе данных. Используя эти сведения, можно выделить в реляционной модели возможные области денормализации, а также оценить последствия денормализации для остальной части базы.

Если для некоторых транзакций требуется частый доступ к определенным отношениям, то приобретает особую важность задача изучения характера их выполнения. Например, если в определенное время требуется только одни транзакции, а затем другие, то риск возникновения проблем, связанных со снижением производительности, уменьшается. А если периоды максимальной частоты применения различных транзакций совпадают, потенциальные проблемы производительности можно решить, более тщательно изучая такие транзакции для определения того, какие изменения могут быть внесены в структуру отношений для повышения производительности

Если говорить конкретнее, на этом этапе рассматривается возможность дублирования отдельных атрибутов или объединения нескольких отношений в одну таблицу с целью сокращения числа запросов на соединение отношений.

Рассмотрим возможность применения денормализации в ситуациях, когда требуется ускорить выполнение часто повторяющихся или важных транзакций.

1. Объединение таблиц со связями типа "один к одному" (1:1).
2. Дублирование неключевых атрибутов в связях "один ко многим" (1:*) для уменьшения количества соединений.
3. Дублирование атрибутов внешнего ключа в связях "один ко многим" (1:*) для уменьшения количества соединений.
4. Дублирование атрибутов в связях "многие ко многим" (1:*) для уменьшения количества соединений.

5. Введение повторяющихся групп полей.
6. Объединение справочных таблиц с базовыми таблицами.
7. Создание таблиц из данных, содержащихся в других таблицах.

Тема 1.3. Отношения и их свойства, ключи отношений. Абстрактные операции манипулирования данными.

Отношение представляет собой множество элементов, называемых кортежами.

Наглядной формой представления отношения является привычная для человеческого восприятия двумерная таблица.
Логическое представление реляционных баз данных упрощается созданием связей между данными на основе (логической) конструкции, называемой таблицей. Под *таблицей* понимается двумерная структура, состоящая из строк и столбцов. Пользователь должен понимать, что *таблица содержит группу связанных сущностей,* т. е. набор сущностей; по этой причине термины *набор сущностей* и *таблица* чаще всего означают одно и то же. Таблица также называется *отношением (relation),* поскольку создатель реляционной модели Э. Ф. Кодд (E. F. Codd) использовал термин "отношение" как синоним слова "таблица".
Таблица имеет строки (записи) и столбцы (колонки). Каждая строка таблицы имеет одинаковую структуру и состоит из полей. Строкам таблицы соответствуют кортежи, а столбцам — атрибуты отношения.
Реляционная модель основана на математическом понятии *отношения,* физи- ческим представлением которого является *таблица.*
Отношение. Плоская таблица, состоящая из столбцов и строк.;
В любой реляционной СУБД предполагается, что пользователь воспринимает базу данных как набор таблиц. Однако следует подчеркнуть, что это восприятие относится только к логической структуре базы данных, т.е. к внешнему и к концептуальному уровням архитектуры ANSI-SPARC, которая рассматривалась нами в разделе 2.1. Подобное восприятие не относится к физической структуре базы данных, которая может быть реализована с помощью различных структур хранения (приложение В),

Свойства реляционной таблицы(отношения)

1. Таблица представляет собой двумерную структуру, состоящую из строк и столбцов
2. Каждая строка таблицы (кортеж, tuple) представляет собой отдельную сущность внутри набора сущностей
3. Каждый столбец таблицы представляет собой атрибут, и у каждого столбца есть свое имя
4. На каждом пересечении строки и столбца имеется единственное значение
5. Каждая таблица должна иметь атрибут или несколько атрибутов, уникально идентифицирующих каждую строку
6. Все значения в столбце должны отображаться в одинаковом формате. Например, если атрибуту присваивается формат целого, то все значения в столбце, представляющем данный атрибут должны быть целыми
7. Каждый столбец имеет определенный диапазон значений, называемый доменом атрибута
8. Порядок следования строк и столбцов не существенен.

В реляционных базах данных используется строгая терминология. К сожалению, в среду баз данных иногда проникает терминология систем файлов. Таким образом, мы можем обнаружить, что строки иногда называют записями, а столбцы полями.

Домен. Набор допустимых значений одного или нескольких атрибутов. Домены представляют собой чрезвычайно мощный компонент реляционной модели. Каждый атрибут реляционной базы данных определяется на некотором домене. Домены могут отличаться для каждого из атрибутов, но два и более ат- рибутов могут определяться на одном и том же домене.

Ключи

Говоря о связях, нельзя не остановиться на понятии ключей и ключевых атрибутов. *Первичным ключом* называется атрибут или группа атрибутов, однозначно идентифицирующих каждый экземпляр сущности. Нередко возможны несколько вариантов выбора первичного ключа. Например, в небольшой организации первичными ключами сущности "сотрудник" могут быть как табельный номер, так и комбинация фамилии, имени и отчества (при уверенности, что в организации нет полных тезок), либо номер и серия паспорта (если паспорта есть у всех сотрудников) . В таких случаях при выборе первичного ключа предпочтение отдается наиболее простым ключам (в данном примере - табельному номеру). Другие кандидаты на роль первичного ключа называются *альтернативными ключами.*

Ключи реляционных баз данных

Тип ключа	**Определение**
Суперключ (superkey)	Атрибут(или комбинация атрибутов), уникально идентифицирующих каждую сущность в таблице
Потенциальный ключ (candidate key)	Минимальный суперключ. Суперключ, который не содержит подмножества атрибутов, которое само по себе является суперключом.
Первичный ключ (primary key)	Потенциальный ключ, выбранный для уникальной идентификации всех остальных значения атрибутов в любой строке. Не может содержать пустых значений
Вторичный ключ (secondary key)	Атрибут(или комбинация атрибутов), используемый исключительно в целях поиска данных
Внешний ключ (foreign key)	Атрибут(или комбинация атрибутов) в одной таблице, значения которого должны или совпадать со значениями первичного ключа другой таблицы, или быть пустыми

Правила целостности

ЦЕЛОСТНОСТЬ НА УРОВНЕ СУЩНОСТИ

Требование Все элементы первичного ключа уникальны и никакая часть первичного ключа не может быть пустой (null)

Назначение Гарантирует, что каждая сущность (логический объект) будет иметь уникальную идентификацию, а значения внешнего ключа могут должным образом ссылаться на значения первичного ключа
Пример Счет не может иметь несколько дублирующихся значений и не может иметь пустое значение (null). Короче говоря, все счета уникально идентифицируются своим номером
ЦЕЛОСТНОСТЬ НА УРОВНЕ ССЫЛКИ
Требования Внешний ключ может иметь или пустое значение (если только он не является частью первичного ключа данной таблицы), или значение, совпадающее со значением первичного ключа в связанной таблице. (Каждое непустое значение внешнего ключа должно ссылаться на существующее значение первичного ключа.)
Назначение Допускается, что атрибут не имеет соответствующего значения, но атрибут не может принимать недопустимые значения. Выполнение правила целостности на уровне ссылки делает невозможным удаление строки в одной таблице, где первичный ключ имеет обязательное соответствие со значением внешнего ключа в другой таблице
Пример Клиенту может быть не назначен (еще) торговый агент, но невозможно назначить клиенту несуществующего агента

Теперь о *внешних ключах*:

- Если сущность C связывает сущности A и B, то она должна включать внешние ключи, соответствующие первичным ключам сущностей A и B.
- Если сущность B обозначает сущность A, то она должна включать внешний ключ, соответствующий первичному ключу сущности A.
- Здесь для обозначения любой из ассоциируемых сущностей (стержней, характеристик, обозначений или даже ассоциаций) используется новый обобщающий термин "Цель" или "Целевая сущность".
- Таким образом, при рассмотрении проблемы выбора способа представления ассоциаций и обозначений в базе данных основной вопрос, на который следует получить ответ: "Каковы внешние ключи?". И далее, для каждого внешнего ключа необходимо решить три вопроса:
- 1. Может ли данный внешний ключ принимать неопределенные значения (NULL-значения)? Иначе говоря, может ли существовать некоторый экземпляр сущности данного типа, для которого неизвестна целевая сущность, указываемая внешним ключом? В случае поставок это, вероятно, невозможно – поставка, осуществляемая неизвестным поставщиком, или поставка неизвестного продукта не имеют смысла. Но в случае с сотрудниками такая ситуация однако могла бы иметь смысл – вполне возможно, что какой-либо сотрудник в данный момент не зачислен вообще ни в какой отдел. Заметим, что ответ на данный вопрос не зависит от прихоти проектировщика базы данных, а определяется фактическим образом действий, принятым в той части реального мира, которая должна быть представлена в рассматриваемой базе данных. Подобные замечания имеют отношение и к вопросам, обсуждаемым ниже.
- 2. Что должно случиться при попытке УДАЛЕНИЯ целевой сущности, на которую ссылается внешний ключ? Например, при удалении поставщика, который осуществил по крайней мере одну поставку. Существует три возможности:

КАСКАДИРУЕТСЯ	Операция удаления "каскадируется" с тем, чтобы удалить также поставки этого поставщика.
ОГРАНИЧИВАЕТСЯ	Удаляются лишь те поставщики, которые еще не осуществляли

поставок. Иначе операция удаления отвергается.

УСТАНАВЛИВАЕТСЯ Для всех поставок удаляемого поставщика NULL-значение внешний ключ устанавливается в неопределенное значение, а затем этот поставщик удаляется. Такая возможность, конечно, неприменима, если данный внешний ключ не должен содержать NULL-значений.

- 3. Что должно происходить при попытке ОБНОВЛЕНИЯ первичного ключа целевой сущности, на которую ссылается некоторый внешний ключ? Например, может быть предпринята попытка обновить номер такого поставщика, для которого имеется по крайней мере одна соответствующая поставка. Этот случай для определенности снова рассмотрим подробнее. Имеются те же три возможности, как и при удалении:

КАСКАДИРУЕТСЯ Операция обновления "каскадируется" с тем, чтобы обновить также и внешний ключ впоставках этого поставщика.

ОГРАНИЧИВАЕТСЯ Обновляются первичные ключи лишь тех поставщиков, которые еще не осуществляли поставок. Иначе операция обновления отвергается.

УСТАНАВЛИВАЕТСЯ Для всех поставок такого поставщика NULL-значение внешний ключ устанавливается в неопределенное значение, а затем обновляется первичный ключ поставщика. Такая возможность, конечно, неприменима, если данный внешний ключ не должен содержать NULL-значений.

- Таким образом, для каждого внешнего ключа в проекте проектировщик базы данных должен специфицировать не только поле или комбинацию полей, составляющих этот внешний ключ, и целевую таблицу, которая идентифицируется этим ключом, но также и ответы на указанные выше вопроса (три ограничения, которые относятся к этому внешнему ключу).

Тема 1.4. Реализация отношений в базах данных, типы данных в языке SQL, операторы языка SQL для создания, удаления, модификации таблиц базы данных

История языка SQL

В языке SQL вместо терминов *отношение* и *переменная отношения* используется термин **таблица,** а вместо терминов *кортеж* и *атрибут* — **строка** и **столбец.**
Именно эти термины используется в стандарте языка SQL и в поддерживающих его продуктах. Необходимо подчеркнуть, что SQL — язык очень большого объема. Его спецификация содержит свыше 2000 страниц, не считая больше 300 страниц исправлений В языке SQL имеются операции как определения данных, так **и** манипулирования ими. Стандарт SQL включает спецификации стандартного каталога, именуемого в нем **информационной** схемой. Знакомые нам термины *каталог* и *схема* действительно используются в языке SQL, но с особым смыслом, характерным только для языка SQL. Вообще говоря, **каталог** в языке SQL состоит из дескрипторов (метаданных) для отдельной базы данных3, а схема состоит из дескрипторов той части базы данных, которая принадлежит отдельному пользователю. Другими словами, в системе может быть

любое число каталогов (по одному для каждой базы данных), каждый из которых делится на произвольное число
схем. Однако каждый каталог должен содержать одну и только одну схему с именем INFORMATION_SCHEMA (информационная схема), которая с точки зрения пользователя и является схемой (как уже указывалось), выполняющей функции обычного каталога. Таким образом, информационная схема состоит из набора таблиц SQL, содержимое которых фактически отражает (точно заданным способом) все определения из всех остальных схем рассматриваемого каталога. Точнее говоря, информационная схема по умолчанию содержит набор представлений гипотетической "схемы: определения".
В большинстве продуктов SQL операторы языка SQL могут выполняться как **непосредственно** (т.е. интерактивно, с подключенного терминала), так и в виде части прикладной программы (т.е. операторы SQL могут быть **внедренными,** а значит, могут смешиваться с операторами базового языка этой программы). Фундаментальный принцип, лежащий в основе технологии внедрения операторов **SQL,** называется **принципом двухрежимности.** Он заключается в том, что *любое выражение **SQL,** которое можно использовать интерактивно, можно применять и путем внедрения **в** прикладную программу.*
За время своего существования SQL пережил немало изменений, редакций и выхода стандартов, не говоря уже о различных реализациях этих стандартов различными СУБД (называемыми по аналогии с естественными языками диалектами). Основные стандарты приведены в таблице ниже.
Стандарты языка SQL

Год	Название	Другое название	Изменения
1986	SQL-86	SQL-87	Первый вариант стандарта, принятый институтом ANSI и одобренный ISO в 1987 году.
1989	SQL-89	FIPS 127-1	Немного доработанный вариант предыдущего стандарта
1992	SQL-92	SQL2,FIPS-127-2	Значительные изменения (ISO 9075); уровень Entry Level страндарта SQL-92 был принят как стандарт FIPS-127-2
1999	SQL:1999	SQL3	Добавлена поддержка регулярных выражений, рекурсивных запросов, поддержка триггеров, базовые процедурные расширения, нескалярные типы данных и некоторые объектно-ориентированные возможности.
2003	SQL:2003		Введены расширения для работы с XML данными, оконные функции(применяемые для работы с OLAP-базами данных), генераторы последовательностей и основанные на них типы данных, появление команды MERGE.
2006	SQL:2006		Функциональность работы с XML-данными знгачительно расширена. Появилась возможность совместно использовать в запросаз SQL и XQuery
2008	SQL:2008		Улучшены возможности оконных функций, устранены некоторые неоднозначности стандарта SQL:2003
2011	SQL:2011		Delete в операторе MERGE, изменение данных в операторе Select, , изменение синтаксиса вызова процедур, стандартизация возврата

			запросом процента от всех строк (не количества), возможность отклбючать ограничения на данные в таблице(CHECK? UNIQUE? REFERRENCE) ,улучшение работы оконных функций
2016	SQL:2016		Поддержка формата данных JSON, функция LISTAGG (перечисление через запятую значений поля),распознавание шаблонов строк, (регулярные выражения применительно к строкам) , в т.ч работа с оконными функциями

Типы данных в языке SQL

В таблице перечислены скалярные типы данных языка SQL, которые определены стандартом ISO. В некоторых случаях в целях упрощения манипулирования и преобразования, а также из-за сходства основных свойств данные типов *character* и *bit* объединяются под названием "строковые типы данных", а данные типов *exact numeric* и *approximate numeric* — под названием "числовые типы данных". В стандарте SQL3 определены также большие символьные и двоичные объекты.
Обратите внимание, что реализация в различных СУБД может быть различной
Таблица . Основные типы данных языка SQL, определенные в стандарте ISO(SQL -92)

Тип данных	Объявления	Реализация Microsoft SQL Server	Реализация MySQL
boolean (Логический)	BOOLEAN		BOOL, BOOLEAN, TINYINT(1).
character (Символьный)	CHAR	CHAR[(M)], NCHAR[(M)](unicode)	CHAR [(M)]
	VARCHAR	VARCHAR[(M)], NVARCHAR[(M)](unicode)	VARCHAR[(M)]
bit (Битовый)	BIT, BIT VARYING	BIT	BIT BIT[(M)]
exact numeric (Точные числа)	NUMERIC	NUMERIC[(M[,D])]	DECIMAL[(M[,D])] [UNSIGNED] [ZEROFILL]
	DECIMAL	DECIMAL[(M[,D])]	DECIMAL[(M[,D])] [UNSIGNED] [ZEROFILL]
	SMALLINT	SMALLINT	SMALLINT[(M)] [UNSIGNED] [ZEROFILL]
	INTEGER	INT	INT[(M)] [UNSIGNED] [ZEROFILL] INTEGER[(M)] [UNSIGNED] [ZEROFILL]
	TINYINT	TINYINT	TINYINT[(M)] [UNSIGNED] [ZEROFILL] -128 .. 127 или 0..255
	BIGINT	BIGINT	BIGINT (8 байт)
approximate numeric	FLOAT	FLOAT[(N)]	FLOAT[(M,D)] [UNSIGNED] [ZEROFILL]

(Округленные числа)			REAL[(M,D)] [UNSIGNED] [ZEROFILL](если задан режим REAL_AS_FLOAT)
	REAL	REAL (эквивалентно Float(24))	
	DOUBLE PRECISION	DOUBLE PRECISION	DOUBLE[(M,D)] [UNSIGNED] [ZEROFILL], REAL[(M,D)] [UNSIGNED] [ZEROFILL](если не задан режим REAL_AS_FLOAT)
datetime (Дата/время)	DATE	DATE	DATE
	TIME	TIME	TIME
	TIMESTAMP	DATETIME TIMESTAMP	DATETIME TIMESTAMP
interval (Интервал)	INTERVAL		С другим синтаксисом , но используются TIME и TIMESTAMP
LOB (Большой объект)	CHARACTER LARGE OBJECT,	TEXT, VARCHAR(MAX)	TINYTEXT, TEXT, MEDIUMTEXT, and LONGTEXT
	BINARY LARGE OBJECT	VARBINARY(MAX) VARBINARY(n)	TINYBLOB, BLOB, MEDIUMBLOB, LONGBLOB
XML (Правильный XML документ)		XML	XML

Логические данные (тип boolean)
Логические данные состоят из различимых истинностных значений TRUE (истинный) и FALSE (ложный). Логические данные поддерживают также истинностное значение UNKNOWN (неопределенный), заданное как значение NULL, если применение неопределенных значений не запрещено ограничением NOT NULL. Все значения данных логического типа и истинностные значения SQL могут совместно применяться в операторах сравнения и присваивания. Значение TRUE в арифметических операторах сравнения больше значения FALSE, а любое сравнение, в котором участвует значение NULL или истинностное значение UNKNOWN, возвращает результат UNKNOWN.
Символьные данные (тип character)
Символьные данные состоят из последовательностей символов, входящих в определенный создателями СУБД набор символов. Как правило, длина ограничена 255 символами, при превышении используются другие типы (например Text) Поскольку наборы символов являются специфическими для различных диалектов языка SQL, перечень сим- волов, которые могут входить в состав значений данных символьного типа, также зависит от конкретной реализации. В настоящее время чаще всего используются наборы символов ASCII и EBCDIC. Для определения данных символьного типа применяется следующий формат:

CHAR(length)| VARCHAR(length), где

length используется для указания максимального количества символовсимволов, которые могут быть помещены в данный столбец (по умолчанию принимается значение 1)

Обратите внимание, что между этими типами данных есть разница.

CHAR(length) –строка фиксированной длины, где строка меньшей длины будет дополнена пробелами справа.

VARCHAR(length) –строка переменной длины, где строка меньшей длины будет выглядеть как введена.

	Строка меньшей длины	Строка точной длины	Строка большей длины
Вводимая строка	Лес	Строчка	понедельник
CHAR(7)	“Лес_ _ _ _”	“Строчка”	“понедель”
VARCHAR(7)	“Лес”	“Строчка”	“понедель”

Битовые данные (тип bit)

Битовый тип данных используется для определения битовых строк, т.е. по- следовательности двоичных цифр (битов), каждая из которых может иметь зна- чение либо 0, либо 1. Для опредеяения данных битового типа используется фор- мат, сходный с определением символьных данных:

BIT [IVARYING] [*length]*.

Например, для сохранения битовой строки с фиксированной длиной и значе- нием ' 0011 ' может быть объявлен столбец bitstring:

bitString BIT(4)

Точные числовые данные (тип **exact numeric)**

Тип точных числовых данных используется для определения чисел, которые имеют точное представление в компьютере. Числа состоят из цифр и необязательных символов (десятичной точки, знака "плюс" или "минус"). Данные точного числового типа определяются *значностъю* (precision) и *длиной, дробной части* (scale). Значность задает общее количество значащих десятичных цифр числа, в которое входят длина целой и дробной частей, но без учета самой десятичной точки. Дробная часть указывает количество дробных десятичных разрядов числа. Например, точное число -12,345 имеет значность, равную 5 цифрам, и дробную часть длиной 3. (Типы NUMERIC и DECIMAL предназначены для хранения чисел в десятичном формате. По умолчанию длина дробной части равна нулю, а принимаемая по умолчанию значность зависит от реализации.)

NUMERIC -[precision - [, scale]]

DECIMAL [precision [, scale]]

Столбец salary таблицы Staff может быть объявлен следующим образом:

salary DECIMAL(7,2)

В этом случае максимальное значение заработной платы составит 99 999,99 фунтов стерлингов,

Особой разновидностью точных чисел являются целые числа. Они делятся по диапазонам чисел, которые могут представлять, который определяется количеством битов (чаще измеряют в байтах (по 8 бит)), занимаемым числом. Если число со знаком, то оно имеет диапазон , $[-2^{n-1} .. 2^{n-1} -1]$

а если без знака . то диапазон $[0 .. 2^n -1]$,

Где n – количество битов.

