БАЗЫ ДАННЫХ

КОНСПЕКТ ЛЕКЦИЙ

ЛЕКЦИЯ № 1. Введение

1. Системы управления базами данных

Системы управления базами данных (СУБД) — это специализированные программные продукты, позволяющие:

1) постоянно хранить сколь угодно большие (но не бесконечные) объемы данных;

2) извлекать и изменять эти хранящиеся данные в том или ином аспекте, используя при этом так называемые запросы;

3) создавать новые базы данных, т. е. описывать логические структуры данных и задавать их структуру, т. е. предоставляют интерфейс программирования;

4) обращаться к хранящимся данным со стороны нескольких пользователей одновременно (т. е. предоставляют доступ к механизму управления транзакциями).

Соответственно, **базы данных** — это наборы данных, находящиеся под контролем систем управления.

Сейчас системы управления базами данных являются наиболее сложными программными продуктами на рынке и составляют его основу. В дальнейшем предполагается вести разработки по сочетанию обычных систем управления базами данных с объектно-ориентированным программированием (ООП) и интернет-технологиями.

Изначально СУБД были основаны на **иерархических** и **сетевых моделях данных**, т. е. позволяли работать только с древовидными и графовыми структурами. В процессе развития в 1970 г. появились системы управления базами данных, предложенные Коддом (Codd), основанные на **реляционной модели данных**.

2. Реляционные базы данных

Термин «реляционный» произошел от английского слова «relation» — «отношение».

В самом общем математическом смысле (как можно помнить из классического курса алгебры множеств) **отношение** — это множество

$R = \{(x_1, \ldots, x_n) \mid x_1 \in A_1, \ldots, x_n \in A_n\}$, где $A_1, \ldots, A_n$ — множества, образующие декартово произведение. Таким образом, **отношение R** — это подмножество декартова произведения множеств: $A_1 \times \ldots \times A_n$:

$R \subseteq A_1 \times \ldots \times A_n$.

Например, рассмотрим бинарные отношения строгого порядка «больше» и «меньше» на множестве упорядоченных пар чисел $A_1 = A_2 = \{3, 4, 5\}$:

$R_> = \{(3, 4), (4, 5), (3, 5)\} \subset A_1 \times A_2$;

$R_< = \{(5, 4), (4, 3), (5, 3)\} \subset A_1 \times A_2$.

Эти же отношения можно представить в виде таблиц.

Отношение «больше» $R_>$:

5	4
4	3
5	3

Отношение «меньше» $R_<$:

3	4
4	5
3	5

Таким образом, мы видим, что в реляционных базах данных самые различные данные организовываются в виде отношений и могут быть представлены в форме таблиц.

Нужно заметить, что эти два рассмотренных нами отношения $R_>$ и $R_<$ не эквивалентны между собой, другими словами, таблицы, соответствующие этим отношениям, не равны друг другу.

Итак, формы представления данных в реляционных БД могут быть разными. В чем проявляется эта возможность различного представления в нашем случае? Отношения $R_>$ и $R_<$ — это множества, а множество — структура неупорядоченная, значит, в таблицах, соответствующих этим отношениям, строки можно менять между собой местами. Но в то же время элементы этих множеств — это упорядоченные наборы, в нашем случае — упорядоченные пары чисел 3, 4, 5, значит, столбцы менять местами нельзя. Таким образом, мы показали, что представление отношения (в математическом смысле) в виде таблицы с произвольным порядком строк и фиксированным числом столбцов является приемлемой, правильной формой представления отношений.

Но если рассматривать отношения $R_>$ и $R_<$ с точки зрения заложенной в них информации, то понятно, что они эквивалентны. Поэтому

в реляционных базах данных понятие «отношение» имеет несколько другой смысл, нежели отношение в общей математике. А именно оно не связано с упорядоченностью по столбцам в табличной форме представления. Вместо этого вводятся так называемые схемы отношений «строка — заголовок столбцов», т. е. каждому столбцу дается заголовок, после чего их можно беспрепятственно менять местами.

Вот как будут выглядеть наши отношения $R_>$ и $R_<$ в реляционной базе данных.

Отношение строгого порядка (вместо отношения $R_>$):

Оценка большая	**Оценка меньшая**
5	4
4	3
5	3

Отношение строгого порядка (вместо отношения $R_<$):

Оценка большая	**Оценка меньшая**
3	4
4	5
3	5

Обе таблицы-отношения получают новое (в данном случае одинаковое, так как введением дополнительных заголовков мы стерли различия между отношениями $R_>$ и $R_<$) название.

Итак, мы видим, что при помощи такого несложного приема, как дополнение таблиц необходимыми заголовками, мы приходим к тому, что отношения $R_>$ и $R_<$ становятся эквивалентными друг другу.

Таким образом, делаем вывод, что понятие «отношение» в общем математическом и в реляционном смысле совпадают не полностью, не являются тождественными.

В настоящее время реляционные системы управления базами данных составляют основу рынка информационных технологий. Дальнейшие исследования ведутся в направлении сочетания той или иной степени реляционной модели.

ЛЕКЦИЯ № 2. Отсутствующие данные

В системах управления базами данных для определения отсутствующих данных описаны два вида значений: пустые (или Empty-значения) и неопределенные (или Null-значения).

В некоторой (преимущественно коммерческой) литературе на Null-значения иногда ссылаются как на пустые или нулевые значения, однако это неверно. Смысл пустого и неопределенного значения принципиально различается, поэтому необходимо внимательно следить за контекстом употребления того или иного термина.

1. Пустые значения (Empty-значения)

Пустое значение — это просто одно из множества возможных значений какого-то вполне определенного типа данных.

Перечислим наиболее «естественные», непосредственные **пустые значения** (т. е. пустые значения, которые мы могли бы выделить самостоятельно, не имея никакой дополнительной информации):

1) 0 (нуль) — нулевое значение является пустым для числовых типов данных;
2) false (неверно) — является пустым значением для логического типа данных;
3) B'' — пустая строка бит для строк переменной длины;
4) "" — пустая строка для строк символов переменной длины.

В приведенных выше случаях определить, пустое значение или нет, можно путем сравнивания имеющегося значения с константой пустого значения, определенной для каждого типа данных. Но системы управления базами данных в силу реализованных в них схем долговременного хранения данных могут работать только со строками постоянной длины. Из-за этого пустой строкой бит можно назвать строку двоичных нулей. Или строку, состоящую из пробелов или каких-либо других управляющих символов, — пустой строкой символов.

Вот несколько примеров пустых строк постоянной длины:

1) B'0';
2) B'000';
3) ' '.

Как же в этих случаях определить, является ли строка пустой?

В системах управления базами данных для проверки на пустоту применяется логическая функция, т. е. предикат **IsEmpty (<выражение>)**, что буквально означает «есть пустой». Этот предикат обычно встроен в систему управления базами данных и может применяться к выражению абсолютно любого типа. Если такого предиката в системах управления базами данных нет, то можно написать логическую функцию самим и включить ее в список объектов проектируемой базы данных.

Рассмотрим еще один пример, когда не так просто определить,пустое ли мы имеем значение. Данные типа «дата». Какое значение в этом типе считать пустым значением, если дата может варьироваться в диапазоне от 01.01.0100. до 31.12.9999? Для этого в СУБД вводится специальное обозначение для **константы пустой даты {...}**, если значения этого типа записывается: {ДД. ММ. ГГ} или {ГГ. ММ. ДД}. С этим значением и происходит сравнение при проверке значения на пустоту. Оно считается вполне определенным, «полноправным» значением выражения этого типа, причем наименьшим из возможных.

При работе с базами данных пустые значения часто используются как значения по умолчанию или применяются, если значения выражений отсутствуют.

2. Неопределенные значения (Null-значения)

Слово **Null** используется для обозначения **неопределенных значений** в базах данных.

Чтобы лучше понять, какие значения понимаются под неопределенными, рассмотрим таблицу, являющуюся фрагментом базы данных:

№	**Фамилия**	**Год рождения**	**№ паспорта**
1	Хайретдинов	1980	Null
2	Карамазов	2000	Null
3	Коваленко	Null	Null

Итак, **неопределенное значение** или **Null-значение** — это:

1) неизвестное, но обычное, т. е. применимое значение. Например, у господина Хайретдинова, который является номером один в нашей базе данных, несомненно, имеются какие-то паспортные данные (как у человека 1980 г. рождения и гражданина страны), но они не известны, следовательно, не занесены в базу данных. Поэтому в соответствующую графу таблицы будет записано значение Null;

2) неприменимое значение. У господина Карамазова (№ 2 в нашей базе данных) просто не может быть никаких паспортных дан-

ных, потому что на момент создания этой базы данных или внесения в нее данных, он являлся ребенком;

3) значение любой ячейки таблицы, если мы не можем сказать применимое оно или нет. Например, у господина Коваленко, который занимает третью позицию в составленной нами базе данных, неизвестен год рождения, поэтому мы не можем с уверенностью говорить о наличие или отсутствии у него паспортных данных. А следовательно, значениями двух ячеек в строке, посвященной господину Коваленко будет Null-значение (первое — как неизвестное вообще, второе — как значение, природа которого неизвестна).

Как и любые другие типы данных, Null-значения тоже имеют определенные **свойства**. Перечислим самые существенные из них:

1) с течением времени понимание Null-значения может меняться. Например, у господина Карамазова (№ 2 в нашей базе данных) в 2014 г., т. е. по достижении совершеннолетия, Null-значение изменится на какое-то конкретное вполне определенное значение;

2) Null-значение может быть присвоено переменной или константе любого типа (числового, строкового, логического, дате, времени и т. д.);

3) результатом любых операций над выражениями с Null-значениями в качестве операндов является Null-значение;

4) исключением из предыдущего правила являются операции конъюнкции и дизъюнкции в условиях законов поглощения (подробнее о законах поглощения смотрите в п. 4 лекции № 2).

3. Null-значения и общее правило вычисления выражений

Поговорим подробнее о действиях над выражениями, содержащими Null-значения.

Общее правило работы с Null-значениями (то, что результат операций над Null-значениями есть Null-значение) применяется к следующим операциям:

1) к арифметическим;

2) к побитным операциям отрицания, конъюнкции и дизъюнкции (кроме законов поглощения);

3) к операциям со строками (например, конкотинации — сцепления строк);

4) к операциям сравнения ($<$, $\leq$, $=$, $\neq$, $\geq$, $>$).

Приведем примеры. В результате применений следующих операций будут получены Null-значения:

$$(3 + \text{Null}),\ \frac{1}{\text{Null}},\ (\text{Иванов}' + '' + \text{Null}) := \text{Null}$$

Здесь вместо обычного равенства использована **операция подстановки** «:=» из-за особого характера работы с Null-значениями. Далее в подобных ситуациях также будет использоваться этот символ, который означает, что выражение справа от символа подстановки может заменить собой любое выражение из списка слева от символа подстановки.

Характер Null-значений приводит к тому, что часто в некоторых выражениях вместо ожидаемого нуля получается Null-значение, например:

$$(x - x),\ y * (x - x),\ x * 0 := \text{Null при } x = \text{Null}.$$

Все дело в том, что при подстановке, например, в выражение $(x - x)$ значения $x = \text{Null}$, мы получаем выражение $(\text{Null} - \text{Null})$, и в силу вступает общее правило вычисления значения выражения, содержащего Null-значения, и информация о том, что здесь Null-значение соответствует одной и той же переменной теряется.

Можно сделать вывод, что при вычислении любых операций, кроме логических, Null-значения интерпретируются как **неприменимые**, и поэтому в результате получается тоже Null-значение.

К не менее неожиданным результатам приводит использование Null-значений в операциях сравнения. Например, в следующих выражениях также получаются Null-значения вместо ожидаемых логических значений True или False:

$(\text{Null} < \text{Null})$; $(\text{Null} \le \text{Null})$; $(\text{Null} = \text{Null})$; $(\text{Null} \ne \text{Null})$;

$(\text{Null} > \text{Null})$; $(\text{Null} \ge \text{Null}) := \text{Null}$;

Таким образом, делаем вывод, что нельзя говорить о том, что Null-значение равно или не равно самому себе. Каждое новое вхождение Null-значения рассматривается как независимое, и каждый раз Null-значения воспринимаются как различные неизвестные значения. Этим Null-значения кардинально отличаются от всех остальных типов данных, ведь мы знаем, что обо всех пройденных ранее величинах и их типах с уверенностью можно было говорить, что они равны или не равны друг другу.

Итак, мы видим, что Null-значения не являются значениями переменных в обычном смысле этого слова. Поэтому становится невоз-

можным сравнивать значения переменных или выражения, содержащие Null-значения, поскольку в результате мы будем получать не логические значения True или False, а Null-значения, как в следующих примерах:

(x < Null); (x ≤ Null); (x = Null); (x ≠ Null); (x > Null); (x ≥ Null) := Null;

Поэтому по аналогии с пустыми значениями для проверки выражения на Null-значения необходимо использовать специальный предикат:

IsNull (<выражение>), что буквально означает «есть Null».

Логическая функция возвращает значение True, если в выражении присутствует Null или оно равно Null, и False — в противном случае, но никогда не возвращает значение Null. Предикат IsNull может применяться к переменным и выражению любого типа. Если применять его к выражениям пустого типа, предикат всегда будет возвращать False.

Например:

IsNull(0)	False
IsNull(x + 'abc' + Null)	True
IsNull(2 * Null)	True
IsNull(Null)	True

Итак, действительно, видим, что в первом случае, когда предикат IsNull взяли от нуля, на выходе получилось значение False. Во всех случаях, в том числе во втором и третьем, когда аргументы логической функции оказались равными Null-значению, и в четвертом случае, когда сам аргумент и был изначально равен Null-значению, предикат выдал значение True.

4. Null-значения и логические операции

Обычно в системах управления базами данных непосредственно поддерживаются только три логические операции: отрицание $\neg$, конъюнкция & и дизъюнкция $\vee$. Операции следования $\Rightarrow$ и равносильности $\Leftrightarrow$ выражаются через них с помощью подстановок:

$(x \Rightarrow y) := (\neg x \vee y)$;

$(x \Leftrightarrow y) := (x \Rightarrow y)\ \&\ (y \Rightarrow x)$;

Заметим, что эти подстановки полностью сохраняются и при использовании Null-значений.

Интересно, что при помощи операции отрицания « $\neg$ » любая из операций конъюнкция & или дизъюнкция $\vee$ может быть выражена одна через другую следующим образом:

$(x \,\&\, y) := \neg\ (\neg\ x \vee \neg\ y)$;

$(x \vee y) := \neg\ (\neg\ x \,\&\, \neg\ y)$;

На эти подстановки, как и на предыдущие, Null-значения влияния не оказывают.

А теперь приведем таблицы истинности логических операций отрицания, конъюнкции и дизъюнкции, но кроме привычных значений True и False, используем также Null-значение в качестве операндов. Для удобства введем следующие обозначения: вместо True будем писать t, вместо False — f, а вместо Null — n.

1. **Отрицание** $\neg$ x.

x	$\neg\ x$
f	t
n	n
t	f

Стоит отметить следующие интересные моменты касательно операции отрицания с использованием Null-значений:

1) $\neg\ \neg$ x := x — закон двойного отрицания;

2) $\neg$ Null := Null — Null-значение является неподвижной точкой.

2. **Конъюнкция *x & y***.

y / ***x***	**f**	**n**	**t**
f	f	f	f
n	f	n	n
t	f	n	t

Эта операция также имеет свои свойства:

1) $x \,\&\, y := y \,\&\, x$ — коммутативность;

2) x & x := x — идемпотентность;

3) False & y := False, здесь False — поглощающий элемент;

4) True & $y := y$, здесь True — нейтральный элемент.

3. **Дизъюнкция $x \vee y$.**

y / x	f	n	t
f	f	n	t
n	n	n	t
t	t	t	t

Свойства:

1) $x \vee y := y \vee x$ — коммутативность;
2) $x \vee x := x$ — идемпотентность;
3) False $\vee$ $y := y$, здесь False — нейтральный элемент;
4) True $\vee$ y := True, здесь True — поглощающий элемент.

Исключение из общего правила составляют правила вычисления логических операций конъюнкция & и дизъюнкция $\vee$ в условиях действия **законов поглощения**:

(False & y) := (x & False) := False;

(True $\vee$ y) := ($x \vee$ True) := True;

Эти дополнительные правила формулируются для того, чтобы при замене Null-значения значениями False или True результат бы все равно не зависел бы от этого значения.

Как и ранее было показано для других типов операций, применение Null-значений в логических операциях могут также привести к неожиданным значениям. Например, логика на первый взгляд нарушена в **законе исключения третьего** (x $\vee$ $\neg$ x) и в **законе рефлексивности** (x = x), поскольку при x := Null имеем:

(x $\vee$ $\neg$ x), (x = x) := Null.

Законы не выполняются! Объясняется это так же, как и раньше: при подстановке Null-значения в выражение информация о том, что это значение сообщается одной и той же переменной теряется, а в силу вступает общее правило работы с Null-значениями.

Таким образом, делаем вывод: при выполнении логических операций с Null-значениями в качестве операнда эти значения определяются системами управления базами данных как **применимое, но неизвестное**.

5. Null-значения и проверка условий

Итак, из всего вышесказанного можно сделать вывод, что в логике систем управления базами данных имеются не два логических значения (True и False), а три, ведь Null-значение также рассматривается как одно из возможных логических значений. Именно поэтому на него часто ссылаются как на неизвестное значение, значение Unknown.

Однако, несмотря на это, в системах управления базами данных реализуется только двузначная логика. Поэтому условие с Null-значением (неопределенное условие) должно интерпретироваться машиной либо как True, либо как False.

В языке СУБД по умолчанию установлено опознавание условия с Null-значением как значения False. Проиллюстрируем это следующими примерами реализации в системах управления базами данных условных операторов If и While:

If P then A else B;

Эта запись означает: если P принимает значение True, то выполняется действие A, а если P принимает значение False или Null, то выполняется действие B.

Теперь применим к этому оператору операцию отрицания, получим:

If ¬ P then B else A;

В свою очередь, этот оператор означает следующее: если ¬ P принимает значение True, то выполняется действие B, а в том случае, если ¬ P принимает значение False или Null, то будет выполняться действие A.

И снова, как мы видим, при появлении Null-значения мы сталкиваемся с неожиданными результатами. Дело в том, что два оператора If в этом примере не эквивалентны! Хотя один из них получен из другого отрицанием условия и перестановкой ветвей, т. е. стандартной операцией. Такие операторы в общем случае эквивалентны! Но в нашем примере мы видим, что Null-значению условия P в первом случае соответствует команда B, а во втором — A.

А теперь рассмотрим действие условного оператора While:

While P do A; B;

Как работает этот оператор? Пока переменная P имеет значение True, будет выполняться действие A, а как только P примет значение False или Null, выполнится действие B.

Но не всегда Null-значения интерпретируются как False. Например, в ограничениях целостности неопределенные условия опознаются как True (ограничения целостности — это условия, накладываемые на входные данные и обеспечивающие их корректность). Это происходит потому, что в таких ограничениях отвергнуть нужно только заведомо ложные данные.

И опять-таки в системах управления базами данных существует специальная **функция подмены IfNull (ограничения целостности, True)**, с помощью которой Null-значения и неопределенные условия можно представить в явном виде.

Перепишем условные операторы If и While с использованием этой функции:

1) If IfNull (P, False) then A else B;

2) While IfNull (P, False) do A; B;

Итак, функция подмены IfNull (выражение 1, выражение 2) возвращает значение первого выражения, если оно не содержит Null-значения, и значение второго выражения — в противном случае.

Надо заметить, что на тип возвращаемого функцией IfNull выражения никаких ограничений не накладывается. Поэтому с помощью этой функции можно явно переопределить любые правила работы с Null-значениями.

ЛЕКЦИЯ № 3. Реляционные объекты данных

1. Требования к табличной форме представления отношений

1. Самое первое требование, предъявляемое к табличной форме представления отношений, — это конечность. Работать с бесконечными таблицами, отношениями или любыми другими представлениями и организациями данных неудобно, редко оправдываются затраченные усилия, и, кроме того, подобное направление имеет малое практическое приложение.

Но помимо этого, вполне ожидаемого, существуют и другие требования.

2. Заголовок таблицы, представляющей отношение, должен обязательно состоять из одной строки — заголовка столбцов, причем с уникальными именами. Многоярусных заголовков не допускается. Например, таких:

A			**B**		**C**
1	**2**	**3**	**1**	**2**	
…	…	…	…	…	…

Все многоярусные заголовки заменяются одноярусными путем подбора подходящих заголовков. В нашем примере таблица после указанных преобразований будет выглядеть следующим образом:

$\mathbf{A_1}$	$\mathbf{A_2}$	$\mathbf{A_3}$	$\mathbf{B_1}$	$\mathbf{B_2}$	**C**
…	…	…	…	…	…

Мы видим, что имя каждого столбца уникально, поэтому их можно как угодно менять местами, т. е. их порядок становится несущественным.

А это очень важно, поскольку является третьим свойством.

3. Порядок строк должен быть несущественным. Однако это требование также не является строго ограничительным, так как можно без труда привести любую таблицу к требуемому виду. Например,

можно ввести дополнительный столбец, который будет определять порядок строк. В этом случае от перестановки строк тоже ничего не изменится. Вот пример такой таблицы:

...	...	**Порядок**
...	...	1
...	...	3
...	...	2

4. В таблице, представляющей отношение, не должно быть строк-дубликатов. Если же в таблице встречаются повторяющиеся строки, это можно легко исправить введением дополнительного столбца, отвечающего за количество дубликатов каждой строки, например:

...	...	**Число дубликатов**
...	...	0
...	...	3
...	...	1

Следующее свойство также является вполне ожидаемым, потому что лежит в основе всех принципов программирования и проектирования реляционных баз данных.

5. Данные во всех столбцах должны быть одного и того же типа. И кроме того они должны быть простого типа.

Поясним, что такое простой и сложный типы данных.

Простой тип данных — это такой тип, значения данных которого не являются составными, т. е. не содержат составных частей. Таким образом, в столбцах таблицы не должны присутствовать ни списки, ни массивы, ни деревья, ни подобные названным составные объекты.

Такие объекты — **составной тип данных** — в реляционных системах управления базами данных сами представляются в виде самостоятельных таблиц-отношений.

2. Домены и атрибуты

Домены и атрибуты — базовые понятия в теории создания баз данных и управления ими. Поясним, что же это такое.

Формально, **домен атрибута** (обозначается **dom(a)**), где a — некий атрибут, определяется как множество допустимых значений одного и того же типа соответствующего атрибута a. Этот тип должен быть простым, т. е:

$dom(a) \subseteq \{x \mid type(x) = type(a)\}$;

Атрибут (обозначается a), в свою очередь, определяется как упорядоченная пара, состоящая из имени атрибута name(a) и домена атрибута dom(a), т. е.:

a = (name(a): dom(a));

В этом определении вместо привычного знака «,» (как в стандартных определениях упорядоченных пар) используется «:». Это делается для того, чтобы подчеркнуть ассоциацию домена атрибута и типа данных атрибута.

Приведем несколько примеров различных атрибутов:

a_1 = (Курс: {1, 2, 3, 4, 5});

a_2 = (МассаКг: {x | type(x) = real, x 0});

a_3 = (ДлинаСм: {x | type(x) = real, x 0});

Заметим, что у атрибутов a_2 и a_3 домены формально совпадают. Но семантическое значение этих атрибутов различно, ведь сравнивать значения массы и длины бессмысленно. Поэтому домен атрибута ассоциируется не только с типом допустимых значений, но и семантическим значением.

В табличной форме представления отношений атрибут отображается как заголовок столбца таблицы, и при этом домен атрибута не указывается, но подразумевается. Это выглядит следующим образом:

a_1	a_2	a_3
…	…	…
…	…	…

Нетрудно заметить, что здесь каждый из заголовков a_1, a_2, a_3 столбцов таблицы, представляющей какое-то отношение, является отдельным атрибутом.

3. Схемы отношений. Именованные значения кортежей

В теории и практике СУБД понятия схемы отношения и именованного значения кортежа на атрибуте являются базовыми. Приведем их.

Схема отношения (обозначается **S**) определяется как конечное множество атрибутов с уникальными именами, т. е.:

S = {a | a ∈ S};

В каждой таблице, представляющей отношение, все заголовки столбцов (все атрибуты) объединяются в схему этого отношения.

Количество атрибутов в схеме отношений определяет **степень** этого **отношения** и обозначается как мощность множества: |**S**|.

Схема отношений может ассоциироваться с именем схемы отношений.

В табличной форме представления отношений, как нетрудно заметить, схема отношения — это не что иное, как строка заголовков столбцов.

a_1	a_2	a_3	a_4
...	...	...	...

$S = \{a_1, a_2, a_3, a_4\}$ — схема отношений этой таблицы.

Имя отношения изображается как схематический заголовок таблицы.

В текстовой же форме представления схема отношений может быть представлена как именованный список имен атрибутов, например:

Студенты (№ зачетной книжки, Фамилия, Имя, Отчество, Дата рождения).

Здесь, как и в табличной форме представления, домены атрибутов не указываются, но подразумеваются.

Из определения следует, что схема отношения может быть и пустой ($S = \varnothing$). Правда, возможно это только в теории, так как на практике система управления базами данных никогда не допустит создания пустой схемы отношения.

Именованное значение кортежа на атрибуте (обозначается **t(a)**)определяется по аналогии с атрибутом как упорядоченная пара, состоящая из имени атрибута и значения атрибута, т. е.:

$t(a) = (name(a) : x), x \in dom(a)$;

Видим, что значение атрибута берется из домена атрибута.

В табличной форме представления отношения каждое именованное значение кортежа на атрибуте — это соответствующая ячейка таблицы:

...	...	...
$t(a_1)$	$t(a_2)$	$t(a_3)$
...	...	...

Здесь $t(a_1)$, $t(a_2)$, $t(a_3)$ — именованные значения кортежа t на атрибутах a_1, a_2, a_3.

Простейшие примеры именованных значений кортежей на атрибутах:

(Курс: 5), (Балл: 5);

Здесь соответственно Курс и Балл — имена двух атрибутов, а 5 — это одно из их значений, взятое из их доменов. Разумеется, хоть эти значения в обоих случаях равны друг другу, семантически они различны, так как множества этих значений в обоих случаях отличаются друг от друга.

4. Кортежи. Типы кортежей

Понятие кортежа в системах управления базами данных может быть интуитивно найдено уже из предыдущего пункта, когда мы говорили об именованном значении *кортежа* на различных атрибутах. Итак, **кортеж** (обозначается **t**, от англ. tuple — «кортеж») со схемой отношения S определяется как множество именованных значений этого кортежа на всех атрибутах, входящих в данную схему отношений S. Другими словами, атрибуты берутся из **области определения кортежа, def(t)**, т. е.:

$t \equiv t(S) = \{t(a) \mid a \in def(t) \subseteq S;$.

Важно, что одному имени атрибута обязательно должно соответствовать не более одного значения атрибута.

В табличной форме записи отношения кортежем будет любая строка таблицы, т. е.:

...	...	...	...
$t(a_1)$	$t(a_2)$	$t(a_3)$	$t(a_4)$
$t(a_5)$	$t(a_6)$	$t(a_7)$	$t(a_8)$
...	...	...	...

Здесь $t_1(S) = \{t(a_1), t(a_2), t(a_3), t(a_4)\}$ и $t_2(S) = \{t(a_5), t(a_6), t(a_7), t(a_8)\}$ — кортежи.

Кортежи в СУБД различаются по **типам** в зависимости от своей области определения. Кортежи называются:

1) **частичными,** если их область определения включается или совпадает со схемой отношения, т. е. $def(t) \subseteq S$.

Это общий случай в практике баз данных;

2) **полными,** в том случае если их область определения полностью совпадает, равна схеме отношения, т. е. $def(t) = S$;

3) **неполными,** если область определения полностью включается в схему отношений, т. е. $def(t) \subset S$;

4) **нигде не определенными,** если их область определения равна пустому множеству, т. е. $def(t) = \varnothing$.

Поясним на примере. Пусть у нас имеется отношение, заданное следующей таблицей.

a	b	C
10	20	30
10	20	Null
Null	Null	Null

Пусть здесь $t_1 = \{10, 20, 30\}$, $t_2 = \{10, 20, Null\}$, $t_3 = \{Null, Null, Null\}$. Тогда легко заметить, что кортеж t_1 — полный, так как его область определения $def(t_1) = \{a, b, c\} = S$.

Кортеж t_2 — неполный, $def(t_2) = \{a, b\} \subset S$. И, наконец, кортеж t_3 — нигде не определенный, так как его def(t3) = $\varnothing$.

Надо заметить, что нигде не определенный кортеж — это пустое множество, тем не менее ассоциируемое со схемой отношений. Иногда нигде не определенный кортеж обозначается: $\varnothing(S)$. Как мы уже видели в приведенном примере, такой кортеж представляет собой строку таблицы, состоящую только из Null-значений.

Интересно, что **сравнимыми**, т. е. возможно равными, являются только кортежи с одной и той же схемой отношений. Поэтому, например, два нигде не определенных кортежа с различными схемами отношений не будут равными, как могло ожидаться. Они будут различными так же, как их схемы отношений.

5. Отношения. Типы отношений

И наконец дадим определение отношению, как некой вершине пирамиды, состоящей из всех предыдущих понятий. Итак, **отношение** (обозначается **r**, от англ. relation — «отношение») со схемой отношений S определяется как обязательно конечное множество кортежей, имеющих ту же схему отношения S. Таким образом:

$r \equiv r(S) = \{t(S) \mid t \in r\}$;

По аналогии со схемами отношений количество кортежей в отношении называют **мощностью отношений** и обозначают как мощность множества: $|\mathbf{r}|$.

Отношения, как и кортежи, различаются по типам. Итак, отношения называются:

1) **частичными**, если для любого входящего в отношение кортежа выполняется следующее условие: $[def(t) \subseteq S]$.

Это (как и с кортежами) общий случай;

2) **полными**, в том случае если $\forall t \in r(S)$ выполняется: $[def(t) = S]$;
3) **неполными**, если $\exists t \in r(S)\ def(t) \subset S$;
4) **нигде не определенными**, если $\forall t \in r(S)\ [def(t) = \varnothing]$.

Обратим отдельное внимание на нигде не определенные отношения. В отличие от кортежей работа с такими отношениями включает в себя небольшую тонкость. Дело в том, что нигде не определенные отношения могут быть двух видов: они могут быть либо пустыми, либо могут содержать единственный нигде не определенный кортеж (такие отношения обозначаются $\{\varnothing(S)\}$).

Сравнимыми (по аналогии с кортежами), т. е., возможно равными, являются лишь отношения с одной и той же схемой отношения. Поэтому отношения с различными схемами отношений являются различными.

В табличной форме представления, отношение — это тело таблицы, которому соответствует строка — заголовок столбцов, т. е. буквально — вся таблица, вместе с первой строкой, содержащей заголовки.

ЛЕКЦИЯ № 4. Реляционная алгебра. Унарные операции

Реляционная алгебра, как нетрудно догадаться, — это особая разновидность алгебры, в которой все операции производятся над реляционными моделями данных, т. е. над отношениями.

В табличных терминах отношение включает в себя строки, столбцы и строку — заголовок столбцов. Поэтому естественными унарными операциями являются операции выбора определенных строк или столбцов, а также смены заголовков столбцов — переименования атрибутов.

1. Унарная операция выборки

Первой унарной операцией, которую мы рассмотрим, является **операция выборки** — операция выбора строк из таблицы, представляющей отношение, по какому-либо принципу, т. е. выбор строк-кортежей, удовлетворяющих определенному условию или условиям.

Оператор выборки обозначается $\sigma <P>$, **условие выборки** — $P <S>$, т. е., оператор σ берется всегда с определенным условием на кортежи P, а само условие P записывается зависящим от схемы отношения S. С учетом всего этого сама **операция выборки** над схемой отношения S применительно к отношению r будет выглядеть следующим образом:

$$\sigma <P> r(S) \equiv \sigma <P> r = \{t(S) \mid t \in r \ \& \ P <S> t\} =$$
$$= \{t(S) \mid t \in r \ \& \ IfNull(P <S> t, False\};$$

Результатом этой операции будет новое отношение с той же схемой отношения S, состоящее из тех кортежей $t(S)$ исходного отношения-операнда, которые удовлетворяют условию выборки $P <S> t$. Понятно, что для того, чтобы применить какое-то условие к кортежу, необходимо подставить значения атрибутов кортежа вместо имен атрибутов.

Чтобы лучше понять принцип работы этой операции, приведем пример. Пусть дана следующая схема отношения:

S: Сессия (№ зачетной книжки, Фамилия, Предмет, Оценка).

Условие выборки возьмем такое:

$P <S>$ = (Предмет = ‘Информатика’ and Оценка > 3).

Нам необходимо из исходного отношения-операнда выделить те кортежи, в которых содержится информация о студентах, сдавших предмет «Информатика» не ниже, чем на три балла.

Пусть также дан следующий кортеж из этого отношения:

$t_0(S) \in r(S)$: {(№ зачетной книжки: 100), (Фамилия: ‘Иванов’), (Предмет: ‘Базы данных’), (Оценка: 5)};

Применяем наше условие выборки к кортежу t_0, получаем:

$P <S> t_0$ = (‘Базы данных’ = ‘Информатика’ and 5 > 3);

На данном конкретном кортеже условие выборки не выполняется.

А вообще результатом этой конкретной выборки

σ <Предмет = 'Информатика' and Оценка > 3 > Сессия

будет таблица «Сессия», в которой оставлены строки, удовлетворяющие условию выборки.

2. Унарная операция проекции

Еще одна стандартная унарная операция, которую мы изучим, — это операция проекции. **Операция проекции** — это операция выбора столбцов из таблицы, представляющей отношение, по какому-либо признаку. А именно машина выбирает те атрибуты (т. е. буквально те столбцы) исходного отношения-операнда, которые были указаны в проекции.

Оператор проекции обозначается $[S']$ или π $<S'>$. Здесь S' — подсхема исходной схемы отношения S, т. е. ее некоторые столбцы. Что это означает? Это означает, что у S' атрибутов меньше, чем у S, потому что в S' остались только те из них, для которых выполнилось условие проекции. А в таблице, представляющей отношение $r(S')$, строк столько же, сколько их у таблицы $r(S)$, а столбцов — меньше, так как остались только соответствующие оставшимся атрибутам. Таким образом, оператор проекции π $< S'>$ применительно к отношению $r(S)$ дает в результате новое отношение с другой схемой отношения $r(S')$, состоящее из проекций $t(S)$ $[S']$ кортежей исходного отношения. Как определяются эти проекции кортежей? **Проекция** любого кортежа $t(S)$ исходного отношения $r(S)$ на подсхему S' определяется следующей формулой:

$t(S)\ [S'] = \{t(a) \mid a \in def(t) \cap S'\}$, $S' \subseteq S$.

Важно заметить, что дубликаты кортежей из результата исключаются, т. е. в таблице, представляющей новое, результирующее отношение повторяющихся строк не будет.

