ГЛАВА 9

# Целостность данных



## 9.1. ВВЕДЕНИЕ

С годами та часть реляционной модели, которая касается целостности данных, подвергается наиболее существенным изменениям (возможно, здесь следовало сказать не “подвергается изменениям”, а развивается). На первых порах основное внимание было обращено на исследование именно первичных и внешних ключей (сокращенно будем называть их просто *ключами*). Но со временем специалисты по базам данных стали понимать важность (а фактически чрезвычайную важность) ограничений целостности как таковых, и поэтому количество работ в этой области значительно возросло; наряду с этим многие исследователи обнаружили определенные сложности, возникающие при использовании только одних ключей. Структура этой главы отражает данное изменение направления исследований, поскольку в ней вначале дано общее описание ограничений целостности в достаточно большом объеме, а затем изложение переходит к рассмотрению ключей (которые по-прежнему имеют важное практическое значение).

Прежде всего отметим, что, неформально выражаясь, **ограничение целостности** — это логическое выражение, связанное с некоторой базой данных, результатом вычисления которого всегда должно быть значение TRUE. Подобное ограничение может рассматриваться как формальное выражение некоторого *бизнес-правила* [9.15], хотя иногда сами бизнес-правила (которые в данной главе рассматриваются как представленные всегда на естественном языке) иногда также называют *ограничениями целостности*. Но так или иначе, ниже приведено несколько примеров бизнес-правил, которые основаны на материале базы данных поставщиков и деталей.

1. Значение статуса каждого поставщика должно находиться в пределах от 1 до 100 включительно.
2. Каждый поставщик из Лондона имеет статус 20.
3. Если вообще имеются какие-либо детали, то по меньшей мере одна из них должна иметь синий цвет.
4. Разные поставщики не могут иметь одинаковые номера поставщиков.
5. Каждая поставка выполняется существующим поставщиком.
6. Ни один поставщик со статусом меньше 20 не поставляет любые детали в количестве больше 500.

Данные примеры будут широко использоваться в настоящей главе.

Очевидно, что ограничения должны быть формально объявлены для СУБД, после чего СУБД должна предписывать их выполнение. Объявление ограничений сводится просто к использованию соответствующих средств языка базы данных, а соблюдение ограничений осуществляется с помощью контроля со стороны СУБД над операциями обновления, которые могут нарушить эти ограничения, и запрещения тех операций, которые их действительно нарушают. Ниже приведено формальное объявление ограничения, показанного в первом из рассматриваемых примеров, на языке Tutorial D.

```
CONSTRAINT SC1
    IS_EMPTY ( S WHERE STATUS < 1 OR STATUS > 100 ) ;
```

Для соблюдения этого ограничения СУБД должна контролировать все операции, в которых осуществляется попытка вставить новую запись с данными о поставщике или изменить статус существующего поставщика [9.5].

Безусловно, при первоначальном объявлении ограничения система должна проверить, удовлетворяет ли ему в настоящий момент база данных. Если это условие не соблюдается, ограничение должно быть отвергнуто; в противном случае оно принимается (т.е. записывается в каталог системы) и начиная с этого момента соблюдается. Кстати, следует отметить, что в данном примере ограничению присвоено имя — SC1 (“suppliers constraint one” — ограничение для поставщиков номер один). При условии, что это ограничение принято СУБД, оно будет зарегистрировано в каталоге под этим именем, после чего указанное имя будет появляться в диагностических сообщениях, вырабатываемых системой в ответ на попытки нарушить данное ограничение.

Ниже показаны еще две возможные формулировки ограничения из примера 1, в которых теперь используется версия языка Tutorial D на основе реляционного исчисления (здесь SX — переменная области значений, которая принимает свои значения среди номеров поставщиков).

```
CONSTRAINT SC1
    NOT EXISTS SX ( SX.STATUS < 1 OR SX.STATUS > 100 ) ;

CONSTRAINT SC1
    FORALL SX ( SX.STATUS ≥ 1 AND SX.STATUS ≤ 100 ) ;
```

Безусловно, все эти три формулировки являются эквивалентными. Но в данной главе в качестве основы для значительной части изложения используется реляционное исчисление, а не реляционная алгебра, по причинам, которые станут очевидными для читателя по мере дальнейшего изложения материала. Предлагаем читателю в качестве упражнения подготовить алгебраические версии приведенных примеров, основанных на исчислении.

Безусловно, необходимо также предусмотреть способ уничтожения существующих ограничений, если они больше не требуются. Для этого применяется следующий оператор.

```
DROP CONSTRAINT <constraint name> ;
```

## 9.2. ПОДРОБНЫЕ СВЕДЕНИЯ ОБ ОГРАНИЧЕНИЯХ ЦЕЛОСТНОСТИ

В общем, ограничения целостности представляют собой ограничения, налагаемые на значения, которые разрешено принимать некоторой переменной, или комбинации переменных[1]. Поэтому тот факт, что конкретная переменная относится к некоторому определенному типу, представляет собой априорное ограничение, налагаемое на рассматриваемую переменную (это ограничение состоит в том, что значения, которые может принимать данная переменная, должны, безусловно, быть значениями этого типа). Например, переменная отношения S (поставщики) ограничивается тем, что должна содержать значения, являющиеся отношениями, в которых каждое значение S# представляет собой номер поставщика (значение типа S#), каждое значение SNAME является именем (значением типа NAME) и т.д.

Но эти простые априорные ограничения, безусловно, не остаются единственными возможными; и действительно, в этом смысле ни один из шести примеров, приведенных в разделе 9.1, не представлял собой априорное ограничение. Еще раз рассмотрим пример 1.

1. *Значение статуса каждого поставщика должно находиться в пределах от 1 до 100 включительно.*

   Ниже приведена немного более точная формулировка того же ограничения.

*Если* `S` *— поставщик, то* `S` *имеет значение статуса в пределах от 1 до 100 включительно.*

   А здесь показан еще более точный (или более формальный) способ формулировки этого ограничения[2].

---

[1] Как следует из этого замечания, ограничения целостности применяются (по крайней мере, в принципе) к переменным всех типов. Но по очевидным причинам основное внимание в данной книге уделено именно переменным отношения.

[2] Обратите внимание на то, что в этих формальных примерах не используется синтаксис языка Tutorial D (версии этих примеров на языке Tutorial D приведены ниже). Кроме того, этот синтаксис не полностью соответствует тому, который был определен для реляционного исчисления в главе 8, хотя и близок к нему (особенно к версии, касающейся исчисления доменов).

```
FORALL s# ∈ S#, sn ∈ NAME, st ∈ INTEGER, sc ∈ CHAR
  ( IF { S# s#, SNAME sn, STATUS st, CITY sc } ∈ S
    THEN st ≥ 1 AND st ≤ 100 )
```

Это формальное выражение можно прочитать следующим образом (на довольно ломаном естественном языке).

*Для всех номеров поставщиков `S#`, всех имен `SN`, всех целых чисел `ST` и всех символьных строк `SC`, если в переменной отношения поставщиков появляется кортеж с атрибутами `S# S#`, `SNAME SN`, `STATUS ST` и `CITY SC`, то значение `ST` больше или равно 1 и меньше или равно 100.*

Возможно, теперь читатель согласится с тем, что действительно нужно было привести эту альтернативную версию примера 1 на естественном языке, которая представлена выше. Дело в том, что теперь становится очевидным следующий факт — эта альтернативная версия, соответствующее формальное выражение и его аналог на ломаном естественном языке, имеют некоторую общую "форму" (как таковую), которая выглядит примерно следующим образом.

*Если (`IF`) в некоторой переменной отношения присутствует некоторый кортеж, то (`THEN`) этот кортеж удовлетворяет некоторому условию.*

Эта "форма" является примером **логической импликации** (которую иногда называют *материальной импликацией*), или просто импликации. Эта конструкция уже встречалась в главе 8; она имеет следующую общую форму.

```
IF p THEN q
```

Здесь `p` и `q` — логические выражения, соответственно, называемые **посылкой** и **следствием**. Общее выражение (т.е. импликация) является ложным, если `p` — истинно, а `q` — ложно, в противном случае оно является истинным; иными словами, само выражение `IF p THEN q` является логическим и оно логически эквивалентно выражению `(NOT p) OR q`.

Кстати, следует отметить, что показанная выше форма по умолчанию включает необходимый квантор `FORALL`, поскольку выражение "Если (`IF`) ... присутствует некоторый кортеж" по сути означает "Для всех (`FORALL`) присутствующих кортежей, ...".

Теперь перейдем к аналогичному анализу примеров 2–6 (но при этом не используя формулировок на естественном языке).

***Примечание.*** Приведенные ниже формулировки не являются единственно возможными, а также не всегда бывают самыми простыми из всех возможных, но они, по крайней мере, правильны. Следует также отметить, что в каждом примере иллюстрируется по меньшей мере одна новая мысль.

2. *Каждый поставщик из Лондона имеет статус 20.*

```
FORALL s# ∈ S#, sn ∈ NAME, st ∈ INTEGER, sc ∈ CHAR
  ( IF { S# s#, SNAME sn, STATUS st, CITY sc } ∈ S
    THEN ( IF sc = 'London'
           THEN st = 20 ) )
```

В этом примере следствие импликации само является импликацией.

3. *Если вообще имеются какие-либо детали, то по меньшей мере одна из них должна иметь синий цвет.*

```
IF
EXISTS p# ∈ P#, pn ∈ NAME, pl ∈ COLOR, pw ∈ WEIGHT, pc ∈ CHAR
  ( { P# p#, PNAME pn, COLOR pl, WEIGHT pw, CITY pc } ∈ P )
THEN
EXISTS p# ∈ P#, pn ∈ NAME, pl ∈ COLOR, pw ∈ WEIGHT, pc ∈ CHAR
  ( { P# p#, PNAME pn, COLOR pl, WEIGHT pw, CITY pc } ∈ P
    AND pl = COLOR ('Blue') )
```

Обратите внимание на то, что нельзя просто применить формулировку "по меньшей мере одна деталь имеет синий цвет", поскольку необходимо учесть тот случай, когда вообще нет деталей.

***Примечание.*** Хотя это может показаться не совсем очевидным, но данный пример действительно соответствует той же общей форме, как и два предыдущих. Ниже приведена альтернативная формулировка, которая позволяет более наглядно подчеркнуть эту мысль.

```
FORALL p# ∈ P#, pn ∈ NAME, pl ∈ COLOR, pw ∈ WEIGHT, pc ∈ CHAR
  ( IF { P# p#, PNAME pn, COLOR pl, WEIGHT pw, CITY pc } ∈ P
    THEN EXISTS q# ∈ P#, qn ∈ NAME, ql ∈ COLOR,
                         qw ∈ WEIGHT, qc ∈ CHAR
           ( { P# q#, PNAME qn, COLOR ql,
                      WEIGHT qw, CITY qc } ∈ P
           AND ql = COLOR ('Blue') ) )
```

4. *Разные поставщики не могут иметь одинаковые номера поставщиков.*

```
FORALL x# ∈ S#, xn ∈ NAME, xt ∈ INTEGER, xc ∈ CHAR,
       y# ∈ S#, yn ∈ NAME, yt ∈ INTEGER, yc ∈ CHAR
  ( IF { S# x#, SNAME xn, STATUS xt, CITY xc } ∈ S AND
       { S# y#, SNAME yn, STATUS yt, CITY yc } ∈ S
    THEN ( IF x# = y# THEN xn = yn AND xt = yt AND xc = yc ) )
```

Данное выражение представляет собой просто формальное изложение того факта, что {S#} является потенциальным ключом (или, во всяком случае, суперключом) для поставщиков; таким образом, ограничения ключа представляют собой лишь частный случай ограничений как таковых. Синтаксическая конструкция KEY {S#} языка Tutorial D может рассматриваться как сокращение для приведенного выше более сложного выражения. (Кстати, здесь заслуживают внимания фигурные скобки. Они подчеркивают, что ключи всегда представляют собой множества атрибутов, даже если рассматриваемое множество содержит только один атрибут, и поэтому атрибуты ключей в этой книге всегда показаны в фигурных скобках, по меньшей мере, в формальных контекстах.)

***Примечание.*** И потенциальные ключи, и суперключи подробно рассматриваются в разделе 9.10.

Кстати, отметим, что этот пример имеет следующую общую форму.

*Если* (IF) *некоторые кортежи появляются в некоторой переменной отношения, то* (THEN) *эти кортежи удовлетворяют некоторому условию.*

Сравните между собой примеры 2 и 3, которые оба принимают такую же форму, как и пример 1 (вскоре станет очевидно, что это относится и к примеру 5). В отличие от этого, пример 6 принимает следующую общую форму.

```
Если (IF) некоторые кортежи появляются в некоторых переменных
отношения, то (THEN) эти кортежи удовлетворяют некоторому условию.
```

Эта последняя форма характеризуется тем, что она в общем относится ко всем ограничениям целостности (первые две можно рассматривать как частные случаи этого с*а*мого общего случая).

5. *Каждая поставка выполняется существующим поставщиком.*

```
FORALL s# ∈ S#, p# ∈ P#, q ∈ QTY
  ( IF { S# s#, P# p#, QTY q } ∈ SP
    THEN EXISTS sn ∈ NAME, st ∈ INTEGER, sc ∈ CHAR
           ( { S# s#, SNAME sn, STATUS st, CITY sc } ∈ S ) )
```

Это выражение представляет собой формальное утверждение того факта, что {S#} является внешним ключом для поставок, который соответствует потенциальному ключу {S#} для поставщиков. Поэтому ограничения внешнего ключа также являются лишь частным случаем ограничений как таковых (дополнительная информация по этой теме также приведена в разделе 9.10). Следует отметить, что в этом примере участвуют две отдельные переменные отношения, SP и S, а во всех примерах 1–4 участвует только одна переменная отношения[3].

6. *Ни один поставщик со статусом меньше 20 не поставляет любые детали в количестве больше 500.*

```
FORALL s# ∈ S#, sn ∈ NAME, st ∈ INTEGER, sc ∈ CHAR,
       p# ∈ P#, q ∈ QTY
  ( IF { S# s#, SNAME sn, STATUS st, CITY sc } ∈ S AND
       { S# s#, P# p#, QTY q } ∈ SP
    THEN st ≥ 20 OR q ≤ QTY ( 500 ) )
```

В данном примере также рассматриваются две разные переменные отношения, но это ограничение не является ограничением внешнего ключа.

## Примеры на языке Tutorial D

В завершение данного раздела рассмотрим версии примеров 2–6 на языке Tutorial D (на основе исчисления). При этом применяются обычные соглашения, касающиеся имен переменных области значений.

---

[3] В предыдущем издании данной книги для обозначения ограничения, в котором участвует одна и только одна переменная отношения, использовался термин “ограничение переменной отношения”, а для ограничения, в котором участвовало больше переменных отношений, — термин “ограничение базы данных”. Но, как будет показано в разделе 9.9, важность такого разграничения в большей степени относится к сфере практики, чем логики, поэтому в дальнейшем изложении эта тема фактически не рассматривается.

1. *Каждый поставщик из Лондона имеет статус 20.*

```
CONSTRAINT SC2
    FORALL SX ( IF SX.CITY = 'London'
                THEN SX.STATUS = 20 END IF ) ;
```

   Обратите внимание на то, что в языке Tutorial D логические импликации (выражения IF/THEN) включают обозначение конца выражения "END IF".

2. *Если вообще имеются какие-либо детали, то по меньшей мере одна из них должна иметь синий цвет.*

```
CONSTRAINT PC3
    IF EXISTS PX ( TRUE )
    THEN EXISTS PX ( PX.COLOR = COLOR ('Blue') ) END IF ;
```

3. *Разные поставщики не могут иметь одинаковые номера поставщиков.*

```
CONSTRAINT SC4
    FORALL SX FORALL SY ( IF SX.S#      = SY.S#
                          THEN SX.SNAME = SY.SNAME
                          AND SX.STATUS = SY.STATUS
                          AND SX.CITY   = SY.CITY
                          END IF ) ;
```

4. *Каждая поставка выполняется существующим поставщиком.*

```
CONSTRAINT SSP5
    FORALL SPX EXISTS SX ( SX.S# = SPX.S# ) ;
```

5. *Ни один поставщик со статусом меньше 20 не поставляет любые детали в количестве больше 500.*

```
CONSTRAINT SSP6
    FORALL SX FORALL SPX
         ( IF SX.S# = SPX.S#
           THEN SX.STATUS ≥ 20 OR SPX.QTY ≤ 500 END IF ) ;
```

## 9.3. ПРЕДИКАТЫ И ВЫСКАЗЫВАНИЯ

Еще раз рассмотрим формальную версию примера 1 ("Значение статуса каждого поставщика должно находиться в пределах от 1 до 100 включительно").

```
FORALL s# ∈ S#, sn ∈ NAME, st ∈ INTEGER, sc ∈ CHAR
  ( IF { S# s#, SNAME sn, STATUS st, CITY sc } ∈ S
    THEN st ≥ 1 AND st ≤ 100 )
```

Эта формальная версия представляет собой логическое выражение. Но следует отметить, что в нем[4] участвует переменная, а именно переменная отношения поставщиков S. Поэтому мы не можем определить, каковым является значение этого выражения (т.е. не можем определить, какое логическое значение оно принимает), до тех пор, пока не подставим конкретное значение вместо этой переменной (вообще говоря, в действительности

[4] В этом выражении участвуют также несколько переменных области значений, но, как было показано в главе 8, переменные области значений не являются переменными в том смысле, какой им придается в языке программирования, а в данной главе термин "переменная" рассматривается именно в том смысле, который ему придается в языке программирования (если явно не указано иное).

