ЗАПРАШИВАНИЕ МНОГОЧИСЛЕНЫХ ТАБЛИЦ ТАКЖЕ КАК ОДНОЙ

ДО ЭТОГО, КАЖДЫЙ ЗАПРОС КОТОРЫЙ МЫ ИССЛЕДОВАЛИ основывался на одиночной таблице.

ОБЪЕДИНЕНИЕ ТАБЛИЦ

Одна из наиболее важных особенностей запросов SQL - это их способность определять связи между многочисленными таблицами и выводить информацию из них в терминах этих связей, всю внутри одной команды. Этот вид операции называется - объединением, которое является одним из видов операций в реляционных базах данных. Используя объединения, мы непосредственно связываем информацию с любым номером таблицы, и таким образом способны создавать связи между сравнимыми фрагментами данных. При объединении, таблицы представленные списком в предложении FROM запроса, отделяются запятыми. Предикат запроса может ссылаться к любому столбцу любой связанной таблицы и, следовательно, может использоваться для связи между ими. Обычно, предикат сравнивает значения в столбцах различных таблиц чтобы определить, удовлетворяет ли WHERE установленному условию.

ИМЕНА ТАБЛИЦ И СТОЛБЦОВ

Полное имя столбца таблицы фактически состоит из имени таблицы, сопровождаемого точкой и затем именем столбца. Имеются несколько примеров имен :

```
Salers.snum
Salers.city
Orders.odate
```

До этого, вы могли опускать имена таблиц потому что вы запрашивали только одну таблицу одновременно, а SQL достаточно интеллектуален чтобы присвоить соответствующий префикс, имени таблицы. Даже когда вы делаете запрос многочисленных таблиц, вы еще можете опускать имена таблиц, если все ее столбцы имеют различные имена. Но это не всегда так бывает. Например, мы имеем две типовые таблицы со столбцами называемыми city.

Если мы должны связать эти столбцы(кратковременно), мы будем должны указать их с именами Salers.city или Customers.city, чтобы SQL мог их различать.

СОЗДАНИЕ ОБЬЕДИНЕНИЯ

Предположим что вы хотите поставить в соответствии вашему продавцу ваших заказчиков в городе в котором они живут, поэтому вы увидите все комбинации продавцов и заказчиков для этого города.

```
SELECT Customers.cname, Salespeople.sname,
 Salespeople.city
   FROM Salespeople, Customers
   WHERE Salespeople.city = Customers.city;
```

Что SQL в основном делает в объединении - так это исследует каждую комбинацию строк двух или более возможных таблиц, и проверяет эти комбинации по их предикатам. В предыдущем примере, требовалась строка продавца Peel из таблицы Продавцов и объединение ее с каждой строкой таблицы Пользователей, по одной в каждый момент времени. Если комбинация производит значение которое делает предикат верным, и если поле city из строк таблиц Заказчика равно London, то Peel - это то запрашиваемое значение которое комбинация выберет для вывода. То же самое будет затем выполнено для каждого продавца в таблице Продавцов (у некоторых из которых не было никаких заказчиков в этих городах).

ОБЪЕДИНЕНИЕ ТАБЛИЦ ЧЕРЕЗ СПРАВОЧНУЮ ЦЕЛОСТНОСТЬ

Эта особенность часто используется просто для эксплуатации связей встроенных в базу данных. В предыдущем примере, мы установили связь между двумя таблицами в объединении. Это прекрасно. Но эти таблицы, уже были соединены через snum поле. Используя объединение можно извлекать данные в терминах этой связи. Например, чтобы показать имена всех заказчиков соответствующих заказов:

```
SELECT Orders.odate, Salespeople.sname
   FROM Orders, Salespeople
   WHERE Salespeople.snum = Orders.snum;
```

ОБЪЕДИНЕНИЯ ТАБЛИЦ ПО РАВЕНСТВУ ЗНАЧЕНИЙ В СТОЛБЦАХ И ДРУГИЕ ВИДЫ ОБЪЕДИНЕНИЙ

Объединения которые используют предикаты основанные на равенствах называются - объединениями по равенству. Все наши примеры до настоящего времени, относились именно к этой категории, потому что все условия в предложениях WHERE базировались на математических выражениях использующих знак равно (=). Строки 'city = 'London' и 'Salespeople.snum = Orders.snum ' - примеры таких типов равенств найденных в предикатах. Объединения по равенству - это вероятно наиболее общий вид объединения, но имеются и другие. Вы можете, фактически, использовать любой из реляционных операторов в обьединении.

```
SELECT sname, cname FROM Salespeople, Customers WHERE sname < cname AND rating < 200;
```

ОБЪЕДИНЕНИЕ БОЛЕЕ ДВУХ ТАБЛИЦ

```
SELECT onum, cname, Orders.cnum, Orders.snum
   FROM Salespeople, Customers,Orders
   WHERE Customers.city < > Salespeople.city
      AND Orders.cnum = Customers.cnum
      AND Orders.snum = Salespeople.snum;
```

ОБЪЕДИНЕНИЕ ТАБЛИЦЫ С СОБОЙ

ПСЕВДОНИМЫ

Синтаксис команды для объединения таблицы с собой, тот же что и для объединения многочисленых таблиц, в одном экземпляре. Когда вы объединяете таблицу с собой, все повторяемые имена столбца, заполняются префиксами имени таблицы. Чтобы ссылаться к этим столбцам внутри запроса, вы должны иметь два различных имени для этой таблицы.

```
SELECT first.cname, second.cname, first.rating FROM Customers first, Customers second WHERE
first.rating = second.rating;
```

УСТРАНЕНИЕ ИЗБЫТОЧНОСТИ

Обратите внимание что наш вывод имеет два значение для каждой комбинации, причем второй раз в обратном порядке. Это потому, что каждое значение показано первый раз в каждом псевдониме, и второй раз(симметрично) в предикате. Следовательно, значение A в псевдониме сначала выбирается в комбинации со значением B во втором псевдониме, а затем значение A во втором псевдониме выбирается в комбинации со значением B в первом псевдониме. В нашем примере, Hoffman выбрался вместе с Clemens, а затем Clemens выбрался вместе с Hoffman. Тот же самый случай с Cisneros и Grass, Liu и Giovanni, и так далее.