С учетом всего вышесказанного, операция проекции в терминах систем управления базами данных будет выглядеть следующим образом:

π $<S'> r(S) \equiv \pi <S'> r \equiv r(S)\ [S'] \equiv r\ [S'] = \{t(S)\ [S'] \mid t \in r\}$;

Рассмотрим пример, иллюстрирующий принцип работы операции выборки.

Пусть дано отношение «Сессия» и схема этого отношения:

S: Сессия (№ зачетной книжки, Фамилия, Предмет, Оценка);

Нас будут интересовать только два атрибута из этой схемы, а именно «№ зачетной книжки» и «Фамилия» студента, поэтому подсхема S' будет выглядеть следующим образом:

S': (№ зачетной книжки, Фамилия).

Нужно исходное отношение $r(S)$ спроецировать на подсхему S'.

Далее, пусть нам дан кортеж $t_0(S)$ из исходного отношения:

$t_0(S) \in r(S)$: {(№ зачетной книжки: 100), (Фамилия: 'Иванов'), (Предмет: 'Базы данных'), (Оценка: 5)};

Значит, проекция этого кортежа на данную подсхему S' будет выглядеть следующим образом:

$t_0(S)$ S': {(№ зачетной книжки: 100), (Фамилия: 'Иванов')};

Если говорить об операции проекции в терминах таблиц, то проекция Сессия [№ зачетной книжки, Фамилия] исходного отношения — это таблица Сессия, из которой вычеркнуты все столбцы, кроме двух: № зачетной книжки и Фамилия. Кроме того, все дублирующиеся строки также удалены.

3. Унарная операция переименования

И последняя унарная операция, которую мы рассмотрим, — это **операция переименования атрибутов**. Если говорить об отношении как о таблице, то операция переименования нужна для того, чтобы поменять названия всех или некоторых столбцов.

Оператор переименования выглядит следующим образом: $\rho < \varphi >$, здесь φ — **функция переименования**.

Эта функция устанавливает взаимно-однозначное соответствие между именами атрибутов схем S и $\tilde{S}$, где соответственно S — схема исходного отношения, а $\tilde{S}$ — схема отношения с переименованными атрибутами. Таким образом, оператор $\rho < \varphi >$ в применении к отношению $r(S)$ дает новое отношение со схемой $\tilde{S}$, состоящее из кортежей исходного отношения только с переименованными атрибутами.

Запишем операцию переименования атрибутов в терминах систем управления базами данных:

$\rho < \varphi > r(S) \equiv \rho < \varphi > r = \{ \rho < \varphi > t(S) \mid t \in r\}$;

Приведем пример использования этой операции:

Рассмотрим уже знакомое нам отношение Сессия, со схемой:

S: Сессия (№ зачетной книжки, Фамилия, Предмет, Оценка);

Введем новую схему отношения $\tilde{S}$, с другими именами атрибутов, которые мы бы хотели видеть вместо имеющихся:

$\tilde{S}$: (№ ЗК, Фамилия, Предмет, Балл);

Например, заказчик базы данных захотел в вашем готовом отношении видеть другие названия. Чтобы воплотить в жизнь этот заказ, необходимо спроектировать следующую функцию переименования:

φ : (№ зачетной книжки, Фамилия, Предмет, Оценка) $\rightarrow$ (№ ЗК, Фамилия, Предмет, Балл);

Фактически, требуется поменять имя только у двух атрибутов, поэтому законно будет записать следующую функцию переименования вместо имеющейся:

φ : (№ зачетной книжки, Оценка) $\rightarrow$ (№ ЗК, Балл);

Далее, пусть дан также уже знакомый нам кортеж принадлежащий отношению Сессия:

$t_0(S) \in r(S)$: {(№ зачетной книжки: 100), (Фамилия: 'Иванов'), (Предмет: 'Базы данных'), (Оценка: 5)};

Применим оператор переименования к этому кортежу:

$\rho < \varphi > t_0(S)$: {(№ ЗК: 100), (Фамилия: 'Иванов'), (Предмет: 'Базы данных'), (Балл: 5)};

Итак, это один из кортежей нашего отношения, у которого переименовали атрибуты.

В табличных терминах отношение

ρ < № зачетной книжки, Оценка $\rightarrow$ «№ ЗК, Балл > Сессия —

это новая таблица, полученная из таблицы отношения «Сессия», переименованием указанных атрибутов.

4. Свойства унарных операций

У унарных операций, как и у любых других, есть определенные свойства. Рассмотрим наиболее важные из них.

Первым свойством унарных операций выборки, проекции и переименования является свойство, характеризующее соотношение мощностей отношений. (Напомним, что мощность — это количество кортежей в том или ином отношении.) Понятно, что здесь рассматривается соответственно отношение исходное и отношение, полученное в результате применения той или иной операции.

Заметим, что все свойства унарных операций следуют непосредственно из их определений, поэтому их можно легко объяснить и даже при желании вывести самостоятельно.

Итак:

1) **соотношение мощностей:**

а) для операции выборки: $|\sigma <P> r| \leq |r|$;

б) для операции проекции: $|r[S']| \leq |r|$;

в) для операции переименования: $|\rho < \varphi > r| = |r|$;

Итого, мы видим, что для двух операторов, а именно для оператора выборки и оператора проекции, мощность исходных отношений — операндов больше, чем мощность отношений, получаемых из исходных применением соответствующих операций. Это происходит потому, что при выборе, сопутствующему действию этих двух операций выборки и проекции, происходит исключение некоторых строк или столбцов, не удовлетворивших условиям выбора. В том случае, когда условиям удовлетворяют все строки или столбцы, уменьшения мощности (т. е. количества кортежей) не происходит, поэтому в формулах неравенство нестрогое.

В случае же операции переименования, мощность отношения не изменяется, за счет того, что при смене имен никакие кортежи из отношения не исключаются;

2) **свойство идемпотентности:**

а) для операции выборки: $\sigma <P> \sigma <P> r = \sigma <P>$;

б) для операции проекции: $r[S'][S'] = r[S']$;

в) для операции переименования в общем случае свойство идемпотентности неприменимо.

Это свойство означает, что двойное последовательное применение одного и того же оператора к какому-либо отношению равносильно его однократному применению.

Для операции переименования атрибутов отношения, вообще говоря, это свойство может быть применено, но обязательно со специальными оговорками и условиями.

Свойство идемпотентности очень часто используется для упрощения вида выражения и приведения его к более экономичному, актуальному виду.

И последнее свойство, которое мы рассмотрим, — это свойство монотонности. Интересно заметить, что при любых условиях все три оператора монотонны;

3) **свойство монотонности:**

а) для операции выборки: $r_1 \subseteq r_2 \Rightarrow \sigma <P> r_1 \Rightarrow \sigma <P> r_2$;

б) для операции проекции: $r_1 \subseteq r_2 \Rightarrow r_1[S'] \subseteq r_2[S']$;

в) для операции переименования: $r_1 \subseteq r_2 \Rightarrow \rho <\varphi> r_1 \subseteq \rho <\varphi> r_2$;

Понятие монотонности в реляционной алгебре аналогично этому же понятию из алгебры обычной, общей. Поясним: если изначально отношения r_1 и r_2 были связаны между собой таким образом, что $r_1 \subseteq r_2$, то и после применения любого их трех операторов выборки, проекции или переименования это соотношение сохранится.

ЛЕКЦИЯ № 5. Реляционная алгебра. Бинарные операции

1. Операции объединения, пересечения, разности

У любых операций есть свои правила применимости, которые необходимо соблюдать, чтобы выражения и действия не теряли смысла. Бинарные теоретико-множественные операции объединения, пересечений и разности могут быть применены только к двум отношениям обязательно с одной и той же схемой отношения. Результатом таких бинарных операций будут являться отношения, состоящие из кортежей, удовлетворяющих условиям операций, но с такой же схемой отношения, как и у операндов.

1. Результатом **операции объединения** двух отношений $r_1(S)$ и $r_2(S)$ будет новое отношение $r_3(S)$, состоящее из тех кортежей отношений $r_1(S)$ и $r_2(S)$, которые принадлежат хотя бы одному из исходных отношений и с такой же схемой отношения.

Таким образом, пересечение двух отношений — это:

$r_3(S) = r_1(S) \cup r_2(S) = \{t(S) \mid t \in r_1 \cup t \in r_2\};$

Для наглядности, приведем пример в терминах таблиц:

Пусть даны два отношения:

$r_1(S)$:

S	
a	1
b	2

$r_2(S)$:

S	
b	2
c	3
d	4

Мы видим, что схемы первого и второго отношений одинаковы, только имеют различной количество кортежей. Объединением этих

двух отношений будет отношение $r_3(S)$, которому будет соответствовать следующая таблица:

$r_3(S) = r_1(S) \cup r_2(S)$:

S	
a	1
b	2
c	3
d	4

Итак, схема отношения S не изменилась, только выросло количество кортежей.

2. Перейдем к рассмотрению следующей бинарной операции — **операции пересечения** двух отношений. Как мы знаем еще из школьной геометрии, в результирующее отношение войдут только те кортежи исходных отношений, которые присутствуют одновременно в обоих отношениях $r_1(S)$ и $r_2(S)$ (снова обращаем внимание на одинаковую схему отношения).

Операция пересечения двух отношений будет выглядеть следующим образом:

$r_4(S) = r_1(S) \cap r_2(S) = \{t(S) \mid t \in r_1 \ \& \ t \in r_2\}$;

И снова рассмотрим действие этой операции над отношениями, представленными в виде таблиц:

$r_1(S)$:

S	
a	1
b	2

$r_2(S)$:

S	
b	2
c	3
d	4

Согласно определению операции пересечением отношений $r_1(S)$ и $r_2(S)$ будет новое отношение $r_4(S)$, табличное представление которого будет выглядеть следующим образом:

$r_4(S) = r_1(S) \cap r_2(S)$:

S	
b	2

Действительно, если посмотреть на кортежи первого и второго исходного отношений, общий среди них только один: {b, 2}. Он и стал единственным кортежем нового отношения $r_4(S)$.

3. **Операция разности** двух отношений определяется аналогичным с предыдущими операциями образом. Отношения-операнды, так же, как и в предыдущих операциях, должны иметь одинаковые схемы отношения, тогда в результирующее отношение войдут все те кортежи первого отношения, которых нет во втором, т. е.:

$r_5(S) = r_1(S) \setminus r_2(S) = \{t(S) \mid t \in r_1 \ \& \ t \notin r_2\}$;

Уже хорошо знакомые нам отношения $r_1(S)$ и $r_2(S)$, в табличном представлении выглядящие следующим образом:

$r_1(S)$:

S	
a	1
b	2

$r_2(S)$:

S	
b	2
c	3
d	4

Мы рассмотрим как операнды в операции пересечения двух отношений. Тогда, следуя данному определению, результирующее отношение $r_5(S)$ будет выглядеть следующим образом:

$r_5(S) = r_1(S) \setminus r_2(S)$:

S	
a	1

Рассмотренные бинарные операции являются базовыми, на них основываются другие операции, более сложные.

2. Операции декартового произведения и естественного соединения

Операция декартового произведения и операция естественного соединения являются бинарными операциями типа произведения и основываются на операции объединения двух отношений, которую мы рассматривали ранее.

Хотя действие операции декартова произведения многим может показаться знакомым, начнем мы все-таки с операции естественного произведения, так как она является более общим случаем, нежели первая операция.

Итак, рассмотрим операцию естественного соединения. Следует сразу заметить, что операндами этого действия могут являться отношения с разными схемами в отличие от трех бинарных операций объединения, пересечения и переименования.

Если рассмотреть два отношения с различными схемами отношений $r_1(S_1)$ и $r_2(S_2)$, то их **естественным соединением** будет новое отношение $r_3(S_3)$, которое будет состоять только из тех кортежей операндов, которые совпадают на пересечении схем отношений. Соответственно, схема нового отношения будет больше любой из схем отношений исходных, так как является их соединением, «склеиванием». Кстати, кортежи, одинаковые в двух отношениях-операндах, по которым и происходит это «склеивание», называются **соединимыми**.

Запишем определение операции естественного соединения на языке формул систем управления базами данных:

$r_3(S_3) = r_1(S_1) \times r_2(S_2) = \{t(S_1 \cup S_2) \mid t[S_1] \in r_1 \ \& \ t(S_2) \in r_2\}$;

Рассмотрим пример, хорошо иллюстрирующий работу естественного соединения, его «склеивание». Пусть дано два отношения $r_1(S_1)$ и $r_2(S_2)$, в табличной форме представления соответственно равные:

$r_1(S_1)$:

a	1
b	1
c	3

$r_2(S_2)$:

1	x
2	y
3	z

Мы видим, что у этих отношений присутствуют кортежи, совпадающие при пересечении схем S_1 и S_2 отношений. Перечислим их:

1) кортеж {a, 1} отношения $r_1(S_1)$ совпадает с кортежем {1, x} отношения $r_2(S_2)$;

2) кортеж {b, 1} из $r_1(S_1)$ также совпадает с кортежем {1, x} из $r_2(S_2)$;

3) кортеж {c, 3} совпадает с кортежем {3, z}.

Значит, при естественном соединении новое отношение $r_3(S_3)$ получается «склеиванием» именно на этих кортежах. Таким образом, $r_3(S_3)$ в табличном представлении будет выглядеть следующим образом:

$r_3(S_3) = r_1(S_1) \times r_2(S_2)$:

a	1	x
b	2	y
c	3	z

Получается по определению: схема S_3 не совпадает ни со схемой S_1, ни со схемой S_2, мы «склеили» две исходные схемы по пересекающимся кортежам, чтобы получить их естественное соединение.

Покажем схематично, как происходит соединение кортежей при применении операции естественного соединения.

Пусть отношение r_1 имеет условный вид:

.	...

А отношение r_2 — вид:

...	..

Тогда их естественное соединение будет выглядеть следующим образом:

.	...	..

Видим, что «склеивание» отношений-операндов происходит по той самой схеме, что мы приводили ранее, рассматривая пример.

Операция **декартового соединения** является частным случаем операции естественного соединения. Если конкретнее, то, рассматривая действие операции декартового произведения на отношения, мы заведомо оговариваем, что в этом случае может идти речь только о *не-*

пересекающихся схемах отношений. В результате применения обеих операций получаются отношения со схемами, равными объединению схем отношений-операндов, только в декартово произведение двух отношений попадают всевозможные пары их кортежей, так как схемы операндов ни в коем случае не должны пересекаться.

Таким образом, исходя из всего вышесказанного запишем математическую формулу для операции декартового произведения:

$r_4(S_4) = r_1(S_1) \times r_2(S_2) = \{t(S_1 \cup S_2) \mid t[S_1] \in r_1 \ \& \ t(S_2) \in r_2\}, S_1 \cap S_2 = \varnothing;$

Теперь рассмотрим пример, чтобы показать, какой вид будет иметь результирующая схема отношения, при применении операции декартового произведения.

Пусть даны два отношения $r_1(S_1)$ и $r_2(S_2)$, которые в табличном виде представляются следующим образом:

$r_1(S_1)$:

a
b
c

$r_2(S_2)$:

x
y

Итак, мы видим, что ни один из кортежей отношений $r_1(S_1)$ и $r_2(S_2)$, действительно, не совпадает в их пересечении. Поэтому в результирующее отношение $r_4(S_4)$ попадут всевозможные пары кортежей первого и второго отношений-операндов. Получится:

$r_4(S_4) = r_1(S_1) \times r_2(S_2)$:

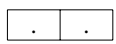
a	x
a	y
b	x
b	y
c	x
c	y

Получилась новая схема отношения $r_4(S_4)$ не «склеиванием» кортежей как в предыдущем случае, а перебором всех возможных различных пар несовпадающих в пересечении исходных схем кортежей.

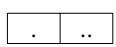
Снова, как и в случае естественного соединения, приведем схематичный пример работы операции декартового произведения.

Пусть r_1 задано следующим условным образом:

А отношение r_2 задано:



Тогда их декартовое произведение схематично можно изобразить следующим образом:



Именно таким образом и получается результирующее отношение при применении операции декартового произведения.

3. Свойства бинарных операций

Из приведенных выше определений бинарных операций объединения, пересечения, разности, декартового произведения и естественного соединения следуют свойства.

1. Первое свойство, как и в случае унарных операций, иллюстрирует **соотношение мощностей** отношений:

1) для операции объединения: $| r_1 \cup r_2 | \leq | r_1 | + | r_2 |$;
2) для операции пересечения: $| r_1 \cap r_2 | \leq min (| r_1 |, | r_2 |)$;
3) для операции разности: $| r_1 \setminus r_2 | \leq | r_1 |$;
4) для операции декартового произведения:
$| r_1 \times r_2 | = | r_1 | \cdot | r_2 |$;
5) для операции естественного соединения:
$| r_1 \times r_2 | \leq | r_1 | \cdot | r_2 |$.

Соотношение мощностей, как мы помним, характеризует, как меняется количество кортежей в отношениях после применения той или иной операции. Итак, что мы видим? Мощность **объединения** двух отношений r_1 и r_2 меньше суммы мощностей исходных отношений-операндов. Почему это происходит? Все дело в том, что при объединении совпадающие кортежи исчезают, накладываясь друг на друга. Так, обратившись к примеру, который мы рассматривали по прохождении этой операции, можно заметить, что в первом отноше-

нии было два кортежа, во втором — три, а в результирующем — четыре, т. е. меньше, чем пять (сумма мощностей отношений-операндов). По совпадающему кортежу {b, 2} эти отношения «склеились».

Мощность результата **пересечения** двух отношений меньше или равна минимальной мощности исходных отношений-операндов. Обратимся к определению этой операции: в результирующее отношение попадают только те кортежи, которые присутствуют в обоих отношениях исходных. А значит, мощность нового отношения никак не может превышать мощности того отношения-операнда, число кортежей которого наименьшее из двух. А равной этой минимальной мощности мощность результата быть может, так как всегда допускается случай, когда все кортежи отношения с меньшей мощностью совпадают с какими-то кортежами второго отношения-операнда.

В случае операции **разности** все достаточно тривиально. Действительно, если из первого отношения-операнда «вычесть» все кортежи, присутствующие также во втором отношении, то их количество (а следовательно, мощность) уменьшится. В том случае, если ни один кортеж первого отношения не совпадет ни с одним кортежем отношения второго, т. е. «вычитать» будет нечего, мощность его не уменьшится.

Интересно, что в случае применения операции **декартового произведения** мощность результирующего отношения в точности равна произведению мощностей двух отношений-операндов. Понятно, что это происходит потому, что в результат записываются все возможные пары кортежей исходных отношений, а ничего не исключается.

И, наконец, операцией **естественного соединения** получается отношение, мощность которого больше или равна произведения мощностей двух исходных отношений. Опять-таки это происходит потому, что отношения-операнды «склеиваются» по совпадающим кортежам, а несовпадающие — из результата исключаются вовсе.

2. **Свойство идемпотентности:**

1) для операции объединения: $r \cup r = r$;
2) для операции пересечения: $r \cap r = r$;
3) для операции разности: $r \setminus r \neq r$;
4) для операции декартового произведения (в общем случае, свойство не применимо);
5) для операции естественного соединения: $r \times r = r$.

Интересно, что свойство идемпотентности верно не для всех операций из приведенных, а для операции декартового произведения

оно и вовсе не применимо. Действительно, если объединить, пересечь или естественно соединить какое-либо отношение само с собой, оно не изменится. А вот если отнять от отношения точно равное ему отношение, в результате получится пустое отношение.

3. **Свойство коммутативности:**

1) для операции объединения: $r_1 \cup r_2 = r_2 \cup r_1$;
2) для операции пересечения: $r_1 \cap r_2 = r_2 \cap r_1$;
3) для операции разности: $r_1 \setminus r_2 \neq r_2 \setminus r_1$;
4) для операции декартового произведения: $r_1 \times r_2 = r_2 \times r_1$;
5) для операции естественного соединения: $r_1 \times r_2 = r_2 \times r_1$.

Свойство коммутативности выполняется для всех операций, кроме операции разности. Это легко понять, ведь от перестановки отношений местами их состав (кортежи) не меняется. А при применении операции разности важно, какое из отношений-операндов стоит на первом месте, потому что от этого зависит, кортежи какого отношения примутся за эталонные, т. е. с какими кортежами будут сравниваться другие кортежи на предмет исключения.

4. **Свойство ассоциативности:**

1) для операции объединения: $(r_1 \cup r_2) \cup r_3 = r_1 \cup (r_2 \cup r_3)$;
2) для операции пересечения: $(r_1 \cap r_2) \cap r_3 = r_1 \cap (r_2 \cap r_3)$;
3) для операции разности: $(r_1 \setminus r_2) \setminus r_3 \neq r_1 \setminus (r_2 \setminus r_3)$;
4) для операции декартового произведения:
$(r_1 \times r_2) \times r_3 = r_1 \times (r_2 \times r_3)$;
5) для операции естественного соединения:
$(r_1 \times r_2) \times r_3 = r_1 \times (r_2 \times r_3)$.

И снова мы видим, что свойство выполняется для всех операций, кроме операции разности. Объясняется это таким же образом, как и в случае применения свойства коммутативности. По большому счету, операциям объединения, пересечения, разности и естественного соединения все равно в каком порядке стоят отношения-операнды. Но при «отнимании» отношений друг от друга порядок играет главенствующую роль.

На основании вышеприведенных свойств и рассуждений можно сделать следующий вывод: три последних свойства, а именно свойство идемпотентности, коммутативности и ассоциативности, верны для всех рассмотренных нами операций, кроме операции разности двух отношений, для которой не выполнилось вообще ни одно из трех означенных свойств, и только в одном случае свойство оказалось неприменимым.

4. Варианты операций соединения

Используя как основу рассмотренные ранее унарные операции выборки, проекции, переименования и бинарные операции объединения, пересечения, разности, декартова произведения и естественного соединения (все они в общем случае называются **операциями соединения**), мы можем ввести новые операции, выведенные с помощью перечисленных понятий и определений. Подобная деятельность называется составлением **вариантов операций соединения**.

Первым таким вариантом операций соединения является операция **внутреннего соединения** по заданному условию соединения.

Операция внутреннего соединения по какому-то определенному условию определяется как производная операция от операций декартового произведения и выборки.

Запишем формульное определение этой операции:

$r_1(S_1) \times_P r_2(S_2) = \sigma <P> (r_1 \times r_2),\ S_1 \cap S_2 = \varnothing$;

Здесь $P = P <S_1 \cup S_2>$ — условие, накладываемое на объединение двух схем исходных отношений-операндов. Именно по этому условию и происходит отбор кортежей из отношений r_1 и r_2 в результирующее отношение.

Следует отметить, что операция внутреннего соединения может применяться к отношениям с разными схемами отношений. Эти схемы могут быть любыми, но они ни в коем случае не должны пересекаться.

Кортежи исходных отношений-операндов, попавшие в результат операции внутреннего соединения, называются **соединимыми кортежами**.

Для наглядного иллюстрирования работы операции внутреннего соединения, приведем следующий пример.

Пусть нам даны два отношения $r_1(S_1)$ и $r_2(S_2)$ с различными схемами отношения:

$r_1(S_1)$:

a1	b1
a	1
b	1
c	3
d	4

$r_2(S_2)$:

b2	c2
1	x
2	y
3	z

Следующая таблица даст результат применения операции внутреннего соединения по условию P = (b1 = b2).

$r_1(S_1) \times_P r_2(S_2)$:

a1	b1	b2	c2
a	1	1	x
b	1	1	x
c	3	3	z

Итак, мы видим, что действительно «слипание» двух таблиц, представляющих отношения, произошло именно по тем кортежам, в которых выполняется условие операции внутреннего соединения P = (b1 = b2).

Теперь на основании уже введенной операции внутреннего соединения мы можем ввести операцию **левого внешнего соединения** и **правого внешнего соединения**. Поясним.

Результатом операции левое внешнее соединение является результат внутреннего соединения, пополненный несоединимыми кортежами левого исходного отношения-операнда. Аналогично результат операции правого внешнего соединения определяется как результат операции внутреннего соединения, пополненный несоединимыми кортежами стоящего справа исходного отношения-операнда.

Вопрос, чем же пополняются результирующие отношения операций левого и правого внешнего соединения, вполне ожидаем. Кортежи одного отношения-операнда дополняются на схеме другого отношения-операнда **Null-значениями**.

Стоит заметить, что введенные таким образом операции левого и правого внешнего соединения являются производными операциями от операции внутреннего соединения.

Чтобы записать общие формулы для операций левого и правого внешнего соединений, проведем некоторые дополнительные построения.

Пусть нам даны два отношения $r_1(S_1)$ и $r_2(S_2)$ с различными схемами отношений S_1 и S_2, не пересекающимися друг с другом.

Так как мы уже оговаривали, что операции левого и правого внутреннего соединения являются производными, то мы можем получить сле-

дующие вспомогательные формулы для определения операции левого внешнего соединения:

1) $r_3(S_2 \cup S_1) := r_1(S_1) \times_P r_2(S_2)$;

$r_3(S_2 \cup S_1)$ — это просто результат внутреннего соединения отношений $r_1(S_1)$ и $r_2(S_2)$. Левое внешнее соединение является производной операцией именно от операции внутреннего соединения, поэтому мы и начинаем наши построения с нее;

2) $r_4(S_1) := r_3(S_2 \cup S_1)\ [S_1]$;

Таким образом, с помощью унарной операции проекции, мы выделили все соединимые кортежи левого исходного отношения-операнда $r_1(S_1)$. Результат обозначили $r_4(S_1)$ для удобства применения;

3) $r_5(S_1) := r_1(S_1) \setminus r_4(S_1)$;

Здесь $r_1(S_1)$ — все кортежи левого исходного отношения-операнда, а $r_4(S_1)$ — его же кортежи, только соединимые. Таким образом, при помощи бинарной операции разности, в отношении $r_5(S_1)$ у нас получились все несоединимые кортежи левого отношения-операнда;

4) $r_6(S_2) := \{\varnothing(S_2)\}$;

$\{\varnothing(S_2)\}$ — это новое отношение со схемой (S_2), содержащее всего один кортеж, причем составленный из Null-значений. Для удобства мы обозначили это отношение $r_6(S_2)$;

5) $r_7(S_2 \cup S_1) := r_5(S_1) \times r_6(S_2)$;

Здесь мы взяли полученные в пункте три, несоединимые кортежи левого отношения-операнда $(r_5(S_1))$ и дополнили их на схеме второго отношения-операнда S_2 Null-значениями, т. е. декартово умножили отношение, состоящее из этих самых несоединимых кортежей на отношение $r_6(S_2)$, определенное в пункте четыре;

6) $r_1(S_1) \rightarrow\times_P r_2(S_2) := (r_1 \times_P r_2) \cup r_7(S_2 \cup S_1)$;

Это и есть **левое внешнее соединение**, полученное, как можно видеть, объединением декартового произведения исходных отношений-операндов r_1 и r_2 и отношения $r_7(S_2 \cup S_1)$, определенного в пункте пятом.

Теперь у нас имеются все необходимые выкладки для определения не только операции левого внешнего соединения, но по аналогии и для определения операции правого внешнего соединения. Итак:

1) операция **левого внешнего соединения** в строгом формулярном виде выглядит следующим образом:

$r_1(S_1) \rightarrow\times_P r_2(S_2) := (r_1 \times_P r_2) \cup [(r_1 \setminus (r_1 \times_P r_2)\ [S_1]) \times \{\varnothing(S_2)\}]$;

2) операция **правого внешнего соединения** определяется подобным образом операции левого внешнего соединения и имеет следующий вид:

$r_1(S_1) \rightarrow\times_P r_2(S_2) := (r_1 \times_P r_2) \cup [(r_2 \setminus (r_1 \times_P r_2)\ [S_2]) \times \{\varnothing(S_1)\}]$;

Эти две производные операции имеют всего два свойства, достойные упоминания.

1. Свойство коммутативности:

1) для операции левого внешнего соединения: $r_1(S_1) \rightarrow\times_P r_2(S_2) \neq$ $\neq r_2(S_2) \rightarrow\times_P r_1(S_1)$;

2) для операции правого внешнего соединения: $r_1(S_1) \leftarrow\times_P r_2(S_2) \neq$ $\neq r_2(S_2) \leftarrow\times_P r_1(S_1)$.

Итак, мы видим, что свойство коммутативности не выполняется для этих операций в общем виде, но при этом операции левого и правого внешнего соединения взаимно обратны друг другу, т. е. выполняется:

1) для операции левого внешнего соединения:

$r_1(S_1) \rightarrow\times_P r_2(S_2) = r_2(S_2) \rightarrow\times_P r_1(S_1)$;

2) для операции правого внешнего соединения:

$r_1(S_1) \leftarrow\times_P r_2(S_2) = r_2(S_2) \leftarrow\times_P r_1(S_1)$.

2. Основным свойством операций левого и правого внешнего соединения является то, что они позволяют **восстановить** исходное отношение-операнд по конечному результату той или иной операции соединения, т. е. выполняются:

1) для операции левого внешнего соединения:

$r_1(S_1) = (r_1 \rightarrow\times_P r_2)\ [S_1]$;

2) для операции правого внешнего соединения:

$r_2(S_2) = (r_1 \leftarrow\times_P r_2)\ [S_2]$.

Таким образои, мы видим, что первое исходное отношение-операнд можно восстановить из результата операции левого правого соединения, а если конкретнее, то применением к результату этого соединения $(r_1 \times r_2)$ унарной операции проекции на схему S_1, $[S_1]$.

И аналогично второе исходное отношение-операнд можно восстановить применением к результату операции правого внешнего соединения $(r_1 \times r_2)$ унарной операции проекции на схему отношения S_2.

Приведем пример для более подробного рассмотрения работы операций левого и правого внешних соединений. Введем уже знакомые нам отношения $r_1(S_1)$ и $r_2(S_2)$ с различными схемами отношения:

$r_1(S_1)$:

a1	b1
a	1
b	1
c	3
d	4

$r_2(S_2)$:

b2	c2
1	x
2	y
3	z

Несоединимый кортеж левого отношения-операнда $r_2(S_2)$ — это кортеж {d, 4}. Следуя определению, именно им следует дополнить результат внутреннего соединения двух исходных отношений-операндов.

Условие внутреннего соединения отношений $r_1(S_1)$ и $r_2(S_2)$ также оставим прежнее: P = (b1 = b2). Тогда результатом операции **левого внешнего соединения** будет следующая таблица:

$r_1(S_1) \overset{\rightarrow}{\times}_P r_2(S_2)$:

a1	b1	b2	c2
a	1	1	x
b	1	1	x
c	3	3	z
d	4	Null	Null

Действительно, как мы можем видеть, в результате воздействия операции левого внешнего соединения, произошло пополнение результата операции внутреннего соединения несоединимыми кортежами левого, т. е. в нашем случае первого отношения-операнда. Пополнение кортежа на схеме второго (правого) исходного отношения-операнда по определению произошло при помощи Null-значений.

И аналогично результатом **правого внешнего соединения** по тому же, что и раньше, условию P = (b1 = b2) исходных отношений-операндов $r_1(S_1)$ и $r_2(S_2)$ является следующая таблица:

$r_1(S_1) \overset{\leftarrow}{\times}_P r_2(S_2)$:

a1	b1	b2	c2
a	1	1	x
b	1	1	x
c	3	3	z
Null	Null	2	y

Действительно, в этом случае пополнять результат операции внутреннего соединения следует несоединимыми кортежами правого, в нашем случае второго исходного отношения-операнда. Такой кортеж, как не трудно видеть, во втором отношении $r_2(S_2)$ один, а именно {2, y}. Далее действуем по определению операции правого внешнего соединения, дополняем кортеж первого (левого) операнда на схеме первого операнда Null-значениями.

И, наконец, рассмотрим третий вариант приведенных ранее операций соединения.

Операция полного внешнего соединения. Эту операцию вполне можно рассматривать не только как операцию, производную от операций внутреннего соединения, но и как объединение операций левого и правого внешнего соединения.

Операция полного внешнего соединения определяется как результат пополнения того же самого внутреннего соединения (как и в случае определения левого и правого внешних соединений) несоединимыми кортежами одновременно и левого, и правого исходных отношений-операндов. Исходя из этого определения дадим формулярный вид этого определения:

$r_1(S_1) \leftrightarrow\times_P r_2(S_2) = (r_1 \rightarrow\times_P r_2) \cup (r_1 \leftarrow\times_P r_2)$;

У операции полного внешнего соединения также имеется свойство, сходное с аналогичным свойством операций левого и правого внешних соединений. Только за счет изначальной взаимно-обратной природы операции полного внешнего соединения (ведь она была определена как объединение операций левого и правого внешних соединений) для нее выполняется **свойство коммутативности**:

$r_1(S_1) \leftrightarrow\times_P r_2(S_2) = r_2(S_2) \leftrightarrow\times_P r_1(S_1)$;

И для завершения рассмотрения вариантов операций соединения, рассмотрим пример, иллюстрирующий работу операции полного внешнего соединения. Введем два отношения $r_1(S_1)$ и $r_2(S_2)$ и условие соединения.

Пусть

$r_1(S_1)$:

a1	b1
a	1
b	1
c	3
d	4

$r_2(S_2)$:

b2	c2
1	x
2	y
3	z

И пусть условием соединения отношений $r_1(S_1)$ и $r_2(S_2)$ будет: P = (b1 = b2), как и в предыдущих примерах.

Тогда результатом операции полного внешнего соединения отношений $r_1(S_1)$ и $r_2(S_2)$ по условию P = (b1 = b2) будет следующая таблица:

$r_1(S_1) \leftrightarrow\times_P r_2(S_2)$:

a1	b1	b2	c2
a	1	1	x
b	1	1	x
c	3	3	z
d	4	Null	Null
Null	Null	2	y

Итак, мы видим, что операция полного внешнего соединения наглядно оправдала свое определение как объединения результатов операций левого и правого внешних соединений. Результирующее отношение операции внутреннего соединения дополнено одновременно несоединимыми кортежами как левого (первого, $r_1(S_1)$), так и правого (второго, $r_2(S_2)$) исходного отношения-операнда.

5. Производные операции

Итак, мы рассмотрели различные варианты операций соединения, а именно операции внутреннего соединения, левого, правого и полного внешнего соединения, которые являются производными восьми исходных операций реляционной алгебры: унарных операций выборки, проекции, переименования и бинарных операций объединения, пересечения, разности, декартова произведения и естественного соединения. Но и среди этих исходных операций есть свои примеры производных операций.

1. Например, операция **пересечения** двух отношений является производной от операции разности этих же двух отношений. Покажем это.

Операцию пересечения можно выразить следующей формулой:

$r_1(S) \cap r_2(S) = r_1 \setminus r_1 \setminus r_2$

или, что дает тот же результат:

$r_1(S) \cap r_2(S) = r_2 \setminus r_2 \setminus r_1$;

2. Еще одним примером, производной базовой операции от восьми исходных операций является операция **естественного соединения**. В самом общем виде эта операция является производной от бинарной операции декартового произведения и унарных операций выборки, проекции и переименования атрибутов. Однако, в свою очередь, операция внутреннего соединения является производной операцией от той же операции декартового произведения отношений. Поэтому, чтобы показать, что операция естественного соединения — производная операция, рассмотрим следующий пример.

