# 2

# Функции: концепция и структура

Функции являются краеугольным камнем любого проекта по созданию СУРБД. Чтобы осознать всю мощь и гибкость, которую предоставляет использование функций в вашей среде, сначала необходимо разобраться с базовыми понятиями. В этой главе мы рассмотрим некоторые основные абстрактные и архитектурные возможности функций. Вначале изучим, что подразумевается под определением "функция", затем узнаем, как функции создаются, компилируются, выполняются и систематизируются. По ходу изложения будет приведено несколько примеров, иллюстрирующих использование функций в среде программирования SQL.

## Понятие функции

*Функцией (function)* называют законченный фрагмент программного кода, к выполнению которого может перейти программа по окончании выполнения другого фрагмента либо в результате ветвления при наступлении определенных условий. По окончании работы функции программа возвращается к тому месту в программном коде, которое следует непосредственно за точкой ветвления. В общем случае функция принимает один или несколько аргументов и возвращает единственное значение. Исследуем, каким образом это происходит.

Каждый раз при запуске программы для ее работы выделяется структура данных, называемая *стеком программы (program stack)*. Стек представляет собой область памяти, доступ к которой осуществляется с помощью "верхушки", откуда объекты могут помещаться в стек (т.е. записываться в память) либо "извлекаться" из него (т.е. удаляться из памяти). Такую дисциплину работы с объектами называют дисциплиной "последний пришел – первый ушел" (Last in, First out – LIFO). Подобная дисциплина находит применение в процессах с рекурсивным возвращением (например, при прохождении древовидной структуры).

При вызове функции (т.е. когда выполнение программы переходит к заданной функции) в стек записывается адрес инструкции, следующей за точкой ветвления.

Таким образом, функции становится известно, куда передать управление по завершении работы. По окончании работы функции из стека извлекается адрес следующей инструкции в теле основной программы. Затем этот адрес помещается в указатель инструкции (регистр процессора, содержащий адрес следующей инструкции для выполнения).

Как удостовериться в том, что после завершения работы функции в начале стека будет храниться адрес следующей инструкции? Современные языки программирования высокого уровня, как правило, обеспечивают корректность работы со стеком. Одной из главных проблем, связанных с использованием низкоуровневых языков программирования, например ассемблера, являлась "несбалансированность стека", возникающая в том случае, если программист не согласовал все операции записи в стек с соответствующими операциями чтения из стека в правильном порядке. В такой ситуации функция просто не знала, куда следует возвращать управление.

Перед вызовом функции вызывающая программа должна поместить в стек ее аргументы. В принципе порядок, в котором будут выполняться эти операции, не имеет значения, поскольку извлечение информации происходит в обратном порядке (в соответствии с дисциплиной LIFO). Однако, поскольку аргументы могут записываться в стек в разной последовательности (если аргументов больше одного), каждый компилятор делает это по-своему. К счастью, предусмотрено всего две базовые нотации: когда последний аргумент записывается в стек первым или же когда первый аргумент записывается в стек первым. Непонятно, почему никто не придумал помещать средний аргумент в начале либо предпоследним в конце. Эти две вышеупомянутые нотации обычно называют передачей аргументов в стиле C или Паскаль (C- или Паскаль-нотация), поскольку в языке C последний аргумент передается первым.

Чтобы быть уверенным в совместимости, необходимо знать, какая нотация используется в вашем компиляторе SQL и с каким языком работает данный компилятор. Главное, чтобы оба компилятора придерживались одного и того же порядка передачи аргументов.

Если система управления базами данных (СУБД) предлагает компилятору самому вести запись функций, вы можете быть уверены, что компилятор использует правильный порядок передачи аргументов. В то же время, если СУБД предлагает свой способ добавления *скомпилированной* функции, вам следует внимательно изучить имеющуюся информацию, чтобы избежать возникновения потенциальных проблем. Обычно в компиляторах предусмотрены переключатели либо иные средства переопределения порядка передачи аргументов по умолчанию.

Разумеется, на различных платформах функции могут быть структурированы поразному. Далее, чтобы проиллюстрировать различия между функциями операционной системы и функциями, созданными в конкретной реализации СУРБД, приведем несколько небольших примеров.

## Пример простой функции оболочки Unix

Показательным примером простой функции оболочки Unix является функция преобразования строки в верхний регистр. Приведенная ниже функция `ToUpper()` принимает в качестве параметра строковую переменную, возвращая в результате ту же строку, но уже преобразованную в верхний регистр.

```
ToUpper()
{
    echo $1 | tr 'abcdefghijklmnopqrstuvwxyz' \
              'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
}
```

В Unix для передачи аргумента используется синтаксис `$x`, где *x* представляет собой числовое значение от 0 до 9, обозначающее порядковый номер параметра, передаваемого функции. В нашем примере видно, что `$1` используется для передачи в первом параметре команды `tr`, выполняющей простое преобразование текста в верхний регистр. Выполнив эту функцию в Unix, вы получите следующий результат:

```
$ ToUpper david
  DAVID
```

## Пример простой функции SQL

Понятие функции в SQL схоже с понятием функции в оболочке Unix, хотя их синтаксис при этом сильно отличается. Создадим в следующем примере простую SQL-функцию Oracle под названием `Get_Tax`. Функция `Get_Tax` используется для расчета суммы налога с оборота для заданного объема наличности при существующей (в США) ставке в 5%.

```
Create or replace function Get_Tax (aAmount IN NUMBER(10,2))
   Return NUMBER
   Is
   Tax_owed NUMBER(10,2);
Begin
    Select aAmount * 0.05 into Tax_owed;
    Return(Tax_owed);
End;
```

В этом примере для объявления функции в системе Oracle применяется синтаксис `CREATE OR REPLACE FUNCTION`. Разработчики, обладающие опытом работы с СУРБД Oracle, поймут, что переменная `aAmount`, как следует из ее объявления, будет иметь разрядность 10, а ее точность составит два знака после запятой. Разработчики на Oracle должны заметить, что результат будет возвращаться в виде переменной `Tax_owed`. Значение переменной `aAmount` умножается на ставку налога в 5%, полученный результат сохраняется в переменной `Tax_owed`, которая возвращается пользователю либо вызывающей процедуре.

# Функции ANSI SQL

Комитет по языкам обработки данных (Committee on Data Systems and Language — CODASYL) привлек Национальный институт по стандартизации США (American National Standards Institute — ANSI), Международный комитет по техническим стандартам информации (International Committee for Information Technical Standards — INCITS) и комитет H2 для разработки стандарта, который позднее стал моделью операций над данными: сетевой моделью данных. К 1980 году была завершена разработка языка определения данных (Data Definition Language — DDL), части нового стандарта. Следующим этапом в 1982 году стала разработка языка манипулирования

данными (ЯМД) (Data Manipulation Language – DML). ЯМД предоставил некоторые новые возможности, которые, в конце концов, получили название SQL-функций.

Реляционная модель данных (Relational Data Model) позволяет отделить логическую структуру данных от ее физической реализации, обеспечивая описательный (декларативный) интерфейс. К концу 1982 года стало очевидно, что реляционная модель данных обладает рядом преимуществ перед сетевой моделью данных, и все усилия были брошены на внедрение новомодной модели. Эта "мода" подтверждает свою жизнеспособность в течение вот уже более двадцати лет и, кажется, не собирается терять своей актуальности.