Существует несколько способов определения данных точного числового типа:

Наименование	Количество байтов	диапазон
SMALLINT	2 байта	$-2^{15} .. 2^{15}-1$, т.е. -32768.. 32767 $0 .. 2^{16}-1$, т.е. 0 ..65535

INTEGER*(может* быть сокращено *до* INT)	4 байта	-2^{31} ..2^{31}-1, т.е. -2147483648 .. 2147483647 0 ..2^{32}-1,т.е 0 .. 4294967295

Часто СУБД дополнительно реализуют целые на 1 байт (-128 ... 127 или 0 .. 255) и целые на 8 байт (часто, называемые INT64)

Округленные числовые данные (тип approximate numeric)
Тип округленных числовых данных используется для описания данных, ко- торые нельзя точно представить в компьютере, например действительных чисел. Для представления округленных чисел или чисел с плавающей точкой использу- ется экспоненциальная система обозначений, в которой число записывается с помощью мантиссы, умноженной на определенную степень десяти (порядок), например: 10Е3, +5.2Е6, -0.2Е-4. Существует несколько способов определения данных с типом округленных числовых данных:
FLOAT [precision]
REAL
•DOUBLE PRECISION
Параметр *precision* задает значность мантиссы. Значность определений типа REAL и DOUBLE PRECISION зависит от конкретной реализации.
Дата и время (тип datetime)
Тип данных "дата/время" используется для определения моментов времени с некоторой установленной точностью. Примерами являются даты, отметки времени и время суток. Стандарт ISO разделяет тип данных "дата/время" на подтипы YEAR (Год), MONTH (Месяц), DAY (День), HOUR (Час), MINUTE (Минута), SECOND (Секунда), TIMEZONE_HOUR (Зональный час) и TIMEZONE_MINUTE (Зональная минута). Два последних типа определяют час и минуты сдвига зонального времени по отношению к всеобщему скоординированному времени (прежнее название — гринвичское время). Поддерживаются три типа полей даты/времени.
DATE
TIME [timePrecision] [WITH TIME ZONS]
TIMESTAMP [timePrecision] [WITH TIME ZONE]
Тип данных DATE используется для хранения календарных дат, включающих поля YEAR, MONTH и DAY. Тип данных TIME используется для- хранения отметок времени, включающих поля HOUR, MINUTE и SECOND. Тип данных TIMESTAMP служит для совместного хранения даты и времени. Параметр *timePrecision* за- дает количество дробных десятичных знаков, определяющих точность представ- ления значений в поле SECOND. Если этот параметр опущен, по умолчанию его значение для столбцов типа TIME принимается равным нулю (т.е. сохраняется целое количество секунд), тогда как для полей типа TIMESTAMP он принимается равным 6 (т.е. отметки времени сохраняются с точностью до микросекунд). Наличие ключевого слова WITH TIME ZONE определяет использование полей TIMEZONE_HOUR и TIMEZONE_MINUTE. Например, столбец date таблицы Viewing, представляющий дату (день, месяц и год) осмотра клиентом сдаваемого в аренду объекта, может быть определен следующим образом:
viewDate DATE
Интервальный тип данных interval
Данные с интервальным типом используются для представления периодов времени. Любой интервальный тип данных состоит из набора полей: YEAR, MONTH, DAY, HOUR, MINUTE и SECOND. Существуют два класса данных с интервальным ти- пом: интервалы *год-месяц* и интерналы *сутки-время суток.* В первом случае данные включают только два поля — YEAR и/или MONTH. Данные второго типа могут состоять из произвольной последовательности полей DAY, HOUR, MINUTE, SECOND.

Данные интервального типа определяются следующим образом:
INTERVAL -{{startField TO.endField} singleDatetimeField}
StartField = YEAR MONTH | DAY | HOUR | MINUTE
[(intervaiLeadingFieldPrecisicm)]
endField = YEAR | MONTH | DAY j .HOUR-.. | MINUTE | SECOND
[(fractionalSecondsPrecision)]
singleDatetimeField = startPield |;SECOND
[(intervaiLeadingFieldPrecisicm)]
[,fractionalSecondsPrecision])]
Для параметра startField должна быть всегда указана размерность первого поля (intervalLeadingFieldPrecision), которая по умолчанию принимается равной двум. Например:
INTERVAL YEAR(2) TO MONTH
Это объявление описывает интервал времени, значение которого может нахо- диться между 0 годом, 0 месяцем и 99 годом, 11 месяцем. Еще один пример:
INTERVAL HOUR TO SECOND(4)
Это объявление описывает интервал времени, значение которого может изме- няться от 0 часов, 0 минут, о секунд до 99 часов, 59 минут 59.9999 секунды. (Число дробных десятичных знаков для секунд установлено равным 4.)

Операторы языка SQL для создания, удаления, модификации таблиц базы данных

Язык SQL включает два компонента:

- язык **определения данных** (Data Definition Language — DDL)
- языка **манипулирования данными** (Data Manipulation Language —DML).

Соответственно DDL занимается созданием, удалением и модификацией объектов базы данных (сама база, таблицы, представления, хранимые процедуры, триггеры и т.д), А DML – вставкой, удалением и модификацией данных нутрии этих объектов.
Для лучшего запоминания приведем ключевые слова операторов языка

	определения данных (Data Definition Language — DDL)	**манипулирования данными** (Data Manipulation Language —DML)
Создание/вставка	CREATE Создание объекта БД	INSERT Вставка данных в таблицу
Изменение	ALTER Изменение (структуры) объекта БД	UPDATE Изменение данных в таблице
Удаление	DROP Удаление объекта БД	DELETE Удаление данных из таблицы

Оператор создания таблиц языка SQL

Общая схема оператора выглядит следующим образом:
CREATE TABLE TableName
((columName dataType .[[NOT] NULL] [UNIQUE] [DEFAULT defaultoption]
[PRIMARY KEY] [CHECK (searchCondition)]]
[PRIMARY KEY(ListOf Columns) ,]
[UNIQUE(ListOfColumns)] [,…]
[FOREIGN KEY (listOfForeignKeyColumns)
REFERENCES ParentTableName [(listOfCandidateKeyColumns)] ,
[MATCH [PARTIAL | FULL]]
[ON UPDATE. referentialAction]
[ON DELETE referentialAction] [, ...]]

[CHECK (searchCondition)] [, . , ,] })

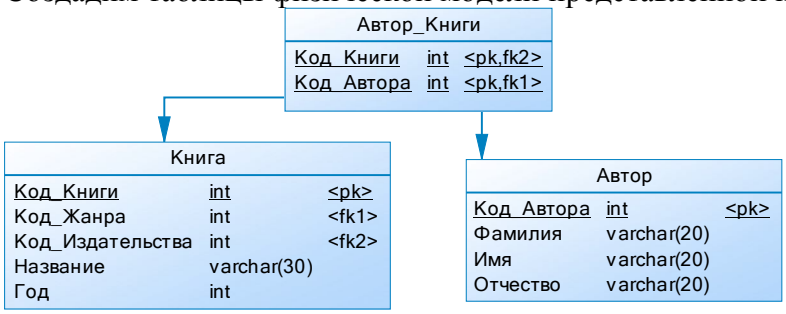
В оператор создания таблиц входят (курсив если необязательно):

1. Имя создаваемой таблицы TableName
2. Описания столбцов таблицы (
    a. имя столбца, columName
    b. тип данных столбца, dataType
    c. *обязательность столбца*, .[NOT NULL]| .[NULL]
    d. *ограничения на данные столбца*, [UNIQUE] [CHECK (searchCondition)]]
    e. *значение столбца по умолчанию*[DEFAULT defaultoption]
    f. *признак первичного ключа, если первичный ключ не является составным* [PRIMARY KEY]
3. *ограничения на данные столбцов (нескольких сразу)* [CHECK (searchCondition)]| [UNIQUE(ListOfColumns)]
4. *состав первичного ключа* [PRIMARY KEY(ListOf Columns)]

ограничения на внешние ключи (ссылка на родительские таблицы и ограничения ссылочной целостности) [FOREIGN KEY (listOfForeignKeyColumns)
REFERENCES ParentTableName [(listOfCandidateKeyColumns)] ,
[MATCH [PARTIAL | FULL]]
[ON UPDATE. referentialAction]
[ON DELETE referentialAction] [, ...]]
[CHECK (searchCondition)] [, . , ,] })

Конструкция PRIMARY KEY определяет один или несколько столбцов, которые образуют первичный ключ таблицы. Если эта конструкция предусмотрена в диалекте SQL, реализованном в конкретной базе данных, то она должна применяться при создании каждой таблицы. По умолчанию для всех столбцов, представляющих первичный ключ, предусмотрено применение ограничения NOT NULL.
При создании таблицы разрешено использование только одной конструкции PRIMARY KEY. База данных отвергает все попытки выполнения операций INSERT или UPDATE, которые влекут за собой создание строки с повторяющимся значением в столбце (столбцах) PRIMARY KEY. Таким образом, в базе данных гарантируется уникальность значений первичного ключа.
Конструкции PRIMARY KEY ,FOREIGN KEY, CHECK, UNIQUE называют ограничениями таблицы и им можно задать имя , предваряя соответствующую конструкцию словом CONSTRAINT за которым следует имя ограничения. Это делается для того чтобы позже работать с ними в операторе изменения таблиц.
В конструкции FOREIGN KEY определяется внешний ключ (дочерней) таблицы и ее связь с другой (родительской) таблицей. Эта конструкция позволяет реализовать ограничения ссылочной целостности и состоит из следующих частей:
• Список *ListOfForeignKeyColumns,* содержащий имена одного или нескольких столбцов создаваемой таблицы, которые образуют внешний ключ.
• Вспомогательная конструкция REFERENCES, указывающая на родительскую таблицу (т.е. таблицу, в которой определен соответствующий потенциальный ключ). Если список *ListOfCandidateKeyColumns* опущен, предполагается, что определение внешнего ключа совпадает с определением первичного ключа родительской таблицы. В таком случае родительская таблица должна иметь в своем операторе CREATE TABLE конструкцию PRIMARY KEY.
• Необязательное правило обновления (ON UPDATE) для определения взаимо- связи между таблицами, которое указывает, какое действие (referentialAction) должно выполняться при обновлении в родительской таблице потенциального ключа, соответствующего внешнему ключу дочерней таблицы. В качестве параметра

referentialAction можно указать CASCADE (каскадирование –повтор действия), SET NULL, SET DEFAULT (установка пустого значения или значения по умолчанию)или NO ACTION (запрет обновления первичных ключей родительской таблицы). Если КОНСТРУКЦИЯ ON UPDATE опущена, то по умолчанию подразумевается, что никакие действия не выполняются, в соответствии со значением NO ACTION.
• Необязательное правило удаления (ON DELETE) для определения взаимосвязи между таблицами, которое указывает, какое действие (referentialAction) должно выполняться при удалении строки из родительской таблицы, которая содержит потенциальный ключ, соответствующий внешнему ключу дочерней таблицы. Определение параметра referentialAction совпадает с определением такого же параметра для правила ON UPDATE. (NO ACTION означает в этом случае запрет удаления данных из родительской таблицы, на которые есть ссылки/)
• По умолчанию ограничение ссылочной целостности удовлетворяется, если любой компонент внешнего ключа имеет значение NULL или в родительской таблице есть соответствующая строка. Опция MATCH позволяет ввести дополнительные ограничения, касающиеся применения значений NULL во внешнем ключе. Если задана опция MATCH FULL, то либо все компоненты внешнего ключа должны быть пусты (NULL), либо все должны иметь непус- тые значения. А если задана опция MATCH PARTIAL, то либо все компонен- ты внешнего ключа должны быть пусты (NULL), либо в родительской таб- лице должна существовать хотя бы одна строка, способная удовлетворить это ограничение, если все остальные значения NULL были подставлены правильно. Некоторые авторы утверждают, что в ограничениях ссылочной целостности следует применять только опцию MATCH FULL.
В операторе создания таблицы может быть задано любое количество конструкций FOREIGN KEY. Конструкции CHECK и CONSTRAINT позволяют определять дополнительные ограничения. Если конструкция CHECK используется в качестве ограничения столбца, то она может ссылаться только на определяемый столбец. Ограничения фактически контролируются после применения каждого оператора SQL к таблице, на которой они заданы, но такая проверка может быть отложена до окончания той транзакции, в состав которой входит текущий оператор SQL.
Пример
Создадим таблицы физической модели представленной ниже



Сперва создадим родительские, независимые таблицы Автор и Книга. В таблице автор запретим полных тёзок

```
Create table Автор
(Код_Автора int NOT NULL PRIMARY KEY,
Фамилия Varchar(20) NULL,
Имя Varchar(20) NULL,
Отчество Varchar(20) NULL,
UNIQUE(Фамилия, Имя, Отчество))
```

Теперь книгу, в которой проверим, что год издания не меньше 1448(Гутенберг, книгопечатание)

```
Create table Книга
(Код_Автора int NOT NULL PRIMARY KEY,
```

Код_Жанра int NOT NULL,
Код_Издательства int NOT NULL,
Год int CHECK(Год>=1448)
)
Или
Create table Книга
(Код_Книги int NOT NULL PRIMARY KEY,
Код_Жанра int NOT NULL,
Код_Издательства int NOT NULL,
Год int ,
CHECK(Год>=1448)
)
Теперь создадим таблицу раскрывающую связь многие-ко-многим Автор_книги, при этом Связка автор_книги не существует без автора или книги, поэтому при удалении будем ставить декларативную каскадную ссылочную целостность.
Create table Автор_книги
(Код_Книги int NOT NULL,
Код_Автора int NOT NULL,
PRIMARY KEY(Код_Книги, Код_Автора),
CONSTRAINT ref_2_book FOREIGN KEY (Код_Книги) references Книга (Код_Книги)
ON UPDATE NO ACTION
ON DELETE CASCADE,
CONSTRAINT ref_2_author FOREIGN KEY (Код_ Автора) references Автор (Код_ Автора)
ON UPDATE NO ACTION
ON DELETE CASCADE)

Оператор модификации структуры таблицы (оператор ALTER TABLE)

В стандарте ISO предусмотрено применение оператора ALTER TABLE для изменения структуры таблицы после ее создания. Определение оператора ALTER TABLE состоит из шести опций, позволяющих выполнить следующие действия:
• ввести новый столбец в таблицу;
• удалить столбец из таблицы;
• ввести новое ограничение та(5лицы;
• удалить ограничение таблицы;
• задать для столбца значение, применяемое по умолчанию**;**
• удалить опцию, предусматривающую применение для столбца значения, заданного по умолчанию.
Ниже приведен основной формат этого оператора.
ALTER TABLE *TableName*
[ADD [COLUMN] *columnName* dataType [NOT NULL] [UNIQUE]
[DEFAULT defaultOption [CHECK (searchCondition)]]
[DROP [COLUMN] **columnName** [RESTRICT | CASCADE]].,
[ADD [CONSTRAINT [ConstralntName]] *tableConstraintDefinition*]
[DROP CONSTRAINT ConstraintName [RESTRICT | CASCADE]]
[ALTER [COLUMN] SET DEFAULT defaultoption]
[ALTER [COLUMN] DROP DEFAULT]

Почти все параметры данного оператора совпадают с параметрами оператора CREATE TABLE, описанного в предыдущем разделе, В качестве параметра с определением ограничения таблицы TableConstraintDefinition может применяться одна из конструкций

PRIMARY KEY, UNIQUE, FOREIGN KEY или CHECK. Конструкция ADD COLUMN аналогична конструкции определения столбца в операторе CREATE TABLE. В конструкции DROP COLUMN задается имя столбца, удаляемого из определения таблицы, и имеется необязательная опция, позволяющая указать, является ли действие операции DROP каскадным или нет, как показано ниже.

- RESTRICT, Операция DROP отвергается, если на данный столбец имеется ссылка в другом объекте базы данных (например, в определении представления). Это значение опции предусмотрено по умолчанию.
- CASCADE. Выполнение операции DROP продолжается в любом случав и ссылки на столбец автоматически удаляются из любых объектов базы данных, где они имеются. Эта операция выполняется каскадно, поэтому если столбец удаляется из объекта, содержащего ссылку, то в базе данных выполняется проверка того, имеются ли ссылки на этот столбец я каком-либо ином объекте, такие ссылки уничтожаются и в этом объекте, и т.д.

Приведем пример: Добавим Автору колонки год_рождения и год смерти ограничение ,что год смерти не меньше (или один из них пуст)
Alter table Автор add column Год_рождения int ;
Alter table Автор add column Год_смерти int ;

Alter table Автор add constraint author_year_constraint
CHECK((Год_рождения is Null)or(Год_смерти is Null)or (Год_смерти> Год_рождения))
Удалим это ограничение
Alter table Автор drop constraint author_year_constraint

Удаление таблиц (оператор DROP TABLE)

С течением времени структура базы данных меняется: создаются новые таб- лицы, а прежние становятся ненужными* Ненужные таблицы удаляются из ба- зы данных с помощью оператора DROP TABLE, имеющего следующий формат:
DROP TABLE.TableName [RESTRICT I CASCADE]
Например, для удаления таблицы Автор можно использовать следующий оператор:
DROP TABLE Автор CASCADE;
Однако следует отметить, что эта команда удалит не только указанную таблицу, но и все входящие в нее строки данных. Если требуется удалить из таблицы лишь строки данных, сохранив в базе описание самой таблицы, то следует использовать оператор. Оператор DROP TABLE дополнительно позволяет указывать, следует ли операцию удаления выполнять каскадно.

- RESTRICT. Операция DROP отвергается, если в базе данных имеются другие объекты, существование которых зависит от того, существует ли в базе данных удаляемая таблица.
- CASCADE. Операция DROP продолжается, и из базы данных автоматически удаляются все зависимые объекты (и объекты, зависящие от этих объектов). (В нашем случае таблица Автор_Книги)

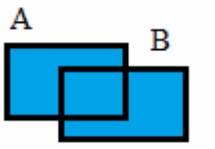
Общий эффект от выполнения оператора DROP TABLE с ключевым словом CASCADE может распространяться на значительную часть базы данных, поэтому подобные операторы следует использовать с максимальной осторожностью. Чаще всего оператор DROP TABLE используется для исправления ошибок, допущенных при создании таблицы. Если таблица была создана с неправильной структурой, можно воспользоваться оператором DROP TABLE для ее удаления, после чего создать таблицу заново.

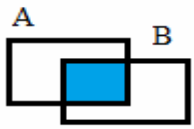
1.5. Реляционная алгебра

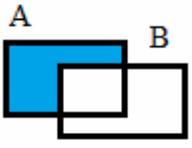
Обзор операторов реляционной алгебры

Реляционная алгебра — это коллекция операций, которые принимают отношения в качестве операндов и возвращают отношение в качестве результата.

Эта алгебра включает восемь базовых операций, которые подразделяются на описанные ниже две группы с четырьмя операциями каждая.

1. Традиционные операции с множествами (все они были немного модифицированы с учетом того факта, что их операндами являются именно отношения, а не произвольные множества):
   a. *объединение: A UNION B, A∪B* 

;
   b. *пересечение: A INTERSECT B, A∩B;* 

;
   c. *разность:* A MINUS B ,A-B 

;
   d. *декартово произведение A* TIMES B

A	B	A TIMES B
a	x	a x
b	y	a y
c		b x
		b y
		c x
		c y

2. Специальные реляционные операции, такие как
   a. *сокращение* (*выборка)*A WHERE p*;*
   b. *проекция;*
   c. *соединение* ;A JOIN B;A⋈B
   d. *деление.*

Необходимо понимать, что данные операции являются базисными и кроме них (и на их основе) может быть определено любое количество операций, которые удовлетворяют простому требованию, чтобы "на входе и выходе были отношения", и такие дополнительные операции были фактически определены многими разными авторами.

Необходимо отметить, что все эти операции предназначены только для чтения (т.е. они "читают", но не обновляют свои операнды). Таким образом, они применяются именно к значениям (разумеется, к значениям отношений) и поэтому, естественно, к тем значениям отношений, которые оказались во время их применения текущими значениями переменных отношения.

Обратите внимание, что предыдущее замечание не означает, что реляционные выражения не могут ссылаться на переменные- отношения. Например, если *R1* и *R2* имена переменных- отношений, то *R1 UNION R2*, конечно, является допустимым реляционным выражением (при условии, что эти переменные- отношения имеют один и тот же тип). Однако в этом выражении *R1* и *R2* обозначают не переменные- отношения как таковые, а отношения, являющиеся текущими значениями этих переменных- отношений в данный момент времени. Иначе говоря, мы, безусловно, можем указывать имя переменной-отношения для обозначения *отношения- операнда*, а такая ссылка на переменную-отношение сама составляет допустимое реляционное выражение, но в принципе с равным успехом мы могли бы на месте этого операнда написать подходящее отношение- литерал. Следующая аналогия поможет прояснить последнюю мысль. Предположим, что N – переменная типа INTEGER, имеющая в момент времени *t* значение 3. Тогда N + 2 – допустимое выражение, которое в момент времени *t* означает в точности 3 + 2, не более и не менее.

Поскольку операторы алгебры способны только читать, значит, INSERT, DELETE и UPDATE (а также реляционное присваивание), несомненно, являющиеся реляционными операторами, не являются частью алгебры как таковой, хотя, к сожалению, в литературе часто можно встретить противоположное утверждение.

Булевы операции. Объединение, пересечение, разность

Для примеров используем следующие отношения (S# -первичный ключ отношения a и b)

A

S#	ФИО	Дисциплина	Оценка
1	Иванов И.И.	Проектир. БД.	5
2	Петров П.П.	ООП	4

B

S#	ФИО	Дисциплина	Оценка
1	Иванов И.И.	Проектир. БД.	5
3	Сидоров С.С.	Информатика	5

C

P#	Преподаватель
Экзамен	Карпов А.А.
Контрольная	Тунцов К.К.