Сравним приведенные ранее примеры для операций естественного и внутреннего соединений.

Пусть нам даны два отношения $r_1(S_1)$ и $r_2(S_2)$ которые будут выступать в качестве операндов. Они равны:

$r_1(S_1)$:

a1	**b1**
a	1
b	1
c	3
d	4

$r_2(S_2)$:

b2	**c2**
1	x
2	y
3	z

Как мы уже получали ранее, результатом операции естественного соединения этих отношений будет являться таблица следующего вида:

$r_3(S_3) := r_1(S_1) \times r_2(S_2)$:

a	1	x
b	1	x
c	3	z

А результатом внутреннего соединения этих же отношений $r_1(S_1)$ и $r_2(S_2)$ по условию P = (b1 = b2) будет следующая таблица:

$r_4(S_4) := r_1(S_1) \times_P r_2(S_2)$:

a1	b1	b2	c2
a	1	1	x
b	1	1	x
c	3	3	z

Сравним эти два результата, получившиеся новые отношения $r_3(S_3)$ и $r_4(S_4)$.

Ясно, что операция естественного соединения выражается через операцию внутреннего соединения, но, что главное, с условием соединения специального вида.

Запишем математическую формулу, описывающую действие операции естественного соединения как производную операции внутреннего соединения.

$r_1(S_1) \times r_2(S_2) = \{ \rho < \varphi_1 > r_1 \times_E \rho < \varphi_2 > r_2 \}[S_1 \cup S_2]$,

где E — **условие соединимости** кортежей;

$E = \forall a \in S_1 \cap S_2 \, [IsNull(b1) \,\&\, IsNull(b2) \cup b1 = b2]$;

$b1 = \varphi_1(name(a))$, $b2 = \varphi_2(name(a))$;

Здесь одна из **функций переименования** φ_1 является тождественной, а другая функция переименования (а именно — φ_2) переименовывает атрибуты, на которых наши схемы пересекаются.

Условие соединимости E для кортежей записывается в общем виде с учетом возможных появлений Null-значений, ведь операция внутреннего соединения (как уже было сказано выше) является производной операцией от операции декартового произведения двух отношений и унарной операции выборки.

6. Выражения реляционной алгебры

Покажем, как можно использовать рассмотренные ранее выражения и операции реляционной алгебры в практической эксплуатации различных баз данных.

Пусть для примера в нашем распоряжении имеется фрагмент какой-то коммерческой базы данных:

Поставщики (*Код поставщика, Имя поставщика, Город поставщика*);

Инструменты (*Код инструмента, Имя инструмента, ...*);

Поставки (*Код поставщика, Код детали*);

Подчеркнутые имена атрибутов являются ключевыми (т. е. идентификационными) атрибутами, причем каждый в своем отношении.

Предположим, что к нам, как разработчикам этой базы данных и хранителям информации по этому вопросу, поступил заказ получить наименования поставщиков (Имя Поставщика) и место их расположения (Город Поставщика) в случае, когда эти поставщики не поставляют каких-либо инструментов с родовым именем «Плоскогубцы».

Чтобы в нашей, возможно, весьма обширной, базе данных определить всех поставщиков, отвечающих этому требованию, запишем несколько выражений реляционной алгебры.

1. образуем естественное соединение отношений «Поставщики» и «Поставки» для того, чтобы сопоставить с каждыу поставщиком коды поставляемых им деталей. Новое отношение — результат применения операции естественного соединения — для удобства дальнейшего применения обозначим через r_1.

Поставщики × Поставки:= r_1 (Код поставщика, Имя поставщика, Город поставщика, ~~Код поставщика~~, Код инструмента);

В скобках мы перечислили все атрибуты отношений, участвующих в этой операции естественного соединения. Мы видим, что атрибут «Код поставщика» дублируется, но в итоговой записи операции каждое имя атрибута должно присутствовать только один раз, т. е.:

Поставщики × Поставки:= r_1 (Код поставщика, Имя поставщика, Город поставщика, Код инструмента);

2. снова образуем естественное соединение, только на этот раз отношения, получившегося в пункте один и отношения Инструменты. Делаем это для того, чтобы с каждым кодом инструмента, получившемуся в предыдущем пункте, — сопоставить имя этого инструмента.

r_1 × Инструменты [Код инструмента, Имя инструмента]:= r_2 (Код поставщика, Имя поставщика, Город поставщика, ~~Код инструмента~~, Код инструмента, Имя инструмента);

Получившийся результат обозначим r_2, дублирующиеся атрибуты исключаем:

r_1 × Инструменты [Код инструмента, Имя инструмента]:= r_2 (Код поставщика, Имя поставщика, Город поставщика, Код инструмента, Имя инструмента);

Заметим, что из отношения Инструменты мы берем только два атрибута: «Код инструмента» и «Имя инструмента». Чтобы это осуществить мы, как можно заметить из записи отношения r_2, применили унарную операцию проекции: Инструменты [Код инструмента, Имя инструмента], т. е., если бы отношение Инструменты было представлено в виде таблицы, результатом этой операции проекции стали бы два первых столбца с заголовками соответственно «Код инструмента» и «Имя инструмента».

Интересно заметить, что два первых шага, нами уже рассмотренных, являются достаточно общими, т. е. они могут быть использованы и для реализации каких-либо других запросов.

А вот два следующих пункта, в свою очередь, представляют собой конкретные шаги для достижения поставленной перед нами конкретной задачи.

3. Напишем унарную операцию выборки по условию <«Имя инструмента» = «Плоскогубцы»> применительно к отношению r_2, полученному в предыдущем пункте. А к результату этой операции применим, в свою очередь, унарную операцию проекции [Код поставщика, Имя поставщика, Город поставщика], для того чтобы получить все значения этих атрибутов, потому что именно эту информацию нам и требуется получить исходя из заказа.

Итак:

(σ<Имя инструмента = «Плоскогубцы»> r_2) [Код поставщика, Имя поставщика, Город поставщика]:= r_3 (Код поставщика, Имя поставщика, Город поставщика, Код инструмента, Имя инструмента).

В результирующем отношении, обозначенном через r_3, оказались только те поставщики (со всеми своими опознавательными данными), которые поставляют инструменты с родовым именем «Плоскогубцы». Но нам в силу заказа необходимо выделить тех поставщиков, которые, наоборот, не поставляют таких инструментов. Поэтому перейдем к следующему действию нашего алгоритма и запишем последнее выражение реляционной алгебры, которое и даст нам искомую информацию.

4. Сначала составим разность отношения «Поставщики» и отношения r_3, а после применения этой бинарной операции применим унарную операцию проекции на атрибуты «Имя поставщика» и «Город поставщика».

(Поставщики \ r_3) [Имя поставщика, Город поставщика] := r_4 (Код поставщика, Имя поставщика, Город поставщика);

Результат обозначили r_4, в это отношение и вошли как раз только те кортежи исходного отношения «Поставщики», которые соответствуют условию нашего заказа.

Итак, мы показали, как можно с помощью выражений и операций реляционной алгебры осуществлять всевозможные действия с произвольными базами данных, выполнять различные заказы и т. п.

ЛЕКЦИЯ № 6. Язык SQL

Дадим сначала небольшую историческую справку.

Язык SQL, предназначенный для взаимодействия с базами данных, появился в середине 1970-х гг. (первые публикации датируются 1974 г.) и был разработан в компании IBM в рамках проекта экспериментальной реляционной системы управления базами данных. Исходное название языка — SEQUEL (Structured English Query Language) — только частично отражало суть этого языка. Первоначально, сразу после его изобретения и в первичный период эксплуатации языка SQL, его название являлось аббревиатурой от словосочетания Structured Query Language, что переводится как «Язык структурированных запросов». Конечно, язык был ориентирован главным образом на удобную и понятную пользователям формулировку запросов к реляционным базам данных. Но, в действительности, он почти с самого начала являлся полным языком баз данных, обеспечивающим, помимо средств формулирования запросов и манипулирования базами данных, следующие возможности:

1) средства определения и манипулирования схемой базы данных;
2) средства определения ограничений целостности и триггеров (о которых будет упомянуто позднее);
3) средства определения представлений баз данных;
4) средства определения структур физического уровня, поддерживающих эффективное выполнение запросов;
5) средства авторизации доступа к отношениям и их полям.

В языке отсутствовали средства явной синхронизации доступа к объектам баз данных со стороны параллельно выполняемых транзакций: с самого начала предполагалось, что необходимую синхронизацию неявно выполняет система управления базами данных.

В настоящее время SQL — это уже не аббревиатура, а название самостоятельного языка.

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

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

С течением времени и работы над языком структурированных запросов удалось достигнуть стандарта четкой стандартизации синтаксиса и семантики операторов выборки данных, манипулирования данными и фиксации средств ограничения целостности баз данных. Были специфицированы средства определения первичного и внешних ключей отношений и так называемых проверочных ограничений целостности, которые представляют собой подмножество немедленно проверяемых ограничений целостности SQL. Средства определения внешних ключей позволяют легко формулировать требования так называемой ссылочной целостности баз данных (о которой мы поговорим позднее). Это распространенное в реляционных базах данных требование можно было сформулировать и на основе общего механизма ограничений целостности SQL, но формулировка на основе понятия внешнего ключа более проста и понятна.

Итак, с учетом всего этого в настоящее время язык структурированных запросов — это название не просто одного языка, а название целого класса языков, поскольку, несмотря на имеющиеся стандарты, в различных системах управления базами данных реализуются различные диалекты языка структурированных запросов, имеющие, разумеется, одну общую основу.

1. Оператор Select — базовый оператор языка структурированных запросов

Центральное место в языке структурированных запросов SQL занимает оператор Select, с помощью которого реализуется самая востребованная операция при работе с базами данных — запросы.

Оператор Select осуществляет вычисление выражений как реляционной, так и псевдореляционной алгебры. В данном курсе мы рас-

смотрим реализацию только уже пройденных нами унарных и бинарных операций реляционной алгебры, а также осуществление запросов, с помощью так называемых подзапросов.

Кстати, необходимо заметить, что в случае работы с операциями реляционной алгебры в результирующих отношениях могут появляться дублирующие кортежи. В правилах языка структурированных запросов нет строгого запрещения на присутствие повторяющихся строк в отношениях (в отличие от обычной реляционной алгебры), поэтому исключать дубликаты из результата не обязательно.

Итак, рассмотрим базовую структуру оператора Select. Она достаточно проста и включает в себя следующие стандартные обязательные фразы:

Select ...

From ...

Where ... ;

На месте многоточия в каждой строчке должны стоять отношения, атрибуты и условия конкретной базы данных и задания к ней. В самом общем случае базовая структура Select должна выглядеть следующим образом:

Select *выбрать такие-то атрибуты*

From *из таких-то отношений*

Where *с такими-то условиями выборки кортежей*

Таким образом, выбираем мы атрибуты из схемы отношений (заголовки некоторых столбцов), при этом указывая, из каких отношений (а их, как видно, может быть несколько) мы производим нашу выборку и, наконец, на основании каких условий мы останавливаем свой выбор на тех или иных кортежах.

Важно заметить, что ссылки на атрибуты происходят с помощью их имен.

Таким образом, получается следующий **алгоритм работы** этого базового оператора Select:

1) запоминаются условия выборки кортежей из отношения;

2) проверяется, какие кортежи удовлетворяют указанным свойствам. Такие кортежи запоминаются;

3) на выход выводятся перечисленные в первой строчке базовой структуры оператора Select атрибуты со своими значениями. (Если говорить о табличной форме записи отношения, то выведутся те столбцы таблицы, заголовки которых были перечислены как необходимые атрибуты; разумеется, столбцы выведутся не полностью, в каждом из них останутся только те кортежи, которые удовлетворили названным условиям.)

Рассмотрим пример.

Пусть нам дано следующее отношение r_1, как фрагмент некой базы данных книжного магазина:

Код книги	Название книги	Автор книги	Цена книги
1258963	Холодные берега	С. Лукьяненко	186,9
1236954	Мобильник	С. Кинг	201,4

Пусть также нам дано следующее выражение с оператором Select:

Select *Название книги, Автор книги*

From r_1

Where *Цена книги > 200*;

Результатом этого оператора будет следующий фрагмент кортежа: (Мобильник, С. Кинг).

(В дальнейшем мы подвергнем рассмотрению множество примеров реализации запросов с использованием этой базовой структуры и ее применение изучим очень подробно.)

2. Унарные операции на языке структурированных запросов

В этом параграфе мы рассмотрим, как реализуются на языке структурированных запросов с помощью оператора Select уже знакомые нам унарные операции выборки, проекции и переименования.

Важно заметить, что если раньше мы могли работать только с отдельными операциями, то даже один оператор Select в общем случае позволяет определить целое выражение реляционной алгебры, а не какую-то одну операцию.

Итак, перейдем непосредственно к анализу представления унарных операций на языке структурированных запросов.

1. **Операция выборки.**

Операция выборки на языке SQL реализуется оператором Select следующего вида:

Select *все атрибуты*

From *имя отношения*

Where *условие выборки*;

Здесь вместо того, чтобы писать «все атрибуты», можно использовать значок «*». В теории языка структурированных запросов этот значок означает выбор всех атрибутов из схемы отношения.

Условие выборки здесь (и во всех остальных реализациях операций) записывается в виде логического выражения со стандартными

связками not (не), and (и), or (или). На атрибуты отношения ссылаемся посредством их имен.

Рассмотрим пример. Определим следующую схему отношения:

Успеваемость (№ зачетной книжки, Семестр, Код предмета, Оценка, Дата);

Здесь, как уже упоминалось ранее, подчеркнутые атрибуты образуют ключ отношения.

Составим оператор Select следующего вида, реализующий унарную операцию выборки:

Select *

From *Успеваемость*

Where *№ зачетной книжки = 100 and Семестр = 6*;

Понятно, что в результат этого оператора машина выведет успеваемость студента с номером зачетки сто за шестой семестр.

2. **Операция проекции.**

Операция проекции на языке структурированных запросов реализуется даже проще, чем операция выборки. Напомним, что при применении операции проекции выбираются не строки (как при применении операции выборки), а столбцы. Поэтому достаточно перечислить заголовки нужных столбцов (т. е. имена атрибутов), без указания каких-либо посторонних условий. Итого, получаем оператор следующего вида:

Select *список имен атрибутов*

From *имя отношения*;

После применения этого оператора машина выдаст те столбцы таблицы-отношения, имена которых были указаны в первой строчке этого оператора Select.

Как мы уже упоминали ранее, повторяющиеся строки и столбцы исключать из результирующего отношения не обязательно. Но если в заказе или в задании требуется обязательно элиминировать дубликаты, следует использовать специальную опцию языка структурированных запросов — **distinct**. Эта опция задает автоматическое исключение дубликатов кортежей из отношения. С применением этой опции оператор Select будет выглядеть следующим образом:

Select *distinct список имен атрибутов*

From *имя отношения*;

В языке SQL существует специальное обозначение для необязательных элементов выражений — квадратные скобки [...]. Поэтому в самом общем виде операция проекции будет выглядеть следующим образом:

Select *[distinct] список имен атрибутов*

From *имя отношения*;

Однако если результат применения операции гарантированно не содержит дубликатов или же дубликаты все-таки допустимы, то опцию **distinct** лучше не указывать, чтобы не загромождать запись, т. е. из соображений производительности работы оператора.

Рассмотрим пример, иллюстрирующий возможность стопроцентной уверенности в отсутствии дубликатов. Пусть дана уже известная нам схема отношений:

Успеваемость (№ зачетной книжки, Семестр, Код предмета, Оценка, Дата).

Пусть дан следующий оператор Select:

Select *№ зачетной книжки, Семестр, Код предмета*

From *Успеваемость*;

Здесь, как легко видеть, три возвращающихся оператором атрибута образуют ключ отношения. Именно поэтому опция **distinct** становится излишней, ведь дубликатов гарантированно не будет. Это следует из требования, накладываемого на ключи, называемого ограничением уникальности. Подробнее это свойство мы рассмотрим дальше, но если атрибут ключевой, то дубликатов в нем нет.

3. **Операция переименования.**

Операция переименования атрибутов на языке структурированных запросов осуществляется довольно просто. А именно воплощается в действительность следующим алгоритмом:

1) в списке имен атрибутов фразы Select перечисляются те атрибуты, которые необходимо переименовать;

2) к каждому указанному атрибуту добавляется специальное ключевое слово as;

3) после каждого вхождения слова as указывается то имя соответствующего атрибута, на которое необходимо поменять имя исходное.

Таким образом, с учетом всего вышесказанного, оператор, соответствующий операции переименования атрибутов, будет выглядеть следующим образом:

Select *имя атрибута 1 as новое имя атрибута 1, ...*

From *имя отношения*;

Покажем работу этого оператора на примере. Пусть дана уже знакомая нам схема отношения:

Успеваемость (№ зачетной книжки, Семестр, Код предмета, Оценка, Дата);

Пусть у нас имеется заказ поменять имена некоторых атрибутов, а именно вместо «№ зачетной книжки» должно стоять «№ зачетки» и вместо «Оценка» — «Балл».

Запишем, как будет выглядеть оператор Select, реализующий эту операцию переименования:

Select *зачетной книжки as № зачетки, Семестр, Код предмета, Оценка as Балл, Дата*

From *Успеваемость*;

Таким образом, результатом применения этого оператора будет новая схема отношения, отличающаяся от исходной схемы отношения «Успеваемость» именами двух атрибутов.

3. Бинарные операции на языке структурированных запросов

Как и унарные операции, операции бинарные также имеют свою реализацию на языке структурированных запросов или SQL. Итак, рассмотрим осуществление на этом языке уже пройденных нами бинарных операций, а именно — операций объединения, пересечения, разности, декартового произведения, естественного соединения, внутреннего и левого, правого, полного внешнего соединения.

1. **Операция объединения.**

Для того чтобы реализовать операцию объединения двух отношений приходится использовать одновременно два оператора Select, каждый из которых соответствует какому-то одному из исходных отношений-операндов. И к этим двум базовым операторам Select необходимо применить специальную операцию **Union**. Учитывая все вышесказанное, запишем, как же операция объединения будет выглядеть с использованием семантики языка структурированных запросов:

Select *список имен атрибутов отношения 1*

From *имя отношения 1*

Union

Select *список имен атрибутов отношения 2*

From *имя отношения 2*;

Важно заметить, что списки имен атрибутов двух объединяемых отношений должны ссылаться на атрибуты совместимых типов и быть перечислены в согласованном порядке. Если это требование не соблюдать, ваш запрос не сможет быть выполнен, и компьютер выдаст сообщение об ошибке.

Но, что интересно отметить, сами имена атрибутов в этих отношениях могут быть различными. В таком случае результирующему отношению приписываются имена атрибутов, указанные в первом операторе Select.

Также необходимо знать, что использование операции Union предполагает автоматическое исключение из результирующего отношения всех дубликатов кортежей. Поэтому, если вам нужно, чтобы все повторяющиеся строки в конечном результате сохранились, вместо операции Union следует применять модификацию этой операции — операцию **Union All**. В таком случае операция объединения двух отношений будет выглядеть следующим образом:

Select *список имен атрибутов отношения 1*
From *имя отношения 1*
Union All
Select *список имен атрибутов отношения 2*
From *имя отношения 2*;

В этом случае из результирующего отношения дубликаты кортежей удаляться не будут.

Используя уже упоминавшееся ранее обозначение для необязательных элементов и опций в операторах Select, запишем самый общий вид операции объединения двух отношений на языке структурированных запросов:

Select *список имен атрибутов отношения 1*
From *имя отношения 1*
Union [All]
Select *список имен атрибутов отношения 2*
From *имя отношения 2*;

2. **Операция пересечения.**

Операция пересечения и операция разности двух отношений на языке структурированных запросов реализуются похожим образом (мы рассматриваем наиболее простой способ представления, так как, чем проще метод, тем он экономичнее, актуальнее и, следовательно, наиболее востребован). Итак, мы подвергнем разбору способ реализации операции пересечения с использованием **ключей**.

Этот способ предполагает участие двух конструкций Select, но они не равноправны (как в представлении операции объединения), одна из них является как бы «подконструкцией», «подциклом». Такой оператор обычно называют **подзапросом**.

Итак, пусть у нас имеются две схемы отношений (R_1 и R_2), приблизительно определенные следующим образом:

R_1 (ключ, ...) и
R_2 (ключ, ...);

Воспользуемся также при записи этой операции специальной опцией **in**, что буквально означает «в» или (как в данном конкретном случае) «содержится в».

Итак, с учетом всего вышесказанного, операция пересечения двух отношений с помощью языка структурированных запросов запишется следующим образом:

Select *

From R_1

Where *ключ* **in**

(**Select** *ключ* **From** R_2);

Таким образом, мы видим, что подзапросом в данном случае будет являться оператор в круглых скобках. Этот подзапрос в нашем случае возвращает список значений ключа отношения R_2. И, как следует из нашей записи операторов, из анализа условия выборки, в результирующее отношение попадут только те кортежи отношения R_1, ключ которых содержится в списке ключей отношения R_2. То есть, в итоговом отношении, если вспомнить определение пересечения двух отношений, останутся лишь те кортежи, которые принадлежат обоим отношениям.

3. Операция разности.

Как уже было сказано ранее, унарная операция разности двух отношений реализуется аналогично операции пересечения. Здесь также, кроме главного запроса с оператором Select, используется второй, вспомогательный запрос — так называемый подзапрос.

Но в отличие от воплощения в жизнь предыдущей операции, при реализации операции разности необходимо использовать другое ключевое слово, а именно **not in**, что в дословном переводе означает «не в» или (как уместно перевести в нашем рассматриваемом случае) — «не содержится в».

Итак, пусть, как и в предыдущем примере, у нас имеются две схемы отношений (R_1 и R_2), приблизительно заданные:

R_1 (ключ, ...) и

R_2 (ключ, ...);

Как видим, среди атрибутов этих отношений снова заданы ключевые атрибуты.

Таким образом, получаем следующий вид для представления в языке структурированных запросов операции разности:

Select *

From R_1

Where ключ **not in**

(**Select** ключ **From** R_2);

Таким образом, в результирующее отношение выбираются только те кортежи отношения R_1, ключ которых не содержится в списке ключей отношения R_2. Если рассматривать запись буквально, то дей-

ствительно получается, что из отношения R_1 «вычли» отношение R_2. Отсюда делаем вывод, что условие выборки в этом операторе записано верно (ведь определение разности двух отношений выполняется) и использование ключей, как и в случае реализации операции пересечения, полностью оправдано.

Два случая применения «метода ключей», которые мы рассмотрели, являются самыми распространенными. На этом изучение использования ключей в составлении операторов, представляющих отношения, завершим. Все оставшиеся бинарные операции реляционной алгебры записываются иными способами.

4. **Операция декартова произведения.**

Как мы помним из предыдущих лекций, декартово произведение двух отношений-операндов составляется как набор всех возможных пар именованных значений кортежей на атрибутах. Поэтому на языке структурированных запросов операция декартова произведения реализовывается при помощи перекрестного соединения, обозначаемого ключевым словом **cross join**, что буквально и переводится «перекрестное объединение» или «перекрестное соединение».

Оператор Select в конструкции, представляющей операцию декартова произведения на языке структурированных запросов, присутствует только один и имеет следующий вид:

Select *

From R_1 **cross join** R_2

Здесь R_1 и R_2 — имена исходных отношений-операндов. Опция **cross join** обеспечивает, что в результирующее отношение запишутся все атрибуты (все, потому что в первой строчке оператора поставлен значок «*»), соответствующие всем парам кортежей отношений R_1 и R_2.

Очень важно помнить одну особенность воплощения в жизнь операции декартова произведения. Эта особенность является следствием определения бинарной операции декартова произведения. Напомним его:

$$r_4(S_4) = r_1(S_1) \times r_2(S_2) = \{t(S_1 \cup S_2) \mid t[S_1] \in r_1 \ \& \ t(S_2) \in r_2\},$$
$$S_1 \cap S_2 = \varnothing;$$

Как видно из приведенного определения, пары кортежей образуются при обязательно непересекающихся схемах отношений. Поэтому и при работе на языке структурированных запросов SQL непременно оговаривается, что исходные отношения-операнды не должны иметь совпадающих имен атрибутов. Но если эти отношения все же имеют одинаковые имена, сложившуюся ситуацию можно легко разрешить с помощью операции переименования атрибутов, т. е. в по-

добных случаях необходимо просто использовать опцию **as**, о которой упоминалось ранее.

Рассмотрим пример, в котором нужно найти декартово произведение двух отношений, имеющих некоторые имена своих атрибутов совпадающими. Итак, пусть даны следующие отношения:

R_1 (A, B),

R_2 (B, C);

Мы видим, что атрибуты R_1.B и R_2.B имеют одинаковые имена. С учетом этого оператор Select, реализующий на языке структурированных запросов эту операцию декартова произведения, будет выглядеть следующим образом:

Select A, R_1.B **as** B1, R_2.B **as** B2, C

From R_1 **cross join** R_2;

Таким образом, с использованием опции переименования as, у машины не возникнет «вопросов», по поводу совпадающих имен двух исходных отношений-операндов.

5. **Операции внутреннего соединения.**

На первый взгляд может показаться странным, что мы рассматриваем операцию внутреннего соединения раньше операции естественного соединения, ведь, когда мы проходили бинарные операции, все было наоборот. Но анализируя выражение операций на языке структурированных запросов, можно прийти к выводу, что операция естественного соединения является частным случаем операции внутреннего соединения. Именно поэтому рационально рассмотреть эти операции как раз в таком порядке.

Итак, для начала вспомним определение операции внутреннего соединения, которое мы проходили раньше:

$r_1(S_1) \times_P r_2(S_2) = \sigma_{<P>}(r_1 \times r_2),\ S_1 \cap S_2 = \varnothing.$

Для нас в этом определении особенно важно то, что рассматриваемые схемы отношений-операндов S_1 и S_2 не должны пересекаться.

Для реализации операции внутреннего соединения в языке структурированных запросов существует специальная опция **inner join**, которая и переводится с английского буквально «внутреннее объединения» или «внутреннее соединение».

Оператор Select в случае осуществления операции внутреннего соединения будет выглядеть следующим образом:

Select *

From R_1 **inner join** R_2;

Здесь, как и раньше, R_1 и R_2 — имена исходных отношений-операндов.

При реализации этой операции нельзя допускать пересечения схем отношений-операндов.

6. **Операция естественного соединения.**

Как мы уже говорили, операция естественного соединения является частным случаем операции внутреннего соединения. Почему? Да потому что при действии естественного соединения кортежи исходных отношений-операндов соединяются по особому условию. А именно по условию равенства кортежей на пересечении отношений-операндов, тогда как при действии операции внутреннего соединения такой ситуации допускать было бы нельзя.

Так как рассматриваемая нами операция естественного соединения является частным случаем операции внутреннего соединения, для ее реализации используется та же опция, что и для предыдущей рассмотренной операции, т. е. опция **inner join**. Но поскольку при составлении оператора Select для операции естественного соединения необходимо еще учесть условие равенства кортежей исходных отношений-операндов на пересечении их схем, то дополнительно к означенной опции применяется ключевое слово **on**. В переводе с английского, это буквально означает «на», а применительно к нашему смыслу, можно перевести как «при условии».

Общий вид оператора Select для выполнения операции естественного соединения следующий:

Select *

From *имя отношения 1* **inner join** *имя отношения 2*

on *условие равенства кортежей*;

Рассмотрим пример.

Пусть даны два отношения:

R_1 (A, B, C),

R_2 (B, C, D);

Операцию естественного соединения этих отношений можно реализовать с помощью следующего оператора:

Select A, $R_1.B$, $R_1.C$, D

From R_1 **inner join** R_2

on $R_1.B = R_2.B$ and $R_1.C = R_2.C$

В итоге этой операции в результат выведутся атрибуты, указанные в первой строке оператора Select, соответствующие кортежам, равным на указанном пересечении.

Следует заметить, что здесь мы обращаемся к общим атрибутам B и C не просто по именам. Это необходимо делать не по той причине, что и в случае реализации операции декартова произведения, а потому, что в противном случае будет не ясно, к какому отношению они относятся.

Интересно, что использованная формулировка условия соединения (R_1.B = R_2.B and R_1.C = R_2.C) предполагает, что общие атрибуты соединяемых отношений Null-значений не допускают. Это изначально встроено в систему языка структурированных запросов.

7. **Операция левого внешнего соединения.**

Выражение на языке структурированных запросов SQL операции левого внешнего соединения получается из реализации операции естественного соединения заменой ключевого слова **inner** на ключевое слово **left outer**.

Таким образом, на языке структурированных запросов эта операция запишется следующим образом:

Select *

From *имя отношения* 1 **left outer join** *имя отношения* 2

on *условие равенства кортежей*;

8. **Операция правого внешнего соединения.**

Выражение для операции правого внешнего соединения на языке структурированных запросов получается из осуществления операции естественного соединения заменой ключевого слова **inner** на ключевое слово **right outer**.

Итак, получаем, что на языке структурированных запросов SQL операция правого внешнего соединения запишется следующим образом:

Select *

From *имя отношения* 1 **right outer join** *имя отношения* 2

on *условие равенства кортежей*;

9. **Операция полного внешнего соединения.**

Выражение на языке структурированных запросов операции полного внешнего соединения получается, как и в двух предыдущих случаях, из выражения для операции естественного соединения путем замены ключевого слова **inner** на ключевое слово **full outer**.

Таким образом, на языке структурированных запросов эта операция запишется так:

Select *

From *имя отношения 1* **full outer join** *имя отношения 2*

on *условие равенства кортежей*;

Очень удобно, что в семантику языка структурированных запросов SQL изначально встроены эти опции, ведь иначе каждому программисту приходилось бы выводить их самостоятельно и вводить в каждую новую базу данных.

4. Использование подзапросов

Как можно было понять из пройденного материала, понятие «подзапрос» в языке структурированных запросов является понятием базовым и довольно широко применимым (иногда, кстати, их еще называют SQL-запросами. Действительно, практика программирования и работы с базами данных показывает, что составление системы подзапросов для решения различных сопутствующих задач — деятельность гораздо более благодарная по сравнению с какими-то другими приемами работы со структурированной информацией. Поэтому, рассмотрим пример для лучшего понимания действий с подзапросами, их составлением и использованием.

Пусть имеется следующий фрагмент некой базы данных, которая вполне может использоваться в каком-либо учебном заведении:

Предметы (Код предмета, Имя предмета);

Студенты (№ зачетной книжки, Фамилия, Имя, Отчество);

Сессия (Код предмета, № зачетной книжки, Оценка);

Сформулируем SQL-запрос, возвращающий ведомость с указанием номера зачетной книжки, фамилии и инициалов студента и оценки для предмета с наименованием «Базы данных». Такую информацию в университетах необходимо получать всегда и своевременно, поэтому приведенный далее запрос является едва ли не самой востребованной единицей программирования с использованием таких баз данных.

Для удобства работы, дополнительно предположим, что атрибуты «Фамилия», «Имя» и «Отчество» не допускают Null-значений и не являются пустыми. Это требование вполне объяснимо и закономерно, ведь в базу данных любого учебного заведения первыми из данных на нового ученика вводятся именно данные о его фамилии, имени и отчестве. И само собой разумеется, что не может быть записи в подобной базе данных, в которой присутствуют данные на ученика, но при этом неизвестно его имя.

Заметим, что атрибут «Имя предмета» схемы отношения «Предметы» является ключом, поэтому, как следует из определения (подробнее об этом будет сказано дальше), все наименования предметов являются уникальными. Это тоже понятно и без пояснения представления ключа, ведь все преподающиеся в учебном заведении предметы должны иметь и имеют различные имена.

Теперь, прежде чем мы приступим к составлению текста самого оператора, введем в рассмотрение две функции, которые нам пригодятся по мере нашей деятельности.

Во-первых, нам будет полезна функция **Trim**, записывается Trim (*«строка»*), т. е. аргументом этой функции является строка. Что делает эта функция? Они возвращает сам аргумент без пробелов, стоящих в начале и в конце этой строки, т. е., эту функцию применяют, например, в случаях: Trim («Богучарников») или Trim («Максименко»), когда после или до аргумента стоят по несколько лишних пробелов.

А во-вторых, необходимо также рассмотреть функцию Left, которая записывается Left (строка, число), т. е. функцию от уже двух аргументов, одним из которых является, как и раньше, строка. Второй ее аргумент — число, оно показывает, сколько символов из левой части строки следует вывести в результат.

Например, результатом операции:

Left («Михаил, 1») + «.» + Left («Зиновьевич, 1»)

будут инициалы «М. З.». Именно для выведения инициалов студентов мы и будем использовать эту функцию в нашем запросе.

Итак, приступим к составлению искомого запроса.

Для начала составим небольшой вспомогательный запрос, который потом используем в основном, главном запросе:

Select № *зачетной книжки, Оценка*
From *Сессия*
Where *Код предмета* = (***Select*** *Код предмета*
From *Предметы*
Where *Имя предмета = «Базы данных»*)
as *«Оценки «Базы данных»*;

Применение здесь опции as означает, что мы присвоили этому запросу псевдоним *«Оценки «Базы данных»*. Сделали мы это для удобства дальнейшей работы с этим запросом.

Далее, в этом запросе подзапрос:

Select *Код предмета*
From *Предметы*
Where *Имя предмета = «Базы данных»*;

позволяет выделить из отношения «Сессия» те кортежи, которые относятся к рассматриваемому предмету, т. е. к базам данных.

Интересно, что этот внутренний подзапрос может возвращать не более одного значения, так как атрибут «Имя предмета» является ключом отношения «Предметы», т. е. все его значения уникальны.

А весь запрос *«Оценки «Базы данных»* позволяет выделить из отношения «Сессия» данные о тех студентах (их номера зачетных книжек и оценки), которые удовлетворяют условию, оговоренному в подзапросе, т. е. информацию о предмете под названием «База данных».

Теперь составим основной запрос, используя уже полученные результаты.

Select *Студенты*. № *зачетной книжки*,
Trim (*Фамилия*) + « » + **Left** (*Имя*, 1) + «.» + **Left**
(*Отчество*, 1) + «.»**as** *ФИО*, *Оценки «Базы данных». Оценка*
From *Студенты* **inner join**
(
Select № *зачетной книжки, Оценка*
From *Сессия*
Where *Код предмета* = (**Select** *Код предмета*
From *Предметы*
Where *Имя предмета* = *«Базы данных»*)
) as *«Оценки «Базы данных».*
on *Студенты*. № *зачетной книжки* = *Оценки «Базы данных»*. № *зачетной книжки.*

Итак, сначала мы перечисляем атрибуты, которые будет необходимо вывести, после окончания работы запроса. Необходимо упомянуть, что атрибут «№ зачетной книжки» из отношения Студенты, оттуда же — атрибуты «Фамилия», «Имя» и «Отчество». Правда, два последних атрибута выводим не полностью, а только первые буквы. Также мы упоминаем атрибут «Оценка» из запроса *Оценки «Базы данных*, которое ввели раньше.

