

Министерство образования и науки, молодежи и
спорта Украины

Харьковский национальный университет

Широкопетлева М.С., Черепанова Ю.Ю., Мазурова О.А.

ПИ

Языки запросов к базам данных

Програмная инженерия

Харьков

2012

Содержание

Теория.....	4
Оптимизация запросов.....	4
Окружение 4-го поколения.....	15
Язык PL / SQL. Общие положения	26
Переменные и константы. Кодовое множество.....	29
Типы данных.....	41
Команды управления.....	45
Создание хранимых процедур и функций	59
Параметры подпрограмм.....	67
Встроенные функции PL / SQL.....	77
Курсоры.....	81
Исключительные ситуации.....	92
Обработчики исключений	98
Пакеты.....	111
Триггеры объектов	123
Практика.....	143
Практика 2. Использование подпрограмм в языках программирования 4-го поколения.....	143
Лабораторная работа 2 Подпрограммы как средство создания серверных частей информационных систем на примере СУБД Oracle	151
Практика 3. Создание и работа с триггерами объектов и событий.....	153
Лабораторная работа 3. Триггеры объектов и событий базы данных	161
Текущий контроль знаний.....	163
Управляющие структуры.....	163
Подпрограммы языка PL/SQL.....	168

Курсоры.....	176
Исключительные ситуации.....	184
Пакеты.....	187
Триггеры.....	190
Словарь терминов.....	196

Теория

Оптимизация запросов

Оптимизация запросов - это:

а) функция СУБД, осуществляющая поиск оптимального **плана выполнения запросов** из всех возможных для заданного запроса,

б) процесс изменения запроса и/или структуры БД с целью уменьшения использования вычислительных ресурсов при выполнении запроса.

Один и тот же результат может быть получен СУБД различными способами (планами выполнения запросов), которые могут существенно отличаться как по затратам ресурсов, так и по времени выполнения. Задача оптимизации заключается в нахождении оптимального способа.

В реляционной СУБД оптимальный план выполнения запроса - это такая последовательность применения операторов реляционной алгебры к исходным и промежуточным отношениям, которое для конкретного текущего состояния БД (её структуры и наполнения) может быть выполнено с минимальным использованием вычислительных ресурсов.

В настоящее время известны две стратегии поиска оптимального плана:

- грубой силы путём оценки всех перестановок соединяемых таблиц;
- на основе генетического алгоритма путём оценки ограниченного числа перестановок.

Планы выполнения запроса сравниваются исходя из следующих факторов:

- потенциальное число строк, извлекаемое из каждой таблицы, получаемое из статистики;
- наличие индексов;
- возможность выполнения слияний (merge-join).

В общем случае соединение выполняется вложенными циклами. Однако этот алгоритм может оказаться менее эффективен, чем специализированные алгоритмы. Если у сливаемых таблиц есть индексы по соединяемым полям, или одна или обе таблицы достаточно малы, чтобы быть отсортированными в памяти, то исследуется возможность выполнения слияний.

Общая схема обработки запроса

Можно представить, что обработка поступившего в систему запроса состоит из фаз,

изображенных на рис. 1.



Рисунок 1 - Фазы выполнения запроса

На **первой** фазе запрос, заданный на языке запросов, подвергается лексическому и синтаксическому анализу. При этом вырабатывается его внутреннее представление, отражающее структуру запроса и содержащее информацию, которая характеризует объекты базы данных, упомянутые в запросе (отношения, поля и константы). Информация о хранимых в базе данных объектах выбирается из каталогов базы данных. Внутреннее представление запроса используется и преобразуется на следующих стадиях обработки запроса. Форма внутреннего представления должна быть достаточно удобной для последующих оптимизационных преобразований.

На **второй** фазе запрос во внутреннем представлении подвергается логической оптимизации. Могут применяться различные преобразования, "улучшающие" начальное представление запроса. Среди преобразований могут быть эквивалентные, после проведения которых получается внутреннее представление, семантически эквивалентное начальному (например, приведение запроса к некоторой канонической форме). Преобразования могут быть и семантическими: получаемое представление не является семантически эквивалентным начальному, но гарантируется, что результат выполнения преобразованного запроса совпадает с результатом запроса в начальной форме при соблюдении ограничений целостности, существующих в базе данных. После выполнения второй фазы обработки запроса его внутреннее представление остается непроцедурным, хотя и является в некотором смысле более эффективным, чем начальное.

Третий этап обработки запроса состоит в выборе на основе информации, которой располагает оптимизатор, набора альтернативных процедурных планов выполнения данного

запроса в соответствии с его внутренним представлением, полученным на второй фазе. Для каждого плана оценивается предполагаемая стоимость выполнения запроса. При оценках используется статистическая информация о состоянии базы данных, доступная оптимизатору. Из полученных альтернативных планов выбирается наиболее дешевый, и его внутреннее представление теперь соответствует обрабатываемому запросу.

На **четвертом** этапе по внутреннему представлению наиболее оптимального плана выполнения запроса формируется выполняемое представление плана. Выполняемое представление плана может быть программой в машинных кодах, что не принципиально, поскольку четвертая фаза обработки запроса уже не связана с оптимизацией.

Наконец, на **пятом** этапе обработки запроса происходит его реальное выполнение. Это либо выполнение соответствующей подпрограммы, либо вызов интерпретатора с передачей ему для интерпретации выполняемого плана.

Стратегии оптимизации

Суть оптимизации заключается в поиске минимума функции стоимости от перестановки таблиц. Независимо от стратегии, оптимизатор обязан уметь анализировать стоимость для произвольной перестановки, в то время как сами перестановки для анализа предоставляются другим алгоритмом. Исследуемое множество перестановок может отличаться от всего пространства перестановок. Исходя из этого, обобщенный алгоритм работы оптимизатора можно записать так: «Перебор всех планов в поисках наилучшего»

Стратегия грубой силы

В теории, при использовании стратегии грубой силы оптимизатор запросов исследует все пространство перестановок всех исходных выбираемых таблиц и сравнивает суммарные оценки стоимости выполнения соединения для каждой перестановки. На практике, было предложено ограничить пространство исследования только левосторонними соединениями, чтобы при выполнении запроса одна из таблиц всегда была представлена образом на диске. Исследование нелевосторонних соединений имеет смысл если таблицы, входящие в соединения, расположены на более чем одном узле.

Для каждой таблицы в каждой из перестановок по статистике оценивается возможность использования индексов. Перестановка с минимальной оценкой и есть итоговый план выполнения запроса.

Синтаксическая оптимизация запросов

При классическом подходе к организации оптимизаторов запросов на этапе логической оптимизации производятся эквивалентные преобразования внутреннего представления запроса, которые "улучшают" начальное внутреннее представление в соответствии с фиксированными стратегиями оптимизатора. Характер "улучшений" связан со спецификой общей организации

оптимизатора, в частности, с особенностями процедуры поиска возможных процедурных планов запросов, выполняемой на третьей фазе обработки запроса.

Поэтому трудно привести полную характеристику и классификацию методов логической оптимизации.

Простые логические преобразования запросов

Очевидный класс логических преобразований запроса составляют преобразования предикатов, входящих в условие выборки, к каноническому представлению. Имеются в виду предикаты, содержащие операции сравнения простых значений.

Преобразования запросов с изменением порядка реляционных операций

В традиционных оптимизаторах распространены логические преобразования, связанные с изменением порядка выполнения реляционных операций. Примером соответствующего правила преобразования в терминах реляционной алгебры может быть следующее (А и В - имена отношений):

(A JOIN B) WHERE restriction-on-A AND restriction-on-B

эквивалентно выражению

A WHERE restriction-on-A) JOIN (B WHERE restriction-on-B).

Здесь JOIN обозначает реляционный оператор естественного соединения отношений; A WHERE restriction - оператор ограничения отношения А в соответствии с предикатом restriction.

Хотя немногие реляционные системы имеют языки запросов, основанные в чистом виде на реляционной алгебре, правила преобразований алгебраических выражений могут быть полезны и в других случаях. Довольно часто реляционная алгебра используется в качестве основы внутреннего представления запроса. Естественно, что после этого можно выполнять и алгебраические преобразования.

Разумное преобразование запроса на SQL к алгебраическому представлению сокращает пространство поиска планов выполнения запроса с гарантией того, что оптимальные планы потеряны не будут.

Приведение запросов со вложенными подзапросами к запросам с соединениями

Основным отличием языка SQL от языка реляционной алгебры является возможность использовать в логическом условии выборки предикаты, содержащие вложенные подзапросы. Глубина вложенности не ограничивается языком, т.е., вообще говоря, может быть произвольной. Предикаты с вложенными подзапросами при наличии общего синтаксиса могут обладать весьма различной семантикой. Единственным общим для всех возможных семантик вложенных подзапросов алгоритмом выполнения запроса является вычисление вложенного подзапроса

всякий раз при вычислении значения предиката. Поэтому естественно стремиться к такому преобразованию запроса, содержащего предикаты со вложенными подзапросами, которое сделает семантику подзапроса более явной, предоставив тем самым в дальнейшем оптимизатору возможность выбрать способ выполнения запроса, наиболее точно соответствующий семантике подзапроса.

Предикаты, допустимые в запросах языка SQL, можно разбить на следующие четыре группы:

- простые предикаты;
- предикаты со вложенными подзапросами;
- предикаты соединения;
- предикаты деления.

Простые предикаты. Это предикаты вида $R_i.C_k \text{ op } X$, где X - константа или список констант, и op - оператор скалярного сравнения ($=$, \neq , $>$, \geq , $<$, \leq) или оператор проверки вхождения во множество (IS IN , IS NOT IN).

Предикаты со вложенными подзапросами. Это предикаты вида $R_i.C_k \text{ op } Q$, где Q - блок запроса, а op может быть таким же, как для простых предикатов. Предикат может также иметь вид $Q \text{ op } R_i.C_k$. В этом случае оператор принадлежности ко множеству заменяется на CONTAINS или DOES NOT CONTAIN . Эти две формы симметричны. Достаточно рассматривать только одну.

Предикаты соединения. Это предикаты вида $R_i.C_k \text{ op } R_j.C_n$, где $R_i \neq R_j$ и op - оператор скалярного сравнения.

Предикаты деления. Это предикаты вида $Q_i \text{ op } Q_j$, где Q_i и Q_j - блоки запросов, а op может быть оператором скалярного сравнения или оператором проверки вхождения в множество.

Приведенная классификация является упрощением реальной ситуации в SQL. Не рассматриваются предикаты соединения общего вида, включающие арифметические выражения с полями более чем двух отношений.

Каноническим представлением запроса на n отношениях называется запрос, содержащий $n-1$ предикат соединения и не содержащий предикатов со вложенными подзапросами. Фактически, каноническая форма - это алгебраическое представление запроса.

Ниже приводятся два примера канонических форм запросов с предикатами разного типа. Соответствующая техника существует и для других видов предикатов.

```
SELECT  $R_i.C_k$  FROM  $R_i$  WHERE  $R_i.C_h \text{ IS IN}$ 
```

```
SELECT  $R_j.C_m$  FROM  $R_j$  WHERE  $R_i.C_n = R_j.C_p$ 
```

эквивалентно


```
SELECT Ri.Ck FROM Ri, Rj WHERE
```

```
Ri.Ch = Rj.Cm AND Ri.Cn = Rj.Cp
```

```
SELECT Ri.Ck FROM Ri WHERE Ri.Ch =
```

```
SELECT AVG (Rj.Cm) FROM Rj WHERE Rj.Cn = Ri.Cp
```

эквивалентно

```
SELECT Ri.Ck FROM Ri, Rt WHERE
```

```
Ri.Ch = Rt.Cm AND Ri.Cp = Rt.Cn
```

```
- Rt ( Cp, Cn ) = SELECT Rj.Cp, AVG (Rj.Cn) FROM Rj
```

```
GROUP BY Rj.Cp
```

Разумность таких преобразований обосновывается тем, что оптимизатор получает возможность выбора большего числа способов выполнения запросов. Часто открывающиеся после преобразований способы выполнения более эффективны, чем планы, используемые в традиционном оптимизаторе. При использовании в оптимизаторе запросов подобного подхода не обязательно производить формальные преобразования запросов.

Семантическая оптимизация запросов

Рассмотренные преобразования запросов основывались на семантике языка запросов, но в них не использовалась семантика базы данных, к которой адресуется запрос. Любое преобразование может быть произведено независимо от того, какая конкретная база данных имеется в виду. На самом же деле, при каждой истинно реляционной базе данных хранится и некоторая семантическая информация, определяющая, например, целостность базы данных.

Если говорить о базах данных, то эта информация хранится в системных каталогах базы данных в виде заданных ограничений целостности. Поскольку СУБД гарантирует целостность базы данных, то ограничения целостности можно рассматривать как аксиомы, в окружении которых формулируются запросы к базе данных.

Семантическая оптимизация запросов СУБД - процесс валидации и преобразования синтаксического дерева запроса в форму, пригодную для дальнейших шагов оптимизации.

На этой стадии выполняется:

а) преобразование запросов в каноническую форму:

1) раскрытие представлений;

2) преобразование подзапросов в соединения;

- 3) спуск предикатов;
- б) упрощение условий и распределение предикатов;
- в) преобразование дерева условий в пути выборки.

Преобразование запросов в каноническую форму

Запросы приводятся в каноническую форму, то есть переписываются так, чтобы они содержали минимальное количество операторов SELECT (соединение-фильтрация-проекция). Везде, где возможно, запрос должен быть приведен к форме единственного оператора SELECT. Это может позволить последующим фазам оптимизации сгенерировать значительно более эффективный (на 2-3 порядка) план выполнения для сложных запросов.

Раскрытие представлений

Раскрытие представлений применяется для того, чтобы итоговый запрос содержал ссылки только на материализованные отношения (таблицы) и было возможным использовать конвейерную обработку данных.

Преобразование подзапросов в соединения

Преобразование подзапросов в соединения необходимо для применения конвейерной обработки данных и минимизации объема результатов подзапросов, аккумулируемых во временной дисковой или в оперативной памяти.

Пример преобразования показан в таблице 1.

Таблица 1 - Пример преобразования

Исходный запрос	Результат
select distinct T.a from T where T.b in (select T1.b from T1 where T1.c < T.c)	select distinct T.a from T,T1 where T.b = T1.b and T1.c < T.c

Упрощение условий

Выполняется путем преобразования дерева логических операций в КНФ и упрощения полученной логической функции.

Преобразования дерева логических операций в КНФ выполняется следующим образом:

Для всех дизъюнкций, входящих в прямом виде, применяется распределительный закон:

$$P \text{ OR } (Q \text{ AND } R) = (P \text{ OR } Q) \text{ AND } (P \text{ OR } R)$$

$$(P \text{ AND } Q) \text{ OR } R = (P \text{ OR } R) \text{ AND } (Q \text{ OR } R)$$

Для всех дизъюнкций, входящих в инверсном виде, применяется правило де Моргана:

$$\text{NOT } (P \text{ OR } Q) = \text{NOT } P \text{ AND NOT } Q$$

Преобразование продолжается рекурсивно до тех пор, пока дерево не будет состоять из конъюнкций конститuent 0.

Полученная логическая функция находится в конъюнктивной нормальной форме, но избыточна. Для упрощения применяют методы оптимизации логических функций, такие как Эспрессо или Куайна-Мак-Класки.

Распределение предикатов

Распределение предикатов выполняется для всех предикатов вида:

$A.B \text{ pred } C$ для которых существует предикат $A.B = D.E$

В результате получаем предикат $D.B \text{ pred } C$

где C - константа; A, D - отношения; B, E - сравниваемые атрибуты. Данное упрощение выполняется на основе предположения, что исходный предикат $A.B \text{ pred } C$ может быть эффективней для отношения D .

Для каждого условия объединения вида:

$A.B \text{ pred } D.E$ генерируется обратное условие $D.E \text{ inversed pred } A.B$

для возможности выполнить соединение в обратном порядке.

Преобразование дерева условий в пути выборки

После упрощения дерева условий каждая конъюнкция в дереве представляет собой путь выборки. Предикаты внутри конъюнкций группируются по принадлежности к отношениям. Для получения итогового результата необходимо объединить результаты каждого из путей выборки.

Оценка стоимости плана запроса

После генерации множества планов выполнения запроса на основе разумных стратегий декомпозиции и эффективных стратегий выполнения элементарных операций нужно выбрать один план, в соответствии с которым будет происходить реальное выполнение запроса. При неверном выборе запрос будет выполнен неэффективно. Прежде всего необходимо определить, что понимается под эффективностью выполнения запроса. Это понятие неоднозначно и зависит от специфики операционной среды СУБД. В одних условиях можно считать, что эффективность выполнения запроса определяется временем его выполнения, т.е. реактивностью системы по отношению к обрабатываемым ею запросам. В других условиях определяющей является общая

пропускная способность системы по отношению к смеси параллельно выполняемых запросов. Тогда мерой эффективности запроса можно считать количество системных ресурсов, требуемых для его выполнения и т.д.

Следуя принятой терминологии, мы будем говорить о стоимости плана выполнения запроса, определяемой ресурсами процессора и устройств внешней памяти, которые расходуются при выполнении запроса.

Оценка числа извлекаемых строк

Оценка числа извлекаемых из таблицы строк используется для принятия решения о полном сканировании таблицы вместо доступа по индексу. Решение принимается на том основании, что каждое чтение листовой страницы индекса с диска влечет за собой 1 или более позиционирований и 1 или более чтений страниц таблицы. Поскольку индекс содержит ещё и нелистовые страницы, то извлечение более 0.1-1 % строк из таблицы, как правило, эффективней выполнять полным сканированием таблицы.

Более точная оценка получится на основе следующих показателей:

- число извлекаемых строк;
- средняя длина ключа в индексе;
- среднее число строк в странице индекса;
- длина страницы индекса;
- высота B^* -дерева в индексе;
- средняя длина строки в таблице;
- среднее число строк в странице таблицы;
- длина страницы таблицы.

СУБД старается организовать хранение блоков данных одной таблицы последовательно с целью исключить накладные расходы на позиционирование при полном сканировании (СУБД Oracle использует предварительное выделение дискового пространства для файлов данных). Эффективность полного сканирования так же увеличивается за счёт упреждающего чтения. При упреждающем чтении СУБД одновременно выдает внешней памяти команды чтения нескольких блоков. Сканирование начинается по завершении чтения любого из блоков. Одновременно продолжается чтение остальных блоков. Эффективность достигается за счёт параллелизма чтения и сканирования.

Статистика

Для оценки потенциального числа строк, извлекаемого из таблицы, РСУБД использует статистику. Статистика имеет вид гистограмм для каждой колонки таблицы, где по горизонтали располагается шкала значений, а высотой столбца отмечается оценка числа строк в процентах от общего числа строк (см. рис. 2).

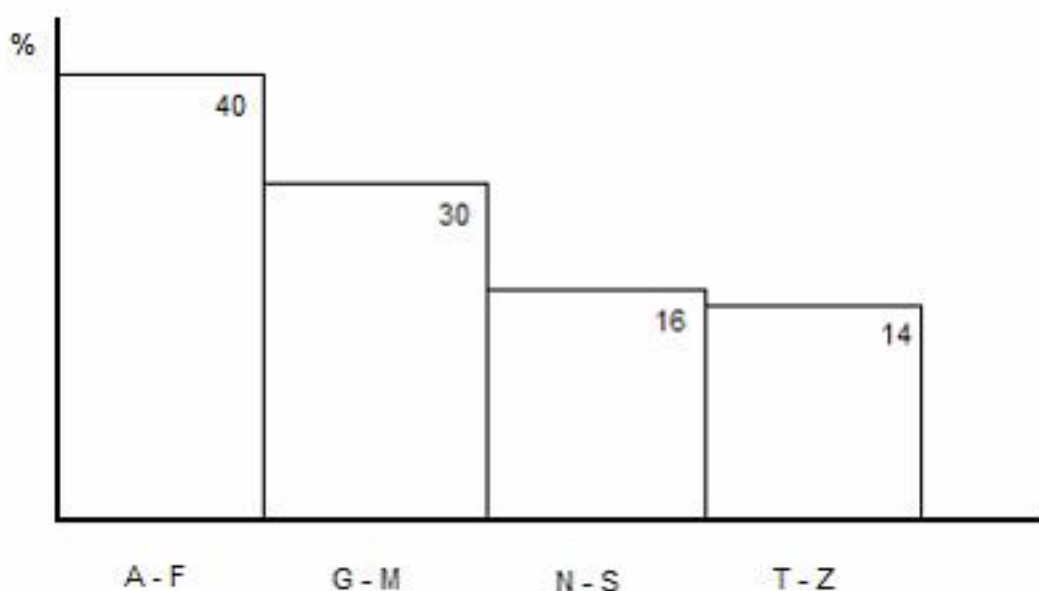


Рисунок 2 - Гистограмма статистики

Таким образом, если из таблицы извлекаются строки со значением колонки С с ограничением $[V1, V2]$, то можно оценить число строк, попадающих в этот интервал. Алгоритм оценки числа извлекаемых строк следующий:

- а) определить, в какие интервалы гистограммы попадает ограничение $[V1, V2]$;
- б) найти оценки числа строк R_i для каждого интервала i в процентах;
- в) если $[V1, V2]$ попадает в некоторый интервал $[S1, S2]$ частично или полностью лежит в интервале, то:
 - 1) найти пересечение $[V1, V2]$ и $[S1, S2]$;
 - 2) откорректировать число значений в частичном интервале (это либо $R_i * (V1 - S2 + 1) / (S1 - S2 + 1)$, либо $R_i * (S1 - V2 + 1) / (S1 - S2 + 1)$, либо $R_i * (V1 - V2 + 1) / (S1 - S2 + 1)$);
- г) иначе оценка для интервала равна R_i ;
- д) просуммировать оценки в процентах для всех интервалов;

е) перевести оценку в процентах в число строк.

Как правило, СУБД не знает и не может знать точное число строк в таблице (даже для выполнения запроса `SELECT COUNT(*) FROM TABLE` выполняется сканирование первичного индекса), поскольку в базе могут храниться одновременно несколько образов одной и той же таблицы с различным числом строк. Для оценки числа строк используются следующие данные:

- число страниц в таблице;
- длина страницы;
- средняя длина строки в таблице.

Статистика так же может храниться нарастающим итогом (см. рис. 3). В этом случае каждый интервал содержит суммарную оценку всех предыдущих интервалов плюс собственную оценку. Для получения оценки числа строк для ограничения $[V1, V2]$ достаточно из оценки интервала, в который попадает $V2$, вычесть оценку интервала, в который попадает $V1$.

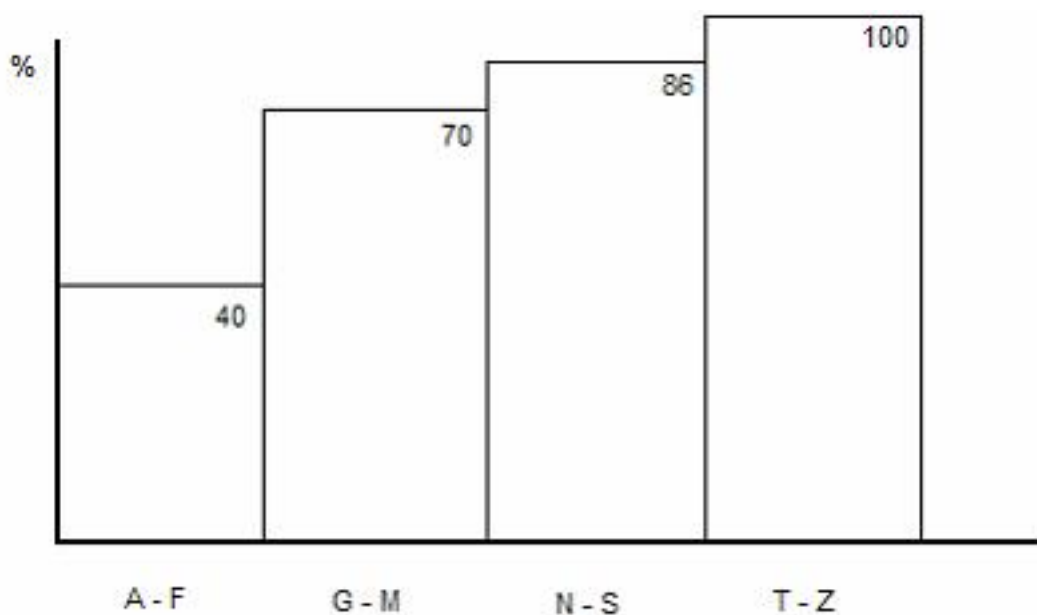


Рисунок 3 - Хранение статистики нарастающим итогом

Сбор статистики для построения гистограмм осуществляется либо специальными командами СУБД, либо фоновыми процессами СУБД. При этом, ввиду того, что база может содержать существенный объем данных, делается выборка меньшего объема из всей генеральной совокупности строк. Оценка репрезентативности (достоверности) выборки может осуществляться, например, по критерию согласия Колмогорова.

Если данные в таблице существенно изменяются в короткий промежуток времени, то статистика перестает быть актуальной и оптимизатор принимает неверные решения о полном сканировании таблиц. Режим работы базы данных должен быть спланирован таким образом, чтобы поддерживать актуальную статистику, либо не использовать оптимизацию на основе статистики.

Окружение 4-го поколения

Аббревиатура 4GL представляет собой сокращенный английский вариант написания термина *язык четвертого поколения* (Fourth-Generation Language).

Четкого определения этого понятия не существует, хотя, по сути, речь идет о некотором стенографическом варианте языка программирования. Если для организации некоторой операции с данными на языке третьего поколения (3GL) типа COBOL потребуется написать сотни строк кода, то для реализации этой же операции на языке четвертого поколения достаточно 10-20 строк

История языков программирования

С начала 70-х годов по настоящее время продолжается период языков четвертого поколения (4GL). После первых восторгов по поводу безграничных способностей ЭВМ стали более ясны возможности существующих языков программирования. Несмотря на рождение новых технологий (ООП, визуальное программирование, CASE-методологии, системный анализ), процесс создания больших программных комплексов оказался очень трудоемкой задачей, так как для реализации крупных проектов требовался более глобальный подход, чем тот, который предлагали имевшиеся средства разработки. Языки 4GL частично снимали эту проблему. Целью их создания было стремление увеличить скорость разработки проектов, снизить число ошибок и повысить общую надежность работы больших программных комплексов, получить возможность быстро и легко вносить изменения в готовые проекты, активно внедрять технологии визуальной разработки и т. д.

Все языки четвертого поколения интегрированы в мощные пользовательские оболочки и обладают простым и удобным интерфейсом. Они чаще всего используются для проектирования баз данных и работы с ними (встроенные языки СУБД), что объясняется возможностью формализации всех понятий, используемых при построении реляционных баз данных. Языки 4GL активно применяются в различных специализированных областях, где высоких результатов можно добиться, используя не универсальные, а проблемно-ориентированные языки, оперирующие конкретными понятиями узкой предметной области. Как правило, в эти языки встраиваются мощные примитивы, позволяющие в одном операторе описать такую функциональность, для реализации которой на языках младших поколений потребовались бы тысячи строк кода.