На протяжении 1980-х годов оптимальным вариантом корпоративной системы управления базами данных являлась IBM DB2, что неудивительно, учитывая объем инвестиций, необходимых для того, чтобы попасть на рынок мэйнфреймовых коммерческих программных систем. Любой производитель, который хотел выйти на рынок систем управления базами данных, должен был обеспечить хотя бы минимальную совместимость с СУБД IBM DB2. Именно представитель компании IBM, Фил Шо, подал на утверждение спецификацию языка управления реляционными базами данных (Relational Database Language – RDL). После принятия этот стандарт претерпел несколько переделок, после чего превратился в стандарт ANSI SQL 1986 года. Стандарт SQL 86 (также известный как SQL 1) имел два уровня совместимости: начальный (уровень 1) и полный (уровень 2). Это был достаточно простой стандарт, полное описание которого занимало всего 105 страниц. Данный стандарт был практически сразу реализован несколькими производителями программного обеспечения, почувствовавшими шанс бросить вызов железной хватке компании IBM на этом рынке. Однако в этом стандарте отсутствовали некоторые важные функции, в первую очередь, поддержка ссылочной целостности. Поэтому вскоре возникла необходимость переработки стандарта, в результате чего на свет появилась его новая версия – SQL 89. К счастью, описание этого стандарта занимало всего 120 страниц, причем в этой версии стандарта насчитывалось менее десятка SQL-функций.

С тех пор появилось огромное количество СУРБД со встроенной поддержкой языка SQL. СУРБД Oracle использовалась преимущественно для систем нижнего и среднего уровня, тогда же появились быстро развивающаяся Ingres (которая использовала в качестве языка запросов QUEL), Informix, IBM System R (которая была заменена в 1982 году стандартом DB2).

Развитие персональных компьютеров привело к лавинообразному появлению сотен новых производителей, многие из которых уже исчезли (например, RIM, RBASE 5000, Dbase III/IV, WatcomSQL). И все они (почти) общались на все том же языке SQL! Однако из-за того, что стандарт не являлся обязательным или же производители шли по пути привлечения клиентов за счет уникальности предлагаемых ими функций, присущих тому или иному программному продукту, вскоре появилось большое количество диалектов языка SQL. Больше всего отличий наблюдалось в сфере SQL-функций. В каждую конкретную СУРБД включались десятки SQL-функций, хотя количество функций согласно стандарту было на порядок меньшим.

С увеличением мощности компьютеров возрастали требования к функциональности СУРБД. Старые производители набирались опыта, и одновременно на рынке СУРБД появлялись новые игроки. Приближалось время нового стандарта. Новый стандарт увидел свет в 1992 году, и для его полного описания потребовалось уже 575 страниц. В то же время количество "стандартных" функций возросло до 20.

После того как "золотая лихорадка", характерная для рынка СУРБД в самом начале, прошла, производители начали приходить к пониманию необходимости обеспечения своего постоянного присутствия на рынке СУРБД. Столкнувшись с потоком производителей-однодневок, они осознали необходимость принятия некоего стандарта. В моду вошла совместимость со стандартом ANSI SQL.

Чтобы иметь право заявлять о совместимости с SQL 92, СУРБД должна поддерживать начальный уровень Entry Level SQL 92. Минимальные требования по совместимости для следующего стандарта (SQL 99, или, как его еще называют, SQL3) носят названия уровня Core SQL 99. Текущий стандарт, очень кстати названный SQL:2003, требует уровня совместимости Core SQL:2003.

Компания Oracle заявляет о совместимости своих флагманских продуктов Oracle 9i и 10g со стандартом Core SQL 99, тогда как Sybase и Microsoft MySQL пока что претендуют только на уровень совместимости Entry Level SQL 92. СУРБД PostgreSQL (которая считается совместимой с наибольшим количеством стандартов) заявляет о совместимости со стандартом Core SQL 99.

Выше мы уже обращали ваше внимание на то, что в самом стандарте SQL определено ограниченное количество функций. Поэтому любая СУРБД в этом плане обладает полным (или почти полным) соответствием стандарту SQL 92. Стандарты SQL 99 и Core SQL:2003 следует рассматривать как отдельный случай, поскольку в них включено большое количество дополнительных функций. Тем не менее можно считать, что большинство рассматриваемых в книге СУРБД соответствует и этим стандартам, если не буквально, то по духу.

# Встроенные функции

Библиотека встроенных функций представляет собой самый мощный источник ресурсов для программиста на SQL. Встроенные функции позволяют программисту многократно использовать предварительно заданную логику, встраивать ее в запросы SQL, языки программирования третьего (3GL) или четвертого поколения (4GL), о которых речь пойдет ниже, в разделе "Создание, компиляция и выполнение SQL-функций". В этом разделе рассматриваются вопросы запуска и практического применения встроенных SQL-функций.

## Запуск встроенных функций

Запустить встроенную функцию несложно: необходимо указать имя и задать нужные параметры. Благодаря такой гибкости встроенные функции находят широкое применение, превращаясь в мощный инструмент разработчика. В нашем первом примере создается переменная, в которую встроенная функция `GETDATE()` впоследствии записывает результат.

```
DECLARE @THISDATE DATETIME
SELECT @THISDATE=GETDATE()
```

Аналогично, можно воспользоваться встроенной функцией `GETDATE()` в сочетании с функцией `ADD_MONTHS` внутри оператора `SELECT` для того, чтобы извлечь из базы данных информацию о номерах счетов, которые существуют более полугода.

```
SELECT INVOICE_NUM
FROM ORDERS
WHERE
   INVOICE_DATE < ADD_MONTHS(GETDATE(),-6)
```

## Практическое применение функций

Две наиболее популярные области применения функций связаны с возможностью повторного использования кода и упрощенным представлением сложных запросов. Одно из основных преимуществ функций заключается в том, что вы можете использовать их взамен программного кода, который в обычных условиях пришлось бы писать снова и снова. Функции дают возможность компактно представить большие фрагменты кода для повторного использования. Рассмотрим простейший вариант: вам необходимо, используя SQL Server, извлечь наибольшее значение столбца Amount таблицы Sales, чтобы выяснить максимальный показатель за сутки. Если бы не встроенные функции, нам пришлось бы прибегнуть к таким непрактичным приемам, как курсоры и переменные, и написать много строк кода в случае повторного использования в проекте. К счастью, среди встроенных функций предусмотрена специальная функция `MAX()`, пригодная для решения данной задачи. Функция `MAX()` возвращает максимальное значение столбца, и ее можно использовать следующим образом:

```
CREATE TABLE SALES(
                AMOUNT   NUMERIC(5,2)
               );

INSERT INTO SALES(AMOUNT) VALUES(100.00);
INSERT INTO SALES(AMOUNT) VALUES(1435.50);
INSERT INTO SALES(AMOUNT) VALUES(456.87);
INSERT INTO SALES(AMOUNT) VALUES(4500.00);
INSERT INTO SALES(AMOUNT) VALUES(564.55);
INSERT INTO SALES(AMOUNT) VALUES(3456.34);

SELECT MAX(AMOUNT) AS BIG_SALE FROM SALES;

BIG_SALE
4500.00
```

Еще одна область практического применения функций — инкапсуляция в функции сложного программного фрагмента, позволяющая упростить конечный текст программы и повысить его удобочитаемость. Предположим, что государственной организации необходимо составить серию отчетов, демонстрирующих расходы за текущий фискальный год. Задача не представляет трудностей, если финансовый год соответствует календарному. К сожалению, в Штатах финансовый год начинается с 1 октября[1], что несколько усложняет решение проблемы. Один из вариантов решения проблемы сводится к написанию запросов с помощью стандартного синтаксиса языка SQL, без помощи встроенных функций.