различные подстановки должны приводить к получению разных истинностных значений). Иными словами, данное выражение является **предикатом**, а переменная S служит **формальным параметром** этого предиката. Поэтому, когда возникает необходимость "конкретизировать" данный предикат (иначе говоря, когда нам требуется проверить ограничение), мы предоставляем в качестве **фактического параметра** такое отношение, которое является текущим значением переменной отношения S (а переменная отношения S — единственный формальный параметр), и только после этого появляется возможность вычислить это выражение.

Теперь, после конкретизации предиката (которая, по сути, сводится к замене его единственного формального параметра некоторым фактическим параметром), мы приходим к некоторому выражению с истинностным значением, которое вообще не включает переменных, а содержит лишь значения. Аналогичные замечания касаются и ограничений, которые распространяются на два, три, четыре или на любое другое количество переменных отношения; так или иначе, когда возникает необходимость вычислить выражение (т.е. когда требуется проверить ограничение), достаточно заменить каждый формальный параметр отношением, которое является текущим значением соответствующей переменной отношения, после чего мы приходим к выражению с истинностным значением, которое вообще не имеет переменных или, иными словами, является высказыванием. Вне всякого сомнения, любое *высказывание* может быть либо истинным, либо ложным (его можно рассматривать как *вырожденный* предикат, т.е. предикат, для которого множество формальных параметров является пустым, как было описано в предыдущей главе). Ниже приведено несколько простых примеров высказываний.

- Солнце — это звезда.
- Луна — это звезда.
- Солнце находится от Земли дальше, чем Луна.
- Джордж Буш победил на президентских выборах в США в 2000 году.

Выяснение того, какие из этих высказываний являются истинными, а какие ложными, оставляем в качестве упражнения для читателя. Но следует отметить, что не все высказывания являются истинными; широко распространенная ошибка состоит в том, что все высказывания рассматриваются исключительно как истинные.

На основании материала, приведенного в этом разделе, можно сделать следующие выводы: ограничение, определенное формально, становится предикатом, но при проверке этого ограничения вместо формальных параметров предиката подставляются фактические параметры, в результате чего предикат сводится к высказыванию, и к такому высказыванию затем предъявляется требование, чтобы его значение было равно TRUE.

## 9.4. ПРЕДИКАТЫ ПЕРЕМЕННОЙ ОТНОШЕНИЯ И ПРЕДИКАТЫ БАЗЫ ДАННЫХ

Безусловно, что в общем любая конкретная переменная отношения может становиться объектом действия многих ограничений. Предположим, что R — переменная отношения. В таком случае **предикатом переменной отношения** R является результат применения логической операции "И", или операции конъюнкции ко всем ограничениям, которые распространяются на переменную отношения R (иными словами, в которых она упоминается). Следует учитывать, что здесь возникает определенная опасность противоречивого

толкования: как уже было сказано, каждое отдельное ограничение само является предикатом в полном смысле этого понятия, но **предикат** переменной отношения для R — это конъюнкция *всех* отдельных предикатов, которые относятся к R. Например, если для упрощения предположим, что только шесть ограничений, описанных в разделе 9.1, относятся к базе данных поставщиков и деталей (не считая априорных ограничений), то предикат переменной отношения для поставщиков является конъюнкцией ограничений с номерами 1, 2, 4, 5 и 6, а предикат переменной отношения для поставок — конъюнкцией предикатов с номерами 5 и 6. Обратите внимание, что эти два предиката переменной отношения в некотором смысле "перекрываются", поскольку они имеют некоторые общие составляющие ограничения[5].

Теперь допустим, что R — переменная отношения, а RP — предикат переменной отношения для R. Итак, безусловно, ни в коем случае нельзя допускать, чтобы переменная R приобретала такое значение, что его подстановка в RP вместо R (а также любая другая необходимая подстановка фактических параметров вместо формальных параметров, которая должна быть выполнена в RP), становилась причиной того, чтобы предикат RP принимал значение FALSE. Итак, теперь мы можем ввести **золотое правило** (первую версию), которое имеет большое значение при анализе ограничений целостности.

> *Ни одна операция обновления не должна приводить к присваиванию любой переменной отношения такого значения, которое вызывает то, что предикат этой переменной отношения получает значение* FALSE.

Теперь предположим, что D является базой данных[6] и что D включает переменные отношения R1, R2, ..., Rn (и только эти переменные отношения). Допустим, что предикатами переменной отношения для этих переменных отношения, соответственно, являются RP1, RP2, ..., RPn. Тогда **предикат базы данных** для D, скажем, DP, представляет собой конъюнкцию всех таких предикатов переменной отношения.

```
DP ≡ RP1 AND RP2 AND ... AND RPn
```

Ниже приведена расширенная (более общая и фактически окончательная) версия золотого правила.

> *Ни одна операция обновления не должна приводить к присваиванию любой базе данных такого значения, которое вызывает то, что предикат этой базы данных получает значение* FALSE.

---

[5] В предыдущем издании данной книги было приведено определение, что предикат переменной отношения для переменной отношения R представляет собой конъюнкцию всех ограничений переменной отношения, которые относятся к R (где, как указано в разделе 9.2, ограничением переменной отношения называется ограничение, в котором упоминается только одна переменная отношения). Но теперь, в соответствии с [3.3], мы принимаем определение, что предикат переменной отношения R является конъюнкцией всех ограничений, а не только ограничений переменной отношения, которые относятся к R. Автор приносит свои извинения всем, кто считает такое изменение в терминологии причиной дополнительной путаницы.

[6] D, безусловно, также является переменной (см. аннотацию к [3.3]) и поэтому служит объектом действия ограничений целостности.

Безусловно, предикат базы данных принимает значение FALSE тогда и только тогда, когда по меньшей мере один из ее составляющих предикатов переменной отношения принимает это значение. А сам предикат переменной отношения принимает значение FALSE тогда и только тогда, когда по меньшей мере одна из составляющих ограничений принимает это значение.

***Примечание.*** Как уже было сказано, два разных предиката переменной отношения, RPi и RPj (i ≠ j), могут иметь некоторые общие составляющие ограничения. Из этого следует, что одно и то же ограничение может появиться в предикате базы данных DP несколько раз. С логической точки зрения такое положение дел не влечет за собой каких-либо затруднений, поскольку если c — ограничение, то выражение c AND c логически эквивалентно просто c. Итак, хотя, безусловно, в такой ситуации желательно, чтобы система вычисляла ограничение c один, а не два раза, эта проблема относится к реализации, а не к модели.

## 9.5. ПРОВЕРКА ОГРАНИЧЕНИЙ

В данном разделе рассматриваются две темы. Одна из них относится к реализации, а другая — к модели, и обе эти темы касаются вопроса о том, как фактически должны проверяться объявленные ограничения. Вначале рассмотрим проблему реализации. Еще раз вернемся к примеру 1, в котором, как известно, фактически утверждается, что если некоторый кортеж присутствует в переменной отношения S, то этот кортеж должен удовлетворять определенному условию (в данном случае условию "статус должен находиться в пределах от 1 до 100"). В частности, следует отметить, что в этом ограничении речь идет о кортежах в переменной отношения. Поэтому очевидно, что при осуществлении попытки вставить новый кортеж с данными о поставщике со статусом (скажем) 200, должна происходить описанная ниже последовательность событий.

1. Вставка нового кортежа.
2. Проверка ограничения.
3. Отмена обновления (поскольку проверка окончилась неудачей).

Но это же нелепо! Безусловно, что необходимо обнаружить эту ошибку еще до того, как будет выполнена вставка. Поэтому реализация, очевидно, должна отвечать такому требованию, что в ней формальное выражение ограничения должно использоваться для формирования подходящей процедуры (процедур) проверки, которая должна применяться к кортежам, предлагаемым для вставки, еще до того, как фактически будет выполнена эта вставка.

В принципе, данный процесс формирования процедуры проверки является довольно простым. Предположим, что предикат базы данных включает ограничение в следующей форме.

```
IF { S# s#, SNAME sn, STATUS st, CITY sc } ∈ S THEN ...
```

Это означает, что если посылка в некоторой импликации в составе всего предиката находится в форме, соответствующей утверждению "Некоторый кортеж присутствует в S", то следствие в этой импликации, по сути, представляет собой ограничение на кортежи, предназначенные для вставки в переменную отношения S.

***Примечание.*** Кстати, следует отметить, что если база данных разработана в соответствии с принципом ортогонального проектирования (см. главу 13), и при условии, что СУБД имеет информацию обо всех применимых ограничениях, то любой конкретный кортеж приходится проверять на соответствие не больше чем одному предикату переменной отношения, поскольку он будет представлять собой приемлемого кандидата для вставки с помощью операции `INSERT`, самое большее, в одну переменную отношения.

Теперь перейдем к проблеме модели (которая, безусловно, является более фундаментальной). Снова рассмотрим приведенное выше золотое правило.

> *Ни одна операция обновления не должна приводить к присваиванию любой базе данных такого значения, которое вызывает то, что предикат этой базы данных получает значение* `FALSE`.

Хотя эта мысль в разделе 9.4 явно не подчеркивалась, необходимо учитывать, что из этого правила следует требование о немедленном выполнении проверок всех ограничений. С чем это связано? Дело в том, что это правило сформулировано в терминах операций обновления, а не в терминах транзакций (см. следующий абзац). Поэтому, по сути, **золотое правило** требует соблюдения ограничений целостности на этапе перехода от выполнения одного оператора к выполнению другого[7] и в нем не подразумевается понятие *отложенной* проверки ограничений целостности или проверки во время выполнения операции фиксации `COMMIT`.

Итак, читатель мог уже обнаружить, что выраженная здесь позиция, согласно которой все проверки должны выполняться немедленно, является весьма нетрадиционной; в литературе (включая предыдущие издания этой книги) чаще всего утверждается или просто предполагается, что *единицей контроля целостности* является транзакция и что, по меньшей мере, часть проверок следует откладывать до конца транзакции (т.е. ко времени выполнения операции `COMMIT`). Но существуют серьезные причины, по которым транзакции являются неприменимыми в качестве *единицы контроля целостности*, и такой единицей вместо них должны служить операторы. К сожалению, эти причины невозможно доходчиво объяснить, не дав вначале более подробного описания всего, что вообще связано с понятием транзакции. Поэтому отложим подробное обсуждение этой темы до главы 16; до этой главы мы просто предполагаем, что требование немедленной проверки является логически оправданным, не пытаясь дополнительно обосновать нашу позицию. (Но один важный довод в пользу позиции автора можно найти в аннотации к [9.16] в конце данной главы.)

## 9.6. СОПОСТАВЛЕНИЕ ВНУТРЕННИХ И ВНЕШНИХ ПРЕДИКАТОВ

Как было показано выше, каждая переменная отношения имеет предикат переменной отношения, а вся база данных имеет предикат базы данных. Безусловно, все рассматриваемые предикаты являются такими, о которых “в системе имеется информация”.

---

[7] По сути, в этом изложении требуется немного более высокая точность, но для внесения таких уточнений будет необходимо в определенной степени коснуться темы конкретного языка базы данных, используемого в рассматриваемых операциях. Но для описания данной темы достаточно отметить, что ограничения целостности должны удовлетворяться к концу выполнения любого оператора, который не содержит в себе синтаксически вложенных других операторов. Или, неформально выражаясь, “Ограничения должны удовлетворяться на этапах обработки базой данных точек с запятыми, отделяющих друг от друга операторы”.

Это означает, что подобные предикаты определены формально (т.е. они фактически входят в состав определения базы данных), кроме того, их соблюдение контролируется системой. По этим причинам иногда удобно называть рассматриваемые предикаты **внутренними** предикатами, поскольку переменные отношения и базы данных в принципе имеют также **внешние** предикаты, описание которых приведено ниже[8].

Первое и наиболее важное замечание состоит в том, что внутренние предикаты — это формальные конструкции, тогда как внешние предикаты — просто неформальные конструкции. Внутренние предикаты (это — не строгая формулировка) являются описанием того, что означают данные для системы; в отличие от этого, внешние предикаты описывают, что означают данные для пользователя. Безусловно, пользователи должны знать не только о внешних предикатах, но и о внутренних, однако еще раз повторим, что система должна иметь информацию о внутренних предикатах (и действительно быть способной учитывать лишь внутренние предикаты). Можно фактически неформально утверждать, что любой внутренний предикат представляет собой аппроксимацию в системе соответствующего внешнего предиката.

В дальнейшем изложении, если не указано иное, речь будет идти исключительно о переменных отношения. Итак, как уже было сказано, внешним предикатом данной конкретной переменной отношения по сути является то, что*о* эта переменная отношения означает для пользователя. Например, в случае переменной отношения поставщиков `S` внешний предикат может формулироваться таким образом, как показано ниже.

> *Поставщик с указанным номером поставщика (`S#`) работает по контракту, имеет указанное имя (`SNAME`) и указанный статус (`STATUS`), а также находится в указанном городе (`CITY`). Кроме того, значение его статуса должно находиться в пределах от 1 до 100 включительно и должно быть равно 20, если городом является Лондон. К тому же никакие два различных поставщика не имеют один и тот же номер поставщика.*

Но для удобства дальнейшего обсуждения заменим этот предикат более простым, приведенным ниже.

> *Поставщик `S#` работает по контракту, имеет имя `SNAME`, имеет статус `STATUS` и находится в городе `CITY`.*

(Вообще говоря, все внешние предикаты являются лишь неформальными, поэтому мы имеем право формулировать их сколь угодно просто или сложно, безусловно, в разумных пределах.)

Теперь отметим, что приведенное выше утверждение действительно является предикатом, поскольку оно имеет четыре формальных параметра (`S#`, `SNAME`, `STATUS` и `CITY`), соответствующие четырем атрибутам переменной отношения[9], и после подстановки вместо

---

[8] Внешние предикаты уже рассматривались в главах 3 и 6, но тогда они именовались просто предикатами. До сих пор фактически по умолчанию термин “предикат” использовался во всей данной книге как обозначение именно внешнего предиката. Единственным важным исключением было обсуждение операции ограничения, приведенное в главе 7, где условие ограничения было названо предикатом; так оно и есть, но это — не внешний предикат.

[9] В этом изложении термин “формальный параметр” используется немного в ином смысле, чем в разделах 9.3 и 9.4. В этих разделах формальные параметры (и соответствующие фактические параметры) обозначали целые отношения, а в данном изложении они обозначают отдельные атрибуты.

этих формальных параметров фактических параметров соответствующих типов образуется высказывание (т.е. некий логический объект, безусловно принимающий либо истинное, либо ложное значение). Поэтому каждый кортеж, присутствующий в переменной отношения S в любой указанный момент времени, может рассматриваться как обозначающий такое высказывание, полученное путем конкретизации данного предиката. Кроме того (что очень важно!), в данный момент времени рассматриваются как истинные лишь данные конкретные высказывания (т.е. те высказывания, которые теперь представлены кортежами в переменной отношения S). Например, рассмотрим следующий кортеж.

```
{ S# S#('S1'), SNAME NAME('Smith'), STATUS 20, CITY 'London' }
```

Если этот кортеж присутствует в переменной отношения S в некоторый момент времени, то следует считать *истинным фактом*, что в данный момент существует поставщик, работающий по контракту, который имеет номер поставщика S1, имя Smith, статус 20 и находится в Лондоне. Вообще говоря, справедлива следующая формула:

```
IF ( s S ) = TRUE THEN XPS ( s ) = TRUE
```

В этой формуле применяются описанные ниже обозначения.

- s — это кортеж в следующей форме:

```
{ S# s#, SNAME sn, STATUS st, CITY sc }
```

  Здесь s# — значение типа S#, sn — значение типа NAME, st — значение типа INTEGER и sc — значение типа CITY.

- XPS — внешний предикат для поставщиков.
- XPS(s) — это высказывание, полученное путем конкретизации предиката XPS значениями формальных параметров S# = s#, SNAME = sn, STATUS = st и CITY = sc.

Но как было отмечено в главе 6, применительно к внешним предикатам можно принять более широкие допущения, чем ко внутренним. Говоря точнее, по отношению к ним принимается предположение о замкнутости мира, в котором утверждается, что если некоторый кортеж, допустимый по всех прочим признакам, не присутствует в переменной отношения в некоторый момент времени, то на основании принятого соглашения соответствующее высказывание в данный момент рассматривается как ложное. Например, предположим, что в некоторый конкретный момент времени в переменной отношения S не присутствует следующий кортеж.

```
{ S# S#('S6'), SNAME NAME('Lopez'), STATUS 30, CITY 'Madrid' }
```

В этом случае данный факт следует рассматривать таким образом, что в данный момент не существует работающего по контракту поставщика с номером поставщика S6, который имеет имя Lopez, статус 30 и находится в Мадриде. Вообще говоря, справедлива следующая формула:

```
IF ( s ∈ S ) = FALSE THEN XPS ( s ) = FALSE
```

В более краткой форме она выглядит таким образом:

```
IF NOT ( s ∈ S ) THEN NOT XPS ( s )
```

На основании изложенного выше можно вывести следующую формулу:

```
s ∈ S ≡ XPS ( s )
```

Иными словами, любой конкретный кортеж присутствует в конкретной переменной отношения в конкретный момент времени тогда и только тогда, когда в данный момент времени присутствие данного кортежа приводит к тому, что внешний предикат этой переменной отношения принимает значение TRUE. Из этого следует, что данная переменная отношения содержит те и только те кортежи, которые соответствуют истинным конкретизациям внешнего предиката данной переменной отношения в рассматриваемый момент времени.

## 9.7. СРАВНЕНИЕ ПОНЯТИЙ ПРАВИЛЬНОСТИ И НЕПРОТИВОРЕЧИВОСТИ

По определению, внешние предикаты и высказывания, полученные путем конкретизации таких предикатов, не известны (и фактически не могут быть известными) в системе. Например, система не может иметь информации о том, что некий "поставщик" где-то "находится", или о том, что означает утверждение, будто "поставщик" имеет "некоторый статус" (и т.д.). Все эти вопросы относятся к области интерпретации фактических данных, поскольку данные имеют смысл только для пользователя, но не для системы. Рассмотрим более конкретный пример. Допустим, что в одном и том же кортеже оказались данные о номере поставщика S1 и названии города Лондона. Тогда пользователь может рассматривать этот факт так, как будто он означает, что поставщик S1 находится в Лондоне[10], но (повторяем) нет никакого способа, с помощью которого можно было бы вынудить систему прийти к аналогичным выводам.