Кроме того каждая строка была сравнена сама с собой, чтобы вывести строки такие как - Liu и Liu. Простой способ избежать этого состоит в том, чтобы налагать порядок на два значения, так чтобы один мог быть меньше чем другой или предшествовал ему в алфавитном порядке. Это делает предикат ассиметричным, поэтому те же самые значения в обратном порядке не будут выбраны снова, например:

```
SELECT first.cname, second.cname, first.rating FROM Customers first, Customers
  second WHERE first.rating = second.rating AND first.cname < second.cname;
```

ЕЩЕ БОЛЬШЕ КОМПЛЕКСНЫХ ОБЪЕДИНЕНИЙ

Вы можете использовать любое число псевдонимов для одной таблицы в запросе, хотя использование более двух в данном предложении SELECT * будет излишеством. Предположим что вы еще не назначили ваших заказчиков к вашему продавцу. Компании должна назначить каждому продавцу первоначально трех заказчиков, по одному для каждого рейтингового значения. Вы лично можете решить какого заказчика какому продавцу назначить, но следующий запрос вы используете чтобы увидеть все возможные комбинации заказчиков которых вы можете назначать. (Вывод показывается в Рисунке 9.3):

```
SELECT a.cnum, b.cnum, c.cnum
    FROM Customers a, Customers b, Customers c
    WHERE a.rating = 100
      AND b.rating = 200
      AND c.rating = 300;
```

ВСТАВКА ОДНОГО ЗАПРОСА ВНУТРЬ ДРУГОГО

КАК РАБОТАЕТ ПОДЗАПРОС?

С помощью SQL вы можете вкладывать запросы внутрь друга друга. Обычно, внутренний запрос генерирует значение, которое проверяется в предикате внешнего запроса, определяющего верно оно или нет. Например, предположим что мы знаем имя продавца: Motika, но не знаем значение его поля snum, и хотим извлечь все заказы из таблицы Заказов. Имеется один способ чтобы сделать это:

```
SELECT * FROM Orders WHERE snum = ( SELECT snum FROM Salespeople WHERE sname = 'Motika');
```

вариант со множественным ответом (например, при поиске по названию города) использование DISTINCT

```
SELECT * FROM Orders WHERE ( SELECT DISTINCT snum FROM Orders WHERE cnum =
2001 ) = snum;
```

ИСПОЛЬЗОВАНИЕ АГРЕГАТНЫХ ФУНКЦИЙ В ПОДЗАПРОСАХ

Например, вы хотите увидеть все порядки имеющие сумму приобретений выше средней на 4-е Октября:

```
SELECT * FROM Orders WHERE amt > ( SELECT AVG (amt) FROM Orders WHERE odate =
10/04/2005 );
```

ИСПОЛЬЗОВАНИЕ ПОДЗАПРОСОВ КОТОРЫЕ ВЫДАЮТ МНОГО СТРОК С ПОМОЩЬЮ ОПЕРАТОРА IN

ПОДЗАПРОСЫ ВЫБИРАЮТ ОДИНОЧНЫЕ СТОЛБЦЫ

Смысл всех обсужденных подзапросов в том, что все они выбирают одиночный столбец. Это обязательно, поскольку выбранный вывод сравнивается с одиночным значением. Подтверждением этому то, что SELECT * не может использоваться в

подзапросе. Имеется исключение из этого, когда подзапросы используются с оператором EXISTS

ПОДЗАПРОСЫ В ПРЕДЛОЖЕНИИ HAVING

Вы можете также использовать подзапросы внутри предложения HAVING. Эти подзапросы могут использовать свои собственные агрегатные функции если они не производят многочисленых значений или использовать GROUP BY или HAVING. Следующий запрос является этому примером:

```
SELECT rating, COUNT ( DISTINCT cnum ) FROM Customers GROUP BY rating HAVING
  rating > ( SELECT AVG (rating) FROM Customers WHERE city = " San Jose');
```

СООТНЕСЕННЫЕ ПОДЗАПРОСЫ.

Когда вы используете подзапросы в SQL, вы можете обратиться к внутреннему запросу таблицы в предложении внешнего запроса FROM , сформировав - соотнесенный подзапрос. Когда вы делаете это, подзапрос выполняется неоднократно, по одному разу для каждой строки таблицы основного запроса. Соотнесенный подзапрос - один из большого количества тонких понятий в SQL из-за сложности в его оценке. Если вы сумеете овладеть им, вы найдете что он очень мощный, потому что может выполнять сложные функции с помощью очень лаконичных указаний.

Например, имеется один способ найти всех заказчиков в заказах на 3-е Октября:

```
SELECT * FROM Customers outer WHERE 10/03/1990 IN ( SELECT odate FROM Orders
inner WHERE outer.cnum = inner.cnum );
```

Следовательно, процедура оценки выполняемой соотнесенным подзапросом - это:

1. Выбрать строку из таблицы именованной в внешнем запросе. Это будет текущая строка-кандидат.
2. Сохранить значения из этой строки-кандидата в псевдониме с именем в предложении FROM внешнего запроса.
3. Выполнить подзапрос. Везде, где псевдоним данный для внешнего запроса найден (в этом случае "внешний"), использовать значение для текущей строки-кандидата. Использование значения из строки-кандидата внешнего запроса в подзапросе называется - внешней ссылкой.
4. Оценить предикат внешнего запроса на основе результатов подзапроса выполняемого в шаге 3. Он определяеть - выбирается ли строка-кандидат для вывода.
5. Повторить процедуру для следующей строки-кандидата таблицы, и так далее пока все строки таблицы не будут проверены.

Например, мы можем найти все заказы со значениями сумм приобретений выше среднего для их заказчиков:

```
SELECT * FROM Orders outer WHERE amt > ( SELECT AVG amt FROM Orders inter WHERE
inner.cnum = outer.cnum );
```

КАК РАБОТАЕТ EXISTS?

EXISTS - это оператор, который производит верное или неверное значение, другими словами, выражение Буля. Это означает что он может работать автономно в предикате или в комбинации с другими выражениями Буля использующими Булевы операторы AND, OR, и NOT. Он берет подзапрос как аргумент и оценивает его как верный если тот производит любой вывод или как неверный если тот не делает этого. Этим он отличается от других операторов предиката, в которых1 он не может быть неизвестным. Например, мы можем решить, извлекать ли нам некоторые данные из таблицы Заказчиков если, и только если, один или более заказчиков в этой таблице находятсяся в San Jose:

```
SELECT cnum, cname, city FROM Customers WHERE EXISTS ( SELECT * FROM Customers
WHERE city = " San Jose' );
```

ИСПОЛЬЗОВАНИЕ NOT EXISTS

EXISTS И АГРЕГАТЫ

Одна вещь которую EXISTS не может сделать - взять функцию агрегата в подзапросе. Это имеет значение. Если функция агрегата находит любые строки для операций с ними, EXISTS верен, не взирая на то, что это - значение функции ; если же агрегатная функция не находит никаких строк, EXISTS неправилен.