[1] Нам такая проблема не присуща — у нас финансовый год совпадает с календарным и длится с 1 января по 31 декабря. — *Примеч. пер.*

```
SELECT
   INVOICE_NUM, AMOUNT
FROM EXPENDITURES
WHERE
   FISCAL_YEAR =
       CASE
          WHEN MONTH(GETDATE())>= 10
          THEN YEAR(GETDATE()) + 1
          ELSE YEAR(GETDATE())
        END
ORDER BY INVOICE_NUM
```

Хотя подобный способ написания запроса допускается, можно упростить программный код, воспользовавшись функцией вычисления финансового года. Следуя данной логике, напишем следующий фрагмент кода для создания собственной функции:

```
CREATE FUNCTION DBO.CURRENT_FISCAL_YEAR(@ADATE DATETIME)
RETURNS INT AS
BEGIN
DECLARE @RTNDATE INT
SELECT @RTNDATE =
           CASE
              WHEN MONTH(@ADATE)>= 10
              THEN YEAR(@ADATE) + 1
              ELSE YEAR(@ADATE)
           END
RETURN @RTNDATE
END
```

Теперь можно не только повторно использовать этот фрагмент кода, как было только что сказано, но и упростить структуру конечного SQL-запроса. Это позволит уменьшить объем кода, который необходимо будет писать, и сделает его более читаемым, что, в свою очередь, облегчит последующую поддержку этого программного кода. Вышесказанное замечательно иллюстрирует следующий запрос по расходам, выполненный с помощью нашей новой функции:

```
SELECT
   INVOICE_NUM, AMOUNT
FROM EXPENDITURES
WHERE
   FISCAL_YEAR = DBO.CURRENT_FISCAL_YEAR(GETDATE())
ORDER BY INVOICE_NUM
```

# Создание, компиляция и выполнение SQL-функций

Идеальная функция представляет собой изолированный “черный ящик”, содержащий программный код, предназначенный для решения одной четко сформулированной проблемы и возвращения единственного результата. Изолированный фрагмент кода, не обладающий определенным возвращаемым значением, обычно называют

*процедурой (procedure)*. Благодаря растущей популярности языка C (который целиком состоит из функций) термин *функция* часто применяется в том числе и для обозначения процедур. В языке SQL важно четко различать между собой функции и процедуры. Правильно написанная SQL-программа состоит из функций и процедур, если только она не включает в себя объекты, взаимодействующие с другими частями программы при помощи методов (которые представляют собой ни больше ни меньше, как те же функции и процедуры).

Некоторые наиболее сложные системы разработки программного обеспечения (например, Visual Basic) содержат *встроенные (built-in)* функции, которые не нужно создавать самостоятельно. Например, чтобы разделить строку на части, можно воспользоваться встроенными функциями `Left`, `Right` и `Mid`. Аналогично, вам не придется изобретать способ считывания системного времени благодаря существованию встроенной функции `Now`.

Язык программирования Visual Basic разрабатывался специально для облегчения труда программистов. В него встроен компилятор, который анализирует написанный разработчиком программный код и транслирует его в понятные для компьютера исполняемые инструкции. В Visual Basic есть библиотеки функций (коллекции), которые специально написаны и откомпилированы для того, чтобы вы могли включать их в другие программы. Такие встроенные функции представляет собой изолированные фрагменты кода, являющиеся частью системы программирования (а не отдельной программы), пригодные как для одноразового, так и для многократного использования.

Система управления базами данных (СУБД) в свою очередь представляет собой коллекцию программ, хотя и предназначенных для других целей (сохранения и извлечения информации). Хранение и извлечение информации – непростая задача, для решения которой необходимо использовать многочисленные параметры и их сочетания. Язык SQL разрабатывался исключительно для работы с СУБД. Язык SQL для СУБД – это то же самое, что язык программирования для компьютерной системы. Подобно тому, как вычислительная система требует компиляции программ (их трансляции в машинные коды), так и СУБД должна компилировать SQL-запросы. И подобно тому, как вычислительная система помогает работе программ благодаря существованию встроенных функций, так и в СУБД предусматриваются встроенные функции, которые могут быть вызваны при помощи запросов SQL.

Такие языки программирования, как C, Паскаль, Ада и Бейсик, относятся к языкам программирования третьего поколения (3GL). Эти языки используются для того, чтобы научить систему, *как* нужно выполнять поставленную задачу.

Язык SQL, с другой стороны, относится к языкам программирования четвертого поколения. Языки программирования четвертого поколения сообщают системе, *что* должно быть сделано. В этой формулировке скрыто важно допущение: языки программирования четвертого поколения подразумевают, что системе известно, как нужно решать поставленную задачу.

Имея в распоряжении лишь язык программирования третьего поколения, нам пришлось бы написать следующий программный код:

```
read a record
while record id does not match the specified key do
    if not the end of file then
          read a record
```

```
    else
          stop reading
    end if
end while
if not the end of file then
 record.field = value
end if
```

При помощи языка программирования четвертого поколения достаточно написать единственную строку:

```
UPDATE table SET field=value
```

Последний вариант намного проще, но как же все происходит?

В языках программирования четвертого поколения предусмотрено хранение специальных алгоритмов, играющих роль диспетчера промежуточного уровня. Программист ставит задачу, а диспетчер говорит системе, как ее нужно решать. Фактически компилятор языка программирования четвертого поколения заменяет название задания соответствующим алгоритмом.

Но что если возникнет необходимость решить задачу, способ решения которой системе не известен? Тогда нам придется научить этому систему.

Каждый раз переписывать компилятор при добавлении новой команды несколько утомительно, не говоря уже о временны*х* затратах. Гораздо эффективнее заранее предусмотреть средства *расширения* функциональности. Еще лучше иметь средства, предоставляющие возможность добавления новых функций, которые бы не требовали модификации существующих компиляторов. Здесь нам на помощь приходят библиотеки функций.

Если попытаться рассмотреть внутреннюю реализацию компилятора SQL изнутри, можно увидеть, что команды вроде `UPDATE` или `SELECT` преобразуются в вызовы процедур и функций из соответствующих библиотек. Таким образом, все, что нам нужно, чтобы расширить функциональные возможности языка, — это добавить новые функции в существующие библиотеки либо подключить к компилятору дополнительные библиотеки.

Почему бы также не предоставить пользователям возможность добавления новых функций, написанных на подходящем языке третьего поколения? Одни производители предлагают использовать собственные языки программирования третьего поколения (например, PL/SQL в Oracle). Другие разрешают использовать компиляторы стандартных языков третьего поколения (например, C или Java). Некоторые компании по выпуску программного обеспечения позволяют включать в систему ваши собственные алгоритмы путем связывания предварительно откомпилированных функций и библиотек.

Почему мы говорим об SQL-*функциях*, но не говорим об SQL-*процедурах*? Дело в том, что SQL представляет собой узко специализированный язык, предназначенный для работы с базами данных. Над базой данных можно выполнять ровно столько операций, сколько адекватных команд предусмотрено разработчиками языка SQL для выполнения каждой операции. Как следствие, средства расширения языка отсутствуют. Вспомним, что роль функции сводится не к выполнению работы, а к тому, чтобы связать одно или несколько значений (говоря строго математическим языком, даже нулевое количество значений) с другим единственным значением (что, разумеется, требует выполнения определенной работы, но само понятие “работа” играет в дан-

ном случае второстепенную роль). Однако даже виртуальной сборной самых умных разработчиков языка не под силу предвидеть и предусмотреть все случаи, когда необходимо извлечь либо вычислить некоторую величину. Еще важнее отметить, что *логически* язык типа SQL не должен касаться деталей реализации, связанных с добычей информации. Это задача самой базы данных. Если для получения нужного результата необходимо произвести какие-либо вычисления, то это уже задание *расширения языка (language extension)*, т.е. отдельной логической сущности. Именно по этой причине средства языка SQL позволяют добавлять новые функции, но не процедуры.