Более того, даже если бы системе можно было сообщить, что подразумевается под утверждением, будто *поставщик где-то находится*, все равно система не могла бы априори иметь информацию о том, являются ли истинными данные, сообщенные пользователем! Когда пользователь вносит в систему сведения о том, что поставщик S1 находится в Лондоне (обычно путем выполнения оператора INSERT), система не может применить какой-либо способ, чтобы определить, действительно ли эти сведения являются истинными. Все, что может сделать система, — проверить, не приведет ли ввод этой информации к нарушению каких-либо ограничений (т.е. не вызовет ли это такую конкретизацию любого внутреннего предиката, при которой будет получено значение FALSE). При условии, что этого не происходит, система должна принять эти сведения и с этих пор рассматривать их как истинные (по меньшей мере, пока пользователь не сообщит системе, что данные сведения больше не являются истинными, обычно путем выполнения оператора DELETE).

Кстати, следует отметить, что изложенное выше наглядно показывает, почему предположение о замкнутости мира не относится к внутренним предикатам. А именно, кортеж может удовлетворять внутреннему предикату для данной переменной отношения и

---

[10] Этот факт может также означать, что поставщик S1 в данный момент проживает в Лондоне, или поставщик S1 имеет офис в Лондоне, или поставщик S1 не имеет офиса в Лондоне, а также может иметь бесконечное количество других возможных толкований (соответствующих бесконечному количеству возможных внешних предикатов).

вместе с тем не присутствовать в этой переменной отношения на законных основаниях, поскольку он не соответствует истинному высказыванию в реальном мире.

Подведем итог данному обсуждению, высказав такое неформальное утверждение, что внешний предикат для определенной переменной отношения представляет собой **намеченную интерпретацию** для этой переменной отношения. Как таковая, намеченная интерпретация имеет смысл только для пользователя, но не для системы. Поэтому можно привести еще одно неформальное утверждение о том, что внешний предикат для указанной переменной отношения является **критерием приемлемости обновлений** для рассматриваемой переменной отношения; это означает, что внешний предикат определяет (по меньшей мере, в принципе), можно ли разрешить успешно выполнить затребованные операции `INSERT`, `DELETE` или `UPDATE` на этой переменной отношения. Поэтому в идеальной ситуации система должна иметь информацию о внешнем предикате для каждой переменной отношения, чтобы обладать способностью успешно действовать при всех возможных попытках обновить эту переменную отношения. Но, как было показано выше, эта цель является недостижимой; система не может иметь информацию о внешнем предикате ни для одной переменной отношения. Но система имеет информацию о достаточно качественной аппроксимации этого внешнего предиката, поскольку ей известен соответствующий внутренний предикат, и она следит за его соблюдением. Поэтому прагматическим "критерием приемлемости обновлений" служит внутренний предикат, а не внешний (а внешний может быть таковым лишь в идеальной ситуации). Еще одна, более строгая формулировка этого утверждения приведена ниже.

> *Система может контролировать только непротиворечивость, но не истинность хранимых в ней данных.*

Это означает, что система не может гарантировать наличие в базе данных только истинных высказываний; все, что она может сделать, — это гарантировать отсутствие каких-либо данных, вызывающих нарушение ограничений целостности (т.е. гарантировать то, что она не содержит каких-либо данных, не совместимых с этими ограничениями). Но, к сожалению, истинность и непротиворечивость — не одно и то же! Фактически можно отметить следующее:

- если база данных содержит только истинные высказывания, то она непротиворечива, но из этого не следует обратное утверждение;
- если база данных не является непротиворечивой, то она содержит по меньшей мере одно ложное высказывание, но из этого не следует обратное утверждение.

Приведенные выше формулировки можно более кратко изложить следующим образом: из того, что данные являются правильными, следует, что они непротиворечивы (но не обратное), а из того, что данные не являются непротиворечивыми, следует, что они неправильны (но не обратное). Здесь под словом "правильные" подразумевается, что в базе данных содержатся правильные данные тогда и только тогда, когда она полностью отражает истинное состояние дел в реальном мире.

## 9.8. ОГРАНИЧЕНИЯ ЦЕЛОСТНОСТИ И ПРЕДСТАВЛЕНИЯ

Важно отметить, что почти все рассуждения, приведенные выше в этой главе, касались в общем всех, а не только базовых переменных отношения. В частности, они касаются и представлений (которые являются виртуальными переменными отношения).

Поэтому представления также служат объектом действия ограничений и имеют предикаты переменной отношения (как внутренние, так и внешние). Например, предположим, что определено представление путем применения к переменной отношения поставщиков операции проекции по атрибутам S#, SNAME и STATUS (что фактически приводит к удалению атрибута CITY). В таком случае внешний предикат для этого представления определен примерно так, как показано ниже.

*Существует некоторый город CITY, такой что работающий по контракту поставщик S# имеет имя SNAME, статус STATUS и находится в городе CITY.*

Следует отметить, что сам этот предикат, как и требуется, имеет три формальных параметра, соответствующих трем атрибутам представления, а не четыре (атрибут CITY больше не является параметром, а выполняет функции *связанной переменной* в силу того, что к нему применен квантор существования в виде утверждения "существует некоторый город"). Еще один способ выразить ту же мысль (возможно, более наглядный) состоит в следующем. Можно отметить, что рассматриваемый предикат логически эквивалентен приведенному ниже.

*Поставщик S# работает по контракту, имеет имя SNAME, имеет статус STATUS и находится в некотором городе.*

Вполне очевидно, что данная версия предиката имеет только три формальных параметра. А что можно в этом случае утверждать в отношении внутреннего предиката? Еще раз рассмотрим шесть примеров, применяемых в данной главе.

1. Значение статуса каждого поставщика должно находиться в пределах от 1 до 100 включительно.
2. Каждый поставщик из Лондона имеет статус 20.
3. Если вообще имеются какие-либо детали, то по меньшей мере одна из них должна быть синего цвета.
4. Разные поставщики не могут иметь одинаковые номера поставщиков.
5. Каждая поставка выполняется существующим поставщиком.
6. Ни один поставщик со статусом меньше 20 не поставляет любые детали в количестве больше 500.

Предположим, что рассматриваемое представление (проекция переменной отношения поставщиков по атрибутам S#, SNAME и STATUS) называется SST. В таком случае, если речь идет о представлении SST, то пример 3, безусловно, к нему не относится, поскольку в нем рассматриваются детали, а не поставщики. А что касается других примеров, то каждый из них в определенной степени связан с представлением SST, но в несколько модифицированной форме. В частности, ниже показана модифицированная форма примера 5.

```
FORALL s# ∈ S#, p# ∈ P#, q ∈ QTY
  ( IF { S# s#, P# p#, QTY q } ∈ SP
    THEN EXISTS sn ∈ NAME, st ∈ INTEGER
           ( { S# s#, SNAME sn, STATUS st } ∈ SST ) )
```

Изменения наблюдаются в третьей и четвертой строках. В них все упоминания об атрибуте CITY удалены, а ссылка на S заменена ссылкой на SST. Обратите внимание на то, что это ограничение для SST может рассматриваться как производное от соответствующего ограничения для S точно так же, как сама переменная отношения SST происходит от переменной отношения S (и в конечном итоге сам внешний предикат для SST является производным от внешнего предиката для S)[11].

Аналогичные замечания относятся непосредственно к примерам 1, 2, 4 и 6. Но, как показано ниже, пример 2 немного сложнее, поскольку он требует введения конструкции EXISTS, соответствующей атрибуту, который был удален в результате выполнения операции проекции.

```
FORALL s# ∈ S#, sn ∈ NAME, st ∈ INTEGER
  ( IF { S# s#, SNAME sn, STATUS st } ∈ SST
    THEN EXISTS sc ∈ CHAR
      ( { S# s#, SNAME sn, STATUS st, CITY sc } ∈ S AND
        ( IF sc = 'London'
          THEN st = 20 ) )
```

Однако и в данном случае это ограничение может рассматриваться как производное от соответствующего ограничения для S.

## 9.9. СХЕМА КЛАССИФИКАЦИИ ОГРАНИЧЕНИЙ

В данном разделе будет кратко намечена схема классификации для ограничений (по сути, это та же схема, которая была принята в [3.3]). Кратко отметим, что здесь предусмотрено распределение ограничений по четырем основным категориям: ограничения базы данных, ограничения переменной отношения, ограничения атрибута и ограничения типа. Краткие определения этих ограничений приведены ниже.

- *Ограничением базы данных* называется ограничение на значения, которые разрешено принимать указанной базе данных.
- *Ограничением переменной отношения* называется ограничение на значения, которые разрешено принимать указанной переменной отношения.
- *Ограничением атрибута* называется ограничение на значения, которые разрешено принимать указанному атрибуту.
- *Ограничение типа* представляет собой не что иное, как определение множества значений, из которых состоит данный тип.

Но задачу описания этих ограничений проще всего выполнить, рассматривая их в обратном порядке.

### Ограничения типа

До настоящего времени в этой главе ограничения типа вообще не упоминались. Но эти ограничения достаточно подробно рассматривались в главе 5, поэтому приведенное ниже описание предназначено лишь для того, чтобы напомнить читателю, о чем было

[11] Обратите также внимание на то, что это производное ограничение фактически является ограничением внешнего ключа от базовой переменной отношения к представлению! (См. раздел 9.10.)

сказано в той главе. Прежде всего, еще раз отметим, что *ограничение типа* является не чем иным, как спецификацией значений, из которых состоит рассматриваемый тип. Ниже приведен один пример (повторение примера из главы 5).

```
TYPE WEIGHT POSSREP { D DECIMAL (5,1)
                        CONSTRAINT D > 0.0 AND D < 5000.0 } ;
```

Он имеет следующий смысл: допустимыми значениями типа `WEIGHT` являются такие и только такие значения, которые можно представить десятичными числами с точностью пять цифр и с одной цифрой после десятичной точки, где рассматриваемое десятичное число больше нуля и меньше 5000.

Теперь должно быть очевидно, что единственный способ, благодаря которому любое выражение может принять значение типа `WEIGHT`, состоит в использовании некоторого вызова селектора `WEIGHT`. Поэтому единственная возможность нарушения в любом таком выражении ограничения типа `WEIGHT` состоит в том, что это ограничение будет нарушено в вызове селектора. Из этого следует, что ограничения типа всегда могут рассматриваться (по меньшей мере, концептуально) как проверяемые во время некоторого вызова селектора. Например, рассмотрим следующий вызов селектора для типа`WEIGHT`.

```
WEIGHT ( 7500.0 )
```

Это выражение активизирует исключение во время прогона программы (которое словесно выражается как “нарушение ограничения типа `WEIGHT` — значение выходит за допустимые пределы”).

На основании сказанного можно утверждать, что ограничения типа всегда проверяются немедленно, а значит, в частности, считать, что ни одна переменная отношения ни при каких условиях не получит значения для любого атрибута любого кортежа, которое не имело бы соответствующего типа (безусловно, речь идет о системе, которая поддерживает ограничения типа должным образом!).

Поскольку ограничения типа фактически представляют собой просто спецификацию значений, из которых состоит рассматриваемый тип, в языке Tutorial D предусмотрена возможность увязывать такие ограничения с определением соответствующего типа и обозначать их с помощью соответствующего имени типа. Из этого следует, что ограничение типа может быть удалено только путем удаления самого определения типа.

## Ограничения атрибута

*Ограничения атрибута*, по сути, состоят в том, что в разделе 9.2 было названо *априорными ограничениями*; иными словами, ограничением атрибута фактически является просто объявление, которое гласит, что указанный атрибут указанной переменной отношения имеет указанный тип. Например, снова рассмотрим определение переменной отношения поставщиков, которое приведено ниже.

```
VAR S BASE RELATION
    { S#     S#,
      SNAME  NAME,
      STATUS INTEGER,
      CITY   CHAR } ... ;
```

В этой переменной отношения на значения атрибутов `S#`, `SNAME`, `STATUS` и `CITY` наложено ограничение, согласно которому они должны, соответственно, иметь типы `S#`,

NAME, INTEGER и CHAR. Иными словами, ограничения атрибута входят в состав определения рассматриваемого атрибута и сами могут быть обозначены с помощью соответствующего имени атрибута. Из этого следует, что ограничение атрибута может быть уничтожено только путем уничтожения самого атрибута (а это на практике обычно означает уничтожение содержащей этот атрибут переменной отношения).

***Примечание.*** В принципе, любая попытка ввести в базу данных (с помощью операции INSERT или UPDATE) значение атрибута, не имеющее допустимого типа, должна быть просто отвергнута. Но на практике подобная ситуация не должна даже возникать, если система действительно контролирует соблюдение ограничений типа, как описано в предыдущем подразделе.

## Ограничения переменной отношения и базы данных

До сих пор в настоящей главе в основном рассматривались только ограничения переменной отношения и базы данных; различие между ними состоит в том, что ограничение переменной отношения распространяется на одну и только одну переменную отношения, а ограничение базы данных распространяется на две или больше переменных отношения. Но как было указано в разделе 9.2, с теоретической точки зрения это различие не является слишком важным (хотя и может иметь смысл стремление его учитывать с точки зрения практики).

Но мы еще до сих пор не касались одной темы, которая заключается в том, что некоторые ограничения переменной отношения или базы данных могут представлять собой ограничения перехода. **Ограничением перехода** называется ограничение, регламентирующее допустимые переходы для определенной переменной (в частности, для конкретной переменной отношения или базы данных), которые она может выполнять, переходя от одного значения к другому[12]; например, семейное положение любого лица может измениться со значения “никогда не состоял в браке” на значение “состоит в браке”, но возврат к прежнему состоянию невозможен. При условии, что предусмотрен способ включить в одно выражение одновременно и то значение, которое рассматриваемая переменная имела до любой произвольной операции обновления, и значение той же переменной после того же обновления, то существует возможность сформулировать любое желаемое ограничение перехода. Ниже приведен один пример формулировки подобного ограничения (“статус поставщика ни в коем случае не должен уменьшаться”).

```
CONSTRAINT TRC1 FORALL SX'
    FORALL SX ( SX'.S# ≠ SX.S# OR SX'.STATUS ≤ SX.STATUS ) ;
```

***Пояснение.*** Примем такое соглашение, что имя переменной области значений со штрихом, такое как SX' в данном примере, обозначает ссылку на соответствующую переменную отношения, которой она была до выполнения рассматриваемой операции обновления. Таким образом, ограничение, показанное в этом примере, можно описать следующим образом: “Если SX' — кортеж поставщика перед выполнением обновления, то не должен существовать кортеж поставщика SX после обновления с таким же номером поставщика, как SX', и со значением статуса меньше чем SX'”.

---

[12] Ограничения, не относящиеся к типу ограничений перехода, иногда называют ограничениями состояния.

Следует отметить, что приведенное выше ограничение TRC1 является ограничением перехода переменной отношения (оно касается только одной переменной отношения — S). Ниже для сравнения приведено ограничение перехода базы данных ("Общее количество деталей любого конкретного типа, имеющихся у всех поставщиков, ни в коем случае не должно уменьшаться").

```
CONSTRAINT TRC2
    FORALL PX
        SUM ( SPX' WHERE SPX'.P# = PX.P#, QTY ) ≤
        SUM ( SPX WHERE SPX .P# = PX.P#, QTY ) ;
```

Понятие ограничений перехода не распространяется на ограничения типа или атрибута.

## 9.10. КЛЮЧИ

Как было отмечено в разделе 9.1, в реляционной модели всегда придавалось большое значение понятию ключей, хотя, как было показано выше, ключи фактически представляют собой лишь частный случай более общего феномена (пусть даже и важный с точки зрения практики). В этом разделе рассматриваются именно ключи.

### Потенциальные ключи

Допустим, что R — переменная отношения. По определению, множество всех атрибутов R обладает свойством уникальности; это означает, что в любой конкретный момент времени никакие два кортежа в значении R не являются дубликатами друг друга. На практике часто встречается ситуация, в которой определенное собственное подмножество множества всех атрибутов R также обладает свойством уникальности, например, в случае переменной отношения поставщиков S таким свойством обладает подмножество, содержащее только атрибут S#. На основании этих соображений может быть сформулировано приведенное ниже неформальное определение, продиктованное интуицией[13].

- Допустим, что K — множество атрибутов переменной отношения R. В таком случае K является потенциальным ключом для R тогда и только тогда, когда оно обладает одновременно двумя перечисленными ниже свойствами.
  - **а)** ***Уникальность.*** Ни одно допустимое значение R никогда не содержит два разных кортежа с одним и тем же значением K.
  - **б)** ***Несократимость.*** Никакое строгое подмножество K не обладает свойством уникальности.

Каждая переменная отношения имеет по меньшей мере один потенциальный ключ. Свойство уникальности подобных ключей не требует пояснения. А что касается свойства несократимости, то его смысл состоит в том, что если бы был определен *потенциальный ключ*, не являющийся несократимым, то система не имела бы информации об истинном состоянии дел и поэтому не могла бы следить за выполнением соответствующего ограничения целостности должным образом. Например, предположим, что в качестве потенциального

[13] Следует отметить, что это определение касается именно переменных отношения; аналогичное понятие может быть также определено для значений отношения [3.3], но переменные отношения — наиболее важный случай. Заслуживает также внимания то, что и в этом случае мы исходим из понятия равенства кортежей (точнее, это понятие применяется в определении свойства уникальности).