Становление и развитие языков программирования могут быть схематически отображены на рис. 4

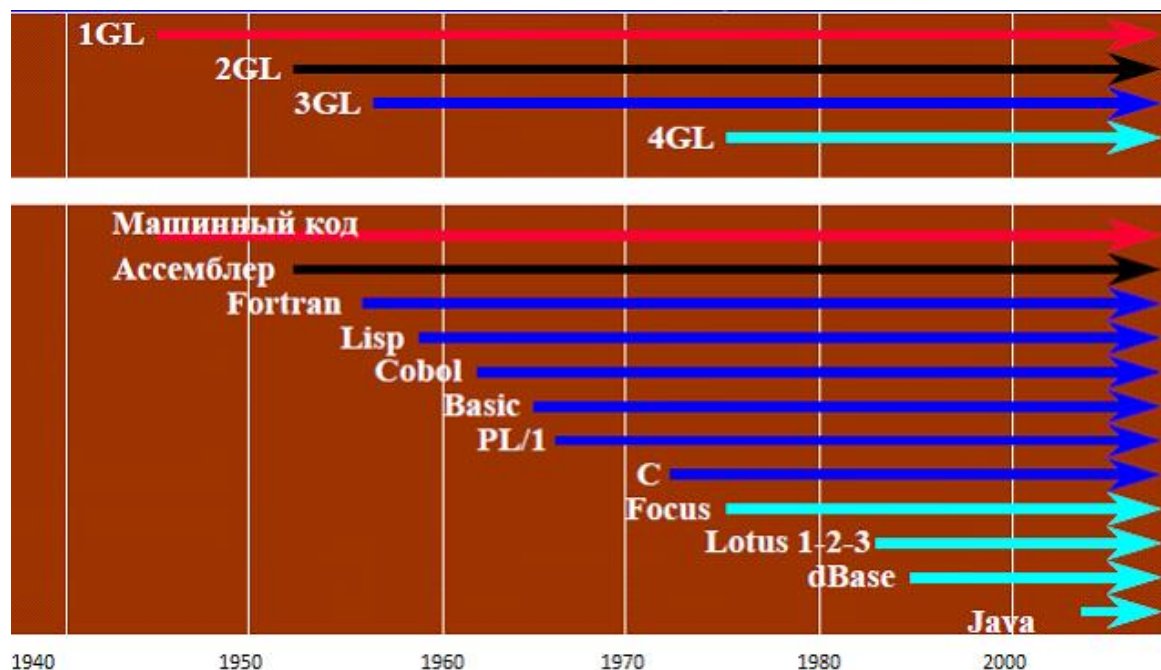


Рисунок 4 - История языков программирования

Понятие языка четвертого поколения

Языки четвертого поколения (4GL) отличаются от других языков программирования тем, что они:

- представляют собой широкий набор средств, ориентированных на проектирование и разработку программного обеспечения. Они строятся на основе оперирования не синтаксическими структурами языка и описания элементов, а представляющими их графическими образами;

- высокоуровневые языки, могут использовать английский язык или визуальные конструкции; алгоритмы и структуры данных обычно выбираются компилятором;

- созданные для конкретных задач языки программирования, позволяющие увеличить производительность разработки систем.

Выделяют следующие типы языков четвертого поколения:

- языки представления информации, например языки запросов или генераторы отчетов;

- специализированные языки, например языки электронных таблиц и баз данных;
- генераторы приложений, которые при создании приложений обеспечивают определение, вставку, обновление или извлечение сведений из базы данных;
- языки очень высокого уровня, предназначенные для генерации кода приложений.

Языки 4GL могут быть непосредственно встроены в сами СУБД, а могут существовать в виде отдельных сред программирования. В последнем случае в таких средах разрабатываются прикладные части информационных систем, реализующие только интерфейс и высокоуровневые функции по обработке данных. За низкоуровневым, как говорят, «сервисом» к данным такие прикладные системы обращаются к SQL-серверам, являющимися отдельными специализированными разновидностями СУБД. «Общение» между прикладными системами и SQL-серверами происходит соответственно на языке SQL.

Свои языки 4GL имеют практически все развитые профессиональные СУБД-Oracle, SyBase, Informix, Ingres, DB2, СУБД ЛИНТЕР. Кроме того, существующие CASE-средства автоматизированного проектирования - PowerBuilder фирмы PowerSoft, Oracle Designer фирмы Oracle, SQLWindows фирмы Gupta и др., также, как правило, имеют свои встроенные языки 4GL.

Преимущества 4GL:

- увеличение скорости разработки проектов;
- снижение числа ошибок и повышение общей надежности работы больших программных комплексов;
- возможность быстрого и легкого внесения изменений в готовые проекты;
- упрощение самих языков для конечного пользователя;
- активное внедрение технологий визуальной разработки;
- все средства разработки 4-го поколения имеют мощные интегрированные оболочки и обладают простым и удобным пользовательским интерфейсом;
- использование для проектирования баз данных и работы с ними (встроенные языки СУБД).

Недостатки 4GL:

- необходимо кодировать программу вручную, используя обычный процесс последовательного ввода команд;
- языки 4-го поколения (также как и предыдущих поколений) в значительной степени ориентированы на чуждую человеческому мышлению чисто компьютерную идеологию (работа с памятью, переменными, базами данных, последовательностями абстрактных операторов и т. п.), что требует от людей хорошего понимания принципов функционирования компьютера и

операционных систем;

- существование парадигмы функционального программирования, которая не позволяет перейти к более высокому уровню абстракций при разработке программных систем.

Процедурные расширения

Поскольку SQL не является «привычным» языком программирования (то есть не предоставляет средств для построения циклов, ветвлений и т. д.), вводимые разными производителями расширения касались в первую очередь процедурных расширений. Это хранимые процедуры (stored procedures) и процедурные языки-«надстройки». Практически в каждой СУБД применяется свой процедурный язык. Стандарт для процедурных расширений представлен спецификацией SQL/PSM. Перечень процедурных расширений для самых популярных СУБД приведён в таблице 2.

Таблица 2 - Перечень процедурных расширения SQL

СУБД	Краткое имя	Описание
Borland InterBase / Firebird	PSQL	Procedural SQL
IBM DB2	SQL PL	SQL Procedural Language (расширяет SQL/PSM); также в DB2 хранимые процедуры могут быть написаны на обычных языках программирования (C, Java и т.п.)
MicrosoftSQL Server / SybaseASE	Transact-SQL	Transact-SQL
MySQL	SQL/PSM	SQL/Persistent Stored Module (соответствует стандарту SQL:2003)
Oracle	PL/SQL	Procedural Language/SQL (основан на языке Ada)
PostgreSQL	PL/pgSQL	Procedural Language/PostgreSQL Structured Query Language (похож на Oracle PL/SQL)

Рассмотрим подробнее некоторые из возможностей перечисленных языков.

Язык PSQL

PSQL - Procedural SQL - разновидность, применяемая при создании хранимых процедур/триггеров/PSQL-блоков. Имеет управляющие структуры FOR, WHILE, IF.

Используя язык PSQL (процедурный SQL) Firebird, возможно создавать сложные хранимые процедуры для обработки данных полностью на стороне сервера. Для генерации отчётов особенно удобны хранимые процедуры с возможностью выборки, возвращающие данные в виде набора записей. Такие процедуры можно использовать в запросах точно так же как и обычные

таблицы. Также для каждой таблицы возможно назначение нескольких триггеров, срабатывающих до или после вставки, обновления или удаления записей. Для триггеров используется язык PSQL, позволяя вносить начальные значения, проверять целостность данных, вызывать исключения, и т. д. В Firebird 1.5 появились «универсальные» триггеры, позволяющие в одном триггере обрабатывать вставки, обновления и удаления записей таблицы

Язык SQL PL

Хранимые процедуры DB2 могут быть написаны с использованием C/C++, COBOL, Java (JDBC или SQLJ), .NET (языки, совместимые с CLR), и SQL Procedure Language (SQL PL). Для написания хранимых процедур разработчик может выбрать любой язык, однако наиболее часто используемыми языками являются Java и SQL PL.

Хранимые процедуры на SQL Procedure Language (SQL PL) рассматриваются как собственные, присущие среде DB2, поскольку они хранятся в базах данных DB2 для Windows или Linux как объекты. SQL PL развивался как единственный язык хранимых процедур на основе стандартов ANSI SQL. SQL PL поддерживается всеми представителями семейства серверов базы данных DB2 для Linux, UNIX, Windows, iSeries, и z/OS. DB2 Express-C предоставляет очень полезный графический инструмент, Development Center, для управления процессами разработки, тестирования и развертывания хранимых процедур SQL, включая процедуры Java и SQL PL. Поскольку выполнение хранимой процедуры может в деталях контролироваться DB2, она обеспечивает удобный метод обеспечения безопасности доступа пользователей к данным DB2.

Язык Transact-SQL

Transact-SQL (T-SQL) - процедурное расширение языка SQL компаний Microsoft (для Microsoft SQL Server) и Sybase (для Sybase ASE).

Язык T-SQL - это разновидность языка SQL, определяемого стандартом ANSI, которая применяется исключительно в СУБД Server SQL. Язык T-SQL совместим со стандартом ANSI 92 в минимальной конфигурации, а также включает целый ряд собственных расширений. В СУБД SQL Server в целях обеспечения обратной совместимости допускается использование множества различных вариантов синтаксиса, которые фактически не отличаются по своим возможностям от синтаксиса, соответствующего стандарту ANSI, но везде, где это возможно, следует использовать форму ANSI.

SQL был расширен такими дополнительными возможностями как:

- управляющие операторы;
- локальные и глобальные переменные;
- различные дополнительные функции для обработки строк, дат, математики и т. п.;

- поддержка аутентификации Microsoft Windows.

Язык Transact-SQL является ключом к использованию MS SQL Server. Все приложения, взаимодействующие с экземпляром MS SQL Server, независимо от их реализации и пользовательского интерфейса, отправляют серверу инструкции Transact-SQL.

В Transact-SQL существуют специальные команды, которые позволяют управлять потоком выполнения сценария, прерывая его или направляя в нужную логику.

К управляющим структурам языка относят:

- блок группировки - структура, объединяющая список выражений в один логический блок (BEGIN ... END);
- блок условия - структура, проверяющая выполнения определенного условия (IF ... ELSE);
- блок цикла - структура, организующая повторение выполнения логического блока (WHILE ... BREAK ... CONTINUE);
- переход - команда, выполняющая переход потока выполнения сценария на указанную метку (GOTO);
- задержка - команда, задерживающая выполнение сценария (WAITFOR);
- вызов ошибки - команда, генерирующая ошибку выполнения сценария (RAISERROR). Спецификация Transact-SQL значительно расширяет стандартные возможности SQL благодаря большому набору операторов, встроенным функциям и выражениям

Язык SQL/PSM

SQL/PSM - определяет процедурные расширения языка SQL.

Этот уровень соответствия полностью описан в документе SQL/PSM стандарта SQL99. Так, язык SQL расширяется операторами управления CASE, IF, WHILE, REPEAT, LOOP и FOR. Вводится поддержка процедур и функций, создаваемых операторами CREATE PROCEDURE и CREATE FUNCTION. В язык SQL введено использование переменных и применение обработчиков ошибок.

До появления SQL/PSM переменные в SQL использовать было нельзя. С появлением SQL/PSM SQL наконец-то получил возможность, которую всегда имели даже самые примитивные процедурные языки, - возможность присваивания значения переменной. SQL/PSM получил в свое распоряжение аналогичные управляющие структуры, позволяя тем самым решать многие задачи без привлечения других языков программирования.

Хранимые процедуры (stored procedures) находятся на сервере баз данных, а не на компьютере пользователя (до появления SQL/PSM). Хранимая процедура должна быть

определена, после чего ее можно вызвать с помощью команды CALL. Хранение процедуры на сервере уменьшает сетевой обмен и повышает производительность. Команда CALL является единственным сообщением, передаваемым от пользователя к серверу. Также SQL/PSM добавляет к уже существующим еще один вид полномочий - полномочия на выполнение.

Язык PL/SQL

PL/SQL - процедурный язык, разработанный фирмой Oracle для написания хранимых в БД подпрограмм. PL/SQL обеспечивает общую основу процедурного программирования как в клиентских приложениях, так и на стороне сервера, в том числе хранимых на сервере подпрограмм, пакетов и триггеров базы данных.

PL/SQL - это полностью переносимый, высокопроизводительный язык, предлагающий следующие преимущества:

- поддержку SQL;
- повышение продуктивности разработки ;
- улучшение производительности выполнения ;
- переносимость ;
- интеграцию с ORACLE .

Поддержка SQL. SQL стал признанным языком баз данных благодаря своей гибкости, мощи и простоте изучения. Небольшое число предложений, напоминающих естественный английский язык, позволяет легко манипулировать данными, хранящимися в реляционной базе данных. PL/SQL позволяет вам использовать как все предложения манипулирования данными языка SQL, команды управления курсорами и транзакциями, так и все функции, операторы и псевдосто́лбцы SQL. Таким образом, вы имеете возможность гибко и безопасно манипулировать данными ORACLE.

Улучшенная продуктивность. PL/SQL придает дополнительную функциональность непроцедурным инструментам, таким как SQL*Forms, SQL*Menu и SQL*ReportWriter. Когда PL/SQL встроен в эти инструменты, разработчики программного обеспечения могут использовать привычные конструкции процедурных языков при написании приложений.

Например, можно ввести целый блок PL/SQL как процедуру или функцию. Таким образом, улучшенный инструментарий в руках разработчиков повышает продуктивность разработки. Более того, PL/SQL один и тот же в любом окружении. Поэтому, освоив PL/SQL с одним инструментом, разработчики могут улучшить свою продуктивность во всех прочих инструментах, поддерживающих PL/SQL.

Улучшенная производительность. Без PL/SQL система ORACLE должна обрабатывать

предложения SQL по одному за раз. Каждое предложение SQL приводит к очередному обращению к ORACLE и дополнительным накладным расходам. Эти накладные расходы могут стать существенными, когда вы выдаете много предложений SQL в сетевой среде. Каждое выдаваемое предложение SQL должно быть послано по сети, утяжеляя сетевой трафик. При PL/SQL, однако, целый блок предложений может быть послан в ORACLE за один раз. Это позволяет радикально сократить общение между приложением и ORACLE.

PL/SQL повышает производительность. PL/SQL может также взаимодействовать с инструментами разработки приложений Oracle, такими как SQL*Forms, SQL*Menu и SQL*ReportWriter. Добавляя мощь процедурной обработки в такие инструменты, PL/SQL увеличивает их производительность. При помощи PL/SQL инструмент способен выполнять все вычисления с данными быстро и эффективно, не обращаясь к ORACLE. Это экономит время, а в сетевом окружении уменьшает сетевой трафик.

Переносимость. Приложения, написанные на PL/SQL, переносимы на любое оборудование и в среду любой операционной системы, на которых выполняется ORACLE. Иными словами, программы PL/SQL могут выполняться всюду, где может выполняться ORACLE; вам не требуется перенастраивать их на каждое новое окружение. Это значит, что вы можете разрабатывать библиотеки переносимых программ, которые можно использовать в различных окружениях.

Интеграция с ORACLE. Как PL/SQL, так и ORACLE основываются на SQL. Более того, PL/SQL поддерживает все типы данных SQL. В сочетании с прямым доступом к ORACLE, который обеспечивает SQL, эти объявления естественных для ORACLE типов данных интегрируют PL/SQL со словарем данных ORACLE.

Модульность. Модульность позволяет вам разбивать приложение на управляемые, хорошо определенные логические модули. Путем последовательного уточнения вы можете свести комплексную проблему к множеству простых проблем, имеющих легко реализуемые решения. PL/SQL предлагает для этой цели конструкции, называемые ПРОГРАММНЫМИ ЕДИНИЦАМИ. Помимо таких программных единиц, как блоки и подпрограммы, предусматривается специальный конструкт ПАКЕТ, который позволяет вам группировать взаимосвязанные объекты программы в единицы большего размера.

Язык программирования 4-го поколения 4GL Informix

Включает следующие возможности:

- средства стандартных языков программирования 3-го поколения;
- программные переменные;
- изготовление операторов по ходу программы;
- операторы манипуляции базой данных. SQL;

- переброска данных из базы данных в программные переменные, и обратно;
- курсоры;
- печать результатов запросов, формирование отчетов;
- экранный обмен с пользователем;
- меню, окна;
- экранные формы, экранные поля, экранные массивы;
- файл описания экранной формы;
- файлы с исходными текстами;
- описание состава программы.

Язык DataFlex

DataFlex - это объектно-ориентированный язык программирования 4-го поколения и система управления базами данных. Он позволяет создавать надёжные, масштабируемые, переносимые и производительные приложения. Сохраняя преемственность поколений, он предлагает ту же методологию и структуру построения прикладных программ, которая используется в Visual DataFlex.

DataFlex является полностью переносимой средой разработки. Приложения, работающие в текстовом режиме, могут быть созданы для Microsoft Windows, GNU/Linux и основных UNIX-систем. Код приложения может легко переноситься в любую из поддерживаемых платформ без каких-либо дополнительных операций, за исключением перекомпиляции.

DataFlex 3.2 в комплексе с Visual DataFlex может предложить полностью переносимые решения для большинства наиболее популярных сред - Microsoft Windows, Unix.

Программы, созданные на Visual DataFlex используются более чем на 450 тысячах предприятий и организаций, работающих в области здравоохранения, транспорта, промышленного производства, экономики и финансов, дистрибуции, страхования, сельского хозяйства, армии, полиции и многих других.

Поддерживаемые операционные системы

- GNU/Linux;
- UNIX;
- консольный режим Windows;

Поддерживаемые СУБД. DataFlex имеет встроенную поддержку четырёх промышленных СУБД:

- Oracle;
- Microsoft SQL Server;
- IBM DB2;
- Pervasive PSQL;

- а также любые СУБД по стандарту ODBC. От сторонних разработчиков доступны драйвера данных для PostgreSQL и MySQL.

Преимущества DataFlex:

- встроенная система управления базами данных с поддержкой транзакций и высокой производительностью;

- база данных DataFlex широко известна в мире благодаря своей скорости, надёжности и простоте администрирования. Сотни пользователей одновременно и миллионы записей в базе данных не составляют никаких проблем для данной СУБД;

- возможность использования клиент-серверной архитектуры DataFlex поддерживает использование драйверов баз данных для MS SQL, IBM DB2 и ODBC в консольном режиме Windows. DB2 также поддерживается на GNU/Linux. Драйвера от сторонних производителей дают возможность использовать Oracle и MySQL;

- поддержка очень больших приложений DataFlex может поддерживать до 4095 таблиц базы данных в каждом отдельном приложении.

Язык Progress 4GL

Progress 4GL - это высокоуровневый язык программирования четвёртого поколения (4GL), разработанный в Progress Software Corporation.

Progress 4GL является высокоуровневым, процедурным языком разработки приложений, который позволяет удовлетворять всем требованиям, предъявляемым к современным приложениям, в то же время уменьшая сложность и повышая производительность их разработки.

4GL содержит все необходимые программные конструкции для решения самых различных аспектов программирования сложных приложений без необходимости прибегать к менее эффективным и менее переносимым языкам третьего поколения. Кроме этого, 4GL обеспечивает поддержку и переход между тремя основными принципами программирования: структурированным, событийно-управляемым и объектно-ориентированным, - от Вас не требуется осваивать новые принципы программирования для того, чтобы успешно работать с

PROGRESS. Для завершения процесса разработки промышленного приложения Вам потребуются средства разработки не только логики взаимодействия с пользователем, но также потребуются средства для решения таких важных задач, как:

- автоматический контроль транзакций и блокирование записей;
- получение и обработка информации из баз данных;
- сложные вычисления и обработка данных;
- пакетная обработка;
- генерация отчетов;
- целостность базы данных и требования безопасности;
- поддержка двухбайтовых кодировок.

Язык 4GL содержит все функции и операторы, необходимые для удовлетворения вышеперечисленных требований. Но, в отличие от остальных инструментальных средств, менее ориентированных на разработку приложения в архитектуре клиент/сервер, PROGRESS не требует использования различных языков программирования для отдельного программирования обработки данных на клиенте, серверных процессов и пакетной обработки на сервере. Всё это уменьшает стоимость затрат по изучению языка и продолжению разработки.

Используется в СУБД Progress. Кроме традиционных SQL-запросов, реализация поддержки которых не очень удобна, позволяет использовать выражения FOR EACH, FIND, FIND FIRST. Кроме того, существует возможность сокращённого написания операторов.

При этом очевидны недостатки:

- язык SQL служит для описания запросов, а не для их программирования, в то время как на 4GL приходится именно программировать запросы;
- многие вещи, с которыми СУБД типа Oracle или MS SQL Server прекрасно справляются в автоматическом режиме, в Progress приходится прописывать явно;
- не слишком удобные и не богатые средства для манипуляции и визуализации данных;
- Progress имеет всего три типа блокировок и не слишком интеллектуально работает с индексами.

Преимущество, заключающееся в том, что помимо средств манипуляции данными в языке содержатся средства для создания интерфейса немного сомнительно, так как приводит к созданию однозвенного приложения.

В связи с этим ни язык 4GL, ни СУБД Progress не получили широкое распространение.

В заключение следует отметить, что в последнее время наметилась тенденция встраивания

развитых языков уровня 4GL и в СУБД для конечных пользователей.

Язык PL / SQL. Общие положения

Введение

Даже, если вы не очень опытный программист, вы вероятно отметили, что сам по себе SQL не очень полезен при написании программ. Самое очевидное ограничение - это то, что SQL в основном выполняет по одной команде в каждый момент времени. Типы логических конструкций типа if ... then ("если ... то"), for ... do ("для ... выполнить") и while ... repeat("пока ... повторять") - используемых для структур большинства компьютерных программ, здесь отсутствуют.

Кроме того, интерактивный SQL не может делать многого со значениями, кроме ввода их в таблицу, размещения или распределения их с помощью запросов, и конечно вывода их на какое-то устройство. Более традиционные языки, однако, сильны именно в этих областях. Они разработаны так чтобы программист мог начинать обработку данных, и основываясь на ее результатах, решать, делать ли это действие или другое, или же повторять действие до тех пока не встретится некоторое условие, создавая логические маршруты и циклы. Значения сохраняются в переменных, которые могут использоваться и изменяться с помощью любого числа команд. Это дает вам возможность указывать пользователям на ввод или вывод этих команд из файла, и возможность форматировать вывод сложными способами.

Цель вложенного SQL состоит в том, чтобы объединить эти возможности, позволяющие вам создавать сложные процедурные программы, которые адресуют базу данных посредством SQL - позволяя вам устранить сложные действия в таблицах на процедурном языке, который не ориентирован на такую структуру данных, в тоже время поддерживая структурную строгость процедурного языка.

Эта тема представляет собой общий обзор языка PL/SQL, процедурного расширения SQL фирмы Oracle и включает обсуждение следующих вопросов:

- что из себя представляет язык с PL / SQL ;
- как выполняется PL/SQL;
- типы данных PL/SQL.

Возможности языка PL/SQL

PL/SQL не только позволяет вам вставлять, удалять, обновлять и извлекать данные ORACLE и управлять потоком предложений для обработки этих данных. Более того, вы можете объявлять константы и переменные, определять подпрограммы (процедуры и функции) и перехватывать ошибки времени выполнения. Таким образом, PL/SQL комбинирует мощь манипулирования

данными SQL с мощностью обработки данных процедурных языков.

PL/SQL имеет два типа подпрограмм - ПРОЦЕДУРЫ и ФУНКЦИИ; всем им можно передавать параметры при вызове.

PL/SQL позволяет вам объединять логически связанные типы, программные объекты и подпрограммы в ПАКЕТ. Каждый пакет легко понять, а интерфейсы между пакетами просты, ясны и хорошо определены. Это облегчает разработку приложений. Если у вас есть Процедурное расширение базы данных, то пакеты можно компилировать и сохранять в базе данных ORACLE, где их содержимое может совместно использоваться многими приложениями. Когда вы первый раз обращаетесь к пакетированной подпрограмме, в память загружается весь пакет. Поэтому последующие обращения к другим подпрограммам пакета не требуют дисковых операций. Таким образом, пакеты могут повысить продуктивность разработки и улучшить производительность выполнения.

Если у вас есть процедурное расширение базы данных, то именованные блоки (подпрограммы) PL/SQL можно компилировать отдельно и передавать на постоянное хранение в базу данных ORACLE, где они готовы к исполнению. Подпрограмма, явно созданная (посредством CREATE) с помощью инструмента ORACLE, называется ХРАНИМОЙ подпрограммой. Однажды откомпилированная и сохраненная в словаре данных, она является объектом базы данных, к которому можно обращаться из любого приложения, соединенного с этой базой данных.

Хранимые подпрограммы, определенные внутри пакета, называются ПАКЕТИРОВАННЫМИ подпрограммами; подпрограммы, определенные вне пакета, называются НЕЗАВИСИМЫМИ подпрограммами. (Подпрограммы, определенные внутри других подпрограмм или внутри блока PL/SQL, называются ЛОКАЛЬНЫМИ подпрограммами. Такие подпрограммы недоступны для других приложений, и существуют лишь для удобства окружающего блока.)

Хранимые подпрограммы повышают продуктивность, улучшают производительность, экономят память, обеспечивают лучшую целостность и безопасность. Например, проектируя приложения на базе библиотеки хранимых процедур и функций, вы избежите повторов в кодировании и увеличите вашу продуктивность.

Подпрограммы хранятся в синтаксически разобранной, откомпилированной форме. Поэтому при вызове они загружаются и передаются процессору PL/SQL немедленно. Более того, хранимые подпрограммы используют преимущества разделяемой памяти в ORACLE. Лишь одна копия подпрограммы должна быть загружена в память, чтобы быть доступной многим пользователям.

Архитектура

Исполнительная система PL/SQL - это технология, а не независимый продукт. Рассматривайте эту технологию как процессор, который исполняет блоки и подпрограммы PL/SQL. Этот процессор может быть установлен в сервере ORACLE или в инструменте

разработки приложений, таком как SQL*Forms, SQL*Menu или SQL*ReportWriter. Таким образом, PL/SQL может располагаться в двух окружениях:

- в сервере ORACLE;
- в инструментах ORACLE.

Эти два окружения независимы. PL/SQL может быть доступен в сервере, но недоступен в инструментах, или наоборот. В любом окружении, процессор PL/SQL принимает как ввод любой действительный блок или подпрограмму PL/SQL.

Инструменты разработки приложений, в которых нет локального процессора PL/SQL, вынуждены обращаться к ORACLE для обработки блоков и подпрограмм PL/SQL. Сервер ORACLE, когда он содержит процессор PL/SQL, может обрабатывать как блоки и подпрограммы PL/SQL, так и одиночные предложения SQL. Сервер передает блоки и подпрограммы своему локальному процессору PL/SQL.

Инструмент разработки приложений может обрабатывать блоки PL/SQL, если он содержит процессор PL/SQL. Инструмент передает эти блоки своему локальному процессору PL/SQL. Процессор PL/SQL исполняет все процедурные предложения на стороне приложения, а в ORACLE посылает лишь предложения SQL. Таким образом, большая часть работы выполняется на стороне приложения, а не на стороне сервера.

Более того, если блок не содержит предложений SQL, то процессор исполняет весь блок на стороне приложения. Это полезно, если ваше приложение выгадывает за счет условного и итеративного управления.

Как исполняется PL/SQL

Процессор PL/SQL (engine) - это специальная компонента многих продуктов ORACLE, в том числе сервера ORACLE, которая занимается обработкой PL/SQL.