ИСПОЛЬЗОВАНИЕ ОПЕРАТОРОВ ANY, ALL, И SOME

ANY, ALL, и SOME напоминают EXISTS который воспринимает подзапросы как аргументы; однако они отличаются от EXISTS тем, что используются совместно с реляционными операторами. В этом отношении, они напоминают оператор IN когда тот используется с подзапросами; они берут все значения выведенные подзапросом и обрабатывают их как модуль. Однако, в отличие от IN, они могут использоваться только с подзапросами.

СПЕЦИАЛЬНЫЕ ОПЕРАТОРЫ ANY *или* SOME

Операторы SOME и ANY - взаимозаменяемы везде и там где мы используем ANY, SOME будет работать точно так же. Различие в терминологии состоит в том чтобы позволить людям использовать тот термин который наиболее однозначен. Это может создать проблему; потому что, как мы это увидим, наша интуиция может иногда вводить в заблуждение.

Имеется новый способ нахождения продавца с заказчиками размещенными в их городах:

```
SELECT * FROM Salespeople WHERE city = ANY (SELECT city FROM Customers );
```

Мы можем также использовать оператор IN чтобы создать запрос аналогичный предыдущему :

```
SELECT * FROM Salespeople WHERE city IN ( SELECT city FROM Customers );
```

Однако, оператор ANY может использовать другие реляционные операторы кроме равняется (=), и таким образом делать сравнения которые являются выше возможностей IN. Например, мы могли бы найти всех продавцов с их заказчиками которые следуют им в алфавитном порядке

```
SELECT * FROM Salespeople WHERE sname < ANY ( SELECT cname FROM Customers);
```

КАК ANY МОЖЕТ СТАТЬ НЕОДНОЗНАЧНЫМ

Как подразумевалось выше, ANY не полностью однозначен. Если мы создаем запрос чтобы выбрать заказчиков которые имеют больший рейтинг чем любой заказчик в Риме, мы можем получить вывод который несколько отличался бы от того что мы

ожидали:

```
SELECT * FROM Customers WHERE rating > ANY ( SELECT rating FROM Customers WHERE
city = Rome );
```

СПЕЦИАЛЬНЫЙ ОПЕРАТОР ALL

С помощью ALL, предикат является верным, если каждое значение выбранное подзапросом удовлетворяет условию в предикате внешнего запроса. Если мы хотим пересмотреть наш предыдущий пример чтобы вывести только тех заказчиков чьи оценки, фактически, выше чем у каждого заказчика в Rome, мы можем ввести следующее чтобы произвести вывод показанный в:

```
SELECT * FROM Customers WHERE rating > ALL ( SELECT rating FROM Customers WHERE city =
Rome );
```

КОГДА ПОДЗАПРОС ВОЗВРАЩАЕТСЯ ПУСТЫМ

Одно значительное различие между ALL и ANY - способ действия в ситуации когда подзапрос не возвращает никаких значений. В принципе, всякий раз, когда допустимый подзапрос не в состоянии сделать вывод, ALL - автоматически верен, а ANY автоматически неправилен.

ОБЪЕДИНЕНИЕ МНОГОЧИСЛЕННЫХ ЗАПРОСОВ В ОДИН

Вы можете поместить многочисленые запросы вместе и объединить их вывод используя предложение UNION. Предложение UNION обьединяет вывод двух или более SQL запросов в единый набор строк и столбцов. Например чтобы получить всех продавцов и заказчиков размещенных в Лондоне и вывести их как единое целое вы могли бы ввести:

```
SELECT snum, sname
   FROM Salespeople
   WHERE city = 'London'

   UNION

SELECT cnum, cname
   FROM Customers
```

КОГДА ВЫ МОЖЕТЕ ДЕЛАТЬ ОБЪЕДИНЕНИЕ МЕЖДУ ЗАПРОСАМИ ?

Когда два (или более) запроса подвергаются объединению, их столбцы вывода должны быть совместимы для объединения. Это означает, что каждый запрос должен указывать одинаковое число столбцов и в том же порядке что и первый, второй, третий, и так далее, и каждый должен иметь тип, совместимый с каждым. Значение совместимости типов - меняется.

ANSI следит за этим очень строго и поэтому числовые поля должны иметь одинаковый числовой тип и размер, хотя некоторые имена используемые ANSI для этих типов являются - синонимами. Кроме того, символьные поля должны иметь одинаковое число символов (значение предначначенного номера, не обязательно такое же как используемый номер). Хорошо, что некоторые SQL программы обладают большей гибкостью чем это определяется ANSI. Типы не определенные ANSI, такие как DATA и BINARY, обычно должны совпадать с другими столбцами такого же нестандартного типа.

Длина строки также может стать проблемой. Большинство программ разрешают поля переменной длины, но они не обязательно будут использоваться с UNION. С другой стороны, некоторые программы (и ANSI тоже) требуют чтобы символьные поля

были точно равной длины. В этих вопросах вы должны проконсультироваться с документацией вашей собственной программы.

Другое ограничение на совместимость - это когда пустые значения(NULL) запрещены в любом столбце объединения, причем эти значения необходимо запретить и для всех соответствующих столбцов в других запросах объединения. Кроме того, вы не можете использовать UNION в подзапросах, а также не можете использовать агрегатные функции в предложениии SELECT запроса в объединении. (Большинство программ пренебрегают этими ограничениями.)

UNION будет автоматически исключать дубликаты строк из вывода. Это нечто несвойственное для SQL, так как одиночные запросы обычно содержат DISTINCT чтобы устранять дубликаты.

UNION ALL

ИСПОЛЬЗОВАНИЕ UNION С ORDER BY

До сих пор, мы не оговаривали что данные многочисленых запросов будут выводиться в каком то особом порядке. Мы просто показывали вывод сначала из одного запроса а затем из другого. Конечно, вы не можете полагаться на вывод приходящий в произвольном порядке. Мы как раз сделаем так чтобы этот способ для выполнения примеров был более простым. Вы можете, использовать предложение ORDER BY чтобы упорядочить вывод из объединения, точно так же как это делается в индивидуальных запросах. Давайте пересмотрим наш последний пример чтобы упорядочить имена с помощью их порядковых номеров.