ключа для поставщиков определена комбинация атрибутов {S#,CITY} вместо одного только атрибута {S#}. В таком случае система не могла бы следить за соблюдением ограничения, согласно которому номера поставщиков являются "глобально" уникальными, а обладала бы способностью контролировать только более слабое ограничение, состоящее в том, что номера поставщиков являются уникальными "локально", в пределах одного города. Кроме всего прочего, по этой причине в приведенном выше определении требуется, чтобы потенциальные ключи не содержали никаких атрибутов, которые не обеспечивают решения задачи уникальной идентификации[14].

Следует отметить, что понятие несократимости, определение которого приведено выше, в литературе (включая первые издания этой книги) часто обозначается термином *минимальность*. Но в действительности термин *минимальность* является не вполне подходящим, поскольку из утверждения, что потенциальный ключ K1 является *минимальным*, не следует, что не может быть найден другой потенциальный ключ, K2, с меньшим количеством компонентов; вполне возможно, что, например, K1 имеет четыре компонента, а K2 — только два. Поэтому автор придерживается термина *несократимость*.

В языке Tutorial D для определения потенциального ключа рассматриваемой переменной отношения в объявлении переменной отношения используется следующий синтаксис.

```
KEY { <attribute name commalist> }
```

Ниже приведено еще несколько примеров.

```
VAR S BASE RELATION
  { S#     S#,
    SNAME  NAME,
    STATUS INTEGER,
    CITY   CHAR }
  KEY { S# } ;
```

***Примечание.*** В предыдущих главах это определение было приведено с конструкцией PRIMARY KEY, а не с неуточненной конструкцией KEY. Дополнительные описания и пояснения даны в подразделе "Первичные и альтернативные ключи" данного раздела.

```
VAR SP BASE RELATION
     { S#  S#,
       P#  P#,
       QTY QTY }
KEY { S#, P# } ... ;
```

В данном примере показана переменная отношения с составным потенциальным ключом (т.е. ключом, состоящим из двух или большего количества атрибутов). *Простым потенциальным ключом* называется ключ, который не является составным.

```
VAR ELEMENT BASE RELATION { NAME    NAME,
                            SYMBOL  CHAR,
                            ATOMIC# INTEGER }
                      KEY { NAME }
                      KEY { SYMBOL }
                      KEY { ATOMIC# } ;
```

[14] Еще одна существенная причина, по которой потенциальные ключи должны быть несократимыми, состоит в следующем: любой внешний ключ, который ссылается на "сократимый" потенциальный ключ (если бы это было возможно), также был бы "сократимым", и поэтому содержащая его переменная отношения почти наверняка нарушала бы принципы дальнейшей нормализации, как будет описано в главе 12.

В этом примере показана переменная отношения с несколькими различными потенциальными ключами, причем все они являются простыми.

```
VAR MARRIAGE BASE RELATION { HUSBAND                              NAME,
                             WIFE                                 NAME,
                             DATE /* Дата регистрации брака */ DATE }
/* Предполагается, что во всех рассматриваемых браках отсутствуют такие */
/* ситуации, как многомужие, многоженство и повторная регистрация брака */
/* одних и тех же супругов после их развода ...                          */
                  KEY { HUSBAND, DATE }
                  KEY { DATE, WIFE }
                  KEY { WIFE, HUSBAND } ;
```

В данном примере показана переменная отношения с несколькими различными потенциальными ключами, причем все они являются составными. Обратите внимание также на то, что некоторые из этих ключей перекрываются.

Безусловно, как было указано в разделе 9.2, определение потенциального ключа по сути является просто сокращением для некоторого ограничения переменной отношения. Такое сокращение является полезным по многим причинам. Одна из них просто состоит в том, что потенциальные ключи имеют важное значение с точки зрения практики. В частности, в реляционной модели они предоставляют основной **механизм адресации на уровне кортежей**. Это означает, что единственный гарантируемый системой способ точного определения некоторого конкретного кортежа состоит в использовании определенного значения потенциального ключа. Например, гарантируется, что следующее выражение обеспечивает получение не больше одного кортежа (точнее, оно обеспечивает получение значения отношения, содержащего не больше одного кортежа).

```
S WHERE S# = S# ('S3')
```

В отличие от этого, приведенное ниже выражение в общем обеспечивает получение непредсказуемого количества кортежей (вернее, отношения, содержащего непредсказуемое количество кортежей).

```
S WHERE CITY = 'Paris'
```

Из этого следует, что *понятие потенциальных ключей является настолько же важным для успешной эксплуатации реляционных систем, как и понятие адресов оперативной памяти для успешной эксплуатации операционной системы самого компьютера*. Из этого следуют приведенные ниже выводы.

1. С позволения сказать, такие “переменные отношения”, которые не имеют ни одного потенциального ключа (т.е. “переменные отношения”, которые допускают наличие дубликатов кортежей), неизбежно проявляют время от времени странную и аномальную реакцию на выполняемые с ними операции.
2. Система, в которой не реализовано понятие потенциальных ключей, неизбежно проявляет время от времени такую реакцию, которую нельзя назвать “полностью соответствующей реляционной модели”, даже если представленные в ней переменные отношения действительно являются в полном смысле слова настоящими переменными отношения и не содержат дубликатов кортежей.

Реакции на выполняемые операции, которые были названы выше “странными и аномальными” и “не полностью соответствующими реляционной модели”, касаются таких

вопросов, как обновление представлений и оптимизация (эти темы рассматриваются, соответственно, в главах 10 и 18).

В завершение этого раздела приведем несколько замечаний.

- Потенциальные ключи должны иметь не только базовые переменные отношения! Это требование распространяется на все переменные отношения, включая, в частности, представления. Но что касается, например, представлений, вопрос о том, могут ли или должны быть объявлены такие ключи, отчасти зависит от того, предусмотрена ли в системе возможность использовать **ссылки на потенциальный ключ** [3.3].
- Надмножество потенциального ключа называется **суперключом** (например, суперключом для переменной отношения S является множество атрибутов {S#,CITY}). Суперключ обладает свойством уникальности, но не обязательно обладает свойством несократимости. Безусловно, потенциальный ключ — это частный случай суперключа.
- Если SK — суперключ для переменной отношения R, а A — атрибут R, то в R обязательно существует **функциональная зависимость** SK → A. В действительности, мы можем определить суперключ как подмножество SK атрибутов R, такое что функциональная зависимость SK → A имеет место для всех атрибутов A в R.

  ***Примечание.*** Важное понятие функциональной зависимости подробно рассматривается в главе 11.
- Наконец, следует отметить, что логическое понятие потенциального ключа не следует путать с физическим понятием *уникального индекса* (даже несмотря на то, что последний часто используется для реализации первого). Иными словами, нет никаких оснований требовать, чтобы на некотором потенциальном ключе был определен индекс (или, скажем, какой-либо иной специальный физический путь доступа). На практике, по-видимому, должен быть действительно предусмотрен некоторый путь доступа, но рассмотрение вопроса о том, существует ли он или нет, выходит за рамки реляционной модели как таковой.

### Первичные и альтернативные ключи

Как уже было сказано выше, в любой переменной отношения возможно наличие двух или большего количества потенциальных ключей. В таком случае в реляционной модели традиционно предъявлялось такое требование (по меньшей мере, к базовым переменным отношения), чтобы точно один из этих ключей был выбран в качестве **первичного** ключа; с тех пор остальные ключи назывались **альтернативными**. Например, в случае определения таблицы элементов Менделеева, ELEMENT, можно выбрать {SYMBOL} в качестве первичного ключа, после чего применять {NAME} и {ATOMIC#} как альтернативные ключи. При наличии даже одного потенциального ключа в реляционной модели по традиции предъявлялось требование, чтобы этот потенциальный ключ рассматривался как первичный ключ для данной базовой переменной отношения. Поэтому считалось, что каждая базовая переменная отношения всегда имеет первичный ключ.

Во многих случаях (возможно, даже в большинстве случаев) действительно может оказаться, что определение одного из потенциальных ключей в качестве первичного (если между ними есть выбор) может стать целесообразным, но, безусловно, такое решение не является бесспорным во всех без исключения случаях. Аргументированные доводы в

поддержку такой позиции приведены в [9.14], а в этом разделе достаточно привести лишь один из них, который состоит в том, что выбор первичного ключа не диктуется логикой, а вместо этого, по сути, является произвольным. (Процитируем слова Кодда [9.9]: "Обычно основанием [для выбора первичного ключа] является упрощение работы, но этот аспект выходит за рамки реляционной модели".) В примерах, приведенных в этой книге, первичный ключ иногда указан, а иногда — нет. Но в них всегда определен по меньшей мере один потенциальный ключ.

### Внешние ключи

Неформально выражаясь, *внешний ключ* представляет собой множество атрибутов некоторой переменной отношения R2, значения которых должны совпадать со значениями некоторого потенциального ключа некоторой переменной отношения R1. Например, рассмотрим множество атрибутов {S#} переменной отношения SP. Должно быть ясно, что заданное значение {S#} может присутствовать в переменной отношения SP, только если такое же значение присутствует и в переменной отношения S в качестве значения единственного потенциального ключа {S#} (поскольку поставка не может быть выполнена несуществующим поставщиком). Аналогичным образом, любое конкретное значение для множества атрибутов {P#} может присутствовать в переменной отношения SP, только если такое же значение присутствует в переменной отношения P в качестве значения единственного потенциального ключа {P#} (поскольку не может быть также выполнена поставка несуществующей детали). На основании этих примеров может быть сформулировано приведенное ниже определение[15].

- Допустим, что R2 — переменная отношения. В таком случае **внешним ключом** в R2 является множество атрибутов R2, скажем, FK, такое что выполняются следующие требования:
  - **а)** существует переменная отношения R1 (R1 и R2 не обязательно должны быть разными) с потенциальным ключом CK;
  - **б)** существует возможность переименования некоторого подмножества атрибутов FK, такое что FK преобразуется (скажем) в FK', а FK' и CK относятся к одному и тому же типу (кортежу);
  - **в)** в любое время каждое значение FK в текущем значении R2 приводит к получению значения для FK', которое идентично значению CK в некотором кортеже в текущем значении R1.

Из этого следуют приведенные далее выводы.

1. На практике необходимость действительно выполнять *переименование* возникает редко; это означает, что подмножество атрибутов FK, требующих переименования, обычно бывает пустым (пример такой ситуации, в которой это условие не соблюдается, показан в п. 7). Поэтому для упрощения в дальнейшем предполагается, что подмножества FK и FK' идентичны, если явно не указано иное.
2. Следует отметить, что каждое значение FK должно присутствовать в качестве значения CK, но обратное требование не предъявляется; это означает, что R1 может

[15] Обратите внимание на то, что это определение также базируется на понятии равенства кортежей.

содержать значение CK, которое в настоящее время не присутствует в R2 в качестве значения FK. Например, а случае поставщиков и деталей (примеры значений в этой базе данных приведены на рис. 3.8 на стр. 119) номер поставщика S5 присутствует в переменной отношения S, но не в переменной отношения SP, поскольку поставщик S5 в настоящее время не поставляет каких-либо деталей.

3. Внешний ключ FK является **простым** или **составным** в зависимости от того, является ли простым или составным потенциальный ключ CK.
4. Любое значение FK представляет собой **ссылку** на кортеж, содержащий соответствующее значение CK (он называется **кортежем, упомянутым в ссылке**). Ограничение, согласно которому значения FK должны соответствовать значениям CK, называется **ограничением ссылочной целостности**. Переменная отношения R2 называется **ссылающейся** переменной отношения, а переменная отношения R1 — переменной отношения, **указанной в ссылке**. Задача обеспечения того, чтобы база данных не включала каких-либо недопустимых значений внешнего ключа, называется задачей поддержки **ссылочной целостности** (см. п. 12).
5. Рассмотрим еще раз базу данных поставщиков и деталей. Ограничения ссылочной целостности в этой базе данных можно представить с помощью следующей **ссылочной диаграммы**.

   S ⟵ SP ⟶ P

   Каждая стрелка означает, что в той переменной отношения, из которой исходит стрелка, имеется внешний ключ, ссылающийся на некоторый потенциальный ключ, заданный в переменной отношения, на которую указывает стрелка.

   ***Примечание.*** Для наглядности иногда целесообразно обозначать каждую стрелку на ссылочной диаграмме именем (именами) атрибута (атрибутов), из которого состоит соответствующий внешний ключ[16], например, как показано ниже.

   ```
      S#      P#
   S ⟵ SP ⟶ P
   ```

   Но в данной книге подобные метки показаны только в тех случаях, если их отсутствие может привести к путанице или к неоднозначности.
6. Любая конкретная переменная отношения может быть одновременно и указанной в ссылке, и ссылающейся, как в случае переменной отношения R2, показанной ниже.

   R3 ⟶ R2 ⟶ R1

   Вообще говоря, допустим, что существуют переменные отношения Rn, R(n-1), ..., R2, R1, такие что имеется ограничение ссылочной целостности, связывающее Rn с R(n-1), ограничение ссылочной целостности, связывающее R(n-1) с R(n-2), ..., и ограничение ссылочной целостности, связывающее R2 с R1 следующим образом.

   Rn ⟶ R(n-1) ⟶ R(n-2) ⟶ ... ⟶ R2 ⟶ R1

[16] Иной способ (который, возможно, является предпочтительным) предусматривает присваивание имен внешним ключам, а затем использование этих имен для обозначения стрелок.

В таком случае цепочка стрелок, проходящая от Rn до R1, представляет собой **ссылочный путь** от Rn до R1.

7. Следует отметить, что переменные отношения R1 и R2, указанные в определении внешнего ключа, не обязательно должны быть разными. Это означает, что любая переменная отношения может иметь внешний ключ, значения которого должны совпадать со значениями некоторого потенциального ключа в той же переменной отношения. В качестве примера рассмотрим следующее определение переменной отношения (его синтаксис будет описан чуть позже, но в любом случае он должен быть достаточно очевидным).

```
VAR EMP BASE RELATION
        { EMP# EMP#, ..., MGR_EMP# EMP#, ... }
  KEY { EMP# }
  FOREIGN KEY { RENAME MGR_EMP# AS EMP# } REFERENCES EMP ;
```

Здесь атрибут MGR_EMP# обозначает табельный номер руководителя того служащего, который обозначен атрибутом EMP#; например, кортеж EMP для служащего E4 может включать значение MGR_EMP# E3, равное E3, которое представляет собой ссылку на кортеж EMP для служащего E3. (Как было обещано в п. 1, здесь показан пример, в котором требуется некоторое явное переименование атрибута.) Переменные отношения, подобные EMP, иногда называют *ссылающимися сами на себя* (или *самоссылающимися*).

***Упражнение.*** Подготовьте данные, которые могли бы использоваться в качестве примера значения переменной отношения EMP.

8. Самоссылающиеся переменные отношения фактически представляют частный случай более общей ситуации, а именно, они показывают, что могут существовать **ссылочные циклы**. Переменные отношения Rn, R(n-1), R(n-2), ..., R2, R1 образуют такой цикл, если Rn содержит внешний ключ, ссылающийся на R(n-1), а R(n-1) содержит внешний ключ, ссылающийся на R(n-2), ..., и т.д., наконец, R1 содержит внешний ключ, снова ссылающийся на Rn. Кратко можно сформулировать это определение таким образом, что ссылочный цикл существует, если имеется ссылочный путь от некоторой переменной отношения Rn к ней самой, как показано ниже.

```
Rn ⟶ R(n-1) ⟶ R(n-2) ⟶ ... ⟶ R2 ⟶ R1 ⟶ Rn
```

9. Иногда встречается такое небеспочвенное утверждение, что согласованность внешних и потенциальных ключей представляет собой своего рода “клей”, который скрепляет воедино всю базу данных. Еще один способ формулировки той же идеи состоит в том, что подобные согласования представляют собой определенные связи. Но необходимо учитывать важное замечание, что не все такие связи определены лишь ключами, которые применяются указанным способом. Например, может существовать некоторая связь между поставщиками и деталями (“нахождение в одном городе”), для представления которой применяются атрибуты CITY переменных отношения S и P; некоторый поставщик и некоторая деталь считаются *находящимися в одном городе*, если данный поставщик находится в том же городе, где хранится данная деталь. Но эта связь не определена с помощью ключей.

10. По традиции, понятие внешнего ключа было сформулировано только для базовых переменных отношения, и этот факт сам по себе вызывает некоторые вопросы (см. обсуждение принципа взаимозаменяемости в разделе 10.2 главы 10). Сам автор не налагает здесь указанное ограничение, но для упрощения сводит обсуждение только к базовым переменным отношения (причем эти указания во многом относятся непосредственно и к другим переменным отношения).
11. В определении реляционной модели первоначально предъявлялось требование, чтобы внешние ключи ссылались именно лишь на первичные ключи, а не просто на любые потенциальные ключи (например, еще раз рекомендуем ознакомиться с [9.9]). Автор в целом отвергает это ограничение как ненужное и вообще нежелательное, хотя на практике оно может часто служить хорошей рекомендацией [9.14].
12. Наряду с понятием внешнего ключа в реляционной модели определено приведенное ниже правило (правило ссылочной целостности).

    - **Ссылочная целостность.** База данных не должна содержать каких-либо несогласованных значений внешнего ключа[17].

    В этом определении термин "несогласованное значение внешнего ключа" обозначает значение внешнего ключа в некоторой ссылающейся переменной отношения, для которого не существует согласованного значения соответствующего потенциального ключа в соответствующей переменной отношения, указанной в ссылке. Иными словами, это ограничение можно сформулировать просто как следующее требование: "Если значение B ссылается на A, то A должно существовать".