Выбираем мы все эти атрибуты из внутреннего соединения отношения «Студенты» и запроса *«Оценки «Базы данных»*. Это внутреннее соединение, как мы можем видеть, берется нами по условию равенства номеров зачетной книжки. В результате этой операции внутреннего соединения, к отношению «Студенты» добавляются оценки.

Надо заметить, что так как атрибуты «Фамилия», «Имя» и «Отчество» по условию не допускают Null-значений и не являются пустыми, то формула вычисления, возвращающая атрибут «ФИО» (**Trim** (Фамилия) + « » + **Left** (Имя, 1) + «.» + **Left** (Отчество, 1) + «.»**as** ФИО), соответственно не требует дополнительных проверок, упрощается.

ЛЕКЦИЯ № 7. Базовые отношения

Как мы уже знаем, базы данных — это как бы своеобразный контейнер, основное предназначение которого заключается в хранении данных, представленных в виде отношений.

Необходимо знать, что в зависимости от своей природы и структуры, отношения делятся на:

1) **базовые отношения;**

2) **виртуальные отношения.**

Отношения базового вида содержат только независимые данные и не могут быть выражены через какие-либо другие отношения баз данных.

В коммерческих системах управления базами данных базовые отношения обычно называются просто **таблицами** в отличие от представлений, соответствующих понятию виртуальных отношений. В данном курсе мы будем довольно подробно рассматривать только базовые отношения, основные приемы и принципы работы с ними.

1. Базовые типы данных

Типы данных, как и отношения, делятся на **базовые** и **виртуальные**.

(О виртуальных типах данных мы поговорим чуть позже, посвятим этой теме отдельную главу.)

Базовые типы данных — это любые типы данных, заданные в системах управления базами данных изначально, т. е. присутствующие там по умолчанию (в отличие от пользовательского типа данных, который мы проанализируем сразу после прохождения типа данных базового).

Прежде чем перейти к рассмотрению собственно базовых типов данных, перечислим, каких типов данные вообще бывают:

1) числовые данные;
2) логические данные;
3) строковые данные;
4) данные, определяющие дату и время;
5) идентификационные данные.

В системах управления базами данных по умолчанию ввели несколько наиболее распространенных типов данных, каждый из которых принадлежит какому-то из перечисленных типов данных.

Назовем их.

1. В **числовом** типе данных выделяют:

1) Integer. Этим ключевым словом обычно обозначают целый тип данных;

2) Real, соответствующий вещественному типу данных;

3) Decimal (n, m). Это десятичный тип данных. Причем в обозначении n — это число, фиксирующее общее количество знаков числа, а m показывает, сколько символов из них стоит после десятичной точки;

4) Money или Currency, введен специально для удобного представления данных денежного типа данных.

2. В **логическом** типе данных обычно выделяют только один базовый тип, это Logical.

3. **Строковый** тип данных насчитывает четыре базовых типа (имеются в виду, разумеется, наиболее распространенные):

1) Bit (n). Это строки бит с фиксированной длиной n;

2) Varbit (n). Это тоже строки бит, но с переменной длиной, не превышающей n бит;

3) Char (n). Это строки символов с постоянной длиной n;

4) Varchar (n). Это строки символов, с переменной длиной, не превышающей n символов.

4. Тип **дата и время** включает в себя следующие базовые типы данных:

1) Date — тип данных даты;

2) Time — тип данных, выражающих время суток;

3) Date-time — тип данных, выражающий одновременно и дату, и время.

5. **Идентификационный** тип данных содержит в себе только один включенный по умолчанию в систему управления базами данных тип, и это GUID (глобальный уникальный идентификатор).

Необходимо заметить, что все базовые типы данных могут иметь варианты различного по диапазону представления данных. Приведем пример: вариантами четырехбайтового типа данных integer могут быть восьмибайтовые (bigint) и двухбайтовые (smallint) типы данных.

Поговорим отдельно о базовом типе данных GUID. Этот тип предназначен для хранения шестнадцатибайтовых значений так называемого глобального уникального идентификатора. Все различные значения этого идентификатора генерируются автоматически при вызове специальной встроенной функции **NewId ()**. Это обозначение происходит от полного английского словосочетания New Identification, что

в переводе буквально и означает «новое значение идентификатора». Каждое генерируемое на конкретном компьютере значение идентификатора уникально в пределах всех производимых компьютеров.

GUID-идентификатор используется, в частности, для организации репликации баз данных, т. е. при создании копий каких-то уже имеющихся баз данных.

Такие GUID-идентификаторы могут быть использованы и разработчиками баз данных наравне с другими базовыми типами.

Промежуточное положение между типом GUID и другими базовыми типами занимает еще один специальный базовый тип — тип **счетчика**.

Для обозначения данных этого типа используется специальное ключевое слово **Counter (x_0, Δx)**, что в буквальном переводе с английского и означает «счетчик». Параметр x_0 задает начальное значение, а Δx — шаг приращения.

Значения этого типа Counter обязательно являются целочисленными.

Необходимо отметить, что работа с этим базовым типом данных включает в себя ряд очень интересных особенностей. Например, значения этого типа Counter не задаются, как мы привыкли при работе со всеми другими типами данных, они генерируются по требованию, почти как для значений типа глобального уникального идентификатора. Также необычно, что тип счетчика может быть задан только при определении таблицы и только тогда! В программном коде этот тип использовать нельзя. Еще нужно помнить, что и при определении таблицы тип счетчика может быть задан исключительно для одного столбца.

Значения данных типа счетчик генерируются автоматически при вставки строк. Причем эта генерация проводится без повторений, так что счетчик всегда будет уникально идентифицировать каждую строку. Но это создает некоторые неудобства при работе с таблицами, содержащими данные типа счетчик. Если, например, данные в отношении, заданном таблицей, изменятся и их придется удалить или поменять местами, значения счетчика легко могут «спутать карты», особенно если работает неопытный программист. Приведем пример, иллюстрирующий подобную ситуацию. Пусть дана следующая таблица, представляющая какое-то отношение, в которую введены четыре строки:

Счетчик типа Counter (1, 1)	**...**
1	...
2	...
3	...
4	...

Счетчик каждой новой строке автоматически дал уникальное имя.

И пусть теперь необходимо удалить вторую и четвертую строчки из таблицы, а потом добавить одну дополнительную строчку. Эти операции приведут к следующему преобразованию исходной таблицы:

Счетчик типа Counter (1, 1)	**…**
1	…
3	…
5	…

Таким образом, счетчик удалил вторую и четвертую строчки вместе с их уникальными именами, а не стал «переприсваивать» их новым строчкам, как можно было ожидать. Причем изменить вручную значение счетчика система управления базами данных никогда не позволит, так же как она не позволит объявить в одной таблице несколько счетчиков одновременно.

Обычно счетчик используется как суррогатный, т. е. искусственный ключ в таблице.

Интересно знать, что уникальных значений четырехбайтового счетчика при скорости генерации одно значение в секунду хватит более чем на 100 лет. Покажем, как это подсчитано:

1 год = 365 дней * 24 ч * 60 с * 60 с < 366 дней * 24 ч * 60 с * 60 с < 2^{25} с.

1 секунда > 2^{-25} год.

2^{4*8} значений / 1 значение/секунду = 2^{32} с > 2^{7} год > 100 лет.

2. Пользовательский тип данных

Пользовательский тип данных отличается от всех базовых типов тем, что он не был изначально вшит в систему управления базами данных, он не был описан как тип данных по умолчанию. Этот тип может создать для себя любой пользователь и программист баз данных в соответствии с собственными запросами и требованиями.

Таким образом, пользовательский тип данных — это подтип некоторого базового типа, т. е. это базовый тип с некоторыми ограничениями множества допустимых значений.

В записи на псевдокоде, пользовательский тип данных создается с помощью следующего стандартного оператора:

Create subtype *имя подтипа*

Type *имя базового типа*

As *ограничение подтипа*;

Итак, в первой строчке мы должны указать имя нашего нового, пользовательского типа данных, во второй — какой из имеющихся базовых типов данных мы взяли за образец, создавая собственный, и, наконец, в третьей — те ограничения, которые нам необходимо добавить в уже имеющиеся ограничения множества значений базового типа данных — образца. Ограничения подтипа записываются как условие, зависящее от имени определяемого подтипа.

Чтобы лучше понять принцип действия оператора Create, рассмотрим следующий пример. Пусть нам необходимо создать свой специализированный тип данных, допустим, для работы на почте. Это будет тип для работы с данными вида чисел почтового индекса. От обычных десятичных шестизначных чисел наши числа будут отличаться тем, что они могут быть только положительными. Запишем оператор, для создания нужного нам подтипа:

Create subtype *Почтовый индекс*

Type *decimal* (6, 0)

As *Почтовый индекс* > 0.

Почему мы взяли именно decimal (6, 0)? Вспоминая обычный вид индекса, мы видим, что такие числа должны состоять из шести целых чисел от нуля до девяти. Именно поэтому мы и взяли в качестве базового типа данных — десятичный тип.

Любопытно заметить, что в общем случае условие, накладываемое на базовый тип данных, т. е. ограничение подтипа, может содержать логические связки not, and, or и вообще быть выражением любой произвольной сложности. Определенные таким образом пользовательские подтипы данных могут беспрепятственно использоваться наряду с другими базовыми типами данных и в программном коде, и при определении типов данных в столбцах таблицы, т. е. базовые типы данных и пользовательские при работе с ними совершенно равноправны. В визуальной среде разработки они появляются в списках допустимых типов вместе с другими базовыми типами данных.

Вероятность того, что нам при проектировании новой собственной базы данных может понадобиться недокументированный (пользовательский) тип данных, достаточно велика. Ведь по умолчанию в систему управления базами данных вшиты только самые общие ти-

пы данных, пригодные соответственно для решения самых общих задач. При составлении предметных баз данных без проектирования собственных типов данных обойтись практически невозможно. Но, что любопытно, с равной вероятностью нам может понадобиться и удалить созданный нами подтип, чтобы не загромождать и не усложнять код. Для этого в системах управления базами данных обычно встроен специальный оператор **drop**, что и означает «удалить». Общий вид этот оператор удаления ненужных пользовательских типов имеет следующий:

Drop subtype *имя пользовательского типа*;

Пользовательские типы данных, как правило, рекомендуется вводить для подтипов достаточно общего назначения.

3. Значения по умолчанию

Системы управления базами данных могут иметь возможность создания любых произвольных значений по умолчанию или, как их еще называют, умолчаний. Эта операция в любой среде программирования имеет достаточно большой вес, ведь практически в любой задаче может возникнуть необходимость введения констант, неизменяемых значений по умолчанию.

Для создания умолчания в системах управления базами данных используется уже знакомая нам по прохождению пользовательского типа данных функция **Create**. Только в случае создания значения по умолчанию используется также дополнительное ключевое слово **default**, которое и означает «умолчание». Другими словами, чтобы создать в имеющейся базе данных значение по умолчанию, необходимо использовать следующий оператор:

Create default *имя умолчания*

As *константное выражение*;

Понятно, что на месте константного значения при применении этого оператора нужно написать значение или выражение, которое мы хотим сделать значением или выражением по умолчанию. И, разумеется, надо решить, под каким именем нам будет удобно его использовать в нашей базе данных, и записать это имя в первую строчку оператора.

Необходимо заметить, что в данном конкретном случае этот оператор Create отвечает синтаксису языка Transact-SQL, встроенному в систему Microsoft SQL Server.

Итак, что мы получили? Мы вывели, что умолчание представляет собой именованную константу, сохраняемую в базах данных, как и ее

объект. В визуальной среде разработки умолчания появляются в списке выделенных значений по умолчанию.

Приведем пример создания умолчания. Пусть для правильной работы нашей базы данных необходимо, чтобы в ней функционировало значение со смыслом неограниченного срока действия чего-либо. Тогда нужно ввести в список значений этой базы данных значение по умолчанию, отвечающему данному требованию. Это может быть необходимо хотя бы потому, что каждый раз при встрече в тексте кода с этим довольно громоздким выражением будет крайне неудобно выписывать его заново. Именно поэтому воспользуемся означенным выше оператором Create для создания умолчания, со смыслом неограниченного срока действия чего-либо.

Create default *«срок не ограничен»*
As '9999-12-31 23: 59:59'

Здесь также был использован синтаксис языка Transact-SQL, согласно которому значения констант типа «дата — время» (в данном случае, '9999-12-31 23: 59:59') записываются как строки символов определенного направления. Интерпретация строк символов как значений типа «дата — время» определяется контекстом использования этих строк. Например, в нашем конкретном случае, сначала в константной строчке записано предельное значение года, а потом времени.

Однако при всей своей полезности умолчания, как и пользовательский тип данных, иногда тоже могут требовать того, чтобы их удалили. В системы управления базами данных обычно есть специальный встроенный предикат, аналогичный оператору, удаляющему ненужный более пользовательский тип данных. Это предикат **Drop** и сам оператор выглядят следующим образом:

Drop default *имя умолчания*;

4. Виртуальные атрибуты

Все атрибуты в системах управления базами данных делятся (по абсолютной аналогии с отношениями) на базовые и виртуальные. Так называемые **базовые атрибуты** — это хранимые атрибуты, которые необходимо использовать не один раз, а следовательно, целесообразно сохранить. А, в свою очередь, **виртуальные атрибуты** — это не хранимые, а вычисляемые атрибуты. Что это значит? Это значит, что значения так называемых виртуальных атрибутов реально не хранятся, а вычисляются через базовые атрибуты на ходу посредством задаваемых формул. При этом домены вычисляемых виртуальных атрибутов определяются автоматически.

Приведем пример таблицы, задающей отношение, в которой два атрибута — обычные, базовые, а третий атрибут — виртуальный. Он будет вычисляться по специально введенной формуле.

...	Вес Кг	Цена Руб за Кг	Стоимость = Вес Кг * Цена Руб за Кг	...
...	2	10	20	...
...	3	20	60	...

Итак, мы видим, что атрибуты «Вес Кг» и «Цена Руб за Кг» — базовые атрибуты, потому что они имеют обыкновенные значения и хранятся в нашей базе данных. А вот атрибут «Стоимость» — виртуальный атрибут, потому что он задан формулой своего вычисления и реально в базе данных храниться не будет.

Интересно заметить, что в силу своей природы, виртуальные атрибуты не могут принимать значения по умолчанию, да и вообще, само понятие значения по умолчанию для виртуального атрибута лишено смысла, а следовательно, не применяется.

И еще необходимо знать, что, несмотря на то что домены виртуальных атрибутов определяются автоматически, тип вычисляемых значений иногда нужно заменить с имеющегося на какой-нибудь другой. Для этого в языке систем управления базами данных имеется специальный предикат Convert, с помощью которого и может быть переопределен тип вычисляемого выражения. Convert — это так называемая функция явного преобразования типов. Записывается она следующим образом:

Convert (*тип данных, выражение*);

Выражение, стоящее вторым аргументом функции Convert, посчитается и выведется в виде таких данных, тип которых указан первым аргументом функции.

Рассмотрим пример. Пусть нам необходимо посчитать значение выражения «2 * 2», но вывести это нужно не в виде целого числа «4», а строкой символов. Для выполнения этого задания запишем следующую функцию Convert:

Convert (Char (1), 2 * 2).

Таким образом, можно увидеть, что данная запись функции Convert в точности даст необходимый нам результат.

5. Понятие ключей

При объявлении схемы базового отношения могут быть заданы объявления нескольких ключей. С этим мы уже не раз сталкивались

прежде. Наконец настало время поговорить более подробно о том, что же такой ключи отношения, а не ограничиваться общими фразами и приближенными определениями.

Итак, дадим строгое определение ключа отношения.

Ключ схемы отношения — это подсхема исходной схемы, состоящая из одного или нескольких атрибутов, для которых декларируется **условие уникальности** значений в кортежах отношений. Для того чтобы понять, что такое условие уникальности или, как его еще называют, **ограничение уникальности**, вспомним для начала определения кортежа и унарной операции проекции кортежа на подсхему. Приведем их:

$t \equiv t(S) = \{t(a) \mid a \in def(t) \subseteq S\}$ — определение кортежа,

$t(S)[S'] = \{t(a) \mid a \in def(t) \cap S'\}, S' \subseteq S$ — определение унарной операции проекции;

Понятно, что проекции кортежа на подсхему соответствует подстрока строки таблицы.

Итак, что же такое ограничение уникальности ключевых атрибутов?

Объявление ключа K для схемы отношения S приводит к формулированию следующего инвариантного условия, называемого, как мы уже говорили, **ограничением уникальности** и обозначаемого как:

$\boldsymbol{Inv < K \rightarrow S > r(S)}$:

$Inv < K \rightarrow S > r(S) = \forall\, t_1, t_2 \in r\,(t_1[K] = t_2[K] \rightarrow t_1(S) = t_2(S))$,

$K \subseteq S$;

Итак, данное ограничение уникальности $Inv < K \rightarrow S > r(S)$ ключа K означает, что если любые два кортежа t_1 и t_2, принадлежащие отношению $r(S)$, равны в проекции на ключ K, то это непременно влечет за собой равенство этих двух кортежей и в проекции на всю схему отношения S. Другими словами, все значения кортежей, принадлежащих ключевым атрибутам, уникальны, единственны в своем отношении.

И второе важное требование, предъявляемое к ключу отношения, — это **требование неизбыточности**. Что это значит? Это требование означает, что ни для какого строгого подмножества ключа требование уникальности не предъявляется.

На интуитивном уровне понятно, что ключевой атрибут — это тот атрибут отношения, который однозначно и точно определяет каждый кортеж отношения. Например, в следующем отношении, заданном таблицей:

№ зачетной книжки	Фамилия	Имя	Отчество	...
...	...	...	...	...
...	...	...	...	...

Ключевым атрибутом является атрибут «№ зачетной книжки», потому что у различных студентов не может быть одинакового номера зачетной книжки, т. е. для этого атрибута выполняется ограничение уникальности.

Интересно, что в схеме любого отношения могут встретиться самые разные ключи. Перечислим основные виды ключей:

1) **простой ключ** — это ключ, состоящий из одного и не более атрибута.

Например, в экзаменационной ведомости по конкретному предмету простым ключом является номер зачетки, потому что по нему можно однозначно идентифицировать любого студента;

2) **составной ключ** — это ключ, состоящий из двух и более атрибутов.

Например, составным ключом в списке учебных аудиторий являются номер корпуса и номер аудитории. Ведь каким-то одним из этих атрибутов однозначно определить каждую аудиторию не представляется возможным, а их совокупностью, т. е. составным ключом, это сделать довольно легко;

3) **суперключ** — это любое надмножество любого ключа. Следовательно, сама схема отношения заведомо является суперключом. Из этого можно сделать вывод, что любое отношение теоретически имеет, как минимум, один ключ, а может иметь их и несколько. Однако объявление суперключа вместо обычного ключа логически недопустимо, так как связано с ослаблением автоматически контролируемого ограничения уникальности. Ведь супер ключ хоть и обладает свойством уникальности, но не обладает свойством неизбыточности;

4) **первичный ключ** — это просто ключ, который при задании базового отношения был объявлен первым. Важно, что допустимо объявление одного и только одного первичного ключа. Кроме того, атрибуты первичного ключа ни в коем случае не могут принимать Null-значений.

При создании базового отношения в записи на псевдокоде первичный ключ обозначается **primary key** и в скобках указывается имя атрибута, который и является этим ключом;

5) **кандидатные ключи** — это все остальные ключи, объявленные после первичного ключа.

В чем заключаются основные отличия кандидатных ключей от ключей первичных? Во-первых, кандидатных ключей может быть несколько, тогда как первичный ключ, как было сказано выше, может быть только один. И, во-вторых, если атрибуты первично-

го ключа не могут принимать Null-значений, то на атрибуты кандидатных ключей это условие не накладывается.

На псевдокоде при задании базового отношения кандидатные ключи объявляются при помощи слов **candidate key** и в скобках рядом, как и в случае объявления первичного ключа, указывается имя атрибута, который и является данным кандидатным ключом;

6) **внешний ключ** — это ключ, объявленный в базовом отношении, который при этом ссылается на первичный или кандидатный ключ того же самого или какого-то другого базового отношения.

При этом отношение, на которое ссылается внешний ключ, называется ссылочным (или **родительским)** отношением. А отношение, содержащее внешний ключ, называется **дочерним**.

В записи на псевдокоде внешний ключ обозначается как **foreign key**, в скобках непосредственно после этих слов указывается имя атрибута данного отношения, являющегося внешним ключом, а после этого записывается ключевое слово **references** («ссылается») и указать имя базового отношения и имя атрибута, на который и ссылается данный конкретный внешний ключ.

Также при создании базового отношения для каждого внешнего ключа записывается условие, называемое **ограничением ссылочной целостности**, но подробно мы будем говорить об этом позднее.

ЛЕКЦИЯ № 8. Создание базовых отношений

Предметом этой лекции будет довольно подробное рассмотрение оператора создания базового отношения. Мы подвергнуть разбору сам оператор в записи на псевдокоде, проанализируем все его составляющие и их работу, разберем способы модификации, т. е. изменения базовых отношений.

1. Металингвистические символы

При описании синтаксических конструкций, использующихся в записи оператора создания базового отношения на псевдокоде, применяются различные **металингвистические символы**. Это всевозможные открывающие и закрывающие скобки, разнообразные сочетания точек и запятых, словом, символы, несущие каждый свой смысл и облегчающие программисту задачу написания кода.

Введем в рассмотрение и поясним смысл основных металингвистических символов, наиболее часто использующихся при проектировании базовых отношений. Итак:

1) металингвистический символ «{}». Синтаксические конструкции в фигурных скобках представляют собой **обязательные** синтаксические единицы. При задании базового отношения, обязательными элементами являются, например, базовые атрибуты; без объявления базовых атрибутов ни одно отношение не может быть спроектировано. Поэтому при записи оператора создания базового отношения на псевдокоде базовые атрибуты перечисляются в фигурных скобках;

2) металингвистический символ «[]». В этом случае все наоборот: синтаксические конструкции в квадратных скобках представляют собой **необязательные** синтаксические элементы. Необязательными синтаксическими единицами в операторе создания базового отношения, в свою очередь, являются виртуальные атрибуты и первичный, и кандидатный, и внешний ключи. Здесь, разумеется, тоже присутствуют свои тонкости, но о них мы поговорим позднее, когда перейдем непосредственно к проектированию оператора создания базового отношения;

3) металингвистический символ «|». Этот символ буквально означает **«либо»**, как аналогичный символ в математике. Примене-

ние этого металингвистического символа означает, что необходимо выбрать между двумя или более конструкциями, разделенными, соответственно этим символом;

4) металингвистический символ «...». Многоточие, поставленное непосредственно после каких-либо синтаксических единиц, означает возможность **повторения** этих предшествующих металингвистическому символу синтаксических элементов;

5) металингвистический символ «,..». Этот символ означает почти тоже самое, что и предыдущий. Только в случае применения металингвистического символа «,..», **повторение** синтаксических конструкций происходит **через запятую**, что зачастую гораздо более удобно.

С учетом этого, можно говорить об эквивалентности следующих двух синтаксических конструкций:

единица [, единица] ...

и

единица,.. ;

2. Пример создания базового отношения в записи на псевдокоде

Теперь, когда мы выяснили значения основных металингвистических символов, использующихся при записи оператора создания базового отношения на псевдокоде, мы можем перейти собственно к рассмотрению самого этого оператора. Как можно было понять по упоминаниям выше, оператор создания базового отношения в записи на псевдокоде включает в себя объявления базовых и виртуальных атрибутов, первичного, кандидатных и внешних ключей. Кроме того, как будет показано и разъяснено выше, этот оператор охватывает также ограничения значений атрибутов и ограничения кортежей, а еще так называемые ограничения ссылочной целостности.

Первые два ограничения, а именно ограничение значения атрибута и ограничение кортежа, объявляются после специального зарезервированного слова **check**.

Ограничения ссылочной целостности могут быть двух видов: **on update**, что означает «при обновлении», и **on delete**, что означает «при удалении». Что это значит? Это значит, что при обновлении или при удалении атрибутов отношений, на которые ссылается внешний ключ, необходимо поддерживать целостность по состоянию. (Подробнее об этом мы поговорим позднее.)

Сам оператор создания базового отношения используется нами уже изученный — оператор **Create**, только для создания именно базового отношения добавляется ключевое слово **table** («отношение»). И, разумеется, так как отношение само по себе больше и включает в себя все рассмотренные ранее конструкции, а также новые дополнительные конструкции, оператор создания получится довольно внушительного вида.

Итак, запишем на псевдокоде общий вид оператора, используемого для создания базовых отношений:

Create table *имя базового отношения*
{*имя базового атрибута*
    тип значений базового атрибута
    check (*ограничение значения атрибута*)
    {Null | not Null}
    default (*значение по умолчанию*)
},..
[*имя виртуального атрибута*
    as (*формула вычисления*)
],..
[,**check** (*ограничение кортежа*)]
[,**primary key** (*имя атрибута*,..)]
[,**candidate key** (*имя атрибута*,..)]...
[,**foreign key** (*имя атрибута*,..) **references** *имя ссылочного отношения* (*имя атрибута*,..)
    on update {Restrict | Cascade | Set Null}
    on delete {Restrict | Cascade | Set Null}
]...

Итак, мы видим, что базовых и виртуальных атрибутов, кандидатных и внешних ключей может быть объявлено несколько, так как после соответствующих синтаксических конструкций стоит металингвистический символ «,..». После объявления первичного ключа этого символа нет, потому что базовые отношения, как уже было сказано ранее, допускают наличие только одного первичного ключа.

Далее рассмотрим подробнее механизм объявления **базовых атрибутов**.

При описании в операторе создания базового отношения любого атрибута в общем случае задаются его имя, тип, ограничения его значений, флажок допустимости Null-значений и значения по умолча-

нию. Нетрудно понять, что тип атрибута и ограничения его значений определяют его домен, т. е. буквально множество допустимых значений данного конкретного атрибута. **Ограничение значений атрибута** записывается как условие, зависящее от имени атрибута. Вот небольшой пример для облегчения понимания этого материала:

Create table *имя базового отношения*
Курс
integer
check (1 <= Курс and Курс <= 5;

Здесь условие «1 <= Курс and Курс <= 5» вместе с определением целого типа данных действительно полностью обусловливают множество допустимых значений атрибута, т. е. буквально его домен.

Флажок допустимости Null-значений (Null | not Null) запрещает (not Null) или, наоборот, разрешает (Null) появление Null-значений среди значений атрибутов.

Если взять рассмотренный только что пример, то механизм применения флажков допустимости Null-значений выглядит следующим образом:

Create table *имя базового отношения*
Курс
integer
check (1 <= Курс and Курс <= 5);
Not Null;

Итак, порядковый номер курса студента никогда не может принимать Null-значения, не может быть неизвестным составителям базы данных и не может не существовать.

Значения по умолчанию (**default** (*значение по умолчанию*)) используются при вставке кортежа в отношения, если в операторе вставки значения атрибутов явно не заданы.

Интересно заметить, что значения по умолчанию могут быть и Null-значениями, если только Null-значения для данного конкретного атрибута объявлены допустимыми.

Теперь рассмотрим определение **виртуального атрибута** в операторе создания базового отношения.

Как мы уже говорили ранее, задание виртуального атрибута заключается в задании формул его вычисления через другие базовые атрибуты. Рассмотрим пример объявления виртуального атрибута «Стоимость Руб.» в виде формулы, зависящей от базовых атрибутов «Вес Кг» и «Цена Руб. за Кг».

Create table *имя базового отношения*
Вес Кг
тип значений базового атрибута Вес Кг
check (*ограничение значения атрибута Вес Кг*)
not Null
default (*значение по умолчанию*)
Цена Руб. за Кг
тип значений базового атрибута Цена Руб. за Кг
check (*ограничение значения атрибута Цена Руб. за Кг*)
not Null
default (*значение по умолчанию*)
...
Стоимость Руб.
as (Вес Кг * Цена Руб. за Кг)

Чуть выше мы рассмотрели ограничения атрибутов, которые записывались как условия, зависящие от имен атрибутов. Теперь рассмотрим второй вид ограничений, объявляемых при создании базового отношения, а именно **ограничения кортежа**.

Что такое ограничение кортежа, чем оно отличается от ограничения атрибута? Ограничение кортежа тоже записывается как условие, зависящее от имени базового атрибута, но только в случае ограничения кортежа, условие может зависеть от нескольких имен атрибутов одновременно.

Рассмотрим пример, иллюстрирующий механизм работы с ограничениями кортежей:

Create table *имя базового отношения*
min Вес Кг
тип значений базового атрибута min Вес Кг
check (*ограничение значения атрибута min Вес Кг*)
not Null
default (*значение по умолчанию*)
max Вес Кг
тип значений базового атрибута max Вес Кг
check (*ограничение значения атрибута max Вес Кг*)
not Null
default (*значение по умолчанию*)
check (0 < min Вес Кг **and** min Вес Кг < max Вес Кг);

Таким образом, применение ограничения к кортежу сводится к подстановке значений кортежа вместо имен атрибутов.

Продвигаемся далее в рассмотрении оператора создания базового отношения. После объявления базовых и виртуальных атрибутов мо-

гут быть объявлены, а могут и не объявляться **ключи**: первичный, кандидатные и внешние.

Как мы уже говорили ранее, подсхема базового отношения, которой в другом (или в том же самом) базовом отношении соответствует первичный или кандидатный ключ в контексте первого отношения называется **внешним ключом**. Внешние ключи представляют **механизм ссылок** кортежей одних отношений на кортежи других отношений, т. е. есть объявления внешних ключей связаны с навязыванием уже упоминавшихся так называемых **ограничений ссылочной целостности**. (Этим ограничением будет посвящена вся следующая лекция, поскольку целостность по состоянию (т. е. целостность, обеспечиваемая ограничениями целостности) — это крайне важное условие успешного функционирования базового отношения и всей базы данных.)

Объявление первичных и кандидатных ключей, в свою очередь, навязывает схеме базового отношения соответствующие ограничения уникальности, о который мы уже говорили ранее.

И, наконец, следует сказать о возможности удаления базового отношения. Нередко в практике проектирования баз данных необходимо удалить старой ненужное отношение, чтобы не загромождать программный код. Сделать это можно с помощью уже знакомого нам оператора **Drop**. В полном общем виде оператор удаления базового отношения выглядит следующим образом:

Drop table *имя базового отношения*;

3. Ограничение целостности по состоянию

Ограничение целостности реляционного объекта данных **по состоянию** — это так называемый инвариант данных.

При этом целостность следует уверенно отличать от безопасности, которая, в свою очередь, подразумевает защиту данных от несанкционированного доступа к ним с целью раскрытия, изменения или разрушения данных.

В общем случае ограничения целостности реляционных объектов данных классифицируются **по уровням иерархии** этих самых реляционных объектов данных (иерархия реляционных объектов данных — это последовательность вложенных друг в друга понятий: «атрибут — кортеж — отношение — база данных»).

Что это означает? Это означает, что ограничения целостности зависят:

1) на уровне атрибута — от значений атрибута;

2) на уровне кортежа — от значений кортежа, т. е. от значений нескольких атрибутов;

3) на уровне отношений — от отношения, т. е. от нескольких кортежей;

4) на уровне базы данных — от нескольких отношений.

Итак, теперь нам остается только рассмотреть более подробно ограничения целостности по состоянию каждого из приведенных понятий. Но прежде дадим понятия процедурной и декларативной поддержки ограничений целостности по состоянию.

Итак, поддержка ограничений целостности может быть двух видов:

1) **процедурной**, т. е. созданной при помощи написания программного кода;

2) **декларативной**, т. е. созданной путем объявлений тех или иных ограничений для каждого из названных выше вложенных понятий.

Декларативная поддержка ограничений целостности реализуется в контексте оператора Create создания базового отношения. Поговорим об этом подробнее. Начнем рассмотрение совокупности ограничений снизу нашей иерархической лестницы реляционных объектов данных, т. е. с понятия атрибута.

Ограничение уровня атрибута включает в себя:

1) ограничения типа значений атрибута.

Например, условие целочисленности значений, т. е. условие integer для атрибута «Курс» из одного из рассмотренного ранее базового отношения;

2) ограничение значений атрибута, записываемое как условие, зависящее от имени атрибута.

Например, анализируя то же самое базовое отношение, что и в предыдущем пункте, видим, что в том отношении имеется и ограничение значений атрибута с использованием опции **check**, т. е.:

check (1 <= Курс and Курс <= 5);

3) ограничение уровня атрибутов включает в себя ограничения Null-значений, определяемые уже знакомым нам флажком допустимости (Null) или, наоборот, недопустимости (not Null) Null-значений.

Как мы уже упоминали ранее, первые два ограничения определяют ограничение домена атрибута, т. е. значение его множества определения.

Далее согласно иерархической лестнице реляционных объектов данных, нужно говорить о кортежах. Итак, **ограничение уровня кортежа** сводится к ограничению кортежа и записывается как условие, зависящее от имен нескольких базовых атрибутов схемы отношения, т. е., это ограничение целостности по состоянию значительно меньше и проще аналогичного, только соответствующего атрибуту.

И снова целесообразно будет вспомнить пройденный нами ранее пример базового отношения, имеющего необходимое нам сейчас ограничение кортежа, а именно:

check (0 < min Вес Кг **and** min Вес Кг < max Вес Кг);

И, наконец, последнее значимое в контексте ограничения целостности по состоянию понятие — это понятие уровня отношений. Как мы уже говорили раньше, **ограничение уровня отношения** включает в себя ограничение значений первичного (**primary key**) и кандидатного (**candidate key**) ключей.

Любопытно, что ограничения, накладываемые на базы данных, относятся уже не к ограничениям целостности по состоянию, а к ограничениям ссылочной целостности.

4. Ограничения ссылочной целостности

Итак, ограничение уровня баз данных включает ограничение ссылочной целостности внешних ключей (**foreign key**). Коротко мы уже упоминали об этом, когда говорили об ограничениях ссылочной целостности при создании базового отношения и внешних ключах. Теперь настало время поговорить об этом понятии более подробно.

Как мы уже говорили раньше, внешний ключ объявляемого базового отношения ссылается на первичный или кандидатный ключ какого-то другого (чаще всего) базового отношения. Напомним, что при этом отношение, на которое ссылается внешний ключ, называется **ссылочным** или **родительским**, потому что оно как бы «порождает» один атрибут или несколько атрибутов в ссылающемся базовом отношении. А, в свою очередь, отношение, содержащее внешний ключ, называется **дочерним**, тоже по вполне понятным причинам.

В чем же заключается **ограничение ссылочной целостности**? А заключается оно в том, что каждому значению внешнего ключа дочернего отношения обязательно должно соответствовать значение какого-либо ключа отношения родительского, если только значение внешнего ключа не содержит Null-значений в каких-либо атрибутах.