ОБЪЯВЛЕНИЕ ОГРАНИЧЕНИЙ

Вы вставляете ограничение столбца в конец имени столбца после типа данных и перед запятой. Ограничение таблицы помещаются в конец имени таблицы после последнего имени столбца, но перед заключительной круглой скобкой. Далее показан синтаксис для команды CREATE TABLE, расширенной для включения в нее ограничения:

```
CREATE TABLE < table name >
   (< column name > <data type > < column constraint >,
   < column name > < data type > < column constraint > ...
   < table constraint > ( < column name >
   [, < column name > ])... );
```

УНИКАЛЬНОСТЬ КАК ОГРАНИЧЕНИЕ СТОЛБЦА

```
CREATE TABLE Salespeople

 ( Snum     integer NOT NULL UNIQUE,
   Sname   char (10) NOT NULL UNIQUE,
   city      char (10),
   comm   decimal );
```

УНИКАЛЬНОСТЬ КАК ОГРАНИЧЕНИЕ ТАБЛИЦЫ

```
CREATE TABLE Customers

      ( cnum       integer NOT NULL,
        cname     char (10) NOT NULL,
        city        char (10),
        rating     integer,
```

```
    snum      integer NOT NULL,
    UNIQUE  (cnum, snum))
```

ОГРАНИЧЕНИЕ ПЕРВИЧНЫХ КЛЮЧЕЙ

```
CREATE TABLE Salestotal

      ( snum     integer NOT NULL PRIMARY KEY,
        sname    char(10) NOT NULL UNIQUE,
        city       char(10),
        comm     decimal);
```

До этого мы воспринимали первичные ключи исключительно как логические понятия. Хоть мы и знаем что такое первичный ключ, и как он должен использоваться в любой таблице, мы не ведаем "знает" ли об этом SQL.

Поэтому мы использовали ограничение UNIQUE или уникальные индексы в первичных ключах чтобы предписывать им уникальность. В более ранних версиях языка SQL , это было необходимо, и могло выполняться этим способом. Однако теперь, SQL поддерживает первичные ключи непосредственно с ограничением Первичный Ключ (PRIMARE KEY). Это ограничение может быть доступным или недоступным вашей системе.

PRIMARY KEY может ограничивать таблицы или их столбцы. Это ограничение работает так же как и ограничение UNIQUE, за исключением когда только один первичный ключ (для любого числа столбцов) может быть определен для данной таблицы. Имеется также различие между первичными ключами и уникальностью столбцов в способе их использоваться с внешними ключами.

Первичные ключи не могут позволять значений NULL. Это означает что,подобно полям в ограничении UNIQUE, любое поле используемое в ограничении PRIMARY KEY должно уже быть объявлено NOT NULL.

ПЕРВИЧНЫЕ КЛЮЧИ БОЛЕЕ ЧЕМ ОДНОГО ПОЛЯ

Ограничение PRIMARY KEY может также быть применено для многочисленных полей, составляющих уникальную комбинацию значений. Предположим что ваш первичный ключ - это имя, и вы имеете первое имя и последнее имя сохраненными в двух различных полях (так что вы можете организовывать данные с помощью любого из них). Очевидно, что ни первое ни последнее имя нельзя заставить быть уникальным самостоятельно, но мы можем каждую из этих двух комбинаций сделать уникальной.

```
CREATE TABLE  Namefield

      ( firstname         char (10) NOT NULL,
        lastname         char (10) NOT NULL
        city               char (10),
        PRIMARY KEY  ( firstname, lastname ));
```

ПРОВЕРКА ЗНАЧЕНИЙ ПОЛЕЙ

Конечно, имеется любое число ограничений которые можно устанавливать для данных вводимых в ваши таблицы, чтобы видеть, например, находятся ли данные в соответствующем диапазоне или правильном формате, о чем SQL естественно не может знать заранее. По этой причине, SQL обеспечивает вас ограничением CHECK, которое позволяет вам установить условие которому должно удовлетворять значение вводимое в таблицу, прежде чем оно будет принято. Ограничение CHECK состоит из

ключевого слова CHECK сопровождаемого предложением предиката, который использует указанное поле. Любая попытка модифицировать или вставить значение поля которое могло бы сделать этот предикат неверным - будет отклонена.

Давайте рассмотрим еще раз таблицу Продавцов. Столбец комиссионных выражается десятичным числом и поэтому может быть умножен непосредственно на сумму приобретений в результате чего будет получена сумма комиссионных(в долларах) продавца. Кто-то может использовать понятие процента, однако ведь, можно об этом и не знать. Если человек введет по ошибке 14 вместо .14 чтобы указать в процентах свои комиссионные, это будет расценено как 14.0 , что является законным десятичным значением, и будет нормально воспринято системой. Чтобы предотвратить эту ошибку, мы можем наложить ограничение столбца - CHECK чтобы убедиться что вводимое значение меньше чем 1.

```
CREATE TABLE Salespeople
   ( snum     integer NOT NULL PRIMARY KEY,
     sname    char(10) NOT NULL UNIQUE,
     city       char(10),
     comm     decimal CHECK ( comm < 1 ));
```

ИСПОЛЬЗОВАНИЕ - CHECK, ЧТОБЫ ПРЕДОПРЕДЕЛЯТЬ ДОПУСТИМОЕ ВВОДИМОЕ ЗНАЧЕНИЕ

```
CREATE TABLE Salespeople (snum     integer NOT NULL UNIQUE, sname    char(10)
NOT NULL UNIQUE, city     char(10) CHECK, (city IN ('London', 'New York', 'San
Jose', 'Barselona')), comm     decimal CHECK (comm < 1 ));
```

ПРОВЕРКА УСЛОВИЙ БАЗИРУЮЩИЙСЯ НА МНОГОЧИСЛЕНЫХ ПОЛЯХ

Вы можете также использовать CHECK в качестве табличного ограничения. Это полезно в тех случаях когда вы хотите включить более одного поля строки в условие. Предположим что комиссионные .15 и выше, будут разрешены только для продавца из Барселоны. Вы можете указать это со следующим табличным ограничением CHECK:

```
 CREATE TABLE Salespeople
     ( snum        integer NOT NULL UNIQUE,
       sname      char (10) NOT NULL UNIQUE,
       city         char(10),
       comm       decimal,
       CHECK    (comm < .15 OR city = 'Barcelona'));
```