Ниже показан синтаксис определения внешнего ключа.

```
FOREIGN KEY { <item commalist> } REFERENCES <relvar name>
```

Эта конструкция присутствует в определении ссылающейся переменной отношения; здесь *<relvar name>* обозначает переменную отношения, указанную в ссылке, а каждый элемент *<item>* представляет собой либо имя атрибута *<attribute name>* в ссылающейся переменной отношения, либо выражение в следующей форме.

```
RENAME <attribute name> AS <attribute name>
```

(Пример применения варианта с использованием ключевого слова RENAME приведен в определении самоссылающейся переменной отношения EMP в п. 7.) Примеры объявлений внешнего ключа были приведены выше во многих разделах данной книги (в частности, см. рис. 3.9 в главе 3). Безусловно, как было указано в разделе 9.2, определение внешнего ключа фактически представляет собой просто сокращение для некоторого ограничения базы данных (или для некоторого ограничения переменной отношения, в

---

[17] Обратите внимание на то, что это правило ссылочной целостности может рассматриваться как "метаограничение" (сверхограничение), поскольку в нем подразумевается, что любая отдельно взятая база данных должна быть объектом действия некоторых конкретных ограничений, свойственных рассматриваемой базе данных и совместно гарантирующих, чтобы в этой определенной базе данных не нарушалась ссылочная целостность. Кстати, следует отметить, что реляционная модель обычно рассматривается как включающая еще одно подобное "метаограничение" — правило целостности сущностей. Отложим обсуждение этого правила до главы 19.

случае самоссылающейся переменной отношения), за исключением той ситуации, когда определение внешнего ключа распространяется на некоторые "ссылочные действия"; тогда оно представляет собой нечто большее, чем просто ограничение ссылочной целостности как таковое. Эта тема рассматривается в следующем подразделе, "Ссылочные действия".

### Ссылочные действия

Рассмотрим следующий оператор на языке Tutorial D.

```
DELETE S WHERE S# = S# ('S1') ;
```

Предположим, что операция `DELETE` выполняет именно то, что в ней указано, т.е. удаляет кортеж поставщика `S1`, не больше и не меньше. Предположим также, что соблюдаются еще два условия: во-первых, база данных включает сведения о поставках, выполненных поставщиком `S1`, во-вторых, приложение не удаляет эти сведения о поставках. Но после того как система проверяет ограничение ссылочной целостности, которое связывает данные о поставке с данными о поставщиках, она обнаруживает нарушение и активизирует исключение.

Однако возможен альтернативный подход, который может оказаться предпочтительным в некоторых ситуациях. Он заключается в том, что система выполняет соответствующее компенсирующее действие, которое позволяет гарантировать, что общий результат всегда будет удовлетворять ограничению. В данном примере очевидно, что компенсирующее действие со стороны системы состоит в "автоматическом" удалении системой данных о поставках, выполненных поставщиком `S1`. Такого эффекта можно достичь, дополнив определение внешнего ключа, как показано ниже.

```
VAR SP BASE RELATION { ... } ...
    FOREIGN KEY { S# } REFERENCES S
                      ON DELETE CASCADE ;
```

Спецификация `ON DELETE CASCADE` определяет правило удаления для данного конкретного внешнего ключа, а спецификация `CASCADE` является *ссылочным действием* для этого правила удаления. Смысл данных спецификаций состоит в том, что выполнение операции `DELETE` на переменной отношения поставщиков "каскадно" приводит к удалению также соответствующих кортежей (если они имеются) в переменной отношения поставок.

Еще одним широко применяемым вариантом ссылочного действия является `RESTRICT` (оно не имеет ничего общего с операцией сокращения реляционной алгебры). В данном случае использование ключевого слова `RESTRICT` означает, что операции `DELETE` должны "ограничиваться" теми ситуациями, в которых отсутствуют согласованные поставки (в противном случае они будут отвергнуты). Если же в определении конкретного внешнего ключа ссылочное действие не указано, это эквивалентно применению ключевого слова `NO ACTION`, которое означает именно то, что в нем сказано: операция `DELETE` выполняется точно так же, как в ней указано, не больше и не меньше. (Если в рассматриваемой ситуации задано ключевое слово `NO ACTION`, а с удаляемыми данными о поставщике связаны согласованные данные о поставках, то в дальнейшем возникает нарушение ограничения ссылочной целостности, поэтому конечный результат

становится аналогичным такому, как при использовании ключевого слова RESTRICT.) Из этого следуют приведенные ниже выводы.

1. Операция DELETE не является единственной операцией, для которой имеет смысл определять ссылочные действия. Например, что произойдет при попытке обновить номер такого поставщика, для которого существует по меньшей мере одна согласованная поставка? Очевидно, что требуется не только правило удаления, но и правило обновления. В общем существуют такие же варианты ссылочных действий для операции UPDATE, как и для операции DELETE, которые описаны ниже.
   - CASCADE. Действие операции UPDATE распространяется каскадно для обновления внешнего ключа также и в этих согласованных поставках.
   - RESTRICT. Действие операции UPDATE ограничивается тем случаем, когда отсутствуют такие согласованные поставки (в противном случае эта операция отвергается).
   - NO ACTION. Операция UPDATE выполняется в точном соответствии с указанием (но в дальнейшем может произойти нарушение ссылочной целостности).
2. Безусловно, CASCADE, RESTRICT и NO ACTION не являются единственными возможными ссылочными действиями; они просто относятся к таким типам, которые часто требуются на практике. А в принципе может существовать произвольное количество допустимых ответов (например) на попытку удалить данные о некотором поставщике. Ниже приведено несколько примеров.
   - Такая попытка может быть по определенной причине сразу же отвергнута.
   - Информация может быть записана в некоторую архивную базу данных.
   - Поставки, относящиеся к рассматриваемому поставщику, могут быть переданы некоторому другому поставщику.

   Практически не осуществима задача предусмотреть декларативный синтаксис для всех ответов, которые можно себе представить. Поэтому в общем должна быть предусмотрена возможность определять ссылочное действие, состоящее из произвольной процедуры, определяемой пользователем (см. следующий раздел). Более того, выполнение этой процедуры должно рассматриваться как часть выполнения оператора, вызвавшего соответствующую проверку целостности; эта проверка целостности должна осуществляться повторно, после выполнения указанной процедуры (поскольку очевидно, что эта процедура не должна оставлять базу данных в таком состоянии, при котором в ней нарушено указанное ограничение).
3. Предположим, что R2 и R1, соответственно, — ссылающаяся переменная отношения и связанная с ней переменная отношения, указанная в ссылке, как показано ниже.

   R2 ⟶ R1

   Допустим, что в применяемом правиле удаления задано действие CASCADE. В таком случае выполнение операции DELETE над указанным кортежем R1 влечет за собой (в общем) выполнение операции DELETE над некоторыми кортежами переменной отношения R2. Если же теперь предположить, что на переменную отношения R2, в

свою очередь, ссылается некоторая другая переменная отношения R3, как показано ниже, то, по сути, выполнение подразумеваемой операции DELETE на кортежах R2 равносильно попытке непосредственного удаления этих кортежей.

```
R3 ⟶ R2 ⟶ R1
```

Это означает, что результаты выполнения последней операции зависят от того, какое правило удаления определено для ограничения ссылочной целостности, направленного от R3 к R2. Если эта подразумеваемая операция DELETE завершается неудачей (согласно правилу удаления, которое связывает R3 с R2, или по какой-либо иной причине), то окончится неудачей вся операция и база данных остается неизменной. Подобная процедура распространяется рекурсивно на любое количество уровней.

Аналогичные замечания относятся также к правилу каскадного обновления, с учетом соответствующих поправок, если внешний ключ переменной отношения R2 имеет какие-либо общие атрибуты с потенциальным ключом той переменной отношения, на которую ссылается внешний ключ переменной отношения R3.

4. Из сказанного выше следует, что с логической точки зрения операции обновления базы данных всегда являются атомарными, или неразрывными (выполняются по принципу “все или ничего”), даже если они фактически связаны с осуществлением нескольких операций обновления нескольких переменных отношения, например, в связи с тем, что участвуют в каскадном ссылочном действии.

## 9.11. ТРИГГЕРЫ (НЕБОЛЬШОЕ ОТСТУПЛЕНИЕ)

Из всего сказанного выше в данной главе должно быть очевидно, что для нас особый интерес представляет декларативная поддержка целостности. И хотя ситуация за последние годы улучшилась, остается фактом, что лишь немногие продукты (если они вообще есть) обеспечивают такую поддержку со времени своего первоначального появления на рынке. Вследствие этого ограничения целостности чаще всего реализуются процедурно с использованием **триггерных процедур**. Последние представляют собой заранее откомпилированные процедуры, которые хранятся вместе с базой данных (возможно, в самой базе данных) и вызываются автоматически при возникновении некоторых указанных событий. В частности, пример 1 (“значения статуса должны находиться в пределах от 1 до 100 включительно”) может быть реализован с помощью триггерной процедуры, которая вызывается каждый раз при вставке кортежа в переменную отношения S, проверяет этот вновь вставляемый кортеж и снова его удаляет, если значение статуса не входит в указанные пределы. В этом разделе приведено краткое описание триггерных процедур в связи с тем, что они имеют значительную практическую важность. Но необходимо сразу же привести следующие замечания.

1. Именно потому, что они являются процедурами, триггерные процедуры нельзя считать рекомендуемым способом реализации ограничений целостности. Пользователям сложнее понять, как действуют процедуры, а для системы процедуры создают дополнительные трудности при оптимизации. Следует также отметить, что декларативные ограничения проверяются при всех соответствующих обнов-

лениях[18], а триггерные процедуры выполняются только при возникновении указанного события (допустим, при вставке кортежа в переменную отношения S).

2. Область применения триггерных процедур не ограничивается задачами поддержки целостности, которые являются темой настоящей главы. Вместо этого, если учесть замечания, сделанные в п. 1, фактически они могут выполнять другие полезные задачи и именно поэтому имеют право на существование. Некоторые примеры таких "других полезных задач" приведены ниже.

   **а)** Передача пользователю предупреждения о том, что возникло некоторое исключение (например, выдача предупреждающего сообщения, если наличное количество некоторых деталей на складе становится ниже критического уровня).

   **б)** Отладка (т.е. отслеживание ссылок на указанные переменные и/или контроль над изменениями состояния этих переменных).

   **в)** Аудит (например, регистрация информации о том, кто и когда внес те или иные изменения в определенные переменные отношения).

   **г)** Измерение производительности (например, регистрация времени наступления или трассировка указанных событий в базе данных).

   **д)** Проведение компенсирующих действий (например, каскадная организация удаления кортежа поставщика для удаления также соответствующих кортежей поставок)[19].

   Поэтому данный раздел, как и указано в его названии, носит характер отступления от основной темы.

Рассмотрим следующий пример. (Этот пример основан на языке SQL, а не на языке Tutorial D, поскольку в [3.3] не предписана, и не могла быть предписана какая-либо поддержка триггерных процедур; фактически он основан на коммерческом программном продукте, а не на стандарте SQL, поскольку стандарт SQL не поддерживает конкретное средство, показанное в данном примере.) Предположим, что в базе данных имеется представление LONDON_SUPPLIER, которое определено следующим образом.

```
CREATE VIEW LONDON_SUPPLIER
    AS SELECT S#, SNAME, STATUS
       FROM   S
       WHERE  CITY = 'London' ;
```

При обычных обстоятельствах, если пользователь попытается вставить строку в это представление, то среда поддержки языка SQL действительно вставит строку в соответствующую базовую таблицу S с таким значением CITY, которое задано по умолчанию для

---

[18] Обратите внимание на то, что в спецификациях декларативных ограничений не предусмотрены явные указания для СУБД, когда должны выполняться проверки целостности. А это и не требуется, во-первых, потому, что такие явные указания вынуждали бы пользователя, объявляющего ограничения, выполнять лишнюю работу, во-вторых, потому, что пользователь мог бы их задать неправильно. Вместо этого желательно, чтобы сама система решала, когда должны проводиться эти проверки (см. аннотацию к [9.5]).

[19] И действительно, каскадное удаление — это типичный пример триггерной процедуры. Но заслуживает внимания то, что такое удаление задано декларативно! Автор отнюдь не утверждает, что ссылочные действия — это неудачная идея только потому, что они реализуются "триггерно" (т.е. как реакция на некоторое событие).

столбца CITY (см. главу 10). При условии, что по умолчанию задан город, отличный от Лондона, общий эффект этого действия окажется таким, что новая строка не появится в этом представлении! Поэтому создадим триггерную процедуру следующим образом.

```
CREATE TRIGGER LONDON_SUPPLIER_INSERT
       INSTEAD OF INSERT ON LONDON_SUPPLIER
       REFERENCING NEW ROW AS R
       FOR EACH ROW
       INSERT INTO S ( S#, SNAME, STATUS, CITY )
              VALUES ( R.S#, R.SNAME, R.STATUS, 'London' ) ;
```

Теперь вставка строки в это представление повлечет за собой то, что строка будет вставлена в соответствующую базовую таблицу со значением CITY, равным London, а не со значением, применяемом по умолчанию (и новая строка теперь будет появляться в представлении, в полном соответствии с поставленной задачей).

Из этого примера следуют определенные выводы, приведенные ниже.

***Примечание.*** Эти выводы не относятся только к языку SQL, несмотря на тот факт, что приведенный пример основан на SQL (конкретные сведения о средствах SQL даны в следующем разделе).

1. В общем, в операторе создания триггера CREATE TRIGGER, кроме всего прочего, определены событие, условие и действие следующим образом:

   - **событием** является операция в базе данных (в этом примере "INSERT ON LONDON_SUPPLIER");
   - **условие** — это логическое выражение, которое должно принимать значение TRUE для того, чтобы было выполнено действие (если условие не указано явно, как в данном примере, то оно по умолчанию равно просто TRUE);
   - **действие** — это и есть сама триггерная процедура (в данном примере "INSERT INTO S …").

   Событие и условие вместе иногда называют *триггерным событием*, а сочетание всех трех компонентов (событие, условие и действие) обычно называют просто **триггером**. По очевидным причинам триггеры именуются также правилами "событие–условие–действие" (Event–Condition–Action — ECA) или сокращенно правилами ECA.

2. К возможным событиям относится выполнение операций INSERT, DELETE, UPDATE (возможно, над определенными атрибутами), достижение конца транзакций (COMMIT), наступление указанного времени суток, истечение определенного интервала времени, нарушение указанного ограничения и т.д.
3. В общем, действие может выполняться до (BEFORE), после (AFTER) или вместо (INSTEAD OF) действия, обусловленного указанным событием, при условии, что эти варианты имеют смысл.
4. В общем, действие может выполняться для каждой строки (FOR EACH ROW) или для каждого оператора (FOR EACH STATEMENT), при условии, что эти варианты имеют смысл.

5. В общем, при выполнении действия, определяемого триггером, должен быть предусмотрен способ, позволяющий ссылаться на данные в том виде, какой они имеют до и после возникновения указанного события, при условии, что применение такого средства действительно имеет смысл.
6. Базу данных, которая имеет связанные с ней триггеры, иногда называют *активной базой данных*.

В завершение этого раздела следует отметить, что триггеры, безусловно, имеют важную область применения, но ими следует пользоваться с осторожностью и, вероятно, не прибегать к ним вообще, если есть альтернативный способ решения насущной задачи. Ниже перечислены некоторые причины, по которым применение триггеров на практике может вызвать появление определенных проблем.

- Если одно и то же событие вызывает *запуск* (как принято называть это событие на профессиональном жаргоне) нескольких разных триггеров, то последовательность их активизации может оказаться, с одной стороны, важной, а с другой — неопределенной.
- Могут возникать цепочки запуска триггеров, при которой запуск триггера `T1` вызывает активизацию триггера `T2`, который вызывает активизацию триггера `T3` и т.д.
- Запуск триггера `T` может даже снова вызвать рекурсивный запуск самого этого триггера.
- В результате наличия триггеров даже "простые" операции `INSERT`, `DELETE` или `UPDATE` могут приводить к такому эффекту, который принципиально отличается от ожидаемого пользователем (особенно если задано ключевое слово `INSTEAD OF`, как в приведенном выше примере).

Если учесть все описанные выше замечания, то должно быть очевидно, что суммарные результаты действия некоторой определенной совокупности триггеров могут оказаться весьма сложными для понимания. Декларативные решения, если они возможны, всегда являются более предпочтительными по сравнению с процедурными.

## 9.12. СРЕДСТВА SQL

Начнем с описания поддержки в языке SQL (или, скорее, по большей части, с констатации отсутствия такой поддержки) схемы классификации ограничений, описанной в разделе 9.9.

- В языке SQL вообще не поддерживаются *ограничения типа*, за исключением тех примитивных ограничений, которые являются прямым следствием применения определенного физического представления. Например, как было показано в главе 5, допустимо утверждать, что значения типа `WEIGHT` должны быть представлены в виде чисел `DECIMAL(5,1)`, но нельзя указать, что эти числа должны быть больше нуля и меньше 5000.
- В языке SQL (безусловно) поддерживаются *ограничения атрибута*.
- В языке SQL не поддерживаются *ограничения переменной отношения* как таковые. В нем обеспечивается поддержка ограничений базовой таблицы, но такие ограничения распространяются именно только на базовые таблицы, а не на все таблицы

в целом (в частности, они не охватывают представления), и они не сводятся к упоминанию просто одной такой таблицы, но фактически могут иметь произвольную сложность.

- В языке SQL не поддерживаются *ограничения базы данных* как таковые. В нем поддерживаются общие ограничения, которые в спецификации этого языка называются *утверждениями* (assertion), но не обязательно нужно использовать лишь такие ограничения, если требуется указать в ограничении больше одной таблицы. (В действительности, ограничения базовой таблицы и общие ограничения языка SQL являются логически взаимозаменяемыми, за исключением той странности, которая отмечена в самом конце следующего подраздела, "Ограничения базовой таблицы".)