В следующем разделе вы узнаете о существовании множества весьма полезных процедур (написанных на языке SQL), позволяющих выполнять операции над базами данных и решать другие важные задачи, например, задавать порядок выполнения операций (с помощью языков программирования третьего поколения) в виде последовательности действий. Такие процедуры расширяют *возможности использования* языка, но не влияют на функциональность *самого* языка SQL.

Рассмотрим вкратце процедуру языка 3GL, в состав которой входит выражение на языке SQL. Подобную процедуру можно написать, например, на процедурном расширении языка, используемом в Oracle, Procedural Language SQL (PL/SQL) или Pro*C. Язык PL/SQL разрабатывался специально для создания процедур, включающих в себя операторы SQL. Язык Pro*C сам по себе является расширением, в данном случае языка C, который позволяет добавлять в процедуры на C операторы SQL (если быть более точными, то в данном случае в функции на C).

Что происходит во время компиляции такой процедуры? На самом деле в этом случае выполняются два отдельных процесса: операторы языка 3GL обрабатываются компилятором 3GL, а SQL-выражение обрабатывается компилятором SQL. На определенном этапе результаты двух компиляций объединяются в один исполняемый файл.

Этот подход очевиден при использовании языка Pro*C для Oracle. Язык Pro*C представляет собой, по сути, предкомпилятор, результатом работы которого является синтаксически правильный код на языке C. Изучив полученный код, можно выяснить, что обработчик SQL делает с операторами SQL. Компилятор конвертирует операторы SQL в последовательность команд на языке C, помещая параметры в определенные структуры данных и вызывая нужные функции (в C, как вы помните, каждая подпрограмма представляет собой функцию), в которые эти структуры данных передаются в качестве аргументов. В данном случае "слияние" происходит на раннем этапе. Фактически компилятор языка C имеет дело только с программным кодом на C — все операторы SQL преобразуются заранее.

Ниже приведены фрагменты небольшой программы, написанной на языке Pro*C. Перед вами исходные коды этой программы. Обратите внимание на программный код, написанный в верхнем регистре, — это операторы SQL. Обычный компилятор языка C не смог бы откомпилировать данный фрагмент, поэтому сначала мы обрабатываем его предкомпилятором Pro*C и только затем передаем компилятору C.

```
/* Раздел объявления основных переменных */
EXEC SQL BEGIN DECLARE SECTION;

VARCHAR userid[20];
VARCHAR passwd[20];
VARCHAR prod_name[15];
```

```
EXEC SQL END DECLARE SECTION;
<некоторый код . . .>
/* Автоматическое обнаружение ошибок и передача их в подпрограмму обработки
ошибок */
    EXEC SQL WHENEVER SQLERROR GOTO error;
<некоторый код. . .>
    EXEC SQL CONNECT :userid IDENTIFIED BY :passwd;
<некоторый код. . .>
```

Далее приводится тот же самый фрагмент кода, но уже после обработки предкомпилятором Pro*C. Как видим, все операторы SQL преобразованы в программный код на языке C. Исходные операторы SQL помещены в комментарии.

```
/* Раздел объявления основных переменных */
/* EXEC SQL BEGIN DECLARE SECTION; */

/*обратите внимание, как объявления переменных были преобразованы в
структуры данных */
/* VARCHAR userid[20]; */
struct { unsigned short len; unsigned char arr[20]; } userid;

/* VARCHAR passwd[20]; */
struct { unsigned short len; unsigned char arr[20]; } passwd;
/* VARCHAR prod_name[15]; */
struct { unsigned short len; unsigned char arr[15]; } prod_name;
/* EXEC SQL END DECLARE SECTION; */
<некоторый код. . . >
/* Автоматическое обнаружение ошибок и передача их в подпрограмму обработки
ошибок*/
/* EXEC SQL WHENEVER SQLERROR GOTO error; */

    /* Подключение к базе данных Oracle */

    /* EXEC SQL CONNECT :userid IDENTIFIED BY :passwd; */
```

Обратите внимание на схему преобразования оператора `CONNECT`. Специально для него объявляется большая структура данных, в которую загружается разнообразная необходимая информация, а в последних строках вызывается функция языка C, осуществляющая подключение к базе данных и проверку ошибок. Результат преобразования SQL-оператора `CONNECT` приведен ниже.

```
{
 struct sqlexd sqlstm;
 sqlstm.sqlvsn = 10;
 sqlstm.arrsiz = 4;
 sqlstm.sqladtp = &sqladt;
 sqlstm.sqltdsp = &sqltds;
 sqlstm.iters = (unsigned int  )10;
... (здесь еще много-много операторов !) ...
 sqlstm.sqphss = sqlstm.sqhsts;
 sqlstm.sqpind = sqlstm.sqindv;
 sqlstm.sqpins = sqlstm.sqinds;
 sqlstm.sqparm = sqlstm.sqharm;
 sqlstm.sqparc = sqlstm.sqharc;
 sqlstm.sqpadto = sqlstm.sqadto;
 sqlstm.sqptdso = sqlstm.sqtdso;
```

```
и, наконец, вызов функции:
 sqlcxt((void **)0, &sqlctx, &sqlstm, &sqlfpn);
 if (sqlca.sqlcode < 0) goto error;
}
```

Данный пример иллюстрирует, каким образом воплощается в жизнь "магия" языков четвертого поколения (4GL): каждый оператор языка 4GL транслируется в программный код на языке программирования третьего поколения (3GL), который манипулирует данными и вызывает функции, как это обычно принято в языках 3GL.

# Передача параметров с помощью значения либо ссылки

*Параметром (parameter)* называют любую используемую функцией переменную, в которую обязательно следует поместить некоторое значение, прежде чем функция сможет начать вычисления. Параметры могут передаваться в функцию с помощью *значений (by value)* или *ссылок (by reference)*. Если в функцию передается копия некоторого значения, то говорят, что передача параметра осуществляется с помощью значения. Если же функции предоставляется доступ к самой переменной, содержащей реальное значение, то говорят о передаче параметров с помощью ссылки. В языке C, где эти различия не столь четко определены, передача с помощью ссылки осуществляется путем передачи не самой переменной, а указателя на нее (или ее адреса).

Для SQL-функции очевидный способ передачи параметра с помощью ссылок заключается в передаче буфера (представляющего собой контейнер для хранения переменной), в который функция должна поместить определенную информацию, извлеченную из базы данных. (Ключ, по которому осуществляется поиск необходимой информации, обычно передается в виде значения.)

Рассмотрим механизмы передачи параметров с помощью ссылок и значений более подробно. Начнем с краткого обзора небезызвестных тем памяти, ее адресации и указателей, которые всегда вызывают немало вопросов.

Любая программа хранит переменные в памяти. С точки зрения программы память (memory) представляет собой непрерывный массив ячеек, который можно сравнить с множеством почтовых ящиков в большом жилом здании или же с набором отделений для бумаг в картотеке. При этом каждой ячейке памяти присваивается порядковый номер.