Кортежи дочернего отношения, нарушающие это условие, называются **висящими**.

Действительно, если внешний ключ дочернего отношения ссылается на атрибут, которого на самом деле в родительском отношении нет,

то он не ссылается ни на что. Именно таких ситуаций и требуется всячески избегать, это и означает поддерживать ссылочную целостность.

Но, зная, что ни одна база данных никогда не допустит создания висящего кортежа, разработчики следят за тем, чтобы изначально висящих кортежей в базе данных не было и все имеющиеся ключи ссылались на вполне реальный атрибут отношения родительского. Но тем не менее бывают ситуации, когда висящие кортежи образуются уже в процессе функционирования базы данных. Что это за ситуации? Известно, что при удалении кортежей из родительского отношения или при обновлении значения ключа кортежа родительского отношения ссылочная целостность может нарушиться, т. е. могут возникнуть висящие кортежи.

Для исключения возможности их появления при объявлении значения внешнего ключа задается одно из **трех** имеющихся **правил** поддержания ссылочной целостности, применяемых соответственно при обновлении значения ключа в родительском отношении (т. е., как мы уже упоминали раньше, **on update**) или при удалении кортежа из родительского отношения (**on delete**). Необходимо отметить, что добавление нового кортежа в родительское отношение не может нарушить ссылочную целостность по вполне понятным причинам. Ведь, если этот кортеж только что добавили в базовое отношение, раньше на него не мог ссылаться ни один атрибут по причине его отсутствия!

Итак, что же это за три правила, применяющиеся для поддержания в базах данных ссылочной целостности? Перечислим их.

1. **Restrict,** или **правило ограничения**. Если мы при задании нашего базового отношения, при объявлении внешних ключей в ограничении ссылочной целостности применили это правило ее поддержания, то обновление ключа в родительском отношении или удаление кортежа из родительского отношения просто не может быть выполнено в том случае, если на этот кортеж ссылается хотя бы один кортеж дочернего отношения, т. е. операция **Restrict** банально запрещает производить какие-либо действия, могущие привести к появлению висящих кортежей.

Проиллюстрируем применение этого правила следующим примером.

Пусть даны два отношения:

Родительское отношение

...	Primary_key
...	1
...	2
...	3
...	4

Дочернее отношение

Foreign_key	**...**
2	...
2	...
Null	...
3	...
~~100~~	~~...~~

Мы видим, что кортежи дочернего отношения (2, ...) и (2, ...) ссылаются на кортеж (..., 2) родительского отношения, а кортеж (3, ...) дочернего отношения ссылается на кортеж (..., 3) родительского отношения. Кортеж (100, ...) дочернего отношения является висящим, он недопустим.

Здесь только кортежи родительского отношения (..., 1) и (..., 4) допускают обновление значений ключа и удаление кортежей, потому что на них не ссылается ни один из внешних ключей дочернего отношения.

Составим оператор создания базового отношения, включающего в себя объявление всех вышеназванных ключей:

Create table *Родительское отношение*
Primary_key
    Integer
    not Null
primary key (Primary_key)

Create table *Дочернее отношение*
Foreign_key
    Integer
    Null
foreign key (Foreign_key) **references** *Родительское отношение* (Primary_key)
    on update Restrict
    on delete Restrict

2. **Cascade,** или **правило каскадной модификации**. Если при объявлении внешних ключей в нашем базовом отношении мы использовали правило поддержания ссылочной целостности **Cascade**, то обновление ключа в родительском отношении или удаление кортежа из родительского отношения вызывает автоматическое обновление или удаление соответствующих ключей и кортежей дочернего отношения.

Рассмотрим пример, чтобы лучше понять механизм работы правила каскадной модификации. Пусть даны уже знакомые нам базовые отношения из предыдущего примера:

Родительское отношение

...	**Primary_key**
...	1
...	2
...	3
...	4

и

Дочернее отношение

Foreign_key	...
2	...
2	...
Null	...
3	...

Допустим, мы в таблице, задающей отношение «Родительское отношение» обновим некоторые кортежи, а именно заменим кортеж (..., 2) на кортеж (..., 20), т. е. получим новое отношение:

Родительское отношение

...	**Primary_key**
...	1
...	20
...	3
...	4

И пусть при этом в операторе создания нашего базового отношения «Дочернее отношение» при объявлении внешних ключей мы использовали правило поддержания ссылочной целостности **Cascade**, т. е. оператор создания наших базовых отношений выглядит следующим образом:

Create table Родительское отношение
Primary_key
    Integer
    not Null
primary key (Primary_key)

Create table *Дочернее отношение*
Foreign_key
Integer
Null
foreign key (Foreign_key) **references** *Родительское отношение*
(Primary_key)
on update Cascade
on delete Cascade

Тогда что же произойдет с отношением дочерним при обновлении родительского отношения указанным выше образом? Оно примет следующий вид:

Дочернее отношение

Foreign_key	**...**
20	...
20	...
Null	...
3	...

Таким образом, действительно, правило **Cascade** обеспечивает каскадное обновление всех кортежей дочернего отношения в ответ на обновления отношения родительского.

3. **Set Null,** или **правило присвоения Null-значений**. Если же мы в операторе создания нашего базового отношения при объявлении внешних ключей применяем правило поддержания ссылочной целостности **Set Null,** то обновление ключа родительского отношения или удаление кортежа из родительского отношения влечет за собой автоматическое присвоение Null-значений тем атрибутам внешнего ключа дочернего отношения, который Null-значения допускают. Следовательно, правило применимо, если такие атрибуты имеются.

Рассмотрим пример, который мы уже использовали ранее. Пусть нам даны два базовых отношения:

«Родительское отношение»

...	**Primary_key**
...	1
...	2
...	3
...	4

Дочернее отношение

Foreign_key	**...**
2	...
2	...
Null	...
3	...

Как можно заметить, атрибуты дочернего отношения допускают Null-значения, следовательно, правило **Set Null** в данном конкретном случае применимо.

Допустим теперь, что из родительского отношения был удален кортеж (..., 1), а кортеж (..., 2) обновлен, как и в предыдущем примере. Таким образом, родительское отношение принимает следующий вид:

Родительское отношение

...	**Primary_key**
...	20
...	3
...	4

Тогда с учетом того, что при объявлении внешних ключей дочернего отношения нами применялось правило поддержания ссылочной целостности **Set Null**, дочернее отношение примет следующий вид:

Дочернее отношение

Foreign_key	**...**
Null	...
Null	...
Null	...
3	...

На кортеж (..., 1) не ссылался ни один ключ дочернего отношения, поэтому его удаление не влечет за собой никаких последствий.

Сам оператор создания базового отношения с использованием правила **Set Null** при объявлении внешних ключей отношения выглядит следующим образом:

Create table Родительское отношение
Primary_key
    Integer
    not Null
primary key (Primary_key)

Create table Дочернее отношение
Foreign_key
    Integer
    Null
foreign key (Foreign_key) **references** Родительское отношение
(Primary_key)
    on update Set Null
    on delete Set Null

Итак, мы видим, что наличие трех различных правил поддержания ссылочной целостности обеспечивают то, что во фразах **on update** и **on delete** функции могут быть разными.

Необходимо помнить и понимать, что вставка кортежей в дочернее отношение или обновление значений ключа дочерних отношений не будут выполнены, если это будет приводить к нарушению ссылочной целостности, т. е. к появлению так называемых висящих кортежей. Удаление же кортежей из дочернего отношения ни при каких условиях не может привести к нарушению ссылочной целостности.

Интересно, что дочернее отношение одновременно может выступать и родительским со своими правилами поддержания ссылочной целостности, если внешние ключи других базовых отношений ссылаются на какие-то его атрибуты, как на первичные ключи.

Если у программистов возникает желание обеспечить выполнение ссылочной целостности какими-то отличными от приведенных стандартных правил, то процедурная поддержка таких нестандартных правил поддержания ссылочной целостности обеспечивается с помощью так называемых триггеров. К сожалению, подробное рассмотрение этого понятия не сходит в наш курс лекций.

5. Понятие индексов

Создание ключей в базовых отношениях автоматически связано с созданием индексов.

Дадим определение понятия индекса.

Индекс — это системная структура данных, в которой размещается обязательно упорядоченный перечень значений какого-либо клю-

ча со ссылками на те кортежи отношения, в которых эти значения встречаются.

Индексы в системах управления базами данных бывают двух видов:

1) **простые.**

Простой индекс берется для подсхемы схемы базового отношения из одного атрибута;

2) **составные.**

Соответственно составной индекс — это индекс для подсхемы, состоящей из нескольких атрибутов.

Но, кроме деления на простые и составные индексы, в системах управления базами данных существует деление индексов на уникальные и неуникальные. Итак:

1) **уникальные** индексы — это индексы, ссылающиеся не более чем на один атрибут.

Уникальные индексы, как правило, соответствуют первичному ключу отношения;

2) **неуникальные** индексы — это индексы, могущие соответствовать нескольким атрибутам одновременно.

Неуникальные ключи, в свою очередь, чаще всего соответствуют внешним ключам отношения.

Рассмотрим пример, иллюстрирующий деление индексов на уникальные и неуникальные, т. е. рассмотрим следующие отношения, заданные таблицами:

Индекс	**Primary key**
‘a’	2
‘b’	4
‘c’	1
‘d’	3

№	**Primary_key**	**…**	**Foreign key**
‘a’	‘c’	...	20
‘b’	‘a’	...	Null
‘c’	‘b’	...	20
‘d’	‘d’	...	10

Индекс	**Foreign key**
Null	2
10	4
20	1, 3

Здесь соответственно Primary key — первичный ключ отношения, Foreign key — внешний ключ. Понятно, что в этих отношениях, индекс атрибута Primary key — уникальный, так как он соответствует первичному ключу, т. е. одному атрибуту, а индекс атрибута Foreign key — неуникальный, ведь он соответствует ключам внешним. И его значение «20» соответствует одновременно первой и третьей строкам таблицы-отношения.

Но иногда индексы могут создаваться без отношения к ключам. Это делается в системах управления базами данных для поддержки производительности операций сортировки и поиска.

Например, дихотомический поиск значения индекса в кортежах будет реализован в системах управления базами данных за двадцать итераций. Откуда получены эти сведения? Они были получены путем несложных вычислений, т. е. следующим образом:

$10^6 = (10^3)^2 = 2^{20}$;

Создаются индексы в системах управления базами данных при помощи уже известного нам оператора Create, но только с добавлением ключевого слова index. Выглядит такой оператор следующим образом:

Create index *имя индекса*

On *имя базового отношения (имя атрибута,..)*;

Здесь мы видим знакомый нам металингвистический символ «,..», обозначающий возможность повтора аргумента через запятую, т. е. в этом операторе может быть создан индекс, соответствующий нескольким атрибутам.

Если требуется объявить уникальный индекс, перед словом index добавляют ключевое слово unique, и тогда весь оператор создания в базовом отношении индекса принимает следующий вид:

Create unique index *имя индекса*

On *имя базового отношения (имя атрибута)*;

Тогда в самом общем виде, если вспомнить правило обозначения необязательных элементов (металингвистический символ []), оператор создания индекса в базовом отношении будет выглядеть следующим образом:

Create [unique] index *имя индекса*

On *имя базового отношения (имя атрибута,..)*;

Если требуется удалить из базового отношения уже имеющийся индекс, используют оператор Drop, также уже известный нам:

Drop index {*имя базового отношения. Имя индекса*},.. ;

Почему здесь используется уточненное имя индекса «*имя базового отношения. Имя индекса*»? В операторе удаления индекса всегда используется его уточненное имя, потому что имя индекса должно быть уникальным в пределах одного отношения, но не больше.

6. Модификация базовых отношений

Для успешной и продуктивной работы с различными базовыми отношениями очень часто разработчикам необходимо каким-либо образом модифицировать это базовые отношения.

Какие основные необходимые варианты модификации встречаются чаще всего в практике проектирования баз данных? Перечислим их:

1) вставка кортежей.

Очень часто нужно в уже сформированное базовое отношение вставить новые кортежи;

2) обновление значений атрибутов.

А необходимость этой модификации в практике программирования встречается еще чаще, чем предыдущая, ведь при поступлении новой информации об аргументах вашей базы данных неминуемо придется какую-то старую информацию обновлять;

3) удаление кортежей.

И с примерно равной вероятностью возникает необходимость удалить из базового отношения те кортежи, присутствие которых в вашей базе данных более не требуется в силу новой поступившей информации.

Итак, мы обозначили основные моменты модификации базовых отношений. Как же можно достичь каждой из поставленных целей?

В системах управления базами данных чаще всего существуют встроенные, базовые операторы модификации отношений. Дадим их описание в записи на псевдокоде:

1) **оператор вставки** в базовое отношение новых кортежей. Это оператор **Insert**. Выглядит он следующим образом:

Insert into *имя базового отношения* (*имя атрибута,..*)

Values (*значение атрибута*,..);

Металингвистический символ «,..», поставленный после имени атрибута и значения атрибута, говорит нам, что этот оператор допускает одновременное добавление нескольких атрибутов в базовое отношение. В этом случае необходимо имена атрибутов и значения атрибутов перечислять через запятую в согласованном порядке.

Ключевое слово **into** в сочетании с общим названием оператора **Insert** означает «вставить в» и показывает, в какое отношение необходимо вставить указанные в скобках атрибуты.

Ключевое слово **Values** в этом операторе и означает «значения», «величины», которые и присваиваются этим вновь объявленным атрибутам;

2) теперь рассмотрим **оператор обновления** значений атрибутов в базовом отношении. Этот оператор называется **Update**, что в пе-

реводе с английского и означает буквально «обновить». Дадим полный общий вид этого оператора в записи на псевдокоде и расшифруем ее:

Update *имя базового отношения*
Set {*имя атрибута — значение атрибута*},..
Where *условие*;

Итак, в первой строчке оператора после ключевого слова **Update** записывается имя базового отношения, в котором необходимо произвести обновления.

Ключевое слово Set переводится с английского «задать», и в этой строчке оператора указываются имена атрибутов, которые необходимо обновить, и соответствующие новые значения атрибутов. В одном операторе можно обновить сразу несколько атрибутов, что следует из применения металингвистического символа «,..».

В третьей строке после ключевого слова **Where** записывается условие, показывающее, какие именно атрибуты данного базового отношения необходимо обновить;

3) оператор **Delete**, позволяющий **удалять** какие-либо кортежи из базового отношения. Запишем его полный вид на псевдокоде и разъясним значение всех отдельных синтаксических единиц:

Delete from *имя базового отношения*
Where *условие*;

Ключевое слово **from** в сочетании с названием оператора **Delete** переводится как «удалить из». И после этих ключевых слов в первой строчке оператора указывается имя базового отношения, из которого необходимо удалить какие-либо кортежи.

А во второй строчке оператора после ключевого слова **Where** («где») указывается условие, по которому отбираются кортежи, более не требующиеся в нашем базовом отношении.

ЛЕКЦИЯ № 9. Функциональные зависимости

1. Ограничение функциональной зависимости

Ограничения уникальности, накладываемые объявлениями первичного и кандидатных ключей отношения, является частным случаем ограничений, связанных с понятием **функциональных зависимостей**.

Для объяснения понятия функциональной зависимости, рассмотрим следующий пример.

Пусть нам дано отношение, содержащее данные о результатах какой-то одной конкретной сессии. Схема этого отношения выглядит следующим образом:

Сессия (№ зачетной книжки, Фамилия, Имя, Отчество, Предмет, Оценка);

Атрибуты «№ зачетной книжки» и «Предмет» образуют составной (так как ключом объявлены два атрибута) первичный ключ этого отношения. Действительно, по двум этим атрибутам можно однозначно определить значения всех остальные атрибутов.

Однако, помимо ограничения уникальности, связанной с этим ключом, на отношение непременно должно быть наложено то условие, что одна зачетная книжка выдается обязательно одному конкретному человеку и, следовательно, в этом отношении кортежи с одинаковым номером зачетной книжки должны содержать одинаковые значения атрибутов «Фамилия», «Имя» и «Отчество».

№ зачетной книжки	Фамилия	Имя	Отчество	Предмет	Оценка
100	Тюряпина	Елизавета	Николаевна	Информатика	Удовлетворительно
…	…	…	…	…	…
1100	Михайлова	Анастасия	Александровна	Базы данных	Отлично
…	…	…	…	…	…
100	Тюряпина	Елизавета	Николаевна	Литература	Отлично

Если у нас имеется следующий фрагмент какой-то определенной базы данных студентов учебного заведения после какой-то сессии, то

в кортежах с номером зачетной книжки 100, атрибуты «Фамилия», «Имя» и «Отчество» совпадают, а атрибуты «Предмет» и «Оценка» — не совпадают (что и понятно, ведь в них речь идет о разных предметах и успеваемости по ним). Это значит, что атрибуты «Фамилия», «Имя» и «Отчество» **функционально зависят** от атрибута «№ зачетной книжки», а атрибуты «Предмет» и «Оценка» функционально не зависят.

Таким образом, **функциональная зависимость** — это однозначная зависимость, затабулированная в системах управления базами данных.

Теперь дадим строгое определение функциональной зависимости.

Определение: пусть X, Y — подсхемы схемы отношения S, определяющие над схемой S **схему функциональной зависимости X → Y** (читается «X стрелка Y»). Определим **ограничения функциональной зависимости inv < X → Y >** как утверждение о том, что в отношении со схемой S любые два кортежа, совпадающие в проекции на подсхему X, должны совпадать и в проекции на подсхему Y.

Запишем это же определение в формулярном виде:

$\mathbf{Inv < X \rightarrow Y >} r(S) = t_1, t_2 \in r\,(t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y])$,

$X, Y \subseteq S$;

Любопытно, что в этом определении использовано понятие унарной операции проекции, с которым мы сталкивались раньше. Действительно, как еще, если не использовать эту операцию, показать равенство друг другу двух столбцов таблицы-отношения, а не строк? Поэтому мы и записали в терминах этой операции, что совпадение кортежей в проекции на какой-то атрибут или несколько атрибутов (подсхему X) непременно влечет за собой совпадение этих же столбцов-кортежей и на подсхеме Y в том случае, если Y функционально зависит от X.

Интересно заметить, что в случае функциональной зависимости Y от X, говорят также, что X **функционально определяет** Y или что Y **функционально зависит** от X. В схеме функциональной зависимости X → Y подсхема X называется левой частью, а подсхема Y — правой частью.

На практике проектирования баз данных на схему функциональной зависимости для краткости обычно ссылаются как на функциональную зависимость.

Конец определения.

В частном случае, когда правая часть функциональной зависимости, т. е. подсхема Y, совпадает со всей схемой отношения, ограничение функциональной зависимости переходит в ограничение уникальности первичного или кандидатного ключа. Действительно:

$Inv < K \rightarrow S > r(S) = \forall\, t_1, t_2 \in r\,(t_1[K] = t_2[K] \rightarrow t_1(S) = t_2(S))$,

$K \subseteq S$;

Просто в определении функциональной зависимости вместо подсхемы X нужно взять обозначение ключа K, а вместо правой части функциональной зависимости, подсхемы Y взять всю схему отношений S, т. е., действительно, ограничение уникальности ключей отношений является частным случаем ограничения функциональной зависимости при равенстве правой части схемы функциональной зависимости всей схеме отношения.

Приведем примеры изображения функциональной зависимости:

{№ зачетной книжки} $\rightarrow$ {Фамилия, Имя, Отчество};

{№ зачетной книжки, Предмет} $\rightarrow$ {Оценка};

2. Правила вывода Армстронга

Если какое-либо базовое отношение удовлетворяет векторно определенным функциональным зависимостям, то с помощью различных специальных правил вывода можно получить другие функциональные зависимости, которым данное базовое отношение будет заведомо удовлетворять.

Хорошим примером таких специальных правил являются правила вывода Армстронга.

Но прежде чем приступать к анализу самих правил вывода Армстронга, введем в рассмотрение новый металингвистический символ «$\vdash$», который называется **символом метаутверждения о выводимости**. Этот символ при формулировании правил записывается между двумя синтаксическими выражениями и свидетельствует о том, что из формулы, стоящей слева от него, выводится формула, стоящая справа от него.

Сформулируем теперь сами правила вывода Армстронга в виде следующей теоремы.

Теорема. Справедливы следующие правила, называемые правилами вывода Армстронга.

Правило вывода 1. $\vdash X \rightarrow X$;

Правило вывода 2. $X \rightarrow Y \vdash X \cup Z \rightarrow Y$;

Правило вывода 3. $X \rightarrow Y, Y \cup W \rightarrow Z \vdash X \cup W \rightarrow Z$;

Здесь X, Y, Z, W — произвольные подсхемы схемы отношения S. Символ метаутверждения о выводимости разделяет списки посылок и списки утверждений (заключений).

1. Первое правило вывода называется «**рефлексивность**» и читается следующим образом: «выводится правило: “X функционально влечет за собой X”». Это самое простое из правил вывода Армстронга. Оно выводится буквально из воздуха.

Интересно заметить, что функциональная зависимость, обладающая и левой, и правой частями, называется **рефлексивной**. Согласно правилу рефлексивности ограничение рефлексивной зависимости выполняется автоматически.

2. Второе правило вывода называется «**пополнение**» и читается таким образом: «если X функционально определяет Y, то выводится правило: "объединение подсхем X и Z функционально влечет за собой Y"». Правило пополнения позволяет расширять левую часть ограничения функциональных зависимостей.

3. Третье правило вывода называется «**псевдотранзитивность**» и читается следующим образом: "если подсхема X функционально влечет за собой подсхему Y и объединение подсхем Y и W функционально влекут за собой Z, то выводится правило: «объединение подсхем X и W функционально определяют подсхему Z»".

Правило псевдотранзитивности обобщает правило транзитивности, соответствующее частному случаю W: = $\varnothing$. Приведем формулярную запись этого правила:

$X \to Y, Y \to Z \vdash X \to Z$.

Необходимо отметить, что посылки и заключения, приведенные ранее, были представлены в сокращенной форме обозначениями схем функциональной зависимости. В расширенной форме им соответствуют следующие ограничения функциональных зависимостей.

Правило вывода 1. $inv < X \to X > r(S)$;

Правило вывода 2. $inv < X \to Y > r(S) \Rightarrow inv < X \cup Z \to Y > r(S)$;

Правило вывода 3. $inv < X \to Y > r(S)\ \&\ inv < Y \cup W \to Z > r(S) \Rightarrow$
$\Rightarrow inv < X \cup W \to Z > r(S)$;

Проведем **доказательства** этих правил вывода.

1. Доказательство правила **рефлексивности** следует непосредственно из определения ограничения функциональной зависимости при подстановке вместо подсхемы Y — подсхемы X.

Действительно, возьмем ограничение функциональной зависимости:

$Inv < X \to Y > r(S)$ и подставим в него X вместо Y, получим:

$Inv < X \to X > r(S)$, а это и есть правило рефлексивности.

Правило рефлексивности доказано.

2. Доказательство правила **пополнения** проиллюстрируем на диаграммах функциональной зависимости.

Первая диаграмма — это диаграмма посылки:
посылка: $X \to Y$

X	Y
••	•
...	...
••	•

Вторая диаграмма:
заключение: $X \cup Z \to Y$

X	Y	Z
•••	•	••
...	...	...
•••	•	••

Пусть кортежи равны на $X \cup Z$. Тогда они равны на X. Согласно посылке они будут равны и на Y.

Правило пополнения доказано.

3. Доказательство правила **псевдотранзитивности** также проиллюстрируем на диаграммах, которых в этом конкретном случае будет три.

Первая диаграмма — первая посылка:
посылка 1: $X \to Y$

X	Y
••	•
...	...
••	•

посылка 2: $Y \cup W \to Z$

X	Z	W
•••	•	••
...	...	...
•••	•	••

И, наконец, третья диаграмма — диаграмма заключения:
заключение: $X \cup W \to Z$

X	Z	W
•••	•	••
...	...	...
•••	•	••

Пусть кортежи равны на $X \cup W$. Тогда они равны и на X, и на W. Согласно Посылке 1, они будут равны и на Y. Отсюда, согласно Посылке 2, они будут равны и на Z.

Правило псевдотранзитивности доказано.

Все правила доказаны.

3. Производные правила вывода

Другим примером правил, с помощью которых можно, при необходимости вывести новые правила функциональной зависимости, являются так называемые **производные правила вывода**.

Что это за правила, как они получаются?

Известно, что если из одних правил, уже существующих, законными логическими методами вывести другие, то эти новые правила, называемые **производными**, можно использовать наряду с исходными правилами.

Необходимо специально отметить, что эти самые произвольные правила являются «производными» именно от пройденных нами ранее правил вывода Армстронга.

Сформулируем производные правила вывода функциональных зависимостей в виде следующей теоремы.

Теорема.

Следующие правила являются производными от правил вывода Армстронга.

Правило вывода 1. $\vdash X \cup Z \rightarrow X$;

Правило вывода 2. $X \rightarrow Y, X \rightarrow Z \vdash X \cup Y \rightarrow Z$;

Правило вывода 3. $X \rightarrow Y \cup Z \vdash X \rightarrow Y, X \rightarrow Z$;

Здесь X, Y, Z, W, так же как и в предыдущем случае, — произвольные подсхемы схемы отношения S.

1. Первое производное правило называется **правилом тривиальности** и читается следующим образом:

«Выводится правило: “объединение подсхем X и Z функционально влечет за собой X”».

Функциональная зависимость с левой частью, являющейся подмножеством правой части, называется **тривиальной**. Согласно правилу тривиальности ограничения тривиальной зависимости выполняются автоматически.

Интересно, что правило тривиальности является обобщением правила рефлексивности и, как и последнее, могло бы быть получено непосредственно из определения ограничения функциональной

зависимости. Тот факт, что это правило является производным, не случаен и связан с полнотой системы правил Армстронга. Подробнее о полноте системы правил Армстронга мы поговорим чуть позднее.

2. Второе производное правило называется **правилом аддитивности** и читается следующим образом: «Если подсхема X функционально определяет подсхему Y, и X одновременно функционально определяет Z, то из этих правил выводится следующее правило: "X функционально определяет объединение подсхем Y и Z"».

3. Третье производное правило называется **правилом проективности** или правилом «**обращение аддитивности**». Оно читается следующим образом: «Если подсхема X функционально определяет объединение подсхем Y и Z, то из этого правила выводится правило: "X функционально определяет подсхему Y и одновременно X функционально определяет подсхему Z"», т. е., действительно, это производное правило является обращенным правилом аддитивности.

Любопытно, что правила аддитивности и проективности применительно к функциональным зависимостям с одинаковыми левыми частями позволяют объединять или, наоборот, расщеплять правые части зависимости.

При построении цепочек вывода после формулировки всех посылок применяется правило транзитивности с той целью, чтобы включить функциональную зависимость с правой частью, находящейся в заключении.

Проведем **доказательства** перечисленных произвольных правил вывода.

1. Доказательство правила **тривиальности**.

Проведем его, как и все последующие доказательства, по шагам:

1) имеем: $X \rightarrow X$ (из правила рефлексивности вывода Армстронга);

2) имеем далее: $X \cup Z \rightarrow X$ (получаем, применяя сначала правило пополнения вывода Армстронга, а потом как следствие первого шага доказательства).

Правило тривиальности доказано.

2. Проведем пошаговое доказательство правила **аддитивности**:

1) имеем: $X \rightarrow Y$ (это посылка 1);

2) имеем: $X \rightarrow Z$ (это посылка 2);

3) имеем: $Y \cup Z \rightarrow Y \cup Z$ (из правила рефлексивности вывода Армстронга);

4) имеем: $X \cup Z \rightarrow Y \cup Z$ (получаем при помощи применения правила псевдотранзитивности вывода Армстронга, а потом как следствие первого и третьего шагов доказательства);

5) имеем: $X \cup X \rightarrow Y \cup Z$ (получаем, применяя правило псевдотранзитивности вывода Армстронга, а после следует из второго и четвертого шагов);

6) имеем $X \rightarrow Y \cup Z$ (следует из пятого шага).

Правило аддитивности доказано.

3. И, наконец, проведем построение доказательства правила **проективности**:

1) имеем: $X \rightarrow Y \cup Z$, $X \rightarrow Y \cup Z$ (это посылка);

2) имеем: $Y \rightarrow Y$, $Z \rightarrow Z$ (выводится при помощи правила рефлексивности вывода Армстронга);

3) имеем: $Y \cup Z \rightarrow Y$, $Y \cup Z \rightarrow Z$ (получается из правила пополнения вывода Армстронга и следствием из второго шага доказательства);

4) имеем: $X \rightarrow Y$, $X \rightarrow Z$ (получается, применением правила псевдотранзитивности вывода Армстронга, а затем как следствие из первого и третьего шагов доказательства).

Правило проективности доказано.

Все производные правила вывода доказаны.

4. Полнота системы правил Армстронга

Пусть $F(S)$ — заданное множество функциональных зависимостей, заданных над схемой отношения S.

Обозначим через **inv** < $F(S)$ > ограничение, накладываемое этим множеством функциональных зависимостей. Распишем его:

$Inv < F(S) > r(S) = \forall X \rightarrow Y \in F(S)\ [inv < X \rightarrow Y > r(S)]$.

Итак, это множество ограничений, накладываемое функциональными зависимостями, расшифровывается следующим образом: для любого правила из системы функциональных зависимостей $X \rightarrow Y$, принадлежащего множеству функциональных зависимостей $F(S)$, действует ограничение функциональных зависимостей inv $<X \rightarrow Y> r(S)$, определенных над множеством отношения $r(S)$.

Пусть какое-то отношение $r(S)$ удовлетворяет этому ограничению.

Применяя правила вывода Армстронга к функциональным зависимостям, определенным для множества $F(S)$, можно получить новые функциональные зависимости, как уже было сказано и доказано нами ранее. И, что показательно, ограничениям этих функциональных зависимостей отношение $F(S)$ будет автоматически удовлетво-

рять, что видно из расширенной формы записи правил вывода Армстронга. Напомним общий вид этих расширенных правил вывода:

Правило вывода 1. $inv < X \rightarrow X > r(S)$;

Правило вывода 2. $inv <X \rightarrow Y> r(S) \Rightarrow inv <X \cup Z \rightarrow Y> r(S)$;

Правило вывода 3. $inv <X \rightarrow Y> r(S)$ & $inv <Y \cup W \rightarrow Z> r(S) \Rightarrow$
$\Rightarrow inv <X \cup W \rightarrow Z>$;

Возвращаясь к нашим рассуждениям, пополним множество $F(S)$ новыми, выведенными из него же с помощью правил Армстронга зависимостями. Будем применять эту процедуру пополнения до тех пор, пока у нас не перестанут получаться новые функциональные зависимости. В результате этого построения мы получим новое множество функциональных зависимостей, называемое **замыканием** множества $F(S)$ и обозначаемое $\boldsymbol{F^+(S)}$.

Действительно, такое название вполне логично, ведь мы собственноручно путем длительного построения «замкнули» множество имеющихся функциональных зависимостей само на себе, прибавив (отсюда «+») все новые функциональные зависимости, получившиеся из имеющихся.

Необходимо заметить, что этот процесс построения замыкания конечен, ведь конечна сама схема отношения, на которой и проводятся все эти построения.

Само собой разумеется, что замыкание является надмножеством замыкаемого множества (действительно, ведь оно больше!) и ни сколько не изменяется при своем повторном замыкании.

Если записать только что сказанное в формулярном виде, то получим:

$F(S) \subseteq F^+(S)$, $[F^+(S)]^+ = F^+(S)$;

Далее из доказанной истинности (т. е. законности, правомерности) правил вывода Армстронга и определения замыкания следует, что любое отношение, удовлетворяющее ограничениям заданного множества функциональных зависимостей, будет удовлетворять ограничению зависимости, принадлежащей замыканию.

$X \rightarrow Y \in F^+(S) \Rightarrow \forall r(S)\ [inv <F(S)> r(S) \Rightarrow inv <X \rightarrow Y> r(S)]$;

Итак, теорема полноты системы правил вывода Армстронга утверждает, что внешняя импликация может совершенно законно и обоснованно быть заменена эквивалентностью.

(Доказательство этой теоремы мы рассматривать не будем, так как сам процесс доказательства не столь важен в нашем конкретном курсе лекций.)

ЛЕКЦИЯ № 10. Нормальные формы

1. Смысл нормализации схем баз данных

Понятие, которое мы будем рассматривать в данном разделе, связано с понятием функциональных зависимостей, т. е. смысл нормализации схем баз данных неразрывно связан с понятием ограничений, накладываемых системой функциональных зависимостей, и во многом следует из этого понятия.

Исходной точкой любого проектирования базы данных является представление предметной области в виде одного или нескольких отношений, и на каждом шаге проектирования производится некоторый набор схем отношений, обладающих «улучшенными» свойствами. Таким образом, процесс проектирования представляет собой процесс нормализации схем отношений, причем каждая следующая нормальная форма обладает свойствами, в некотором смысле лучшими, чем предыдущая.

Каждой нормальной форме соответствует определенный набор ограничений, и отношение находится в некоторой нормальной форме, если удовлетворяет свойственному ей набору ограничений. Примером может служить ограничение первой нормальной формы — значения всех атрибутов отношения атомарны.

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

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

(В данный курс лекций включается подробное рассмотрение первых четырех нормальных форм базовых отношений, поэтому мы не будем подробно разбирать четвертую и пятую нормальные формы.)

Основные свойства нормальных форм состоят в следующем:

1) каждая следующая нормальная форма в некотором смысле лучше предыдущей нормальной формы;

2) при переходе к следующей нормальной форме свойства предыдущих нормальных форм сохраняются.

В основе процесса проектирования лежит метод нормализации, т. е. декомпозиции отношения, находящегося в предыдущей нормальной форме, на два или более отношений, которые удовлетворяют требованиям следующей нормальной формы (с этим мы столкнемся, когда нам самим придется по мере прохождения материала проводить нормализацию того или иного базового отношения).

Как уже упоминалось в разделе, посвященном созданию базовых отношений, заданные множества функциональных зависимостей, накладывают соответствующие ограничения на схемы базовых отношений. Эти ограничения в общем случае реализуются двумя методами:

1) декларативно, т. е. с помощью объявления в базовом отношении различного вида первичных, кандидатных и внешних ключей (это метод, получивший наибольшее распространение);

2) процедурно, т. е. написанием программного кода (использованием упомянутых выше так называемых триггеров).

При помощи простой логики можно понять, в чем же заключается смысл нормализации схем баз данных. Нормализовывать базы данных или приводить базы данных к нормальному виду — это значит определять такие схемы базовых отношений, чтобы максимально уменьшить необходимость написания программного кода, увеличить производительность работы базы данных, облегчить поддержку целостности данных по состоянию и ссылочной целостности. То есть сделать код и работу с ним максимально простой и удобной разработчикам и пользователям.

Для того чтобы наглядно в сравнении продемонстрировать работу ненормализованной и нормализованной базы данных, рассмотрим следующий пример.