Процедура (или пакет) хранится в базе данных. Когда процедура, хранящаяся в базе данных, вызывается из приложения, откомпилированная процедура (или пакет, в котором она содержится) загружается в разделяемый пул в области SGA, и исполнители PL/SQL и SQL совместно обрабатывают предложения в этой процедуре.

Процессор PL/SQL встроен в следующие продукты Oracle:

- Сервер ORACLE с процедурной опцией;
- SQL*Forms (версия 3 и позже);
- SQL*Menu (версия 5 и позже);
- SQL*ReportWriter (версия 2 и позже);

- Oracle Graphics (версия 2 и позже);

Если вы имеете ORACLE с процедурной опцией, то вы можете вызывать хранимую процедуру из другого блока PL/SQL, который может быть как анонимным блоком, так и другой хранимой процедурой. Кроме того, вы можете передавать ORACLE анонимные блоки из приложений, разработанных с помощью следующих инструментов:

- Прекомпиляторов ORACLE (включая пользовательские выходы);
- Интерфейсов вызова ORACLE (OCI) * SQL*Plus * SQL*DBA.

Переменные и константы. Кодовое множество

Общие положения

PL/SQL - это язык, структурированный блоками. Это значит, что основные единицы (процедуры, функции и анонимные блоки), составляющие программу PL/SQL, являются логическими БЛОКАМИ, которые могут содержать любое число вложенных в них подблоков. Обычно каждый логический блок соответствует некоторой проблеме или подпроблеме, которую он решает.

Блок (или подблок) позволяет вам группировать логически связанные объявления и предложения. Благодаря этому вы можете размещать объявления близко к тем местам, где они используются. Объявления локальны в блоке, и перестают существовать, когда блок завершается.

Как показывает рисунок 5, блок PL/SQL имеет три части: декларативную часть, исполняемую часть и часть обработки исключений. (ИСКЛЮЧЕНИЕМ в PL/SQL называется условие, вызывающее предупреждение или ошибку.) Исполняемая часть обязательна; две остальные части блока могут отсутствовать.

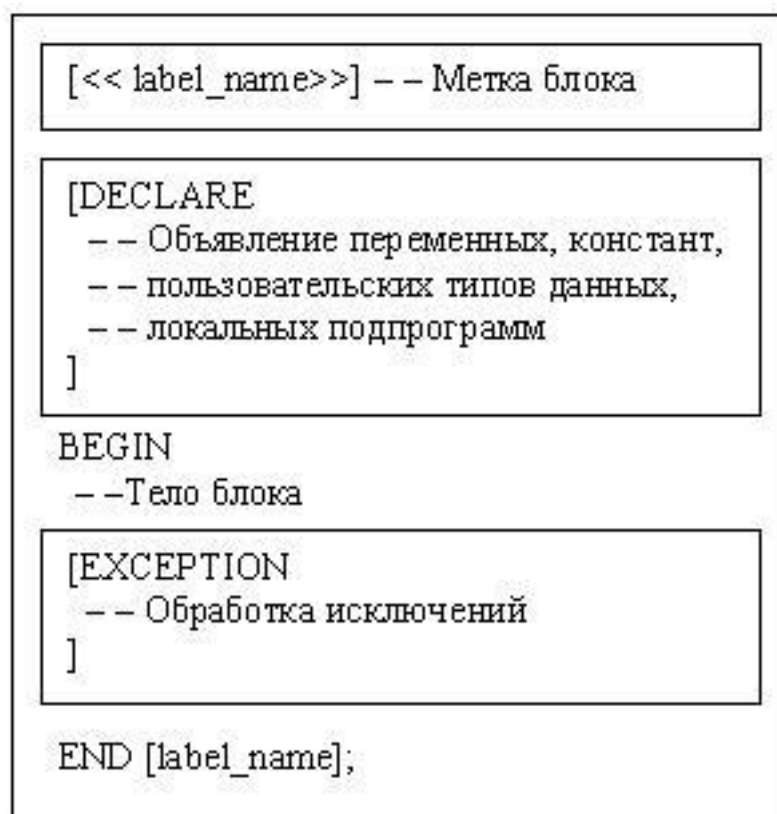


Рисунок 5 - Структура блока PL/SQL

Порядок частей блока логичен. Блок начинается с декларативной части, в которой объявляются объекты. С объявленными объектами осуществляются манипуляции в исполнительной части. Исключения, возбуждаемые во время исполнения, могут быть обработаны в части обработки исключений.

Каждый блок может содержать другие блоки; иными словами, блоки могут быть вложены друг в друга. Вложенный блок называется подблоком; он вложен в окружающий блок. Вы можете вкладывать блоки в исполнительную часть или части обработки исключений блока PL/SQL, но не в декларативной части. Кроме того, вы можете определять локальные подпрограммы в декларативной части любого блока. Однако вызывать локальные подпрограммы можно только из того блока, в котором они определены.

Переменные и константы

PL/SQL позволяет вам объявить переменные и константы, а затем использовать их в SQL и процедурных предложениях в любом месте, где допускается использование выражения. Однако ссылки вперед не допускаются. Таким образом, вы должны объявить переменную или константу прежде, чем сможете сослаться на нее в других предложениях, в том числе в других

объявлениях.

Объявления переменных

Ваша программа хранит значения в переменных и константах. Во время выполнения программы значения переменных могут изменяться, а значения констант не могут.

Вы можете объявлять переменные и константы в декларативной части любого блока PL/SQL, подпрограммы или пакета. Объявление распределяет место для значения, специфицирует его тип данных и задает имя, по которому можно обращаться к этому значению. Объявление может также присвоить начальное значение и специфицировать ограничение NOT NULL.

Примеры :

```
Begin_time DATE;  
Storage_id SMALLINT := 0;  
name VARCHAR2(20) NOT NULL := 'picture';
```

Первое объявление именует переменную типа DATE. Второе объявление именует переменную типа SMALLINT и использует оператор присваивания (:=), чтобы присвоить этой переменной нулевое начальное значение. Третье объявление именует переменную типа VARCHAR2, специфицирует для нее ограничение NOT NULL и присваивает ей начальное значение 'AP001'.

Нельзя присваивать значения NULL переменным или константам, объявленным как NOT NULL. Если вы попытаетесь это сделать, будет возбуждено предопределенное исключение VALUE_ERROR. За ограничением NOT NULL должна следовать фраза инициализации; в противном случае вы получите ошибку компиляции.

Например, следующее объявление незаконно:

```
name VARCHAR2(20) NOT NULL; -- нет начального значения
```

Как показывают следующие примеры, инициализирующее выражение может быть сколь угодно сложным и может ссылаться на ранее инициализированные переменные и константы:

```
pi CONSTANT REAL := 3.14159;  
radius REAL := 1;  
area REAL := pi * radius**2;
```

В объявлениях констант зарезервированное слово CONSTANT должно предшествовать спецификатору типа, как показывает следующий пример:

```
Price_max CONSTANT REAL := 5000.00;
```

Использование DEFAULT

Если хотите, вы можете использовать зарезервированное слово DEFAULT вместо оператора присваивания, чтобы инициализировать переменную или константу:

```
Storage_id SMALLINT DEFAULT 15;  
valid BOOLEAN DEFAULT FALSE;
```

Можно также использовать DEFAULT для инициализации параметров подпрограмм, параметров курсоров и полей в пользовательских записях.

Объявляемая переменная может иметь любой тип данных, присущий SQL, такой как NUMBER, CHAR и DATE, или присущий PL/SQL, такой как BOOLEAN или BINARY_INTEGER. Например, предположим, что вы хотите объявить переменную с именем num_var так, чтобы она могла хранить 4-значные числовые значения, и переменную с именем bool, которая может принимать булевские значения TRUE или FALSE. Вы объявляете эти переменные так:

```
Goods_id NUMBER(4);  
bool BOOLEAN;
```

Вы можете также объявлять записи и таблицы PL/SQL, используя составные типы данных PL/SQL: RECORD и TABLE.

Присваивания переменным значений

Вы можете присваивать переменным значения двумя способами. Первый способ использует оператор присваивания := (двоеточие, за которым следует знак равенства). Слева от оператора присваивания кодируется имя переменной, а справа - выражение.

Примеры правильных присваиваний :

```
summa:= price * quantity;  
new_price:= price * 0.10;  
str1 := SUBSTR('adcbnn', 2, 4);  
val := FALSE;
```

Второй способ присвоить значение переменной - это извлечь в нее значение из базы данных посредством фразы INTO предложения SELECT или FETCH. Например, вы можете заставить ORACLE вычислить 10% повышение цены товара при извлечении его из таблицы:

```
SELECT price* 0.10 INTO new_price FROM goods WHERE good_id=5;
```

После этого значение переменной new_price можно использовать в других вычислениях, либо вставить его в таблицу базы данных.

Объявления констант

Объявление константы аналогично объявлению переменной, с той разницей, что вы должны добавить ключевое слово **CONSTANT** и немедленно присвоить константе значение. Впоследствии никакие присваивания константе не допускаются. В следующем примере вы объявляете константу с именем `min_balance`:

```
min _ balance CONSTANT REAL := 10.00;
```

Атрибуты

Переменные и константы PL/SQL имеют АТРИБУТЫ, т.е. свойства, позволяющие вам ссылаться на тип данных и структуру объекта, не повторяя его объявление. Аналогичные атрибуты имеются у таблиц и столбцов базы данных, что позволяет вам упростить объявления переменных и констант.

Атрибут %TYPE

Атрибут **%TYPE** представляет тип данных переменной, константы или столбца базы данных. В следующем примере, **%TYPE** представляет тип данных переменной:

```
credit REAL(7,2);  
debit credit%TYPE;
```

Переменные и константы, объявленные с атрибутом **%TYPE**, трактуются так, как если бы они были объявлены с явным типом данных. Например, в примере выше PL/SQL рассматривает переменную `debit` как переменную типа `REAL(7,2)`.

Следующий пример показывает, что объявление через **%TYPE** может включать фразу инициализации:

```
balance NUMBER(7,2);  
minimum_balance balance%TYPE := 10.00;
```

Атрибут **%TYPE** особенно полезен при объявлении переменных, которые ссылаются на столбцы базы данных. Вы можете ссылаться на таблицу и столбец, или указывать также и владельца таблицы, как показывает следующий пример:

```
Volume1 scott.goods.volume%TYPE;
```

Использование атрибута **%TYPE** при объявлении `volume 1` имеет два преимущества. Во-первых, вы не обязаны знать точный тип столбца `volume`. Во-вторых, если определение столбца `volume` изменится, то тип данных переменной `volume 1` изменится соответственно во

время выполнения.

Например, таблица `goods` содержит столбец с именем `name`. Чтобы дать переменной `name_new` тот же тип данных, что у столбца `name`, не зная точного определения этого столбца в базе данных, объявите `name_new` с использованием атрибута `%TYPE`:

```
name_new goods.name%TYPE;
```

Атрибут `%ROWTYPE`

Атрибут `%ROWTYPE` возвращает тип записи, представляющей строку в таблице (или обзоре). Такая запись может содержать целую строку данных, выбранных из таблицы или извлеченных курсором. В следующем примере вы объявляете запись, которая хранит строку, выбранную из таблицы `goods`.

```
DECLARE goods_rec goods%ROWTYPE;
```

Столбцы в строке таблицы и соответствующие поля в записи имеют одинаковые имена и типы данных. В следующем примере вы выбираете значения столбцов в запись с именем `goods_rec`:

```
DECLARE  
goods_rec goods%ROWTYPE; ...  
BEGIN  
SELECT * INTO goods_rec FROM goods WHERE ...  
...  
END;
```

Значения столбцов, возвращаемые предложением `SELECT`, размещаются в индивидуальных полях записи. Вы обращаетесь к конкретному полю, используя квалифицированные ссылки. Например, вы могли бы обратиться к полю `goods_id` следующим образом:

```
IF goods_rec.goods_id = 20 THEN ...
```

Кроме того, вы можете присваивать значение выражения PL/SQL конкретному полю, как показывают следующие примеры:

```
goods_rec.name := 'books';  
goods_rec.price:= goods_rec.price * 1.15;
```

Нельзя включать выражений инициализации в объявления тех переменных, которые используют `%ROWTYPE`. Тем не менее, есть способ присвоить значения сразу всем полям записи - вы можете присвоить записи список значений столбцов, используя предложения `SELECT...INTO` или `FETCH...INTO` (работа с курсорами будет рассмотрена далее)

Соглашения об именах

Одни и те же соглашения об именах действительны для всех программных объектов и единиц PL/SQL, включая константы, переменные, курсоры, исключения, процедуры, функции и пакеты. Имена могут быть простыми, квалифицированными, удаленными или квалифицированными удаленными.

Например, вы можете обращаться к процедуре с именем `insert _ goods` любым из следующих способов:

`insert _ goods (...);` -- простое

`pac _ goods . insert _ goods (...);` -- квалифицированное

`insert _ goods @ newyork (...);` -- удаленное

`pac _ goods . insert _ goods @ newyork (...);` -- квалифиц. удаленное

В первом случае вы просто указываете имя процедуры. Во втором случае вы должны квалифицировать имя процедуры именем пакета, потому что процедура хранится в пакете с именем `pac _ goods`. В третьем случае вы обращаетесь к связи баз данных `newyork`, потому что (независимая) процедура находится на удаленной базе данных. В четвертом случае вы квалифицируете имя.

Сфера и видимость

Ссылки на идентификатор разрешаются согласно его сфере и видимости. СФЕРА идентификатора - это та область программной единицы (блока, подпрограммы или пакета), из которой вы можете сослаться на этот идентификатор. Идентификатор называется ВИДИМЫМ в тех областях, из которых вы можете сослаться на него, используя неквалифицированное имя.

Например, идентификаторы, объявленные в блоке PL/SQL, считаются локальными в этом блоке и глобальными для всех его подблоков. Если глобальный идентификатор переопределяется в подблоке, то оба идентификатора остаются в сфере. В подблоке, однако, будет видимым лишь локальный идентификатор, потому что для ссылок к глобальному идентификатору вам придется использовать квалифицированное имя.

Хотя нельзя объявить идентификатор дважды в одном и том же блоке, можно объявить одинаковые идентификаторы в двух разных блоках. Объекты, представленные этими идентификаторами, различны, и любое изменение одного из этих объектов не затрагивает другой.

Управляющие структуры

Управляющие структуры составляют наиболее важное расширение языка SQL в PL/SQL. Благодаря им вы не просто можете манипулировать данными ORACLE, но можете управлять

потоком выполнения, используя предложения условного, итеративного и последовательного управления выполнением, такими как IF-THEN-ELSE, FOR-LOOP, WHILE-LOOP, EXIT-WHEN и GOTO. В совокупности, эти предложения могут обработать любую ситуацию (см. тему 7).

Кодовое множество

Вы пишете программу PL/SQL как строки текста, используя специфический набор символов. В этот набор символов входят:

- прописные и строчные буквы A .. Z, a .. z;
- цифры 0 .. 9;
- символы табуляция, пробел и возврат каретки ("пропуски");
- символы ()+*/<>=!~;:.'@%,"#\$^&_[]?[].

PL/SQL не различает прописных и строчных букв, и рассматривает строчные буквы как эквиваленты соответствующих прописных букв, исключая строковые и символьные литералы.

Лексические единицы

Строка текста программы PL/SQL распадается на группы символов, называемые ЛЕКСИЧЕСКИМИ ЕДИНИЦАМИ, которые можно классифицировать следующим образом:

- разделители (простые и составные символы);
- идентификаторы, в том числе зарезервированные слова;
- литералы;
- комментарии.

РАЗДЕЛИТЕЛЬ - это простой или составной символ, имеющий в PL/SQL специальный смысл. Например, вы используете разделители для представления арифметических операций, таких как сложение и вычитание.

Простые символы кодируются как одиночные символы:

- + оператор сложения
- оператор вычитания/отрицания
- * оператор умножения

/ оператор деления

= оператор сравнения

< оператор сравнения

> оператор сравнения

(ограничитель выражения или списка

) ограничитель выражения или списка

; терминатор предложения

% индикатор атрибута

, разделитель элементов

. селектор компоненты

@ индикатор удаленного доступа

' ограничитель символьной строки

" ограничитель идентификатора

: индикатор хост-переменной

Составные символы кодируются как пары символов:

** оператор возведения в степень

<> оператор сравнения

!= оператор сравнения

~= оператор сравнения

^= оператор сравнения

<= оператор сравнения

>= оператор сравнения

:= оператор присваивания

=> оператор ассоциации

.. оператор интервала

|| оператор конкатенации

<< ограничитель метки

>> ограничитель метки

-- индикатор однострочного комментария

/* (начальный) ограничитель многострочного комментария

*/ (конечный) ограничитель многострочного комментария.

Идентификаторы

Вы используете идентификаторы для именования программных объектов и единиц PL/SQL, к которым относятся константы, переменные, исключения, курсоры, подпрограммы и пакеты. Некоторые примеры идентификаторов : X, t2, phone# , credit_limit, LastName,oracle\$number.

Идентификатор состоит из буквы, за которой (необязательно) следуют одна или несколько букв, цифр, знаков доллара, подчеркиваний или знаков номера (#). Другие символы, такие как дефис, наклонная черта или пропуск, в идентификаторе незаконны, как показывают следующие примеры:

- mine&yours -- незаконный амперсанд;
- debit-amount -- незаконный дефис ;
- on/off -- незаконная косая черта ;
- user id -- незаконный пробел.

Буквы в идентификаторах могут быть как прописными, так и строчными. PL/SQL не различает их, за исключением строковых и символьных литералов. Поэтому, если единственным различием между идентификаторами является регистр соответствующих букв, то PL/SQL трактует такие идентификаторы как одинаковые, как показывает следующий пример:

lastname LastName -- то же, что lastname LASTNAME -- то же, что lastname и LastName

Длина идентификатора не может превышать 30 символов. Однако значащим считается каждый символ в идентификаторе, включая знаки доллара, подчеркивания и знаки номера. Например, следующие два идентификатора считаются в PL/SQL различными:

lastname
last_name

Зарезервированные слова

Некоторые идентификаторы, называемые ЗАРЕЗЕРВИРОВАННЫМИ СЛОВАМИ, имеют специальный смысл в PL/SQL и не могут быть переопределены. Например, слова BEGIN и END, которые окружают исполнительную часть блока или подпрограммы, зарезервированы.

Однако зарезервированные слова можно включать как составные части в идентификаторы, как показывает следующий пример:

```
DECLARE end_of_game BOOLEAN; -- законно ...
```

Как правило, зарезервированные слова пишутся прописными буквами, чтобы облегчить читабельность. Однако это необязательно; как и любые другие идентификаторы PL/SQL, зарезервированные слова можно кодировать строчными или смешанными буквами.

Идентификаторы в кавычках

Для большей гибкости, PL/SQL позволяет вам заключать идентификаторы в двойные кавычки. Идентификаторы в кавычках необходимы нечасто, но иногда они могут быть полезными. Такой идентификатор может содержать любую последовательность печатных символов, включая пробелы, но исключая двойные кавычки: "X+Y", "last name", "**** header info ****".

Максимальная длина идентификатора в кавычках составляет 30 символов, не считая кавычек.

Использование в качестве идентификаторов в кавычках зарезервированных слов PL/SQL допускается, но НЕ рекомендуется. Использование зарезервированных слов является плохой практикой программирования.

Литералы

Литерал - это явное число, символ, строка или булевское значение, не представленное идентификатором. Примерами могут служить числовой литерал 147 и булевский литерал FALSE.

Числовые литералы

В арифметических выражениях могут использоваться два вида числовых литералов: целочисленные и вещественные. Целочисленный литерал - это целое число с необязательным знаком и без десятичной точки. Примеры целочисленных литералов: 6; -14; 0; +32767.

Вещественный литерал - это целое или дробное число с необязательным знаком и с десятичной точкой. Примеры вещественных литералов: 6.6667; -12.0; 3.14159.

Символьные литералы

Символьный литерал - это одиночный символ, окруженный одиночными апострофами. Примеры: 'Z'; '%'; '7'; ' '; 'z'; '('.

Символьные литералы включают все печатные символы в наборе символов PL/SQL: буквы, цифры, пропуски и специальные символы. PL/SQL чувствителен к регистру букв в символьных литералах. Так, литералы 'Z' и 'z' считаются различными.

Строковые литералы

Символьное значение может быть представлено идентификатором или явно записано в виде строкового литерала, который должен быть последовательностью из нуля или более символов, заключенной в апострофы: 'Hello, world!'; 'XYZ Corporation'; '10-NOV-91'.

PL/SQL чувствителен к регистру букв в строковых литералах. Например, следующие литералы считаются различными: 'baker' и 'Baker'.

Булевские литералы

Булевские литералы - это предопределенные значения TRUE и FALSE, а также "не-значение" NULL, которое обозначает отсутствие, неизвестность или неприменимость значения. Не забывайте, что булевские литералы НЕ являются строками.

Комментарии

Добавление комментариев в вашу программу способствует ее читабельности и облегчает ее понимание. Обычно комментарии используются для описания назначения и использования каждого сегмента кода. PL/SQL поддерживает два стиля комментариев: однострочные и многострочные.

Однострочный комментарий начинается с двойного дефиса (--) и заканчивается концом строки. Примеры:

```
-- начало обработки
SELECT price INTO new_price FROM goods -- взять текущую цену
WHERE goods_id= 10;
chet:= new_price * 150; -- вычислить величину счета
```

Заметьте, что однострочный комментарий может начинаться на одной строке с предложением (или частью предложения).

Многострочный комментарий начинается с пары символов /* и заканчивается парой символов */. Пример:


```
/* вычислить 15% премию для  
сотрудников с высоким рейтингом */  
IF rating > 90 THEN bonus := salary * 0.15;  
END IF;
```

Этот стиль позволяет, например, легко "закомментировать" секцию блока, которую вы хотите временно исключить из выполняемого кода.

Нельзя вкладывать комментарии друг в друга. Кроме того, нельзя использовать однострочные комментарии в том блоке PL/SQL, который будет обрабатываться динамически программой прекомпилятора ORACLE, потому что в этом случае символы конца строки игнорируются, и, как следствие, однострочный комментарий растянется до конца блока, а не только до конца строки. Поэтому в таких случаях используйте многострочные комментарии.

Типы данных

Каждая константа и переменная имеет ТИП ДАННЫХ, который специфицирует ее формат хранения, ограничения и допустимый интервал значений. PL/SQL предусматривает разнообразие предопределенных скалярных и составных типов данных. СКАЛЯРНЫЙ тип не имеет внутренних компонент. СОСТАВНОЙ тип имеет внутренние компоненты, которыми можно манипулировать индивидуально.

Рассмотрим скалярные типы данных ORACLE (см.табл. 3):

- CHAR ;
- VARCHAR2 ;
- VARCHAR ;
- NUMBER;
- DATE ;
- LONG ;
- RAW ;
- LONG RAW ;
- ROWID ;
- MLSLABEL ;
- BINARY_INTEGER .

Таблица 3 - Типы данных

Тип данных	Таблицы	Переменные PL/SQL	Комментарии
BINARY_INTEGER NATURAL POSITIVE	[-231-1; 231-1] (-2147483647 .. 2147483647) (0 .. 2147483647) (1 .. 2147483647)		Целые числа со знаком
NUMBER [(точность, масштаб)] DEC DECIMAL DOUBLE PRECISION FLOAT SMALLINT INT / INTEGER NUMERIC REAL	[10-130;9.99*10125] До 38 значащих цифр		Точность - общее число цифр Масштаб - число цифр справа от десятичной точки
Boolean	НЕТ	NULL, TRUE, FALSE	Не принимает параметров. Нельзя выбирать значения столбцов в Boolean-переменные
CHAR [(байт)] CHARACTER STRING	1..255	До 32767 байт	Хранит строки фиксированной длины
VARCHAR2 (символов)	1..2000	До 32767 байт	Хранит символьные строки переменной длины
VARCHAR	Аналог VARCHAR2		Иное сравнение
Long	До 2Гб	32767 байт	Нельзя использовать в вызовах функций, фразах SQL: WHERE, GROUP BY и CONNECT BY
RAW (целое число)	255 байт	32767 байт	Предназначены для двоичных данных или байтовых строк. НЕИНТЕРПРЕТИРУЕМЫЕ ДАННЫЕ
LONG RAW	До 2Гб	32767 байт	
DATE	1.01.4712 г. до н.э. до 31.12.4712 г. н.э.	от 1.01.14712 г. до н.э. до 31.12.314712 г. н.э.	Хранит значения в виде точек времени (т.е. дату и время. Время хранится в секундах после полуночи.

Тип ROWID

Внутренне, каждая таблица в базе данных ORACLE имеет псевдостолбец ROWID, в котором хранятся 6-байтовые двоичные значения, называемые ИДЕНТИФИКАТОРАМИ СТРОК. Идентификатор строки уникально идентифицирует строку в таблице и предоставляет самый

быстрый способ доступа к конкретной строке. Вы используете тип данных ROWID для хранения идентификаторов строк в читабельном формате.

ROWID - это подтип типа CHAR. Поэтому, после выбора или извлечения идентификатора строки в переменную ROWID, вы можете использовать функцию ROWIDTOCHAR, которая преобразует двоичное значение в 18-байтовую символьную строку, возвращая ее в формате

BBBBBBBBB.RRRR.FFFF

где BBBBBBBB - номер блока в файле базы данных (блоки нумеруются с 0),
RRRR - номер строки в блоке (строки нумеруются с 0),
FFFF - номер файла базы данных.

Все эти числа шестнадцатеричные.

Преобразования типов данных

Иногда бывает необходимо преобразовать значение из одного типа данных в другой. Например, если вы хотите исследовать идентификатор строки, вы должны преобразовать его в символьную строку. PL/SQL поддерживает как явные, так и неявные (автоматические) преобразования типов данных.

Явные преобразования типов

Чтобы специфицировать явные преобразования типов, вы используете встроенные функции, которые преобразуют значения из одних типов данных в другие (см.рис 6).

Куда		CHAR	DATE	NUMBER	RAW	ROWID
Откуда	CHAR		TO_DATE	TO_NUMBER	HEXTORAW	HARTORAWID
	DATE	TO_CHAR				
	NUMBER	TO_CHAR	TO_DATE			
	RAW	RAWTOHEX				
	ROWID	ROWIDTOCHAR				

Рисунок 6 - Функции преобразования типов данных

Неявные преобразования типов

Когда это имеет смысл, PL/SQL преобразует тип данных значения неявно. Это позволяет вам использовать литералы, переменные и параметры одного типа там, где ожидается другой тип. В следующем примере, символьные переменные `start_time` и `finish_time` хранят строковые значения, представляющие число секунд после полуночи.

Разность между этими значениями должна быть присвоена числовой переменной `elapsed_time`. Поэтому PL/SQL неявно преобразует значения CHAR в тип NUMBER.

```
DECLARE
start_time CHAR(5);
finish_time CHAR(5);
elapsed_time NUMBER(5);
BEGIN
/* Получить системное время
в секундах после полуночи. */
SELECT TO_CHAR(SYSDATE, 'SSSSS') INTO start_time FROM sys.dual;
-- выполнить какие-нибудь действия
/* Снова получить системное время. */
SELECT TO_CHAR(SYSDATE, 'SSSSS') INTO finish_time FROM sys.dual;
/* Вычислить затраченное время в секундах. */
elapsed_time := finish_time - start_time;
INSERT INTO results VALUES (... , elapsed_time);
END;
```



Перед присваиванием выбранного значения столбца переменной PL/SQL, если необходимо, преобразует это значение из типа данных исходного столбца в тип данных целевой переменной. Это происходит, например, когда вы выбираете значение столбца DATE в переменную VARCHAR2. Аналогично, перед присваиванием или сравнением значения переменной со значением столбца базы данных PL/SQL, если необходимо, преобразует значение из типа данных переменной в тип данных целевого столбца.