<table>
<tr><th>Операция</th><th>Определение</th><th>Пример</th></tr>
<tr><td>Объединение

a∪b</td><td>Если даны отношения a и b одного и того же типа, то объединение этих отношений a UNION b является отношением того же типа с телом, которое состоит из всех кортежей t, присутствующих в a или b или в обоих отношениях.

a∪b={t: t∈a ⋁ t∈b }</td><td>A UNION B
<table><tr><th>S#</th><th>ФИО</th><th>Дисциплина</th><th>Оценка</th></tr><tr><td>1</td><td>Иванов И.И.</td><td>Проектир. БД.</td><td>5</td></tr><tr><td>2</td><td>Петров П.П.</td><td>ООП</td><td>4</td></tr><tr><td>3</td><td>Сидоров С.С.</td><td>Информатика</td><td>5</td></tr></table></td></tr>
<tr><td>Пересечение

a∩b</td><td>Если даны отношения a и b одного и того же типа, то пересечением этих отношений a INTERSECT b является отношение того же типа с телом, состоящим из всех кортежей t, таких, что t присутствует одновременно в a и b.

a∩b={t: t∈a, t∈b }</td><td>A INTERSECT B
<table><tr><th>S#</th><th>ФИО</th><th>Дисциплина</th><th>Оценка</th></tr><tr><td>1</td><td>Иванов И.И.</td><td>Проектир. БД.</td><td>5</td></tr></table></td></tr>
<tr><td>Разность

a-b</td><td>Если даны отношения a и b одного и того же типа, то разностью этих отношений a MINUS b (в указанном порядке), является отношение того же типа с телом, состоящим из всех кортежей t, таких, что t присутствует в a, но не в b.</td><td>A MINUS B
<table><tr><th>S#</th><th>ФИО</th><th>Дисциплина</th><th>Оценка</th></tr><tr><td>2</td><td>Петров П.П.</td><td>ООП</td><td>4</td></tr></table>
B MINUS A
<table><tr><th>S#</th><th>ФИО</th><th>Дисциплина</th><th>Оценка</th></tr><tr><td>3</td><td>Сидоров С.С.</td><td>Информатика</td><td>5</td></tr></table></td></tr>
</table>

<table>
<tr><td></td><td>a-b={t: t∈a, : t∉b }</td><td></td></tr>
<tr><td>Декартово произведение</td><td>декартово произведение a TIMES b отношений a и b, не имеющих общих атрибутов, как отношение, заголовок которого представляет собой (теоретико-множественное) объединение заголовков отношений a и b, а тело состоит из всех кортежей t, таких, что t является (теоретико-множественным) объединением кортежа, принадлежащего к отношению a, и кортежа, принадлежащего к отношению b.

a×b={t:t=(t₁,t₂) t₁∈a,

t₂∈b }</td><td>A TIMES C
<table><tr><th>S#</th><th>ФИО</th><th>Дисциплина</th><th>Оценка</th><th>P#</th><th>Преподавате</th></tr><tr><td>1</td><td>Иванов И.И.</td><td>Проектир. БД</td><td>5</td><td>Экзамен</td><td>Карпов А.А.</td></tr><tr><td>1</td><td>Иванов И.И.</td><td>Проектир. БД</td><td>5</td><td>Контрольная</td><td>Тунцов К.К.</td></tr><tr><td>2</td><td>Петров П.П.</td><td>ООП</td><td>4</td><td>Экзамен</td><td>Карпов А.А.</td></tr><tr><td>2</td><td>Петров П.П.</td><td>ООП</td><td>4</td><td>Контрольная</td><td>Тунцов К.К.</td></tr></table></td></tr>
</table>

:

Двуместный оператор *g* называется *коммутативным*, если *g(a,b) =g(b,a)* для любых *a* и *b*. Например, в обычной арифметике сложение и умножение коммутативны, а вычитание и

деление – нет. В реляционной алгебре **коммутативными** являются операции **пересечения, объединения и соединения**, но не разности.
Двуместный оператор *g* называется *ассоциативным*, если *g(a,g(b,c)) =g(g(a,b),c)* для любых *a*, *b*, *c*. В арифметике сложение и умножение ассоциативны, а умножение и деление – нет. В реляционной алгебре **ассоциативными** являются операции **пересечения, объединения** и **соединения**, но не разности.

Оператор переименования атрибутов

RENAME, назначение которого состоит в *переименовании* атрибутов указанного отношения. Точнее, оператор RENAME принимает заданное отношение и возвращает другое, идентичное заданному, за исключением того, что один из его атрибутов имеет другое имя. (Заданное отношение должно быть указано с использованием некоторого реляционного выражения, которое может включать иные реляционные операции.)
Например, можно записать следующее выражение.
S RENAME CITY AS SCITY
Это выражение приводит к получению отношения с тем же заголовком и телом, что и отношение, которое является текущим значением переменной отношения S, за исключением того, что атрибут с указанием города в нем называется SCITY, а не CITY,

Оператор выборки

Определение:

отношение **a** имеет атрибуты X, Y, . . ., Z (и, возможно, другие атрибуты), а p является функцией с истинностными значениями, формальные параметры которой представляют собой строгое под- множество атрибутов X, Y, . . ., Z. В таком случае сокращение(выборка) **a** в соответствии с p, которое определяется с помощью приведенной ниже функции, является отношением с тем же заголовком, что и в **a**, и с телом, состоящим из всех кортежей отношения **a**, таких, что функция p принимает значение TRUE для

рассматриваемого кортежа. $\sigma_p(a)= \{t: t \in a, t(X)=x\}$

Или

$\sigma_{X=x}(a)= \{t: t \in a, t(X)=x\}$

Свойства
В реляционной алгебре **выборка дистрибутивна относительно пересечения, объединения и разности**.
Выборка дистрибутивна также **относительно соединения при условии**, что **условие выборки не сложнее, чем конъюнкция (AND) двух различных условий, по одному для каждого из двух операндов**.

пример

Оператор проекции

Определение:

Предположим, что отношение **а** имеет атрибуты X, Y, . . ., Z (и, возможно, другие атрибуты). В таком случае проекция отношения **а** по атрибутам X, y, . . . , z, которая определяется с помощью следующего выражения **а** { X, Y, . . . , Z } является отношением, соответствующим описанным ниже требованиям:

- Его заголовок формируется из заголовка отношения а путем удаления всех атрибутов, не указанных в множестве { X, Y, . . . , z }.
- Тело состоит из всех кортежей { X x , У y, . . . , z z }, таких что в отноше- нии а присутствует кортеж со значением x атрибута X, y атрибута Y... и z атрибута Z.

$$\pi_x(a)=\{t(x), : t\in a \}$$

свойства

В общем случае одноместный оператор *f* называется *дистрибутивным* относительно двуместного оператора *g*, если *f(g(a,b)) = g(f(a),f(b))* для любых *a* и *b*.

Упомянем еще некоторые частные случаи дистрибутивного закона, на этот раз касающиеся проекции. Во-первых**, проекция дистрибутивна относительно объединения**, но не относительно пересечения и разности. Во-вторых**, она дистрибутивна относительно соединения при условии, что все атрибуты, по которым производится соединение, включены в проекцию.**

Ни один из атрибутов не может быть указан в разделенном запятыми списке имен атрибутов больше одного раза.

пример

Оператор соединения

Соединение бывает:

- Естественное
- θ- соединение

Естественное

Предположим, что отношения а и b, соответст венно, имеют следующие атрибуты. X 1 , X 2 , . . . , X m , Y 1 , Y 2 , . . . , Yn
Yl , Y2, . . . , Yn, Z l , Z 2 , . . . , Zp
ЭТО означает, что два рассматриваемых отношения имеют общее множество атрибутов Y, состоящее из атрибутов Yl, Y2 , . . . , Yn (и только из этих атрибутов), другие атрибуты

отношения a образуют множество x, состоящее из атрибутов X1*, X2,* Xm, а другие атрибуты отношения b образуют множество z, состоящее из атрибутов Z1, Z2, . . . , Zp.
В таком случае (**естественное**) **соединение** a и b выражается следующим образом.
a JOIN b
Оно представляет собой отношение с заголовком { X, Y, Z } и телом, состоящим из всех таких кортежей { X x, Y y, z z }, что любой из этих кортежей присутствует и в отношении a, со значением x атрибута x и значением y атрибута Y, и в отношении b, со значением y атрибута Y и значением z атрибута Z.

Если п = 0 (а это означает, что отношения a и b не имеют общих атрибутов), то операция a JOIN b вырождается2 в операцию a TIMES b.
Если m = p = 0 (а это означает, что отношения a и b относятся к одинаковому типу), то операция a JOIN b вырождается в операцию a INTERSECT b.

θ- соединение

неуточненный термин *соединение* почти всегда рассматривается как обозначающий именно естественное соединение.

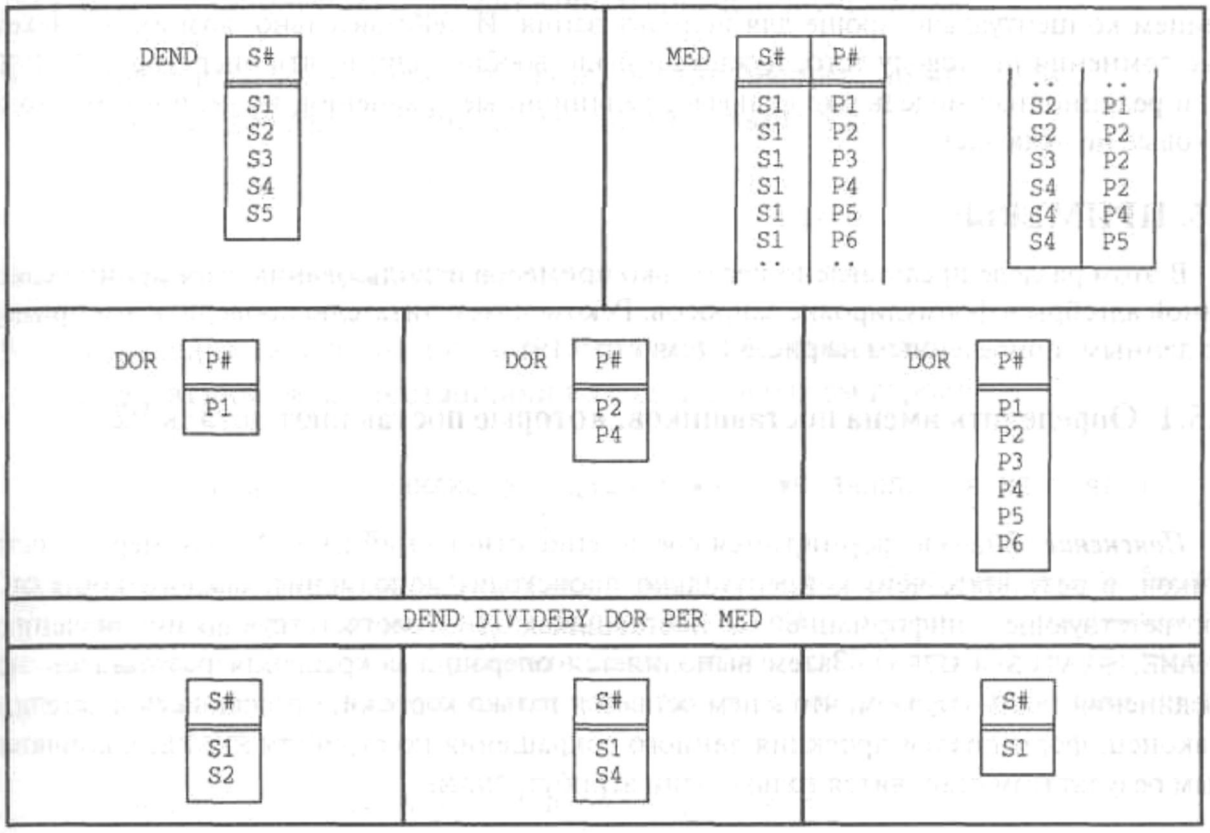
Эта операция предназначена для тех случаев (сравнительно редких, но тем не менее достаточно важных), когда возникает необходимость соединить два отношения на основе некоторого оператора сравнения, отличного от сравнения на равенство. Предположим, что отношения a и b удовле творяют требованиям для декартова произведения (т.е. не имеют общих имен атрибутов);
пусть a имеет атрибут X и b — атрибут У, а x, y и θ удовлетворяют требованиям для θ-соединения. В таком случае операция **θ-соединения** отношения a по атрибуту X с отно шением b по атрибуту y определена как результат вычисления следующего выражения. (a TIMES b) WHERE X θ У
Иными словами, результатом становится отношение с тем же заголовком, как и у де картова произведения a и b, и с телом, состоящим из множества всех кортежей t, таких что t присутствует в этом декартовом произведении, и выражение X θ У принимает значение TRUE для данного кортежа t.

Оператор деления

Предположим, что отношения a и b, соответственно, имеют следующие атрибуты.
XI, X2, . . . , Xm И
Yl, Y2, ..., Yn
Здесь ни один из атрибутов xi (i = 1, 2, . . ., m) не имеет одинакового имени с любым из атрибутов Yj (j = 1, 2, . . ., п). Пусть отношение c имеет следующие атрибуты:
X I , X 2 , . . . , Xm, Y l , Y 2 , . . . , Yn
Это означает, что c имеет заголовок, представляющий собой (теоретико-множественное) объединение заголовков a и b. Будем рассматривать множества { XI, X2, ..., Xm }
и { Yl, Y2, . . ., Yn }, соответственно, как составные атрибуты X и Y. В таком случае операция деления a на b по c (где a — *делимое, b — делитель,* а c — *посредник)* может быть представлена с помощью следующего выражения.
a DIVIDEBY b PER c
Оно представляет собой отношение с заголовком { X } и телом, состоящим из всех кортежей { X x }, присутствующих в a, причем таких, что кортеж { x x, Y y } присутствует в c для всех кортежей { Y y }, присутствующих в b.
Иными словами, неформально выражаясь, данный результат состоит из тех значений X, присутствующих в a, для которых соответствующие значения Y в c включают все

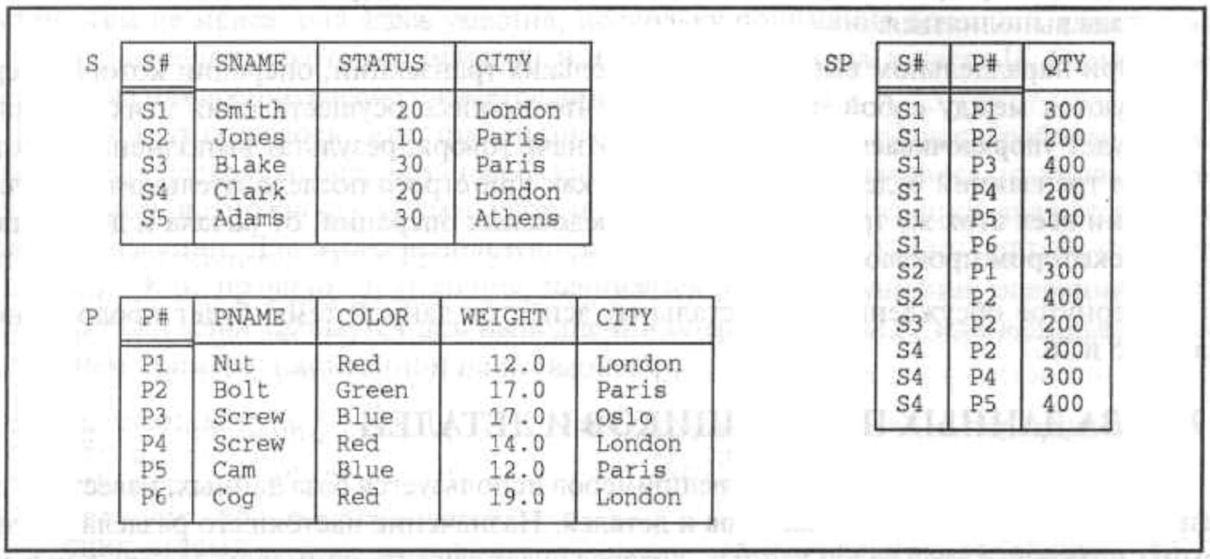
значения Y из b. Обратите внимание на то, что и в этом определении применяется понятие равенства кортежей!

Примеры деления



На рисунке выше приведены некоторые примеры деления. В каждом случае делимое (DEND) представляет собой проекцию текущего значения переменной отношения S по атрибуту S#; посредник (MED) в каждом случае является проекцией текущего значения перемен-
ной отношения SP по атрибутам S# и P#; а три делителя (DOR) являются такими, как указано на этом рисунке. В частности, заслуживает особого внимания последний пример, в котором делителем служит отношение, содержащее номера всех деталей, известных в настоящее время; результат (что вполне очевидно) показывает номера тех поставщиков, которые поставляют все эти детали. На основании данного примера можно сделать вывод, что оператор DIVIDEBY предназначен для запросов подобного общего характера; в действительности, каждый раз, когда версия запроса на естественном языке содержит в условной части слово "все" (например, "Определить поставщиков, которые поставляют все детали"), весьма велика вероятность того, что потребуется деление.
Здесь в примерах и ниже использованы отношения поставщики и детали (Предполагается, что эта база данных имеет следующее назначение.
■ Переменная отношения S представляет *поставщиков* (точнее, поставщиков, работающих по контракту). Каждый поставщик имеет уникальный номер (s#); имя (SNAME), не обязательно уникальное (хотя оно может быть уникальным, как в случае, представленном на рис. 3.9); значение рейтинга или статуса (STATUS); место расположения (CITY). Предполагается, что для каждого поставщика может быть указан только один город.
■ Переменная отношения P представляет *детали* (точнее, *виды* деталей). У каждого вида детали есть номер детали (P#), который является уникальным; название детали (PNAME); цвет (COLOR); вес (WEIGHT); город, где находится склад с деталями этого вида (CITY). Предполагается, что если вес детали имеет значение, то он указан в фунтах).

Предполагается также, что каждый отдельный вид детали имеет только один цвет и хранится на складе только в одном городе.



S

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

SP

S#	P#	QTY
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

P

P#	PNAME	COLOR	WEIGHT	CITY
P1	Nut	Red	12.0	London
P2	Bolt	Green	17.0	Paris
P3	Screw	Blue	17.0	Oslo
P4	Screw	Red	14.0	London
P5	Cam	Blue	12.0	Paris
P6	Cog	Red	19.0	London

Определить имена поставщиков, которые поставляют все детали
((S { S# } DIVIDEBY P { P# } PER SP { S#, P# })
JOIN S) { SNAME }
Еще одна формулировка этого запроса приведена ниже.
(S WHERE
((SP RENAME S# AS X) WHERE X = S#) { P# } = P {P# }) { SNAME }
И в этом случае результат содержит единственный атрибут, SNAME.

Тема 1.6. Исчисление кортежей

Реляционная алгебра и реляционное исчисление представляют собой два альтернативных подхода. Принципиальное различие между ними состоит в следующем. *Реляционная алгебра* определяет в явном виде набор операций (соединение, объединение, проекция и т.д.), которые можно использовать, чтобы сообщить системе, как в базе данных из определенных отношений *сформировать* некоторое требуемое отношение, а *реляционное исчисление* просто задает систему обозначений для *определения* требуемого отношения в терминах существующих отношений. Например, рассмотрим запрос: "Выбрать номера поставщиков и названия городов, в которых находятся поставщики детали с номером P2".

Алгебраическая версия этого запроса может быть составлена примерно так.

- Выполнить соединение отношений поставщиков и поставок SP по атрибуту S#.
- С помощью операции сокращения выделить из результатов этого соединения кортежи, которые относятся к детали с номером P2.
- Сформировать проекцию результатов этой операции сокращения по атрибутам s#
- И CITY.

Этот же запрос в терминах реляционного исчисления формулируется приблизительно

следующим образом.

- Получить атрибуты s# и CITY для таких поставщиков, для которых в отношении SP существует запись о поставке с тем же значением атрибута s# и со значением атрибута P#, равным P2.

В этой формулировке пользователь лишь указывает определенные характеристики требуемого результата, предоставляя системе решать, что именно и в какой последовательности соединять, проецировать и т.д., чтобы получить необходимый результат. Итак, можно сказать, что, по крайней мере, внешне формулировка запроса в терминах реляционного исчисления носит характер описания, а в терминах реляционной алгебры — *предписания.* В реляционном исчислении просто указывается, в чем *заключается* проблема, тогда как в реляционной алгебре задается *процедура решения* этой проблемы. Или, говоря *очень* неформально, алгебра имеет процедурный характер (пусть на высоком уровне, но все же процедурный, поскольку задает необходимые для выполнения процедуры), а исчисление — непроцедурный.

Подчеркнем, однако, что упомянутые различия существуют только внешне. На самом деле *реляционная алгебра и реляционное исчисление логически эквивалентны.* Каждому выражению в алгебре соответствует эквивалентное выражение в исчислении, и точно также каждому выражению в исчислении соответствует эквивалентное выражение в алгебре.