В языке SQL отсутствует также непосредственная поддержка ограничений перехода, но такие ограничения могут быть реализованы процедурно с помощью триггеров. Кроме того, в этом языке не представлено явно понятие предиката переменной отношения (или таблицы), а эта особенность языка SQL является очень важной, как показано в следующей главе.

### Ограничения базовой таблицы

Ограничения базовой таблицы в языке SQL задаются в операторе `CREATE TABLE` или `ALTER TABLE`. Каждое такое ограничение представляет собой ограничение потенциального ключа, ограничение внешнего ключа или ограничение проверки. Рассмотрим каждое из них по очереди.

***Примечание.*** Перед определением любого из этих ограничений может находиться необязательная спецификация `CONSTRAINT <constraint name>`, которая позволяет присвоить ограничению имя. Автор не рассматривает эту опцию для сокращения изложения (но необходимо отметить, что на практике, по-видимому, следует присваивать имена всем ограничениям). Здесь также не рассматриваются некоторые сокращения, например, возможность задавать потенциальный ключ (как "встроенный" в составе определения столбца), по той же причине.

### Потенциальные ключи

Любое определение потенциального ключа SQL принимает одну из следующих двух форм.

```
PRIMARY KEY ( <column name commalist> )
UNIQUE ( <column name commalist> )
```

В обоих случаях выражение `<column name commalist>` с разделенным запятыми списком имен столбцов не должно быть пустым[20]. Любая конкретная базовая таблица может иметь не больше одной спецификации с определением первичного ключа `PRIMARY KEY`, но любое количество спецификаций потенциального ключа `UNIQUE`. В случае `PRIMARY KEY` каждый указанный столбец дополнительно рассматривается как соответствующий требованию `NOT NULL`, т.е. не содержащий пустых значений, даже если ключевое слово `NOT NULL` не задано явно (см. приведенное ниже описание ограничений проверки).

---

[20] См. упражнение 9.10.

### Внешние ключи

Определение внешнего ключа языка SQL принимает следующую форму.

```
FOREIGN KEY ( <column name commalist> )
  REFERENCES <base table name> [ ( <column name commalist> ) ]
[ ON DELETE <referential action> ]
[ ON UPDATE <referential action> ]
```

Здесь спецификация ссылочного действия <*referential action*> может принимать значение NO ACTION (по умолчанию), RESTRICT, CASCADE, SET DEFAULT или SET NULL[21]. Отложим обсуждение конструкций SET DEFAULT и SET NULL до главы 19; другие опции описаны в разделе 9.10. Вторая спецификация <*column name commalist*> требуется, если внешний ключ ссылается на потенциальный ключ, который не является первичным ключом.

***Примечание.*** Согласование внешнего и потенциального ключей осуществляется не на основе имен столбцов, а с учетом позиции столбца (слева направо) в разделенном запятыми списке.

### Ограничения проверки

Любое ограничение проверки в языке SQL принимает следующую форму.

```
CHECK ( <bool exp> )
```

Допустим, что CC — ограничение проверки для базовой таблицы T. В таком случае считается, что в таблице T ограничение CC нарушается тогда и только тогда, когда эта таблица в настоящее время содержит по меньшей мере одну строку (см. последний абзац данного подраздела) и проверка текущего значения T влечет за собой то, что логическое выражение <*bool exp*> для CC принимает значение FALSE.

***Примечание.*** В общем, следует отметить, что выражения <*bool exp*> языка SQL могут быть сколь угодно сложными; даже в данном рассматриваемом контексте они явно не ограничиваются ссылками только на базовую таблицу T, но могут вместо этого ссылаться на любую доступную часть базы данных.

Ниже приведен пример оператора CREATE TABLE, в котором иллюстрируется применение ограничений базовой таблицы всех трех видов.

```
CREATE TABLE SP
     ( S# S# NOT NULL, P# P# NOT NULL, QTY QTY NOT NULL,
       PRIMARY KEY ( S#, P# ),
       FOREIGN KEY ( S# ) REFERENCES S
                          ON DELETE CASCADE
                          ON UPDATE CASCADE,
       FOREIGN KEY ( P# ) REFERENCES P
                          ON DELETE CASCADE
                          ON UPDATE CASCADE,
       CHECK ( QTY ≥ QTY ( 0 ) AND QTY ≤ QTY ( 5000 ) ) ) ;
```

---

[21] Кстати, следует отметить, что для поддержки некоторых действий типа <referential action> (в частности CASCADE) требуется, чтобы система (хотя бы неявно) поддерживала некоторые виды операций множественного реляционного присваивания! Причем это требование существует несмотря на то, что подобные операции не поддерживаются в языке SQL как таковые.

Здесь предполагается, что атрибуты S# и P# были явно определены как предназначенные для использования, соответственно, в качестве первичных ключей для таблиц S и P. Кроме того, в этом операторе используется сокращение, согласно которому ограничение проверки в указанной ниже форме может быть заменено простой спецификацией NOT NULL в определении рассматриваемого столбца <column name>.

```
CHECK ( <column name> IS NOT NULL )
```

Поэтому в данном примере три ограничения проверки, которые могли оказаться довольно сложными, заменены тремя спецификациями NOT NULL.

В заключение этого подраздела еще раз повторим замечание, что ограничение базовой таблицы SQL всегда считается удовлетворенным, если рассматриваемая базовая таблица оказалась пустой, даже если это ограничение имеет форму (скажем) "1 = 2" (или даже, если на то пошло, оно по сути имеет такую форму, что "эта таблица не должна быть пустой"!).

### Утверждения

Теперь перейдем к описанию общих ограничений SQL, или утверждений. Такие ограничения определяются с помощью оператора CREATE ASSERTION, который имеет показанный ниже синтаксис.

```
CREATE ASSERTION <constraint name>
       CHECK ( <bool exp> ) ;
```

А ниже показан синтаксис оператора DROP ASSERTION.

```
DROP ASSERTION <constraint name> ;
```

Обратите внимание на то, что в отличие от большинства других форм оператора DROP языка SQL (например, DROP TYPE, DROP TABLE, DROP VIEW), в операторе DROP ASSERTION не предусмотрены две противоположные опции RESTRICT и CASCADE.

Ниже приведено шесть примеров из раздела 9.1, выраженных в форме утверждений языка SQL. Рекомендуем читателю в качестве упражнения попытаться вместо этого сформулировать данные примеры в виде ограничений базовой таблицы.

1. *Значение статуса каждого поставщика должно находиться в пределах от 1 до 100 включительно.*

```
CREATE ASSERTION SC1 CHECK
     ( NOT EXISTS ( SELECT * FROM S
                    WHERE  S.STATUS < 0
                    OR     S.STATUS > 100 ) ) ;
```

2. *Каждый поставщик из Лондона имеет статус 20.*

```
CREATE ASSERTION SC2 CHECK
     ( NOT EXISTS ( SELECT * FROM S
                    WHERE  S.CITY = 'London'
                    AND    S.STATUS ≠ 20 ) ) ;
```

3. *Если вообще имеются какие-либо детали, то по меньшей мере одна из них должна быть синего цвета.*

```
CREATE ASSERTION PC3 CHECK
     ( NOT EXISTS ( SELECT * FROM P )
       OR  EXISTS ( SELECT * FROM P
                    WHERE  P.COLOR = COLOR ('Blue') ) ) ;
```

4. *Разные поставщики не могут иметь одинаковые номера поставщиков.*

```
CREATE ASSERTION SC4 CHECK
     ( UNIQUE ( SELECT S.S# FROM S ) ) ;
```

   В этом операторе UNIQUE представляет собой операцию SQL, которая принимает в качестве фактического параметра таблицу и возвращает значение TRUE, если эта таблица не содержит дубликатов строк, а в противном случае возвращает значение FALSE.

5. *Каждая поставка выполняется существующим поставщиком.*

```
CREATE ASSERTION SSP5 CHECK
     ( NOT EXISTS
         ( SELECT * FROM SP
           WHERE  NOT EXISTS
                ( SELECT * FROM S
                  WHERE  S.S# = SP.S# ) ) ) ;
```

6. *Ни один поставщик со статусом меньше 20 не поставляет любые детали в количестве больше 500.*

```
CREATE ASSERTION SSP6 CHECK
     ( NOT EXISTS ( SELECT * FROM S, SP
                    WHERE  S.STATUS < 20
                    AND    S.S# = SP.S#
                    AND    SP.QTY > QTY ( 500 ) ) ) ;
```

Кратко рассмотрим еще один пример. Для этого обратимся к приведенному ниже определению представления из предыдущего раздела.

```
CREATE VIEW LONDON_SUPPLIER
    AS SELECT S#, SNAME, STATUS
       FROM   S
       WHERE  CITY = 'London' ;
```

Как уже было сказано, в это определение представления нельзя включать спецификацию в такой форме.

```
UNIQUE ( S# )
```

Но, как ни странно, в этом определении можно задать общее ограничение в следующей форме.

```
CREATE ASSERTION LSK CHECK
     ( UNIQUE ( SELECT S# FROM LONDON_SUPPLIER ) ) ;
```

### Отложенная проверка

Ограничения SQL отличаются также от ограничений, описанных в предыдущих разделах данной главы, в той части, которая касается выполнения проверки. В рассматриваемой выше схеме все ограничения проверяются немедленно. В отличие от этого, в

языке SQL ограничения[22] могут быть определены как допускающие отложенную проверку (DEFERRABLE) или не допускающие такую проверку (NOT DEFERRABLE); если заданное ограничение объявлено как DEFERRABLE, оно может быть дополнительно определено как отложенное первоначально (INITIALLY DEFERRED) или немедленно выполняемое с самого начала (INITIALLY IMMEDIATE); эти ключевые слова определяют состояние ограничения в начале каждой транзакции. Ограничения NOT DEFERRABLE всегда проверяются немедленно, а проверку ограничений DEFERRABLE можно динамически включать и выключать с помощью следующего оператора.

```
SET CONSTRAINTS <constraint name commalist> <option> ;
```

Здесь опция <*option*> принимает значение IMMEDIATE или DEFERRED. Пример применения такой опции приведен ниже.

```
SET CONSTRAINTS SSP5, SSP6 DEFERRED ;
```

Ограничения DEFERRABLE проверяются, только если они находятся в состоянии IMMEDIATE. Перевод ограничения DEFERRABLE в состояние IMMEDIATE вызывает немедленную проверку этого ограничения; если данная проверка оканчивается неудачей, то и выполнение оператора SET IMMEDIATE оканчивается неудачей. Выполнение оператора COMMIT влечет за собой вызов операторов SET IMMEDIATE для всех ограничений DEFERRABLE; если после этого хотя бы одна из проверок целостности оканчивается неудачей, происходит откат транзакции.

### Триггеры

Оператор CREATE TRIGGER языка SQL выглядит следующим образом.

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

Пояснения к этому оператору приведены ниже.

1. Спецификация с указанием времени проверки до или после активизации триггера, <*before or after*>, принимает значение BEFORE или AFTER (в стандарте SQL не поддерживается ключевое слово INSTEAD OF, но в некоторых программных продуктах такая поддержка предусмотрена).
2. Событие <*event*> может принимать значение INSERT, DELETE или UPDATE. Значение UPDATE может дополнительно уточняться с помощью спецификации OF <*column name commalist*>.
3. Каждое определение именования <*naming*> может принимать одну из следующих форм.

---

[22] Но некоторые ограничения должны быть обязательно указаны как относящиеся к типу NOT DEFERRABLE. Например, если FK — внешний ключ, то ограничение потенциального ключа для соответствующего потенциального ключа должно быть задано как NOT DEFERRABLE.

```
OLD ROW   AS <name>
NEW ROW   AS <name>
OLD TABLE AS <name>
NEW TABLE AS <name>
```

4. Спецификация с определением строки или оператора `<row or statement>` принимает значение `ROW` или `STATEMENT` (`STATEMENT` применяется по умолчанию). Ключевое слово `ROW` означает, что триггер активизируется для каждой отдельной строки, на которую распространяется действие триггерного оператора, а `STATEMENT` означает, что триггер активизируется только один раз для данного оператора, рассматриваемого как единое целое.
5. Если определена конструкция `WHEN`, она означает, что действие `<action>` должно выполняться, только если логическое выражение `<bool exp>` принимает значение `TRUE`.
6. Спецификация `<action>` задает отдельный оператор SQL (но этот отдельный оператор может быть достаточно сложным, т.е. составным, а это неформально означает, что такой оператор может состоять из последовательности операторов, обозначенных разграничителями `BEGIN` и `END`).

   Наконец, ниже показан синтаксис оператора `DROP TRIGGER`.

```
DROP TRIGGER <trigger name> ;
```

Как и в операторе `DROP ASSERTION`, в операторе `DROP TRIGGER` не предусмотрено использование пары противоположных опций `RESTRICT` и `CASCADE`.

## 9.13. РЕЗЮМЕ

В настоящей главе рассматривается важное понятие **целостности**. Задача обеспечения целостности представляет собой задачу обеспечения правильности данных в базе данных (по меньшей мере, обеспечения правильности в максимально возможной степени; к сожалению, лучшее, чего мы можем достичь, состоит в обеспечении совместимости данных с установленными ограничениями). Безусловно, для нас наибольший интерес представляют **декларативные** решения этой задачи.

В начале этой главы было показано, что ограничения целостности приведенную ниже общую форму.

*Если (`IF`) некоторые кортежи присутствуют в некоторых переменных отношения, то (`THEN`) эти кортежи удовлетворяют некоторому условию.*

(Ограничения типа характеризуются определенными отличиями, как показано в приведенном ниже описании.) В этой главе рассматривается синтаксис определения таких ограничений, основанный на той версии языка Tutorial D, в которой применяется исчисление предикатов, и указано, что в этом синтаксисе не предусмотрено каких-либо способов, с помощью которых пользователь мог бы сообщить СУБД, когда должна быть выполнена проверка таких ограничений, поскольку желательно, чтобы время этой проверки определяла сама СУБД. Кроме того, в данной главе утверждается (но пока еще без обоснования такой позиции), что вся проверка ограничений должна выполняться **немедленно**.

Затем было показано, что любое ограничение в том виде, в каком оно сформулировано, представляет собой **предикат**, а при его проверке (т.е. при подстановке текущих значений отношений вместо переменных отношения, указанных в этом предикате), оно становится **высказыванием**. **Предикатом переменной отношения** для определенной переменной отношения является логическое выражение, состоящее из всех предикатов, которые применяются к данной переменной отношения, соединенных операторами "И", а **предикатом базы данных** для определенной базы данных является логическое выражение, состоящее из всех предикатов, которые применяются к этой базе данных, соединенных операторами "И". Кроме того, в этой главе сформулировано **золотое правило**, приведенное ниже.

> *Ни одна операция обновления не должна приводить к присваиванию любой базе данных такого значения, которое вызывает то, что предикат этой базы данных получает значение* FALSE.

Затем в этой главе показано различие между **внутренними** и **внешними** предикатами. Внутренние предикаты заданы формально. Об этих предикатах имеются сведения в системе, а их проверка выполняется СУБД (предикаты переменной отношения и предикаты базы данных, о которых шла речь в предыдущем абзаце, являются внутренними предикатами). В отличие от внутренних предикатов, внешние предикаты задаются только неформально. Сведения о них известны только пользователю, но не системе. **Предположение о замкнутости мира** относится к внешним предикатам, а не ко внутренним.

Кстати, как уже может быть известно читателю, то, что обычно называется *обеспечением целостности*, в контексте базы данных фактически определяет **семантику**, поскольку **смысл** данных определяется именно ограничениями целостности (в частности предикатами переменной отношения и базы данных). И в этом состоит одна из причин, по которой обеспечение целостности является такой исключительно важной задачей, как было указано во введении к этой главе.

Кроме того, было указано (и это вполне укладывается в рамки здравого смысла), что требования поддержки целостности распространяются на все переменные отношения (в частности, они распространяются на представления), несмотря на то, что, безусловно, ограничения, применяемые к конкретному представлению, могут быть выведены из тех ограничений, которые применяются к переменным отношения, послужившим основой для создания рассматриваемого представления.

В дальнейшем изложении было показано, что ограничения целостности подразделяются на четыре описанные ниже категории.

- Ограничения **типа** определяют допустимые значения для конкретного типа (или домена) и проверяются во время вызова соответствующего селектора.
- Ограничения **атрибута** задают допустимые значения для конкретного атрибута, и если предусмотрена проверка ограничений типа, то вероятность нарушения ограничений атрибута исключена.
- Ограничения **переменной отношения** задают допустимые значения для конкретной переменной отношения и проверяются при обновлении рассматриваемой переменной отношения.
- Ограничения **базы данных** задают допустимые значения для конкретной базы данных и проверяются при обновлении рассматриваемой базы данных.

Но было указано, что различия между ограничениями переменной отношения и базы данных в большей степени относятся к сфере практики, чем логики. Кроме того, были кратко описаны ограничения **перехода**.

Затем были описаны такие важные с точки зрения практики частные случаи ограничений, как **потенциальные**, **первичные**, **альтернативные** и **внешние** ключи. Потенциальные ключи обладают свойствами **уникальности** и **несократимости**, причем каждая переменная отношения (без каких-либо исключений!) должна иметь по меньшей мере один такой ключ. Ограничения, согласно которым значения определенного внешнего ключа должны совпадать со значениями соответствующего потенциального ключа, называются **ограничениями ссылочной целостности**; в этой главе описано несколько областей применения идеи ссылочной целостности, включая, в частности, понятие **ссылочных действий** (причем наиболее важной из этих областей применения является каскадное распространение действий с помощью ключевого слова `CASCADE`). По материалам обсуждения этой последней темы было сделано краткое отступление, касающееся темы **триггеров**.