Если PL/SQL не может определить, какое неявное преобразование необходимо, вы получите ошибку компиляции. В таких случаях вы должны использовать явные функции преобразования типов данных.

В общем случае, не следует полагаться на неявные преобразования типов данных, потому что они могут повредить производительности и подвержены изменениям от одной версии программного обеспечения к другой. К тому же, неявные преобразования чувствительны к контексту, и потому не всегда предсказуемы. Вместо этого используйте функции преобразования

типов данных. Это сделает ваши приложения более надежными и сопровождаемыми.

Команды управления

Условное управление: предложения IF

Часто бывает необходимо предпринять альтернативные действия в зависимости от обстоятельств. Предложение IF позволяет вам выполнить последовательность предложений условно. Это значит, что, будет выполнена эта последовательность или нет, зависит от значения условия. Есть три формы предложений IF: IF-THEN, IF-THEN-ELSE и IF-THEN-ELSIF.

IF-THEN

Простейшая форма предложения IF ассоциирует условие с последовательностью предложений, окружаемой ключевыми словами THEN и END IF (не ENDIF), как показано ниже:

```
IF условие THEN  
ряд_предложений;
```

```
END IF;
```

Последовательность предложений выполняется, только если условие дает TRUE. Если условие дает FALSE или NULL, то предложение IF ничего не делает. В любом случае, управление передается на следующее предложение.

Пример :

```
IF price > max_price THEN  
Skidka:=ycenka(goods_id_1);  
UPDATE goods SET price = price - skidka WHERE goods_id = goods_id_1;  
END IF;
```

Вы можете, если хотите, записывать короткие предложения IF в одну строку, например:

```
IF x > y THEN z:= x; END IF;
```

IF-THEN-ELSE

Вторая форма предложения IF добавляет ключевое слово ELSE, за которым следует альтернативная последовательность предложений:

```
IF условие THEN
```

```
ряд_предложений1;  
ELSE  
ряд_предложений2;  
END IF;
```

Последовательность предложений в фразе ELSE выполняется, только если условие дает FALSE или NULL. Таким образом, фраза ELSE гарантирует, что одна из последовательностей предложений будет выполнена. В следующем примере, первое или второе предложение UPDATE будет выполнено, когда условие соответственно истинно или ложно:

```
IF storage_need1 = 'refregirator' THEN  
INSERT INTO storages VALUES (5, ref1, 3, ....);  
ELSE  
INSERT INTO storages VALUES (5, ref2, 3, ....);  
END IF;
```

Последовательности предложений, включаемые в фразы IF и ELSE, сами могут содержать предложения IF. Следовательно, предложения IF могут вкладываться друг в друга, как показывает следующий пример:

```
IF storage_need1 = 'refregirator' THEN  
INSERT INTO storages VALUES (5, ref1, 3, ....);  
ELSE  
IF (volume_goods < volume_ref2) THEN  
INSERT INTO storages VALUES (5, ref2, 3, ....);  
ELSE  
INSERT INTO storages VALUES (5, ref3, 3, ....);  
END IF;  
END IF;
```

IF-THEN-ELSIF

Иногда вы хотите выбрать действие из нескольких взаимно исключающих альтернатив. Третья форма предложения IF использует ключевое слово ELSIF (не ELSE IF), чтобы ввести дополнительные условия:

```
IF условие1 THEN  
ряд_предложений1;  
ELSIF условие2 THEN  
ряд_предложений2;  
ELSE ряд_предложений3;  
END IF;
```

Если первое условие дает FALSE или NULL, фраза ELSIF проверяет следующее условие. В предложении IF может быть сколько угодно фраз ELSIF; последняя фраза ELSE необязательна. Условия вычисляются по одному сверху вниз. Если любое условие даст TRUE, выполняется

соответствующая последовательность предложений, и управление передается на следующее за IF предложение (без вычисления оставшихся условий). Если все условия дадут FALSE или NULL, выполняется последовательность предложений в фразе ELSE, если она есть.

Рассмотрим следующий пример:

```
IF quantity1 > 10000 THEN
payment:= 100;
ELSIF quantity1 > 5000 THEN
payment:= 70; ELSE payment := 50;
END IF;
INSERT INTO payment_book VALUES (storage_id, payment, ...);
```

Если значение quantity 1 превышает 10000, истинны как первое, так и второе условия. Тем не менее, переменной payment присваивается правильное значение 100, потому что второе условие проверяться не будет, а управление сразу будет передано на предложение INSERT.

Избегайте неуклюжих предложений IF, подобных следующему примеру:

```
DECLARE
...
over BOOLEAN;
BEGIN
...
IF price1 > max_price THEN
over:= TRUE;
ELSE over:= FALSE;
END IF;
...
IF over = TRUE THEN
Ycenka(goods_id1, price1);
END IF;
END;
```

Этот код игнорирует два полезных факта. Во-первых, значение булевского выражения можно непосредственно присваивать булевой переменной. Так, первое предложение IF можно заменить простым присваиванием:

```
over:= price > max_price;
```

Во-вторых, булевская переменная сама имеет значение TRUE либо FALSE. Поэтому условие во втором предложении IF можно упростить:

```
IF over THEN ...
```

По мере возможности, используйте фразу ELSIF вместо вложенных предложений IF. При этом ваш код будет легче читать и понимать.

Итеративное управление

Предложения LOOP и EXIT

Предложения LOOP позволяют выполнить последовательность предложений несколько раз. Есть три формы предложения LOOP: LOOP , WHILE - LOOP и FOR - LOOP .

LOOP

Простейшую форму предложения LOOP представляет основной (или бесконечный) цикл, который окружает последовательность предложений между ключевыми словами LOOP и END LOOP:

```
LOOP  
ряд_предложений  
END LOOP;
```

При каждой итерации цикла последовательность предложений выполняется, а затем управление передается на начало цикла. Если дальнейшее повторение нежелательно или невозможно, вы можете использовать предложение EXIT, чтобы закончить цикл. Вы можете поместить сколько угодно предложений EXIT внутри цикла, но только не вне цикла. Есть две формы предложения EXIT: EXIT и EXIT WHEN.

EXIT

Предложение EXIT форсирует безусловное завершение цикла. Когда встречается предложение EXIT, цикл немедленно заканчивается, и управление передается на следующее (за END LOOP) предложение.

Пример:

```
LOOP  
...  
IF ... THEN  
...  
EXIT; -- немедленно выходит из цикла  
END IF;  
END LOOP; -- управление передается сюда
```

Как показывает следующий пример, вы не можете использовать предложение EXIT, чтобы завершить блок PL/SQL:


```
BEGIN
...
IF ... THEN
...
EXIT; -- незаконно
END IF;
END;
```

Не забывайте, что предложение EXIT можно применять только внутри цикла. Чтобы выйти из блока PL/SQL до достижения его нормального конца, можно использовать предложение RETURN.

EXIT-WHEN

Предложение EXIT-WHEN позволяет завершить цикл условно. Когда встречается это предложение, вычисляется условие в фразе WHERE. Если это условие дает TRUE, цикл завершается, и управление передается на предложение, следующее за циклом.

Пример :

```
LOOP
SELECT quantity INTO quantity1 FROM storages WHERE storage_id=45;
EXIT WHEN SQL %NOTFOUND; -- выйти из цикла при условии
...
END LOOP;
```

Пока условие не станет истинным, цикл не может завершиться. Поэтому, предложения внутри цикла должны изменять значение условия. В последнем примере, если предложение SELECT извлекает строку, условие дает FALSE. Когда предложение SELECT не сможет вернуть строку, условие даст TRUE, цикл завершится, и управление будет передано на предложение CLOSE. Предложение EXIT-WHEN заменяет простое предложение IF.

Например, сравните следующие предложения:

```
IF price1 > 100 THEN
EXIT;
EXIT WHEN price1> 100;
END IF;
```

Эти предложения логически эквивалентны, но предложение EXIT-WHEN легче читается и понимается.

Метки циклов

Как и блоки PL/SQL, циклы могут иметь метки. Метка, необъявляемый идентификатор в двойных угловых скобках, должна появиться в начале предложения LOOP:

```
<<имя_метки>>  
LOOP  
ряд_предложений  
END LOOP;
```

Имя метки цикла может также (необязательно) появиться в конце цикла в предложении END LOOP, как показывает следующий пример:

```
<< my _ loop >>  
LOOP  
...  
END LOOP my _ loop ;
```

Когда вы используете вложенные циклы, конечные метки циклов улучшают читаемость программы.

Обе формы предложения EXIT позволяют выйти не только из текущего цикла, но из любого окружающего цикла. Просто дайте метку тому циклу, который вы хотите завершить, а затем укажите эту метку в предложении EXIT :

```
<< outer >>  
LOOP  
...  
LOOP  
...  
EXIT outer WHEN ... -- выйти из обоих циклов  
END LOOP;  
...  
END LOOP outer;
```

Выход осуществляется из всех окружающих циклов, вплоть до того, чья метка специфицирована в предложении EXIT.

WHILE-LOOP

Предложение WHILE-LOOP ассоциирует условие с последовательностью предложений, окруженной ключевыми словами LOOP и END LOOP:

```
WHEN условие LOOP  
ряд_предложений;  
END LOOP;
```

Перед каждой итерацией цикла условие проверяется. Если оно дает TRUE, то

последовательность предложений выполняется, и управление возвращается на начало цикла. Если условие дает FALSE или NULL, то цикл обходится, и управление передается на следующее предложение.

Пример:

```
WHILE summa <= 50000 LOOP
```

```
...
```

```
SELECT price*quantity INTO sum1 FROM goods, storages WHERE  
goods.goods_id=storages.goods_id AND ...  
summa := summa + sum1;  
END LOOP;
```

Число повторений цикла зависит от условия и неизвестно до тех пор, пока цикл не завершится. Поскольку условие проверяется в начале цикла, последовательность предложений может не выполниться ни разу. В последнем примере, если начальное значение переменной summa окажется больше 50000, условие даст FALSE, и цикл будет обойден.

В некоторых языках имеется структура LOOP UNTIL или REPEAT UNTIL, которая проверяет условие не в начале, а в конце итерации цикла. Поэтому гарантируется хотя бы однократное выполнение тела цикла. В PL/SQL такой структуры нет, но вы легко можете ее смоделировать:

```
LOOP  
ряд_предложений;  
EXIT WHEN булевское_выражение;  
END LOOP;
```

Чтобы гарантировать хотя бы однократное выполнение цикла WHILE, используйте в условии инициализированную булевскую переменную:

```
done := FALSE;  
WHILE NOT done LOOP  
ряд_предложений;  
done := булевское_выражение;  
END LOOP;
```

Какое-то предложение внутри цикла должно присвоить булевской переменной новое значение. В противном случае вы получите бесконечный цикл. Например, следующие предложения LOOP логически эквивалентны:

```
WHILE TRUE LOOP  
...  
END LOOP;  
LOOP  
...  
END LOOP ;
```

FOR - LOOP

В то время как число итераций цикла WHILE неизвестно до тех пор, пока цикл не завершится, для цикла FOR число итераций известно до того, как войти в цикл. Циклы FOR осуществляют свои итерации по заданному интервалу целых чисел. (Курсорные циклы FOR, которые повторяются по активному множеству курсора, обсуждаются в теме 9)

Этот интервал является частью СХЕМЫ ИТЕРАЦИЙ, которая окружается ключевыми словами FOR и LOOP. Синтаксис имеет следующий вид:

```
FOR счетчик IN [REVERSE] нижняя_граница..верхняя_граница LOOP  
ряд_предложений;  
END LOOP;
```

Интервал вычисляется один раз, при первом входе в цикл, и больше не перевычисляется. Как показывает следующий пример, последовательность предложений выполняется один раз для каждого целого в заданном интервале. После каждой итерации выполняется приращение индекса цикла.

```
FOR i IN 2..5 LOOP -- присваивает переменной i значения 2, 3, 4, 5  
ряд_предложений; -- будет выполнен 4 раза  
END LOOP;
```

Как показывает следующий пример, если нижняя граница интервала совпадает с верхней, цикл выполняется один раз:

```
FOR i IN 3..3 LOOP -- присваивает переменной i значение 3  
ряд_предложений; -- будет выполнен один раз  
END LOOP;
```

По умолчанию индекс наращивается на 1 от нижней до верхней границы. Однако, если вы используете ключевое слово REVERSE, индекс будет изменяться в обратном направлении, от верхней границы к нижней, как показывает следующий пример. После каждой итерации индекс уменьшается на 1.

```
FOR i IN REVERSE 1..3 LOOP -- присваивает переменной i 3, 2, 1  
ряд_предложений; -- будет выполнен три раза  
END LOOP;
```

Тем не менее, и в этом случае вы записываете границы интервала в возрастающем (а не убывающем) порядке. Внутри цикла FOR к индексу цикла можно обращаться как к константе. Поэтому индекс может встречаться в выражениях, но ему нельзя присваивать значений, как показывает следующий пример:

```
FOR goods_id1 IN 1..10 LOOP
```

```
...  
IF NOT finished THEN  
INSERT INTO ... VALUES (goods_id1, ...); -- законно  
goods_id1 := goods_id1 + 1; -- не законно  
END IF;  
  
END LOOP;
```

Схемы итераций

Границами интервала цикла могут быть литералы , переменные или выражения, но их значения должны быть целочисленными. Например, следующие схемы итераций законны:

```
j IN -5..5  
k IN REVERSE first .. last  
step IN 0..TRUNC(high/low) * 2  
code IN ASCII('A')..ASCII('J')
```

Как видите, нижняя граница не обязана быть равна 1. Однако приращение (или отрицательное приращение) счетчика цикла всегда равно 1. В некоторых языках существует фраза, с помощью которой можно задать другую величину приращения. В PL/SQL такой структуры не существует, но вы можете легко смоделировать ее.

Рассмотрим следующий пример:

```
FOR j IN 5..15 LOOP -- присваивает j значения 5,6,7,...  
IF MOD(j, 5) = 0 THEN -- выбирает кратные 5  
ряд_предложений; -- j имеет значения 5,10,15  
END IF;  
END LOOP;
```

Внутри последовательности предложений счетчик цикла будет иметь лишь значения 5, 10 и 15. Вы можете предпочесть не столь эlegantный, но более эффективный способ, показанный в следующем примере. Внутри последовательности предложений при каждом обращении к счетчику цикла его значение умножается на 5.

```
FOR j IN 1..3 LOOP -- присваивает j значения 1,2,3  
ряд_предложений; -- вместо j обращаться к j*5  
END LOOP;
```

Динамические интервалы PL/SQL позволяет определять интервал цикла динамически во время выполнения, как показывает следующий пример:

```
SELECT COUNT(goods_id) INTO goods_count FROM goods;
```

```
FOR i IN 1..goods_count LOOP
...
END LOOP;
```

Значение `goods_count` неизвестно во время компиляции; предложение `SELECT` возвращает это значение во время выполнения.

Если вычисленная нижняя граница интервала окажется больше верхней (как показывает следующий пример), последовательность предложений внутри цикла не будет выполняться, и управление будет передано на следующее за циклом предложение:

```
-- limit получает значение 1
FOR i IN 2..limit LOOP
ряд_предложений; -- не выполнится ни разу
END LOOP; -- управление будет передано сюда
```

Правила сферы

Счетчик цикла определен только внутри цикла. Вы не можете обратиться к нему вне цикла. После выхода из цикла значение счетчика цикла не определено, как показывает следующий пример:

```
FOR goods_id1 IN 1..10 LOOP
...
END LOOP;
Next_goods:= goods_id1 +1; -- незаконно
```

Вы не обязаны явно объявлять счетчик цикла, потому что он неявно объявляется как локальная переменная типа `INTEGER`. Как показывает следующий пример, это локальное объявление перекрывает любое глобальное объявление:

```
DECLARE goods_id1 INTEGER;
BEGIN
...
FOR goods_id1 IN 1..15 LOOP
...
IF goods_id1 > 10 THEN ... -- обращается к счетчику цикла
END LOOP;
END;
```

Чтобы в этом примере обратиться к глобальной переменной `goods_id1`, вы должны использовать метку и квалифицированную ссылку:

```
<<main>>
DECLARE goods_id1 INTEGER;
BEGIN
```

```
...
FOR goods_id1 IN 1..15 LOOP
...
IF main . goods _ id 1 > 10 THEN ... --обращается к глоб.переменной
END LOOP ;
END main ;
```

Такие же правила сферы применимы к вложенным циклам FOR. Рассмотрим следующий пример. Оба счетчика циклов имеют одно и то же имя. Поэтому для того, чтобы обратиться из внутреннего цикла к счетчику внешнего цикла, вы должны использовать метку и квалифицированную ссылку:

```
<<outer>>
FOR step IN 1..25 LOOP
FOR step IN 1..10 LOOP
...
IF outer . step > 15 THEN
...
END LOOP;
END LOOP outer;
```

Использование предложения EXIT

Предложение EXIT позволяет завершить цикл FOR прежде времени. Предложение EXIT позволяет выйти не только из текущего цикла FOR, но из любого окружающего цикла. Просто дайте метку тому циклу, который вы хотите завершить, а затем укажите эту метку в предложении EXIT :

```
<<outer>>
FOR i IN 1..5 LOOP
...
FOR j IN 1..10 LOOP
SELECT price INTO price1 FROM goods WHERE goods_id=2;
EXIT outer WHEN price1>max+price; -- выход из обоих циклов
...
END LOOP ;
END LOOP outer; -- управление будет передано сюда
```

Последовательное управление: предложения GOTO и NULL

В отличие от предложений IF и LOOP, предложения GOTO и NULL не являются ключевыми в

программировании на языке PL/SQL. Структура языка такова, что предложение GOTO требуется редко. Иногда его применение может быть оправдано некоторым упрощением логики. Предложение NULL может прояснить смысл условных предложений в программе и улучшить читаемость.

Предложение GOTO

Предложение GOTO выполняет безусловный переход к указанной метке. Метка должна быть уникальной в своей сфере, и должна предшествовать выполняемому предложению или блоку PL/SQL. Предложение GOTO передает управление на помеченное предложение или блок. В следующем примере управление передается на выполняемое предложение, находящееся дальше в последовательности предложений:

```
BEGIN
...
GOTO insert_row ;
...
<<insert_row>>
INSERT INTO goods VALUES ...
END;
```

В следующем примере управление передается на блок PL/SQL, расположенный выше в последовательности предложений:

```
BEGIN
...
<<update_row>>
BEGIN
UPDATE goods SET ... END;
...
GOTO update_row;
...
END ;
```

В следующем примере метка <<end_loop>> незаконна, потому что она стоит не перед выполняемым предложением:

```
DECLARE
done BOOLEAN ;
BEGIN
...
FOR i IN 1..10 LOOP
```



```
IF done THEN
GOTO end_loop;
END IF;
...
<<end_loop>> -- незаконно
END LOOP; -- это не выполняемое предложение END;
```

Чтобы исправить последний пример, просто добавьте за меткой предложение NULL :

```
DECLARE

BEGIN
...
FOR i IN 1..10 LOOP
IF done THEN
GOTO end_loop;
END IF;
...
<<end_loop>>
NULL; -- выполняемое предложение
END LOOP;
END;
```

Как показывает следующий пример, предложение GOTO может передать управление в окружающий блок из текущего блока:

```
DECLARE
New_name CHAR(20);
BEGIN
...
<<get_name>>
SELECT name INTO new_ename FROM goods WHERE ...
...
BEGIN
...
GOTO get_name; -- переход в окружающий блок
END;
END;
```

Предложение GOTO передает управление в первый из окружающих блоков, в котором встретится указанная метка.

Некоторые возможные назначения предложения GOTO незаконны. В частности, предложение GOTO не может передавать управление в предложение IF, в предложение LOOP или в подблок. Например, следующее предложение GOTO незаконно:

```
BEGIN
...
GOTO update_row; -- незаконный переход в предложение IF
...
IF done THEN
...
<<update_row>>
UPDATE goods SET ...
END IF ;
END;
```

Далее, предложение GOTO не может передавать управление из одной фразы предложения IF в другую, как показывает следующий пример:

```
BEGIN
...
IF done THEN ...
GOTO update _ row ; -- незаконный переход в фразу ELSE
ELSE
...
<<update_row>>
UPDATE goods SET ...
END IF;
END;
```

Следующий пример показывает, что предложение IF не может передавать управление из окружающего блока в подблок:

```
BEGIN
...
IF storage_need = 'refregirator' THEN
GOTO insert_ref; -- незаконный переход в подблок
END IF;
...
BEGIN
...
<<insert_ref>>
INSERT INTO storages VALUES ...
END ;
END ;
```

Чрезмерное применение предложений GOTO может привести к сложному, неструктурированному коду, который трудно понимать и сопровождать. Поэтому старайтесь использовать предложения GOTO пореже.

Предложение NULL

Предложение NULL явно специфицирует отсутствие действия; оно ничего не делает, и управление передается на следующее предложение. Оно, однако, может улучшить читабельность программы. В конструкте, допускающем альтернативные действия, предложение NULL используется для обозначения места действия. Оно напоминает читателю программы о том, что соответствующая альтернатива не была пропущена нечаянно, а просто не требует никакого действия.

Каждая фраза в предложении IF должны содержать хотя бы одно выполнимое предложение. Предложение NULL помогает удовлетворить этому требованию. Поэтому вы можете использовать предложение NULL в фразах, соответствующих тем обстоятельствам, в которых никаких действий не требуется.

В следующем примере предложение NULL подчеркивает, что премии получают лишь сотрудники с высоким рейтингом:

```
IF price1 > max_price THEN
усенка(goods_id1, price1);
ELSE
NULL;
END IF;
```

Предложение NULL предоставляет также удобный способ создания "затычек" при разработке приложения сверху вниз. Затычки - это фиктивные подпрограммы, с помощью которых вы откладываете определения процедур и функций до тех пор, пока не проверите и не отладите главную программу.

В следующем примере предложение NULL помогает удовлетворить тому требованию языка, что выполняемая часть подпрограммы должна содержать хотя бы одно предложение:

```
PROCEDURE усенка (goods_id_ INTEGER, price_ REAL) IS
BEGIN
NULL;
END усенка;
```

Создание хранимых процедур и функций

Подпрограмма - это именованный блок PL/SQL, который принимает параметры и может быть вызван. PL/SQL имеет два типа подпрограмм, называемых ПРОЦЕДУРАМИ и ФУНКЦИЯМИ.

Обычно процедуру вызывают для того, чтобы выполнить некоторое действие, а функцию - для того, чтобы вычислить некоторое значение. Как и неименованные (АНОНИМНЫЕ) блоки PL/SQL, подпрограммы имеют декларативную часть, исполняемую часть и необязательную часть обработки исключений.

Декларативная часть содержит объявления типов, курсоров, констант, переменных, исключений и вложенных подпрограмм. Все эти объекты локальны, и перестают существовать после выхода из подпрограммы.

Исполняемая часть содержит предложения, которые присваивают значения, управляют выполнением и манипулируют данными ORACLE.

Часть обработки исключений содержит обработчики, которые имеют дело с исключениями, возбуждаемыми при исполнении.

Подпрограммы можно определять в любом инструменте ORACLE, который поддерживает PL/SQL. Их можно объявлять в блоках PL/SQL, процедурах, функциях и пакетах. Однако подпрограммы должны объявляться в конце декларативной секции, после всех других программных объектов. Например, следующее объявление процедуры не на месте:

```
DECLARE
PROCEDURE ysenka (...) IS -- не на месте
Goods_id1 NUMBER;
...
BEGIN
...
END;
```

Обычно инструменты ORACLE, такие как SQL* Plus , которые инкорпорируют в себя процессор PL/SQL, способны сохранять подпрограммы для последующего, строго локального, исполнения. Однако для того, чтобы быть общедоступными для всех инструментов ORACLE, такие подпрограммы должны быть сохранены в базе данных.

Преимущества подпрограмм

Хранимые подпрограммы предоставляют расширяемость, модульность, способствуют абстракции, предоставляют более высокую продуктивность разработки, лучшую производительность, экономию памяти, целостность приложений и более строгую защиту.

Расширяемость. Подпрограммы они позволяют вам приспособливать средства PL/SQL для ваших потребностей. Например, если вам нужна процедура, которая добавляла бы новые склады, вы легко можете написать ее:

```
CREATE PROCEDURE insert_ware (new_address VARCHAR2, new_volume NUMBER,
new_storage_have VARCHAR2) AS
BEGIN
INSERT INTO warehouses VALUES (ware_seq.NEXTVAL, new_address, new_volume, new_volume,
new_storage_have);
END insert_ware;
```

Модульность. Подпрограммы позволяют вам разбивать ваши программы на управляемые, хорошо определенные логические модули. Это поддерживает методы проектирования сверху вниз и пошагового уточнения, характерные для структурного подхода к решению проблем. Помимо этого, подпрограммы способствуют использованию и сопровождаемости. Однажды проверенную подпрограмму можно с уверенностью использовать в любом количестве приложений. Более того, лишь одна подпрограмма затрагивается, если изменяется ее определение. Это упрощает сопровождение и развитие.

Абстракции. Подпрограммы способствуют умственному отделению от частных деталей. Чтобы использовать подпрограммы, вам нужно знать, что они делают, а не как они это делают. Поэтому вы можете проектировать приложения сверху вниз, не заботясь о деталях реализации. С помощью фиктивных подпрограмм (затычек) вы можете отложить определение процедур и функций до тех пор, пока не протестируете и не отладите главную программу.

Повышение продуктивности. Проектируя приложения вокруг библиотеки хранимых подпрограмм, вы можете избежать неоднократного кодирования повторяющихся операций, повышая продуктивность разработки. Предположим, например, что несколько различных приложений вызывают хранимую процедуру, которая управляет таблицей goods в базе данных. Если метод управления изменится, потребуется пересмотреть лишь одну процедуру, а не все приложения.

Улучшение производительности. Использование подпрограмм может сократить число обращений ваших приложений к ORACLE. Например, для исполнения десяти индивидуальных предложений SQL требуется десять обращений к ORACLE, но для исполнения подпрограммы, содержащей десять предложений SQL, необходим лишь один вызов. Уменьшение числа обращений может резко увеличить производительность, особенно если ваше приложение взаимодействует с ORACLE через сеть.

Хранимые процедуры могут улучшить производительность базы данных. Использование процедур существенно сокращает объем информации, пересылаемой по сети, по сравнению с выдачей индивидуальных предложений SQL или пересылкой в ORACLE текста целого блока PL/SQL. Более того, так как откомпилированная форма процедуры хранится в базе данных готовой к выполнению, для исполнения ее кода не требуется шага компиляции. Кроме того, если процедура уже присутствует в разделяемом пуле в SGA, то не требуется ее извлечения с диска, и выполнение может начаться немедленно.

Экономия памяти. Хранимые подпрограммы используют преимущества разделяемой памяти ORACLE. Так, лишь одна копия подпрограммы должна быть загружена в память, чтобы быть доступной всем пользователям. Использование одного и того же кода совместно многими пользователями приводит к существенной экономии памяти, требуемой приложениям.