Реляционное исчисление основано на разделе математической логики, который называется исчислением предикатов. Идея использования исчисления предикатов в качестве основы языка баз данных впервые была высказана в статье Кунса. Понятие *реляционного исчисления,* т.е. специального метода применения исчисления предикатов в реляционных базах данных, впервые было сформулировано Коддом. На основе его наработок и созданного им языка ALPHA был создан язык QUEL, некоторое время серьезно конкурировавший с языком SQL.
Основным средством реляционного исчисления является понятие переменной области значений. Согласно краткому определению, переменная области значений — это переменная, *принимающая значения* из некоторого заданного отношения, т.е. переменная, допустимыми значениями которой являются кортежи заданного отношения. Другими словами, если переменная области значений v изменяется в пределах отношения r, то в любой конкретный момент выражение "v" представляет некоторый кортеж t отношения r. Например, запрос "Получить идентификаторы студентов, обучающихся в группе Z4331" может быть представлен на языке QUEL следующим образом.

```
RANGE OF SX IS S ;
RETRIEVE ( SX.S# ) WHERE SX.GROUP = " Z4331" ;
```

Единственной переменной области значений здесь является переменная SX, которая принимает значения из отношения, представляющего собой текущее значение переменной отношения S (оператор RANGE — *оператор определения* этой переменной области значений). Оператор RETRIEVE означает следующее: "Для каждого возможного значения переменной SX выбирать компонент S# этого значения тогда и только тогда, когда компонентом GROUP этого значения является Z4331".
Каждая ссылка на переменную области значений (в некотором контексте, в частности в некоторой правильно построенной формуле) является либо **свободной,** либо **связанной.**

Приведем несколько примеров правильно построенных формул, содержащих ссылки на переменные области значений.

■ *Простые сравнения*

SX.S# = S# {'S1')

SX.S# = SPX.S#

SPX.P# ≠ PX.P#

Здесь все ссылки на переменные SX, PX и SPX являются свободными.

■ *Простые операции сравнения, объединенные с помощью логических выражений*

PX.WEIGHT < WEIGHT (15.5) AND PX.CITY = 'Oslo'

NOT (SX.CITY = 'London')

SX.S# = SPX.S# AND SPX.P# ≠ PX.P#

PX.COLOR = COLOR ('Red') OR PX.CITY = 'London'

Здесь также все ссылки на переменные sx, PX и SPX являются свободными.

■ *Правильно построенные формулы с кванторами*

EXISTS SPX (SPX.S# = SX.S# AND SPX.P# =P# ('P2')) FORALL PX (PX.COLOR = COLOR('Red'))

В этих примерах ссылки на переменные SPX и PX являются связанными, а ссылка на переменную SX остается свободной. Подробнее данные примеры описаны ниже, в подразделе "Кванторы".

Кванторы

Существует два квантора2: EXISTS и FORALL. Квантор EXISTS является квантором **существования,** а FORALL — квантором **всеобщности.** По сути, если выражение p — правильно построенная формула, в которой переменная v свободна, то выражения

EXISTS V (p

) И

FORALL V (p)

также являются допустимыми правильно построенными формулами, но переменная V в них обеих связана. Первая формула означает следующее: **"Существует по меньшей мере одно значение** переменной V, при котором формула p становится *истинной".* Вторая формула означает следующее: "Формула p является *истинной* **при всех значениях** переменной V".

Предположим, например, что переменная V принимает значения из множества "Члены сената США в 2003 году", и предположим также, что выражение p — следующая правильно построенная формула: "V— женщина" (разумеется, мы не пытаемся использовать здесь формальный синтаксис). Тогда выражение EXISTS V(p) будет допустимой правильно построенной формулой, имеющей значение TRUE *{истина);* выражение FORALL v(p) также будет допустимой правильно построенной формулой, но ее значение будет равно FALSE *{ложь),* поскольку не все члены сената— женщины.

Теперь рассмотрим квантор существования EXISTS более внимательно. Еще раз обратимся к примеру, приведенному в конце предыдущего раздела.

EXISTS SPX (SPX.S# = SX.S# AND SPX.P# = P# ('P2'))

Из приведенных выше рассуждений следует, что эта правильно построенная формула может быть прочитана следующим образом.

В текущем значении переменной отношения SP существует кортеж (скажем, SPX), такой, что значение атрибута S# в этом кортеже равно значению атрибута SX. S# (каким бы оно ни было), а значение атрибута P# в кортеже SPX равно P2.

Каждая ссылка на переменную SPX в этом примере является связанной. Единственная ссылка на переменную SX свободна.

Формально квантор существования EXISTS определяется как **повторно применяемая операция OR** (ИЛИ). Другими словами, если, во-первых, r — это отношение с кортежами tl, t2, .. . , tm, во-вторых, V—это переменная области значений, принимающая значения из данного отношения, и, в третьих, p(V) — это правильно построенная формула, в которой

переменная V используется как свободная переменная, то правильно построенная формула вида
EXISTS V (p (V))
эквивалентна следующей правильно построенной формуле.
FALSE OR p (tl } OR . . . OR p (tm)
В частности, следует отметить, что если отношение R пустое (т.е. m=0), то данное выражения принимает значение FALSE.
Подобно тому, что квантор EXISTS был определен как результат многократного применения операции OR, квантор существования FORALL определяется как результат **многократного применения операции AND** (И). Другими словами, если обозначения r, V и p (V) имеют тот же смысл, что и в приведенном выше определении квантора EXISTS, ТО правильно построенная формула вида
FORALL V (p (V))
равносильна следующей правильно построенной формуле.
TRUE AND p (tl) AND . . . AND p (tm)
В частности, следует отметить, что если отношение r пустое, то данное выражение принимает значение TRUE.

Операция	Синтаксис реляционной алгебры	Синтаксис реляционного исчисления
Определить имена поставщиков по крайней мере одной детали красного цвета	{ ((P WHERE COLOR = COLOR ('Red')) JOIN SP) { S# } JOIN S) { SNAME } **Или** (((P WHERE COLOR = COLOR ('Red')) { P# } JOIN SP) JOIN S) { SNAME }	SX.SNAME WHERE EXISTS SPX (SX.S# = SPX.S# AND . EXISTS PX (PX.P# = SPX.P# AND PX.COLOR = COLOR ('Red'))) **Или** SX.SNAME WHERE EXISTS SPX (EXISTS PX (SX.S# = SPX.S# AND SPX.P# = PX.P# AND PX.COLOR = COLOR ('Red')))
Получить полную информацию о поставщиках детали с номером P2	{ (SP JOIN S) WHERE P# = P# ('P2')) { SNAME } Пояснение. Вначале формируется соединение отношений SP и S по номерам поставщиков, в результате чего концептуально происходит дополнение каждого кортежа SP соответствующей информацией о поставщиках (т.е. соответствующими значениями SNAME, STATUS и CITY). Затем выполняется операция сокращения результатов этого соединения таким образом, что в нем остаются только кортежи, относящиеся к детали P2.Наконец, формируется проекция данного сокращения по атрибуту SNAME. Окончатель-ным результатом становится только	SX WHERE EXISTS SPX (SPX.S# = SX.S# AND SPX.P# = P# ('P2')) Этот пример является сокращенной записью следующего выражения. { SX.S#, SX.SNAME, SX.STATUS, SX.CITY } WHERE EXISTS SPX (SPX.S# = SX.S# AND SPX.P# = P# ('P2'))

	один атрибут, SNAME.*	

Тема 2.1. Оператор выборки в языке SQL

Оператор выборки в языке SQL

Назначение оператора SELECT состоит в выборке и отображении данных одной или более таблиц базы данных. Это исключительно мощный оператор, способный выполнять действия, эквивалентные операторам реляционной алгебры выборки, проекции и соединения, причем в пределах единственной выполняемой команды. Оператор SELECT является чаще всего используемой командой языка SQL. Общий формат оператора SELECT имеет следующий вид:
SELECT [DISTINCT | ALL] .{ *| *[columnExpression [*AS *newName]] [, ...] }*
FROM TableName [alias] [,..]
[WHERE condition]
[GROUP BY *columnList*] [HAVING condition]
[ORDER BY *columnList]*

Слово distinct позволяет убрать дублирующиеся строки из результата запроса.
Здесь параметр *columnExpression* представляет собой имя столбца или выражение из нескольких имен.
columnExpression может включать только следующие типы элементов:
• имена столбцов;
• агрегирующие функции;
• константы;
• выражения, включающие комбинации перечисленных выше элементов.
В противоположность списку столбцов * обозначает возврат всех столбцов всех таблиц запроса.

Параметр *TableName* является именем существующей в базе данных таблицы (или представления), к которой необходимо получить доступ. Необязательный параметр alias — это сокращение, псевдоним, устанавливаемое для имени таблицы *TableName.*
Обработка элементов оператора SELECT выполняется в следующей последовательности.
• FROM. Определяются имена используемой таблицы или нескольких таблиц, перечисленные через запятую. Также в качестве источника данных в этом разделе могут быть таблицы связанные различными видами соединения, представления и другие запросы (смотри запросы с подзапросами)
• WHERE. Выполняется фильтрация строк объекта в соответствии с заданными условиями.
• GROUP BY. Образуются группы строк, имеющих одно и то же значение в указанном столбце (использовать только с агрегатными функциями).
• HAVING. Фильтруются группы строк объекта в соответствии с указанным условием , относящимся к результату агрегатной функции.
• SELECT. Устанавливается, какие столбцы должны присутствовать в выходных данных.
• ORDER BY. Определяется упорядоченность результатов выполнения оператора.
При этом каждому полю в списке сортировки может быть приписано ASC (по возрастанию- это параметр сортировки поля по умолчанию) или DESC (по убыванию)

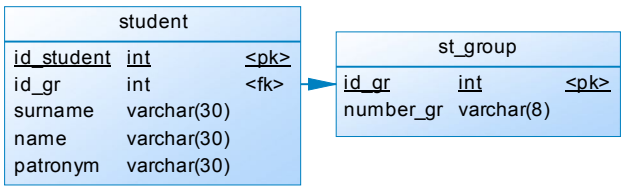
Порядок конструкций в операторе SELECT *не может* быть изменен. Только две конструкции оператора — SELECT и FROM — являются обязательными, все остальные конструкции могут быть опущены.

Условия в конструкции WHERE
(применимы как к оператору SELECT, так и UPDATE или INSERT)

В приведенных выше примерах в результате выполнения операторов SELECT выбирались все строки указанной таблицы. Однако очень часто требуется тем или иным образом ограничить набор строк, помещаемых в результирующую таблицу запроса. Это достигается с помощью указания в запросе конструкции WHERE. Она состоит из ключевого слова WHERE, за которым следует перечень условий поиска, определяющих те строки, которые должны быть выбраны при выполнении запроса. Существует пять основных типов условий поиска (или *предикатов,* если пользоваться терминологией ISO).
• Сравнение. Сравниваются результаты вычисления одного выражения с результатами вычисления другого выражения. (<,>,=,<>)
• Диапазон. Проверяется, попадает ли результат вычисления выражения в заданный диапазон значений. (имя_поля BETWEEN знач_1 AND знач_2)
• Принадлежность к множеству. Проверяется, принадлежит ли результат вычисления выражения к заданному множеству значений. (имя_поля in (знач_1,знач_2…, знач_n))
• Значение NULL. Проверяется, содержит ли данный столбец NULL (неопределенное значение) (имя_поля IS NULL).
• Соответствие шаблону. Проверяется, отвечает ли некоторое строковое значение заданному шаблону. (имя_поля LIKE шаблон)
В
• %. Символ процента представляет любую последовательность из нуля или более символов (поэтому часто именуется также *подстановочным символом).* (В ACCESS вместо него используется *, но при вызове из программы через ADO или DAO должен быть все равно %*)*
• _. Символ подчеркивания представляет любой отдельный символ. . (В ACCESS вместо него используется ?, но при вызове из программы через ADO или DAO должно быть все равно _*)*
Все остальные символы в шаблоне представляют сами себя.
Пример
• address LIKE 'Ш%'. Этот шаблон означает, что первый символ значения обязательно должен быть символом Ш, а все остальные символы не представляют интереса и не проверяются.
• address LIKE 'Ш_ _ _ _'. Этот шаблон означает, что значение должно иметь длину, равную строго четырем символам, причем первым символом обязательно должен быть символ ' Ш ' . (пробелы между подчеркиваниями только для визуализации –их нет в шаблоне)
• address LIKE ' %ич'. Этот шаблон определяет любую последовательностьсимволов длиной не менее двух символов, причем последними символами обязательно должен быть символы ич. (поиск мужчин по отчеству)
• address LIKE '%слово%'. Этот шаблон означает, что нас интересует
любая последовательность символов, включающая подстроку «слово»;
• address NOT LIKE ' Ш%'. Этот шаблон указывает, что требуются любые строки, которые не начинаются с символа Ш.
Если требуемая строка должна включать также служебный символ, обычно применяемый в качестве символа подстановки, то следует определить с помощью конструкции ESCAPE "маскирующий" символ, который указывает, что следующий за ним символ больше не имнет специального значения, и поместить его перед символом подстановки. Например, для проверки значений на соответствие литеральной строке 45%' можно воспользоваться таким предикатом:LIKE '15#%' ESCAPE '#'

по умолчанию маскирующий символ- ‘\‘ (LIKE '15\%'),
а для строк его содержащих в шаблоне должен иметь вид \\
Пример

Представим себе таблицы группа и студент.



Выберем всех студентов группы Z4432K

select id_student, student.id_gr, surname, name, patronym from student, st_group where student.id_gr =st_group.id_gr and number_gr='Z4432K'

Виды соединений в языке SQL

Виды соединений в SQL:

- Декартово произведение CROSS JOIN
- Внутреннее соединение INNER JOIN (часто просто JOIN)
- Внешние соединения:
- Левое соединение LEFT JOIN
- Правое соединение RIGHT JOIN
- Полное (внешнее) соединение OUTER JOIN

Соединение является подмножеством более общей комбинации данных двух таблиц, называемой *декартовым произведением* . Декартово произведение двух таблиц представляет собой другую таблицу, состоящую из всех возможных пар строк, входящих в состав обеих таблиц. Набор столбцов результирующей таблицы представляет собой все столбцы первой таблицы, за которыми следуют все столбцы второй таблицы. Если ввести запрос к двум таблицам без задания конструкции WHERE, результат выполнения запроса в среде SQL будет представлять собой декартово произведение этих таблиц. Кроме того, стандарт ISO предусматривает специальный формат оператора SELECT, позволяющий вычислить декартово произведение двух таблиц:
SELECT [DISTINCT. | ALL] {* | *columnList}*
FROM *tableName1* CROSS JOIN *tableName2*

При выполнении операции соединения данные из двух таблиц комбинируются с образованием пар связанных строк, в которых значения сопоставляемых столбцов являются одинаковыми. Если одно из значений в сопоставляемом столбце одной таблицы не совпадает ни с одним из значений в сопоставляемом столбце другой таблицы, то соответствующая строка удаляется из результирующей таблицы. Именно это правило применялось во всех рассмотренных выше примерах соединения таблиц. Стандартом ISO предусмотрен и другой набор операторов соединений, называемых *внешними соединениями*. Во внешнем соединении в результирующую таблицу помещаются также строки, не удовлетворяющие условию соединения.
Внутреннее соединение используется для связи таблиц по совпадающим значениям столбцов. Внешние соединения нужны в случае, когда значениям в одной таблице не всегда будут иметь соответствующие значения в другой (т.е. внешний ключ – пуст (NULL))

Свойства соединений:

- Внутреннее соединение симметрично, т.е

SELECT *tableName1.*, tableName2.*,* FROM *tableName1* INNER JOIN *tableName2*
Эквивалентно
SELECT *tableName1.*, tableName2.*,* FROM *tableName2* INNER JOIN *tableName1*

- Полное внешнее соединение симметрично, т.е.

SELECT *tableName1.*, tableName2.*,* FROM *tableName1* OUTER JOIN *tableName2*
Эквивалентно
SELECT *tableName1.*, tableName2.*,* FROM *tableName2* OUTER JOIN *tableName1*

- Левое и правое соединения симметричны друг по отношению к другу, т.е.

SELECT *tableName1.*, tableName2.*,* FROM *tableName1* LEFT JOIN *tableName2*
Эквивалентно
SELECT *tableName1.*, tableName2.*,* FROM *tableName2* RIGHT JOIN *tableName1*

Для примеров различных соединений используем таблицы группа и студент

student

id_student	id_gr	surname	name	patronym
1	1	Петров	Петр	Петрович
2	2	Иванов	Иван	Иванович
3	*NULL*	Пупкин	Василий	Федорович

st_group

id_gr	number_gr
1	Z4431K
2	Z5432K
3	B5433

Select number_gr, id_student, surname, name, patronym from st_group inner join student on student.id_gr=st_group.id_gr

Результат такого запросы будет таким

st_group INNER JOIN student

number_gr	id_student	surname	name	patronym
Z4431K	1	Петров	Петр	Петрович
Z5432K	2	Иванов	Иван	Иванович

Select number_gr, id_student, surname, name, patronym from st_group left join student on student.id_gr=st_group.id_gr

Результат такого запросы будет таким

st_group LEFT JOIN student

number_gr	id_student	surname	name	patronym
Z4431K	1	Петров	Петр	Петрович
Z5432K	2	Иванов	Иван	Иванович
B5433	*NULL*	*NULL*	*NULL*	*NULL*

Select number_gr, id_student, surname, name, patronym from st_group right join student on student.id_gr=st_group.id_gr

Ему эквивалентен запрос Select number_gr, id_student, surname, name, patronym from student left join st_group on student.id_gr=st_group.id_gr
Результат такого запросы будет таким

st_group RIGHT JOIN student

number_gr	id_student	surname	name	patronym
Z4431K	1	Петров	Петр	Петрович
Z5432K	2	Иванов	Иван	Иванович
NULL	3	Пупкин	Василий	Федорович

Select number_gr, id_student, surname, name, patronym from st_group outer join student on student.id_gr=st_group.id_gr

Результат такого запросы будет таким

st_group OUTER JOIN student

number_gr	id_student	surname	name	patronym
Z4431K	1	Петров	Петр	Петрович
Z5432K	2	Иванов	Иван	Иванович
NULL	3	Пупкин	Василий	Федорович
B5433	*NULL*	*NULL*	*NULL*	*NULL*

Агрегатные функции в операторе выборки языка SQL

Стандарт ISO содержит определение следующих пяти *агрегирующих функций:*
• COUNT — возвращает количество значений в указанном столбце;
• SUM — возвращает сумму значений в указанном столбце;
• AVG — возвращает усредненное значение в указанном столбце;
• MIN — возвращает минимальное значение в указанном столбце;
• MAX — возвращает максимальное значение в указанном столбце.
.Все эти функции оперируют со значениями в единственном столбце таблицы и возвращают единственное значение. Функции COUNT, MIN и MAX применимы как к числовым, так и к нечисловым полям, тогда как функции SUM и AVG могут использоваться только в случае числовых полей. За исключением COUNT (*) , при вычислении результатов любых функций сначала исключаются все пустые значения, после чего требуемая операция применяется только к оставшимся непустым значениям столбца. Вариант COUNT (*) является особым случаем использования функции COUNT — его назначение состоит в подсчете всех строк в таблице, независимо от того, содержатся там пустые, повторяющиеся или любые другие значения. Если до применения агрегирующей функции необходимо исключить повторяющиеся значения, следует перед именем столбца в определении функции поместить ключевое слово DISTINCT. Стандарт ISO допускает использование ключевого слова ALL с целью явного указания того, что исключение повторяющихся значений не требуется, хотя это ключевое слово подразумевается по умолчанию, если никакие иные определители не заданы. Ключевое слово DISTINCT не имеет смысла для функций MIN и MAX. Однако его использование может оказывать влияние на результаты выполнения функций SUM и AVG, поэтому следует заранее обдумать, должно ли оно присутствовать в каждом конкретном случае. Кроме того, ключевое слово DISTINCT в каждом запросе может быть указано не более

одного раза. Следует отметить, что агрегирующие функции могут использоваться только в списке выборки SELECT и в конструкции HAVING Во всех других случаях применение этих функций недопустимо. Если список выборки SELECT содержит агрегирующую функцию, а в тексте запроса отсутствует конструкция GROUP BY, обеспечивающая объединение данных в группы, то ни один из элементов списка выборки SELECT не может включать каких-либо ссылок на столбцы, за исключением случая, когда этот столбец используется как параметр агрегирующей функции.__

5.3.4. Группирование результатов (конструкция GROUP BY)

Приведенные выше примеры сводных данных подобны итоговым строкам, обычно размещаемым в конце отчетов. В итогах все детальные данные отчета сжимаются в одну обобщающую строку. Однако очень часто в отчетах требуется формировать и промежуточные итоги. Для этой цели в операторе SELECT может указываться конструкция GROUP BY. Запрос, в котором присутствует конструкция GROUP BY, называется *группирующим запросом,* поскольку в нем группируются данные, полученные в результате выполнения операции SELECT, после чего для каждой отдельной группы создается единственная итоговая строка. Столбцы, перечисленные в конструкции GROUP BY, называются *группируемыми столбцами.* Стандарт ISO требует, чтобы конструкции SELECT и GROUP BY были тесно связаны между собой. При использовании в операторе SELECT конструкции GROUP BY каждый элемент списка в списке выборки SELECT должен иметь *единственное значение для всей группы.*

Все имена столбцов, приведенные в списке выборки SELECT, должны присутствовать и в конструкции GROUP BY, за исключением случаев, когда имя столбца используется только в агрегирующей функции. Противоположное утверждение не всегда справедливо — в конструкции GROUP BY могут присутствовать имена столбцов, отсутствующие в списке выборки SELECT. Если совместно с конструкцией GROUP BY используется конструкция WHERE, то она обрабатывается в первую очередь, а группированию подвергаются только те строки, которые удовлетворяют условию поиска. Стандартом ISO определено, что при проведении группирования все отсутствующие значения рассматриваются как равные. Если две строки таблицы в одном и том же группируемом столбце содержат значения NULL и идентичные значения во всех остальных непустых группируемых столбцах, они помещаются в одну и ту же группу.