УСТАНОВКА ЗНАЧЕНИЙ ПО УМОЛЧАНИЮ

```
CREATE TABLE Salespeople

         ( snum     integer NOT NULL UNIQUE,
           sname    char(10) NOT NULL UNIQUE,
           city       char(10) DEFAULT = 'New York',
           comm     decimal CHECK (comm < 1);
```

ПОДДЕРЖКА ЦЕЛОСТНОСТИ ВАШИХ ДАННЫХ

ВНЕШНИЙ КЛЮЧ И РОДИТЕЛЬСКИЙ КЛЮЧ

Когда все значения в одном поле таблицы представлены в поле другой таблицы, мы

говорим что первое поле ссылается на второе. Это указывает на прямую связь между значениями двух полей. Например, каждый из заказчиков в таблице Заказчиков имеет поле snum которое указывает на продавца назначенного в таблице Продавцов. Для каждого порядка в таблице Заказов, имеется один и только этот продавец и один и только этот заказчик. Это отображается с помощью полей snum и cnum в таблице Заказовв.

Когда одно поле в таблице ссылается на другое, оно называется - внешним ключом; а поле на которое оно ссылается, называется - родительским ключом. Так что поле snum таблицы Заказчиков - это внешний ключ, а поле snum на которое оно ссылается в таблице Продавцов - это родительский ключ.

ОГРАНИЧЕНИЕ FOREIGN KEY

Эта функция должна ограничивать значения которые вы можете ввести в вашу базу данных чтобы заставить внешний ключ и родительский ключ соответствовать принципу справочной целостности. Одно из действий ограничения Внешнего Ключа - это отбрасывание значений для полей ограниченных как внешний ключ который еще не представлен в родительском ключе. Это ограничение также воздействует на вашу способность изменять или удалять значения родительского ключа.

```
FOREIGN KEY <column list> REFERENCES <pktable> [ <column list> ]

CREATE TABLE Customers

            ( cnum   integer NOT NULL PRIMARY KEY
              cname  char(10),
              city   char(10),
              snum   integer,
              FOREIGN KEY (snum) REFERENCES Salespeople
              ( snum );

CREATE TABLE Customers

          ( cnum   integer NOT NULL PRIMARY KEY,
            cname  char(10),
            city    char(10),
            snum   integer REFERENCES Salespeople);

// если это явно PRIMARY KEY в Salespeople
```

ДЕЙСТВИЕ ОГРАНИЧЕНИЙ

Как такие ограничения воздействуют на возможность и невозможность Вами использовать команды модификации DML? Для полей, определенных как внешние ключи, ответ довольно простой: любые значения которые вы помещаете в эти поля с командой INSERT или UPDATE должны уже быть представлены в их родительских кючах. Вы можете помещать пустые(NULL) значения в эти поля, несмотря на то что значения NULL не позволительны в родительских ключах, если они имеют ограничение NOT NULL. Вы можете удалять (DELETE) любые строки с внешними ключами не используя родительские ключи вообще. Поскольку затронут вопрос об изменении значений родительского ключа, ответ, по определению ANSI, еще проще, но возможно несколько более ограничен: любое значение родительского ключа ссылаемого с помощью значения внешнего ключа, не может быть удалено или изменено.

Имеются некоторые другие возможные действия изменения родительского ключа, которые не являются частью ANSI, но могут быть найдены в некоторых коммерческих программах. Если вы хотите изменить или удалить текущее ссылочное значение родительского ключа, имеется по существу три возможности:

* Вы можете ограничить, или запретить, изменение (способом ANSI), обозначив, что изменения в родительском ключе - ограничены.
* Вы можете сделать изменение в родительском ключе и тем самым сделать изменения во внешнем ключе автоматическим, что называется - каскадным изменением.
* Вы можете сделать изменение в родительском ключе, и установить внешний ключ в NULL, автоматически (полагая, что NULLS разрешен во внешнем ключе), что называется - пустым изменением внешнего ключа.

Даже в пределах этих трех категорий, вы можете не захотеть обрабатывать все команды модификации таким способом. INSERT, конечно, к делу не относится. Он помещает новые значения родительского ключа в таблицу, так что ни одно из этих значений не может быть вызвано в данный момент. Однако, вы можете захотеть позволить модификациям быть каскадными, но без удалений, и наоборот. Лучшей может быть ситуация которая позволит вам определять любую из трех категорий, независимо от команд UPDATE и DELETE. Мы будем следовательно ссылаться на эффект модификации (update effects) и эффект удаления (delete effects), которые определяют, что случится если вы выполните команды UPDATE или DELETE в родительском ключе. Эти эффекты, о которых мы говорили, называются:

Ограниченные (RESTRICTED) изменения, Каскадируемые (CASCADES) изменения, и Пустые (NULL) изменения.

Фактические возможности вашей системы должны быть в строгом стандарте ANSI - это эффекты модификации и удаления, оба, автоматически ограниченнные - для более идеальной ситуации описаной выше. В качестве иллюстрации, мы покажем несколько примеров того, что вы можете делать с полным набором эффектов модификации и удаления. Конечно, эффекты модификации и удаления, являющиеся нестандартными средствами, испытывают недостаток в стандартном госинтаксисе. Синтаксис который мы используем здесь, прост в написании и будет служить в дальнейшем для иллюстрации функций этих эффектов.

```
CREATE TABLE Customers
  (cnum integer NOT NULL PRIMARY KEY,
   cname char(10) NOT NULL,
   city  char(10),
   rating integer,
   snum   integer REFERENCES Salespeople,
   UPDATE OF Salespeople CASCADES,
   DELETE OF Salespeople RESTRICTED);
```

Третий эффект - Пустые (NULL) изменения. Бывает, что когда продавцы оставляют

компанию, их текущие порядки не передаются другому продавцу.

```
CREATE TABLE Orders
   (onum integer NOT NULL PRIMARY KEY,
    amt decimal,
    odate date NOT NULL
    cnum integer NOT NULL REFERENCES Customers
    snum integer REFERENCES Salespeople,
    UPDATE OF Customers CASCADES,
    DELETE OF Customers CASCADES,
    UPDATE OF Salespeople CASCADES,
    DELETE OF Salespeople NULLS);
```

Конечно, в команде DELETE с эффектом Пустого изменения в таблице Продавцов, ограничение NOT NULL должно быть удалено из поля snum.