Пусть у нас имеется базовое отношение, содержащее информацию о результатах экзаменационной сессии. Такую базу данных мы уже рассматривали раньше.

Итак, **вариант 1** схемы базы данных.

Сессия (№ зачетной книжки, Фамилия, Имя, Отчество, Предмет, Оценка).

В этом отношении, как видно из изображения схемы базового отношения, задан составной первичный ключ:

Primary key (№ зачетной книжки, Предмет);

Также в этом отношении задана система функциональных зависимостей:

{№ зачетной книжки} → {Фамилия, Имя, Отчество};

Приведем табличный вид небольшого фрагмента базы данных с данной схемой отношения. Этот фрагмент мы уже применяли в рассмотрении ограничений функциональных зависимостей, поэтому на его примере нам будет довольно легко понять и данную тему.

Сессия					
№ зачетной книжки	**Фамилия**	**Имя**	**Отчество**	**Предмет**	**Оценка**
100	Тюряпина	Елизавета	Николаевна	Информатика	Удовлетворительно
…	…	…	…	…	…
1100	Михайлова	Анастасия	Александровна	Базы данных	Отлично
…	…	…	…	…	…
100	Тюряпина	Елизавета	Николаевна	Литература	Отлично

Здесь для поддержания целостности данных по состоянию, т. е. для выполнения ограничения системы функциональной зависимости {№ зачетной книжки} → {Фамилия, Имя, Отчество} при изменении, например, фамилии необходимо просматривать все кортежи этого базового отношения и последовательно вводить необходимые изменения. Однако так как это довольно громоздкий и трудоемкий процесс (особенно если мы имеем дело с базой данных большого учебного заведения), разработчики систем управления базами данных пришли к выводу, что этот процесс необходимо автоматизировать, т. е. сделать автоматическим. Теперь контроль выполнения этой (и любой другой) функциональной зависимости можно организовывать автоматически при помощи правильного объявления в базовом отношении различных ключей и так называемой декомпозиции (т. е. разбиения чего-либо на несколько самостоятельных частей) этого отношения.

Итак, нашу имеющуюся схему отношения «Сессия» разобьем на две схемы: схему «Студенты», содержащую только информацию о студентах данного учебного заведения, и схему «Сессия», содержащую информацию о последней прошедшей сессии. А затем объявим ключи таким образом, чтобы можно было без труда получить любую необходимую информацию.

Покажем, как будут выглядеть эти новые схемы отношений со своими ключами.

Вариант 2 схемы базы данных.

Студенты (№ зачетной книжки, Фамилия, Имя, Отчество),
Primary key (№ зачетной книжки).

Сессия (№ зачетной книжки, Предмет, Оценка),
Primary key (№ зачетной книжки, Предмет),
Foreign key (№ зачетной книжки) references Студенты (№ номер зачетной книжки).

Что мы имеем теперь? В отношении «Студенты» первичный ключ «№ зачетной книжки» функционально определяет остальные три атрибута: «Фамилия», «Имя» и «Отчество». А в отношении «Сессия» составной первичный ключ «№ зачетной книжки, Предмет» также однозначно, т. е. буквально функционально определяет последний атрибут этой схемы отношения — «Оценка». И связь между этими двумя отношениями налажена: она осуществляется посредством внешнего ключа отношения «Сессия» «№ зачетной книжки», который ссылается на одноименный атрибут отношения «Студенты» и при соответствующем запросе представляет всю необходимую информацию.

Покажем теперь, как будут выглядеть отношения, представленные таблицами, отвечающие второму варианту задания соответствующих схем баз данных.

Студенты			
№ зачетной книжки	**Фамилия**	**Имя**	**Отчество**
100	Тюряпина	Елизавета	Николаевна
…	…	…	…
1100	Михайлова	Анастасия	Александровна

Сессия		
№ зачетной книжки	**Предмет**	**Оценка**
100	Информатика	Удовлетворительно
…	…	…
1100	Базы данных	Отлично
…	…	…
100	Литература	Отлично

Таким образом, мы видим, что целью нормализации в аспекте ограничений, накладываемых функциональными зависимостями, является необходимость навязать любой базе данных требуемые функциональные зависимости при помощи объявлений различного вида первичных, кандидатных и внешних ключей базовых отношений.

2. Первая нормальная форма (1NF)

На ранних стадиях проектирования баз данных и разработки схем их управления использовались простые и однозначные атрибуты как наиболее продуктивные и рациональные единицы кода. Тогда применяли наряду с простыми и составные атрибуты, а также наряду с однозначными и многозначные атрибуты. Поясним значения каждого из этих понятий.

Составные атрибуты, в отличие от простых, — это атрибуты, составленные из нескольких простых атрибутов.

Многозначные атрибуты, в отличие от однозначных, — это атрибуты, представляющие множество значений.

Приведем примеры простых, составных, однозначных и многозначных атрибутов.

Рассмотрим следующую таблицу, представляющую отношение:

...	Адрес	Телефон
...	...	...

Здесь атрибут «Телефон» — простой, однозначный, а атрибут «Адрес» — простой, но многозначный.

Теперь рассмотрим другую таблицу, с другими атрибутами:

...	Адреса	Телефоны
...	...	...

В этом отношении, представленном таблицей, атрибут «Телефоны» — простой, но многозначный, а атрибут «Адреса» — и составной, и многозначный.

Вообще возможны различные комбинации простых или составных атрибутов. В разных случаях таблицы, представляющие отношения, могут выглядеть следующим общим образом:

Атрибут	Простой	Составной
Однозначный	Телефон	Адрес
Многозначный	Телефоны	Адреса

При нормализации схем базовых отношений программистами может быть использована одна из четырех наиболее распространенных видов нормальных форм: первая нормальная форма (1NF), вторая нормальная форма (2NF), третья нормальная форма (3NF) или нормальная форма Бойса — Кодда (NFBC). Поясним: сокращение NF — это аббревиатура от англоязычного словосочетания Normal Form. Формально, кроме вышеназванных, существуют и другие виды нормальных форм, но вышеназванные — одни из самых востребованных.

В настоящее время разработчики баз данных стараются избегать составных и многозначных атрибутов, чтобы не усложнять написание кода, не перегружать его структуру и не запутывать пользователей. Из этих соображений логически и вытекает определение первой нормальной формы.

Определение. Любое базовое отношение находится в **первой нормальной форме** тогда и только тогда, когда схема этого отношения содержит только простые и только однозначные атрибуты, причем обязательно с одной и той же семантикой.

Для наглядного объяснения различий нормализованных и ненормализованных отношений рассмотрим пример.

Пусть, имеется ненормализованное отношение, со следующей схемой.

Итак, **вариант 1** схемы отношения с заданным на ней простым первичным ключом:

Сотрудники (№ табельный, Фамилия Имя Отчество, Код должности, Телефоны, Дата приема или увольнения);

Primary key (№ табельный);

Перечислим, какие в этой схеме отношения имеются ошибки, т. е. назовем те признаки, которые и делают собственно эту схему ненормализованной:

1) атрибут «Фамилия Имя Отчество» является составным, т. е. составленным из разнородных элементов;

2) атрибут «Телефоны» является многозначным, т. е. его значением является множество значений;

3) атрибут «Дата приема или увольнения» не имеет однозначной семантики, т. е. в последнем случае не понятно, какая именно дата внесена.

Если, например, ввести дополнительный атрибут, чтобы поточнее определить смысл даты, то для этого атрибута значение будет семантически понятно, но тем не менее остается возможность хранения только какой-то одной из указанных дат для каждого сотрудника.

Что же необходимо сделать для приведения этого отношения к нормальной форме?

Во-первых, необходимо провести разбиение составных атрибутов на простые, для того, чтобы исключить эти самые составные атрибуты, а также атрибуты с составной семантикой.

А во-вторых, необходимо провести декомпозицию этого отношения, т. е. нужно разбить его на несколько новых самостоятельных отношений, с тем чтобы исключить многозначные атрибуты.

Таким образом, с учетом всего вышесказанного после приведения отношения «Сотрудники» к первой нормальной форме или 1NF путем его декомпозиции мы получим систему следующих отношений с заданными на них первичными и внешними ключами.

Итак, **вариант 2** отношения:

Сотрудники (№ табельный, Фамилия, Имя, Отчество, Код должности, Дата приема, Дата увольнения);

Primary key (№ табельный);

Телефоны (№ табельный, Телефон);

Primary key (№ табельный, Телефон);

Foreign key (№ табельный) references Сотрудники (№ табельный);

Итак, что мы видим? Составного атрибута «Фамилия Имя Отчество» больше в нашем отношении нет, вместо него присутствуют три простых атрибута «Фамилия», «Имя» и «Отчество», поэтому эта причина «ненормальности» отношения исключилась.

Кроме того, вместо атрибута с неясной семантикой «Дата приема или увольнения» у нас появилось два атрибута «Дата приема» и «Дата увольнения», каждый из которых имеет однозначную семантику. Следовательно, вторая причина того, что наше отношение «Сотрудники» не находится в нормальной форме, также благополучно устранена.

И, наконец, последняя причина того, что отношение «Сотрудники» не было приведено к нормальной форме, — это наличие многозначного атрибута «Телефоны». Чтобы избавиться от этого атрибута, и необходимо было провести декомпозицию всего отношения. Из исходного отношения «Сотрудники» в результате этой декомпозиции был исключен атрибут «Телефоны» вообще, но зато образовалось второе отношение — «Телефоны», в котором присутствуют два атрибута: «№ табельный» сотрудника и «Телефон», т. е. все атрибуты — опять-таки простые, условие принадлежности к первой нормальной форме выполняется. Эти атрибуты «№ табельный» и «Телефон» образуют составной первичный ключ отношения «Телефоны», а атрибут «№ табельный», в свою очередь,

является внешним ключом, ссылающимся на одноименный атрибут отношения «Сотрудники», т. е. в отношении «Телефоны» атрибут первичного ключа «№ табельный» является одновременно внешним ключом, ссылающимся на первичный ключ отношения «Сотрудники». Таким образом, обеспечивается связь между этими двумя отношениями. Посредством этой связи можно по номеру табельному любого сотрудника без особого труда и затрат времени вывести весь список его телефонов, не прибегая к использованию составных атрибутов.

Заметим, что в случае наличия в отношении системы ограничений функциональных зависимостей после всех вышеприведенных преобразований нормализация не была бы завершена. Однако в данном конкретном примере нет ограничений функциональных зависимостей, поэтому дальнейшая нормализация этого отношения не требуется.

3. Вторая нормальная форма (2NF)

Более сильные требования накладывает на отношения вторая нормальная форма, или 2NF.

Это происходит потому, что определение второй нормальной формы отношений предполагает, в отличие от первой нормальной формы, наличие системы ограничений функциональных зависимостей.

Определение. Базовое отношение находится во **второй нормальной форме** относительного заданного множества функциональных зависимостей тогда и только тогда, когда оно находится в первой нормальной форме и, кроме того, каждый неключевой атрибут полностью функционально зависит от каждого ключа.

В этом определении **неключевой атрибут** — это любой атрибут отношения, не содержащийся в каком-либо первичном или кандидатном ключе отношения.

Полная функциональная зависимость от ключа предполагает отсутствие функциональной зависимости от какой-либо части этого ключа.

Таким образом, теперь при нормализации отношения мы должны следить и за выполнением условий пребывания отношения в первой нормальной форме, т. е. следить, чтобы его атрибуты были простыми и однозначными, а также за выполнением второго условия, касающегося ограничений функциональных зависимостей.

Ясно, что отношения с простыми ключами (первичными и кандидатными) заведомо находятся во второй нормальной форме. Ведь в таком случае, зависимость от части ключа просто не представляется возможной, потому что никаких отдельных частей ключ банально не имеет.

Теперь, как и при прохождении предыдущей темы, рассмотрим пример ненормализованной схемы отношения и сам процесс нормализации.

Итак, **вариант 1** схемы отношения:

Аудитории (№ корпуса, № аудитории, Площадь кв. м, № табельный коменданта корпуса);

Primary key (№ корпуса, № аудитории);

Кроме того, определена следующая система функциональной зависимости:

{№ корпуса} → {№ табельный коменданта корпуса};

Что мы видим? Все условия пребывания этого отношения «Аудитории» в первой нормальной форме выполнены, ведь все до единого атрибуты этого отношения однозначны и просты. Но то условие, что каждый неключевой элемент должен полностью функционально зависеть от ключа, не выполняется. Почему? Да потому, что атрибут «№ табельный коменданта корпуса» функционально зависит не от составного ключа «№ корпуса, № аудитории», а от части этого ключа, т. е. от атрибута «№ корпуса». Действительно, ведь именно номер корпуса полностью определяет, какой именно комендант к нему приписан, а, в свою очередь, ни от каких номеров аудиторий табельный номер коменданта корпуса зависеть никак не может.

Таким образом, основной задачей нашей нормализации становится задача добиться того, чтобы ключи распределялись таким образом, чтобы, в частности, атрибут «№ табельный коменданта корпуса» полностью функционально зависел от всего ключа, а не от его какой-то части.

Для того, чтобы этого добиться, придется снова, как и в предыдущем параграфе, применить декомпозицию отношения. Итак, следующая система отношений, представляющая собой **вариант 2** отношения «Аудитории», как раз и получилась из исходного отношения путем его декомпозиции на несколько новых самостоятельных отношений:

Корпуса (№ корпуса, № табельный коменданта корпуса);

Primary key (№ корпуса);

Аудитории (№ корпуса, № аудитории, Площадь кв. м);

Primary key (№ корпуса, № аудитории);

Foreign key (№ корпуса) references Корпуса (№ корпуса);

Что мы видим теперь? В отношении «Корпуса» неключевой атрибут «№ табельный коменданта корпуса» полностью функционально зависит от первичного ключа «№ корпуса». Здесь условие нахождения отношения во второй нормальной форме полностью выполнились.

Теперь перейдем к рассмотрению второго отношения — «Аудитории». В отношении «Аудитории» атрибут первичного ключа «№ кор-

пуса» является одновременно внешним ключом, ссылающемся на первичный ключ отношения «Корпуса». В этом отношении неключевой атрибут «Площадь кв. м» полностью зависит от всего составного первичного ключа «№ корпуса, № аудитории» и не зависит, даже не может зависеть ни от какой из его частей.

Таким образом, путем декомпозиции исходного отношения, мы пришли к тому, что все условия из определения второй нормальной формы полностью выполнились.

В данном примере все требования функциональной зависимости навязаны объявлением первичных ключей (кандидатных ключей здесь нет) и внешних ключей. Поэтому дальнейшая нормализация не требуется.

4. Третья нормальная форма (3NF)

Следующей нормальной формой, которую мы подвергнем рассмотрению, является третья нормальная форма (или 3NF). В отличие от первой нормальной формы, так же как и вторая нормальная форма, третья — подразумевает задание вместе с отношением системы функциональных зависимостей. Сформулируем, какими свойствами должно обладать отношение, чтобы оно было приведенным к третьей нормальной форме.

Определение. Базовое отношение находится в **третьей нормальной форме** относительно заданного множества функциональных зависимостей тогда и только тогда, когда оно находится во второй нормальной форме и каждый неключевой атрибут полностью функционально зависит только от ключей.

Таким образом, требования, предъявляемые третьей нормальной формой, сильнее требований, накладываемых первой и второй нормальной формой, даже вместе взятых. Фактически в третьей нормальной форме каждый неключевой атрибут зависит от ключа, причем от всего ключа целиком и ни от чего другого, кроме как от ключа.

Проиллюстрируем процесс приведения ненормализованного отношения к третьей нормальной форме. Для этого рассмотрим пример: отношение, находящееся не в третьей нормальной форме.

Итак, **вариант 1** схемы отношения «Сотрудники»:

Сотрудники (№ табельный, Фамилия, Имя, Отчество, Код должности, Оклад);

Primary key (№ табельный);

Кроме того, над данным отношением «Сотрудники» задана следующая система функциональных зависимостей:

{Код должности} → {Оклад};

Действительно, как правило, от должности, а следовательно, от ее кода в соответствующей базе данных напрямую зависит размер оклада, т. е. размер заработной платы.

Именно поэтому это отношение «Сотрудники» и не находится в третьей нормальной форме, ведь получается, что неключевой атрибут «Оклад» полностью функционально зависит от атрибута «Код должности», хотя этот атрибут и не является ключевым.

Любопытно, что к третьей нормальной форме любое отношение приводится точно таким же методом, как и к двум формам до этой, а именно, путем декомпозиции.

Проведя декомпозицию отношения «Сотрудники», получим следующую систему новых самостоятельных отношений:

Итак, **вариант 2** схемы отношения «Сотрудники»:

Должности (Код должности, Оклад);

Primary key (Код должности);

Сотрудники (№ табельный, Фамилия, Имя, Отчество, Код должности);

Primary key (Код должности);

Foreign key (Код должности) references Должности (Код должности);

Теперь, как мы видим, в отношении «Должности» неключевой атрибут «Оклад» полностью функционально зависит от простого первичного ключа «Код должности» и только от этого ключа.

Заметим, что в отношении «Сотрудники» все четыре неключевых атрибута «Фамилия», «Имя», «Отчество» и «Код должности» полностью функционально зависят от простого первичного ключа «№ табельный». В этом отношении атрибут «Код должности» — внешний ключ, ссылающийся на первичный ключ отношения «Должности».

В данном примере все требования навязаны объявлением простых первичных и внешних ключей, поэтому дальнейшая нормализация не требуется.

Интересно и полезно знать, что на практике обычно ограничиваются приведением баз данных к третьей нормальной форме. При этом, возможно, не навязанными остаются некоторые функциональные зависимости ключевых атрибуты от других атрибутов этого же отношения.

Поддержка таких нестандартных функциональных зависимостей реализуется при помощи уже упоминаемых ранее триггеров (т. е. процедурно, путем написания соответствующего программного кода). Причем триггеры должны оперировать кортежами этого отношения.

5. Нормальная форма Бойса—Кодда (NFBC)

Нормальная форма Бойса—Кодда следует по «сложности» сразу после третьей нормальной формы. Поэтому нормальную форму Бойса—Кодда еще иногда называют просто **усиленной третьей нормальной формой** (или усиленной 3 NF). Почему же она именно усиленная? Сформулируем определение нормальной формы Бойса—Кодда:

Определение. Базовое отношение находится в **нормальной форме Бойса — Кодда** тогда и только тогда, когда она находится в третьей нормальной форме, и при этом не только любой неключевой атрибут полностью функционально зависит от любого ключа, но и любой ключевой атрибут должен полностью функционально зависеть от любого ключа.

Таким образом, требование о фактической зависимости неключевых атрибутов от всего ключа целиком и ни от чего другого, кроме как от ключа, распространяется и на ключевые атрибуты.

В отношении, находящемся в нормальной форме Бойса—Кодда, все функциональные зависимости в пределах отношения навязаны объявлением ключей. Однако при приведении отношений баз данных к форме Бойса—Кодда, возможны ситуации, при которых не навязанными функциональными зависимостями оказываются зависимости между атрибутами различных отношений. Поддержка таких функциональных зависимостей при помощи триггеров, оперирующих кортежами различных отношений, сложнее, чем в случае третьей нормальной формы, когда триггеры оперируют кортежами единственного отношения.

Кроме всего прочего, практика проектирования систем управления базами данных показала, что не всегда удается привести базовое отношение к нормальной форме Бойса—Кодда.

Причиной отмеченных аномалий является то, что в требованиях второй нормальной формы и третьей нормальной формы не требовалась минимальная функциональная зависимость от первичного ключа атрибутов, являющихся компонентами других возможных ключей. Эту проблему и решает нормальная форма, которую исторически принято называть нормальной формой Бойса—Кодда и которая является уточнением третьей нормальной формы в случае наличия нескольких перекрывающихся возможных ключей.

Вообще нормализация схемы базы данных способствует более эффективному выполнению системой управления базами данных операций обновления базы данных, поскольку сокращается число проверок и вспомогательных действий, поддерживающих целостность базы данных. При проектировании реляционной базы данных почти всегда добиваются второй нормальной формы всех входящих в базу данных от-

ношений. В часто обновляемых базах данных обычно стараются обеспечить третью нормальную форму отношений. На нормальную форму Бойса—Кодда внимание обращают гораздо реже, поскольку на практике ситуации, в которых у отношения имеется несколько составных перекрывающихся возможных ключей, встречаются нечасто.

Все вышеназванное делает нормальную форму Бойса—Кодда не слишком удобной в использовании при разработке программного кода, поэтому, как уже было сказано ранее, на практике разработчики обычно ограничиваются приведением своих баз данных к третьей нормальной форме. Однако здесь тоже есть своя довольно любопытная особенность. Дело в том, что ситуации, когда отношение находится в третьей нормальной форме, но не находится в нормальной форме Бойса—Кодда крайне редки на практике, т. е. после приведения к третьей нормальной форме обычно все функциональные зависимости оказываются навязанными объявлениями первичных, кандидатных и внешних ключей, так что необходимость в триггерах для поддержки функциональных зависимостей отпадает.

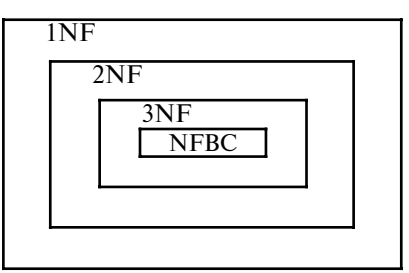
Однако необходимость в триггерах остается для поддержки ограничения целостности, не связанных функциональными зависимостями.

6. Вложенность нормальных форм

Что означает вложенность нормальных форм друг в друга?

Вложенность нормальных форм — это отношение понятий ослабленной и усиленной формы по отношению друг к другу.

Вложенность нормальных форм полностью следует из их соответствующих определений. Представим диаграмму, иллюстрирующую отношение вложенности известных нам нормальных форм:



Поясним понятия ослабленной и усиленной нормальной формы по отношению друг к другу на конкретных примерах.

Первая нормальная форма является ослабленной по отношению ко второй нормальной форме (да и по отношению ко всем остальным нормальным формам тоже). Действительно, вспоминая определения всех пройденных нами нормальных форм, можно заметить, что требования каждой нормальной формы включали в себя требование принадлежности именно к первой нормальной форме (ведь она входила в каждое последующее определение).

Вторая нормальная форма является усиленной по отношению к первой нормальной форме, но ослабленной по отношению к третьей нормальной форме и нормальной форме Бойса—Кодда. На самом деле принадлежность второй нормальной форме включается в определение третьей, а сама вторая форма, в свою очередь, включает в себя первую нормальную форму.

Нормальная форма Бойса—Кодда является усиленной не только по отношению к третьей нормальной форме, но также и по отношению ко всем остальным, предшествующим ей.

А третья нормальная форма, в свою очередь, является ослабленной только по отношению к нормальной форме Бойса—Кодда.

ЛЕКЦИЯ № 11. Проектирование схем баз данных

Наиболее распространенным средством абстрактного представления схем баз данных при проектировании на логическом уровне является так называемая **модель «сущность — связь»**. Ее еще иногда называют **ER-модель**, где ER — аббревиатура английского словосочетания Entity — Relationship, что буквально и переводится как «сущность — связь».

Элементами таких моделей являются классы сущностей, их атрибуты и связи.

Дадим объяснения и определения каждого из этих элементов.

Класс сущностей — это как бы лишенный методов класс объектов в смысле объектно-ориентированного программирования. При переходе к физическому уровню классы сущностей преобразовываются в базовые отношения реляционных баз данных для конкретных систем управления базами данных. У них, как и у собственно базовых отношений, существуют собственные атрибуты.

Дадим более точное и строгое определение только что приведенных объектов.

Классом называется именованное описание совокупности объектов с общими атрибутами, операциями, связями и семантикой. Графически обычно класс изображается в виде прямоугольника. У каждого класса должно быть имя (текстовая строка), уникально отличающее его от всех других классов.

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

Так называемые связи реализуются с помощью объявления внешних ключей (подобные явления нам уже встречались раньше), т. е. в отношении объявляются внешние ключи, ссылающиеся на первичные или кандидатные ключи каких-то других отношений. И посредством этого и происходит «связывание» нескольких различных самостоятельных базовых отношений в единую систему, называемую базой данных.

Далее диаграмма, составляющая графическую основу модели «сущность — связь», изображается при помощи унифицированного языка моделирования UML.

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

Вообще, UML позволяет моделировать разные виды систем: чисто программные, чисто аппаратные, программно-аппаратные, смешанные, явно включающие деятельность людей и т. д.

Но, помимо прочего, как мы уже упоминали, язык UML активно применяется для проектирования реляционных баз данных. Для этого используется небольшая часть языка (диаграммы классов), да и то не в полном объеме. С точки зрения проектирования реляционных баз данных, модельные возможности не слишком отличаются от возможностей ER-диаграмм.

Мы также хотели показать, что в контексте проектирования реляционных баз данных структурные методы проектирования, основанные на использовании ER-диаграмм, и объектно-ориентированные методы, основанные на использовании языка UML, различаются главным образом, лишь терминологией. ER-модель концептуально проще UML, в ней меньше понятий, терминов, вариантов применения. И это понятно, поскольку разные варианты ER-моделей разрабатывались именно для поддержки проектирования реляционных баз данных, и ER-модели почти не содержат возможностей, выходящих за пределы реальных потребностей проектировщика реляционной базы данных.

Язык UML принадлежит объектному миру. Этот мир гораздо сложнее (если угодно, непонятнее, запутаннее) реляционного мира. Поскольку UML может использоваться для унифицированного объектно-ориентированного моделирования всего чего угодно, в этом языке содержится масса различных понятий, терминов и вариантов использования, избыточных с точки зрения проектирования реляционных баз данных. Если вычленить из общего механизма диаграмм классов то, что действительно требуется для проектирования реляционных баз данных, то мы получим в точности ER-диаграммы с другой нотацией и терминологией.

Любопытно, что при формировании имен классов в UML допускается использование произвольной комбинации букв, цифр и даже знаков препинания. Однако на практике рекомендуется использовать в качестве имен классов короткие и осмысленные прилагательные и существительные, каждое из которых начинается с заглавной буквы.

(Подробнее понятие диаграммы мы рассмотрим в следующем параграфе нашей лекции.)

1. Различные типы и кратности связей

Связь между отношениями при проектировании схем баз данных изображается в виде линий, соединяющих классы сущностей.

При этом каждый из концов связи может (и вообще должен) характеризоваться наименованием (т. е. типом связи) и кратностью роли класса в связи. Рассмотрим подробнее понятия кратности и типы связей.

Кратностью (multiplicity) называется характеристика, указывающая, сколько атрибутов класса сущности с данной ролью может или должно участвовать в каждом экземпляре связи какого-либо вида.

Наиболее распространенным способом задания кратности роли связи является прямое указание конкретного числа или диапазона. Например, указание «1» говорит о том, что каждый класс с данной ролью должен участвовать в некотором экземпляре данной связи, причем в каждом экземпляре связи может участвовать ровно один объект класса с данной ролью. Указание диапазона «0..1» говорит о том, что не все объекты класса с данной ролью обязаны участвовать в каком-либо экземпляре данной связи, но в каждом экземпляре связи может участвовать только один объект. Поговорим о кратностях подробнее.

Типичными, самыми распространенными кратностями в системах проектирования баз данных являются следующие кратности:

1) 1 — кратность связи на соответствующем ее конце равна единице;

2) 0... 1 — такая форма записи означает, что кратность данной связи на соответствующем своем конце не может превышать единицы;

3) 0... ∞ — такая кратность расшифровывается просто «много». Любопытно, что, как правило, «много» означает «ничего»;

4) 1... ∞ — такое обозначение получила кратность «один или более».

Приведем пример простой диаграммы для иллюстрирования работы с различными кратностями связей.

Согласно этой диаграмме, можно легко понять, что каждая касса имеет много билетов, а, в свою очередь, каждый билет находится в какой-то одной (и не более того) кассе.

Теперь рассмотрим наиболее распространенные типы или наименования связей. Перечислим их:

1) 1 : 1 — такое обозначение получила связь «**один к одному**», т. е. это как бы взаимно-однозначное соответствие двух множеств;

2) 1 : 0... ∞ — это обозначение связи типа «**один ко многим**». Для краткости такую связь называют «1 : М». В рассмотренной ранее диаграмме, как можно заметить, присутствует связь именно с таким наименованием;

3) 0... ∞ : 1 — это обращение предыдущей связи или связь типа «**многие к одному**»;

4) 0... ∞ : 0... ∞ — это обозначение связи типа «**многие ко многим**», т. е. с каждого конца связи присутствует много атрибутов;

5) 0... 1 : 0... 1 — это связь, аналогичная введенной ранее связи типа «один к одному», она, в свою очередь, называется «**не более одного к не более одному**»;

6) 0... 1 : 0... ∞ — это связь, аналогичная связи типа «один ко многим», она называется «не более одного ко многим»;

7) 0... ∞ : 0... 1 — это связь, в свою очередь, аналогичная связи типа «многие к одному», она называется «**многие к не более одному**».

Как можно заметить, три последние связи получились из связей, которые в нашей лекции перечислены под номерами один, два и три путем замены кратности «один» на кратность «не более одного».

2. Диаграммы. Виды диаграмм

И теперь перейдем, наконец, непосредственно к рассмотрению диаграмм и их видов.

Вообще выделяют три уровня логической модели. Эти уровни различаются по глубине представления информации о структуре данных. Этим уровням соответствуют следующие диаграммы:

1) презентационная диаграмма;

2) ключевая диаграмма;

3) полная атрибутивная диаграмма.

Разберем каждый из названных видов диаграмм и подробно поясним смысл их различия по глубине представления информации о структуре данных.

1. **Презентационная диаграмма.**

Такие диаграммы описывают только самые основные классы сущностей и их связи. Ключи в таких диаграммах могут не описываться совсем, и соответственно, связи — никак не индивидуализироваться. Поэтому допустимыми являются связи типа «многие ко

многим», хотя обычно их стараются избегать или, если они все же присутствуют, — детализировать. Также вполне допустимыми являются составные и многозначные атрибуты, хотя мы и писали раньше, что базовые отношения с такими атрибутами не являются приведенными к какой бы то ни было нормальной форме. Интересно, что из рассмотренных нами трех видов диаграмм только последний вид (полная атрибутивная диаграмма) предполагает, что данные, представленные с е помощью, находятся в некой нормальной форме. Тогда как уже рассмотренная презентационная диаграмма и следующая на очереди ключевая диаграмма ничего подобного не предполагают.

Такие диаграммы, как правило, используются для презентаций (отсюда и их название — презентационные, т. е. использующиеся для презентаций, демонстраций, где чрезмерная детализация и не нужна).

Иногда при проектировании баз данных необходимо проконсультироваться с экспертами той предметной области, информацией которой эта конкретная база данных и занимается. Тогда также используются презентационные диаграммы, ведь для получения нужных сведений у специалистов профессии, далекой от программирования, чрезмерное уточнение специфических деталей совсем не требуется.

2. **Ключевая диаграмма.**

В отличие от презентационных диаграмм ключевые диаграммы описывают обязательно все классы сущностей и их связи, правда, в терминах только первичных ключей. Здесь связи «многие ко многим» уже непременно детализируются (т. е. связей такого типа в чистом виде здесь просто не может быть задано). Многозначные атрибуты все еще допускаются так же, как и в презентационной диаграмме, но если они присутствуют в диаграмме ключевой, то, как правило, они преобразуются в самостоятельные классы сущностей. Но, что любопытно, однозначные атрибуты все еще могут быть представлены не полностью или описаны как составные. Эти «вольности», еще допустимые в таких диаграммах, как презентационная и ключевая, не допустимы уже в следующем виде диаграммы, ведь они определяют, что базовое отношение не является нормализованным.

Таким образом, можно сделать вывод, что ключевые диаграммы предполагают в дальнейшем лишь «навешивание» атрибутов на уже описанные классы сущностей, т. е. при помощи презентационной диаграммы достаточно описать самые необходимые классы сущностей, а потом уже при помощи диаграммы ключевой добавить в нее все нужные атрибуты и конкретизировать все наиболее важные связи.

3. **Полная атрибутивная диаграмма.**

Полные атрибутивные диаграммы наиболее детально из всех вышеназванных описывают все классы сущностей, их атрибуты и связи

между этими классами сущностей. Как правило, такие диаграммы представляют данные, находящиеся в третьей нормальной форме, поэтому естественно, что в базовых отношениях, описанных такими диаграммами, не допускается наличия составных или многозначных атрибутов, так же как и не допускается наличия недетализированных связей типа «многое ко многим».

Однако у полных атрибутивных диаграмм все-таки есть недостаток, т. е. и их нельзя в полной мере назвать наиболее полными из диаграмм в смысле представления данных. Например, особенность конкретных систем управления базами данных при использовании полных атрибутивных диаграмм все еще не учитывается, и в частности, тип данных конкретизируется только в той мере, в какой это необходимо для необходимого логического уровня моделирования.

3. Связи и миграция ключей

Немного раньше мы уже говорили о том, что такое связи в базах данных. В частности, связь устанавливалась при объявлении внешних ключей отношений.

Но в этом разделе нашего курса речь идет уже не о базовых отношениях, а о кассах сущностей. В этом смысле процесс установления связей все равно связан с объявлениями различных ключей, но теперь мы говорим уже ключах классов сущностей. А именно процесс установления связей связан с переносом простого или составного первичного ключа одного класса сущностей в другой класс. Сам процесс такого переноса еще называют **миграцией ключей**. При этом класс сущностей, первичные ключи которого переносятся, называется **родительским классом**, а класс сущностей, в чьи внешние ключи и происходит миграция, называется **дочерним классом** сущностей.

В дочернем классе сущностей атрибуты ключа получают статус атрибутов внешнего ключа и при этом могут участвовать или, наоборот, не участвовать в формировании его собственного первичного ключа. Таким образом, при миграции первичного ключа из родительского класса сущностей в дочерний в дочернем классе возникает внешний ключ, ссылающийся на первичный ключ родительского класса.

Для удобства формулярного представления миграции ключей, введем следующие маркеры ключей:

1) PK — так мы будем обозначать любой атрибут первичного ключа (primary key);

2) FK — этим маркером мы будем обозначать атрибуты внешнего ключа (foreign key);

3) ${}^{P}_{F}K$ — таким маркером мы будем обозначать атрибут первичного/внешнего ключа, т. е. любой такой атрибут, который входит в состав единственного первичного ключа некоторого класса сущностей и одновременно в состав некоторого внешнего ключа этого же класса сущностей.

Таким образом, атрибуты класса сущностей с маркерами PK и FK образуют первичный ключ этого класса. А атрибуты с маркерами FK и ${}^{P}_{F}K$ входят в состав каких-то некоторых внешних ключей этого класса сущностей.

Вообще, ключи могут мигрировать различным образом, и в каждом такой различном случае возникает какой-то свой вид связи. Итак, рассмотрим, какие же бывают виды связей в зависимости от схемы миграции ключей.

Всего различают две схемы миграции ключей.