конструкция HAVING

Конструкция HAVING предназначена для использования совместно с конструкцией GROUP BY для задания ограничений, указываемых с целью отбора тех *групп,* которые будут помещены в результирующую таблицу запроса. Хотя кон- струкции HAVING и WHERE имеют сходный синтаксис, их назначение различно.

В основном HAVING действует так же, как выражение WHEREв операторе SELECT. Тем не менее, выражение WHERE применяется к столбцам и выражениям для отдельной строки, в то время как оператор HAVING применяетсяк результату действия команды GROUP BY (агрегатной функции). Стандарт ISO требует, чтобы имена столбцов, применяемые в конструкции HAVING, обязательно присутствовали в списке элементов GROUP BY или применялись в агрегирующих функциях. На практике условия поиска в конструкции HAVING всегда включают, по меньшей мере, одну агрегирующую функцию; в противном случае эти условия поиска должны быть помещены в конструкцию WHERE и применены для отбора отдельных строк. (Помните, что агрегирующие функции не могут использоваться в конструкции WHERE.)

Конструкция HAVING не является необходимой частью языка SQL —- любой запрос, написанный с использованием конструкции HAVING, может быть пред- ставлен в ином виде, без ее применения.

Приведем пример использования агрегетных функций и группировки.

Допустим существует таблица Student_Uni

Student_uni		
id_student	int	<pk>
surname	varchar(30)	
name	varchar(30)	
patronym	varchar(30)	
Form_study	varchar(1)	
Faculty	varchar(1)	
department	varchar(2)	

,

Где From_study- форма обучения («О», «В» «З»), Faculty- факультет, department- кафедра.

Если мы хотим посчитать всех студентов вуза, необходимо написать запрос

SELECT count (id_student) as all_st from Student_Uni

, тогда в результате мы получим одно число равное общему количеству студентов.

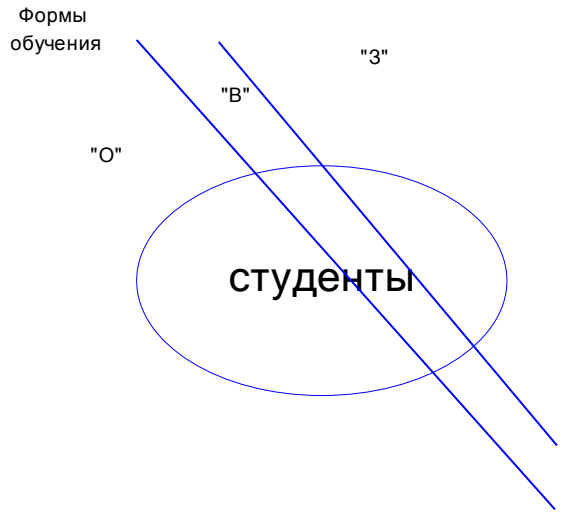
Если мы хотим узнать сколько учится студентов на каждой форме обучения надо написать

SELECT count (id_student) as all_st, Form_study from Student_Uni GROUP BY Form_study

И получим в результате, что-то вроде

all_st	Form_study
400	O
20	B
200	З

На самом деле сгруппировав по форме обучения, мы сделали равенство форм обучения критерием отнесения студента к той или иной группе, в которой потом будет производиться подсчет. Т.е. разбили все множество студентов по форме обучения



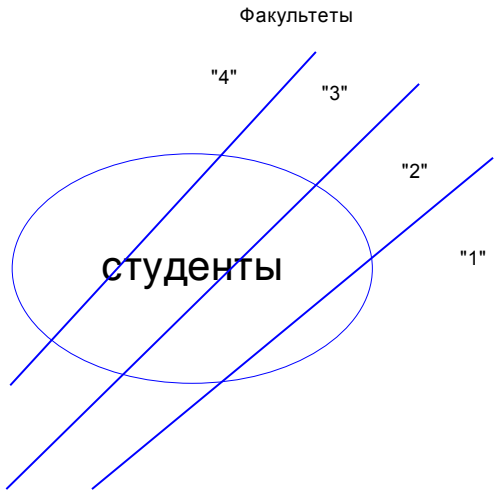
Если мы захотим узнать численность студентов на факультетах, запрос будет выглядеть:

SELECT count (id_student) as all_st, Faculty from Student_Uni GROUP BY Faculty

Допустим в вузе только 4 факультета, тогда мы получим результат типа

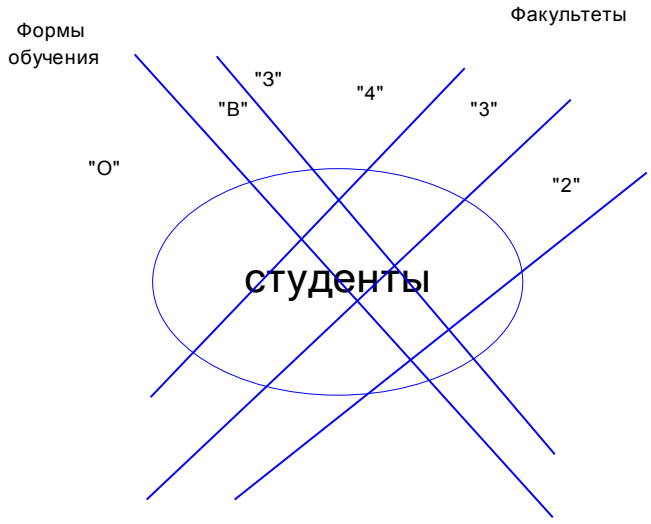
all_st	Faculty
100	1
100	2
200	3

220	4



Ячейка уже объединяется по факультету.
А если мы сгруппируем и по факультету и по форме обучения
SELECT count (id_student) as all_st, Faculty, Form_study from Student_Uni GROUP BY Faculty, Form_study

То получим ячейку у которой одинаковы оба эти значения



И результат (числа не вписаны- это общий вид)

All_st	Faculty	Form_study
	1	O
	1	B
	1	З
	2	O
	2	B
	2	З
	3	O
	3	B
	3	З
	4	O

	4	B
	4	З

Хотя строк может быть меньше, если не на всех факультетах есть все формы обучения!!!
Каждый раз добавляя в группировку поля мы уменьшаем ячейку, поэтому например запрос типа
SELECT count (id_student) as all_st from Student_Uni GROUP BY id_student
Выдаст и кучу строк с единицами (размер ячейки подсчета студентов- один студент)
Поэтому обратите внимание !

Если у Вас группировка и агрегатная функция заданы для одного поля, то это ,вероятно, ошибка

Объединение, пересечение, разность запросов в языке SQL

В языке SQL можно использовать обычные операции над множествами — объединение (union), пересечение (intersection) и разность (difference), — позволяющие комбинировать результаты выполнения двух и более запросов в единую результирующую таблицу.

• *Объединением* двух таблиц A и B называется таблица, содержащая все строки, которые имеются в первой таблице (A), во второй таблице (B) или в обеих этих таблицах одновременно,
• *Пересечением,* двух таблиц называется таблица, содержащая все строки, присутствующие в обеих исходных таблицах одновременно.
• *Разностью* двух таблиц A и B называется таблица, содержащая все строки, которые присутствуют в таблице A, но отсутствуют в таблице B.
Все эти операции над множествами графически представлены на рис. 5.2. На таблицы, которые могут комбинироваться с помощью операций над множествами, накладываются определенные ограничения. Самое важное из них состоит в том, что таблицы должны быть *совместимы, по соединению* — т.е. они должны иметь одну и ту же структуру. Это означает, что таблицы должны иметь одинаковое количество столбцов, причем в соответствующих столбцах должны размещаться данные одного и того же типа и длины. Обязанность убедиться в том, что значения данных соответствующих столбцов принадлежат одному и тому же домену, возлагается на пользователя.
Три операции над множествами, предусмотренные стандартом ISO, носят название UNION, INTERSECT и EXCEPT. В каждом случае формат конструкции с операцией над множествами должен быть следующим:
operator .[ALL] [CORRESPONDING [BY {columnl [, . . .]}]]
При указании конструкции CORRES POND ING BY операция над множествами выполняется для указанных столбцов. Если задано только ключевое слово CORRESPONDING, а конструкция BY отсутствует, операция над множествами выполняется для столбцов, которые являются общими для обеих таблиц. Если указано ключевое слово ALL, результирующая таблица может содержать повторяющиеся строки. Одни диалекты языка SQL не поддерживают операций INTERSECT и EXCEPT, а в других вместо ключевого слова EXCEPT используется ключевое слово MINUS.

Пример выберем имена и фамилии преподавателей или студентов с непустым отчеством. (логическое или)

```
(SELECT surname, name
FROM student
WHERE patronym IS NOT NULL)
UNION
(SELECT surname, name
FROM teacher
WHERE patronym IS NOT NULL);
```
Или
```
(SELECT *
FROM student
WHERE patronym IS NOT NULL)
UNION CORRESPONDING BY surname, name
(SELECT *
FROM teacher
WHERE patronym IS NOT NULL);
```

Выберем фамилии и имена и отчества преподавателей , которые еще учатся (в магистратуре)(логическое И)(считаем, что полных тезок нет)
```
(SELECT surname, name, patronym
FROM student )
INTERSECT
(SELECT surname, name, patronym
FROM teacher );
```
Или
```
(SELECT *
FROM student)
INTERSECT CORRESPONDING BY surname, name, patronym
(SELECT *
FROM teacher);
```

Выберем фамилии и имена и отчества студентов , которые не преподают (в магистратуре) (студенты минус преподаватели)(считаем, что полных тезок нет)
```
(SELECT surname, name, patronym
FROM student )
EXCEPT
(SELECT surname, name, patronym
FROM teacher );
```
Или
```
(SELECT *
FROM student)
EXCEPT CORRESPONDING BY surname, name, patronym
(SELECT *
FROM teacher);
```

Запросы с подзапросами в языке SQL

Здесь мы обсудим использование законченных операторов SELECT, внедренных в тело другого оператора SELECT. *Внешний* (второй) оператор SELECT использует результат выполнения *внутреннего* (первого) оператора для определения содержания окончательного результата всей операции. Внутренние запросы могут находиться в конструкциях WHERE и HAVING внешнего оператора SELECT — в этом случае они получают название *подзапросов,* или *вложенных запросов.* Кроме того, внутренние операторы SELECT могут использоваться в операторах INSERT, UPDATE и DELETE . Также возможно нахождение подзапроса в конструкции FROM , где он должен получить псевдоним после скобок и с ним можно работать как с любой таблицей.

Существуют три типа подзапросов.

• *Скалярный подзапрос* возвращает значение, выбираемое из пересечения одного столбца с одной строкой, т.е. единственное значение. В принципе скалярный подзапрос может использоваться везде, где требуется указать единственное значение.

• *Строковый подзапрос* возвращает значения нескольких столбцов таблицы, но в виде единственной строки. Строковый подзапрос может использоваться везде, где применяется конструктор строковых значений, — обычно это предикаты.

• *Табличный подзапрос* возвращает значения одного или нескольких столбцов таблицы, размещенные в более чем одной строке. Табличный подзапрос может использоваться везде, где допускается указывать таблицу, например как операнд предиката IN.

Пример скалярного подзапроса

Вывести группу, в которой учится столько же человек, как и в группе Z4431

```
Select st_group.* from st_group left join student on student.id_gr=st_group.id_gr
Group by student.id_gr
Having count (id_student)=
(select count(*) from st_group gr
  left join student st on st.id_gr=gr.id_gr where  number_gr=' Z4431')
```

Left join, т.к в группе может не быть студентов

Табличный подзапрос

Вывести всех студентов из групп ,где учится Иванов

```
Select student.* from student where id_gr in
(select st_group.id_gr from from st_group inner join student on student.id_gr=st_group.id_gr
where student.surname='Иванов')
```

К подзапросам применяются следующие правила и ограничения.

1. В подзапросах не должна использоваться конструкция ORDER BY, хотя она может присутствовать во внешнем операторе SELECT.
2. Список выборки SELECT подзапроса должен состоять из имен отдельных столбцов или составленных из них выражений, за исключением случая, когда в подзапросе используется ключевое слово EXISTS.
3. По умолчанию имена столбцов в подзапросе относятся к таблице, имя которой указано в конструкции FROM подзапроса. Однако разрешается ссылаться и на столбцы таблицы,

указанной в конструкции FROM внешнего запроса, для чего используются уточненные имена столбцов (как описано ниже).
4. Если подзапрос является одним из двух операндов, участвующих в операции сравнения, то подзапрос должен указываться в правой части этой операции. Например, приведенный ниже вариант записи запроса является некорректным, поскольку подзапрос размещен в левой части операции сравнения со значением столбца salary.

```
SELECT staffNo, fName, IName, position, salary
FROM Staff
WHERE (SELECT AVG(salary) FROM Staff) < salary;
```

Ключевые слова ANY и ALL

Ключевые слова ANY и ALL могут использоваться с подзапросами, возвращающими один столбец чисел. Если подзапросу будет предшествовать ключевое слово ALL, условие сравнения считается выполненным только в том случае, если оно выполняется для всех значений в результирующем столбце подзапроса. Если тексту подзапроса предшествует ключевое слово ANY, то условие сравнения будет считаться выполненным, если оно удовлетворяется хотя бы для какого-либо (одного или нескольких) значения в результирующем столбце подзапроса. Если в результате выполнения подзапроса будет получено пустое значение, то для ключевого слова ALL условие сравнения будет считаться выполненным, а для ключевого слова ANY — невыполненным. Согласно стандарту ISO дополнительно можно использовать ключевое слово SOME, являющееся синонимом ключевого слова ANY.

Пример

Найдите всех работников, чья зарплата превышает зарплату хотя бы одного *работника отделения компании под номером 'вооз '.*

```
SELECT staffNo, fName, IName, position, salary
FROM Staff
WHERE salary > SOME(SELECT salary
FROM Staff
WHERE branchNo = 'B003');
```

Далее, *коррелированный* подзапрос – это особый вид подзапроса (табличного, однострочного или скалярного), а именно такой, в котором есть ссылка на некоторую «внешнюю» таблицу. В следующем примере заключенное в скобки выражение после ключевого слова IN и есть коррелированный подзапрос, потому что включает ссылку на внешнюю таблицу

S (запрос звучит так: «Получить названия поставщиков, которые поставляют деталь P1»).

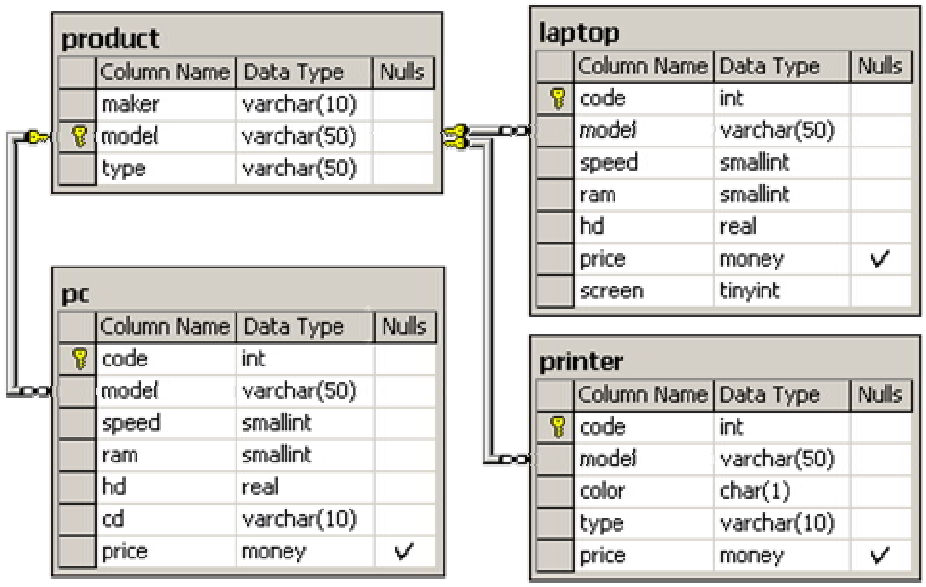
```
SELECT DISTINCT S.SNAME

FROM S

WHERE ‘P1’ IN ( SELECT PNO

FROM SP

WHERE SP.SNO = S.SNO )
```

Экзистенциальные запросы в языке SQL

Ключевые слова EXISTS и NOT EXISTS предназначены для использования только совместно с подзапросами или управляющими конструкциями. Результат их обработки представляет собой логическое значение TRUE или FALSE. Для ключевого слова EXISTS результат равен TRUE в том и только в том случае, если в возвращаемой подзапросом результирующей таблице присутствует хотя бы одна строка. Если результирующая таблица подзапроса пуста, результатом обработки ключевого слова EXISTS будет значение FALSE. Для ключевого слова NOT EXISTS используются правила обработки, обратные по отношению к ключевому слову EXISTS. Поскольку по ключевым словам EXISTS и NOT EXISTS проверяется лишь наличие строк в результирующей таблице подзапроса, то эта таблица может содержать произвольное количество столбцов.
Например группа где есть студенты будет получена запросом
Select st_group.* from st_group where
exists (select id_student from student where student.id_gr=st_group.id_gr)

Часто условие NOT EXISTS используется для реализации разности.
Например Группа без студентов может быть найдена запросом
Select st_group.* from st_group where
not exists (select id_student from student where student.id_gr=st_group.id_gr)
Обратите внимание, что в подзапросе сквозная видимость

Одно из назначений NOT EXISTS – реализация реляционного деления (так называемые запросы на «все»).
Для реализации реляционного деления с помощью NOT EXISTS необходимо переформулировать запрос в форму с двойным отрицанием. Например запрос для схемы «Компьютерная фирма»



«Определить производителей, которые выпускают модели всех типов»

Может быть переформулирован
«Определить производителей, для которых Не должно существовать такого типа продукции, для которого (типа продукции) бы не было данного производителя.
Это деление производителя (maker) на тип (type)

И запрос будет выглядеть

```
SELECT DISTINCT maker
FROM Product Pr1
WHERE NOT EXISTS
 (SELECT type
   FROM Product
   WHERE NOT EXISTS
          (SELECT *
            FROM Product Pr2
            WHERE Pr1.maker = Pr2.maker
           AND Product.type=Pr2.type
          )
  );
```

На самом деле такой запрос можно разделить на 3 части- запроса:
A
NOT EXISTS
(B
NOT EXISTS (
C))
При этом каждая часть имеет свое назначение.
Части A и C отвечают за связку между делимым и делителем (производитель и продукция)
Запросы A и C как правило похожи с точностью до псевдонимов (сквозная видимость-псевдонимы обязательны), однако в C идет дополнительная связка с A по ключу делимого (в данном случае maker) и связка C и B по ключу делителя (в данном случае type)
Запрос B задает делитель.
То есть если бы была необходимость выбрать поставщика, который поставляет все типы товаров на букву 'A', то изменилась бы только часть B и запрос бы выглядел:

```
SELECT DISTINCT maker
FROM Product Pr1
WHERE NOT EXISTS
 (SELECT type
   FROM Product
   WHERE  type like ‘A%’
    AND NOT EXISTS
          (SELECT *
            FROM Product Pr2
            WHERE Pr1.maker = Pr2.maker
           AND Product.type=Pr2.type
          )
  );
```

Тема 2.2. Операторы манипулирования данными в языке SQL

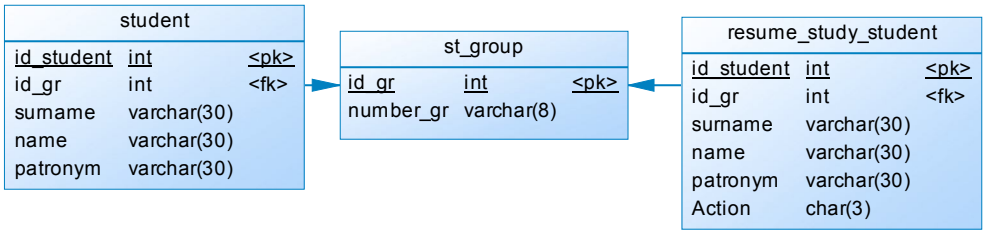
Добавление новых данных в таблицу (оператор INSERT)

Существуют две формы оператора INSERT. Первая предназначена для вставки единственной строки в указанную таблицу. Эта форма оператора INSERT имеет следующий формат:
INSERT INTO *TableName* [(*columnList*)]
VALUES (*dataValueList*)

Здесь параметр *TableName* (Имя таблицы) может представлять либо имя таблицы базы данных, либо имя обновляемого представления (раздел 6.4). Параметр *columnList* (Список столбцов) представляет собой список, состоящий из имен одного или более столбцов, разделенных запятыми. Параметр coIumnList является необязательным. Если он опущен, то предполагается использование списка из имен всех столбцов таблицы, указанных в том порядке, в котором они были описаны в операторе CREATE TABLE. Если в операторе INSERT указывается конкретный список имен столбцов, то любые опущенные в нем столбцы должны быть объявлены при создании таблицы как допускающие значение NULL — заисключением случаев, когда при описании столбца использовался параметр DEFAULT . Параметр *dataValueList* (Список значений данных) должен следующим образом соответствовать параметру *columnList:*

- количество элементов в обоих списках должно быть одинаковым;
- должно существовать прямое соответствие между позицией одного и того же элемента в обоих списках, поэтому первый элемент списка *dataValueList* считается относящимся к первому элементу списка *columnList*, второй элемент списка *dataValueList* — ко второму элементу списка *columnList* и т.д.;
- типы данных элементов списка *dataValueList* должны быть совместимы с типом данных соответствующих столбцов таблицы.