ПРЕДСТАВЛЕНИЕ (VIEW) - ОБЪЕКТ ДАННЫХ КОТОРЫЙ не содержит никаких данных его владельца. Это - тип таблицы, чье содержание выбирается из других таблиц с помощью выполнения запроса. Поскольку значения в этих таблицах меняются, то автоматически, их значения могут быть показаны представлением.

ЧТО ТАКОЕ ПРЕДСТАВЛЕНИЕ ?

Типы таблиц, с которыми вы имели дело до сих пор, назывались - базовыми таблицами. Это - таблицы, которые содержат данные. Однако имеется другой вид таблиц: - представления. *Представления - это таблицы чье содержание выбирается или получается из других таблиц.* Они работают в запросах и операторах DML точно также как и основные таблицы, но не содержат никаких собственных данных. Представления - подобны окнам, через которые вы просматриваете информацию(как она есть, или в другой форме, как вы потом увидите), которая фактически хранится в базовой таблице. Представление - это фактически запрос, который выполняется всякий раз, когда представление становится темой команды. Вывод запроса при этом в каждый момент становится содержанием представления.

КОМАНДА CREATE VIEW

Вы создаете представление командой CREATE VIEW. Она состоит из слов CREATE VIEW (СОЗДАТЬ ПРЕДСТАВЛЕНИЕ), имени представления которое нужно создать, слова AS (КАК), и далее запроса, как в следующем примере:

```
CREATE VIEW Londonstaff          AS SELECT *
   FROM Salespeople           WHERE city = 'London';
```

Представления значительно расширяют управление вашими данными. Это превосходный способ дать публичный доступ к некоторой, но не всей информации в таблице.

МОДИФИЦИРОВАНИЕ ПРЕДСТАВЛЕНИЙ

Представление может теперь изменяться командами модификации DML, но модификация не будет воздействовать на само представление. Команды будут на самом деле перенаправлены к базовой таблице:

```
UPDATE Salesown  SET city = 'Palo Alto' WHERE snum = 1004;
```

Его действие идентично выполнению той же команды в таблице Продавцов. Однако, если значение комиссионных продавца будет обработано командой UPDATE

```
UPDATE Salesown SET comm = .20  WHERE snum = 1004;
```

она будет отвергнута, так как поле comm отсутствует в представлении Salesown. Это важное замечание, показывающее что не все представления могут быть модифицированы.

КОМБИНИРОВАНИЕ ПРЕДИКАТОВ ПРЕДСТАВЛЕНИЙ И ОСНОВНЫХ ЗАПРОСОВ В ПРЕДСТАВЛЕНИЯХ

```
CREATE VIEW Ratingcount (rating, number)
          AS SELECT rating, COUNT (*)
          FROM Customers
          GROUP BY rating;
```

Это дает нам число заказчиков которые мы имеем для каждого уровня оценки(rating). Вы можете затем сделать запрос этого представления чтобы выяснить, имеется ли какая-нибудь оценка, в настоящее время назначенная для трех заказчиков:

```
       SELECT *
          FROM Ratingcount
          WHERE number = 3;
```

Посмотрим что случится если мы скомбинируем два предиката:

```
       SELECT rating, COUNT (*)
          FROM Customers
          WHERE COUNT (*) = 3
          GROUP BY rating;
```

ГРУППОВЫЕ ПРЕДСТАВЛЕНИЯ

Групповные представления - это представления, наподобии запроса Ratingcount в предыдущем примере, который содержит предложение GROUP BY, или который основывается на других групповных представлениях.

Групповые представления могут стать превосходным способом обрабатывать полученную информацию непрерывно. Предположим, что каждый день вы должны следить за порядком номеров заказчиков, номерами продавцов принимающих заказы, номерами заказов, средним от заказов, и общей суммой приобретений в порядках.

Чем конструировать каждый раз сложный запрос, вы можете просто создать следующее представление:

```
   CREATE VIEW Totalforday
        AS SELECT odate, COUNT (DISTINCT cnum), COUNT
                (DISTINCT snum), COUNT (onum), AVG
                (amt), SUM (amt)
                FROM Orders
        GROUP BY odate;
```

Теперь вы сможете увидеть всю эту информацию с помощью простого запроса:

```
        SELECT *
           FROM Totalforday;
```

ПРЕДСТАВЛЕНИЯ И ОБЬЕДИНЕНИЯ

Представления не требуют чтобы их вывод осуществлялся из одной базовой таблицы. Так как почти любой допустимый запрос SQL может быть использован в представлении, он может выводить информацию из любого числа базовых таблиц, или из других представлений.

ПРЕДСТАВЛЕНИЯ И ПОДЗАПРОСЫ

Представления могут также использовать и подзапросы, включая соотнесенные подзапросы.

ЧТО НЕ МОГУТ ДЕЛАТЬ ПРЕДСТАВЛЕНИЯ

Имеются большое количество типов представлений, которые являются доступными только для чтения. Это означает, что их можно запрашивать, но они не могут подвергаться действиям команд модификации.

Имеются также некоторые виды запросов, которые не допустимы в определениях представлений. Одиночное представление должно основываться на одиночном запросе; ОБЪЕДИНЕНИЕ (UNION) и ОБЪЕДИНЕНИЕ ВСЕГО (UNIOM ALL) не разрешаются.

УПОРЯДОЧЕНИЕ ПО(ORDER BY) никогда не используется в определении представлений. Вывод запроса формирует содержание представления, которое напоминает базовую таблицу и является - по определению - неупорядоченным.