Целостность приложений. Хранимые подпрограммы улучшают целостность и согласованность ваших приложений. Создавая приложения вокруг библиотеки хранимых

подпрограмм, вы снижаете вероятность ошибок кодирования. Например, процедуру или функцию требуется протестировать лишь однократно, чтобы гарантировать, что она возвращает точный результат. После проверки, процедура может использоваться в любом числе приложений, поставляя им гарантированно достоверные результаты. При любом изменении структур данных, к которым обращается процедура, перекомпилировать потребуется лишь эту процедуру; приложения, вызывающие эту процедуру, не потребуют никаких модификаций (даже перекомпиляции).

Повышенная безопасность. Хранимые подпрограммы повышают безопасность данных. Ваш АБД может ограничить доступ пользователей к определенным операциям с базой данных, предоставляя доступ лишь через подпрограммы. Например, АБД может предоставить пользователям право выполнения хранимой процедуры, обновляющей таблицу goods, но не давать доступа к самой таблице goods. Когда пользователь вызывает процедуру, эта процедура выполняется с привилегиями владельца процедуры. Так как пользователи имеют лишь привилегию выполнять процедуру, но не имеют привилегий опрашивать, обновлять или удалять данные из соответствующих таблиц, возможности пользователей по манипулированию данными ограничиваются тем, что заложено в процедуру.

Процедуры

Процедура - это подпрограмма, которая выполняет специфическое действие.

Вы пишете процедуры, используя синтаксис:

```
PROCEDURE имя [ (параметр [, параметр, ...]) ] IS  
[локальные объявления]  
BEGIN  
исполняемые предложения  
[EXCEPTION  
обработчики исключений]  
END [имя];
```

где каждый "параметр" имеет следующий синтаксис:

```
имя_перем [IN | OUT | IN OUT] тип_данных [{:= | DEFAULT} знач]
```

Здесь моды IN | OUT | IN OUT, определяющие тип параметра, являются необязательными. Как необязательным является определение значений по умолчанию, которые могут быть заданы двумя способами:

:=значение либо DEFAULT значение.

В отличие от спецификатора типа данных в объявлении переменной, спецификатор типа данных для параметра не может иметь ограничений (см. тему 11).

Например, следующее объявление name1 незаконно (должно быть просто VARCHAR 2):

```
PROCEDURE ... (name1 VARCHAR2(20)) IS -- незаконно ;
BEGIN
...
END;
```

Процедура имеет две части: спецификацию и тело. Спецификация процедуры начинается с ключевого слова PROCEDURE и заканчивается именем процедуры или списком параметров. Объявления параметров необязательны. Если процедура не принимает параметров, скобки также не кодируются.

Тело процедуры начинается с ключевого слова IS и заканчивается ключевым словом END, за которым может следовать имя процедуры. Тело процедуры состоит из трех частей: декларативной части, исполняемой части и необязательной части обработки исключений.

Декларативная часть содержит локальные объявления, которые помещаются между ключевыми словами IS и BEGIN. Ключевое слово DECLARE, которое начинает декларативную часть в анонимном блоке PL/SQL, здесь не используется.

Исполняемая часть содержит предложения, которые помещаются между ключевыми словами BEGIN и EXCEPTION (или END). В исполняемой части процедуры должно быть хотя бы одно предложение.

Часть обработки исключений содержит обработчики исключений, которые помещаются между ключевыми словами EXCEPTION и END.

Рассмотрим процедуру buy _ goods , которая проводит в БД операцию покупки товара:

```
CREATE OR REPLACE PROCEDURE buy_goods (good_id1 NUMBER, quantity1 NUMBER) IS
Storage _ id 1 NUMBER (4); -- переменная для номера партии
ware _ id 1 NUMBER (4); -- переменная для номера склада
vol NUMBER (6,2); -- переменная для объема единицы товара
BEGIN
SELECT storage_id, ware_id INTO storage_id1, ware_id1 FROM storages WHERE
begin_time IN (SELECT MIN(begin_time) FROM storages
WHERE Goods_id=Good_id1 AND quantity>quantity1);
-- определяем объем единицы покупаемого товара
SELECT volume INTO vol FROM goods WHERE goods_id=good_id1;
-- фиксируем покупку части партии товара
UPDATE storages SET quantity = quantity - quantity1 WHERE storage_id = storage_id1;
--фиксируем освобождение соответствующего объема на складе
UPDATE warehouses SET volume_rest = volume_rest + quantity1* vol WHERE ware_id=ware_id1;
EXCEPTION
WHEN NO_DATA_FOUND THEN
INSERT INTO opt_audit VALUES (OPT_CUR.NEXTVAL, good_id1, 'No such goods');
```

```
END buy _ goods ;
```

При своем вызове эта процедура принимает номер товара и количество товара, которое хотят купить. Она использует номер, чтобы выбрать информацию о партии этого товара, завезенной раньше всего из таблицы базы данных storages . Если информации о партии искомого товара нет, возбуждается исключение. В противном случае количество товара на складе уменьшается.

Процедура вызывается как предложение PL/SQL. Например, процедура buy _ goods может быть вызвана так:

```
buy_goods (goods_need, count_goods);
```

Функции

Функция - это подпрограмма, которая вычисляет значение. Функции структурируются так же, как и процедуры, с той разницей, что функции содержат фразу RETURN. Вы пишете функции, используя синтаксис

```
FUNCTION имя [ (аргумент [, аргумент, ...]) ] RETURN тип_данных IS  
[локальные объявления]  
BEGIN  
исполняемые предложения  
[EXCEPTION  
обработчики исключений]  
END [имя];
```

где каждый "аргумент" имеет следующий синтаксис:

```
имя_перемен [IN | OUT | IN OUT] тип_данных [{:= | DEFAULT} знач]
```

Здесь моды IN | OUT | IN OUT, определяющие тип параметра, являются необязательными. Как необязательным является определение значений по умолчанию, которые могут быть заданы двумя способами:

:=значение либо DEFAULT значение.

В отличие от спецификатора типа данных в объявлении переменной, спецификатор типа данных для параметра не может иметь ограничений. Как и процедура, функция имеет две части: спецификацию и тело. Спецификация функции начинается с ключевого слова FUNCTION и заканчивается фразой RETURN, которая специфицирует тип результирующего значения. Объявления аргументов необязательны. Если функция не принимает аргументов, скобки также не кодируются. Тело функции начинается с ключевого слова IS и заканчивается ключевым словом END, за которым может следовать имя функции. Тело функции состоит из трех частей: декларативной части, исполняемой части и необязательной части обработки исключений.

Декларативная часть содержит локальные объявления, которые помещаются между ключевыми словами IS и BEGIN. Ключевое слово DECLARE, которое начинает декларативную часть в анонимном блоке PL/SQL, здесь не используется.

Исполняемая часть содержит предложения, которые помещаются между ключевыми словами BEGIN и EXCEPTION (или END). В исполняемой части процедуры должно встретиться хотя бы одно предложение RETURN.

Часть обработки исключений содержит обработчики исключений, которые помещаются между ключевыми словами EXCEPTION и END (будет рассмотрено позже).

Рассмотрим процедуру quantity_ok, которая определяет, есть ли товар с заданным номером на складе в нужном объеме:

```
FUNCTION quantity_ok (goods_id1 NUMBER, quantity1 NUMBER) RETURN BOOLEAN IS
```

```
Sum_quantity NUMBER(6);
```

```
BEGIN
```

```
SELECT SUM(quantity) INTO Sum_quantity FROM storages WHERE goods_id = goods_id1;
```

```
RETURN (Sum_quantity >= quantity1);
```

```
END quantity_ok;
```

При своем вызове эта функция принимает номер товара и требуемое количество товара. Она использует номер товара, чтобы посчитать суммарное количество товара на складах из таблицы базы данных storages . Если суммарное количество превышает требуемое, функция возвращает TRUE, в противном случае - FALSE .

Функции вызываются как часть выражения.

Например, функция quantity _ ok может быть вызвана так:

```
IF quantity_ok (new_goods_id, need_quantity) THEN
```

```
...
```

```
END IF ;
```

```
...
```

```
done := quantity_ok (new_goods_id, need_quantity);
```

Идентификатор функции является выражением, которое заменяется своим значением.

Предложение RETURN

Предложение RETURN немедленно завершает выполнение подпрограммы и возвращает управление вызывающей программе. Выполнение продолжается с предложения, следующего за вызовом подпрограммы. Подпрограмма может содержать несколько предложений RETURN, ни одно из которых не обязано быть последним лексическим предложением в подпрограмме. Выполнение любого из них немедленно завершает подпрограмму. Однако наличие в

подпрограмме нескольких точек выхода не является хорошей практикой программирования. В процедурах предложение RETURN не может содержать выражение. Это предложение просто возвращает управление вызывающей программе до достижения нормального конца процедуры. Однако в функциях предложение RETURN должно содержать выражение, которое вычисляется при выполнении предложения RETURN. Результирующее значение присваивается идентификатору функции.

Поэтому функция должна содержать хотя бы одно предложение RETURN. В противном случае PL/SQL возбуждает предопределенное исключение PROGRAM_ERROR во время выполнения.

Не путайте предложение RETURN с фразой RETURN, которая специфицирует тип данных результирующего значения в спецификации функции.

Вызов подпрограмм

Итак, встроенные функции PL / SQL являются собственными функциями этого языка и они хранятся на сервере ORACLE. Если у вас есть Процедурное расширение базы данных, то вы можете компилировать собственные (пользовательские) подпрограммы отдельно и сохранять их в базе данных ORACLE, где они готовы к выполнению.

Вы можете вызывать хранимые встроенные и пользовательские подпрограммы из триггера базы данных, другой хранимой подпрограммы, приложения прекомпилятора ORACLE, приложения OCI или из инструмента ORACLE, такого как SQL*Plus.

Хранимая подпрограмма может вызывать другую хранимую подпрограмму. Например, в анонимном блоке в вызове процедуры может появиться вызов встроенной функции

```
BEGIN
...
insert_ware (CONCAT(country, Kharkov'), 2000);
...
END;
```

Вы можете вызывать хранимые подпрограммы интерактивно из инструментов ORACLE, таких как SQL*Plus, SQL*Forms или SQL*DBA. Например, из SQL*Plus вы могли бы вызвать независимую процедуру insert_ware следующим образом:

```
SQL> EXECUTE insert_ware (CONCAT(country, Kharkov'), 2000);
```

или

```
SQL> EXEC insert_ware (CONCAT(country, Kharkov'), 2000);
```

Вызов функций из SQL*Plus происходит следующим образом:

```
SQL> SELECT fac(5) FROM SYS.DUAL;
```

или

```
SQL> SELECT fac(5) FROM DUAL;
```

Поскольку функция должна вернуть значение, SQL*Plus использует с этой целью системную таблицу DUAL , расположенную в схеме системного администратора SYS .

Рассмотрим пример вызова в SQL*Plus встроенной функции преобразования типов данных TO _ CHAR (рис. 7):

```
SQL> SELECT TO_CHAR(SYSDATE, 'YEAR MONTH DAY DD') FROM DUAL;
```

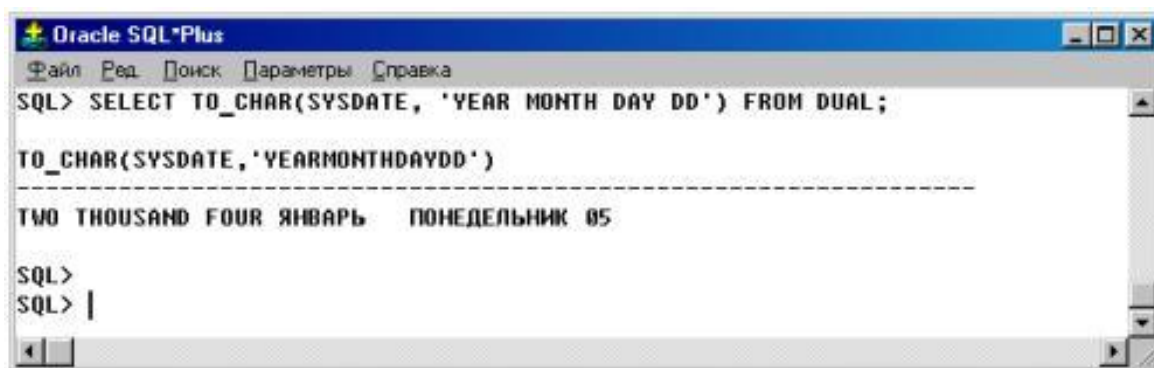


Рисунок 7 - Пример вызова встроенной функции

Функция первый параметр получает с помощью другой встроенной функции SYSDATE , а затем полученное значение приводит к формату YEAR MONTH DAY DD '. Таким образом, показанные вызов функции позволяет вернуть текущую дату в формате: год прописью, название месяца, название дня недели и число.

Перед выполнением хранимой подпрограммы ORACLE устанавливает неявную точку сохранения. Если подпрограмма сбивается в результате необработанного исключения, ORACLE осуществляет откат к этой точке сохранения. Тем самым отменяется вся работа, проделанная подпрограммой.

Тело независимой или пакетированной хранимой подпрограммы может содержать любое предложение SQL или PL/SQL. Однако подпрограммы, участвующие в распределенной транзакции, триггерах базы данных и приложениях SQL*Forms, не могут вызывать хранимых подпрограмм, содержащих предложения COMMIT, ROLLBACK или SAVEPOINT. Обращения к хранимым функциям могут появляться в процедурных предложениях, но НЕ в предложениях SQL.

Параметры подпрограмм

Фактические и формальные параметры

Подпрограммы принимают и передают информацию через ПАРАМЕТРЫ. Переменные или выражения, которые специфицированы в списке параметров в вызове подпрограммы, называются ФАКТИЧЕСКИМИ параметрами.

Например, следующий вызов функции передает два фактических параметра, new _ goods _ id и need _ quantity :

```
done := quantity_ok (new_goods_id, need_quantity);
```

Как показывает следующий вызов функции, в некоторых случаях в качестве фактических параметров можно использовать выражения:

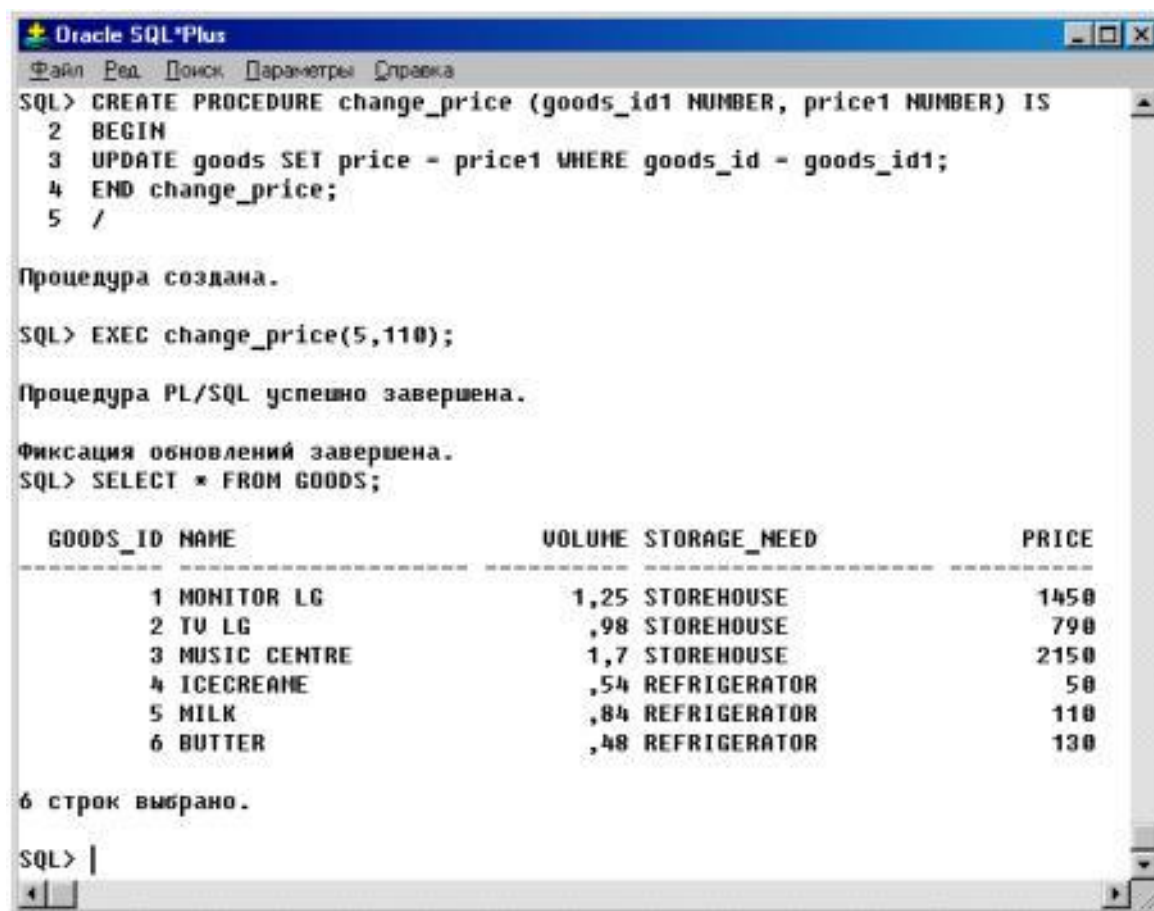
```
done := quantity_ok (new_goods_id, quan_1+ quan_2);
```

Переменные, объявленные в спецификации подпрограммы и используемые в теле подпрограммы, называются ФОРМАЛЬНЫМИ параметрами.

Например, следующая процедура объявляет два формальных параметра с именами goods _ id 1 и price 1:

```
PROCEDURE change_price (goods_id1 NUMBER, price1 NUMBER) IS  
BEGIN  
UPDATE goods SET price = price1 WHERE goods_id = goods_id1;  
END change_price;
```

Вызов процедуры показан на рисунке 8.



```
Oracle SQL*Plus
Файл  Редактирование  Поиск  Параметры  Справка
SQL> CREATE PROCEDURE change_price (goods_id1 NUMBER, price1 NUMBER) IS
2 BEGIN
3 UPDATE goods SET price = price1 WHERE goods_id = goods_id1;
4 END change_price;
5 /

Процедура создана.

SQL> EXEC change_price(5,110);

Процедура PL/SQL успешно завершена.

Фиксация обновлений завершена.
SQL> SELECT * FROM GOODS;
```

GOODS_ID	NAME	VOLUME	STORAGE_NEED	PRICE
1	MONITOR LG	1,25	STOREHOUSE	1450
2	TV LG	,98	STOREHOUSE	790
3	MUSIC CENTRE	1,7	STOREHOUSE	2150
4	ICECREAME	,54	REFRIGERATOR	50
5	MILK	,84	REFRIGERATOR	110
6	BUTTER	,48	REFRIGERATOR	130

```
6 строк выбрано.

SQL> |
```

Рисунок 8 - Пример создания и вызова процедуры в утилите SQL * Plus

Хотя это и не обязательно, хорошая практика программирования рекомендует использовать разные имена для фактических и формальных параметров. Когда вы вызываете процедуру `change _ price` , фактические параметры вычисляются, и их значения присваиваются соответствующим формальным параметрам. При этом PL/SQL преобразует значение из одного типа данных в другой, если необходимо.

Например, следующий вызов процедуры `change _ price` законен:

```
change_price (change_goods, 2500);
```

Фактический параметр и соответствующий ему формальный параметр должны иметь совместимые типы данных. Например, PL/SQL не может преобразовать друг в друга типы данных `DATE` и `REAL`. Кроме того, значение результата также должно быть совместимо с новым типом данных. Следующий вызов процедуры возбудит предопределенное исключение `VALUE_ERROR`, потому что PL/SQL не может преобразовать второй фактический параметр в число:

```
change _ price ( change _ goods , $2500'); -- из-за использования строкового значения
```

Позиционная и именная нотация

При вызове подпрограммы можно записывать фактические параметры, используя позиционную или именную нотацию. Иными словами, вы можете указывать соответствие между фактическими и формальными параметрами через позиции этих параметров или через их имена.

Например, при объявлениях

```
DECLARE  
change_goods NUMBER(4);  
new_price NUMBER(8,2);  
PROCEDURE change_price (goods_id1 NUMBER, price1 NUMBER) IS ...
```

Вы можете вызвать процедуру change _ price четырьмя логически эквивалентными способами:

```
BEGIN  
change_price(change_goods, new_price); -- позиционная нотация  
change_price(change_goods=> goods_id1, new_price=> price1); -- именная нотация  
change_price(new_price=> price1, change_goods=> goods_id1); -- именная нотация  
change_price(change_goods, new_price=> price1); -- смешанная нотация  
...  
END;
```

Первый вызов процедуры использует позиционную нотацию. Компилятор PL/SQL ассоциирует первый фактический параметр, change _ goods , с первым формальным параметром, goods _ id 1, а второй фактический параметр, new _ price , - со вторым формальным параметром, price 1.

Второй вызов процедуры использует именную нотацию. Стрелка ассоциирует формальный параметр слева от стрелки с фактическим параметром справа от стрелки.

Третий вызов процедуры также использует именную нотацию и показывает, что вы можете задавать пары параметров в любом порядке. Поэтому вы не обязаны знать порядок, в котором перечислены формальные параметры.

Четвертый вызов процедуры показывает, что вы можете смешивать позиционную и именную нотации. В данном случае первый параметр задан в позиционной, а второй - в именной нотации. Позиционная нотация в этом варианте должна предшествовать именной. Обратное не допускается. Например, следующий вызов процедуры незаконен:

```
change_price(new_price=> price1, change_goods); -- незаконно
```

Моды параметров

Вы используете моды параметров, чтобы определить поведение формальных параметров подпрограммы. Все три возможные моды: IN (умалчиваемая), OUT и IN OUT, могут использоваться в любой процедуре. Что касается функций, то избегайте использования моды OUT или IN OUT в функциях. Назначение функции - принять нуль или более аргументов и вернуть единственное значение. Возврат функцией нескольких результирующих значений является плохой практикой программирования. Кроме того, функции должны быть свободны от ПОБОЧНЫХ ЭФФЕКТОВ, то есть не должны изменять значений переменных, не локальных для данной функции.

IN

Параметр с модой IN передает значение вызываемой подпрограмме. Внутри подпрограммы такой параметр выступает как константа. Поэтому ему нельзя присвоить значение. Например, следующее предложение присваивания вызовет ошибку компиляции:

```
PROCEDURE insert_storage (ware_id1 IN NUMBER, goods_id1 IN NUMBER, quantity1 IN NUMBER)
IS
Storage_id1 NUMBER(4);
Volume_goods NUMBER(6,2);
Volume_goods_sum NUMBER(6,2);
Volume_ware NUMBER(6,2);
Over_goods NUMBER(6,2);
BEGIN
SELECT volume INTO Volume_goods FROM goods WHERE goods_id= goods_id1;
Volume_goods_sum:= Volume_goods*quantity1;
SELECT volume_rest INTO Volume_ware FROM warehouses WHERE ware_id= ware_id1;
IF (Volume_ware< Volume_goods_sum) THEN
Over_goods:=(Volume_goods_sum-Volume_ware)/Volume_goods;
quantity1:= quantity1- Over_goods; -- незаконно
END IF;
INSERT INTO storages VALUES (stor_seq.nextval, ware_id1, goods_id1, quantity1, SYSDATE, NULL);
UPDATE Warehouses SET Volume_rest= Volume_rest- Volume_goods_sum WHERE
ware_id= ware_id1;
END insert_storage;
```

Фактический параметр, соответствующий формальному параметру с модой IN, может быть константой, литералом, инициализированной переменной или выражением. В отличие от параметров OUT и IN OUT, параметры IN могут инициализироваться умалчиваемыми значениями.

OUT

Параметр с модой OUT позволяет возвращать значение вызывающей программе. Внутри подпрограммы такой параметр выступает как неинициализированная переменная. Поэтому его

значение нельзя присваивать другим переменным или переприсвоить самому себе. Например, следующее определив параметр quantity 1 с модой OUT мы получим неверное его использование в следующих местах:

```
PROCEDURE insert_storage (ware_id1 IN NUMBER, goods_id1 IN NUMBER, quantity1 OUT
NUMBER) IS
...
BEGIN
SELECT volume INTO Volume_goods FROM goods WHERE goods_id= goods_id1;
Volume_goods_sum:= Volume_goods*quantity1; -- незаконно
...
quantity1:= quantity1- Over_goods; -- незаконно
...

END insert _ storage ;
```

Фактический параметр, соответствующий формальному параметру с модой OUT, должен быть переменной; он не может быть константой или выражением. Например, следующий вызов процедуры незаконен:

```
insert_storage (15, 34, quan1+ quan2); -- синтаксическая ошибка
```

PL/SQL проверяет на такие синтаксические ошибки во время компиляции, не допуская возможного перекрытия констант и выражений. Формальный параметр OUT может (но не обязан) иметь значение в момент вызова подпрограммы. Однако это значение теряется, когда вы вызываете подпрограмму. Внутри подпрограммы формальный параметр OUT нельзя использовать в выражении; единственная операция, допустимая на таком параметре - это присваивание ему значения. Перед выходом из подпрограммы не забывайте явно присвоить значения параметрам OUT. В противном случае значения соответствующих фактических параметров будут не определены. При успешном выходе из подпрограммы PL/SQL присваивает значения фактическим параметрам. Однако, если вы выходите с необработанным исключением, PL/SQL НЕ присваивает значений фактическим параметрам.

IN OUT

Параметр IN OUT позволяет вам передавать в подпрограмму начальные значения и возвращать обновленные значения вызывающей программе. Внутри подпрограммы такой параметр выступает как инициализированная переменная. Поэтому ему можно присвоить значение, а его значение можно присваивать другим переменным. Иными словами, параметр IN OUT можно рассматривать как обычную переменную. Вы можете изменять его значение или обращаться к этому значению любыми способами, как показывает следующий пример с переписанной процедурой insert _ storage :

```
PROCEDURE insert_storage (ware_id1 IN NUMBER, goods_id1 IN NUMBER, quantity1 IN OUT
NUMBER) IS
Storage_id1 NUMBER(4);
```



```
Volume_goods NUMBER(6,2);
Volume_goods_sum NUMBER(6,2);
Volume_ware NUMBER(6,2);
Over_goods NUMBER(6,2);
BEGIN
SELECT volume INTO Volume_goods FROM goods WHERE goods_id= goods_id1;
Volume_goods_sum:= Volume_goods*quantity1;
SELECT volume_rest INTO Volume_ware FROM warehouses WHERE ware_id= ware_id1;
IF (Volume_ware< Volume_goods_sum) THEN
Over_goods:=(Volume_goods_sum-Volume_ware)/Volume_goods;
quantity1:= quantity1- Over_goods;
END IF;
INSERT INTO storages VALUES (stor_seq.nextval, ware_id1, goods_id1, quantity1, SYSDATE, NULL);
UPDATE Warehouses SET Volume_rest= Volume_rest- Volume_goods_sum WHERE
ware_id= ware_id1;
quantity1:=Over_goods;
COMMIT;
END insert_storage;
```

Теперь процедура получает номер склада, куда нужно определить товар, номер товара и его количество. Используя информацию о свободном объеме склада и подсчитав объем, необходимый для хранения товара, процедура определяет остаток товара, который необходимо поместить на другой склад. Информацию об остатке процедура возвращает через переменную quantity 1.

Фактический параметр, соответствующий формальному параметру IN OUT, должен быть переменной; он не может быть константой или выражением.