Пример:



Вставим данные в таблицу студент (студент в группу с идентификатором 2)

```
INSERT INTO student (id_student, id_gr, surname, name, patronym)
                    values (10,          2,     'Иванов','Иван','Иванович')
```

Или

```
INSERT INTO student  values (10, 2, 'Иванов','Иван','Иванович')
```

Или допустим, что мы не знаем в какую группу студент должен быть зачислен, тогда вставка будет выглядеть следующим образом:

```
INSERT INTO student (id_student, surname, name, patronym)
                    values (10,          'Иванов','Иван','Иванович')
```

Или

```
INSERT INTO student (id_student, id_gr, surname, name, patronym)
                    values (10,          NULL,     'Иванов','Иван','Иванович')
```

Вторая форма оператора INSERT позволяет скопировать множество строк одной таблицы в другую. Этот оператор имеет следующий формат:
INSERT INTO *TableName* [(*columnList*)]
SELECT …
Здесь параметры TableName и *columnList* имеют тот же формат и смысл, что и при вставке в таблицу одной строки. Конструкция SELECT может представлять собой любой

допустимый оператор SELECT. Строки, вставляемые в указанную таблицу, в точности соответствуют строкам результирующей таблицы, созданной при выполнении вложенного запроса. Все ограничения, указанные выше для первой формы оператора INSERT, применимы и в этом случае.
Пример: вставка студентов, восстановившихся в вузе (считаем, что их удаляют из рабочей базы) из таблицы resume_study_student

```
INSERT INTO student (id_student, id_gr, surname, name, patronym)
SELECT id_student, id_gr, surname, name, patronym from resume_study_student
```

Или

```
INSERT INTO student
SELECT id_student, id_gr, surname, name, patronym from resume_study_student
```

Модификация данных в базе (оператор UPDATE)

Оператор UPDATE позволяет изменять содержимое уже существующих строк указанной таблицы. Этот оператор имеет следующий формат:
UPDATE. *TableName*
SET *columnName1 = dataValue1 [, columnName2= dataValue2]*
[WHERE *searchCondition]*

Здесь параметр *TableName* представляет либо имя таблицы базы данных, либо имя обновляемого представления. В конструкции SET указываются имена одного или более столбцов, данные в которых необходимо изменить. Конструкция WHERE является необязательной. Если она опущена, значения указанных столбцов будут изменены во *всех* строках таблицы. Если конструкция WHERE присутствует, то обновлены будут только те строки, которые удовлетворяют условию поиска, заданному в параметре *searchCondition.* Параметры *dataValue1, dataValue2...* представляют новые значения соответствующих столбцов и должны быть совместимы с ними по типу данных.
Условия применимые в разделе WHERE приведены в описании оператора Select

Удаление данных из базы (оператор DELETE)

Оператор DELETE позволяет удалять строки данных из указанной таблицы.
Этот оператор имеет следующий формат:
DELETE FROM *TableName*
[WHERE searchCondition]

Условия применимые в разделе WHERE приведены в описании оператора Select

Слияние данных (оператор MERGE)

Оператор MERGE в стандарте с SQL:2003, улучшен в SQL:2008, удаление с SQL:2011)

Выполняет операции вставки, обновления или удаления для целевой таблицы на основе результатов соединения с исходной таблицей. Например, можно синхронизировать две таблицы путем вставки, обновления или удаления строк в одной таблице на основании отличий, найденных в другой таблице.

MERGE

```
   [ TOP ( expression ) [ PERCENT ] ]
   [ INTO ] <target_table> [ WITH ( <merge_hint> ) ] [ [ AS ] table_alias ]
   USING <table_source>
   ON <merge_search_condition>
   [ WHEN MATCHED [ AND <clause_search_condition> ]
      THEN <merge_matched> ] [ ...n ]
   [ WHEN NOT MATCHED [ BY TARGET ] [ AND <clause_search_condition> ]
      THEN <merge_not_matched> ]
   [ WHEN NOT MATCHED BY SOURCE [ AND <clause_search_condition> ]
      THEN <merge_matched> ] [ ...n ]
   [ <output_clause> ]
   [ OPTION ( <query_hint> [ ,...n ] ) ]
;
```

Аргументы

TOP (expression) [PERCENT]

> Указывает количество или процент строк, которые подпадают под эту операцию. Аргумент expression может быть либо числом, либо процентной долей строк. Строки, на которые ссылается выражение TOP, не расположены в определенном порядке.

Предложение TOP применяется после соединения всей исходной таблицы и всей целевой таблицы и удаления соединенных строк, которые не рассматриваются как предназначенные для выполнения операций вставки, обновления или удаления. Предложение TOP дополнительно сокращает количество соединенных строк до указанного значения, а затем к оставшимся соединенным строкам применяются операции вставки, обновления или удаления без учета порядка. Иными словами, порядок, в котором строки подвергаются операциям, определенным в предложениях WHEN, не задан. Например, указание значения TOP (10) затрагивает 10 строк. Из них 7 могут быть обновлены и 3 вставлены или 1 может быть удалена, 5 обновлено и 4 вставлено и т. д.

Инструкция MERGE выполняет полный просмотр исходной и целевой таблиц, поэтому при использовании предложения TOP для изменения большой таблицы путем создания нескольких пакетов производительность ввода-вывода может снизиться. В этом случае необходимо обеспечить, чтобы во всех подряд идущих пакетах осуществлялась обработка новых строк.

target_table

> Таблица или представление, с которыми выполняется сопоставление строк данных из таблицы <table_source> по условию <clause_search_condition>. Параметр target_table является целевым объектом любых операций вставки, обновления или удаления, указанных предложениями WHEN инструкции MERGE.

USING <table_source>

> Указывается источник данных, который сопоставляется со строками данных в таблице target_table на основе условия <merge_search condition>. Результат этого

совпадения обуславливает действия, которые выполняются предложениями WHEN инструкции MERGE. Аргумент <table_source> может быть удаленной таблицей или производной таблицей, которая обращается к удаленным таблицам.

ON <merge_search_condition>

Указываются условия, при которых таблица <table_source> соединяется с таблицей target_table для сопоставления. Необходимо указать столбцы целевой таблицы, которые сравниваются с соответствующим столбцом исходной таблицы.

WHEN MATCHED THEN <merge_matched>

Указывается, что все строки target_table, которые соответствуют строкам, возвращенным <table_source> ON <merge_search_condition>, и удовлетворяют дополнительным условиям поиска, обновляются или удаляются в соответствии с предложением <merge_matched>.

Инструкция MERGE может иметь не больше двух предложений WHEN MATCHED. Если указаны два предложения, первое предложение должно сопровождаться предложением AND <search_condition>. Для любой строки второе предложение WHEN MATCHED применяется только в тех случаях, если не применяется первое. Если имеются два предложения WHEN MATCHED, одно должно указывать действие UPDATE, а другое — действие DELETE. Если действие UPDATE указано в предложении <merge_matched> и более одной строки в <table_source>соответствует строке в target_table на основе <merge_search_condition>, то SQL Server возвращает ошибку. Инструкцию MERGE нельзя использовать для обновления одной строки более одного раза, а также использовать для обновления и удаления одной и той же строки.

WHEN NOT MATCHED [BY TARGET] THEN <merge_not_matched>

Указывает, что строка вставлена в таблицу target_table для каждой строки, возвращенной выражением <table_source> ON <merge_search_condition>, которая не соответствует строке в таблице target_table, но удовлетворяет дополнительному условию поиска (при наличии такового). Значения для вставки указываются с помощью предложения <merge_not_matched>. Инструкция MERGE может иметь только одно предложение WHEN **NOT** MATCHED.

WHEN NOT MATCHED BY SOURCE THEN <merge_matched>

Указывается, что все строки target_table, которые не соответствуют строкам, возвращенным <table_source> ON <merge_search_condition>, и удовлетворяют дополнительным условиям поиска, обновляются или удаляются в соответствии с предложением <merge_matched>.

Инструкция MERGE может иметь не более двух предложений WHEN NOT MATCHED BY SOURCE. Если указаны два предложения, то первое предложение должно сопровождаться предложением AND <clause_search_condition>. Для любой выбранной строки второе предложение WHEN NOT MATCHED BY SOURCE применяется только в тех случаях, если не применяется первое. Если имеется два предложения WHEN NOT MATCHED BY SOURCE, то одно должно указывать

действие UPDATE, а другое — действие DELETE. В условии <clause_search_condition> можно ссылаться только на столбцы целевой таблицы.

<merge_matched>

Указывает действие обновления или удаления, которое применяется ко всем строкам target_table, не соответствующим строкам, которые возвращаются <table_source> ON <merge_search_condition> и удовлетворяют каким-либо дополнительным условиям поиска.

UPDATE SET <set_clause>

Указывается список имен столбцов или переменных, которые необходимо обновить в целевой таблице, и значений, которые необходимо использовать для их обновления.

<merge_not_matched>

Указываются значения для вставки в целевую таблицу.

(column_list)

Список, состоящий из одного или нескольких столбцов целевой таблицы, в которые вставляются данные. Столбцы необходимо указывать в виде однокомпонентного имени, так как в противном случае инструкция MERGE возвращает ошибку. Список column_list должен быть заключен в круглые скобки, а его элементы должны разделяться запятыми.

VALUES (values_list)

Список с разделителями-запятыми констант, переменных или выражений, которые возвращают значения для вставки в целевую таблицу. Выражения не могут содержать инструкцию EXECUTE.

DEFAULT VALUES

Заполняет вставленную строку значениями по умолчанию, определенными для каждого столбца.

<search condition>

Указываются условия поиска, используемые для указания <merge_search_condition> или <clause_search_condition>.

Пример сольем таблицу студентов и восстановившихся студентов (результат в таблице Student)

Поле Action может иметь 3 значения :

‘New’ – добавить строку со студентом в таблицу Student

‘Mod’ – изменить строку со студентом в таблице Student (например другая группа)

‘Del’ – удалить студента из таблицы Student

```
MERGE INTO Student AS S
USING resume_study_student AS R
ON S.id_student = R.id_student
WHEN MATCHED AND
R.Action = 'Mod'
THEN UPDATE
SET  id_gr=R.id_gr, surname= R.surname, name=R.name, patronym =R.patronym
WHEN MATCHED AND
R.Action = 'Del'
THEN DELETE
WHEN NOT MATCHED AND
I.Action = 'New'
THEN INSERT
VALUES (R.id_student,R.id_gr, R.surname, R.name, R.patronym)
```

Трехзначная логика

В повседневной жизни часто приходится сталкиваться с проблемой отсутствия некоторой информации. Весьма типичны ситуации, когда, например, "дата рождения не известна", "имя докладчика будет объявлено дополнительно", "адрес лица в данный момент не известен" и т.д. Поэтому в системах баз данных должен существовать механизм обработки подобных ситуаций. На практике наиболее типичный подход к решению этой проблемы (используемый, в частности, в языке SQL, а значит, и в большинстве коммерческих продуктов) основан на применении **неопределенных значений** (NULL- значений) и **трехзначной логики.** Например, вес детали, скажем, с номером P7, может быть не известен, поэтому упрощенно можно сказать, что ее вес "является неопределенным". В более точном смысле это выражение означает следующее: а) известно, что деталь существует; б) несомненно, деталь имеет вес; в) ее вес нам не известен. Рассмотрим этот пример более подробно. Ясно, что мы не можем поместить истинное значение веса детали в атрибут WEIGHT кортежа, описывающего деталь с номером P7. Следовательно, все, что остается, — *пометить* или как-то *обозначить,* что значение атрибута WEIGHT этого кортежа является неопределенным. В дальнейшем наличие подобной метки будет интерпретироваться как указание, что истинное значение атрибута не известно. Для удобства можно неформально сказать, что атрибут WEIGHT "содержит неопределенное значение" или что значение этого атрибута "равно NULL".

В следующем разделе будет показано, что при сравнении любых скалярных значений, в которых какой-либо из операндов не определен (содержит NULL), вместо значения *true* (истина) или *false* (ложь) будет получено логическое значение *unknown* (не известно).Причиной такого состояния дел является принятая нами интерпретация обозначения NULL как указателя "значение не известно". Если значение переменной A не известно, то не известен и результат любого ее сравнения, например вида A > в, причем независимо от значения в (даже если предположить, что значение переменной в также не известно). В частности, следует отметить, что два неопределенных значения (NULL) нельзя считать равными одно другому, поэтому, если обе переменные, A и в, содержат неопределенныезначения (NULL), результатом сравнения A = в всегда будет *unknown,* а

не *true.* Однако эти переменные не считаются и неравными, т.е. результатом сравнения A не =B также будет *unknown.* На этом и основано понятие трехзначной логики, поскольку концепция неопределенных значений, по крайней мере, в общепринятом смысле, неизбежно приводит нас к необходимости использования логики с тремя логическими значениями: *true (истина), false* (ложь) и *unknown* (не известно).

Логические операторы

Выше уже отмечалось, что результатом операций сравнения скаляров, в которых хотя бы один из операндов является величиной UNK, будет логическое значение *unknown,* а не значение *true* или *false.* Поэтому в таких случаях приходится иметь дело с трехзначной логикой. Третьим логическим значением здесь является значение *unknown,* на которое мы достаточно часто, но не постоянно, будем ссылаться с помощью сокращения *unк.* Ниже приведены таблицы истинности для операторов AND, OR и NOT в трехзначной логике (в таблицах используются следующие сокращения: t — *true, f—false, u — unк).*

AND	t	u	f
t	t	u	f
u	u	u	f
f	f	f	f

OR	t	u	f
t	t	t	t
u	t	u	u
f	t	u	f

NOT	
t	f
u	u
f	t

Предположим, что A = 3, b = 4 и переменная с имеет значение UNK. Тогда приведенные ниже выражения будут иметь следующие результаты (которые показаны справа).

A > B AND B > C : *false*
A > B OR B > C : *unk*
A < B OR B < C : true
NOT (A = C) : *unk*

Тем не менее, для реализации трехзначной логики одних только операторов AND, OR и NOT недостаточно. Еще одним важным оператором является оператор MAYBE (возможно). Таблица истинности данного оператора показана ниже.

MAYBE	
t	f
u	t
f	f

Преобразование выражений

Прежде всего, отметим, что выражения, которые в двухзначной логике всегда возвращают значение *true,* в трехзначной логике не обязательно будут всегда давать тот же результат. Ниже приведено несколько примеров с комментариями, но следует иметь в виду, что этот список далеко не полный.

- *Сравнение* x = x *не обязательно в результате даст true.*

В двухзначной логике любая переменная x всегда равна самой себе. В трехзначной логике переменная x не равна самой себе, если она содержит величину UNK.

- *Логическое выражение* p OR NOT (p) *не обязательно в результате даст true.*

Здесь p — это некоторое логическое выражение. В двухзначной логике выражение p OR NOT(p) всегда имеет значение *true* (т.е. истинно), независимо от значения p. Но в трехзначной логике, если p равно *unk,* общее выражение сводится к *unk* OR NOT (*unk),* т.е. к выражению *unk* OR *unk,* что в свою очередь упрощается до значения *unk,* а не *true.* Этот частный пример демонстрирует хорошо известное, но сложное для понимания свойство трехзначной логики, которое можно описать следующим образом. Если выполнить два запроса, "Получить сведения обо всех поставщиках из Лондона" и "Получить сведения обо всех поставщиках не из Лондона", а затем объединить результаты обоих запросов, то *не* обязательно будут по лучены сведения обо *всех* поставщиках. Чтобы получить список всех поставщиков, к двум запросам нужно добавить еще один: "Получить сведения обо всех поставщиках, которые, *возможно* (maybe), находятся в Лондоне" (иными словами, вт рехзначной логике выражением, которое всегда принимает

значение *true,* т.е. является аналогом выражения p OR NOT (p) двухзначной логики, служит выражение p OR NOT(p) OR MAYBE (p)).
Вычисление логического выражения r JOIN r *не обязательно в результате даст r.*
В двухзначной логике соединение отношения r с самим собой всегда дает в результате исходное отношение r (т.е. операция естественного соединения является*идемпотентной).*
Однако в трехзначной логике кортеж, содержащий величину UNK
в любой из позиций, не будет соединен сам с собой, поскольку операция соединения, в отличие от операции объединения, предусматривает проверку
на равенство "в стиле выборки", а не проверку на равенство "в стиле исключения дубликатов кортежей" (именно так!?).

- *Операция INTERSECT больше не является частным случаем операции JOIN.*

Это заключение является следствием того факта, что, опять же, операция соединения предусматривает проверку на равенство в стиле выборки, тогда как операция пересечения основана на проверке на равенство в стиле исключения дубликатов.

- *Из равенства A = B AND B = C вовсе не обязательно следует равенство A = C.*

Вопреки всему сказанному в предыдущем разделе, на практике неопределенные значения (NULL) и трехзначная логика широко поддерживаются в большинстве современных программных продуктов.

Средства Языка Sql

Поддержка неопределенных значений (NULL) и трехзначной логики в языке SQL отражает весь широкий спектр подходов, описанных в предыдущих разделах. Так, например, когда в языке SQL условие WHERE применяется к некоторой таблице т, при этом
исключаются из рассмотрения все строки таблицы т, для которых указанное в конструкции WHERE выражение принимает значение *false* или *unk* (т.е. не *true).*
Аналогичным образом, когда к результату выполнения некоторой операции группирования, представленному таблицей G, применяется конструкция HAVING, ИЗ дальнейшего рассмотрения
исключаются все группы кортежей таблицы G, для которых указанное в конструкции HAVING выражение принимает значение *false* или *unk* (т.е. не *true).*
Из этого следует, что мы просто обратили внимание читателя на некоторые средства поддержки трехзначной логики, характерные для языка SQL как такового, а не являющиеся неотъемлемой частью описанного выше подхода, основанного на использовании трехзначной логики.
Как уже было показано , в языке SQL предусмотрен встроенный тип BOOLEAN (он был введен в стандарт в 1999 году, но в настоящее время его поддерживают
лишь немногие продукты, если вообще таковые имеются). Предусмотрены обычные логические операторы AND, OR и NOT, и логические выражения могут присутствовать в любом месте, где обычно допускается наличие скалярных выражений. Но, как указано в данной главе, теперь определены три истинностных значения, а не два (соответствующими литералами являются TRUE, FALSE и UNKNOWN), но несмотря на этот факт, тип BOOLEAN включает только два значения, а не три. Таким образом, неизвестное *{unknown)* истинностное значение совершенно неправильно представлено с помощью неопределенного значения NULL! Ниже описаны некоторые следствия из этого факта.
■ Присваивание значения UNKNOWN переменной в типа BOOLEAN фактически равно сильно присваиванию этой переменной неопределенного значения.
■ После такого присваивания операция сравнения в = UNKNOWN не дает в результате *true* (или, скорее, TRUE); вместо этого результатом становится неопределенное значение.
■ В действительности, операция сравнения в = UNKNOWN всегда приводит к полу

чению неопределенного значения, независимо от значения в! Дело в том, что та кая операция логически эквивалентна операции сравнения "в = NULL" (которая не рассматривается как синтаксически допустимая).

Логические выражения

Не удивительно то, что в языке SQL логические выражения испытывают сильное влияние неопределенных значений и трехзначной логики. Остановимся на рассмотрении лишь нескольких наиболее важных частных случаев, которые описаны ниже.

■ *Проверка наличия неопределенного значения.* В языке SQL предусмотрены два специальных оператора сравнения, IS NULL и IS NOT NULL, предназначенных для проверки наличия или отсутствия неопределенных значений. Синтаксис использования этих операторов показан ниже.

<row value constructor IS [NOT] NULL

Если в выражении *<row value constructor* с определением конструктора значения строки "конструируется" строка со степенью один, то в языке SQL это выражение рассматривается так, как будто оно фактически указывает на значение, содержащееся в этой строке, а не на саму строку как таковую; в противном случае это выражение рассматривается как указывающее на строку. Но в последнем случае считается, что строка имеет неопределенное значение тогда и только тогда, когда каждый ее компонент является неопределенным, а не имеет неопределенного

значения — тогда и только тогда, когда каждый ее компонент имеет значение, отличное от неопределенного! Одним следствием этой ошибки является то, что если r —строка с двумя компонентами, то выражения r IS NOT NULL и NOT (r IS NULL) *не являются* эквивалентными; первое из них эквивалентно выражению cl is NOT NULL AND c2 is NOT NULL, а второе — выражению с 1 IS NOT NULL OR c2 IS NOT NULL. Еще одним следствием из указанного является то, что если строка r одновременно включает и компоненты с неопределенными, и компоненты с определенными значениями, то сама строка r, безусловно, не может рассматриваться ни как имеющая неопределенное значение, ни как имеющая определенное значение.

Условия EXISTS. Оператор EXISTS языка SQL не является аналогичным квантору существования в трехзначной логике, поскольку он всегда возвращает значение *true* или *false,* но не *unk,* даже если *unk* — логически правильный ответ. А именно, этот оператор возвращает *false,* если таблица, заданная в нем в качестве фактического параметра, пуста, и true — в ином случае (Поэтому он иногда возвращает *true,* тогда как *unк*— логически правильный ответ).

■ Условия UNIQUE. Неформально говоря, условия UNIQUE служат для проверки того, что указанная таблица не содержит дубликатов строк (именно так!). Точнее, выражение UNIQUE (*< table exp>)* возвращает true, если таблица, обозначенная с помощью параметра *< table exp>,* не включает двух разных строк, скажем, rl и r2, таких что операция сравнения rl = r2 возвращает true; в противном случае данное выражение возвращает *false.* Поэтому UNIQUE, как и EXISTS, иногда возвращает true, даже когда *unк* — логически правильный ответ.