В завершение данной главы были описаны соответствующие средства языка SQL. Ограничения типа в языке SQL развиты очень слабо; по сути, они сводятся к определению того, что рассматриваемый тип должен иметь определенное физическое представление. Ограничения базовой таблицы SQL (которые обеспечивают в качестве частного случая поддержку для ключей) и общие ограничения ("утверждения"), представляют собой аналоги ограничений переменной отношения и базы данных (кроме ограничений перехода), но их классификация определена гораздо менее четко по сравнению с языком Tutorial D (фактически они являются почти взаимозаменяемыми и не совсем понятно, почему в языке SQL предусмотрены и ограничения базовой таблицы, и общие ограничения). Кроме того, язык SQL поддерживает **отложенную проверку ограничений**. Наконец, в этой главе кратко показано, как в языке SQL осуществляется поддержка триггеров.

## УПРАЖНЕНИЯ

**9.1.** Какие операции могут вызвать нарушение ограничений, которые определены в примерах 1–6 раздела 9.1?

**9.2.** Приведите формулировки примеров 1–6 из раздела 9.1 на той версии языка Tutorial D, в которой применяется алгебра. Какие формулировки, по вашему мнению, являются более удобными — на основе исчисления предикатов или алгебры? Почему?

**9.3.** Составьте ограничения целостности для приведенных ниже *бизнес-правил* для базы данных поставщиков, деталей и проектов с использованием синтаксиса Tutorial D на основе исчисления предикатов, который описан в разделе 9.2.

**а)** В базе данных могут находиться данные только о следующих городах: Лондон, Париж, Рим, Афины, Осло, Стокгольм, Мадрид и Амстердам (`London`, `Paris`, `Rome`, `Athens`, `Oslo`, `Stockholm`, `Madrid`, `Amsterdam`).

**б)** Единственно допустимыми номерами поставщиков являются такие номера, которые могут быть представлены в виде символьной строки с длиной меньше двух символов, из которых первым является "S", а остальные обозначают десятичное целое число в пределах от 0 до 9999.

**в)** Все детали красного цвета должны иметь вес меньше 50 фунтов.

**г)** Никакие два проекта не могут находиться в одном и том же городе.

**д)** В Афинах не могут находиться одновременно больше одного поставщика.

**е)** Ни одна поставка не может иметь объем, превышающий больше чем в два раза средний объем всех таких поставок.

**ж)** Поставщик с самым высоким статусом не должен находиться в том же городе, где находится поставщик с самым низким статусом.

**з)** Каждый проект должен находиться в том городе, где находится по меньшей мере один поставщик.

**и)** Каждый проект должен находиться в том городе, где находится по меньшей мере один поставщик деталей для этого проекта.

**к)** Должна существовать по меньшей мере одна деталь красного цвета.

**л)** Средний статус поставщика должен быть больше 19.

**м)** Каждый поставщик из Лондона должен поставлять деталь с номером P2.

**н)** По меньшей мере одна деталь красного цвета должна иметь вес меньше 50 фунтов.

**о)** Поставщики из Лондона должны поставлять больше деталей разных видов, чем поставщики из Парижа.

**п)** В общем поставщики из Лондона должны поставлять больше деталей, чем поставщики из Парижа.

**р)** Ни один объем поставки не может быть сокращен (в одной операции обновления) меньше чем наполовину его текущего значения.

**с)** Поставщики из Афин могут переезжать только в Лондон или Париж, а поставщики из Лондона могут переезжать только в Париж.

**9.4.** Применительно к каждому из полученных вами ответов на упр. 9.3 выполните следующее:

**а)** укажите, является ли сформированное ограничение ограничением переменной отношения или ограничением базы данных;

**б)** приведите примеры операций, которые могут вызвать нарушение ограничения.

**9.5.** С использованием примеров данных о поставщиках, деталях и проектах, приведенных на рис. 4.5 (см. стр. 154), определите, каким будет результат каждой из следующих операций:

**а)** обновление с помощью операции UPDATE данных о проекте J7 — присваивание атрибуту CITY значения New York;

**б)** обновление с помощью операции UPDATE данных о детали P5 — присваивание атрибуту P# значения P4;

**в)** обновление с помощью операции UPDATE данных о поставщике S5 — присваивание атрибуту S# значения S8, если в качестве соответствующего ссылочного действия определено RESTRICT;

**г)** удаление с помощью операции DELETE данных о поставщике S3, если в качестве соответствующего ссылочного действия определено CASCADE;

**д)** удаление с помощью операции DELETE данных о детали P2, если в качестве соответствующего ссылочного действия определено RESTRICT;

**е)** удаление с помощью операции DELETE данных о проекте J4, если в качестве соответствующего ссылочного действия определено CASCADE;

**ж)** обновление с помощью операции UPDATE данных о поставке S1-P1-J1 — присваивание атрибуту S# значения S2;

**з)** обновление с помощью операции UPDATE данных о поставке S5-P5-J5 — присваивание атрибуту J# значения J7;

**и)** обновление с помощью операции UPDATE данных о поставке S5-P5-J5 — присваивание атрибуту J# значения J8;

**к)** вставка с помощью операции INSERT данных о поставке S5-P6-J7;

**л)** вставка с помощью операции INSERT данных о поставке S4-P7-J6;

**м)** вставка с помощью операции INSERT данных о поставке S1-P2-jjj (где jjj представляет собой заданный по умолчанию номер проекта).

**9.6.** В данной главе рассматривались правила удаления и обновления внешнего ключа, но ни разу не упоминалось какое-либо "правило вставки" внешнего ключа. Объясните, почему не существует такого правила.

**9.7.** В базе данных повышения квалификации содержится информация о внутрифирменной системе повышения квалификации служащих компании. Для каждого курса обучения в базе данных имеются сведения обо всех подготовительных курсах, необходимых для освоения этого курса, и обо всех потоках, предусмотренных для прохождения этого курса; для каждого потока в базе данных имеются сведения обо всех преподавателях и обо всех служащих, зачисленных на этот поток. Кроме того, в базе данных хранится информация о служащих. Ниже приведены краткие определения соответствующих переменных отношения.

```
COURSE     { COURSE#, TITLE }           /* Курс          */
PREREQ     { SUP_COURSE#, SUB_COURSE# } /* Подготовительный курс */
OFFERING   { COURSE#, OFF#, OFFDATE, LOCATION } /* Поток    */
TEACHER    { COURSE#, OFF#, EMP# }      /* Преподаватель   */
ENROLLMENT { COURSE#, OFF#, EMP#, GRADE }  /* Зачисление  */
EMPLOYEE   { EMP#, ENAME, JOB }           /* Служащий    */
```

Смысл переменной отношения PREREQ состоит в том, что в ней показано, какой вспомогательный курс (SUB_COURSE#) применяется для непосредственной подготовки к усвоению основного курса (SUP_COURSE#). Назначение остальных переменных отношения должно быть понятно без каких-либо дополнительных пояснений. Начертите для этой базы данных соответствующую ссылочную диаграмму. Приведите также соответствующие определения базы данных (т.е. запишите соответствующий набор определений типов и переменных отношения).

**9.8.** Две следующие переменные отношения представляют базу данных, содержащую информацию об отделах и служащих.

```
DEPT { DEPT#, ..., MGR_EMP#, ... }
EMP { EMP#, ..., DEPT#, ... }
```

В каждом отделе есть руководитель (MGR_EMP#), и каждый из служащих работает в одном из отделов (DEPT#). Начертите ссылочную диаграмму и составьте необходимые определения данных для этой базы данных.

**9.9.** Две следующие переменные отношения представляют базу данных, содержащую информацию о служащих и программистах.

```
EMP  { EMP#, ..., JOB, ... }
PGMR { EMP#, ..., LANG, ... }
```

Каждый программист является служащим, но обратное утверждение неверно. И в этом случае начертите ссылочную диаграмму и составьте необходимые определения данных для этой базы данных.

**9.10.** Потенциальные ключи по определению являются множествами атрибутов. Что произойдет, если рассматриваемое множество окажется пустым (т.е. не содержащим атрибутов)? Можете ли вы представить себе какую-либо область применения такого "пустого" (или "нуль-арного") потенциального ключа?

**9.11.** Допустим, что R — переменная отношения степени n. Каково максимальное количество потенциальных ключей, которыми может обладать R?

**9.12.** Допустим, что A и B — две переменные отношения. Определите потенциальный ключ (ключи) для каждого из следующих операторов.

**а)** `A WHERE ...`

**б)** `A {...}`

**в)** `A TIMES B`

**г)** `A UNION B`

**д)** `A INTERSECT B`

**е)** `A MINUS B`

**ж)** `A JOIN B`

**з)** `EXTEND A ADD exp AS Z`

**и)** `SUMMARIZE A PER B ADD exp AS Z`

**к)** `A SEMIJOIN B`

**л)** `A SEMIMINUS B`

В каждом случае предполагается, что A и B соответствуют требованиям к рассматриваемой операции (например, в случае UNION они имеют один и тот же тип).

**9.13.** Повторно выполните упр. 9.10, заменив слово "потенциальный" словом "внешний" (дважды).

**9.14.** Приведите решение упр. 9.3 на языке SQL.

**9.15.** Составьте определения базы данных по условиям упр. 9.7–9.9 на языке SQL.

**9.16.** В данной главе было показано, что каждая переменная отношения (и фактически каждое отношение) соответствует некоторому предикату. Является ли истинным обратное утверждение?

**9.17.** В одной из сносок в разделе 9.7 указано, что если значения S1 и London присутствуют вместе в некотором кортеже, то это может означать (в числе многих других возможных интерпретаций), что поставщик S1 не имеет офиса в городе London. Фактически данная конкретная интерпретация чрезвычайно маловероятна. Объясните, почему.

***Подсказка.*** Вспомните предположение о замкнутости мира.

## СПИСОК ЛИТЕРАТУРЫ

**9.1.** Aiken A., Hellerstein J.M., and Widom J. Static Analysis Techniques for Predicting the Behavior of Active Database Rules // ACM TODS. — March 1995. — 20, № 1.

В этой статье продолжена работа, начатая в [9.2], [9.5], в том числе, над *системами экспертных баз данных* (здесь они именуются *системами активных баз данных*). В частности, в статье описана система правил, применяемая в прототипе Starburst компании IBM (см. [18.21], [18.48], [26.19], [26.23] , [26.29], [26.30] и [9.25]).

**9.2.** Baralis E. and Widom J. An Algebraic Approach to Static Analysis of Active Database Rules // ACM TODS. — September 2000. — 25, № 3. Ранняя версия этой статьи: Baralis E. and Widom J. An Algebraic Approach to Rule Analysis in Expert Database Systems // Proc. 20th Int. Conf. on Very Large Data Bases. — Santiago, Chile. — September 1994.

В этой статье под словом “правила”, вынесенным в заголовок, по сути подразумеваются триггеры. Одна из проблем, связанных с использованием таких правил, состоит в том, что (как отмечено в разделе 9.11) их поведение является исключительно трудным как для понимания, так и для прогнозирования. В статье предложены методы определения (еще до этапа вызова правил на выполнение) того, обладает ли данное множество правил свойствами завершенности и конфлюэнтности. Свойство *завершенности* означает, что обработка правила гарантированно не будет продолжаться бесконечно. Свойство *конфлюэнтности* означает, что окончательное состояние базы данных не зависит от порядка, в котором выполнялись эти правила.

**9.3.** Bernstein P.A., Blaustein B.T., Clarke E.M. Fast Maintenance of Semantic Integrity Assertions Using Redundant Aggregate Data // Proc. 6th Intern. Conf. on Very Large Data Bases. — Montreal, Canada. — October 1980.

Здесь представлен эффективный метод приведения в действие ограничений целостности особого рода, например, таких: “Любое значение в множестве A должно быть меньше любого значения в множестве B”. Метод предписания этих ограничений основан, в частности, на том наблюдении, что данное ограничение, по сути, логически эквивалентно ограничению: “Максимальное значение в множестве A должно быть меньше минимального значения в множестве B”. Распознавая ограничения подобного рода и автоматически сопровождая соответствующие максимальные и минимальные значения в скрытых переменных, система может сократить количество сравнений, связанных с соблюдением ограничения на данное обновление от величины, порядок которой определяется кардинальностью множеств A или B (в зависимости от того, к какому множеству применяется обновление)

примерно до единицы, безусловно, за счет расходов на сопровождение скрытых переменных.

**9.4.** Buneman O.P., Clemons E.K. Efficiently Monitoring Relational Databases // ACM TODS. — September 1979. — 4, № 3.

В статье рассматриваются задачи эффективной реализации триггерных процедур (которые здесь именуются *предупреждающими*), в частности, анализируется возможность определения того, удовлетворяется ли триггерное условие, без обязательного вычисления этого условия. В ней описан метод (алгоритм предотвращения — avoidance algorithm), позволяющий обнаружить обновления, которые, по всей вероятности, не будут удовлетворять данному триггерному условию. Кроме того, представлен способ сокращения издержек обработки в тех случаях, когда выполнение алгоритма предотвращения оканчивается неудачей. Этот способ предусматривает проведение оценки триггерного условия для некоторого небольшого подмножества (топологического фильтра) полного множества охватываемых кортежей.

**9.5.** Ceri S., Widom J. Deriving Production Rules for Constraint Maintenance // Proc. 16th Intern. Conf. on Very Large Data Bases. — Brisbane, Australia. — August 1990.

В статье рассматривается язык на основе SQL, предназначенный для определения ограничений, и приведен алгоритм, с помощью которого система может обнаружить все операции, способные нарушить указанное ограничение. (Краткое предварительное описание этого алгоритма было приведено в [9.12].) Данная статья касается также вопросов оптимизации и правильности.

**9.6.** Ceri S., Cochrane R.J., Widom J. Practical Applications of Triggers and Constraints: Successes and Lingering Issues // Proc. 26th Int. Conf. on Very Large Data Bases, Cairo, Egypt. — September 2000.

Приведем цитату из резюме: “Значительная часть ... триггерных приложений фактически представляет собой не что иное, как средства поддержки ограничений целостности различных классов”. По ходу изложения в данной статье показано, что многие триггеры, включая, в частности, триггеры, названные в статье “средствами поддержки ограничений”, действительно могут быть сгенерированы автоматически на основании декларативных спецификаций.

**9.7.** Ceri S., Fraternali P., Paraboschi S., and Tanca L. Automatic Generation of Production Rules for Integrity Maintenance // ACM TODS. — September 1994. — 19, № 3.

В статье, основанной на работе [9.5], представлены возможности автоматического исправления искажений, внесенных в результате нарушения ограничения. Ограничения транслируются в порождающие правила с помощью следующих компонентов.

1. Список операций, которые могут нарушить ограничение.
2. Логическое выражение, в результате вычисления которого будет получено значение TRUE, если ограничение нарушено (по сути, оно представляет собой просто отрицание первоначального ограничения).
3. Восстановительная процедура на языке SQL.

В статье приведен также широкий обзор работ, связанных с этой темой.

**9.8.** Cochrane R., Pirahesh H., and Mattos N. Integrating Triggers and Declarative Constraints in SQL Database System // Proc. 22 Int. Conf. on Very Large Data Bases. — Mumbai (Bombay), India. — September 1996.

Приведем цитату из этой статьи: "Семантика взаимодействия триггеров и декларативных ограничений должна быть тщательно определена, чтобы можно было избежать нарушений совместимости базы данных с ограничениями при выполнении операций и предоставить пользователям всестороннюю модель, позволяющую понять такие взаимодействия. Такая модель определена [в данной статье]". Рассматриваемая модель стала основой для разработки соответствующих спецификаций в стандарте SQL:1999.

**9.9.** Codd E.F. Domains, Keys, and Referential Integrity in Relational Databases // InfoDB3. — 1988. — 3, № 1.

Обсуждение понятий домена, первичного ключа и внешнего ключа. Безусловно, данная статья Кодда представляет значительный интерес, поскольку все эти три понятия были предложены Коддом. Тем не менее, по мнению автора данной книги, в этой статье слишком много вопросов осталось нерешенными или необъясненными. Кстати, в ней приведен следующий довод в пользу правила, согласно которому один из потенциальных ключей обязательно должен быть выбран в качестве первичного: "Отказ от использования этого правила равносилен стремлению использовать компьютер со схемой адресации ..., в которой основание системы счисления изменяется всякий раз, когда происходит событие определенного вида (например, встречается адрес, который представляет собой простое число)". Но если мы согласимся с этим доводом, то почему бы не согласиться с его логическим заключением и не использовать идентичную схему адресации для всех данных? Разве мы не допускаем такое же "упущение", когда адресуем данные о поставщике с помощью номера поставщика и данные о деталях с помощью номеров деталей, не говоря уже о поставках, когда применяются составные "адреса". (Фактически, по поводу идеи глобально единообразной схемы адресации можно сказать еще очень многое. См. обсуждение темы свернутых ключей в аннотации к [14.21] в главе 14.)

**9.10.** Date C.J. Referential Integrity // Proc. 7th Intern. Conf. on Very Large Data Bases. — Cannes, France. — September 1981. Позднее была издана пересмотренная версия этой статьи (Date C.J. Relational Database: Selected Writings. — Reading, Mass.: Addison-Wesley, 1986).

В статье впервые дано определение понятия *ссылочных действий* (преимущественно `RESTRICT` и `CASCADE`), которые обсуждались в разделе 9.10 этой главы. Основное различие между первой версией статьи (*VLDB*, 1981) и пересмотренной версией состоит в том, что в первой версии, которая была написана под влиянием [14.7], допускалось, чтобы внешний ключ ссылался на любое количество переменных отношения, тогда как в пересмотренной версии (по причинам, подробно описанным в [9.10]) автор уже не придерживается такой чрезмерно общей позиции.

**9.11.** Date C.J. Referential Integrity and Foreign Keys (в двух частях) // Relational Database Writings 1985–1989. — Reading, Mass.: Addison-Wesley, 1990.