УДАЛЕНИЕ ПРЕДСТАВЛЕНИЙ

Синтаксис удаления представления из базы данных подобен синтаксису удаления базовых таблиц:

```
DROP VIEW < view name >
```

ИЗМЕНЕНИЕ ЗНАЧЕНИЙ С ПОМОЩЬЮ ПРЕДСТАВЛЕНИЙ

Один из наиболее трудных и неоднозначных аспектов представлений - непосредственное их использование с командами модификации DML.

```
CREATE VIEW Citymatch (custcity, salescity) AS SELECT DISTINCT a.city, b.city
FROM Customers a, Salespeople b WHERE a.snum = b.snum;
```

ОПРЕДЕЛЕНИЕ МОДИФИЦИРУЕМОСТИ ПРЕДСТАВЛЕНИЯ

Если команды модификации могут выполняться в представлении, представление как сообщалось будет модифицируемым; в противном случае оно предназначено только для чтения при запросе. Непротивореча этой терминологии, мы будем использовать выражение "модифицируемое представление"(updating a view), чтобы означает возможность выполнения в представление любой из трех команд модификации DML (Вставить, Изменить и Удалить), которые могут изменять значения. Как вы определите, является ли представление модифицируемым? В теории базы данных, это – пока обсуждаемая тема. *Основной ее принцип такой: модифицирумое представление – это представление в котором команда модификации может выполниться, чтобы изменить одну и только одну строку основной таблицы в каждый момент времени, не воздействуя на любые другие строки любой таблицы.*

Использование этого принципа на практике, однако, затруднено. Кроме того, некоторые представления, которые являются модифицируемыми в теории, на самом деле не являются модифицируемыми в SQL. Критерии по которые определяют, является ли представление модифицируемым или нет, в SQL, следующие:

* Оно должно выводиться в одну и только в одну базовую таблицу.
* Оно должно содержать первичный ключ этой таблицы (это технически не предписывается стандартом ANSI, но было бы неплохо придерживаться этому).
* Оно не должно иметь никаких полей, которые бы являлись агрегатными функциями.
* Оно не должно содержать DISTINCT в своем определении.
* Оно не должно использовать GROUP BY или HAVING в своем определении.
* Оно не должно использовать подзапросы (это - ANSI_ограничение которое не предписано для некоторых реализаций)
* Оно может быть использовано в другом представлении, но это представление

должно также быть модифицируемыми.
* Оно не должно использовать константы, строки, или выражения значений (например: comm * 100) среди выбранных полей вывода.
* Для INSERT, оно должно содержать любые поля основной таблицы которые имеют ограничение NOT NULL, если другое ограничение по умолчанию, не определено.

МОДИФИЦИРУЕМЫЕ ПРЕДСТАВЛЕНИЯ И ПРЕДСТАВЛЕНИЯ ТОЛЬКО_ЧТЕНИЕ.

Одно из этих ограничений то, что модифицируемые представления, фактически, подобны окнам в базовых таблицах. Они показывают кое-что, но не обязательно все, из содержимого таблицы. Они могут ограничивать определенные строки (использованием предикатов), или специально именованные столбцы (с исключениями), но они представляют значения непосредственно и не выводит их информацию, с использованием составных функций и выражений.

Они также не сравнивают строки таблиц друг с другом (как в обьединениях и подзапросах, или как с DISTINCT).

Различия между модифицируемыми представлениями и представлениями только_чтение неслучайны.

Цели для которых вы их используете, часто различны. Модифицируемые представления, в основном, используются точно так же как и базовые таблицы. Фактически, пользователи не могут даже осознать, является ли объект который они запрашивают, базовой таблицей или представлением.
Это превосходный механизм защиты для сокрытия частей таблицы, которые являются конфиденциальными или не относятся к потребностям данного пользователя.

Представления только_чтение, с другой стороны, позволяют вам получать и переформатировать данные более рационально. Они дают вам библиотеку сложных запросов, которые вы можете выполнить и повторить снова, сохраняя полученную вами информацию до последней минуты. Кроме того, результаты этих запросов в таблицах, которые могут затем использоваться в запросах самостоятельно (например, в обьединениях) имеют преимущество над просто выполнением запросов.

Представления только_чтение могут также иметь прикладные программы защиты. Например, вы можете захотеть, чтобы некоторые пользователи видели агрегатные данные, такие как усредненное значение комиссионных продавца, не видя индивидуальных значений комиссионных.

ЧТО ЯВЛЯЕТСЯ - МОДИФИЦИРУЕМЫМИ ПРЕДСТАВЛЕНИЕМ

Имеются некоторые примеры модифицируемых представлений и представлений только_чтение:

```
CREATE VIEW Dateorders (odate, ocount)
   AS SELECT odate, COUNT (*)
      FROM Orders
      GROUP BY odate;
```

Это - представление только_чтение из-за присутствия в нем агрегатной функции и GROUP BY.

```
CREATE VIEW Londoncust
   AS SELECT *
      FROM Customers
      WHERE city = 'London';
```

А это - представление модифицируемое.

```
CREATE VIEW SJsales (name, number, percentage)
   AS SELECT sname, snum, comm * 100
      FROM Salespeople
      WHERE city = 'SanJose';
```

Это - представление только_чтение из-за выражения " comm * 100 " . При этом, однако, возможны переупорядочение и переименование полей.

```
CREATE VIEW Someorders AS SELECT snum, onum, cnum FROM Orders WHERE odate IN
(10/03/1990,10/05/1990);
```

Это - модифицируемое представление.

ПРОВЕРКА ЗНАЧЕНИЙ ПОМЕЩАЕМЫХ В ПРЕДСТАВЛЕНИЕ

Другой вывод о модифицируемости представления тот, что вы можете вводить значения которые " проглатываются " (swallowed) в базовой таблице. Рассмотрим такое представление:

```
CREATE VIEW Highratings
   AS SELECT cnum, rating
      FROM Customers
      WHERE rating = 300;
```

Это - представление модифицируемое. Оно просто ограничивает ваш доступ к определенным строкам и столбцам в таблице. Предположим, что вы вставляете (INSERT) следующую строку:

```
INSERT INTO Highratings
   VALUES (2018, 200);
```

Это - допустимая команда INSERT в этом представлении. Строка будет вставлена, с помощью представления Highratings, в таблицу Заказчиков. Однако когда она появится там, она исчезнет из представления, поскольку значение оценки не равно 300. Это - обычная проблема.

Значение 200 может быть просто напечатано, но теперь строка находится уже в таблице Заказчиков где вы не можете даже увидеть ее. Пользователь не сможет понять, почему введя строку он не может ее увидеть, и будет неспособен при этом удалить ее. Вы можете быть гарантированы от модификаций такого типа с помощью включения WITH CHECK OPTION (С ОПЦИЕЙ ПРОВЕРКИ) в определение представления. Мы можем использовать WITH CHECK OPTION в определении представления Highratmgs.

```
CREATE VIEW Highratings AS SELECT cnum, rating FROM Customers WHERE rating = 300
WITH CHECK OPTION;
```

Вышеупомянутая вставка будет отклонена.

WITH CHECK OPTION - производит действие все_или_ничего (all-or-nothing). Вы помещаете его в определение представления, а не в команду DML, так что или все команды модификации в представлении будут проверяться, или ни одна не будет проверена. Обычно вы хотите использовать опцию проверки, используя ее в определении представления, что может быть удобно.