■ Условия DISTINCT. Условия DISTINCT, вообще говоря, предназначены для про верки того, не являются ли две строки дубликатами друг друга. Обозначим две рассматриваемые строки как Left и Right; строки Left и Right должны иметь одинаковую степень, скажем, п. Допустим, что i-ми компонентами Left и Right, соответственно, являются Li и Ri (i = 1, 2, . . ., п); все компоненты Li и Ri должны быть такими, что операция сравнения Li = Ri является действительной.

В таком случае следующее выражение

Left IS DISTINCT FROM Right

возвращает *false,* если для всех i либо операция сравнения "Li = Ri" возвращает true, либо оба значения, Li и Ri, являются неопределенными; в противном случае указанное выражение возвращает true.

Другие скалярные выражения И снова рассмотрим лишь несколько важных частных случаев, которые описаны ниже.
■ "Литералы". Ключевое слово NULL может иногда использоваться как своего рода литеральное представление неопределенного значения (например, в операторе INSERT), но не во всех контекстах; в стандарте SQL указано, что "ключевое слово NULL ... может применяться для обозначения неопределенного значения... в некоторых контекстах, но повсеместное использование этого литерала не допускается". В частности, следует отметить, что ключевое слово NULL нельзя применять для обозначения операнда простого оператора сравнения; например, выражение "WHERE x = NULL" является .недопустимым (правильная форма, безусловно,WHERE X IS NULL).
■ COALESCE. Оператор COALESCE — это аналог предложенного автором оператора IF_UNK в языке SQL. Точнее, выражение COALESCE (x,y, . . . , z) возвращает неопределенное значение, если все его параметры x, y, . . ., z возвращают не определенное значение; в противном случае оно возвращает первый свой операнд, отличный от неопределенного значения.
■ Агрегирующие операторы. Агрегирующие операторы SQL (SUM, AVG и т.д.) не действуют в соответствии с правилами для скалярных операторов, а вместо этого просто игнорируют все неопределенные значения своих фактических параметров (не считая выражения COUNT (*), в котором неопределенные значения рассматриваются так, как если бы они были определенными
значениями). Кроме того, если оказалось, что фактический параметр такого оператора соответствует пустому множеству, то оператор COUNT возвращает нуль, а все остальные операторы — неопределенное значение. (Как было указано в главе 8, такая организация функционирования логически не обоснована, но язык SQL определен именно так.)
■ "Скалярные подзапросы". Если скалярное выражение фактически представляет собой табличное выражение, заключенное в круглые скобки, например, (SELECT S.CITY FROM S WHERE S . S# = S# (' S1 ')) , ТО В обычных условиях это табличное выражение должно давать в результате таблицу, содержащую точно один столбец и точно одну строку. Поэтому значением этого скалярного выражения должно служить единственное скалярное значение, содержащееся внутри этой таблицы. Но если результатом вычисления данного табличного выражения становится таблица с одним столбцом, вообще не содержащая строк, то в языке SQL значение скалярного выражения задается как неопределенное.

Тема 2.3.Хранисмые процедуры, программирование серверной части базы данных.

Управляющие конструкции в языке SQL

До появления стандарта SQL3 язык SQL включал только команды определения и манипулирования данными; в нем отсутствовали какие-либо команды управления ходом вычислений. Другими словами, в этом языке не было команд IF ... THEN ... ELSE, GO TO, DO ... WHILE и любых других, предназначенных для управления ходом вычислительного процесса. Подобные задачи должны были решаться программным путем (с помощью языков программирования или управления заданиями) либо интерактивно (в результате действий, выполняемых самим пользователем).
Поэтому синтаксис в различных СУБД может заметно различаться. Рассмотрим синтаксис для Transact SQL (Microsoft)
Управляющие конструкции могут быть использованы в пакетах, хранимых процедурах и нерегламентированных запросах.
BEGIN...END
Включает в себя последовательность инструкций языка Transact-SQL, позволяя выполнять группу инструкций Transact-SQL.
Синтаксис

BEGIN
{ sql_statement | statement_block }
END

Аргументы

{ *sql_statement* | *statement_block* }
Любая допустимая инструкция или группа инструкций языка Transact-SQL, определенная с помощью блока инструкций.

IF...ELSE

Инструкция языка Transact-SQL, следующая за ключевым словом IF и его условием, выполняется только в том случае, если логическое выражение возвращает TRUE. Необязательное ключевое слово ELSE представляет другую инструкцию языка Transact-SQL, которая выполняется, если условие IF не удовлетворяется и логическое выражение возвращает FALSE.

Синтаксис

IF Boolean_expression
{ sql_statement | statement_block }
[ELSE
{ sql_statement | statement_block }]

Аргументы

Boolean_expression
Выражение, возвращающее значение TRUE или FALSE. Если логическое выражение содержит инструкцию SELECT, инструкция SELECT должна быть заключена в скобки.
{ *sql_statement*| *statement_block* }
Любая инструкция или группа инструкций языка Transact-SQL, указанная с помощью блока инструкций. Без использования блока инструкций условия IF и ELSE могут повлиять на выполнение только одной инструкции языка Transact-SQL.

WHILE

Ставит условие повторного выполнения SQL-инструкции или блока инструкций. Эти инструкции вызываются в цикле, пока указанное условие истинно. Вызовами инструкций в цикле WHILE можно контролировать из цикла с помощью ключевых слов BREAK и CONTINUE.

Синтаксис

WHILE Boolean_expression
{ sql_statement | statement_block | BREAK | CONTINUE }

Аргументы

Boolean_expression
— Выражение , возвращающее **TRUE** или **FALSE**. Если логическое выражение содержит инструкцию SELECT, инструкция SELECT должна быть заключена в скобки.

{*sql_statement* | *statement_block*}
Любая инструкция или группа инструкций Transact-SQL, определенная в виде блока инструкций. Для определения блока инструкций используйте ключевые слова потока управления BEGIN и END.

BREAK
Приводит к выходу из ближайшего цикла WHILE. Вызываются инструкции, следующие за ключевым словом END, обозначающим конец цикла.

CONTINUE
Выполняет цикл WHILE для перезагрузки, не учитывая все инструкции, следующие после ключевого слова CONTINUE.

Если вложенными являются два цикла WHILE или более, внутренний оператор BREAK существует до следующего внешнего цикла. Все инструкции после окончания внутреннего цикла выполняются в первую очередь, а затем перезапускается следующий внешний цикл.

Пример

Использование ключевых слов BREAK и CONTINUE внутри вложенных конструкций IF...ELSE и WHILE

В следующем примере в случае, если средняя цена продуктов из списка меньше чем $300, цикл WHILE удваивает цены, а затем выбирает максимальную. В том случае, если максимальная цена меньше или равна $500, цикл WHILE повторяется и снова удваивает цены. Этот цикл продолжает удваивать цены до тех пор, пока максимальная цена не будет больше чем $500, затем выполнение цикла WHILE прекращается, о чем выводится соответствующее сообщение.

```
USE AdventureWorks2012;
GO
WHILE (SELECT AVG(ListPrice) FROM Production.Product) < $300
BEGIN
  UPDATE Production.Product
    SET ListPrice = ListPrice * 2
  SELECT MAX(ListPrice) FROM Production.Product
  IF (SELECT MAX(ListPrice) FROM Production.Product) > $500
    BREAK
  ELSE
    CONTINUE
END
PRINT 'Too much for the market to bear';
```

Выражение CASE

Case Оценивает список условий и возвращает одного из нескольких возможных результирующих выражений.

CASE имеет два формата:

- простое выражение CASE для определения результата сравнивает выражение с набором простых выражений;
- поисковое выражение CASE для определения результата вычисляет набор логических выражений.

Оба формата поддерживают дополнительный аргумент ELSE.

Выражение CASE может использоваться в любой инструкции или предложении, которые допускают допустимые выражения. Например, выражение CASE можно использовать в таких инструкциях, как SELECT, UPDATE, DELETE и SET, а также в таких предложениях, как select_list, IN, WHERE, ORDER BY и HAVING.

Синтаксис

Simple CASE expression:

```
CASE input_expression
    WHEN when_expression THEN result_expression [ ...n ]
    [ ELSE else_result_expression ]
END
```

Searched CASE expression:

```
CASE
    WHEN Boolean_expression THEN result_expression [ ...n ]
    [ ELSE else_result_expression ]
END
```

Аргументы

input_expression
Выражение, полученное при использовании простого формата функции CASE. *input_expression* может быть любым допустимым выражением.

WHEN *when_expression*
Простое выражение, с которым *input_expression* сравнивается при использовании простого формата функции CASE. *when_expression* любое допустимое выражение. Типы данных *input_expression* и каждого *when_expression* должны быть одинаковыми или иметь неявное преобразование между типами.

THEN *result_expression*
Это выражение, возвращаемое когда *input_expression* = *when_expression* возвращает значение TRUE, или *Boolean_expression* возвращает значение TRUE. *result_expression* может быть любым допустимым выражением.

ELSE *else_result_expression*
Это выражение, возвращаемое, если ни одна из операций сравнения не дает в результате TRUE. Если этот аргумент опущен и ни одна из операций сравнения не дает в результате TRUE, функция CASE возвращает NULL. else_result_expression — любое допустимое выражение. Типы данных else_result_expression и result_expression должны быть одинаковыми или иметь неявное преобразование между типами.

WHEN *Boolean_expression*
Логическое выражение, полученное при использовании поискового формата функции CASE. *Boolean_expression* любое допустимое логическое выражение.

Возвращаемые значения

Простое выражение CASE:

Простое выражение CASE сравнивает первое выражение с выражением в каждом предложении WHEN. Если эти выражения эквивалентны, то возвращается выражение в предложении THEN.

- Допускается только проверка равенства.
- В указанном порядке вычисляет input_expression = when_expression для каждого предложения WHEN.
- Возвращает *result_expression* первого *input_expression = when_expression* , имеющего значение TRUE.
- Если ни одно сравнение *input_expression = when_expression* не вернуло TRUE, Компонент SQL Server Database Engine возвращает *else_result_expression,* если часть else указана, или значение NULL, если часть ELSE не указана.

Поисковое выражение CASE:

- Вычисляет в указанном порядке *Boolean_expression* для каждого предложения WHEN.
- Возвращает *result_expression* первого *Boolean_expression* , имеющего значение TRUE.
- Если не *Boolean_expression* имеет значение TRUE, Компонент Database Engine возвращает *else_result_expression случая* Если указано предложение ELSE, или значение NULL, если предложение ELSE не указано.

Замечания

SQL Server допускает применение в выражениях CASE не более 10 уровней вложенности.

Выражение CASE нельзя использовать для управления потоком выполнения инструкций Transact-SQL, блоков инструкций, определяемых пользователем функций и хранимых процедур.

Пример (SELECT с простым выражением CASE)

```
USE AdventureWorks2012;
GO
SELECT   ProductNumber, Category =
    CASE ProductLine
      WHEN 'R' THEN 'Road'
      WHEN 'M' THEN 'Mountain'
      WHEN 'T' THEN 'Touring'
      WHEN 'S' THEN 'Other sale items'
      ELSE 'Not for sale'
    END,
  Name
FROM Production.Product
ORDER BY ProductNumber;
GO
```

Пример (SELECT с поисковым выражением CASE)

```
USE AdventureWorks2012;
GO
```

```
SELECT   ProductNumber, Name, "Price Range" =
     CASE
       WHEN ListPrice =  0 THEN 'Mfg item - not for resale'
       WHEN ListPrice < 50 THEN 'Under $50'
       WHEN ListPrice >= 50 and ListPrice < 250 THEN 'Under $250'
       WHEN ListPrice >= 250 and ListPrice < 1000 THEN 'Under $1000'
       ELSE 'Over $1000'
     END
FROM Production.Product
ORDER BY ProductNumber ;
GO
```

Пример (CASE в предложении ORDER BY)

```
SELECT BusinessEntityID, SalariedFlag
FROM HumanResources.Employee
ORDER BY CASE SalariedFlag WHEN 1 THEN BusinessEntityID END DESC
      ,CASE WHEN SalariedFlag = 0 THEN BusinessEntityID END;
GO
```

GOTO

Переводит поток выполнения на метку. Инструкции языка Transact-SQL или инструкции, следующие за инструкцией GOTO, пропускаются, и выполнение продолжается с метки. Инструкция GOTO и метки могут быть включены в процедуру, пакет или блок инструкций. Инструкции GOTO могут быть вложенными.

Define the label:
label:
Alter the execution:
GOTO label

Аргументы

label
Точка, с которой начинается обработка инструкций после перехода на текущую метку с помощью инструкции GOTO. Метки должны соответствовать правилам для идентификаторов. Метка может применяться без инструкции GOTO как метод комментирования.

Замечания

Инструкция GOTO может существовать внутри условных инструкций, управляющих потоком, блоков инструкций или процедур, однако не может ссылаться на метку, расположенную вне этого пакета. Инструкция GOTO может ссылаться на метку, расположенную как до нее, так и после.

Пример

В следующем примере показано, как использовать разрешения GOTO в качестве механизма ветви.

```
DECLARE @Counter int;
SET @Counter = 1;
WHILE @Counter < 10
BEGIN
   SELECT @Counter
   SET @Counter = @Counter + 1
   IF @Counter = 4 GOTO Branch_One --Jumps to the first branch.
   IF @Counter = 5 GOTO Branch_Two  --This will never execute.
END
Branch_One:
   SELECT 'Jumping To Branch One.'
   GOTO Branch_Three; --This will prevent Branch_Two from executing.
Branch_Two:
   SELECT 'Jumping To Branch Two.'
Branch_Three:
   SELECT 'Jumping To Branch Three.';
```

WAITFOR

Блокирует выполнение пакета, хранимой процедуры или транзакции до наступления указанного времени или интервала времени, либо заданная инструкция изменяет или возвращает, по крайней мере, одну строку. Во время выполнения инструкции WAITFOR выполняется транзакция, и другие запросы не могут быть выполнены в рамках этой транзакции.

Синтаксис

```
WAITFOR
{
   DELAY 'time_to_pass'
  | TIME 'time_to_execute'


}
```

Аргументы

DELAY
Заданный период времени (не более 24 часов), который должен пройти до выполнения пакета, хранимой процедуры или продолжения транзакции.

"*time_to_pass*"
Период времени ожидания. *time_to_pass* может быть указан в одном из допустимых форматов для **datetime** данных или его можно указать в локальной переменной. Невозможно указать даты; Таким образом, часть дата типа **datetime** недопустима.

TIME
Заданное время выполнения пакета, хранимой процедуры или транзакции.

"*time_to_execute*"
Время, в которое инструкция WAITFOR завершает работу. *time_to_execute* может быть указан в одном из допустимых форматов для **datetime** данных или его можно указать в локальной переменной. Невозможно указать даты; Таким образом, часть дата типа **datetime** недопустима.

Примеры

А. Использование инструкции WAITFOR TIME

В следующем примере выполняется хранимая процедура sp_update_job в базе данных msdb в 22:20. (22:20).

```
EXECUTE sp_add_job @job_name = 'TestJob';
BEGIN
   WAITFOR TIME '22:20';
   EXECUTE sp_update_job @job_name = 'TestJob',
      @new_name = 'UpdatedJob';
END;
GO
```

Б. Использование WAITFOR DELAY

В следующем примере хранимая процедура выполняется после 2-часовой задержки.

```
BEGIN
   WAITFOR DELAY '02:00';
   EXECUTE sp_helpdb;
END;
GO
```

Хранимые процедуры

Хранимые процедуры представляют, по существу, предварительно откомпилированные программы, которые *хранятся на узле сервера* (и известны серверу). Клиент обращается к хранимой процедуре с помощью механизма вызова удаленных процедур (Remote Procedure Call — RPC). Поэтому, в частности, потери в производительности, связанные с обработкой данных на уровне записей в системе "клиент/сервер", могут быть частично компенсированы за счет создания подходящих хранимых процедур, позволяющих выполнить обработку данных непосредственно на узле сервера.
Другие преимущества хранимых процедур приведены ниже.

- Хранимые процедуры позволяют скрыть от пользователя множество специфических особенностей СУБД и базы данных и благодаря этому достичь более высокой степени независимости от данных по сравнению с тем случаем, когда хранимыепроцедуры не используются.
- Одна хранимая процедура может совместно использоваться многими клиентами.
- Оптимизация может быть осуществлена при создании хранимой процедуры, а не во время выполнения. (Безусловно, это преимущество проявляется лишь в тех системах, в которых оптимизация обычно осуществляется во время выполнения.)

- Хранимые процедуры позволяют обеспечить более высокую степень безопасности данных. Например, некоторому пользователю может быть разрешено вызывать определенную процедуру, но не разрешено непосредственно обрабатывать данные, к которым он может иметь доступ через эту хранимую процедуру.

Недостатком хранимых процедур является то, что поставщики программного обеспечения предоставляют в этой области слишком отличающиеся между собой средства, а расширение языка SQL для поддержки хранимых процедур появилось, к сожалению, лишь в 1996 году. Это средство называется SQL/PSM (Persistent Stored Module — постоянный хранимый модуль).

Рассмотрим синтаксис создания процедуры для Ьшскщыщае `SQL Server.`
`Обратите вминание , что в других СУБД он будет немного отличаться`

SQL Server Stored Procedure Syntax

```
CREATE { PROC | PROCEDURE } [schema_name.] procedure_name [ ; number ]
   [ { @parameter [ type_schema_name. ] data_type }
      [ VARYING ] [ = default ] [ OUT | OUTPUT | [READONLY]
   ] [ ,...n ]
[ WITH <procedure_option> [ ,...n ] ]
[ FOR REPLICATION ]
AS { [ BEGIN ] sql_statement [;] [ ...n ] [ END ] }
[;]

<procedure_option> ::=
   [ ENCRYPTION ]
   [ RECOMPILE ]
   [ EXECUTE AS Clause ]
```

schema_name

Имя схемы, которой принадлежит процедура. Процедуры привязаны к схеме. Если имя схемы не указано при создании процедуры, то автоматически назначается схема по умолчанию для пользователя, который создает процедуру.

procedure_name

Имя процедуры. Имена процедур должны соответствовать требованиям, предъявляемым к идентификаторам, и должны быть уникальными в схеме.

При задании имен для процедур не следует пользоваться префиксом **sp_**. Этим префиксом в SQL Server обозначаются системные процедуры. Использование этого префикса может нарушить работу кода приложения, если обнаружится системная процедура с таким же именем.

; number

Необязательный целочисленный аргумент, используемый для группирования одноименных процедур. Все сгруппированные процедуры можно удалить, выполнив одну инструкцию DROP PROCEDURE.

@ parameter

Параметр, объявленный в процедуре. Укажите имя параметра, начинающееся со знака **@**. Имя параметра должно соответствовать правилам для идентификаторов.

Параметры являются локальными в пределах процедуры; в разных процедурах могут быть использованы одинаковые имена параметров.

Можно объявить от 1 до 2100 параметров. При выполнении процедуры значение каждого из объявленных параметров должно быть указано пользователем, если для параметра не определено значение по умолчанию или значение не задано равным другому параметру.

Параметры не могут быть объявлены, если указан параметр FOR REPLICATION.

[type_schema_name**.**] data_type

Тип данных параметра и схема, к которой принадлежит этот тип.

VARYING

Указывает результирующий набор, поддерживаемый в качестве выходного параметра. Этот параметр динамически формируется процедурой, и его содержимое может различаться. Применяется только к параметрам типа cursor. Этот параметр недопустим для процедур CLR.

default

Значение по умолчанию для параметра. Если для некоторого параметра определено значение по умолчанию, то процедуру можно выполнить без указания значения этого параметра. Значение по умолчанию должно быть константой или может быть равно NULL. Значение константы может иметь вид шаблона, что позволяет использовать ключевое слово LIKE при передаче параметра в процедуру. См. пример B далее.

Значения по умолчанию записываются в столбец sys.parameters.default только для процедур CLR. В случае параметров процедуры Transact-SQL этот столбец будет содержать значения NULL.

OUT | OUTPUT

Показывает, что параметр процедуры является выходным. Используйте параметры OUTPUT для возврата значений в вызвавший процедуру код. Параметры text, ntext и image не могут использоваться как параметры OUTPUT, если процедура не является процедурой CLR. Выходным параметром с ключевым словом OUTPUT может быть заполнитель курсора, если процедура не является процедурой CLR. Возвращающий табличное значение тип данных не может быть указан в качестве выходного параметра процедуры.

READONLY

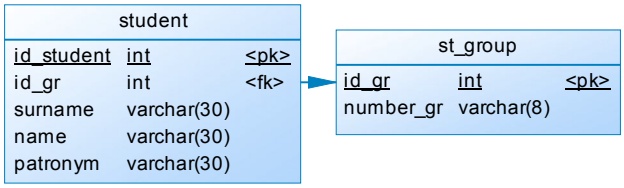
Указывает, что параметр не может быть обновлен или изменен в тексте процедуры. Если тип параметра является возвращающим табличное значение типом, то должно быть указано ключевое слово READONLY.

RECOMPILE

Показывает, что компонент Компонент Database Engine не кэширует план запроса для этой процедуры, что вызывает ее компиляцию при каждом выполнении. Дополнительные сведения о причинах принудительной повторной компиляции см. в разделе Перекомпиляция хранимой процедуры. Этот параметр нельзя использовать, если указано предложение FOR REPLICATION, а также для процедур CLR.