Ячейки памяти можно сравнить с земельными участками, размещенными вдоль центральной улицы поселка. Все участки пронумерованы. Разработчик может объединить два соседних участка и построить дом нестандартного размера. Адресом нового дома станет адрес первого участка. Для стороннего наблюдателя будет казаться, что этот дом, как и любой другой, занимает один участок. Выступая в роли стороннего наблюдателя, мы говорим, что каждая ячейка памяти может содержать значение, не затрагивая вопрос размерности этого значения.

Каждая ячейка, которая содержит (или может содержать) полезное значение, необходимое для работы программы, называют *программной переменной (program variable)*. Для работы с переменными программа использует адреса (номера ячеек памяти). Программисты присваивают программным переменным мнемонические или, наоборот, бессмысленные имена, в зависимости от уровня культуры программирования

и склада ума. Программу не волнует, какие будут называться переменные. Во время компиляции вместо "умных" названий подставляются адреса ячеек памяти. (Либо значения смещения от некоторого базового адреса, поскольку программа может загружаться в различные области памяти, но для простоты мы говорим об адресах памяти. Исходя из тех же соображений будем считать, что для обозначения адреса в памяти нам хватит четырехзначного числа, хотя в современных компьютерах для обозначения адреса понадобится больше цифр. Данный момент не влияет на суть изложения.)

Когда программе необходимо прочесть значение переменной, она говорит процессору: "Дай мне то, что хранится по адресу [1800]". Или, наоборот, программа может попросить процессор записать значение в память по адресу [1800], как показано ниже.

```
int x = 5; /* запись значения по адресу 1800 */
...|___|_5_|___|...
 1799 1800 1801
x = x * 3; /* извлечение значения по адресу 1800, умножение его на 3 и
запись в ту же ячейку */
...|___|15_|___|...
 1799 1800 1801
```

Заметьте, что в данном фрагменте *значение* является независимым от *переменной*, в которой оно хранится. Можно извлечь значение 5 из ячейки [1800], умножить его на 3, извлечь кубический корень и вообще сделать с ним все, что заблагорассудится. До того момента, как мы сохраним результат вычислений в ячейке [1800], переменная в ячейке [1800] будет по-прежнему содержать значение 5. (В предыдущем фрагменте кода мы сохранили результат в переменной, поэтому значение ячейки [1800], естественно, изменилось.)

Теперь представим набор команд, которые программа должна выполнить над серией схожих значений.

Вместо того чтобы заново писать инструкции для каждой переменной, мы можем написать цикл, который программа будет запускать каждый раз, когда ей необходимо "перейти к ячейке памяти [XXXX] и извлечь из нее значение". Под XXXX подразумевается *адрес переменной, которая будет использоваться на этот раз*. Здесь нам также понадобится переменная-*указатель*, т.е. переменная, содержащая адрес следующей переменной. (Разумеется, в цикле должна находиться команда, которая будет изменять содержимое ячейки памяти XXXX).

```
...|___|_5_|_3_|_7_|12_|...
 1799 1800 1801 1802 1803
int *xp = &x;
...|___|1800|___|...
 2155  2156 2157
```

Передавая параметр с помощью значения, программа говорит: "Извлечь значение из ячейки [1800] и поместить его в стек". Передавая параметр с помощью ссылки, программа говорит: "Поместить '1800' в стек". В первом случае в стеке будет сохранено число 5; во втором случае там будет храниться число 1800. Кроме того, программа знает способ передачи данной переменной — с помощью значения или ссылки.

Когда аргумент передается функции в виде значения, функция поступает с ним по своему усмотрению. Функция может принять значение 5, превратить его в 15 или 357, но в ячейке памяти с адресом [1800], откуда было извлечено это значение, попрежнему будет храниться 5, а не 15 или 357. Поскольку параметр передавался с помощью значения, функция работает с локальной копией переменной, существующей лишь в пределах видимости функции.

Во втором случае ситуация несколько иная, так как функция, обнаружив, что параметр был передан при помощи ссылки, передает в стек не само значение (в данном случае 5), а ссылку на адрес памяти, где это значение хранится. Функция может выполнять над этим значением те же операции, например, умножать 5 на 3, чтобы получить 15, но при этом последовательность операций будет следующей: "Взять переданное значение, *воспользоваться указанным адресом, чтобы извлечь из памяти размещенное по этому адресу значение*, утроить его, затем снова взять переданное значение и *воспользоваться указанным адресом, чтобы записать утроенное значение в нужную ячейку памяти*".

Теперь даже через некоторое время после завершения работы функции в указанной ячейке памяти по-прежнему будет храниться число 15, которое сможет прочесть любая другая программа. Очевидно, что в первом случае результат выполнения всех математический операций над переменной будет потерян после завершения работы функции (подробнее об этом — ниже, в разделе "Область видимости функции"), только если вы не организуете взаимодействие с программой каким-либо иным способом, чтобы *полученное значение записывалось в независимую переменную.*

> *Хотя передача параметров в функцию может осуществляться с помощью ссылок, что позволяет получить несколько возвращаемых значений, – это не лучший вариант. В большинстве случаев тот факт, что функция возвращает несколько значений, свидетельствует о попытке выполнения данной функцией сразу нескольких задач, а это уже пример неправильного проектного решения. Возвращения значений в виде выходных параметров следует избегать еще и по той причине, что подобную функцию не всегда можно использовать в операторе SQL.*

Единственное место в СУРБД, где можно выбирать способ передачи параметров (по значению или по ссылке), — это определяемые пользователем функции (UDF) или хранимые процедуры. Данное утверждение относится к функциям и процедурам, созданным при помощи процедурных расширений языка SQL (PL/SQL, Transact-SQL и т.д.). Встроенные SQL-функции принимают параметры с помощью ссылок. Более подробно эта тема обсуждается в главе 3.

# Область видимости функции

Блоком кода называется логически завершенный фрагмент. Этот фрагмент может иметь произвольный размер. В роли этого фрагмента может выступать одна или несколько функций, тело цикла либо условное выражение (`IF` *условие* `THEN` *блок кода* `ELSE` *блок кода* `ENDIF`). На данном этапе важно отметить, что объекты, объявленные внутри блока кода, как правило, существуют и являются видимыми только внутри блока. Доступ к переменной, объявленной внутри блока, можно получить только из самого блока кода. (Более того, если мы специально не позаботимся об обратном, то переменная даже *не будет существовать* за пределами блока. Она будет создаваться

при входе в блок и уничтожаться при выходе из него.) Такой принцип называют *инкапсуляцией (encapsulation)*, и он важен по нескольким причинам. Рассмотрим две из них.

- **Логические рассуждения.** Если мы объявили функцию внутри блока, а затем нам понадобилось вызвать ее снаружи блока, то это первый признак возникновения противоречий. В таком случае нам нужно приостановить дальнейшую работу и задуматься над тем, что было сделано неправильно. Возможно, мы вынесли *наружу* блока то, что должно находиться *внутри*, или наша функция носит более общий характер и ее не следует ограничивать рамками единственного блока. Такие рассуждения помогают контролировать мыслительные процессы.
- **Отладка и ограничение возможного урона.** Если переменная объявлена внутри блока, гораздо легче определить, какой программный код может изменять эту переменную, поскольку никакой программный код за пределами блока не в состоянии получить к ней доступ. Программный код, в котором переменная (или функция, или любой другой программный объект) является *видимой* (т.е. доступной), называется областью видимости этого объекта. Блоки кода имеют иерархическую структуру (т.е. одни блоки могут находиться внутри других). Основное правило для области видимости гласит, что объект является видимым внутри блока, в котором он определен, и внутри всех остальных блоков, определенных внутри данного.