В первой части этой статьи обсуждается история понятия ссылочной целостности, а также предлагается ряд предпочитаемых автором основных определений (с пояснениями). Во второй части приведены дополнительные доводы, позволяющие понять, почему были выбраны именно эти определения, и даны некоторые практические рекомендации. В частности, обсуждаются проблемы, которые могут возникать при использовании перекрывающихся внешних ключей, составных внешних ключей, частично представленных пустыми значениями, и *смежных ссылочных путей* (т.е. различимых ссылочных путей, начальная и конечная точки которых совпадают).

***Примечание.*** Определенные положения этой статьи могут быть поставлены под сомнение (но не очень существенно) в соответствии с доводами, изложенными в [9.14].

**9.12.** Date C.J. A Contribution to the Study of Database Integrity // Relational Database Writings 1985–1989. — Reading, Mass.: Addison-Wesley, 1990.

Приведем цитату из резюме: “В статье предпринята попытка в определенной степени структурировать задачу [обеспечения целостности]. Во-первых, в ней предложена схема классификации ограничений целостности, во-вторых, эта схема используется для разъяснения основополагающих понятий целостности данных, в-третьих, кратко описан подход к созданию конкретного языка формулирования ограничений целостности, и, в-четвертых, дано определение конкретных областей для дальнейшего исследования”. Настоящая глава частично основана на материалах этой ранней статьи, но саму предлагаемую схему классификации следует рассматривать как устаревшую и замененную пересмотренной версией, которая описана в разделе 9.9 настоящей главы.

**9.13.** Date C.J. Integrity // Глава 11 книги Date C.J. and Colin J.W. A Guide to DB2 (4th edition). — Reading, Mass.: Addison-Wesley, 1993.

В программном продукте DB2 компании IBM действительно была предусмотрена декларативная поддержка первичных и внешних ключей (фактически этот программный продукт был одним из первых, в которых имелась такая поддержка, возможно, даже самым первым). Но, как показано в [9.13], предусмотренная в DB2 поддержка первичных и внешних ключей страдает от определенных ограничений реализации, основным назначением которых является *обеспечение гарантий предсказуемого поведения*. Рассмотрим следующий простой пример. Предположим, что переменная отношения R в настоящее время содержит только два кортежа со значениями первичного ключа, соответственно, 1 и 2, и рассмотрим такой запрос на обновление: “Удвоить значение каждого первичного ключа в R”. Правильным результатом выполнения этого запроса является то, что кортежи должны теперь иметь значения первичного ключа, соответственно, 2 и 4. Если система вначале обновит значение “2” (заменив его на “4”), а затем обновит “1” (заменив его на “2”), то данный запрос будет выполнен успешно. А если, с другой стороны, система обновит (или, скорее, попытается обновить) вначале “1” (заменив его на “2”), то будет обнаружено нарушение ограничения уникальности и запрос окончится неудачей (база данных останется в неизменном состоянии). Иными словами, *результат данного запроса является непредсказуемым*. В целях предотвращения такого

непредсказуемого поведения в DB2 просто запрещаются ситуации, в которых могли бы возникать подобные варианты. Но, к сожалению, некоторые из создаваемых в результате ограничений становятся весьма строгими [9.20].

Следует отметить, что (как показывает приведенные выше пример) в системе DB2 обычно выполняется "оперативная проверка". Это означает, что в этой системе проверки целостности применяются к каждому отдельному кортежу во время обновления данного кортежа. Но такая оперативная проверка является логически неправильной (см. подраздел "Обновление переменных отношения" в разделе 6.5 главы 6); дело в том, что подобный способ проверки принят в основном в целях повышения производительности.

**9.14.** Date C.J. The Primacy of Primary Keys: An Investigation // Relational Database Writings 1991–1994. — Reading, Mass.: Addison-Wesley, 1995.

В статье представлены доводы в пользу такой позиции, что иногда не совсем оправдан такой подход, когда один потенциальный ключ становится "первым среди равных", т.е. первичным.

**9.15.** Date C.J. WHAT Not HOW: The Business Rules Approach to Application Development. — Reading, Mass.: Addison-Wesley, 2000.

Очень неформальное (и не требующее больших усилий в изучении) введение в понятие *бизнес-правил*. См. также [9.21] и [9.22].

**9.16.** Date C.J. Constraints and Predicates: A Brief Tutorial (в трех частях) // `http://www.dbdebunk.com`, (May 2001).

Настоящая глава написана в основном по материалам этого учебника. То же самое можно сказать по поводу приведенной ниже отредактированной (и сокращенной) версии заключительного раздела учебника.

"Как уже было отмечено, база данных представляет собой коллекцию истинных высказываний. Фактически база данных вместе с операциями, которые могут применяться к высказываниям, хранящимся в этой базе данных, представляет собой **логическую систему**. А здесь под термином *логическая система* подразумевается формальная система (которую можно, например, сравнить с эвклидовой геометрией). В ней имеются аксиомы ("высказывания, рассматриваемые как истинные") и правила вывода, с помощью которых мы можем доказывать теоремы ("производные истинные высказывания") на основании этих аксиом. И действительно, идея Кодда (возникшая в период разработки реляционной модели в 1969 году), согласно которой база данных, несмотря на то, что она носит такое название, фактически является не просто коллекцией данных, а скорее коллекцией фактов, или тех выражений смысла, которые в логике называются *истинными высказываниями*, была исключительно важным выводом. Эти высказывания (после того, как они сформулированы, т.е. представлены в базовых переменных отношения) становятся аксиомами рассматриваемой логической системы. А правилами логического вывода по сути являются правила, с помощью которых из существующих высказываний образуются новые; иными словами, они являются правилами, позволяющими нам узнать, как должны использоваться операции реляционной алгебры. Таким образом, при вычислении в системе некоторого реляционного

выражения (в частности, при формировании в ней ответа на некоторый запрос) фактически происходит формирование новых истинных высказываний на основании существующих. В действительности, система при этом доказывает теорему!

Как только мы признаем справедливость изложенного выше представления, то сумеем понять, что для плодотворного решения *проблем базы данных* может использоваться весь аппарат формальной логики. Иными словами, вопросы, подобные перечисленным ниже (не считая вопроса о том, как должны выглядеть ограничения целостности), становятся фактически логическими задачами, которые могут рассматриваться в логической трактовке и поддаются решению в рамках логического подхода.

- Как база данных должна выглядеть с точки зрения пользователя?
- Каким должен быть язык запросов?
- Каким образом результаты должны быть представлены пользователю?
- Как следует наилучшим образом реализовывать запросы (или, если речь идет о более широкой постановке задачи, вычислять выражения базы данных)?
- Какие задачи необходимо решать в первую очередь при проектировании базы данных?

Безусловно, следует также отметить, что реляционная модель непосредственно поддерживает изложенный выше подход к восприятию общей концепции базы данных. Именно по этой причине, по мнению автора, реляционная модель является абсолютно надежной, обоснованной и имеющей неограниченные перспективы дальнейшего развития.

Наконец, зная о том, что база данных вместе с реляционными операциями действительно представляет собой логическую систему, мы можем понять, в чем состоит **столь значительная важность** ограничений целостности. Если в базе данных нарушены какие-то ограничения целостности, то логическая система, о которой мы здесь упоминаем, становится несовместимой с этими ограничениями. А на ответы, полученные от системы, содержащей противоречия, абсолютно невозможно положиться! Предположим, что рассматриваемая система находится в таком состоянии, что в ней одновременно являются истинными выражения `p` и `NOT p` (в этом и заключается несовместимость с ограничениями), где `p` — некоторое высказывание. Теперь допустим, что `q` — другое, произвольное высказывание. Из этого следуют приведенные ниже выводы.

- На основании того, что выражение `p` является истинным, можно заключить, что выражение `p OR q` является истинным.
- На основании того, что выражения `p OR q` и `NOT p` являются истинными, можно заключить, что выражение `q` является истинным.

Но высказывание `q` было взято произвольным образом! Это означает, что в системе, содержащей противоречия, можно доказать истинность любого высказывания".

**9.17.** Hammer M.M., Sarin S.K. Efficient Monitoring of Database Assertions // Proc. 1978 ACM SIGMOD Int. Conf. on Management of Data. — Austin, Texas. — May/June 1978.

В статье приведен набросок алгоритма, предназначенного для выработки процедур проверки целостности, которые являются более эффективными по сравнению с тем очевидным методом простой проверки ограничений после выполнения обновлений, который можно назвать методом с применением "грубой силы". Предлагаемые процедуры проверки встраиваются в объектный код приложения во время компиляции. В некоторых случаях рассматриваемый алгоритм позволяет обнаружить, что проверки на этапе прогона приложения вообще не требуются. Но даже если такие проверки являются необходимыми, часто имеется возможность значительно сократить количество операций доступа к базе данных с помощью различных способов.

**9.18.** Horowitz B. M. A Run-Time Execution Model for Referential Integrity Maintenance // Proc. 8th IEEE Int. Conf. on Data Engineering. — Phoenix, Ariz. — February 1992.

Широко известно, что определенные комбинации перечисленных ниже информационных компонентов при совместном их применении могут приводить к некоторым конфликтным ситуациям и, в принципе, могут вызывать непредсказуемое поведение со стороны приложения (дополнительные сведения можно найти, например, в [9.11]).

1. Ссылочные структуры (т.е. коллекции переменных отношения, между которыми установлены связи с помощью ограничений ссылочной целостности).
2. Правила удаления и обновления внешнего ключа.
3. Фактические значения данных в базе данных.

Существует три основных подхода к решению этой проблемы:

**а)** предоставить пользователю возможность самому справляться с этими проблемами;

**б)** предусмотреть в системе средства обнаружения и исключения попыток определить структуры, применение которых может в принципе привести к конфликтам во время выполнения;

**в)** предусмотреть в системе средства обнаружения и исключения конфликтов, фактически возникающих во время выполнения.

Вариант а) является, безусловно, неприемлемым, а вариант б) требует исключительно тщательной проработки ([9.13], [9.20]), поэтому автор данной статьи Горовиц (Horowitz) предлагает использовать вариант в). В статье предложен набор правил для определения таких действий, выполняемых во время прогона приложения, и доказана их правильность. Но следует отметить, что в этой статье не рассматривается вопрос о том, как повлияет на производительность такая организация проверки во время прогона приложения.

Горовиц активно участвовал в работе комитета, ответственного за определение стандарта SQL:1992, и в тех частях указанного стандарта SQL, которые касаются

вопросов ссылочной целостности, фактически подразумевается, что должны поддерживаться предложения данной статьи.

**9.19.** Markowitz V. M. Referential Integrity Revisited: An Object-Oriented Perspective // Proc. 16th Int. Conf. on Very Large Data Bases. — Brisbane, Australia. — August 1990.

В заголовке этой статьи есть слова "перспективы объектно-ориентированного подхода" ("Object-Oriented Perspective"), полностью соответствующие утверждению, с которого она начинается: "Ограничения ссылочной целостности лежат в основе реляционного представления объектно-ориентированных структур". Но фактически эта статья вообще не посвящена описанию объектов в объектно-ориентированном смысле. Скорее, она является описанием алгоритма, позволяющего составить такое определение реляционной базы данных, начиная с диаграммы "сущность–связь" (см. главу 14), в котором гарантированно не возникают некоторые проблемные ситуации, указанные в [9.11] (например, перекрывающиеся ключи).

Кроме того, в данной статье с точки зрения ссылочной целостности рассматриваются три коммерческих продукта (DB2, Sybase и Ingres, по состоянию примерно на 1990 год). В ней показано, что в DB2, где предусмотрена декларативная поддержка, ограничения являются слишком жесткими. Что касается Sybase и Ingres, в которых предусмотрена процедурная поддержка (соответственно, с помощью *триггеров* и *правил*), то они являются менее ограничительными, чем DB2, но громоздкими и сложными в эксплуатации (хотя и отмечено, что поддержка ограничений в Ingres является "технически более совершенной" по сравнению с Sybase).

**9.20.** Markowitz V. M. Safe Referential Integrity Structures in Relational Databases // Proc. 17th Int. Conf. on Very Large Data Bases. — Barselona, Spain. — September 1991.

Предложены два формальных *условия безопасности*, гарантирующие, что не могут возникнуть некоторые из проблемных ситуаций, описанных, например, в [9.11] и [9.18]. В этой статье также рассматривается вопрос о том, что требуется для обеспечения этих условий в DB2, Sybase и Ingres (это опять-таки касается состояния дел примерно к 1990 году). Применительно к DB2 показано, что некоторые ограничения реализации, налагаемые в целях обеспечения безопасности [9.13], не являются логически необходимыми, и вместе с тем, что другие ограничения являются недостаточными (т.е. DB2 все равно допускает возникновение некоторых небезопасных ситуаций). А что касается Sybase и Ingres, то в статье утверждается, что процедурная поддержка, предусмотренная в этих программных продуктах, не обеспечивает обнаружения небезопасных (или даже неправильных!) спецификаций в ограничениях ссылочной целостности.

**9.21.** Ross R. G. The Business Rule Book: Classifying, Defining, and Modeling Rules (Version 3.0). — Boston, Mass.: Database Research Group, 1994.

См. аннотацию к [9.22].

**9.22.** Ross R.G. Business Rule Concepts. — Houston, Tex.: Business Rule Solutions Inc., 1998.

За последние несколько лет в сообществе пользователей коммерческих СУБД намного вырос интерес к поддержке *бизнес-правил*. Некоторые ключевые фигуры в

промышленности заняли такую позицию, согласно которой бизнес-правила могли бы служить лучшей основой для проектирования и создания баз данных, а также приложений для баз данных (это означает, что они считают бизнес-правила лучшими по сравнению с такими более сложившимися методами, как применение модели "сущность–связь", объектное моделирование, семантическое моделирование и др.). И автор с этим согласен, поскольку бизнес-правила по сути представляют собой не что иное, как более дружественный (т.е. менее сухой и менее формальный) способ рассуждения о предикатах, высказываниях и всех прочих аспектах целостности, которые рассматривались в настоящей главе. Одним из самых выдающихся защитников подхода, основанного на использовании бизнес-правил, является Росс (Ross), и его книги можно смело рекомендовать самым эрудированным практикам. В [9.21] дано всестороннее описание данной темы, а в [9.22] приведено краткое учебное руководство.

***Примечание.*** Ко времени написания настоящей книги была опубликована еще одна книга Росса (Principles of the Business Rule Approach. Addison-Wesley, 2003).

**9.23.** Stonebraker M.R., Wong E. Access Control in a Relational Data Base Management System by Query Modification // Proc. ACM National Conf. — San Diego, Calif. — November 1974.

В прототипе СУБД University Ingres [8.11] был впервые применен интересный подход к определению ограничений целостности (и ограничений защиты — см. главу 17), основанный на одном из вариантов запроса. В этой СУБД для определения ограничений целостности применялся оператор `DEFINE INTEGRITY`, который имеет следующий синтаксис.

```
DEFINE INTEGRITY ON <relvar name> IS <bool exp>
```

Например, один из таких операторов может быть представлен следующим образом.

```
DEFINE INTEGRITY ON S IS S.STATUS > 0
```

Предположим, что пользователь U предпринимает попытку выполнить следующий оператор `REPLACE`, который представлен на языке QUEL.

```
REPLACE S ( STATUS = S.STATUS - 10 ) WHERE S.CITY = "London"
```

В таком случае СУБД Ingres автоматически преобразует оператор `REPLACE` в следующую форму.

```
REPLACE S ( STATUS = S.STATUS - 10 )
WHERE   S.CITY = "London"
AND     ( S.STATUS - 10 ) > 0
```

Очевидно, что вероятность нарушения ограничения целостности при использовании этого модифицированного оператора практически отсутствует.

Один из недостатков указанного подхода состоит в том, что с помощью такого простого способа нельзя обеспечить соблюдение всех ограничений; фактически в языке QUEL поддерживались только такие ограничения, в которых простым условием ограничения было логическое выражение. Однако в то время даже столь

ограниченная поддержка намного превосходила возможности большинства других систем.

**9.24.** Walker A., Salveter S. C. Automatic Modification of Transactions to Preserve Data Base Integrity Without Undoing Updates: Technical Report 81/026. — State University of New York, Stony Brook, N.Y.: Technical Report 81/026. — June 1981.

В этой работе описан метод автоматического преобразования любого *шаблона транзакции* (т.е. исходного кода транзакции) в соответствующий безопасный шаблон. Он является безопасным в том смысле, что ни одна транзакция, соответствующая этому модифицируемому шаблону, по всей вероятности, не может нарушить какие-либо объявленные ограничения целостности. В основе этого метода лежит процедура добавления к первоначальному шаблону запросов и проверок для обеспечения того, чтобы никакие ограничения не были нарушены (еще до выполнения любых операций обновления). Если на этапе прогона приложения любая из заранее предусмотренных проверок оканчивается неудачей, то запрос на выполнение транзакции отклоняется и вырабатывается сообщение об ошибке.

**9.25.** Widom J. and Ceri S. (eds.). Active Database Systems: Triggers and Rules for Advanced Database Processing. — San Francisco, Calif.: Morgan Kaufmann, 1996.

Полезный сборник исследований и руководств по *активным системам баз данных* (так принято называть системы баз данных, которые автоматически выполняют указанные действия в ответ на указанные события, другими словами, системы базы данных с триггерами). В него включены описания нескольких прототипов систем, в том числе СУБД Starburst, разработанной в лаборатории IBM Research (см. [18.21], [18.48], [26.19], [26.23], [26.29] и [26.30]), и СУБД Postgres, которая разработана в Калифорнийском университете, г. Беркли (см. [26.36], [26.40], [26.42] и [26.43]). Кроме того, в этой книге приведены относящиеся к рассматриваемой теме сведения о спецификации SQL:1992, ранней версии спецификации SQL:1999 и о некоторых коммерческих продуктах (в том числе о СУБД Oracle, Informix и Ingres). В книге представлена обширная библиография.