Чтобы Компонент Database Engine удалил планы отдельных запросов в процедуре, следует использовать указание запроса RECOMPILE в определении запроса.
FOR REPLICATION
Указывает, что процедура создается для репликации. Следовательно, ее нельзя выполнять на подписчике. Процедура, созданная с параметром FOR REPLICATION, используется в качестве фильтра и выполняется только в процессе репликации. Параметры не могут быть объявлены, если указан параметр FOR REPLICATION.
Синтаксис вызова процедуры
[{ exec | execute } [@<имя переменной> =] <имя процедуры> [; <номер>]
[@<имя параметра 1>] = { <значение параметра 1> |
@<имя переменной 1>] [output] }
[, …]
[with recompile]
@<имя переменной> — имя переменной, которой будут присвоен код возврата (ошиб-ки), возвращаемый процедурой;
<значение параметра 1> — выражение, задающее значение для параметра процедуры;
@<имя переменной 1> — значение параметра задается переменной.

Пример реализации добавления с пополнением справочника с возвратом идентификатора студента
Представим себе таблицы группа и студент.



В данном случае справочником будет независимая (родительская таблица группа (st_group), а добавлять будем в зависимую таблицу студент student)

```
Create procedure my_add_student  @surname varchar(30), @name varchar(30), @patronym varchar(30),@number_group varchar(8), @id_st int out
as
declare @id_gr int
if exists (select id_gr from st_group where number_gr=@number_group)
begin
set @id_gr= select id_gr from st_group where number_gr=@number_group
end
else
begin
set @id_gr= select isnull(max(id_gr)+1,0) from st_group
insert into st_group (id_gr,number_gr) values (@id_gr,@number_group)
end
set @id_st= select isnull(max(id_student)+1,0) from student
insert into student (id_student, id_gr, surname, name, patronym) values (@id_st,@id_gr, @surname,@name, @patronym)
go
```

Isnull (x,y)- функция, которая проверяет первый аргумент (x) на то не пустой ли он, и если не пустой, то возвращает x, а если пустой, то возвращает y. Это запись актуальна для вставки в пустую таблицу.
Пример вызова

declare @id_st int
Exec my_add_student ‘Иванов’, ‘Иван’, ‘Иванович’, ‘Z4432K’, @id_st out
Select @id_st

Представления (VIEW)

Приведем определение представления
Представление.- Динамически сформированный результат одной или нескольких реляционных операций, выполненных над отношениями базы данных с целью получения нового отношения. Представление является виртуальным отношением, которое не всегда реально существует в базе данных, но создается позапросу определенного пользователя в ходе выполнения этого запроса.
С точки зрения пользователя базы данных представление выглядит как реальная таблица данных, содержащая набор поименованных столбцов и строк данных. Но в отличие от реальных таблиц представления не всегда существуют в базе как некоторый набор сохраняемых значений данных. В действительности доступные через представления строки и столбцы данных являются результатом выполнения запроса, заданного при определении представления. СУБД сохраняет определение представления в базе данных. Обнаружив ссылку на представление, СУБД применяет один из двух следующих подходов для формирования представления.
При первом подходе СУБД отыскивает определение представления и преобразуют исходный запрос, лежащий в основе представления, в эквивалентный запрос к таблицам, использованным в определении представления, после чего модифицированный запрос выполняется. Этот процесс слияния запросов, называемый *заменой представления* (под этим подразумевается замена представления оператором SQL, который обращается к базовым таблицам).
При втором подходе, который называется *материализацией представления,* готовое представление хранится в базе данных в виде временной таблицы, а его актуальность постоянно поддерживается по мере обновления всех таблиц, лежащих в его основе.

Создание представлений (оператор CREATE VIEW)

Оператор CREATE VIEW имеет следующий формат:

CREATE VIEW *ViewName* [(newColumnName [, . . .])] .
AS *subselect* [WITH [CASCADED | LOCAL] CHECK OPTION]

Представление определяется с помощью подзапроса subselect, оформленного в виде оператора SELECT языка SQL. Существует необязательная возможность присвоения собственного имени каждому из столбцов представления. Если указывается список имен столбцов, то он должен иметь количество элементов, равное количеству столбцов в результирующей таблице запроса, заданного параметром subselect. Если список имен столбцов опущен, каждый столбец представления будет иметь имя соответствующего столбца результирующей таблицы запроса, заданного параметром *subselect.* Список имен столбцов должен обязательно задаваться в том случае, если в именах столбцов результирующей таблицы имеет место неоднозначность. Подобная ситуация возникает в тех случаях, когда подзапрос включает вычисляемые поля, а конструкция AS с именами столбцов результирующей таблицы не содержит для них имен или же когда

результирующая таблица создается с помощью операции соединения и включает столбцы с одинаковыми именами .
Заданный параметром subselect подзапрос принято называть *определяющим запросом.*
Если указана конструкция WITH CHECK OPTION, то гарантируется, что в тех случаях, когда строка данных не удовлетворяет условию, указанному в конструкции WHERE определяющего запроса представления, она не будет добавлена в его базовую таблицу
Представления могут быть горизонтальными и вертикальными.
Горизонтальное представление позволяет ограничить доступ пользователей только определенными строками, выбранными из одной или нескольких таблиц.
Вертикальные представления позволяют ограничить доступ пользователей только определенными столбцами, выбранными из одной или нескольких таблиц базы данных.
Приведем пример горизонтального представления:
Например куратор группы Z4432K может работать только с данными студентовэтой группы

```
CREATE VIEW gr_z4432K
AS
select student.* from student, st_group where student.id_gr =st_group.id_gr and
number_gr='Z4432K'
```

А теперь допустим, что куратору не следует давать доступ к идентификаторам.
Таким образом, это будет вертикальное представление

```
CREATE VIEW gr_z4432K_1
AS
select surname, name, patronym from student, st_group where student.id_gr =st_group.id_gr and
number_gr='Z4432K'
```

Представления могут иметь и более сложный вид с , когда в запросе встречаются подзапросы, агрегатные функции, группировки и т.п.

Удаление представлений (оператор DROP VIEW)

Представление удаляется из базы данных с помощью оператора DROP VIEW, имеющего следующий формат:
DROP VIEW *ViewName* [RESTRICT | CASCADE]
При выполнении оператора DROP VIEW определение представления удаляется из базы данных. Например, представление gr_z4432K_1 можно удалить с помощью следующего оператора:

```
DROP VIEW gr_z4432K_1;
```

Если указано ключевое слово CASCADE, при выполнении оператора DROP VIEW будут также удалены все связанные с ним или зависящие от него объекты. Другими словами, будут удалены все объекты, содержащие ссылки на удаляемое представление. Это означает, что такой оператор DROP VIEW удаляет также все представления, которые определены на основе удаляемого представления. Если указывается ключевое слово RESTRICT и существуют любые прочие объекты, зависящие от существования удаляемого представления, выполнение этого оператора блокируется. По умолчанию принимается значение RESTRICT.

Обновление данных в представлениях
Все обновления, выполненные в таблице базы данных, немедленно отражаются на всех представлениях, предусматривающих обращения к этой таблице. Подобным же образом, можно ожидать, что если данные были изменены в представлении, то это изменение будет отображено и в той таблице (таблицах) базы данных, на которой оно построено.

В стандарте ISO указано, что представления могут быть обновляемыми только в тех системах, которые отвечают некоторым четко определенным требованиям. Согласно стандарту ISO представление может быть обновляемым только в том случае, если:
• в его определении не используется ключевое слово DISTINCT, т.е. из результатов определяющего запроса не исключаются повторяющиеся строки;
• каждый элемент в списке конструкции SELECT определяющего запроса представляет собой имя столбца (а не константу, выражение или агрегирующую функцию), причем имя каждого из столбцов упоминается в этом списке не более одного раза;
• в конструкции FROM указана только одна таблица, т.е. представление должно быть создано на базе единственной таблицы, по отношению к которой пользователь должен обладать необходимыми правами доступа. Если исходная таблица сама является представлением, то это представление также должно отвечать указанным условиям. Данное требование исключает возможность обновления любых представлений, построенных на базе соединения, объединения (UNION), пересечения (INTERSECT) или разности (EXCEPT) таблиц;
• в конструкцию WHERE не входят какие-либо вложенные запросы типа SELECT, которые ссылаются на таблицу, указанную в конструкции FROM;
• определяющий запрос не должен содержать конструкций GROUP BY и HAVING.
Кроме того, любая строка данных, добавляемая с помощью представления, не должна нарушать требования поддержки целостности данных, установленные для исходной таблицы представления. Например, при добавлении с помощью представления новой строки таблицы во все столбцы таблицы, отсутствующие в этом представлении, будут введены значения NULL. Однако при этом не должны нарушаться все требования NOT NULL, указанные в описании исходной таблицы представления. Основную концепцию, используемую при формулировании обсуждаемых ограничений, можно выразить с помощью следующего определения.
*Обновляемое представление***.** Для того чтобы представление было обновляемым, СУБД должна иметь возможность однозначно отобразить любую его строку - или столбец на соответствующую строку или столбец его исходной таблицы.

Использование конструкции WITH CHECK OPTION

В представление помещаются только те строки, которые удовлетворяют условию WHERE в определяющем запросе. Если строка в представлении будет изменена таким образом, что перестанет удовлетворять этому условию, то эта строка должна исчезнуть из данного представления. Аналогичным образом, в представлении будут появляться новые строки всякий раз, когда вставка или обновление данных в представлении приведет к тому, что новые строки будут удовлетворять условию WHERE. Строки, которые добавляются или исключаются из представления в зависимости от содержащихся в них данных, принято называть *мигрирующими.*
В общем случае для предотвращения миграции строк представления используется конструкция WITH CHECK OPTION оператора CREATE VIEW. Необязательные ключевые слова LOCAL/CASCADED применимы в случае существования иерархии представлений, т.е. когда представление создается на базе другого представления. Если указана конструкция WITH LOCAL CHECK OPTION, то такие действия, как вставка или обновление данных в некотором представлении, на базе которого прямо или косвенно определены другие представления, не могут
вызвать исчезновение строки из данного представления, за исключением случаев, когда данная строка исчезает также из представлений или таблиц других уровней иерархии. При указании конструкции WITH CASCADED CHECK OPTION
(это значение принимается по умолчанию) в случае вставки или обновления строки в данном представлении или в любом другом представлении, прямо или косвенно определенном на базе данного, исчезновение этой строки из данного

представления не допускается. Такая функция может оказаться настолько полезной, что работать с представлениями окажется удобнее, чем с таблицами базы данных. В том случае, когда оператор INSERT или UPDATE нарушает условия, указанные в конструкции WHERE определяющего запроса, выполнение затребованной операции отменяется.
В результате появляется возможность реализовать в базе данных дополнительные ограничения, направленные на сохранение целостности данных. Следует отметить, что конструкцию WITH CHECK OPTION можно указывать только для обновляемых представлений.

Основные преимущества и недостатки представлений языка SQL

Преимущества

- Независимость от данных
- Актуальность
- Повышение защищенности данных
- Снижение сложности
- Дополнительные удобства
- Возможность настройки
- Обеспечение целостности данных

Недостатки

- Ограниченные возможности обновления
- Структурные ограничения
- Снижение производительности

Тема 1

Тема 2.4.Триггеры, обеспечение активной целостности.

Особый интерес представляет декларативная поддержка целостности. И хотя ситуация за последние годы улучшилась, остается фактом, что лишь немногие продукты (если они вообще есть) обеспечивают такую поддержку со времени своего первоначального появления на рынке. Вследствие этого ограничения целостности чаще всего реализуются процедурное, с использованием триггерных процедур. Последние представляют собой заранее откомпилированные процедуры, которые хранятся вместе с базой данных (возможно, в самой базе данных) и вызываются автоматически при возникновении некоторых указанных событий.

Но необходимо сразу же привести следующие замечания.

1. Именно потому, что они являются процедурами, триггерные процедуры нельзя считать рекомендуемым способом реализации ограничений целостности. Пользователям сложнее понять, как действуют процедуры, а для системы процедуры создают дополнительные трудности при оптимизации. Следует также отметить, что декларативные ограничения проверяются при всех соответствующих обновлениях, а триггерные процедуры выполняются только при возникновении указанного события
2. Область применения триггерных процедур не ограничивается задачами поддержки целостности, которые являются темой настоящей главы. Вместо этого они могут выполнять другие полезные задачи и именно поэтому имеют право на существование. Некоторые примеры таких "других полезных задач" приведены ниже:

а) Передача пользователю предупреждения о том, что возникло некоторое ис ключение (например, выдача предупреждающего сообщения, если наличное количество некоторых деталей на складе становится ниже критического уровня).
б) Отладка (т.е. отслеживание ссылок на указанные переменные и/или контроль над изменениями состояния этих переменных).
в) Аудит (например, регистрация информации о том, кто и когда внес те или иные изменения в определенные переменные отношения).
г) Измерение производительности (например, регистрация времени наступления или трассировка указанных событий в базе данных).
д) Проведение компенсирующих действий (например, каскадная организация удаления кортежа поставщика для удаления также соответствующих кортежей поставок).
Рассмотрим общие свойства оператора создания триггера
В общем, в операторе создания триггера CREATE TRIGGER, кроме всего прочего, определены событие, условие и действие следующим образом:
■ **событием** является операция в базе;
■ **условие—** это логическое выражение, которое должно принимать значение TRUE для того, чтобы было выполнено действие (если условие не указано явно, как в данном примере, то оно по умолчанию равно nросто TRUE);
■ **действие** — это и есть сама триггерная процедура
Событие и условие вместе иногда называют *триггерным событием,* а сочетание всех трех компонентов (событие, условие и действие) обычно называют просто **триггером.** По очевидным причинам триггеры именуются также правилами "событие-условие- действие" (Event-Condition-Action — ECA) или сокращенно правилами ECA.
2. К возможным событиям относится выполнение операций INSERT, DELETE, UPDATE (возможно, над определенными атрибутами), достижение конца транзакций (COMMIT), наступление указанного времени суток, истечение определенного интервала времени, нарушение указанного ограничения и т.д.
3. В общем, действие может выполняться до (BEFORE), после (AFTER) или вместо (INSTEAD OF) действия, обусловленного указанным событием, при условии, что эти варианты имеют смысл.
4. В общем, действие может выполняться для каждой строки (FOR EACH ROW) или для каждого оператора (FOR EACH STATEMENT), при условии, что эти варианты имеют смысл.
5.В общем, при выполнении действия, определяемого триггером, должен быть предусмотрен способ, позволяющий ссылаться на данные в том виде, какой они имеют до и после возникновения указанного события, при условии, что применениетакого средства действительно имеет смысл.
6. Базу данных, которая имеет связанные с ней триггеры, иногда называют *активнойбазой данных.*
Оператор CREATE TRIGGER языка SQL выглядит следующим образом:

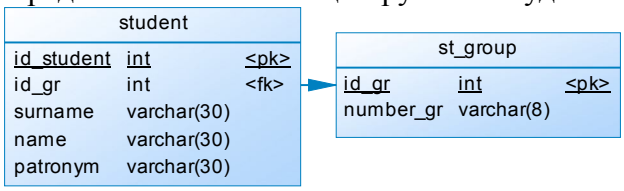
CREATE TRIGGER < *trigger name*>
<*before or after*> <*event*> ON <*base table name*>
[REFERENCING <*naming commalist*> *]*
*[*FOR EACH <*row or statement*> *]*
[WHEN (<*bool exp*> *)]* <*action*> ;

Пояснения к этому оператору приведены ниже.
1. Спецификация с указанием времени проверки до или после активизации триггера, <*before or after*>*,* принимает значение BEFORE или AFTER (в стандарте SQL не поддерживается ключевое слово INSTEAD OF, но в некоторых программных продуктах такая поддержка предусмотрена).

2. Событие <*event*> может принимать значение INSERT, DELETE или UPDATE. Зна чение UPDATE может дополнительно уточняться с помощью спецификации OF <*column name commalist*>.
3. Каждое определение именования <*naming*> может принимать одну из следующих форм.
OLD ROW AS <*name*>
NEW ROW AS <*name*>
OLD TABLE AS <*name*>
NEW TABLE AS <*name*>
4. Спецификация с определением строки или оператора <*row or statement*> принимает значение ROW или STATEMENT (STATEMENT применяется по умолчанию). Ключевое слово ROW означает, что триггер активизируется для каждой от дельной строки, на которую распространяется действие триггерного оператора, а STATEMENT означает, что триггер активизируется только один раз для данного оператора, рассматриваемого как единое целое.
5. Если определена конструкция WHEN, она означает, что действие <*action*> должно выполняться, только если логическое выражение <*bool exp*> принимает значение TRUE.
6. Спецификация <*action*> задает отдельный оператор SQL (но этот отдельный оператор может быть достаточно сложным, т.е. составным, а это неформально означает, что такой оператор может состоять из последовательности операторов, обозначенных разграничителями BEGIN и END).

Однако необходимо обратить внимание, при написании триггеров и хранимых процедур в MS SQL Server, что он считает концом триггера или хранимой процедуры только оператор go

Для доступа к удаленным значения в MS SQL Server используется временная таблица DELETED (В MySQL OLD)
Для доступа к добавленным значения в MS SQL Server используется временная таблица INSERTED (В MySQL NEW)
При обновлении в DELETED старые значения, а в INSERTED – новые.
Пример реализации каскадного удаления
Представим себе таблицы группа и студент




```
Create trigger my_trigger
 instead of delete on st_group
as
begin
delete * from student where id_gr in (select id_gr from deleted)
delete * from st_group where id_gr in (select id_gr from deleted)
end
go
```

Наконец, ниже показан синтаксис оператора DROP TRIGGER.
DROP TRIGGER <*trigger name*> ;

Использованная литература

1. Дейт, К. Дж. Введение в системы баз данных, 8-е издание.: Пер. с англ. — М.: Издательский дом "Вильямс", 2005. — 1328 с.

2. К. Дж. Дейт SQL и реляционная теория. Как грамотно писать код на SQL. – Пер. с англ. –СПб.: Символ-Плюс, 2010. – 480 с.

3. Хомоненко, А. Д. Базы данных [Текст] : учебник для высших учебных заведений / А. Д. Хомоненко, В. М. Цыганков, М. Г. Мальцев ; ред. А. Д. Хомоненко. - 6-е изд., доп. и перераб. - СПб. : КОРОНА-Век, 2010. - 736 с.
4. Фаулер, Мартин, Садаладж, Прамодкумар Дж. NoSQL: новая методология разработки нереляционных баз данных. : Пер. сангл. - М.: ООО "И.Д. Вильяме", 2013. - 192 с.
5. Коннолли, Томас, Бегг, Каролин. Базы данных. Проектирование, реализация и сопровождение. Теория и практика. 3-е издание. : Пер. с англ. — М. : Издательский дом "Вильямс",2003. — 1440 с.
6. MYSQL:: Developer Zone https://dev.mysql.com/ (дата последнего доступа 02.12.2017)
7. История языков программирования: SQL- стандартизация длиною в жизнь https://habrahabr.ru/post/311288/ (последний доступ 2.12.2017)

8. What's new in SQL:2016 http://modern-sql.com/blog/2017-06/whats-new-in-sql-2016 (последний доступ 2.12.2017)

9. What's new in SQL:2011 http://www.sigmod.org/publications/sigmod-record/1203/pdfs/10.industry.zemke.pdf (последний доступ 2.12.2017)

10. MERGE (Transact-SQL) https://msdn.microsoft.com/ru-ru/library/bb510625(v=sql.120).aspx (последний доступ 3.12.2017)
11. Реляционное деление| Интерактивный учебник по SQL | SQL Tutorial.ru http://www.sql-tutorial.ru/ru/book_relational_division.html (последний доступ 3.12.2017)
12. IF...ELSE (Transact-SQL) https://docs.microsoft.com/ru-ru/sql/t-sql/language-elements/if-else-transact-sql (последний доступ 4.12.2017)

13. BEGIN...END (Transact-SQL) https://docs.microsoft.com/ru-ru/sql/t-sql/language-elements/begin-end-transact-sql (последний доступ 4.12.2017)

14. WHILE (Transact-SQL) https://docs.microsoft.com/ru-ru/sql/t-sql/language-elements/while-transact-sql (последний доступ 4.12.2017)

15. Выражение CASE (Transact-SQL) https://docs.microsoft.com/ru-ru/sql/t-sql/language-elements/case-transact-sql (последний доступ 4.12.2017)

16. GOTO (Transact-SQL) https://docs.microsoft.com/ru-ru/sql/t-sql/language-elements/goto-transact-sql (последний доступ 4.12.2017)

17. WAITFOR (Transact-SQL) https://docs.microsoft.com/ru-ru/sql/t-sql/language-elements/waitfor-transact-sql (последний доступ 4.12.2017)

18.

Приложение

1 Словарь

Русский термин	Аббревиатура	Английский термин	Английская аббревиатура
База данных	БД	database	DB
Реляционная база данных	РБД	Relational database	RDB
Распределенная база данных	РДБ	Distributed database	DDB
Система управления базами данных	СУБД	Database management system	DBMS
Первичный ключ		Primary key	PK
Внешний ключ		Foreign key	FK

2 Backus Naur Form — форма Бэкуса-Наура) для записис грамматик.

Прописные буквы используются для записи зарезервированных слов и должны указываться в операторах точно так же, как это будет показано.

• **Строчные буквы** используются для записи слов, определяемых пользователем. (Иногда и угловые скобки <> используются с этой целью)

• Вертикальная черта (|) указывает на необходимость *выбора* одного из нескольких приведенных значений, например a | b | c.

• Фигурные скобки определяют *обязательный элемент,* например {a}.

• Квадратные скобки определяют *необязательный элемент,* например [a].

• Многоточие (...) используется для указания необязательной возможности повторения конструкции от нуля до нескольких раз, например {a b} [,c...]. Эта запись означает, что после a или b может следовать от нуля до нескольких повторений c, разделенных запятыми.

::= -состоит из/является для нелитеральных символов, например:

Символ::= <буква>|<цифра>|<спец символ>

Символ- либо буква либо цифра либо спец. символ