Умалчиваемые значения параметров

Как показывает следующий пример, вы можете инициализировать параметры с модой IN умалчиваемыми значениями. Это позволяет передавать подпрограмме различное число параметров, принимая или перекрывая умалчиваемые значения по вашему желанию. Более того, можно добавлять новые формальные параметры, не требуя каждый раз изменять все существующие вызовы данной подпрограммы.

```
PROCEDURE insert_ware (new_address VARCHAR2 DEFAULT 'KHARKOV',
new_volume NUMBER DEFAULT 1000,
new_volume_rest NUMBER DEFAULT 1000,
new_storage_have VARCHAR2 DEFAULT HANGAR') IS
BEGIN
IF (new_volume<>1000) THEN
INSERT INTO warehouses VALUES (ware_seq.NEXTVAL, new_address, new_volume, new_volume,
new_storage_have);
ELSE
```

```
INSERT INTO warehouses VALUES (ware_seq.NEXTVAL, new_address, new_volume,  
new_volume_rest, new_storage_have);  
END IF;  
END insert_ware;
```

Если фактический параметр не передан, используется умалчиваемое значение соответствующего формального параметра.

Рассмотрим следующие вызовы процедуры insert _ ware :

```
BEGIN  
...  
insert_ware;  
insert_ware ('KIEV');  
insert_ware ('KIEV', 2000);  
...  
END;
```

Первый вызов не передает никаких фактических параметров, так что используются все умолчания. Второй вызов передает один фактический параметр, так что умолчание используется для new _ volume , new _ volume _ rest , new _ storage _ have (т.е. мы создаем новый стандартный склад в Киеве). Третий вызов передает 2 параметра, так что умалчиваемые значения используются для new _ volume _ rest и new _ storage _ have . В большинстве случаев вы можете использовать позиционную нотацию для перекрытия умалчиваемых значений формальных параметров. Однако вы не можете пропустить формальный параметр, опустив для него соответствующий фактический параметр.

Например, следующий вызов логически некорректен, так как он ассоциирует фактический параметр 'refregirator' с формальным параметром new _ volume _ rest , а не с формальным параметром new _ storage _ have :

```
insert_ware ('KIEV', 2000, 'refregirator');-- некорректно
```

Нельзя обойти эту проблему, указав запятую вместо отсутствующего позиционного параметра. Например, следующий вызов незаконен:

```
insert _ ware (' KIEV ', 2000, , ' refregirator '); -- синтаксическая ошибка
```

В таких случаях вы должны использовать именную нотацию, например:

```
insert _ ware (' KIEV ', 2000, new _ storage _ have => ' refregirator ');
```

Не забывайте, что позиционную нотацию нельзя применять после именной.

Перекрытие имен

PL/SQL позволяет вам ПЕРЕКРЫВАТЬ имена подпрограмм. Иными словами, вы можете использовать одно и то же имя для нескольких различных подпрограмм, если только их формальные параметры различаются по количеству, порядку или семействам типов данных.

Перекрывающиеся имена подпрограмм могут появляться только в блоке, подпрограмме или пакете. Иными словами, нельзя перекрывать имен независимых подпрограмм. Нельзя перекрывать две подпрограммы, если их формальные параметры различаются лишь именами или модами параметров. Например, следующее перекрытие незаконно:

```
PROCEDURE delete_my (goods_id1 IN NUMBER) IS BEGIN ... END delete_my;  
PROCEDURE delete_my (ware_id1 IN NUMBER) IS BEGIN ... END delete_my;
```

Более того, нельзя перекрывать две подпрограммы, если их формальные параметры различаются лишь типами данных, причем эти типы данных относятся к одному и тому же семейству. Например, следующее перекрытие незаконно, потому что типы данных INTEGER и NUMBER входят в одно и то же семейство:

```
PROCEDURE delete_my (goods_id1 IN NUMBER) IS  
BEGIN  
...  
END delete_my;  
PROCEDURE delete_my (ware_id1 IN INTEGER) IS  
BEGIN  
...  
END delete_my;
```

Наконец, нельзя перекрывать две функции, если они различаются лишь типами данных результирующего значения, даже если эти типы данных относятся к разным семействам. Например, следующее перекрытие незаконно:

```
FUNCTION count_my (goods_id1 NUMBER) RETURN BOOLEAN IS  
BEGIN  
...  
END count_my;  
FUNCTION count_my (goods_id1 NUMBER) RETURN NUMBER IS  
BEGIN  
...  
END count_my;
```

Рекурсия

Рекурсия является мощным способом упрощения разработки алгоритмов. По существу, РЕКУРСИЯ означает обращение к самому себе. В рекурсивной математической последовательности каждый член выводится путем применения формулы к предыдущим членам.

Рассматривайте рекурсивный вызов как вызов некоторой другой подпрограммы, которая выполняет ту же задачу, что и ваша подпрограмма. Каждый рекурсивный вызов создает новые экземпляры всех объектов, объявленных в подпрограмме, включая параметры, переменные, курсоры и исключения. Аналогично, на каждом уровне рекурсивного спуска создаются новые экземпляры предложений SQL.

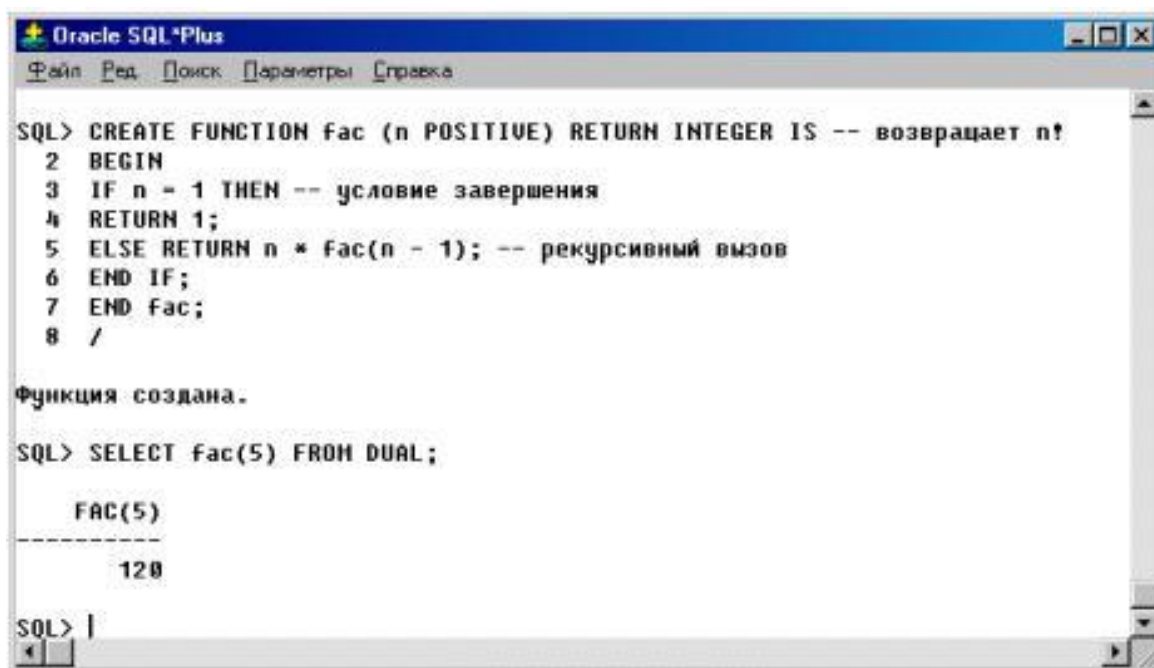
В рекурсивной подпрограмме должно быть по меньшей мере два пути логики: тот, который ведет к рекурсивному вызову, и тот, который не ведет к такому вызову. Иными словами, хотя бы один путь должен приводить к УСЛОВИЮ ЗАВЕРШЕНИЯ. В противном случае рекурсия (теоретически) продолжалась бы бесконечно. На практике, если рекурсивная подпрограмма начинает бесконечно вызывать саму себя, PL/SQL в конце концов переполняет свою память и возбуждает предопределенное исключение `STORAGE_ERROR`.

При решении некоторых задач программирования вы должны повторять последовательность предложений, пока не будет выполнено некоторое условие. Для этого можно использовать либо итерацию, либо рекурсию. Рекурсия применима там, где задача может быть разбита на более простые версии этой же задачи. Например, вы можете вычислить $3!$ следующим способом: $0! = 1$
 $1! = 1 * 0! = 1 * 1 = 1$ $2! = 2 * 1! = 2 * 1 = 2$ $3! = 3 * 2! = 3 * 2 = 6$

Чтобы реализовать этот алгоритм, вы могли бы написать следующую рекурсивную функцию, которая вычисляет факториал положительного целого числа:

```
FUNCTION fac (n POSITIVE) RETURN INTEGER IS -- возвращает n!  
BEGIN  
IF n = 1 THEN -- условие завершения  
RETURN 1;  
ELSE RETURN n * fac(n - 1); -- рекурсивный вызов  
END IF;  
END fac;
```

Вызов функции показан на рисунке 9.



```
Oracle SQL*Plus
Файл  Ред.  Поиск  Параметры  Справка

SQL> CREATE FUNCTION fac (n POSITIVE) RETURN INTEGER IS -- возвращает n!
2 BEGIN
3 IF n = 1 THEN -- условие завершения
4 RETURN 1;
5 ELSE RETURN n * fac(n - 1); -- рекурсивный вызов
6 END IF;
7 END fac;
8 /

Функция создана.

SQL> SELECT fac(5) FROM DUAL;

   FAC(5)
-----
      120

SQL> |
```

Рисунок 9 - Пример создания и вызова функции из утилиты SQL * Plus

При каждом рекурсивном вызове n уменьшается. В конце концов, n станет равным 1, и рекурсия остановится.

Будьте внимательны в выборе места, в которое вы помещаете рекурсивный вызов. Если вы помещаете его в курсорный цикл FOR или между предложениями OPEN и CLOSE, то при каждом вызове открывается очередной курсор. Как следствие, ваша программа может превысить лимит, устанавливаемый параметром инициализации ORACLE OPEN_CURSORS.

В отличие от итерации, рекурсия не является существенной для программирования на PL/SQL. Любая проблема, которая может быть решена рекурсией, может быть решена и итерацией. Концепцию итерации легче усвоить, поскольку примеры рекурсии не столь часты в повседневной жизни. Как следствие, итеративную версию подпрограммы обычно легче спроектировать, чем рекурсивную версию той же программы. Однако рекурсивная версия обычно проще, меньше, и потому ее легче отладить.

Встроенные функции PL / SQL

PL/SQL предоставляет много мощных функций, помогающих манипулировать данными. Возможно использование функций всюду, где допускаются выражения того же типа. Более того, можно вкладывать вызовы функций друг в друга.

Для встроенных функций выделяют следующие категории:

- функции сообщений об ошибках;
- числовые функции;
- символьные функции;
- функции преобразований;
- календарные функции;
- смешанные функции.

В предложениях SQL можно использовать все встроенные функции, за исключением функций сообщений об ошибках SQLCODE и SQLERRM. В процедурных предложениях можно использовать все встроенные функции, за исключением смешанной функции DECODE.

Групповые функции SQL AVG, MIN, MAX, COUNT, SUM не встроены в PL/SQL. Тем не менее, возможно их использование в предложениях SQL (но не в процедурных предложениях PL/SQL).

В дополнительных материалах для каждой встроенной функции приводятся ее аргументы, типы данных этих аргументов, и тип данных возвращаемого значения. Рассмотрим примеры встроенных функций.

Числовые функции принимают числовые аргументы и возвращают числовые значения. Трансцендентные функции включают тригонометрические, логарифмические и экспоненциальные функции.

function MOD (m NUMBER, n NUMBER) return NUMBER

Возвращает остаток от деления m на n. Если n равно 0, возвращается m.

function POWER (m NUMBER, n NUMBER) return NUMBER

Возвращает m в степени n. База m и степень n могут быть любыми числами, но если m отрицательно, то n должно быть целым.

function ROUND (m NUMBER [, n NUMBER]) return NUMBER

Округляет m до n десятичных позиций. Если n опущено, то m округляется до нуля десятичных позиций (т.е. до целого). Число n может быть отрицательным, что позволяет округлять до десятков, сотен и т.п.

Символьные функции принимают символьные аргументы. Некоторые символьные функции возвращают символьные значения, остальные возвращают числовые значения. Функции, возвращающие символьные значения, всегда возвращают значение типа VARCHAR2, с двумя исключениями. Функции UPPER и LOWER возвращают значение типа CHAR, если им передан

аргумент типа CHAR, и значение типа VARCHAR2 в противном случае.

function CONCAT (str1 VARCHAR2, str2 VARCHAR2) return VARCHAR2

Присоединяет строку str2 к строке str1 и возвращает результат. Если один из аргументов пуст, CONCAT возвращает другой аргумент. Если оба аргумента пусты, CONCAT возвращает NULL.

function LENGTH (str CHAR) return NUMBER function LENGTH (str VARCHAR2) return NUMBER

Возвращает число СИМВОЛОВ в строке str. Если строка str имеет тип CHAR, то в длину входят хвостовые пробелы. Если строка str пуста, LENGTH возвращает NULL.

function LOWER (str CHAR) return CHAR function LOWER (str VARCHAR2) return VARCHAR2

Возвращает строку str, в которой все буквы преобразованы в строчные.

function SUBSTR (str VARCHAR2, pos NUMBER [, len NUMBER]) return VARCHAR2

Возвращает подстроку строки str, начинающуюся с СИМВОЛЬНОЙ позиции pos и содержащую len символов (или, если число len опущено, все символы до конца строки str). Значение pos не может быть нулевым. Если значение pos отрицательно, SUBSTR подсчитывает символы от конца строки str. Число len должно быть положительным.

Функции преобразования конвертируют значение из одного типа данных в другой.

function TO_CHAR (dte DATE [, fmt VARCHAR2 [, nlsparms]]) return VARCHAR2

Преобразует дату dte в символьную строку типа VARCHAR2 в формате, заданном моделью формата fmt. (Допустимые модели формата приведены в описании функции TO_DATE.) Если вы опустите fmt, подразумевается умалчиваемый формат даты.

Аргумент nlsparms специфицирует язык, в котором возвращаются названия или сокращения месяцев и дней. Он имеет следующий вид:

'NLS_DATE_LANGUAGE = <язык>'

Если вы опустите nlsparms, то TO_CHAR использует умалчиваемый язык для текущей сессии.

function TO_DATE (str VARCHAR2 [, fmt VARCHAR2 [, nlsparms]]) return DATE function TO_DATE (num NUMBER, [, fmt VARCHAR2 [, nlsparms]]) return DATE

Преобразует строку str или число num в значение даты в формате, заданном fmt. Допустимые модели формата приведены в таблице 4:

Таблица 4 - Допустимые модели формата

Модель формата	Описание
CC,SCC	век (S префиксует даты до н.э. минусом)
YYYY,SYYYY	год (S префиксует даты до н.э. минусом)
IYYY	год в стандарте ISO
YYY,YY,Y	последние три, две или одна цифра года
IYY,IY,I	то же для года ISO
Y,YYY	год с запятой
YEAR,SYEAR	год прописью (S префиксует даты до н.э. минусом)
RR	последние две цифры года в новом веке
BC,AD	индикатор BC или AD
B.C.,A.D.	индикатор B.C. или A.D.
Q	квартал (1-4)
MM	месяц (1-12)
RM	римский номер месяца (I-XII)
MONTH	название месяца
MON	сокращенное название месяца
WW	неделя года (1-53)
IWW	неделя года (1-52 или 1-53) по ISO
W	неделя месяца (1-5)
DDD	день года (1-366)
DD	день месяца (1-31)
D	день недели (1-7)
DAY	название дня
DY	сокращенное название дня
J	юлианский день (число дней с 1 января 4712 г. до н.э.)
AM,PM	индикатор полудня
A.M.,P.M.	индикатор полудня с точками
HH,HH12	час дня (1-12)
HH24	час суток (0-23)
MI	минута (0-59)
SS	секунда (0-59)
SSSSS	секунд после полуночи (0-86399)

Если формат опущен, подразумевается, что строка `str` задана в умалчиваемом формате даты.

Если аргумент `fmt` имеет значение 'J' (юлианский день), то число `num` должно быть целым. Аргумент `nlsparms` специфицирует язык, в котором возвращаются названия или сокращения месяцев и дней. Он имеет следующий вид:


```
'NLS_DATE_LANGUAGE = <язык>'
```

Если опустить `nlsparms`, то `TO_DATE` использует умалчиваемый язык для текущей сессии.

Календарные функции принимают аргументы и возвращают значения в формате `DATE`, например:

function LAST_DAY (dte DATE) return DATE

Возвращает дату последнего дня месяца, содержащего дату `dte`.

function SYSDATE return DATE

Возвращает текущее значение даты/времени в системе. Эта функция не принимает аргументов.

Смешанные функции

function DECODE (expr, search1, result1 [, search2, result2] ... [default])

Значение выражения `expr` сравнивается с каждым из значений `search`. Если `expr` совпадает с каким-либо `search`, возвращается соответствующее значение `result`. Если ни одного совпадения не найдено, возвращается значение `default` (или `NULL`, если значение `default` опущено). `expr` может иметь любой тип данных, но значения `search` должны иметь тот же тип, как у `expr`. Возвращаемое значение принудительно приводится к тому типу данных, как у `result1`. Функция `DECODE` допускается только в предложениях `SQL`.

function USER return VARCHAR2

Возвращает имя текущего пользователя `ORACLE`. Эта функция не имеет аргументов.

Для закрепления на практике изученного теоретического материала рекомендуется выполнить лабораторную работу № 2.

Курсоры

Извлечение записей

Для выполнения предложений `SQL` и хранения их результатов `ORACLE` использует рабочие области, называемые **личными областями** `SQL`. Конструкт `PL/SQL`, называемый `КУРСОРОМ`, позволяет вам обращаться к личной области `SQL` по имени и извлекать из нее информацию. Есть два вида курсоров: **неявные** и **явные**. `PL/SQL` неявно объявляет курсор для любого

предложения манипулирования данными SQL, в том числе для запроса, возвращающего только одну строку. Для запросов, возвращающих более одной строки, вы можете явно объявить курсор, чтобы обрабатывать возвращаемые строки по одной.

Явные курсоры

Множество строк, возвращаемых запросом (активное множество), может состоять из нуля, одной или нескольких строк, в зависимости от того, сколько строк удовлетворяют вашим поисковым условиям. Когда запрос возвращает несколько строк, вы можете явно определить курсор для обработки этих строк.

Вы определяете курсор в декларативной части блока PL/SQL, подпрограммы или пакета путем задания его имени и специфицирования запроса. После этого вы манипулируете курсором при помощи трех команд: OPEN, FETCH и CLOSE.

Прежде всего, вы инициализируете курсор предложением OPEN, которое идентифицирует активное множество. Затем с помощью предложения FETCH вы извлекаете первую строку. Вы можете повторять FETCH неоднократно, пока не будут извлечены все строки. После обработки последней строки вы освобождаете курсор предложением CLOSE. Вы можете обрабатывать параллельно несколько запросов, объявив и открыв несколько курсоров.

Неявные курсоры

ORACLE неявно открывает курсор для обработки каждого предложения SQL, не ассоциированного с явно объявленным курсором. PL/SQL позволяет вам обращаться к последнему открытому неявному курсору через имя "SQL". Поэтому, хотя вы не можете манипулировать неявным курсором посредством предложений OPEN, FETCH и CLOSE, вы можете опрашивать атрибуты этого курсора, чтобы получить полезную информацию о последней выполненной операции SQL.

Объявление курсора

Ссылки вперед недопустимы в PL/SQL. Поэтому вы должны объявить курсор, прежде чем сможете сослаться на него в других предложениях. Объявляя курсор, вы даете ему имя и ассоциируете его с конкретным запросом.

В следующем примере объявляется курсор с именем c1:

```
DECLARE
CURSOR c1 IS SELECT quantity, begin_time, end_time FROM storages
WHERE ware_id=1;
...
BEGIN
```

...

Имя курсора - это необъявленный ранее идентификатор, а не переменная PL/SQL; его можно использовать только для обращения к запросу. Вы не можете присваивать значений имени курсора или использовать его в выражениях. Однако, имена курсоров подчиняются тем же правилам сферы, что и имена переменных.

Параметризованные курсоры

Курсоры могут принимать параметры, как показывает следующий пример. Параметр курсора может появляться в запросе всюду, где допускается появление константы.

```
CURSOR c1(ware_num IN NUMBER) IS SELECT quantity, begin_time, end_time FROM storages  
WHERE ware_id= ware_num;
```

Для объявления формальных параметров курсора используется синтаксис

```
CURSOR имя [ (параметр [, параметр, ...]) ] IS SELECT ...
```

где "параметр", в свою очередь, имеет следующий синтаксис:

```
имя_переменной [IN] тип_данных [{:= | DEFAULT} значение]
```

Формальные параметры курсора должны иметь моды IN. Для обсуждения мод параметров обратитесь к разделу "Моды параметров".

Как показывает следующий пример, вы можете инициализировать параметры курсора умалчиваемыми значениями. Таким способом вы можете передавать курсору различное число фактических параметров, принимая или перекрывая умалчиваемые значения по своему желанию. Более того, вы можете добавлять в курсор новые формальные параметры без необходимости отыскивать и исправлять все обращения к курсору в тексте программы.

```
DECLARE  
CURSOR c1 (ware_num NUMBER DEFAULT 1, goods_num NUMBER DEFAULT 5) IS SELECT ...  
...
```

Сфера параметров курсора локальна в этом курсоре, что означает, что к этим параметрам можно обращаться лишь в запросе, который участвует в объявлении курсора. Значения параметров курсора используются ассоциированным запросом в момент открытия курсора.

Открытие курсора

Открытие курсора предложением OPEN исполняет предложение SELECT и идентифицирует АКТИВНОЕ МНОЖЕСТВО, т.е. все строки, удовлетворяющие поисковым условиям запроса. Для курсоров, объявленных с фразой FOR UPDATE, предложение OPEN также осуществляет

блокировку этих строк. Пример предложения OPEN:

```
OPEN c1;
```

Предложение OPEN не извлекает строк активного множества. Для этого используется предложение FETCH.

Передача параметров

Курсору могут быть переданы параметры при открытии.

Например, при объявлении курсора

```
CURSOR c1 (ware_num NUMBER, goods_num NUMBER) IS SELECT ...
```

любое из следующих предложений открывает этот курсор:

```
OPEN c1( 3, 5);  
OPEN c1(new_ware, 5);  
OPEN c1(3, goods_num);
```

В последнем примере переменная, специфицированная в предложении OPEN, имеет такое же имя, что и параметр в объявлении курсора. Когда идентификатор goods _ num используется в объявлении курсора, он обозначает формальный параметр курсора. Когда этот же идентификатор используется вне объявления курсора, он обозначает переменную PL/SQL с этим именем. Однако, для ясности, рекомендуется использовать уникальные идентификаторы. Если вы не хотите принять умалчиваемые значения, каждому формальному параметру в объявлении курсора следует сопоставить соответствующий фактический параметр в предложении OPEN. Формальные параметры, объявленные с умалчиваемым значением, могут и не иметь соответствующих им фактических параметров. В этом случае они просто принимают свое умалчиваемое значение во время выполнения OPEN. Не забывайте, что формальные параметры курсора должны быть параметрами IN, так что они не могут возвращать значений фактическим параметрам. Чтобы сопоставить фактические параметры в предложении OPEN формальным параметрам в объявлении курсора, вы можете использовать позиционную или именную нотацию. Каждый фактический параметр должен иметь тип данных, совместимый с типом данных соответствующего формального параметра.

Извлечение данных из курсора

Предложение FETCH извлекает очередную строку из активного множества. При каждом выполнении FETCH курсор продвигается к следующей строке в активном множестве.

Пример предложения FETCH :

```
FETCH c1 INTO my_quantity, my_begin_time, my_end_time;
```

Для каждого значения столбца, извлекаемого запросом, ассоциированного с курсором, в списке INTO должна быть соответствующая переменная, имеющая совместимый с этим столбцом тип данных.

Любые переменные в фразе WHERE запроса, ассоциированного с курсором, вычисляются лишь в момент открытия курсора. Как показывает следующий пример, запрос может обращаться к переменным PL/SQL внутри своей сферы:

```
DECLARE
Volume1 warehouses.volume%TYPE;
Volume_rest1 warehouses.volume_rest%TYPE;
My_ware NUMBER(6,2):= 1;
CURSOR c1 (ware_id1 NUMBER) IS SELECT volume, volume_rest FROM warehouses
WHERE ware_id=ware_id1;

BEGIN
...
OPEN c1(my_ware); -- здесь my_ware равен 0
LOOP
FETCH c1 INTO volume1, volume_rest1;
EXIT WHEN c1%NOTFOUND;
...
my_ware := my_ware + 1; -- не окажет влияния на FETCH
END LOOP;
END;
```

Однако для каждой операции FETCH на одном и том же курсоре вы можете использовать собственный список INTO. Каждая FETCH извлекает строку и присваивает значения своим переменным INTO, как показывает следующий пример:

```
DECLARE
CURSOR c1 IS SELECT name FROM goods;
name1 goods.name%TYPE;
name2 goods.name%TYPE;
name3 goods.name%TYPE;

BEGIN
OPEN c1;
FETCH c1 INTO name1; -- извлекает первую строку
FETCH c1 INTO name2; -- извлекает вторую строку
FETCH c1 INTO name3; -- извлекает третью строку
...
CLOSE c1;
END;
```

Если вы выдадите FETCH, но в активном множестве больше нет строк, то значения

переменных в списке INTO не определены.

Заккрытие курсора

Предложение CLOSE деактивирует курсор, и активное множество становится неопределенным.

Пример предложения CLOSE:

```
CLOSE c1;
```

После того, как курсор закрыт, вы можете снова открыть его. Любая иная операция на закрытом курсоре возбуждает предопределенное исключение INVALID_CURSOR, которое соответствует ошибке ORACLE с кодом ORA-01001.

Курсорные циклы FOR

В большинстве ситуаций, требующих курсора, вы можете использовать курсорный цикл FOR, чтобы упростить кодирование. Курсорный цикл FOR неявно объявляет свой индекс цикла как запись типа %ROWTYPE, открывает курсор, в цикле извлекает строки из активного множества в поля записи, и закрывает курсор, когда все строки обработаны или когда вы выходите из цикла. Рассмотрим следующий блок PL/SQL, который анализирует данные, собранные в ходе лабораторных экспериментов, и помещает результаты во временную таблицу. Переменная c1rec, используемая как индекс в курсорном цикле FOR, неявно объявляется как запись, хранящая все элементы данных, возвращаемые одной операцией FETCH для курсора c1. Вы обращаетесь к элементам данных, хранящимся в полях записи, используя квалифицированные ссылки.

```
DECLARE
res itog.volume_use%TYPE;
CURSOR c1 IS SELECT ware_id, volume, volume_rest FROM warehouses;

BEGIN
FOR c1rec IN c1 LOOP
/* вычислить и сохранить результаты */
res := c1rec.volume - c1rec.volume_rest;
INSERT INTO itog VALUES (c1rec.ware_id, res);
END LOOP;
COMMIT;
END ;
```

Перед каждой итерацией курсорного цикла FOR, PL/SQL извлекает данные в неявно объявленную запись, которая эквивалентна следующей явно объявленной записи: c1rec

c1%ROWTYPE; Эта запись определена только внутри цикла. Вы не можете обращаться к ее полям вне цикла.