## Повышенная безопасность

Задание областей видимости способствует повышению безопасности, поскольку, если область видимости защищена с помощью некоторого элемента контроля доступа, под эту защиту подпадает все, что находится внутри данной области. Например, если ваша функция находится внутри пакета, для того чтобы пользователь мог ее выполнить, ему должны быть предоставлены достаточные полномочия для доступа к пакету, даже если с точки зрения программы функция принадлежит к области видимости, доступной для пользователя. В большинстве СУРБД обеспечивается управление доступом на уровне отдельных объектов базы данных (в нашем случае функций). Запускать функцию могут только пользователи, обладающие привилегией `EXECUTE` для данного объекта.

Различные расширения SQL позволяют задавать большие области видимости, включающие в себя функции, процедуры, таблицы и т.д. В БД Oracle, например, существует понятие пакетов и схем. В MS SQL Server и Sybase в качестве области видимости выступает целая база данных и т.д.

*В Oracle одна база данных может состоять из нескольких пакетов. При этом функция, принадлежащая одному пакету, имеет доступ к функциям из другого пакета, даже если исходя из правил области видимости такого быть не должно. Для этого, во-первых, необходимо получить доступ к самому пакету, например, следующим способом: `<имя_пакета>.<имя_функции>`. То же самое справедливо, если функция, объявленная внутри одной схемы, обращается к функции из другой схемы. В этом случае необходимо задавать название схемы: `<имя_схемы>.<имя_пакета>.<имя_функции>`. Разумеется, для того чтобы подобный запрос увенчался успехом, вы должны обладать соответствующими привилегиями.*

## Перегрузка

Концепция *перегрузки (overloading)* была позаимствована из объектно-ориентированного программирования (ООП). Суть концепции сводится к тому, что *одна и та же задача* в зависимости от обстоятельств может решаться *различными способами*.

Поясним это на примере. Если у вас двое детей и один из них звонит вам и просит принести домой его любимое лакомство, вам необходимо выполнить задачу КупитьЛакомствоДляРебенка. В зависимости от того, кто именно попросил это сделать, вы выбираете любимое лакомство для определенного ребенка.

Конечно, можно подойти к решению этого вопроса по-другому, сформулировав две различные задачи: КупитьЛакомствоДляКати и КупитьЛакомствоДляДжейн, однако это будет означать фокусировку на *деталях реализации* там, где речь должна идти о самой концепции. Что нужно сделать? КупитьЛакомствоДляРебенка. Для какого ребенка? Это зависит от обстоятельств (*входного параметра*).

Перегрузкой в SQL называют ситуацию, когда несколько функций с одинаковым наименованием работают с разными наборами аргументов или типами данных (либо отличаются по обеим характеристикам).

Рассмотрим пример на языке PL/SQL в Oracle:

```
create or replace function AddSomething
(
  arg1 IN number,
  arg2 IN number
) return number
is
begin
return arg1 + arg2;
end AddSomething;

create or replace Function AddSomething
(
  arg1 IN varchar,
  arg2 IN varchar
) return varchar
is
begin
return arg1 || arg2;
end AddSomething;
```

Все эти функции имеют одинаковые имена, но при этом каждая функция принимает и передает значения различных типов. Первая функция принимает два числовых значения, складывает их и возвращает сумму (тоже число). Вторая функция принимает две строчные переменные и возвращает новую строку (результат объединения двух строк, заданных в качестве входных параметров). При вызове функции `AddSomething` СУБД Oracle сама решает, какую конкретно функцию следует вызвать, исходя из типа данных аргументов (переданных вызывающей программой и ожидаемых с ее стороны значений).

*В Oracle перегруженные функции могут создаваться только в качестве части пакета. Автономные перегруженные функции не поддерживаются.*

Следует подчеркнуть, что перегрузка функций представляет собой чрезвычайно полезный *инструмент разработчика*. Однако, как и любой инструмент, иногда он может использоваться не по назначению. Чаще всего использование перегрузки не по назначению приводит к запутыванию общей картины и смазыванию оригинальной идеи разработчика.

Сложно привести изолированный пример использования перегрузки не по назначению (потому что вопрос о том, что такое хорошо, а что такое плохо, необходимо рассматривать с точки зрения общего проекта, а это требует наличия определенного контекста). Тем не менее, когда вы выбираете между перегрузкой существующей функции и объявлением новой, прежде всего задайте себе следующий вопрос: выполняют ли эти функции *одинаковые* задачи? Можете ли вы сказать программе, как сказали бы это своему подчиненному: "Послушай, я хочу, чтобы ты выполнил *это задание*, так что иди и сам решай, как это сделать!" Если "да", то смело используйте перегрузку функций. Если же разговор с подчиненным прозвучал бы как: "Ты должен выполнить *несколько заданий*, они очень просты, так что если ты справишься с одним из них, остальные не вызовут у тебя никаких затруднений", тогда лучше создавать отдельные функции.

Концепция перегрузки включена в стандарт SQL 99. Перегрузка определена в пакете "поддержки SQL/MM" как "функция T322". Данная возможность рассматривается как дополнительная, поэтому далеко не каждый производитель СУРБД поддерживает ее в своих программных продуктах.

- СУРБД Oracle поддерживает перегрузку функций, объявленных в качестве части пакета.
- IBM DB2 поддерживает перегрузку функций, объявленных в различных схемах.
- В Microsoft SQL Server и Sybase перегрузка функций запрещена явным образом. Любая попытка создания функции (или же другого объекта базы данных) с использованием уже существующего имени приведет к ошибке компиляции. Что неудивительно, так как для любого объекта существует запись в таблице `SYSOBJECTS` текущей базы данных, на которую наложено ограничение `UNIQUE`. Как следствие, любой объект должен иметь уникальное имя.
- В PostgreSQL полностью поддерживается перегрузка как встроенных, так и определяемых пользователем функций. Вы даже можете создать собственную версию встроенной функции.
- В MySQL перегрузка функций не предусмотрена.

# Классификация SQL-функций: детерминистские и недетерминистские функции

Стремление к порядку является неотъемлемой частью человеческой натуры. Порядок приносит чувство стабильности и часто облегчает жизнь.

*Классификацией* или *систематизацией (classifying)* называют группирование объектов с общими атрибутами, характеристиками либо просто некоторым образом связанных друг с другом. Если вы скажете, что данная формулировка ненаучна, нам придется с вами согласиться. Однако классификация SQL-функций по категориям осуществляется именно так.

Существует множество вариантов классификации SQL-функций, и вам ничто не мешает придумать собственный вариант. Стандарт SQL не предусматривает какой-либо определенной классификации, поэтому каждый производитель в своей документации структурирует их в соответствии с личными предпочтениями.

Знание базовой классификации функций у того или иного производителя баз данных, безусловно, важно для любого, кому приходится работать с несколькими СУРБД. Но еще важнее понимать, по каким признакам все SQL-функции (и не только SQL-функции) делятся на две большие категории: детерминистские и недетерминистские.

Итак, в первую очередь функции классифицируются по типу: *детерминистские (deterministic)* или *недетерминистские (non-deterministic)*. Эта классификация описана в стандарте ANSI SQL 99 (в разделе "SQL/Persistent Storage Module"), и каждый производитель СУРБД в той или иной мере поддерживает данную классификацию. Не существует строгих правил, по которым та или иная SQL-функция может быть отнесена к определенному типу. В стандарте SQL 99 предусмотрено ключевое слово `DETERMINISTIC`, используемое для объявления детерминистских функций. По сути, данное ключевое слово используется исключительно как индикатор намерения.