1. **Схема миграции $\forall$ PK (PK $|\rightarrow {}^{P}_{F}K$);**

В этой записи символ «$|\rightarrow$ » означает понятие «мигрирует», т. е. приведенная формула прочитается следующим образом: любой (каждый) атрибут первичного ключа PK родительского класса сущностей переносится (мигрирует) в состав первичного ключа ${}^{P}_{F}K$ дочернего класса сущностей, который, естественно, является одновременно и внешним ключом для этого класса.

В этом случае речь идет о том, что каждый без исключения атрибут ключа родительского класса сущностей должен мигрировать в дочерний класс сущностей.

Такой вид связи называется **идентифицирующей**, так как ключ родительского класса сущности целиком участвует в идентификации дочерних сущностей.

Среди связей идентифицирующего типа, в свою очередь, выделяют еще два возможных самостоятельных типа связей. Итак, идентифицирующие связи бывают двух следующих типов:

1) **полностью идентифицирующими.**

Идентифицирующая связь называется полностью идентифицирующей в том и только в том случае, когда атрибуты мигрирующего первичный ключа родительского класса сущностей полностью формируют первичный (и одновременно внешний) ключ дочернего класса сущностей.

Полностью идентифицирующую связь еще иногда называют **категориальной**, потому что полностью идентифицирующая связь идентифицирует дочерние сущности по всем категориям;

2) **не полностью идентифицирующими.**

Идентифицирующая связь называется не полностью идентифицирующей в том и только в том случае, когда атрибуты мигрирующего первичного ключа родительского класса сущностей лишь частично формирует первичный (и одновременно внешний) ключ дочернего класса сущностей.

Таким образом, помимо ключа с маркером $^{P}_{F}K$ будет также присутствовать ключ с маркером PK. При этом внешний ключ PFK дочернего класса сущностей будет полностью определяться первичным ключом PK родительского класса сущностей, а просто первичный ключ PK этого дочернего отношения не будет определяться первичным ключом PK родительского класса сущностей, он будет сам по себе.

2. Схема миграции $\exists$ PK (PK $|\rightarrow$ FK);

Такая схема миграции должна читаться следующим образом: существуют такие атрибуты первичного ключа родительского класса сущностей, которые при миграции переносятся в состав обязательно неключевых атрибутов дочернего класса сущностей.

Таким образом, в этом случае речь идет о том, что *некоторые*, а не все, как в предыдущем случае, атрибуты первичного ключа родительского класса сущностей переносятся в дочерний класс сущностей. Кроме того, если предыдущая схема миграции определяла миграцию в первичный ключ дочернего отношения, который при этом становился еще и внешним ключом, то последний вид миграции определяет, что атрибуты первичного ключа родительского класса сущностей мигрируют в обычные, изначально неключевые атрибуты, которые уже после этого приобретают статус внешнего ключа.

Такой тип связи называется **неидентифицирующим**, ведь, действительно, родительский ключ целиком не участвует в формировании дочерних сущностей, он просто не идентифицирует их.

Среди неидентифицирующих связей также выделяют два возможных типа связей. Таким образом, неидентифицирующие связи бывают двух следующих видов:

1) **обязательно неидентифицирующими.**

Неидентифицирующие связи называются обязательно не идентифицирующими в том и только в том случае, когда Null-значения для всех атрибутов мигрирующего ключа дочернего класса сущностей запрещены;

2) **необязательно неидентифицирующими.**

Неидентифицирующие связи называются не обязательно неидентифицирующими в том и только в том случае, когда Null-значения для некоторых атрибутов мигрирующего ключа дочернего класса сущностей разрешены.

Обобщим все вышесказанное в виде следующей таблицы, чтобы облегчить задачу систематизации и понимания приведенного материала. Также в эту таблицу мы включим информацию о том, какие типы связей («не более одного к одному», «многие к одному», «многие к не более одному») соответствуют каким видам связей (полностью идентифицирующими, не полностью идентифицирующими, обязательно не идентифицирующими, не обязательно не идентифицирующими).

Итак, между родительскими и дочерними классами сущностей устанавливается следующий тип связей в зависимости от вида связи.

Родительский класс	**Тип связи**	**Дочерний класс**	**Вид связи**
PK	$1 \leftarrow 0..1$	$^{P}_{F}K$	Полностью идентифицирующая
PK	$1 \leftarrow 0..\infty$	$^{P}_{F}K$ PK	Не полностью идентифицирующая
PK	$1 \leftarrow 0..\infty$	FK: not null PK	Обязательно не идентифицирующая
PK	$0..1 \leftarrow$ $0..\infty$	FK: null PK	Не обязательно не идентифицирующая

Итак, мы видим, что во всех случаях, кроме последнего, ссылка не пустая (not null) $\rightarrow$ 1.

Заметим такую тенденцию, что на родительском конце связи во всех случаях, кроме последнего, устанавливается кратность «один». Это происходит потому, что значению внешнего ключа в случаях этих связей (а именно, полностью идентифицирующая, не полностью идентифицирующая и обязательно не идентифцирующая виды связей) обязательно должно соответствовать (и притом единственное) значение первичного ключа родительского класса сущностей. А в последнем случае из-за того, что значение внешнего ключа допускает равенство Null-значению (флажок допустимости FK: null), на родительском конце связи устанавливается кратность «не более одного».

Проводим наш анализ дальше. На дочернем конце связи во всех случаях, за исключением первого, устанавливается кратность «много». Это происходит потому, что за счет неполной идентификации, как во втором случае, (или вообще отсутствия таковой, во втором и третьем случаях), значение первичного ключа родительского класса сущностей может многократно встречаться среди значений внешнего ключа дочернего класса. А в первом случае связь — полностью идентифицирующая, поэтому атрибуты первичного ключа родительского класса сущностей можгут встречаться среди атрибутов ключей дочернего класса сущностей только однажды.

ЛЕКЦИЯ № 12. Связи классов сущностей

Итак, все пройденные нами понятия, а именно диаграммы и их виды, кратности и виды связей, а также виды миграции ключей, теперь помогут нам в прохождении материала о тех же связях, но уже между конкретными классами сущностей.

Среди них, как мы увидим, тоже бывают связи различных видов.

1. Иерархическая рекурсивная связь

Первым видом связи классов сущностей между собой, который мы рассмотрим, является так называемая **иерархическая рекурсивная связь**.

Вообще **рекурсия** (или **рекурсивная связь**) — это связь класса сущностей с самим собой.

Иногда по аналогии с жизненными ситуациями такую связь еще называют «рыболовный крючок».

Иерархической рекурсивной связью (или просто **иерархической рекурсией**) называется любая рекурсивная связь типа «не более одного ко многим».

Иерархическая рекурсия чаще всего используется для того, чтобы хранить данные древовидной структуры.

При задании иерархической рекурсивной связи первичный ключ родительского класса сущностей (который в данном конкретном случае одновременно выступает и в роли дочернего класса сущностей) должен мигрировать в качестве внешнего ключа в состав обязательно неключевых атрибутов того же класса сущностей. Все это необходимо для поддержания логической целостности самого понятия «иерархическая рекурсия».

Таким образом, с учетом всего вышесказанного, можно сделать вывод, что иерархическая рекурсивная связь может быть только **не обязательно не идентифицирующей** и никакой другой, потому что в случае использования любого другого вида связи, Null-значения для внешнего ключа были бы недопустимы и рекурсия была бы бесконечной.

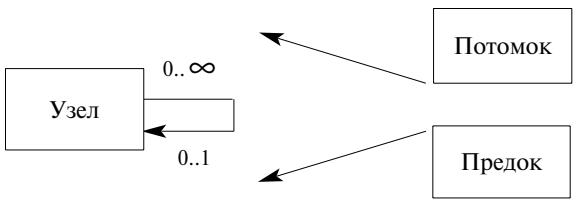
Важно также помнить, что атрибуты не могут появляться дважды в одном и том же классе сущностей под одним и тем же именем. Поэтому атрибуты мигрировавшего ключа обязательно должны получить так называемое имя роли.

Таким образом, в иерархической рекурсивной связи атрибуты узла расширяются внешним ключом, представляющим необязатель-

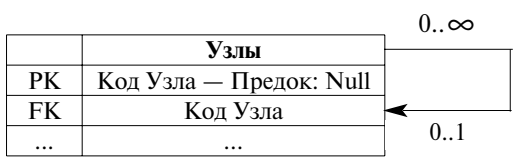
ную ссылку на первичный ключ узла, являющийся его непосредственным предком.

Построим презентационную и ключевую диаграммы, реализующую иерархическую рекурсию в реляционной модели данных, и приведем пример табличной формы.

Сначала составим презентационную диаграмму:



Теперь построим более подробную — ключевую диаграмму:



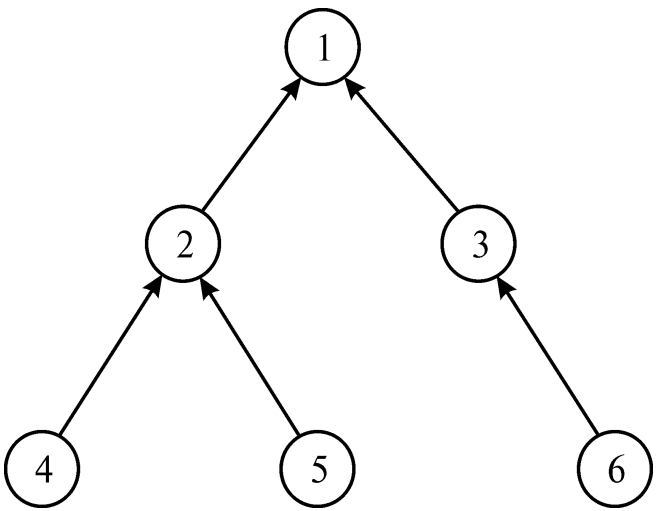
Рассмотрим пример, наглядно иллюстрирующий такой вид связи, как иерархическая рекурсивная связь. Пусть нам дан следующий класс сущностей, состоящий, как и предыдущий пример, из атрибутов «Код Предка» и «Код Узла». Сначала покажем табличную форму представления этого класса сущности:

Код предка	**Код Узла**	**…**
Null	1	…
1	2	…
1	3	…
2	4	…
2	5	…
3	6	…

А теперь построим диаграмму, представляющую этот класс сущностей. Для этого выделим из таблицы все необходимые для этого сведения: предка у узла с кодом «единица» не существует или не определен, из этого делаем вывод, что узел «единица» является вершиной. Этот же самый узел «единица» является предком для узлов с кодом «два» и «три». В свою очередь, у узла с кодом «два» имеются

два потомка: узел с кодом «четыре» и узел с кодом «пять». А у узла с кодом «три» — только один потомок — узел с кодом «шесть».

Итак, с учетом всего вышесказанного построим древовидную структуру, отражающую информацию о данных, заложенную в предыдущей таблице:



Итак, мы увидели, что представлять древовидные структуры действительно удобно при помощи иерархической рекурсивной связи.

2. Сетевая рекурсивная связь

Сетевая рекурсивная связь классов сущностей между собой является как бы многомерным аналогом уже пройденной нами иерархической рекурсивной связи.

Только если иерархическая рекурсия определялась как рекурсивная связь типа «не более одного ко многим», то **сетевая рекурсия** представляет собой такую же рекурсивную связь, только уже типа «многие ко многим». Из-за того что в этой связи с каждой стороны участвует много классов сущностей, ее и называют сетевой.

Как уже можно догадаться по аналогии с рекурсией иерархической, связи вида сетевой рекурсии предназначены для представления графовых структур данных (тогда как иерархические связи применяются, как мы помним, исключительно для реализации древовидных структур).

Но, так как в связи вида сетевой рекурсии заданы связи типа именно «многие ко многим», без их дополнительной детализации не

обойтись. Поэтому для уточнения всех имеющихся в схеме связей типа «многие ко многим» становится необходимым создать новый самостоятельный класс сущностей, содержащий все ссылки на предка или потомка связи «Предок — Потомок». Такой класс в общем случае называется **классом ассоциативных сущностей**.

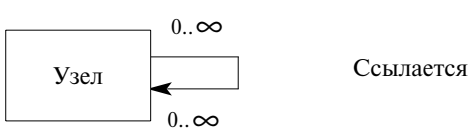
В нашем частном случае (в базах данных, подлежащих рассмотрению в нашем курсе) ассоциативная сущность не имеет собственных дополнительных атрибутов и называется **именующей**, так как именует связи «Предок — Потомок» путем ссылок на них. Таким образом, первичный ключ класса сущностей, представляющего узлы сети, должен дважды мигрировать в классы ассоциативных сущностей. В этом классе мигрировавшие ключи в совокупности должны образовывать составной первичный ключ.

Из всего вышесказанного можно сделать вывод, что устанавливающие связи при использовании сетевой рекурсии должны быть не полностью идентифицирующими и никакими другими.

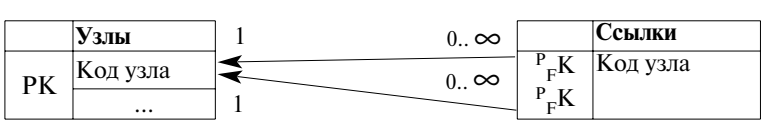
Так же как и при использовании иерархической рекурсивной связи, при применении в качестве связи сетевой рекурсии ни один атрибут не может появляться дважды в одном классе сущностей под одним и тем же именем. Поэтому, как и в прошлый раз, специально оговаривается, что все атрибуты мигрирующего ключа обязательно должны получить имя роли.

Для иллюстрирования работы сетевой рекурсивной связи, построим презентационную и ключевую диаграммы, реализующие сетевую рекурсию в реляционной модели данных.

Сначала представим презентационную диаграмму:



А теперь построим более подробную ключевую диаграмму:



Что мы здесь видим? А видим мы, что обе связи, имеющиеся в данной ключевой диаграмме, являются связями вида «многие к одному». Причем кратность «0... ∞ » или кратность «много» стоит на

конце связи, обращенной к именующему классу сущностей. Действительно, ведь ссылок много, а ссылаются они все на какой-то один код узла, являющийся первичным ключом класса сущностей «Узлы».

И, наконец, рассмотрим пример, иллюстрирующий работу такого вида связи классом сущностей как сетевая рекурсия. Пусть нам дано табличное представление некоторого класса сущностей, а также именующий класс сущностей, содержащий информацию о ссылках. Приведем эти таблицы.

Узлы:

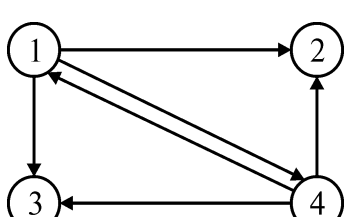
Код Узла	**…**
1	…
2	…
3	…
4	…

Ссылки:

Код Узла Предка	**Код Узла Потомка**
1	2
1	3
1	4
4	1
4	2
4	3

Действительно, вышеприведенное представление исчерпывающе: оно дает всю необходимую информацию для того, чтобы без труда воспроизвести зашифрованную здесь графовую структуру. Например, мы без всяких препятствий можем увидеть, что у узла с кодом «один» имеются три потомка соответственно с кодами «два», «три» и «четыре». Также мы видим, что у узлов с кодами «два» и «три» потомков не имеется вообще, а у узла с кодом «четыре» имеются (также как и у узла «один») три потомка с кодами «один», «два» и «три».

Изобразим граф, заданный классами сущностей, приведенными выше:



Итак, только что построенный нами граф и является теми данными, для связывания классов сущностей которых и использовалась связь вида сетевой рекурсии.

3. Ассоциация

Из всех видов связей, входящих в рассмотрение нашего конкретного курса лекций, рекурсивными связями являются только две. Мы их уже успели рассмотреть, это соответственно иерархическая и сетевая рекурсивные связи.

Все остальные виды связей, которые нам предстоит рассмотреть, не являются рекурсивными, а представляют собой, как правило, связь нескольких родительских и нескольких дочерних классов сущностей. Причем, как можно догадаться, родительские и дочерние классы сущностей теперь уже никогда не будут совпадать (действительно, ведь речь уже не идет о рекурсии).

Связь, о которой пойдет речь в этом параграфе лекции, называется ассоциацией и относится как раз к нерекурсивному виду связей.

Итак, связь, называемая **ассоциацией**, реализуется как взаимосвязь между несколькими родительскими классами сущностей и одним дочерним классом сущностей. И при этом, что любопытно, эта взаимосвязь описывается связями различных типов.

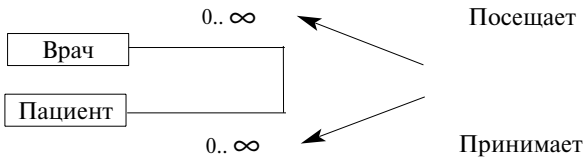
Также стоит отметить, что родительский класс сущностей при ассоциации может быть и один, как в сетевой рекурсии, но даже в такой ситуации число связей, идущих от дочернего класса сущностей, должно быть не менее двух.

Интересно, что при ассоциации, так же как и при сетевой рекурсии, существуют специальные виды классов сущностей. Примером такого класса является дочерний класс сущностей. Ведь в общем случае в ассоциации дочерний класс сущностей называется **классом ассоциативных сущностей**. В частном случае, когда класс ассоциативных сущностей не имеет собственных дополнительных атрибутов и содержит только атрибуты, мигрирующие вместе с первичными ключами из родительских классов сущностей, такой класс называется **классом именующих сущностей**. Как можно обратить внимание, при этом прослеживается почти абсолютная аналогия с понятием ассоциативных и именующих сущностей в сетевой рекурсивной связи.

Чаще всего ассоциация используется для детализации (разрешения) связей вида «многие ко многим».

Проиллюстрируем это утверждение.

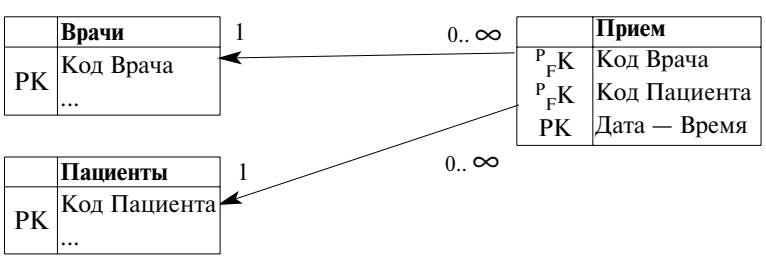
Пусть, например, нам дана следующая презентационная диаграмма, описывающая схему приема некоторого врача в некой больнице:



Эта диаграмма буквально означает, что в больнице имеется много врачей и много пациентов, и больше никак отношения и соответствия между врачами и пациентами не отражено. Таким образом, разумеется, что с такой базой данных в администрации больницы никогда не было бы понятно, как проводить приемы у различных врачей различных пациентов. Ясно, что использованные здесь связи типа «многие ко многим» просто необходимо детализировать, чтобы конкретизировать отношения между различными врачами и пациентами, другими словами, чтобы рационально организовать расписание приемов всех имеющихся в больнице врачей и их пациентов.

А теперь построим более подробную ключевую диаграмму, в которой мы уже детализируем все имеющиеся связи «многие ко многим». Для этого мы соответственно введем новый класс сущностей, назовем его «Прием», который будет выступать в роли класса ассоциативных сущностей (позже мы посмотрим, почему именно это будет классом ассоциативных сущностей, а не просто классом именующих сущностей, о которых мы говорили ранее).

Итак, наша ключевая диаграмма будет выглядеть следующим образом:



Итак, теперь наглядно видно, почему новый класс «Прием» не является классом именующих сущностей. Ведь этот класс имеет свой дополнительный атрибут «Дата — Время», поэтому согласно определению нововведенный класс «Прием» и является классом ассоциативных сущностей. Этот класс «ассоциирует» классы сущностей «Врачи» и «Паци-

енты» друг с другом посредством времени, в которое и проводится тот или иной прием, что делает работу с такой базой данных гораздо удобнее. Таким образом, мы, введя атрибут «Дата — Время», буквально организовали так необходимое расписание работы различных врачей.

Также мы видим, что внешний первичный ключ «Код Врача» класса сущностей «Прием» ссылается на одноименный первичный ключ класса сущностей «Врачи». И аналогично внешний первичный ключ «Код Пациента» класса сущностей «Прием» ссылается на одноименный первичный ключ класса сущностей «Пациенты». В данном случае, что само собой разумеется, классы сущностей «Врачи» и «Пациенты» являются родительскими, а класс ассоциативных сущностей «Прием», в свою очередь, является единственным дочерним.

Мы видим, что теперь имеющаяся в прежней презентационной диаграмме связь типа «многие ко многим» полностью детализирована. Вместо одной связи «многие ко многим», какую мы видим в презентационной диаграмме, приведенной ранее, у нас имеется две связи типа «многие к одному». На дочернем конце первой связи стоит кратность «много», это буквально означает, что в классе сущностей «Прием» записано много врачей (все, которые есть в больнице). А на родительском конце этой связи стоит кратность «один», что это значит? А значит это, что в классе сущностей «Прием» каждый из имеющихся кодов каждого конкретного врача может встречаться неограниченно много раз. Действительно, ведь в расписании в больнице код одного и того же врача встречается много раз, в разные дни и время. А вот этот же код, но уже в классе сущностей «Врачи» может встретиться один и только один раз. Действительно, ведь в списке всех врачей больницы (а класс сущностей «Врачи» представляет собой не что иное, как такой список) код каждого конкретного врача может присутствовать только один раз.

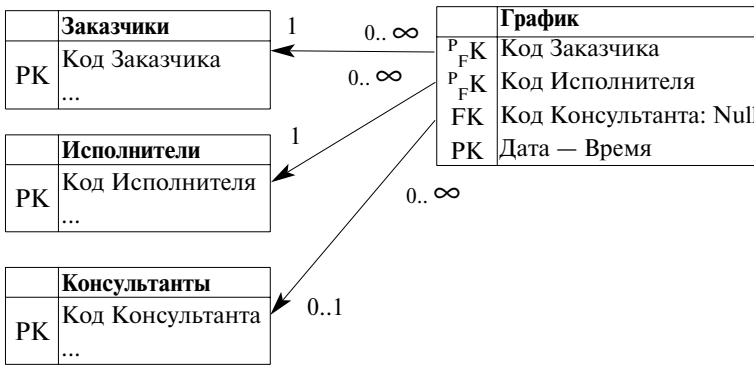
Аналогичное происходит и со связью между родительским классом «Пациенты» и дочерним классом «Пациенты». В списке всех пациентов больницы (в классе сущностей «Пациенты») код каждого конкретного пациента может встретиться только один раз. Но зато в расписании приемов (в классе сущностей «Прием») каждый код конкретного пациента может встретиться сколь угодно много раз. Именно поэтому кратности на концах связи расставлены как раз таким образом.

В качестве примера реализации ассоциации в реляционной модели данных построим модель, описывающую график встреч заказчика с исполнителем при необязательном участии консультантов.

Не будем останавливаться на презентационной диаграмме, потому что нам необходимо рассмотреть построение диаграмм во всех

подробностях, а презентационная диаграмма такой возможности предоставить не может.

Итак, построим ключевую диаграмму, отражающую суть отношений между заказчиком, исполнителем и консультантом.



Итак, начнем подробный разбор приведенной ключевой диаграммы.

Во-первых, класс «График» является классом ассоциативных сущностей, но, так же как и в прошлом примере, не является классом именующихся сущностей, ведь у него есть атрибут, не мигрирующий в него вместе с ключами, а являющийся его собственным атрибутом. Это атрибут «Дата — Время».

Во-вторых, мы видим, что атрибуты дочернего класса сущностей «График» «Код заказчика», «Код исполнителя» и «Дата — Время» образуют составной первичный ключ этого класса сущностей. Атрибут «Код консультанта» является просто внешним ключом класса сущностей «График». Обратим внимание, что этот атрибут допускает среди своих значений Null-значения, ведь по условию присутствие на встрече консультанта не обязательно.

Далее, в-третьих, заметим, что первые две связи (из трех имеющихся связей) являются не полностью идентифицирующими. Именно не полностью идентифицирующими, потому что мигрирующий ключ в обоих случаях (первичные ключи «Код заказчика» и «Код исполнителя») не полностью формирует первичный ключ класса сущностей «График». Действительно, ведь остается атрибут «Дата — Время», который также является частью составного первичного ключа.

На концах обеих этих не полностью идентифицирующих связей проставлены кратности «один» и «много». Это сделано для того, чтобы показать (как и в примере о врачах и пациентах) разницу, между упоминанием кода заказчика или исполнителя в разных классах сущностей. Действительно, в классе сущностей «График» любой код заказчика или

исполнителя может встречаться сколь угодно много раз. Поэтому на этом, дочернем, конце связи стоит кратность «много». А в классе сущностей «Заказчики» или «Исполнители» каждый из кодов соответственно заказчика или исполнителя может встречаться один и только один раз, ведь эти классы сущностей являются каждый не чем иным, как полным списком всех заказчиков и исполнителей. Поэтому на этом, родительском конце связи, и стоит кратность «один».

И, наконец, заметим, что третья связь, а именно связь класса сущностей «График» с классом сущностей «Консультанты», является не обязательно не идентифицирующей.

Действительно, ведь в этом случае речь идет о переносе ключевого атрибута «Код консультанта» класса сущностей «Консультанты» в одноименный неключевой атрибут класса сущностей «График», т. е. первичный ключ класса сущностей «Консультанты» в классе сущностей «График» не идентифицирует первичного ключа уже этого класса. И, кроме того, как уже было упомянуто ранее, атрибут «Код консультанта» допускает Null-значения, поэтому здесь и используется именно не полностью не идентифицирующая связь. Таким образом, атрибут «Код консультанта» приобретает статус внешнего ключа и ничего более того.

Также обратим внимание на кратности связей, поставленных на родительском и дочернем концах этой не полностью не идентифицирующей связи. На ее родительском конце стоит кратность «не более одного». Действительно, если вспомнить определение не полностью не идентифицирующей связи, то мы поймем, что атрибуту «Код консультанта» из класса сущностей «График» не может соответствовать более одного кода консультанта из списка всех консультантов (которым является класс сущностей «Консультанты»). Да и вообще может так получиться, что ему не будет соответствовать ни одного кода консультанта (вспомним о флажке допустимости Null-значений Код консультанта: Null), ведь по условию присутствие консультанта на встрече заказчика и исполнителя, вообще говоря, не обязательно.

4. Обобщения

Очередным видом связи классов сущностей между собой, который мы рассмотрим, является связь вида **обобщение**. Это также нерекурсивный вид связи.

Итак, связь типа **обобщение** реализуется как взаимосвязь одного родительского класса сущностей с несколькими дочерними классами сущностей (в отличие от предыдущей рассмотренной связи Ассо-

циации, в которой речь шла о нескольких родительских классах сущностей и одним дочернем классе сущностей).

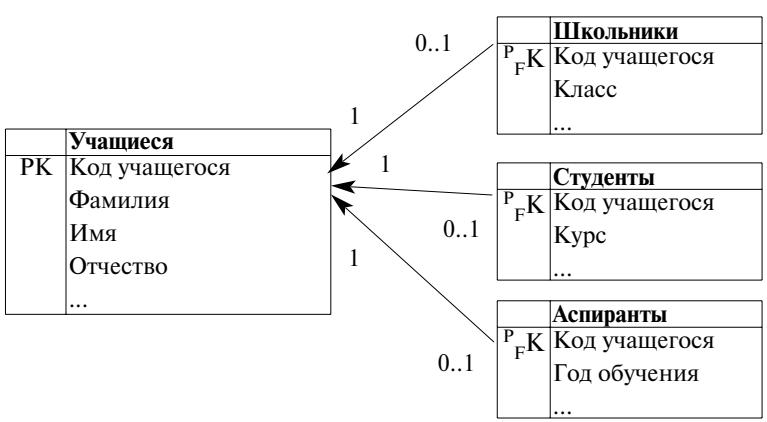
При формулировании правил представления данных при помощи связи Обобщения необходимо сразу сказать, что эта взаимосвязь одного родительского класса сущностей и нескольких дочерних классов сущностей, описывается полностью идентифицирующими связями, т. е. категориальными связями. Вспоминая определение полностью идентифицирующих связей, мы приходим к выводу, что при использовании Обобщения каждый атрибут первичного ключа родительского класса сущностей переносится в состав первичного ключа классов сущностей дочерних, т. е. атрибуты первичного мигрирующего ключа родительского класса сущностей полностью формируют первичные ключи всех дочерних классов сущностей, они их идентифицируют.

Любопытно отметить, что при Обобщении реализуется так называемая **иерархия категорий** или иерархия наследования.

При этом родительский класс сущностей определяет **класс обобщенных сущностей**, характеризующийся атрибутами, общими для сущностей всех дочерних классов или так называемых **категориальных сущностей** т. е. родительский класс сущностей представляет собой буквальное обобщение всех своих дочерних классов сущностей.

В качестве примера реализации обобщения в реляционной модели данных построим следующую модель. Эта модель будет основана на обобщенном понятии «Учащиеся» и будет описывать следующие категориальные понятия (т. е. будет обобщать следующие дочерние классы сущностей): «Школьники», «Студенты» и «Аспиранты».

Итак, построим ключевую диаграмму, отражающую суть взаимоотношений между родительским классом сущности и дочерними классами сущностей, описываемых связью типа Обобщение.



Итак, что же мы видим?

Во-первых, каждому из базовых отношений (или из классов сущностей, что одно и то же) «Школьники», «Студенты» и «Аспиранты» соответствуют свои собственные атрибуты, как то «Класс», «Курс» и «Год обучения». Каждый из этих атрибутов характеризует участников своего собственного класса сущностей. Еще мы видим, что первичный ключ родительского класса сущностей «Учащиеся» мигрирует в каждый дочерний класс сущностей и формирует там первичный внешний ключ. При помощи этих связей мы можем по коду любого учащегося определить его имя, фамилию и отчество, информацию о которых мы не найдем в самих соответствующих дочерних классах сущностей.

Во-вторых, так как мы говорим о полностью идентифицирующей (или категориальной) связи классов сущностей, то обратим внимание на кратности связей между родительским классом сущностей и его дочерними классами. На родительском конце каждой из этих связей стоит кратность «один», а на каждом дочернем конце связей стоит кратность «не более одного». Если вспомнить определение полностью идентифицирующей связи классов сущностей, то становится понятно, что действительно единственный в своем роде код учащегося, являющийся первичным ключом класса сущностей «Учащиеся», задает не более одного атрибута с таким кодом в каждом дочернем классе сущностей «Школьники», «Студенты» и «Аспиранты». Поэтому все связи имеют именно такие кратности.

Запишем фрагмент операторов создания базовых отношений «Школьники» и «Студенты» с определением правил поддержания ссылочной целостности типа cascade. Итак, имеем:

```
Create table Школьники
...
primary key (Код ученика)
foreign key (Код ученика) references Учащиеся (Код ученика)
on update cascade
on delete cascade

Create table Студенты
...
primary key (Код студента)
foreign key (Код студента) references Учащиеся (Код студента)
on update cascade
on delete cascade;
```

Таким образом, мы видим, что в дочернем классе сущностей (или отношений) «Школьники» задается первичный внешний ключ, ссы-

лающийся на родительский класс сущностей (или отношение) «Учащиеся». Правило cascade поддержания ссылочной целостности определяет, что при удалении или при обновлении атрибутов родительского класса сущностей «Учащиеся» соответствующие им атрибуты дочернего отношения «Школьники» будут автоматически (каскадом) обновляться или удаляться. Аналогично при удалении или при обновлении атрибутов родительского класса сущностей «Учащиеся» соответствующие им атрибуты дочернего отношения «Студенты» также будут автоматически обновляться или удаляться.

Необходимо заметить, что здесь используется именно это правило поддержания ссылочной целостности, потому что в данном контексте (перечень учащихся) не рационально запрещать удаление и обновление информации, а также присваивать неопределенное значение вместо реальных сведений.

А теперь приведем пример классов сущностей, описанных в предыдущей диаграмме, только представленных в табличной форме. Итак, имеем следующие таблицы-отношения:

Учащиеся — родительское отношение, объединяющее в себе информацию об атрибутах всех остальных отношений:

Код учащегося	Фамилия	Имя	Отчество	...
1	Заботин	Николай	Владимирович	...
2	Казакова	Наталия	Александровна	...
3	Иголкина	Анастасия	Валерьевна	...
...	...	...	...	...

Школьники — дочернее отношение:

Код учащегося	Класс	...
1	7	...
...	...	...

Студенты — второе дочернее отношение:

Код учащегося	Курс	...
2	3	...
...	...	...

Аспиранты — третье дочернее отношение:

Код учащегося	Год обучения	...
3	1	...
...	...	...

Итак, действительно, мы видим, что в дочерних классах сущностей не прописана информация о фамилии, имени и отчестве учащихся, т. е. школьников, студентов и аспирантов. Эту информацию можно получить только посредством ссылок на родительский класс сущностей.

Также мы видим, что различные коды учащихся в классе сущностей «Учащиеся» могут соответствовать различным дочерним классам сущностей. Так, про учащегося с кодом «1» Заботина Николая в родительском отношении неизвестно ничего, кроме его имени, а всю остальную информацию (кто он, школьник, студент или аспирант) можно узнать только обратившись к соответствующему дочернему классу сущностей (определяется по коду).

Аналогичным образом необходимо работать с остальными учащимися, чьи коды указаны в родительском классе сущностей «Учащиеся».

5. Композиция

Связь классов сущностей типа композиция, так же как и две предыдущие, не принадлежит к виду рекурсивной связи.

Композиция (или, как ее еще иногда называют, **композитная агрегация**) — это взаимосвязь одного родительского класса сущностей с несколькими дочерними классами сущностей, так же как и предыдущая рассмотренная нами связь. Обобщение.

Но если обобщение определялось как взаимосвязь классов сущности, описывающаяся полностью идентифицирующимися связями, то композиция, в свою очередь, описывается не полностью идентифицирующими связями, т. е. при композиции каждый атрибут первичного ключа родительского класса сущностей мигрирует в ключевой атрибут дочернего класса сущностей. И при этом атрибуты мигрирующего ключа лишь частично формируют первичный ключ дочернего класса сущностей.

Итак, при композитной агрегации (при композиции) родительский класс сущностей (или **агрегат**) связывается с несколькими дочерними классами сущностей (или **компонентами**). При этом компоненты агрегата (т. е. компоненты родительского класса сущностей) ссылаются на агрегат посредством внешнего ключа, входящего в состав первичного ключа и, следовательно, не могут существовать вне агрегата.