Например, следующая ссылка незаконна:

```
BEGIN
...
FOR c1rec IN c1 LOOP
...
END LOOP;
res := c1rec.volume - 100; -- незаконно
END;
```

Последовательность предложений внутри цикла выполняется один раз для каждой строки, которая удовлетворяет запросу, ассоциированному с курсором. Когда вы выходите из цикла, курсор закрывается автоматически. Это справедливо даже тогда, когда вы выходите из цикла принудительно, с помощью EXIT или GOTO, или когда внутри цикла возбуждается исключение.

Поля в неявно объявленной записи курсорного цикла FOR содержат значения столбцов из последней извлеченной строки. Эти поля имеют такие же имена, что и соответствующие столбцы в списке SELECT запроса. Однако что, если элемент списка SELECT является выражением? Рассмотрим следующий пример:

```
CURSOR c1 IS SELECT ware_id, volume - volume_rest FROM ...
```

В таких случаях вы должны предоставлять алиас для элемента списка SELECT. В следующем примере для вычисляемого элемента volume - volume _ rest используется алиас use :

```
CURSOR c1 IS SELECT ware_id, volume - volume_rest use FROM ...
```

При обращениях к соответствующему полю вы используете алиас в квалифицированной ссылке, например:

```
IF c1_rec.use < 1000 THEN ...
```

Атрибуты явного курсора

Каждый курсор, явно объявленный вами, имеет четыре атрибута: % NOTFOUND , % FOUND , % ROWCOUNT и % ISOPEN . Атрибуты позволяют вам получать полезную информацию о выполнении многострочного запроса. Для обращения к атрибуту просто присоедините его имя к имени курсора. Атрибуты явного курсора можно использовать в процедурных предложениях, но не в предложениях SQL.

Использование %NOTFOUND

Когда курсор открыт, строки, удовлетворяющие ассоциированному запросу, идентифицированы и образуют активное множество. Эти строки извлекаются операцией FETCH по одной за раз. Если последняя операция FETCH вернула строку, %NOTFOUND дает FALSE. Если последняя операция FETCH не смогла вернуть строку (так как активное множество исчерпано), %NOTFOUND дает TRUE.

Операция FETCH должна в конце концов исчерпать активное множество, так что, когда это происходит, никакого исключения не возбуждается. В следующем примере вы используете %NOTFOUND, чтобы выйти из цикла, когда FETCH не сможет вернуть строку:

```
LOOP
FETCH c1 INTO my_ware, my_use;
EXIT WHEN c1%NOTFOUND;
...
END LOOP;
```

Перед первой операцией FETCH атрибут %NOTFOUND дает NULL. Поэтому, если FETCH ни разу не выполнится успешно, вы никогда не выйдете из этого цикла. Причина в том, что предложение EXIT WHEN выполняется только в том случае, когда условие WHEN дает TRUE. Поэтому для безопасности вы можете предпочесть такой вариант предложения EXIT :

```
EXIT WHEN c1%NOTFOUND OR c1%NOTFOUND IS NULL;
```

Вы можете открыть несколько курсоров, а затем использовать %NOTFOUND, чтобы проверять, в каких курсорах еще есть строки. Если курсор не открыт, обращение к нему через %NOTFOUND возбуждает предопределенное исключение INVALID_CURSOR.

Использование %FOUND

%FOUND логически противоположен атрибуту %NOTFOUND. После открытия явного курсора, но до первой операции FETCH, %FOUND дает NULL. Впоследствии он дает TRUE, если последняя операция FETCH вернула строку, или FALSE, если последняя операция FETCH не смогла извлечь строку, так как больше нет доступных строк. Следующий пример использует %FOUND, чтобы выбрать одно из двух альтернативных действий:

```
LOOP
FETCH c1 INTO my_ware, my_use;
IF c1%FOUND THEN -- извлечение успешно
INSERT INTO itog VALUES (...);
ELSE EXIT;
...
END LOOP;
```

Вы можете открыть несколько курсоров, а затем использовать %FOUND, чтобы проверять, в каких курсорах еще есть строки. Если курсор не открыт, обращение к нему через %FOUND возбуждает предопределенное исключение INVALID_CURSOR.

Использование %ROWCOUNT

Когда вы открываете курсор, его атрибут %ROWCOUNT обнуляется. Перед первой операцией FETCH %ROWCOUNT возвращает 0. Впоследствии, %ROWCOUNT возвращает число строк, извлеченных операциями FETCH из активного множества на данный момент. Это число увеличивается, если последняя FETCH вернула строку. Следующий пример использует %ROWCOUNT, чтобы предпринять определенные действия, если выбрано более 10 строк:

```
LOOP
FETCH c1 INTO my_ware, my_use;
IF c1%ROWCOUNT > 10 THEN -- выбрано больше 10 строк
...
END IF;
END LOOP;
```

Вы можете открыть несколько курсоров, а затем использовать %ROWCOUNT, чтобы проверять, сколько строк извлечено из каждого курсора. Если курсор не открыт, обращение к нему через %ROWCOUNT возбуждает предопределенное исключение INVALID_CURSOR.

Использование %ISOPEN

%ISOPEN дает TRUE, если явный курсор открыт, и FALSE в противном случае. Следующий пример использует %ISOPEN для выбора действия:

```
IF c1%ISOPEN THEN -- курсор открыт
...
ELSE -- курсор закрыт, открыть его
OPEN c1;
END IF;
```

Таблица 5 показывает значения, возвращаемые атрибутами явного курсора перед и после выполнения операций OPEN, FETCH и CLOSE. Обратите внимание, что перед первой операцией FETCH атрибуты %NOTFOUND и %FOUND возвращают NULL на открытом курсоре.

Таблица 5 - Атрибуты курсора

		%NOTFOUND	%FOUND	%ROWCOUNT	%
OPEN	Перед	*	*	*	
	После	NULL	NULL	0	
Первая FETCH	Перед	NULL	NULL	0	
	После	FALSE	TRUE	1	

Промежуточные FETCH	Перед	FALSE	TRUE	1	
	После	FALSE	TRUE	**	
Последняя FETCH	Перед	FALSE	TRUE	**	
	После	TRUE	FALSE	**	
CLOSE	Перед	TRUE	FALSE	**	
	После	*	*	*	

* Возбуждает предопределенное исключение INVALID _ CURSOR

** Зависит от данных

Пример использования явного курсора см. в практическом занятии 2.

Атрибуты неявного курсора

Курсор SQL имеет четыре атрибута: % NOTFOUND , % FOUND , % ROWCOUNT и % ISOPEN . Эти атрибуты, присоединяемые к имени курсора (SQL), дают вам доступ к информации о выполнении предложений INSERT, UPDATE, DELETE и однострочных предложений SELECT INTO. Вы можете использовать атрибуты неявного курсора в процедурных предложениях, но не в предложениях SQL. Значения атрибутов неявного курсора всегда относятся к последней выполненной операции SQL, где бы она ни появилась. Соответствующее предложение SQL может даже находиться в другой сфере (например, в подблоке). Поэтому, если вы хотите сохранить значение такого атрибута, немедленно присвойте его булевской переменной.

Следующий пример показывает, как несоблюдение этого правила может привести к логической ошибке:

```
UPDATE storages SET quantity = quantity - 100 WHERE storage_id=5 AND quantity>100;
Buy _ goods (5, 20); -- вызов процедуры
IF SQL%NOTFOUND THEN -- рискованно !
...
END IF;
```

В этом примере рискованно полагаться на условие IF, потому что процедура Buy _ goods , возможно, изменила значение %NOTFOUND. Вы можете исправить этот код следующим образом, объявив в декларативной части булевскую переменную sql_notfound:

```
UPDATE storages SET quantity = quantity - 100 WHERE storage_id=5 AND quantity>100;
sql_notfound := SQL%NOTFOUND;
Buy _ goods (5, 20); -- вызов процедуры
IF sql_notfound THEN ...
END IF ;
```

Пока ORACLE автоматически не открыл курсор SQL, атрибуты неявного курсора возвращают NULL.

Использование %NOTFOUND дает TRUE, если INSERT, UPDATE или DELETE не обработала ни одной строки, или если операция SELECT INTO не возвратила ни одной строки. В противном случае %NOTFOUND дает FALSE.

В следующем примере вы используете %NOTFOUND, чтобы вставить новую строку, если операция обновления не нашла строки:

```
UPDATE storages SET quantity = quantity - 100 WHERE storage_id=5 AND quantity>100;
IF SQL%NOTFOUND THEN -- обновление не прошло
INSERT INTO storages VALUES (...);
END IF ;
```

Если предложение SELECT INTO не возвращает ни одной строки, возбуждается предопределенное исключение NO_DATA_FOUND, независимо от того, проверяете ли вы %NOTFOUND в следующей строке или нет.

%FOUND логически противоположен атрибуту %NOTFOUND. Перед выполнением предложения манипулирования данными SQL SQL%FOUND дает NULL. Впоследствии, этот атрибут дает TRUE, если операция INSERT, UPDATE или DELETE затронула хотя бы одну строку, или если операция SELECT INTO вернула хотя бы одну строку. В противном случае SQL%FOUND дает FALSE.

В следующем примере вы используете %FOUND, чтобы вставить строку при успехе операции удаления:

```
DELETE FROM storages WHERE ware_id = 5;
IF SQL%FOUND THEN -- удаление успешно
INSERT INTO temp VALUES ...;
END IF ;
```

Использование % ROWCOUNT SQLROWCOUNT возвращает число строк, на которое воздействовала операция INSERT, UPDATE или DELETE, или число строк, возвращенных операцией SELECT INTO. %ROWCOUNT возвращает 0, если операция INSERT, UPDATE или DELETE не затронула ни одной строки, или если операция SELECT INTO не возвратила ни одной строки.

Следующий пример использует %ROWCOUNT, чтобы предпринять определенные действия, если было удалено более 5 строк:

```
DELETE FROM storages WHERE ware_id = 5;
IF SQL % ROWCOUNT > 5 THEN -- удалено больше 5 строк
...
END IF;
```

Если операция SELECT INTO возвращает более одной строки, то возбуждается предопределенное исключение TOO_MANY_ROWS, а атрибут %ROWCOUNT устанавливается в

1, а HE в действительное число строк, удовлетворяющих запросу.

Использование %ISOPEN ORACLE автоматически закрывает неявный курсор после выполнения связанной с ним операции SQL. Поэтому атрибут SQL%ISOPEN всегда возвращает FALSE.

В практическом занятии 2 приведено еще несколько примеров использования курсоров для БД.

Исключительные ситуации

В PL/SQL условие ошибки называется «исключением». Исключения могут быть внутренне определены (исполнителем PL/SQL) или определены пользователем. В общем ситуация с исключительными ситуациями следующая:

- некоторые общие внутренне определенные исключения имеют предопределенные имена, такие как ZERO_DIVIDE и STORAGE_ERROR;
- другим внутренне определенным исключениям имена могут быть присвоены;
- вы можете определить ваши собственные исключения в декларативной части любого блока, подпрограммы или пакета PL/SQL. Например, вы можете определить исключение с именем Not_successful_storage для обработки приема на хранение партии товара, которая не может быть целиком помещена на предлагаемый склад. Итак, внутренние исключения имеют имена либо им имена могут быть присвоены. В отличие от внутренних исключений, пользовательские исключения должны иметь имена.

Когда возникает ошибка, соответствующее исключение возбуждается. Это значит, что нормальное выполнение останавливается, и управление передается на часть обработки исключений вашего блока или подпрограммы PL/SQL.

Внутренние исключения возбуждаются неявно (автоматически) системой исполнения; пользовательские исключения возбуждаются явно, посредством предложений RAISE, которые могут также возбуждать предопределенные исключения.

Для обработки возбуждаемых исключений вы пишете отдельные программы, называемые обработчиками исключений. После выполнения обработчика исключений исполнение текущего блока заканчивается, и окружающий блок продолжает свое выполнение со следующего предложения. Если окружающего блока нет (т.е. текущий блок не вложен в другой блок), то управление возвращается в хост-окружение.

Предопределенные исключения

Внутреннее исключение возбуждается неявно всякий раз, когда ваша программа PL/SQL

нарушает правило ORACLE или превышает установленный системой лимит. Каждая ошибка ORACLE имеет номер, однако исключения должны обрабатываться по их именам. Поэтому PL/SQL внутренне определяет некоторые распространенные ошибки ORACLE как исключения. Например, предопределенное исключение NO_DATA_FOUND возбуждается, когда предложение SELECT INTO не возвращает ни одной строки.

PL/SQL объявляет предопределенные исключения глобально, в пакете STANDARD, который определяет окружение PL/SQL. Поэтому вы не обязаны объявлять их сами. Вы можете писать обработчики для предопределенных исключений, используя имена, приведенные в таблице 6. Эта таблица показывает также и значения возврата функции SQLCODE.

Таблица 6 - Коды ошибок ORACLE

Имя исключения	Ошибка ORACLE	Код SQLCODE
CURSOR_ALREADY_OPEN	ORA-06511	-6511
DUP_VAL_ON_INDEX	ORA-00001	-1
INVALID_CURSOR	ORA-01001	-1001
INVALID_NUMBER	ORA-01722	-1722
LOGIN_DENIED	ORA-01017	-1017
NO_DATA_FOUND	ORA-01403	-100
NOT_LOGGED_ON	ORA-01012	-1012
PROGRAM_ERROR	ORA-06501	-6501
STORAGE_ERROR	ORA-06500	-6500
TIMEOUT_ON_RESOURCE	ORA-00051	-51
TOO_MANY_ROWS	ORA-01422	-1422
TRANSACTION_BACKED_OUT	ORA-00061	-61
ZERO_DIVIDE	ORA-01476	-1476
VALUE_ERROR	ORA-06502	-6502

Ниже приведены краткие описания предопределенных исключений:

CURSOR_ALREADY_OPEN

Возбуждается при попытке открыть уже открытый курсор. Вы должны закрыть (CLOSE) курсор, прежде чем открывать его повторно. Курсорный цикл FOR автоматически открывает свой курсор. Поэтому вы не можете войти в такой цикл, если данный курсор уже открыт. Нельзя также явно открывать курсор внутри цикла FOR.

DUP_VAL_ON_INDEX

Возбуждается, когда операция INSERT или UPDATE пытается создать повторяющееся значение в столбце, ограниченном опцией UNIQUE.

INVALID_CURSOR

Возбуждается, когда вызов PL/SQL специфицирует некорректный курсор (например, при попытке закрыть неоткрытый курсор).

INVALID_NUMBER

Возбуждается в предложении SQL, когда преобразование символьной строки в число сбивается из-за того, что строка не содержит правильного представления числа. Например, следующее предложение INSERT возбудит исключение INVALID_NUMBER, когда ORACLE попытается преобразовать 'HALL' в число:

```
INSERT INTO emp (empno, ename, deptno) VALUES ('HALL', 7888, 20);
```

В процедурных предложениях вместо этого исключения возбуждается VALUE_ERROR.

LOGIN_DENIED

Возбуждается при некорректном имени пользователя или пароле при попытке подключения к ORACLE.

NO_DATA_FOUND

Возбуждается, когда предложение SELECT INTO не возвращает ни одной строки, или при обращении к неинициализированной строке таблицы PL/SQL. Групповые функции SQL, такие как AVG или SUM, ВСЕГДА возвращают значение, даже если это значение есть NULL. Поэтому предложение SELECT INTO, вызывающее групповую функцию, никогда не возбудит исключение NO_DATA_FOUND. Поскольку NO_DATA_FOUND возбуждается, когда предложение SELECT INTO не возвращает строк, вы можете проверять значение SQL%NOTFOUND только в обработчике исключений. Однако, значение атрибута имя_курсора%NOTFOUND можно проверять после каждой операции FETCH. От операции FETCH ожидается, что в конце концов она не сможет вернуть очередную строку, так что, когда это происходит, никакого исключения не возбуждается.

NOT_LOGGED_ON

Возбуждается, когда ваша программа PL/SQL выдает вызов ORACLE, не будучи подключена к ORACLE.

PROGRAM_ERROR

Возбуждается, когда PL/SQL встретился с внутренней проблемой.

STORAGE_ERROR

Возбуждается, когда PL/SQL исчерпал доступную память, или когда память заперчена.

TIMEOUT_ON_RESOURCE

Возбуждается при возникновении таймаута, когда ORACLE ожидает ресурса.

TOO_MANY_ROWS

Возбуждается, когда предложение SELECT INTO возвращает больше одной строки.

TRANSACTION_BACKED_OUT

Обычно возбуждается, если удаленная часть транзакции была подвергнута неявному или явному откату. Причиной может быть несогласованность данных ORACLE в каких-нибудь узлах. В таких случаях выдайте ROLLBACK, а затем повторите транзакцию.

VALUE_ERROR

Возбуждается при возникновении арифметической ошибки, ошибки преобразования, ошибки усечения или ошибки ограничения. Например, VALUE_ERROR возбуждается при усечении строкового значения, присваиваемого переменной PL/SQL. (Однако при усечении строкового значения, присваиваемого хост-переменной, никакого исключения не возбуждается.) В процедурных предложениях VALUE_ERROR возбуждается при ошибке преобразования символьной строки в число. Например, следующее предложение присваивания вызовет RAISE_ERROR, когда PL/SQL попытается преобразовать 'HALL' в числовое значение:

```
DECLARE
my_volume NUMBER(6,2);
my_address VARCHAR2(20);
BEGIN
my_volume := 'KHARKOV';
...
```

В предложениях SQL в таких случаях возбуждается INVALID_NUMBER.

ZERO_DIVIDE

Возбуждается при попытке деления числа на 0.

Пользовательские исключения

Пользовательские исключения PL/SQL позволяют вам определять ваши собственные исключения. В отличие от внутренних исключений, пользовательские исключения должны быть объявлены и должны явно возбуждаться предложениями RAISE.

Вы можете объявлять исключения только в декларативной части блока, подпрограммы или пакета PL/SQL. Вы объявляете исключение, вводя его имя, за которым следует ключевое слово EXCEPTION.

В примере объявляется исключение с именем not _ type _ storage



```
DECLARE
not_type_storage EXCEPTION;
num _ stor NUMBER(5);
BEGIN
...
```

Объявления исключений выглядят так же, как и объявления переменных. Однако не забывайте, что исключение - это условие ошибки, но не объект. В отличие от переменных, исключениям нельзя присваивать значений, и они не могут использоваться в предложениях SQL. Однако к исключениям применимы те же правила сферы, что и к переменным.

Нельзя объявлять исключение дважды в одном и том же блоке. Однако вы можете объявить то же самое исключение в двух различных блоках. Исключения, объявленные в блоке, считаются локальными в этом блоке и глобальными во всех его подблоках. Поскольку блок может обращаться лишь к локальным или глобальным исключениям, окружающие блоки не могут ссылаться на исключения, объявленные в подблоках. Вы можете переобъявить глобальное исключение в подблоке. В таком случае локальное объявление имеет преимущество, и подблок не может обращаться к глобальному исключению (если только оно не было объявлено в помеченном блоке; в этом случае можно использовать синтаксис метка_блока.имя_исключения).

Следующий пример иллюстрирует правила сферы для исключений



```
DECLARE
not_type_storage EXCEPTION;
num_stor NUMBER;
BEGIN
... -- начало подблока
DECLARE
not_type_storage EXCEPTION; -- имеет преимущество в подблоке
num_stor NUMBER;
BEGIN
...
IF ... THEN RAISE not_type_storage; -- это не обрабатывается
END IF;
...
END; -- конец подблока
...
EXCEPTION WHEN past_due THEN -- это не относится к исключению,
```


... -- возбуждаемому в подблоке END;

Здесь окружающий блок не обрабатывает исключение, возбуждаемое в подблоке, потому что объявление `not _ type _ storage` в подблоке имеет преимущество. Два исключения `not _ type _ storage`, хотя их имена одинаковы, являются разными, точно так же, как различны две объявленные в этом примере переменные `num_stor`. Поэтому предложение `RAISE` в подблоке и фраза `WHEN` в окружающем блоке относятся к РАЗНЫМ исключениям. Чтобы заставить окружающий блок обрабатывать исключение, вы должны определить в нем обработчик `OTHERS`, либо удалить объявление исключения из подблока.

Для обработки непоименованных внутренних исключений вы должны использовать обработчик `OTHER` либо прагму `EXCEPTION_INIT`.

ПРАГМА - это директива (указание) компилятору. Прагмы (называемые также псевдоинструкциями) обрабатываются во время компиляции, а не во время выполнения. Они не изменяют смысла программы, а лишь поставляют информацию компилятору.

В PL/SQL, предопределенная прагма `EXCEPTION_INIT` сообщает компилятору имя исключения, которое вы ассоциируете с конкретным кодом ошибки ORACLE. Это позволяет вам обращаться к любому внутреннему исключению по имени, написав для него специальный обработчик. Вы кодируете прагму `EXCEPTION_INIT` в декларативной части блока, подпрограммы или пакета PL/SQL, используя следующий синтаксис:

```
PRAGMA EXCEPTION_INIT(имя_исключения, код_ошибки_ORACLE);
```

Здесь `имя_исключения` - это имя исключения, ранее уже объявленного в этом блоке.

Прагма должна появиться в той же декларативной части, что и соответствующее исключение, как показано в следующем примере:

```
DECLARE
insufficient_privileges EXCEPTION;
PRAGMA EXCEPTION_INIT(insufficient_privileges, -1031);
-- ORACLE возвращает код ошибки -1031, если, например,
-- вы пытаетесь обновить таблицу, для которой имеете
-- лишь полномочия SELECT.
BEGIN
...
EXCEPTION
WHEN insufficient_privileges THEN -- обработать ошибку
...
END;
```

Использование исключений для обработки ошибок имеет несколько преимуществ. Не имея таких средств, вы должны были бы проверять на ошибки выполнения при каждом выполнении команды, как показывает следующий пример:

```
BEGIN
SELECT ... -- проверить на ошибку 'no data found'
SELECT ... -- проверить на ошибку 'no data found'
SELECT ... -- проверить на ошибку 'no data found'
END ;
```

Обработка ошибок при этом не является ни четко отделенной от нормальной обработки, ни концептуально осмысленной. Если вы пренебрежете проверкой в одном месте, ошибка пройдет необнаруженной и может вызвать другие, внешне не связанные ошибки.

При помощи исключений вы можете обрабатывать ошибки, не кодируя многочисленных проверок, как показывает следующий пример:

```
BEGIN
SELECT ...
SELECT ...
SELECT ...
EXCEPTION
WHEN NO_DATA_FOUND THEN -- этот обработчик перехватывает
-- все ошибки 'no data found'
END;
```

Заметьте, как исключения улучшают читабельность, позволяя вам изолировать программы обработки ошибок. Основной алгоритм не затемняется алгоритмами восстановления от ошибок.

Исключения также повышают надежность. Вам нет необходимости беспокоиться о проверке ошибки в любой точке, где она может возникнуть. Просто добавьте соответствующий обработчик исключений в ваш блок PL/SQL. Если данное исключение когда-либо будет возбуждено в этом блоке (или любом его подблоке), у вас есть гарантия, что оно будет обработано.

Обработчики исключений

Когда возбуждается исключение, нормальное исполнение вашего блока PL/SQL или подпрограммы останавливается, и управление передается на обработчик исключений этого блока или подпрограммы, что оформляется следующим образом:

```
...
EXCEPTION
WHEN имя_исключения1 THEN
-- обработчик ряд_предложений1
WHEN имя_исключения2 THEN
-- другой обработчик ряд_предложений2
...
WHEN OTHERS THEN
-- необязательный обработчик ряд_предложений3
```

END;

Чтобы перехватывать возбуждаемые исключения, вы должны написать обработчики исключений. Каждый обработчик состоит, во-первых, из фразы WHEN, которая специфицирует имя исключения, и, во-вторых, из последовательности предложений, которые будут выполняться при возбуждении этого исключения. Эти предложения завершат исполнение блока или подпрограммы при исключении - управление больше не вернется в точку, где возникло исключение. Иными словами, вы не сможете возобновить работу с того места, где возникло исключение.

Необязательный обработчик исключений OTHERS - всегда последний обработчик исключений в блоке; он действует как обработчик для всех исключений, не перечисленных персонально в этом блоке. Таким образом, блок или подпрограмма может содержать только один обработчик OTHERS.

Рассмотрим следующий пример. Использование обработчика OTHERS гарантирует, что ни одно исключение не пройдет необработанным.



```
...  
EXCEPTION  
WHEN ... THEN  
-- обработать ошибку  
WHEN ... THEN  
-- обработать ошибку  
WHEN ... THEN  
-- обработать ошибку  
WHEN OTHERS THEN  
-- обработать все прочие ошибки  
END;
```

В следующем примере, процедура заполняет таблицу статистикой о том, на сколько процентов загружены склады. Если исходный объем склада в таблице был определен как нулевой, возбуждается предопределенное исключение ZERO_DIVIDE. Это останавливает нормальное исполнение процедуры и передает управление на обработчик исключений.



```
CREATE OR REPLACE PROCEDURE statistic_ware IS  
CURSOR c1 IS  
SELECT ware_id, volume, volume_rest FROM warehouses;  
vol NUMBER (6,2); -- переменная для подсчета процента занятости склада  
BEGIN  
FOR c1rec IN c1 LOOP  
Vol:=(c1rec.volume- c1rec.volume_rest)/(c1rec.volume/100); -- может вызвать  
-- ошибку " деление на 0"  
INSERT INTO Stat_table VALUES (stat_CUR.NEXTVAL, c1rec.ware_id, vol, SYSDATE);
```

```

END LOOP ;
COMMIT;
EXCEPTION -- здесь начинаются обработчики исключений
WHEN ZERO _ DIVIDE THEN -- обрабатывает "деление на 0"
raise_application_error(-20010, Exists warehouse with volume=0');
WHEN OTHERS THEN -- обрабатывает все прочие ошибки
ROLLBACK;
END statistic _ ware ; -- здесь заканчиваются обработчики исключений и вся процедура

```

На рисунке 10 приведен пример экранной формы, демонстрирующий вызов процедуры statistic _ ware с обработкой исключительной ситуации.