Сформулировать принципы классификации по типу очень просто. Функции первого типа всегда возвращают один и тот же результат при задании одинаковых параметров. Недетерминистские функции могут возвращать различные результаты при задании одинаковых параметров. Примером детерминистской функции является функция `LENGTH`. При задании аргумента строкового типа данных она всегда возвращает длину строкового параметра. Сколько бы раз вы ни вызывали эту функцию, передавая ей один и тот же параметр, результат всегда будет одним и тем же.

Функция, которая не принимает никаких аргументов, называется *нульместной (niladic)*. В английском языке этот термин является производным от математического термина "dyadic", что означает диадический (двуместный), — так описывается математический оператор, в котором два вектора пишутся рядом друг с другом без знака точки между ними (например, A B). Происходит это слово от греческого "dyo", которое переводится как "два". Производными от данного являются термины *монадический (одноместный)*, *диадический (двуместный)* или *триадический (трехместный)*, которые широко применяются в документации Комитетом по стандарту SQL.

Рассмотрим пример функции, выполняемой через интерфейс Oracle SQL*Plus:

```
SELECT  LENGTH('word') word_length FROM DUAL;
word_length
----------------
             4
```

В качестве примера недетерминистской функции можно привести функцию `GETDATE()`, используемую в Microsoft SQL Server. Если вызвать эту функцию без аргументов, то она возвратит системное время на сервере:

```
SELECT GETDATE() whatstime_now
WAITFOR DELAY '00:00:01'
SELECT GETDATE() whatstime_again

whatstime_now
-----------------------------------------------------
2003-10-30 22:50:08.110
 (1 row(s) affected)
```

```
whatstime_again
-----------------------------------------------------
2003-10-30 22:50:09.108
 (1 row(s) affected)
```

Как видите, секундная задержка между двумя запросами, выполненными в SQL Server Query Analyzer (плюс некоторые не поддающиеся вычислению поправки, вызванные выполнением в это же время разнообразных серверных процессов), приведет к различным результатам, хотя наши запросы абсолютно идентичны.

А вот еще один пример недетерминистской функции, GENERATE_UNIQUE() из IBM DB2 UDB. Эта функция специально предназначена для генерации уникальных значений (которые никогда не повторяются при каждом новом вызове функции, благодаря чему данная функция может применяться для генерации ID).

```
db2 => select generate_unique() as unique_id from sysibm.sysdummy1

UNIQUE_ID
------------------------------
x'20031106225757157066000000'
1 record(s) selected.
```

Тот же самый запрос, выполненный спустя одну минуту, выдаст совершенно иной результат:

```
db2 => select generate_unique() as unique_id from sysibm.sysdummy1
UNIQUE_ID
------------------------------
x'20031106230106388183000000'

1 record(s) selected.
```

В одних СУРБД (например, в Microsoft SQL Server или IBM DB2) различия между типами функций достаточно четкие, в других об этих различиях вообще практически не упоминается. Тот факт, что при определенных условиях одни и те же функции могут являться как детерминистскими, так и недетерминистскими, только добавляет путаницу. Рассмотрим в качестве примера встроенную в СУРБД Oracle аналитическую функцию FIRST_VALUE, которая возвращает первое значение упорядоченного множества. Поскольку любое множество может быть упорядочено по крайней мере двумя способами (например, в порядке убывания или в порядке возрастания), в данном случае детерминизм функции зависит от выражения ORDER BY. Если в транзакцию вовлечена функция SQL, то на детерминизм функции сильно влияет уровень изоляции транзакций, особенно если он отличен от уровня SERIALIZABLE.

Одни производители программного обеспечения (например, Oracle и IBM), которые разрешают создавать определяемые пользователем функции (что в той или иной мере позволяют делать все СУРБД, описываемые в данной книге), предусмотрели специальный синтаксис SQL для задания типа функции (детерминистская или недетерминистская). В СУРБД других производителей (например, у Microsoft с их SQL Server) тип пользовательской функции определяется исходя из используемых в ней типов объектов. Определяемые пользователем функции рассматриваются в главе 4.

Почти у каждого производителя СУРБД существуют некоторые ограничения, накладываемые на использование недетерминистских функций, независимо от того, какими являются эти функции: встроенными или определяемыми пользователем. Это важный отличительный признак любой SQL-совместимой СУРБД, хотя иногда он даже не упоминается в документации.

## Oracle

В документации по Oracle не акцентируется внимание на детерминистском или недетерминистском характере SQL-функций. Тем не менее данное понятие все же используется в основном в *аналитических* функциях. Ниже приведен список ограничений, касающихся использования недетерминистских функций в Oracle.

- Недетерминистские функции нельзя использовать при проверке ограничений в выражении `CHECK`. Кроме того, в ограничения нельзя включать вызов определяемых пользователем функций.
- Недетерминистские функции нельзя использовать в индексах, основанных на функциях. При создании любой определяемой пользователем функции, которую планируется использовать в этих целях, необходимо задавать ключевое слово `DETERMINISTIC`.
- Определенные приемы оптимизации (например, использование `ENABLE QUERY REWRITE`), основанные на функциях, применяемых для создания объектов (представлений, материальных представлений), допускают использование только детерминистских функций.

## IBM DB2 UDB

В IBM DB2 UDB недетерминистские функции (иногда называемые *вариантными (variant)* в документации этого производителя) имеют многочисленные ограничения по применению. Ниже приведены основные ограничения, связанные с использованием недетерминистских функций в среде IBM DB2 UDB.

- Недетерминистские функции нельзя использовать в условии объединения `FULL OUTER JOIN` оператора `SELECT`.
- Недетерминистские функции нельзя использовать в операторе `CASE`.
- Недетерминистские функции нельзя использовать в ограничении `CHECK`.
- Недетерминистские функции нельзя использовать в индексах, основанных на функциях, или в операторе `CREATE INDEX EXTENSION`.
- Недетерминистские функции нельзя использовать в объекте `REFRESH IMMEDIATE SUMMARY TABLE`.
- Недетерминистские функции нельзя использовать при создании типизированных представлений и вложенных представлений или же представлений, созданных с включенной опцией `WITH CHECK`.
- Недетерминистские определяемые пользователем функции нельзя использовать с параметром `PREDICATE`.
- В недетерминистских определяемых пользователем функциях нельзя использовать выражение `ALLOW PARALLEL` (которое определяет, какие функции могут вызываться или выполняться параллельно, — например, на нескольких процессорах).
- Недетерминистские функции нельзя использовать в функциях онлайновой аналитической обработки (OLAP) с параметрами `ORDER BY` или `PARTITION BY`.

В случае нарушения этих (или каких-то других) ограничений IBM DB2 UDB выдает сообщение об ошибке SQLSTATE 42845.

Если хорошо подумать, то такие ограничения действительно оправданы. Как можно создать FULL OUTER JOIN, основываясь на критерии объединения, заданном при помощи недетерминистической функции? Ведь каждый критерий, задаваемый в параметре WHERE, может выдавать "соответствие" и "несоответствие" для одного и того же набора значений! Представьте себе использование параметра ORDER BY с недетерминистской функцией. Если функция возвращает различные значения для сортируемого выражения, как можно быть уверенным в правильности сортировки даже для идентичных наборов данных? Конечно, в случае с другими ограничениями причины не столь очевидны и в большинстве своем касаются особенностей реализации. Хотя некоторые разработчики утверждают, что и такой сценарий имеет право на существование и может найти свою нишу в программировании бизнес-логики, наше мнение таково, что пользователи, создающие подобные запросы, должны хорошо представлять возможные последствия, вызванные ветвлением результата в недетерминистских функциях.