Вообще композитная агрегация представляет собой усиленную форму простой агрегации (о которой мы поговорим чуть

дальше). Композиция (или композитная агрегация) характеризуется тем, что:

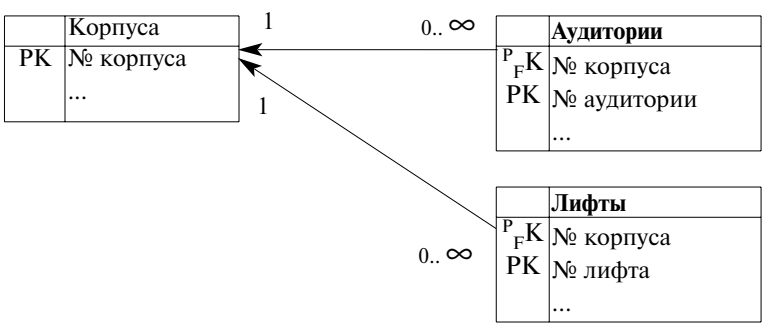
1) ссылка на агрегат участвует в идентификации компонентов;

2) эти компоненты не могут существовать вне агрегата.

Агрегация (связь, которую мы будем рассматривать дальше) с обязательно не идентифицирующими связями также не позволяет компонентам существовать вне агрегата и поэтому близка по смыслу к описанной выше реализации композитной агрегации.

Построим ключевую диаграмму, описывающую взаимосвязь между одним родительским классом сущностей и несколькими дочерними классами сущностей, т. е. описывающую связь классов сущностей типа композитной агрегации.

Пусть это будет ключевая диаграмма, изображающая состав корпусов некого учебного городка, включающего в себя корпуса, их аудитории и лифты. Итак, эта диаграмма будет иметь следующий вид:



Итак, рассмотрим только что построенную диаграмму.

Что мы в ней видим?

Во-первых, мы видим, что связь, использованная в этой композитной агрегации, действительно идентифицирующая и действительно не полностью идентифицирующая. Ведь первичный ключ родительского класса сущностей «Корпуса» участвует в формировании первичного ключа дочерних классов сущностей «Аудитории» и «Лифты», но не определяет его полностью. Первичный ключ «№ корпуса» родительского класса сущностей мигрирует во внешние первичные ключи «№ корпуса» обоих дочерних классов, но, кроме этого мигрировавшего, ключа у обоих дочерних классов сущностей существует и свой собственный первичный ключ, соответственно «№ аудитории» и «№ лифта», т. е .составные первичные ключи дочерних классов сущностей лишь частично оказываются сформированными атрибутами первичного ключа родительского класса сущностей.

Теперь разберемся с кратностями связей, соединяющих родительский и оба дочерних класса. Так как мы имеем дело с не полностью идентифицирующими связями, то кратности присутствуют такие: «один» и «много». Кратность «один» присутствует на родительском конце обеих связей и символизирует то, что в списке всех имеющихся корпусов (а класс сущностей «Корпуса» является именно таким списком) каждый номер может встретиться только один, (и не более того) раз. А, в свою очередь, среди атрибутов классов «Аудитории» и «Лифты» каждый номер корпуса может встретиться много раз, так как аудиторий (или лифтов) больше, чем корпусов, и в каждом корпусе — несколько и аудиторий, и лифтов. Таким образом, при перечислении всех аудиторий и лифтов мы неминуемо будем повторять номера корпусов.

И, наконец, как и при рассмотрении предыдущего вида связи, запишем фрагменты операторов создания базовых отношений (или, что одно и то же, классов сущностей) «Аудитории» и «Лифты», причем сделаем это с определением правил поддержания ссылочной целостности типа cascade.

Итак, этот оператор будет выглядеть следующим образом:

```
Create table Аудитории
...
primary key (№ корпуса, № аудитории)
foreign key (№ корпуса) references Корпуса (№ корпуса)
on update cascade
on delete cascade

Create table Лифты
...
primary key (№ корпуса, № лифта)
foreign key (№ корпуса) references Корпуса (№ корпуса)
on update cascade
on delete cascade;
```

Таким образом, мы и задали все необходимые первичные и внешние ключи дочерних классов сущностей. Правило поддержания ссылочной целостности мы снова взяли cascade, так как уже описали его как наиболее рациональный.

Теперь приведем пример в табличной форме всех только что рассмотренных нами классов сущностей. Опишем те базовые отношения, которые отразили при помощи диаграммы, в виде таблиц, и для наглядности введем туда некоторое количество показательных данных.

Корпуса — родительское отношение имеет следующий вид:

№ **корпуса**	**...**
9	...
10	...
...	...

Аудитории — дочерний класс сущностей:

№ **корпуса**	№ **аудитории**	**...**
9	300	...
...	...	...

Лифты — второй дочерний класс сущностей родительского класса «Корпуса»:

№ **корпуса**	№ **лифта**	**...**
9	1	...
...	...	...

Итак, мы можем видеть, каким образом организована информация по всем корпусам, их аудиториям и лифтам в этой базе данных, которую вполне может использовать любое реально существующее учебное заведение.

6. Агрегация

Агрегация — это последний вид связи между классами сущностей, который подлежит рассмотрению в рамках нашего курса. Она также не является рекурсивной, и один из двух ее видов довольно близок по смыслу к уже рассмотренной ранее композитной агрегации.

Итак, **агрегация** — это взаимосвязь одного родительского класса сущностей с несколькими дочерними классами сущностей. При этом взаимосвязи могут быть описаны связями двух видов:

1) обязательно не идентифицирующими связями;

2) необязательно не идентифицирующими связями.

Напомним, что при обязательно не идентифицирующих связях некоторые атрибуты первичного ключа родительского класса сущностей переносятся в неключевой атрибут дочернего класса, и при этом Null-значения для всех атрибутов мигрирующего ключа запрещены. А при

не обязательно не идентифицирующих связях миграция первичных ключей происходит по точно такому же принципу, но при этом Null-значения для некоторых атрибутов мигрирующего ключа разрешены.

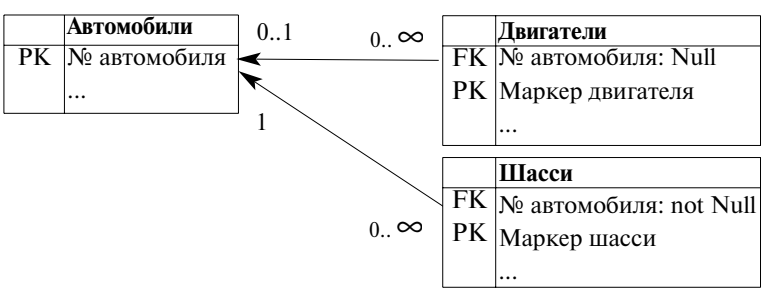
При агрегации, родительский класс сущностей (или **агрегат**) связывается с несколькими дочерними классами сущностей (или **компонентами**). Компоненты агрегата (т. е. родительского класса сущностей) ссылаются на агрегат посредством внешнего ключа, не входящего в состав первичного ключа, и, следовательно, в случае **не обязательно не идентифицирующих связей**, компоненты агрегата могут существовать вне агрегата.

В случае использования агрегации с обязательно не идентифицирующими связями компонентам агрегата не позволяется существовать вне агрегата, и в этом смысле агрегация с обязательно не идентифицирующими связями близка к композитной агрегации.

Теперь, когда стало понятно, что собой представляет связь типа агрегации, построим ключевую диаграмму, описывающую работу этой связи.

Пусть наша будущая диаграмма описывает маркированные компоненты автомобилей (а именно двигатель и шасси). При этом, будем считать, что списывание автомобиля предполагает и списывание шасси вместе с ним, но не предполагает одновременное списывание двигателя.

Итак, наша ключевая диаграмма имеет следующий вид:



Итак, что же мы видим на этой ключевой диаграмме?

Во-первых, связь родительского класса сущностей «Автомобили» с дочерним классом сущностей «Двигатели» является не обязательно не идентифицирующей, потому что атрибут «№ автомобиля» допускает среди своих значений Null-значения. В свою очередь, Null-значения этот атрибут допускает по той причине, что списывание двигателя, по условию не зависит от списывания всего автомобиля и, следовательно, при списывании автомобиля происходит не обязательно. Также мы видим, что первичный ключ «№ двигателя» класса сущностей «Автомобили» мигрирует в неключевой атрибут «№ двигателя»

класса сущностей «Двигатели». И при этом данный атрибут приобретает статус внешнего ключа. А первичным ключом в этом классе сущностей «Двигатели» является атрибут «Маркер двигателя», который не ссылается ни на какой атрибут родительского отношения.

Во-вторых, связь родительского класса сущностей «Двигатели» и дочернего класса сущностей «Шасси» — это обязательно не идентифицирующая связь, потому что атрибут внешнего ключа «№ автомобиля» не допускает среди своих значений Null-значения. Это, в свою очередь, происходит потому, что по условию известно, что списывание автомобиля предполагает обязательное одновременное списывание и шасси. Здесь, так же, как и в случае предыдущей связи, первичный ключ родительского класса сущностей «Двигатели» мигрирует в неключевой атрибут «№ автомобиля» дочернего класса сущностей «Шасси». При этом первичным ключом этого класса сущностей является атрибут «Маркер шасси», который не ссылается ни на какой атрибут родительского отношения «Двигатели».

Идем дальше. Для наилучшего усвоения пройденной темы, запишем снова фрагменты операторов создания базовых отношений «Двигатели» и «Шасси» с определением правил поддержания ссылочной целостности.

```
Create table Двигатели
...
primary key (Маркер двигателя)
foreign key (№ автомобиля) references Автомобили (№ автомобиля)
on update cascade
on delete set Null

Create table Шасси
...
primary key (Маркер шасси)
foreign key (№ автомобиля) references Автомобили (№ автомобиля)
on update cascade
on delete cascade;
```

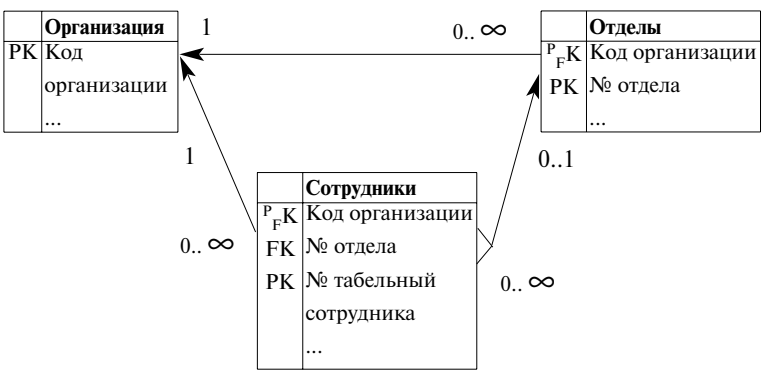
Мы видим, что правило поддержания ссылочной целостности мы использовали везде одно — cascade, так как еще раньше признали его наиболее рациональным из всех. Однако на этот раз мы использовали (помимо правила cascade) еще и правило поддержания ссылочной целостности set Null. Причем использовали мы его при следующем

условии: если какое-то значение первичного ключа «№ автомобиля» из родительского класса сущностей «Автомобили» будет удалено, то значению ссылающегося на него внешнего ключа «№ автомобиля» дочернего отношения «Двигатели» будет присвоено Null-значение.

7. Унификация атрибутов

Если при миграции первичных ключей некоего родительского класса сущностей в один и тот же дочерний класс попадают совпадающие по смыслу атрибуты из разных родительских классов, то эти атрибуты необходимо «слить», т. е. необходимо провести так называемую **унификацию атрибутов**.

Например, в случае, когда сотрудник может работать в организации, числясь не более чем в одном отделе, после унификации атрибута «Код организации» получим следующую ключевую диаграмму:



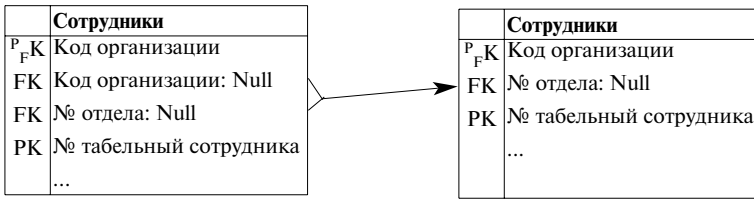
При миграции первичный ключ из родительских классов сущностей «Организация» и «Отделы» в дочерний класс «Сотрудники», атрибут «Код организации» попадает в класс сущностей «Сотрудники». Причем дважды:

1) первый раз с маркером $^{P}_{F}K$ из класса сущностей «Организация» при установлении не полностью идентифицирующей связи;

2) и второй раз, с маркером FK с условием допустимости Null-значений из класса сущностей «Отделы» при установлении не обязательно не идентифицирующей связи.

При унификации атрибут «Код организации» получает статус атрибута первичного / внешнего ключа, поглощающего статус атрибута внешнего ключа.

Построим новую ключевую диаграмму, демонстрирующую сам процесс унификации:



Таким образом и произошла унификация атрибутов.

ЛЕКЦИЯ № 13. Экспертные системы и продукционная модель знаний

1. Назначение экспертных систем

Для ознакомления с таким новым для нас понятием, как **экспертные системы** мы, для начала, пройдемся по истории создания и разработки направления «экспертные системы», а потом определим и само понятие экспертных систем.

В начале 80-х гг. XX в. в исследованиях по созданию искусственного интеллекта сформировалось новое самостоятельное направление, получившее название **экспертных систем**. Цель этих новых исследований по экспертным системам состоит в разработке специальных программ, предназначенных для решения особых видов задач. Что это за особый вид задач, потребовавший создания целой новой инженерии знаний? К этому особому виду задач могут быть отнесены задачи из абсолютно любой предметной области. Главное, что отличает их от задач обычных, — это то, что человеку-эксперту решить их представляется очень сложным заданием. Тогда и была разработана первая так называемая **экспертная система** (где в роли эксперта выступал уже не человек, а машина), причем экспертная система получает результаты, не уступающие по качеству и эффективности решениям, получаемым обычным человеком — экспертом. Результаты работы экспертных систем могут быть объяснены пользователю на очень высоком уровне. Данное качество экспертных систем обеспечивается их способностью рассуждать о собственных знаниях и выводах. Экспертные системы вполне могут пополнять собственные знания в процессе взаимодействия с экспертом. Таким образом, их можно с полной уверенностью ставить в один ряд с вполне оформившимся искусственным интеллектом.

Исследователи в области экспертных систем для названия своей дисциплины часто используют также уже упоминавшийся ранее термин «инженерия знаний», введенный немецким ученым Е. Фейгенбаумом как «привнесение принципов и инструментария исследований из области искусственного интеллекта в решение трудных прикладных проблем, требующих знаний экспертов».

Однако коммерческие успехи к фирмам-разработчикам пришла не сразу. На протяжении четверти века в период с 1960 по 1985 гг. успехи искусственного интеллекта касались в основном исследовательских разработок. Тем не менее, начиная примерно с 1985 г., а в массовом масштабе с 1987 по 1990 гг. экспертные системы стали активно использоваться в коммерческих приложениях.

Заслуги экспертных систем довольно велики и состоят в следующем:

1) технология экспертных систем существенно расширяет круг практически значимых задач, решаемых на персональных компьютерах, решение которых приносит значительный экономический эффект и существенно упрощает все связанные с ними процессы;

2) технология экспертных систем является одним из самых важных средств в решении глобальных проблем традиционного программирования, таких как продолжительность, качество и, следовательно, высокая стоимость разработки сложных приложений, вследствие которой значительно снижался экономический эффект;

3) имеется высокая стоимость эксплуатации и обслуживания сложных систем, которая зачастую в несколько раз превосходит стоимость самой разработки, а также низкий уровень повторной используемости программ и т. п.;

4) объединение технологии экспертных систем с технологией традиционного программирования добавляет новые качества к программным продуктам за счет, во-первых, обеспечения динамичной модификации приложений рядовым пользователем, а не программистом; во-вторых, большей «прозрачности» приложения, лучшей графики, интерфейса и взаимодействия экспертных систем.

По мнению рядовых пользователей и ведущих специалистов, в недалекой перспективе экспертные системы найдут следующее применение:

1) экспертные системы будут играть ведущую роль на всех стадиях проектирования, разработки, производства, распределения, отладки, контроля и оказания услуг;

2) технология экспертных систем, получившая широкое коммерческое распространение, обеспечит революционный прорыв в интеграции приложений из готовых интеллектуально-взаимодействующих модулей.

В общем случае экспертные системы предназначены для так называемых **неформализованных задач**, т. е. экспертные системы не отвергают и не заменяют традиционного подхода к разработке программ,

ориентированного на решение формализованных задач, но дополняют их, тем самым значительно расширяя возможности. Именно этого и не может сделать простой человек-эксперт.

Такие сложные неформализованные задачи характеризуются:

1) ошибочностью, неточностью, неоднозначностью, а также неполнотой и противоречивостью исходных данных;

2) ошибочностью, неоднозначностью, неточностью, неполнотой и противоречивостью знаний о проблемной области и решаемой задаче;

3) большой размерностью пространства решений конкретной задачи;

4) динамической изменчивостью данных и знаний непосредственно в процессе решения такой неформализованной задачи.

Экспертные системы главным образом основаны на эвристическом поиске решения, а не на исполнении известного алгоритма. В этом одно из главных преимуществ технологии экспертных систем перед традиционным подходом к разработке программ. Именно это и позволяет им так хорошо справляться с поставленными перед ними задачами.

Технология экспертных систем используется для решения самых различных задач. Перечислим основные из подобных задач.

1. **Интерпретация.**

Экспертные системы, выполняющие интерпретацию, чаще всего применяют показания различных приборов с целью описания положения дел.

Интерпретирующие экспертные системы способны обрабатывать самые различные виды информации. Примером может послужить использование данных спектрального анализа и изменения характеристик веществ для определения их состава и свойств. Также примером может служить интерпретация показаний измерительных приборов в котельной для описания состояния котлов и воды в них.

Интерпретирующие системы чаще всего имеют дело непосредственно с показаниями. В связи с этим возникают затруднения, которых нет у других видов систем. Что это за затруднения? Эти затруднения возникают из-за того, что экспертным системам приходится интерпретировать засоренную лишним, неполную, ненадежную или неверную информацию. Отсюда неизбежны либо ошибки, либо значительное увеличение обработки данных.

2. **Прогнозирование.**

Экспертные системы, осуществляющие прогноз чего-либо, определяют вероятностные условия заданных ситуаций. Примерами слу-

жат прогноз ущерба, причиненного урожаю хлебов неблагоприятными погодными условиями, оценивание спроса на газ на мировом рынке, прогнозирование погоды по данным метеорологических станций. Системы прогнозирования иногда применяют моделирование, т. е. такие программы, которые отображают некоторые взаимосвязи в реальном мире, чтобы воссоздать их в среде программирования, и потом спроектировать ситуации, которые могут возникнуть при тех или иных исходных данных.

3. **Диагностика различных приборов.**

Экспертные системы производят такую диагностику, применяя описания какой-либо ситуации, поведения или данных о строении различных компонентов, чтобы определить возможные причины неисправно работающей диагностируемой системы. Примерами служат установление обстоятельств заболевания по симптомам, которые наблюдаются у больных (в медицине); определение неисправностей в электронных схемах и определение неисправных компонентов в механизмах различных приборов. Системы диагностики довольно часто являются помощниками, которые не только ставят диагноз, но и помогают в устранении неполадок. В таких случаях данные системы вполне могут взаимодействовать с пользователем, чтобы оказать помощь при поиске неполадок, а потом привести список действий, необходимых для их устранения. В настоящее время многие диагностические системы разрабатываются в качестве приложений к инженерному делу и компьютерным системам.

4. **Планирование различных событий.**

Экспертные системы, предназначенные для планирования, проектируют различные операции. Системы предопределяют практически полную последовательность действий, прежде чем начнется их реализация.

Примерами такого планирования событий могут служить создания планов военных действий как оборонительного, так и наступательного характера, предопределенного на определенный срок с целью получения преимущества перед вражескими силами.

5. **Проектирование.**

Экспертные системы, выполняющие проектирование, разрабатывают различные формы объектов, учитывая сложившиеся обстоятельства и все сопутствующие факторы.

В качестве примера можно рассмотреть генную инженерию.

6. **Контроль.**

Экспертные системы, осуществляющие контроль, сравнивают настоящее поведение системы с ее ожидаемым поведением. Наблю-

дающие экспертные системы обнаруживают контролируемое поведение, которое подтверждает их ожидания по сравнению с нормальным поведением или их предположением о потенциальных отклонениях. Контролирующие экспертные системы по своей сути должны работать в режиме реального времени и реализовывать зависящую как от времени, так и от контекста интерпретацию поведения контролируемого объекта.

В качестве примера можно привести слежение за показаниями измерительных приборов в атомных реакторах с целью обнаружения аварийных ситуаций или оценку данных диагностики пациентов, находящихся в блоке интенсивного лечения.

7. **Управление.**

Ведь широко известно, что экспертные системы, осуществляющие управление, весьма результативно руководят поведением системы в целом. Примером служит управление различными производствами, а также распределением компьютерных систем. Управляющие экспертные системы должны включать в себя наблюдающие компоненты, для того, чтобы контролировать поведение объекта на протяжении длительного времени, но они могут нуждаться и в других компонентах из уже проанализированных типов задач.

Экспертные системы применяются в самых различных областях: финансовых операциях, нефтяной и газовой промышленности. Технология экспертных систем может быть применена также в энергетике, транспортном хозяйстве, фармацевтическом производстве, космических разработках, металлургической и горной промышленностях, химии и многих других областях.

2. Структура экспертных систем

Разработка экспертных систем имеет ряд существенных отличий от разработки обычного программного продукта. Опыт создания экспертных систем показал, что использование при их разработке методологии, принятой в традиционном программировании, либо сильно увеличивает количество времени, затраченного на создание экспертных систем, либо вовсе приводит к отрицательному результату.

Экспертные системы в общем случае подразделяются на **статические** и **динамические**.

Для начала рассмотрим статическую экспертную систему.

Стандартная **статическая экспертная система** состоит из следующих основных компонентов:

1) рабочей памяти, называемой также базой данных;

2) базы знаний;

3) решателя, называемого также интерпретатором;

4) компонентов приобретения знаний;

5) объяснительного компонента;

6) диалогового компонента.

Рассмотрим теперь каждый компонент более подробно.

Рабочая память (по абсолютной аналогии с рабочей, т. е. оперативной памятью компьютера) предназначена для получения и хранения исходных и промежуточных данных решаемой в текущий момент задачи.

База знаний предназначена для хранения долгосрочных данных, описывающих конкретную предметную область, и правил, описывающих рациональное преобразование данных этой области решаемой задачи.

Решатель, называемый также **интерпретатором**, функционирует следующим образом: используя исходные данные из рабочей памяти и долгосрочные данные из базы знаний, он формирует правила, применение которых к исходным данным приводит к решению задачи. Одним словом, он действительно «решает» поставленную перед ним задачу;

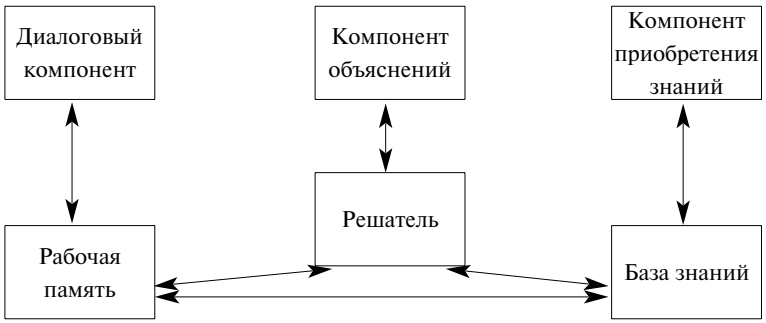
Компонент приобретения знаний автоматизирует процесс заполнения экспертной системы знаниями эксперта, т. е. именно этот компонент обеспечивает базу знаний всей необходимой информацией из данной конкретной предметной области.

Компонент объяснений разъясняет, как система получила решение данной задачи, или почему она это решение не получила и какие знания она при этом использовала. Иначе говоря, компонент объяснений создает отчет о проделанной работе.

Данный компонент является очень важным во всей экспертной системе, поскольку он значительно облегчает тестирование системы экспертом, а также повышает доверие пользователя к полученному результату и, следовательно, ускоряет процесс разработок.

Диалоговый компонент служит для обеспечения дружественного интерфейса пользователя как в ходе решения задачи, так и в процессе приобретения знаний и объявления результатов работы.

Теперь, когда мы знаем, из каких компонент в общем состоит статистическая экспертная система, построим диаграмму, отражающую структуру такой экспертной системы. Она имеет следующий вид:



Статические экспертные системы чаще всего используются в технических приложениях, где можно не учитывать изменения окружающего среды, происходящие во время решения задачи. Любопытно знать, что первые экспертные системы, получившие практическое применение, были именно статическими.

Итак, на этом закончим пока рассмотрение статистической экспертной системы, перейдем к анализу экспертной системы динамической.

К сожалению, в программу нашего курса не входит подробное рассмотрение этой экспертной системы, поэтому ограничимся разбором только самых основных отличий динамической экспертной системы от статических.

В отличие от статической экспертной системы в структуру **динамической экспертной системы** дополнительно вводятся два следующих компонента:

1) подсистема моделирования внешнего мира;

2) подсистема связей с внешним окружением.

Подсистема связей с внешним окружением как раз и осуществляет связи с внешним миром. Делает она это посредством системы специальных датчиков и контроллеров.

Помимо этого, некоторые традиционные компоненты статической экспертной системы подвергаются существенным изменениям, для того чтобы отобразить временную логику событий, происходящих в данный момент в окружающей среде.

Это главное различие между статической и динамической экспертными системами.

Пример динамической экспертной системы — управление производством различных медикаментов в фармацевтической промышленности.

3. Участники разработки экспертных систем

В разработке экспертных систем участвуют представители различных специальностей. Чаще всего конкретную экспертную систему разрабатывают трое специалистов. Это, как правило:

1) эксперт;

2) инженер по знаниям;

3) программист по разработке инструментальных средств.

Разъясним обязанности каждого из приведенных здесь специалистов.

Эксперт — это специалист в той предметной области, задачи которой и будут решаться при помощи этой конкретной разрабатываемой экспертной системы.

Инженер по знаниям — это специалист по разработке непосредственно экспертной системы. Используемые им технологии и методы называются технологиями и методами инженерии знаний. Инженер по знаниям помогает эксперту выявить из всей информации предметной области ту информацию, которая необходима для работы с конкретной разрабатываемой экспертной системой, а затем структурировать ее.

Любопытно, что отсутствие среди участников разработки инженеров по знаниям, т. е. замена их программистами, либо приводит к неудаче всего проекта создания конкретной экспертной системы, либо значительно увеличивает сроки ее разработки.

И, наконец, **программист** разрабатывает инструментальные средства (если инструментальные средства разрабатываются заново), предназначенные для ускорения разработки экспертных систем. Эти инструментальные средства содержат в пределе все основные компоненты экспертной системы; также программист осуществляет сопряжение своих инструментальных средств с той средой, в которой она будет использоваться.

4. Режимы работы экспертных систем

Экспертная система работает в двух основных режимах:

1) в режиме приобретения знаний;

2) в режиме решения задачи (называемом также режимом консультаций, или режимом использования экспертной системы).

Это логично и понятно, ведь сначала необходимо как бы загрузить экспертную систему информацией из той предметной области, в которой ей предстоит работать, это и есть режим «обучения» экспертной

системы, режим, когда она получает знания. А уже после загрузки всей необходимой для работы информации следует и сама работа. Экспертная система становится готовой для эксплуатации, и ее теперь можно использовать для консультаций или для решения каких-либо задач.

Рассмотрим более подробно **режим приобретения знаний**.

В режиме приобретения знаний работу с экспертной системой осуществляет эксперт при посредничестве инженера по знаниям. В этом режиме эксперт, используя компонент приобретения знаний, наполняет систему знаниями (данными), которые, в свою очередь, позволяют системе в режиме решения уже без участия эксперта решать задачи из данной предметной области.

Следует отметить, что режиму приобретения знаний в традиционном подходе к разработке программ соответствуют этапы алгоритмизации, программирования и отладки, выполняемые непосредственно программистом. Отсюда следует, что в отличие от традиционного подхода в случае экспертных систем разработку программ осуществляет не программист, а эксперт, естественно, с помощью экспертных систем, т. е. по большому счету человек, не владеющий программированием.

А теперь рассмотрим второй режим функционирования экспертной системы, т. е. **режим решения задач**.

В режиме решения задачи (или так называемом режиме консультации) общение с экспертными системами осуществляет непосредственно конечный пользователь, которого интересует концевой итог работы и иногда способ его получения. Необходимо отметить, что в зависимости от назначения экспертной системы пользователь не обязательно должен быть специалистом в данной проблемной области. В этом случае он обращается к экспертным системам за результатом, не имея достаточных знаний для получения результатов. Или все же пользователь может обладать уровнем знаний, достаточным для достижения необходимого результата самостоятельно. В этом случае пользователь может сам получить результат, но обращается к экспертным системам с целью либо ускорить процесс получения результата, либо возложить на экспертные системы монотонную работу. В режиме консультации данные о задаче пользователя после обработки их диалоговым компонентом поступают в рабочую память. Решатель на основе входных данных из рабочей памяти, общих данных о проблемной области и правил из базы данных формирует решение задачи. Экспертные системы при решении задачи не только исполняют предписанную последовательность конкретной операции, но и предварительно формирует ее. Это делается для случая, если реакция системы не совсем понятна пользователю. В этой ситуации пользователь

может потребовать объяснения о том, почему данная экспертная система задает тот или иной вопрос или почему данная экспертная система не может выполнить данную операцию, как получен тот или иной результат, поставляемый данной экспертной системой.

5. Продукционная модель знаний

По своей сути **продукционные модели знаний** близки к логическим моделям, что позволяет организовать весьма эффективные процедуры логического вывода данных. Это с одной стороны. Однако, с другой стороны, если рассматривать продукционные модели знаний в сравнении с логическими моделями, то первые более наглядно отображают знания, что является неоспоримым преимуществом. Поэтому, несомненно, продукционная модель знаний является одним из главных средств представления знаний в системах искусственного интеллекта.

Итак, начнем подробное рассмотрение понятия продукционной модели знаний.

Традиционная продукционная модель знаний включает в себя следующие базовые компоненты:

1) набор правил (или продукций), представляющих базу знаний продукционной системы;

2) рабочую память, в которой хранятся исходные факты, а также факты, выведенные из исходных фактов при помощи механизма логического вывода;

3) сам механизм логического вывода, позволяющий из имеющихся фактов, согласно имеющимся правилам вывода, выводить новые факты.

Причем, что любопытно, количество таких операций может быть бесконечно.

Каждое правило, представляющее базу знаний продукционной системы, содержит условную и заключительную части. В условной части правила находится либо одиночный факт, либо несколько фактов, соединенных конъюнкцией. В заключительной части правила находятся факты, которыми необходимо пополнить рабочую память, если условная часть правила является истинной.

Если попытаться схематично изобразить продукционную модель знаний, то под продукцией понимается выражение следующего вида:

(i) Q; P; A $\rightarrow$ B; N;

Здесь i — это имя продукционной модели знаний или ее порядковый номер, с помощью которого данная продукция выделяется из

всего множества продукционных моделей, получая некую идентификацию. В качестве имени может выступать некоторая лексическая единица, отражающая суть данной продукции. Фактически мы именуем продукцию для лучшего восприятия сознанием, чтобы упростить поиск нужной продукции из списка.

Приведем простой пример: покупка тетради» или «набор цветных карандашей. Очевидно, что каждую продукцию обычно именуют словами, подходящими для данного момента. Проще говоря, называют вещи своими именами.

Идем дальше. Элемент Q характеризует сферу применения данной конкретной продукционной модели знаний. Такие сферы легко выделяются в сознании человека, поэтому с определением данного элемента, как правило, сложностей не возникает. Приведем пример.

Рассмотрим следующую ситуацию: допустим, в одной сфере нашего сознания хранятся знания о том, как надо готовить пищу, в другой, как добраться до работы, в третьей, как правильно эксплуатировать стиральную машину. Подобное разделение присутствует и памяти продукционной модели знаний. Это разделение знаний на отдельные сферы позволяет значительно экономить время, затрачиваемое на поиск нужных в данный момент каких-то конкретных продукционных моделей знаний, и тем самым значительно упрощает процесс работы с ними.

Разумеется, что основным элементом продукции является ее так называемое ядро, которое в нашей приведенной выше формуле обозначалось как $A \rightarrow B$. Эта формула может быть интерпретирована, как «если выполняется условие A, то следует выполнить действие B».

Если же мы имеем дело с более сложными конструкциями ядра, то в правой части допускается следующий альтернативный выбор «если выполняется условие A, то следует выполнить действие B_1, иначе следует выполнить действие B_2».

Однако интерпретация ядра продукционной модели знаний может быть различной и зависеть от того, что будет стоять слева и справа от знака секвенции «$\rightarrow$ ». При одной из интерпретаций ядра продукционной модели знаний секвенция может истолковываться в обычном логическом смысле, т. е. в качестве знака логического следования действия B из истинного условия A.

Тем не менее возможны и другие интерпретации ядра продукционной модели знаний. Так, например, A может описывать какое-то условие, выполнение которого необходимо для того, чтобы можно было совершить некое действие B.

Далее рассмотрим элемент продукционной модели знаний P.

Элемент **P** определяется, как условие применимости ядра продукции. Если условие P истинно, то ядро продукции активизируется. В противном случае, если условие P не выполняется, т. е. оно ложно, ядро не может быть активизировано.

В качестве наглядного примера рассмотрим следующую продукционную модель знаний:

«Наличие денег»; «Если хочешь купить вещь A, то следует заплатить в кассу ее стоимость и предъявить чек продавцу».

Смотрим, если условие P истинно, т. е. покупка оплачена и чек предъявлен, то ядро активизируется. Покупка совершена. В случае если в этой продукционной модели знаний условие применимости ядра ложно, т. е. если нет денег, то применить ядро продукционной модели знаний невозможно, и оно не активизируется.

И переходим, наконец, к элементу **N**.

Элемент N называется постусловием продукционной модели данных. Постусловие задает действия и процедуры, которые необходимо выполнить после реализации ядра продукции.

Для лучшего восприятия приведем простой пример: после покупки вещи в магазине необходимо в описи товаров этого магазина уменьшить на единицу количество вещей такого типа, т. е. если покупка совершена (следовательно, реализовано ядро), то в магазине стало на одну единицу данного конкретного товара меньше. Отсюда постусловие «Вычеркнуть единицу купленного товара».

Подводя итог, мы можем сказать, что представление знаний в виде набора правил, т. е. посредством использования продукционной модели знаний, имеет следующие преимущества:

1) это простота создания и понимания отдельных правил;

2) это простота механизма логического выбора.

Однако в представлении знаний в виде набора правил имеются и недостатки, которые все же ограничивают сферу и частоту применения продукционных моделей знаний. Основным таким недостатком считается неясность взаимных отношений между составляющими конкретную продукционную модель знаний правилами, а также правилами логического выбора.

Содержание

БАЗЫ ДАННЫХ
КОНСПЕКТ ЛЕКЦИЙ

Завредакцией БиК: *Д. С. Потапов*
Корректор: *Г. А. Серикова*
Технический редактор: *Т. И. Федорова, И. С. Семенова*

Формат: 84 × 108/32
Гарнитура: «Ньютон»