ПРЕДИКАТЫ И ИСКЛЮЧЕННЫЕ ПОЛЯ

Похожая проблема, которую вы должны знать, включает в себя вставку строк в представление с предикатом, базирующемся на одном или более исключенных полей. Например, может показаться разумным, чтобы создать Londonstaff подобно этому:

```
CREATE VIEW Londonsta1t AS SELECT snum, sname, comm FROM Salespeople WHERE city
= 'London';
```

КТО ЧТО МОЖЕТ ДЕЛАТЬ В БАЗЕ ДАННЫХ

ПОЛЬЗОВАТЕЛИ

Каждый пользователь в среде SQL , имеет специальное идентификационное имя или номер. Терминология везде разная, но мы выбрали (следуя ANSI) ссылку на имя или номер как на Идентификатор (ID) доступа. Команда, посланная в базе данных ассоциируется с определенным пользователем; или иначе, специальным Идентификатором доступа. Посколько это относится к SQL базе данных, ID разрешения - это имя пользователя, и SQL может использовать специальное ключевое слово USER, которое ссылается к Идентификатору доступа связанному с текущей командой. Команда интерпретируется и разрешается (или запрещается) на основе информации связанной с Идентификатором доступа пользователя подавшего команду.

РЕГИСТРАЦИЯ

В системах с многочисленными пользователями, имеется некоторый вид процедуры входа в систему, которую пользователь должен выполнить чтобы получить доступ к компьютерной системе. Эта процедура определяет какой ID доступа будет связан с текущим пользователем. Обычно, каждый человек использующий базу данных должен иметь свой собственный ID доступа и при регистрации превращается в действительного пользователям. Однако, часто пользователи имеющие много задач могут регистрироваться под различными ID доступа, или наоборот один ID доступа может использоваться несколькими пользователями.

С точки зрения SQL нет никакой разницы между этими двумя случаями, он воспринимает пользователя просто как его ID доступа.

SQL база данных может использовать собственную процедуру входа в систему, или она может позволить другой программе, типа операционной системы (основная программа которая работает на вашем компьютере), обрабатывать файл регистрации и получать ID доступа из этой программы. Тем или другим способом, но SQL будет иметь ID доступа чтобы связать его с вашими действиями, а для вас будет иметь значение ключевое слово USER.

ПРЕДОСТАВЛЕНИЕ ПРИВИЛЕГИЙ

Каждый пользователь в SQL базе данных имеет набор привилегий. Это то, что пользователю разрешается делать (возможно это - файл регистрации, который может рассматриваться как минимальная привилегия). Эти привилегии могут изменяться со временем - новые добавляться, старые удаляться. Некоторые из этих привилегий определены в ANSI SQL, но имеются и дополнительные привилегии, которые являются также необходимыми. SQL привилегии как определено ANSI, не достаточны в большинстве ситуаций реальной жизни. С другой стороны, типы привилегий, которые необходимы, могут видоизменяться с видом системы которую вы используете - относительно которой ANSI не может дать никаких рекомендаций. Привилегии которые не являются частью стандарта SQL могут использовать похожий синтаксис и не полностью совпадающий со стандартом.

СТАНДАРТНЫЕ ПРИВИЛЕГИИ

SQL привилегии определенные ANSI - это привелегии объекта. Это означает что пользователь имеет привилегию чтобы выполнить данную команду только на определенном объекте в базе данных. Очевидно, что привилегии должны различать эти объекты, но система привилегии основанная исключительно на привилегиях объекта не может адресовывать все что нужно SQL, как мы увидем это позже в этой главе. Привилегии объекта связаны
одновременно и с пользователями и с таблицами. То есть, привилегия дается определенному пользователю в указаной таблице, или базовой таблице или представлении. Вы должны помнить, что пользователь создавший таблицу (любого вида), является владельцем этой таблицы.

Это означает, что пользователь имеет все привилегии в этой таблице и может передавать привелегии другим пользователям в этой таблице. Привилегии которые можно назначить пользователю:

SELECT Пользователь с этой привилегией может выполнять запросы в таблице.
INSERT Пользователь с этой привилегией может выполнять команду INSERT в таблице.
UPDATE Пользователь с этой привилегией может выполнять команду UPDATE на таблице. Вы можете ограничить эту привилегию для определенных столбцов таблицы.
DELETE Пользователь с этой привилегией может выполнять команду DELETE в таблице.
REFERENCES Пользователь с этой привилегией может определить внешний ключ, который использует один или более столбцов этой таблицы, как родительский ключ. Вы можете ограничить эту привилегию для определенных столбцов.

Кроме того, вы столкнетесь с нестандартными привилегиями объекта, такими например как INDEX (ИНДЕКС) дающим право создавать индекс в таблице, SYNONYM (СИНОНИМ) дающим право создавать синоним для объекта, который будет объяснен в Главе 23, и ALTER (ИЗМЕНИТЬ) дающим право выполнять команду ALTER TABLE в таблице. Механизм SQL назначает пользователям эти привилегии с помощью команды GRANT.

КОМАНДА GRANT

Позвольте предположить, что пользователь Diane имеет таблицу Заказчиков и хочет позволить пользователю Adrian выполнить запрос к ней. Diane должна в этом случае ввести следующую команду:

```
GRANT INSERT ON Salespeople TO Diane;
```

Теперь Adrian может выполнить запросы к таблице Заказчиков. Без других привилегий, он может только выбрать значения; но не может выполнить любое действие, которые бы воздействовало на значения в таблице Заказчиков (включая использование таблицы Заказчиков в качестве родительской таблицы внешнего ключа, что ограничивает изменения которые выполнять со значениям в таблице Заказчиков). Когда SQL получает команду GRANT, он проверяет привилегии пользователя подавшего эту команду, чтобы определить допустима ли команда GRANT.

Adrian самостоятельно не может выдать эту команду. Он также не может предоставить право SELECT другому пользователю: таблица еще принадлежит Diane (позже мы покажем как Diane может дать право Adrian предоставлять SELECT другим пользователям). Синтаксис - тот же самый, что и для предоставления других привилегий. Если Adrian - владелец таблицы Продавцов, то он может позволить Diane вводить в нее строки с помощью следующего предложения

```
GRANT INSERT ON Salespeople TO Diane;
```

Теперь Diane имеет право помещать нового продавца в таблицу.