## Microsoft SQL Server

В Microsoft SQL Server существуют два главных ограничения на использование недетерминистских функций.

- Если выражение для вычисляемого столбца содержит недетерминистскую функцию, то для такого столбца нельзя создать индекс.
- Если представление включает в себя недетерминистские функции, то для него нельзя создать кластерный индекс.

В отличие от IBM DB2 UDB и Oracle, в Microsoft SQL Server использование недетерминистских функций в ограничении CHECK связано с меньшей путаницей, хотя большинство других ограничений справедливо и для этого программного продукта.

Приведем пример. Таблица CHECK_TEST состоит из единственного столбца под названием FIELD1, для которого задано значение по умолчанию, представляющее собой текущую системную дату. В то же время ограничение CHECK требует, чтобы новая запись содержала значение, не равное текущей системной дате (конечно, данное ограничение абсолютно бесполезно, поскольку оно нарушается значением по умолчанию, но нам нужно доказать правильность нашего суждения).

```
CREATE TABLE check_test
(
  field1 DATETIME NOT NULL DEFAULT GETDATE()
  CHECK (field1 < GETDATE())
)

The command(s) completed successfully.
```

После создания таблицы попробуем выполнить приведенный ниже оператор INSERT. К сожалению, нам это не удастся, поскольку данный оператор нарушает заданное ограничение:

```
INSERT INTO check_test VALUES (default)

Server: Msg 547, Level 16, State 1, Line 1
INSERT statement conflicted with COLUMN CHECK constraint 'CK__check_tes__
field__40EF84EB'. The conflict occurred in database 'master', table 'check_
test', column 'field1'.
The statement has been terminated.
```

В то же время выполнение следующего запроса увенчается успехом:

```
insert check_test VALUES(GETDATE()-1)

(1 row(s) affected)
```

Однако утверждать, что даже такое ограничение общего плана будет соблюдено в различных СУРБД, нельзя.

Попытка выполнить аналогичный запрос в Oracle вызовет исключение: "ORA-02436: date or system variable wrongly specified in `CHECK` constraint" (дата или системная переменная неправильно определена в ограничении `CHECK`). В IBM DB2 UDB попытки создания таблицы с ограничением `CHECK`, ссылающимся на регистр `CURRENT DATE`, также будут отклонены. В СУРБД Sybase, связанной общим прошлым с Microsoft SQL Server, ситуация напоминает SQL Server (здесь можно создать таблицу, но при добавлении значения по умолчанию возникнет сообщение об ошибке). PostgreSQL ведет себя аналогично Oracle. В MySQL на момент написания этой книги ограничение `CHECK` еще не было реализовано.

Тогда как у одних производителей (Oracle, IBM), для того, чтобы объявить определяемую пользователем функцию как детерминистическую, нужен специальный параметр, другие производители, например Microsoft, отдают предпочтение более обстоятельному подходу. В SQL Server определяемая пользователем функция считается детерминистической, если:

- функция является привязанной к схеме (т.е. функция создана с использованием опции `SCHEMABINDING`, а это означает, что объекты, на которые ссылается данная функция, не могут изменяться или удаляться);
- каждая функция (неважно, встроенная или определяемая пользователем), вызываемая из тела этой функции, является детерминистской;
- в теле функции отсутствуют ссылки на объекты базы данных (например, таблицы, представления, и другие функции), выходящие за пределы области видимости;
- функция не обращается к расширенным хранимым процедурам (которые могут изменять состояние базы данных).

Любая определяемая пользователем функция, которая не соответствует хотя бы одному из вышеизложенных требований, считается недетерминистской по определению.

## Sybase

В Sybase нет специальных правил по использованию недетерминистских функций, так как в Sybase просто запрещены многие сценарии, доступные в СУРБД других производителей. Например, в Sybase нельзя создавать индексы для вычисляемых

столбцов или представлений. Несмотря на то что используемая вами СУРБД может допускать разный синтаксис, применение недетерминистских функций, например в запросе FULL OUTER JOIN, нельзя назвать удачной идеей просто потому, что результаты подобного запроса непредсказуемы. То же самое справедливо по отношению к использованию недетерминистских функций в параметрах GROUP BY и ORDER BY.

> В Sybase не предусмотрено иных механизмов создания определяемых пользователем функций, кроме языка программирования Java. Хотя в синтаксисе языка SQLJ (диалекта SQL, используемого для создания функций и процедур Java) существуют ключевые слова "deterministic" и "non-deterministic" для "обеспечения синтаксической совместимости со стандартом ANSI", однако, как утверждается в документации, эти ключевые слова до сих пор не реализованы.

## MySQL и PostgreSQL

MySQL и PostgreSQL, две ведущие СУРБД, распространяемые по принципу "открытого кода", не содержат специальных директив, касающихся использования недетерминистских функций. Частично это вызвано тем, что в них отсутствуют многие возможности, реализованные их конкурентами с "закрытым кодом", а частично из-за стремления к простым решениям.

Большинство ограничений, актуальных для корпоративных СУРБД (Oracle, IBM и Microsoft), не распространяются на MySQL и PostgreSQL. Подобные сценарии запрещены либо считаются излишествами. Вы просто должны придерживаться базовых правил языка SQL и не применять недетерминистские функции в запросах FULL OUTER JOIN либо параметрах GROUP BY и ORDER BY, если вам не до конца ясны все возможные ветвления результатов выполнения запроса в данной ситуации.

# Заключение

После прочтения этой главы вы должны составить общее представление о механизмах реализации функций в SQL, а также об основных принципах классификации функций. Кроме того, вы должны четко понимать, каким образом различные СУРБД стараются следовать общей тенденции соблюдения стандарта ANSI, по крайней мере, частично.

Стандарт SQL впервые был утвержден в 1978 году по поручению комитета CODASYL. Количество встроенных функций в стандарте увеличивалось с выходом каждой новой версии, напоминая бег вдогонку по отношению к производителям СУРБД, реализующим в своих программных продуктах сотни встроенных SQL-функций. В последнюю версию стандарта, SQL:2003, официально принятую в 2004 году, были включены новые типы данных (такие как BIGINT, MULTISET и XML) и новые агрегатные функции (такие как COLLECT, FUSION и INTERSECTION).

SQL-функции представляют собой часть ядра стандарта ISO/ANSI и поэтому обязательны для реализации в целях совместимости. Большинство производителей СУРБД заявляют, по крайней мере, о соответствии своих программных продуктов стандарту SQL 92 (SQL2), а некоторые производители (Oracle, IBM DB2 UDB и PostgreSQL) даже говорят о совместимости со стандартом SQL:2003. Важно понимать, что такая совместимость касается конкретных версий программных продуктов

(например, Oracle8i соответствует стандарту SQL 92, тогда как версии Oracle9i/10g претендуют на соответствие стандарту SQL:2003). Не следует забывать об этом, поскольку вы не сможете воспользоваться расширенной функциональностью последних версий стандарта SQL, если будете использовать старые версии программного обеспечения.

Классификация представляет собой процесс группирования объектов по общим характеристикам. Существует множество вариантов классификации SQL-функций. Все SQL-функции можно поделить на две большие категории: детерминистские и недетерминистские. Различие между этими двумя категориями состоит в том, что функции из первой категории всегда возвращают один и тот же результат при задании одинаковых параметров, тогда как функции из второй группы могут давать разный результат при задании одних и тех же аргументов.

В главе 3 сравниваются встроенные SQL-функции различных производителей СУРБД.