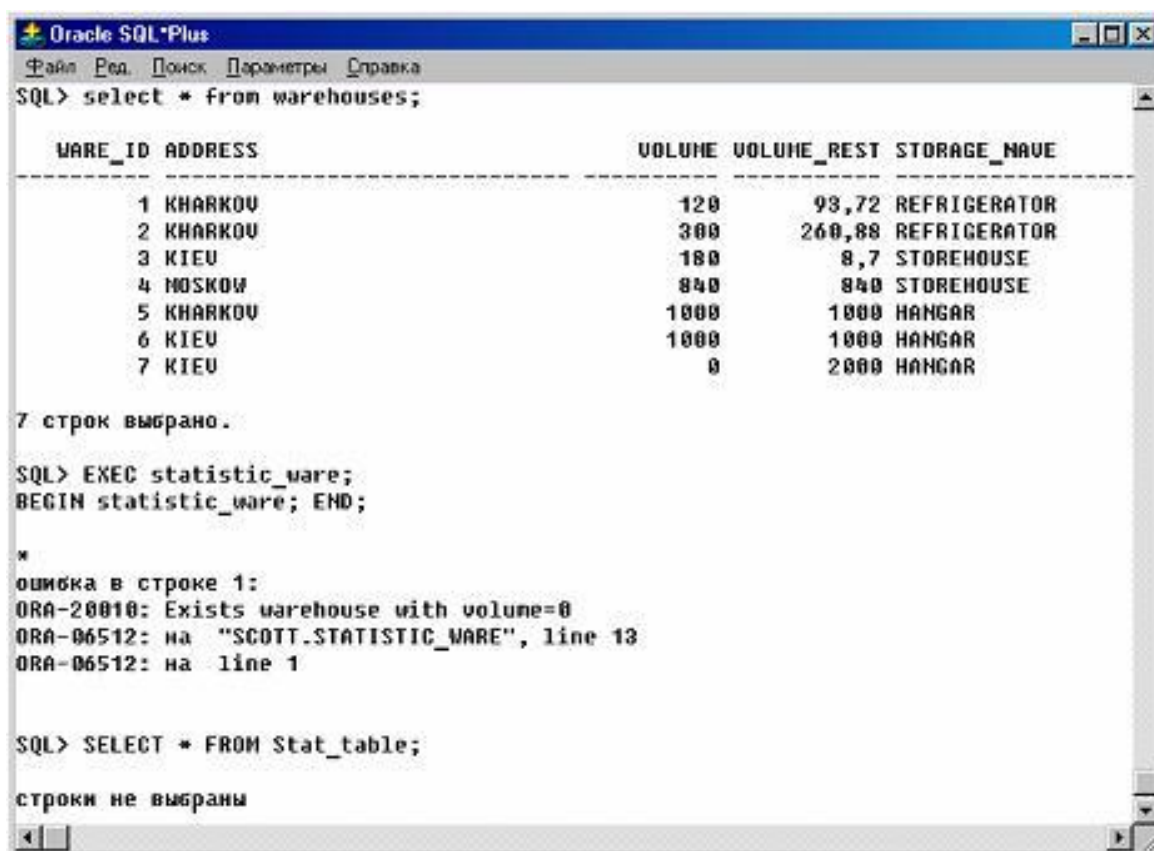


Рисунок 10 - Обработка исключительной ситуации

Необязательный обработчик OTHERS перехватывает все исключения, персонально не перечисленные в блоке.

Если вы хотите выполнять одну и ту же последовательность предложений для двух или более

исключений, перечислите имена этих исключений в фразе WHEN, разделяя их ключевым словом OR, как показано ниже:

```
...  
EXCEPTION  
WHEN over_volume OR under_volume OR VALUE_ERROR THEN  
-- обработать ошибку  
...  
END;
```

Если будет возбуждено любое из перечисленных в фразе WHEN исключений, соответствующий обработчик получит управление. Включение ключевого слова OTHERS в список имен исключений фразы WHEN НЕ ДОПУСКАЕТСЯ. Слово OTHERS может появиться только само по себе. Вы можете иметь сколько угодно обработчиков исключений, а каждый обработчик может ассоциировать последовательность предложений с любым списком исключений. Однако любое имя исключения может появиться лишь один раз в части обработки исключений блока или подпрограммы PL/SQL.

В обработке исключений действуют обычные правила сферы видимости идентификаторов, определенные для переменных PL/SQL, так что обработчик может обращаться лишь к локальным и глобальным переменным. Однако, когда исключение возбуждается внутри курсорного цикла FOR, соответствующий курсор неявно закрывается перед вызовом обработчика. Поэтому значения явных атрибутов курсора НЕДОСТУПНЫ в обработчике.

Переходы в обработчик и из него

Предложение GOTO нельзя использовать для перехода в обработчик исключений либо для перехода из обработчика исключений в текущий блок.

Например, следующее предложение GOTO НЕЗАКОННО



```
DECLARE  
val NUMBER(6,2);  
BEGIN  
SELECT volume / volume_rest INTO val FROM warehouses WHERE ware_id=6;  
-- может возникнуть ошибка деления на 0  
<<label1>>  
INSERT INTO stat VALUES (6, val);  
EXCEPTION  
WHEN ZERO_DIVIDE THEN  
per := 0;  
GOTO my_label ; -- незаконный переход в текущий блок  
END ;
```

Однако предложение GOTO можно использовать для перехода из обработчика исключения в окружающий блок.

Переобъявление предопределенных исключений

Вспомним, что PL/SQL объявляет свои предопределенные исключения глобально, в пакете STANDARD, так что вам не требуется объявлять их самим. Однако переобъявление предопределенного исключения не приведет к ошибке, так как ваше локальное объявление перекрывает глобальное внутреннее объявление.

Например, если вы объявите исключение с именем `invalid_number`, а затем PL/SQL внутренне возбудит предопределенное исключение `INVALID_NUMBER`, то обработчик, написанный вами для вашего `invalid_number`, не увидит внутреннего исключения. В таком случае вы должны обращаться к предопределенному исключению с помощью квалифицированной ссылки, как показывает следующий пример.



```
...  
EXCEPTION  
WHEN invalid_number OR STANDARD.INVALID_NUMBER THEN  
-- обработать ошибку  
...  
END;
```

Использование EXCEPTION_INIT

Для обработки непоименованных внутренних исключений вы должны использовать обработчик OTHER либо прагму `EXCEPTION_INIT`. ПРАГМА - это директива (указание) компилятору. Прагмы (называемые также псевдоинструкциями) обрабатываются во время компиляции, а не во время выполнения. Они не изменяют смысла программы, а лишь поставляют информацию компилятору.

В PL/SQL, предопределенная прагма `EXCEPTION_INIT` сообщает компилятору имя исключения, которое вы ассоциируете с конкретным кодом ошибки ORACLE. Это позволяет вам обращаться к любому внутреннему исключению по имени, написав для него специальный обработчик.

Вы кодируете прагму `EXCEPTION_INIT` в декларативной части блока, подпрограммы или пакета PL/SQL, используя следующий синтаксис:

```
PRAGMA EXCEPTION_INIT(имя_исключения, код_ошибки_ORACLE);
```

Здесь имя_исключения - это имя исключения, ранее уже объявленного в этом блоке. Прагма должна появиться в той же декларативной части, что и соответствующее исключение, как показано в следующем примере:



```
DECLARE
insufficient_privileges EXCEPTION;
PRAGMA EXCEPTION_INIT(insufficient_privileges, -1031);
-- ORACLE возвращает код ошибки -1031, если, например,
-- вы пытаетесь обновить таблицу, для которой имеете
-- лишь полномочия SELECT .
BEGIN
...
EXCEPTION
WHEN insufficient_privileges THEN
-- обработать ошибку
...
END;
```

Внутренние исключения возбуждаются неявно исполнительной системой, как и те пользовательские исключения, которые вы ассоциировали с кодами ошибок ORACLE с помощью прагмы EXCEPTION_INIT. Однако остальные пользовательские исключения должны возбуждаться явно, посредством предложений RAISE.

Использование предложения RAISE

Блоки и подпрограммы PL/SQL должны явно возбуждать исключение лишь в том случае, когда ошибка делает невозможным или нежелательным продолжение обработки. Вы можете закодировать предложение RAISE для данного исключения в любом месте сферы этого исключения.

В следующем примере вы возбуждаете в вашем блоке PL/SQL пользовательское исключение с именем out_of_volume.



```
DECLARE
out_of_volume EXCEPTION;
my_volume NUMBER(6,2);
BEGIN
...
IF my_volume > 1000 THEN
RAISE out_of_volume;
```

```
END IF;  
...  
EXCEPTION  
WHEN out_of_volume THEN  
-- обработать ошибку  
END;
```

Вы можете также явно возбуждать предопределенное исключение, как показывает следующий пример:

```
RAISE INVALID_NUMBER;
```

Это позволяет вам использовать обработчики, созданные для предопределенных исключений, для обработки других ошибок, как показывает следующий пример:

```
DECLARE  
Ware_id_my NUMBER(4);  
...  
BEGIN  
...  
IF Ware_id_my NOT IN (1, 2, 3) THEN  
RAISE INVALID_NUMBER;  
END IF;  
...  
EXCEPTION  
WHEN INVALID_NUMBER THEN  
ROLLBACK;  
...  
END;
```

Рассмотрим пример использования RAISE

Доработаем процедуру buy _ goods и определим в ней исключительную ситуацию, которая возбуждается при продаже товаров оптом, если количество продаваемого товара оказывается меньше границы в 10 единиц. Такую продажу нужно запретить - это не опт, а пользователя, проводящего эту операцию, зафиксировать в таблице Retail _ buy (Buy _ id , goods _ id , quantity , Date _ buy , Seller).



```
CREATE OR REPLACE PROCEDURE buy_goods (good_id1 NUMBER, quantity1 NUMBER) IS  
Storage _ id 1 NUMBER (4); -- переменная для номера партии  
ware _ id 1 NUMBER (4); -- переменная для номера склада  
vol NUMBER (6,2); -- переменная для объема единицы товара  
Small _ quantity EXCEPTION ;  
BEGIN  
-- проверка на исключительную ситуацию
```



```
IF (quantity1<10) THEN
RAISE Small_quantity;
END IF;
SELECT storage_id, ware_id INTO storage_id1, ware_id1 FROM storages WHERE
begin_time IN (SELECT MIN(begin_time) FROM storages
WHERE Goods_id=Good_id1 AND quantity>quantity1);
-- определяем объем единицы покупаемого товара
SELECT volume INTO vol FROM goods WHERE goods_id=good_id1;
-- фиксируем покупку части партии товара
UPDATE storages SET quantity = quantity - quantity1 WHERE storage_id = storage_id1;
--фиксируем освобождение соответствующего объема на складе
UPDATE warehouses SET volume_rest = volume_rest + quantity1* vol WHERE ware_id=ware_id1;
EXCEPTION
WHEN NO_DATA_FOUND THEN
INSERT INTO opt_audit VALUES (OPT_CUR.NEXTVAL, good_id1, 'No such goods');
COMMIT;
WHEN Small_quantity THEN
INSERT INTO Retail_buy VALUES (buy_CUR.NEXTVAL, good_id1, quantity1, SYSDATE, USER);
COMMIT;
END buy_goods;
```

Как распространяются исключения

Что происходит, если в текущем блоке нет обработчика исключений для возбужденного исключения? В этом случае блок завершается, а исключение ПРОДВИГАЕТСЯ в окружающий блок. Теперь окружающий блок становится текущим, а исключение воспроизводит себя в нем. Если обработчик исключения не находится и в этом блоке, процесс поиска повторяется. Если текущий блок не имеет окружающего блока, PL/SQL возвращает ошибку "необрабатываемое исключение" в хост-окружение.

Использование процедуры raise_application_error

Пакет с именем DBMS_STANDARD, входящий в состав Процедурного расширения базы данных, предоставляет средства языка, которые помогают вашему приложению взаимодействовать с ORACLE. Этот пакет включает процедуру raise_application_error, которая позволяет вам выдавать определенные вами сообщения об ошибках из хранимой подпрограммы или триггера базы данных. Вызывающее приложение получает при этом исключение PL/SQL, которое оно может обработать с помощью функций сообщений об ошибках SQLCODE и SQLERRM. Более того, оно может использовать прагму EXCEPTION_INIT, чтобы сопоставить специфические номера ошибок, которые возвращает raise_application_error, своим собственным исключениям.

Необработанные исключения

Как объяснялось выше, при невозможности найти обработчик для возбужденного исключения PL/SQL возвращает ошибку "необработанное исключение" в хост-окружение, которое определяет, что делать дальше.

Например, в среде прекомпиляторов ORACLE выполняется откат всех изменений в базе данных, сделанных сбившимся предложением SQL или блоком PL/SQL.

Необработанные исключения могут влиять на транзакции. Перед выполнением блока PL/SQL или хранимой подпрограммы ORACLE устанавливает неявную точку сохранения. Если блок или подпрограмма сбивается в результате необработанного исключения, ORACLE осуществляет откат к этой точке сохранения. Тем самым отменяется вся работа, сделанная блоком или подпрограммой.

Необработанные исключения могут также влиять на подпрограммы. При успешном выходе из подпрограммы PL/SQL присваивает значения параметрам OUT. Однако, если вы выходите в результате необработанного исключения, значения параметрам OUT НЕ присваиваются. Кроме того, как уже сказано, если подпрограмма сбивается в результате необработанного исключения, ORACLE неявно отменяет ее работу. Однако, если подпрограмма выдала COMMIT до возбуждения необрабатываемого исключения, отменена будет лишь неподтвержденная часть работы.

Вы можете избежать необрабатываемых исключений, кодируя обработчик OTHERS на самом верхнем уровне каждого блока PL/SQL и подпрограммы.

Полезные приемы

В этом разделе вы узнаете два полезных приема: как продолжить работу после исключения и как повторить транзакцию.



Продолжение работы после возбуждения исключения

▼ Подробнее

Обработчик исключений позволяет вам исправить ошибку, которая иначе могла бы стать фатальной, до выхода из блока. Однако если завершить обработчик, то блок больше не получит управления. Вы не можете вернуться в текущий блок из обработчика исключений. Тем не менее, существует способ обработать исключение для предложения, а затем продолжить работу со следующего предложения. Для этого просто поместите предложение в его собственный подблок вместе с его собственными обработчиками исключений, как показано в следующем примере:

```
DECLARE
val NUMBER(6,2);
BEGIN
val:=0;
```

```
----- начало подблока -----  
BEGIN  
SELECT volume / volume_rest INTO val FROM warehouses WHERE ware_id=6;  
-- может возникнуть ошибка деления на 0  
EXCEPTION  
WHEN ZERO_DIVIDE THEN  
val:= 0;  
END;  
----- конец подблока -----  
INSERT INTO stat VALUES (6, val);  
EXCEPTION  
END ;
```

Если в подблоке произойдет ошибка, локальный обработчик сможет обработать это исключение. После завершения подблока окружающий блок продолжит выполнение с той точки, где завершился подблок.

В последнем примере, если предложение SELECT INTO вызовет исключение ZERO_DIVIDE, локальный обработчик перехватит его и установит нулевое значение переменной val . После завершения обработчика подблок завершится, и выполнение продолжится с предложения INSERT.

Повторение транзакции

▼ Подробнее

Когда возникает исключение, вы можете захотеть повторить транзакцию, вместо того чтобы отказаться от нее. Для этого существует простой способ. Сначала заключите транзакцию в подблок. Затем поместите этот подблок в цикл, который повторяет транзакцию.

Рассмотрим следующий пример.

Прежде чем начать транзакцию, вы отмечаете точку сохранения. Если транзакция выполнится успешно, вы выполняете COMMIT и выходите из цикла. Если транзакция сбивается, управление передается обработчику исключений, в котором выполняете откат к точке сохранения, а затем пытаетесь исправить проблему.

Когда обработчик исключений заканчивается, подблок завершается. После этого управление передается на предложение LOOP в окружающем блоке, подблок исполняется повторно, и транзакция повторяется.

Используя цикл FOR или WHILE, вы можете лимитировать число попыток.

```
DECLARE  
name CHAR(20);  
ans1 CHAR(3);
```

```
ans2 CHAR(3);
ans3 CHAR(3);
suffix NUMBER := 1;
BEGIN
...
LOOP
-- можно написать " FOR i IN 1..10 LOOP ", чтобы
-- ограничиться максимально десятью попытками
----- начало подблока -----
BEGIN
...
SAVEPOINT start _ transaction ; -- точка сохранения
/* Удалить результаты опроса. */
DELETE FROM results WHERE answer1 = 'NO';
/* Добавить имя и ответы респондента. */
INSERT INTO results VALUES (name, ans1, ans2, ans3);
/* Это может дать исключение DUP_VAL_ON_INDEX, *
* если два респондента имеют одинаковые имена,*
* так как индекс по столбцу name уникальный. */
COMMIT;
EXIT;
EXCEPTION
WHEN DUP_VAL_ON_INDEX THEN
ROLLBACK TO start_transaction; -- откат
suffix := suffix + 1; -- попробуем исправить имя
name := name || TO_CHAR(suffix);
...
END;
----- конец подблока -----
END LOOP;

END;
```

Использование функций SQLCODE и SQLERRM

В обработке исключений можно использовать функции SQLCODE и SQLERRM, чтобы узнать, какая ошибка произошла и получить сообщение об ошибке.

Для внутренне определенных исключений, SQLCODE возвращает номер ошибки ORACLE, передавшей управление обработчику. Этот номер отрицателен, исключая случай ошибки ORACLE "no data found", когда SQLCODE возвращает +100.

SQLERRM возвращает сообщение, ассоциированное с возникшей ошибкой ORACLE. Это сообщение начинается с кода ошибки ORACLE.

Для пользовательских исключений, SQLCODE возвращает +1, а SQLERRM возвращает сообщение User-Defined Exception, если вы не использовали прагму EXCEPTION_INIT, чтобы ассоциировать ваше исключение с номером ошибки ORACLE; в этом случае SQLCODE возвращает этот номер ошибки, а SQLERRM возвращает соответствующее сообщение об ошибке. Заметим, что максимальная длина сообщения ORACLE об ошибке составляет 512 символов, включая код ошибки, вложенные сообщения и подстановки, такие как имена таблиц и столбцов. Если не возбуждено никакое исключение, то SQLCODE возвращает 0, а SQLERRM возвращает сообщение ORA-0000: normal, successful completion

Вы можете передать функции SQLERRM номер ошибки; в этом случае SQLERRM возвратит сообщение, ассоциированное с этим номером ошибки.

Номер ошибки, передаваемый SQLERRM, должен быть отрицателен. Нулевой код, передаваемый SQLERRM, всегда возвращает сообщение

ORA-0000: normal, successful completion

Передача SQLERRM положительного номера ошибки (за исключением +100) всегда возвратит сообщение

User-Defined Exception

а передача SQLERRM кода +100 возвратит сообщение

ORA -01403: no data found .

В следующем примере, SQLERRM возвратит не то, что ожидалось, потому что передается положительное значение вместо отрицательного:

```
DECLARE
msg CHAR(100);
BEGIN
FOR num IN 1..9999 LOOP
msg := SQLERRM(num); -- надо задавать SQLERRM(-num)
INSERT INTO errors VALUES (msg);
END LOOP;
END;
```

Вы не можете использовать функции SQLCODE и SQLERRM непосредственно в предложениях SQL. Например, следующее предложение незаконно:

```
INSERT INTO errors VALUES (SQLCODE, SQLERRM);
```

Вместо этого вы должны присвоить значения этих функций локальным переменным, а затем использовать эти переменные в ваших предложениях SQL, как показывает следующий пример:

```
DECLARE
err_num NUMBER;
err_msg CHAR(100);
BEGIN
...
EXCEPTION
...
WHEN OTHERS THEN err_num := SQLCODE;
err_msg := SUBSTR(SQLERRM, 1, 100);
INSERT INTO errors VALUES (err_num, err_msg);
END;
```

Строковая функция SUBSTR() гарантирует, что возможное усечение при присваивании переменной err_msg не возбudit исключения VALUE_ERROR. Функции SQLCODE и SQLERRM особенно полезны в исключении OTHERS, потому что они позволяют установить, какое внутреннее исключение было возбуждено.

Повторное возбуждение исключения

Иногда вам требуется повторно возбудить исключение, т.е. сначала обработать его локально, а затем заставить его продвинуться в окружающий блок. Например, вы можете захотеть отменить транзакцию в текущем блоке, а потом зарегистрировать ошибку в окружающем блоке.

Чтобы повторно возбудить исключение, поместите предложение RAISE в локальный обработчик этого исключения, как показано в следующем примере:

```
DECLARE
out_of_volume EXCEPTION;
BEGIN
... ----- начало подблока -----
BEGIN
...
IF ... THEN RAISE out_of_volume; -- возбудить исключение
END IF;
...
EXCEPTION
WHEN out_of_volume THEN -- обработать ошибку RAISE;
-- повторно возбудить текущее исключение ...
END; ----- конец подблока -----
EXCEPTION
WHEN out_of_volume THEN -- обработать ошибку иным способом
END;
```

Если опустить имя исключения в предложении RAISE (что допускается только в обработчике исключений), то подразумевается текущее исключение

Исключения могут также возбуждаться некорректными выражениями инициализации в объявлениях.

Например, следующее объявление неявно возбуждает исключение VALUE_ERROR, потому что limit не может хранить числа, большие 999:

```
DECLARE
limit CONSTANT NUMBER(3):= 5000; -- возбуждает исключение VALUE_ERROR
BEGIN
...
EXCEPTION
WHEN VALUE _ ERROR THEN -- не перехватит это исключение
...
END;
```

Обработчики в текущем блоке не могут перехватывать исключений, возбужденных в объявлениях, потому что такое исключение НЕМЕДЛЕННО продвигается в окружающий блок.

Лишь одно исключение в каждый момент времени может быть активно в части обработки исключений блока или подпрограммы. Так, исключение, возбужденное внутри обработчика, немедленно продвигается в окружающий блок, который просматривается на предмет обнаружения обработчика для вновь возбужденного исключения. С этого момента исключение продвигается обычным образом.

Пакеты

Пакет - это объект базы данных, который группирует логически связанные типы, программные объекты и подпрограммы PL/SQL. Пакеты обычно состоят из двух частей, спецификации и тела, хотя иногда в теле нет необходимости.

Спецификация пакета - это интерфейс с вашими приложениями; она объявляет типы, переменные, константы, исключения, курсоры и подпрограммы, доступные для использования в пакете.

Тело пакета полностью определяет курсоры и подпрограммы, тем самым реализует спецификацию пакета. В отличие от подпрограмм, пакеты нельзя вызывать, передавать им параметры или вкладывать их друг в друга.

В остальном формат пакета аналогичен формату подпрограммы:

```
PACKAGE имя IS -- спецификация (видимая часть)
-- объявления общих типов и объектов
-- спецификации подпрограмм
END [имя];
PACKAGE BODY имя IS -- тело (скрытая часть)
```

```
-- объявления личных типов и объектов
-- тела подпрограмм
[BEGIN
-- предложения инициализации]
END [имя];
```

Спецификация содержит **общие** объявления, которые видимы вашему приложению.

Тело содержит детали реализации и **личные** объявления, которые скрыты от вашего приложения. Представляйте себе спецификацию как функциональный интерфейс, а тело - как "черный ящик"

Пакеты создаются интерактивно в SQL*Plus или SQL*DBA с помощью команд CREATE PACKAGE и CREATE PACKAGE BODY .

В следующем примере пакетируются тип записи, курсор и две процедуры работы с товарами:

```
CREATE PACKAGE p_goods AS -- спецификация
TYPE Good_RecTyp IS RECORD (goods_id INTEGER, Volume Real, Price REAL);
CURSOR Goods_cur RETURN Good_RecTyp;
PROCEDURE change_price (goods_id1 NUMBER, price1 NUMBER);
PROCEDURE delete_ (goods_id1 NUMBER);
END p_goods;
```

```
CREATE PACKAGE BODY p_goods AS -- тело
CURSOR Goods_cur RETURN Good_RecTyp IS
SELECT goods_id, Volume, Price FROM goods;
PROCEDURE change_price (goods_id1 NUMBER, price1 NUMBER) IS
```

```
BEGIN
UPDATE goods SET price = price1 WHERE goods_id = goods_id1;
END change _ price ;
PROCEDURE delete_ (goods_id1 NUMBER) IS
BEGIN
DELETE FROM goods WHERE goods_id = goods_id1;
END delete_;
END p_goods;
```

Видимыми и доступными для приложений являются лишь объявления в спецификации пакета. Детали реализации в теле пакета скрыты и недоступны. Поэтому вы можете исправлять тело (реализацию), не перекомпилируя вызывающих программ.

Спецификация пакета

Спецификация пакета содержит общие объявления. Сфера этих объявлений локальна для

вашей схемы в базе данных и глобальна для самого пакета. Таким образом, объявленные объекты доступны из вашего приложения и из любого места в пакете.

Спецификация перечисляет ресурсы пакета, доступные приложениям. Она содержит всю информацию, необходимую вашему приложению для использования этих ресурсов.

Например, следующее объявление показывает, что функция с именем `fac` принимает один аргумент типа `INTEGER` и возвращает значение типа `INTEGER`:

```
FUNCTION fac (n INTEGER) RETURN INTEGER; -- возвращает n!
```

Это вся информация, необходимая вам для вызова данной функции. Вам нет необходимости рассматривать фактическую реализацию функции `fac` (например, итеративна она или рекурсивна).

Только подпрограммы и курсоры имеют реализацию, или ОПРЕДЕЛЕНИЕ. Поэтому, если спецификация пакета объявляет лишь типы, константы, переменные и исключения, тело пакета не нужно.

Приведем пример такого пакета:

```
-- пакет, состоящий только из спецификации
PACKAGE opt_data IS
TYPE TimeTyp IS RECORD (minute SMALLINT, hour SMALLINT);
TYPE DateTyp IS RECORD (day INTEGER, mounth VARCHAR2, year INTEGER, time TimeTyp);
min_quant CONSTANT INTEGER := 10;
max_volume REAL:=1000.00;
not_type_storage EXCEPTION;
END opt_data ;
```

Пакет `opt_data` не нуждается в теле, потому что типы, константы, переменные и исключения не требуют реализации. Такие пакеты позволяют вам определять глобальные переменные - для использования подпрограммами и триггерами - которые существуют на протяжении всей сессии.

Обращение к содержимому пакета

Для обращения к типам, объектам и подпрограммам, объявленным в спецификации пакета, используются квалифицированные ссылки:

`имя_пакета.имя_типа`

`имя_пакета.имя_объекта`

`имя_пакета.имя_подпрограммы`

Вы можете обращаться к содержимому пакета из триггеров базы данных, хранимых

подпрограмм, встроенных блоков PL/SQL, а также анонимных блоков PL/SQL, посылаемых в ORACLE интерактивно через SQL*Plus или SQL*DBA.

В следующем примере вы обращаетесь к пакетированной переменной `max _ volume`, которая объявлена в пакете `opt _ data` :

```
DECLARE
new_volume REAL;
...
BEGIN
...
IF new_volume > max_volume THEN
...
END IF;
...
END;
```

Тело пакета

Тело пакета реализует спецификацию пакета. Оно содержит определения всех курсоров и подпрограмм, объявленных в спецификации пакета. Не забывайте, что любая подпрограмма, определенная в теле пакета, доступна извне пакета лишь в том случае, если ее спецификация также появляется в спецификации пакета. Тело пакета может также содержать личные объявления, которые определяют типы и объекты, необходимые для внутренней работы пакета. Сфера таких объявлений локальна в теле пакета. Поэтому объявленные здесь типы и объекты недоступны нигде, кроме тела пакета. В отличие от спецификации пакета, декларативная часть тела пакета может содержать тела подпрограмм. За декларативной частью тела пакета может следовать необязательная часть инициализации, которая обычно содержит предложения, инициализирующие некоторые из переменных, ранее объявленных в пакете. Часть инициализации пакета не играет большой роли, потому что, в отличие от подпрограмм, пакет нельзя вызывать или передавать ему параметры. Следовательно, часть инициализации пакета обрабатывает лишь один раз, при первом обращении к пакету.

Преимущества пакетов

Пакеты предлагают несколько преимуществ: модульность, облегчение проектирования приложений, скрытие информации, расширенная функциональность и лучшая производительность.

▼ Подробнее

Модульность

Пакеты позволяют вам инкапсулировать логически связанные типы, объекты и подпрограммы в поименованный модуль PL/SQL. Каждый пакет легко понять, а интерфейсы между пакетами просты, ясны и хорошо определены. Это облегчает разработку приложений.

Облегчение проектирования

Все, что вам надо изначально знать при проектировании приложения - это информация интерфейса для спецификации пакета. Вы можете кодировать и компилировать спецификацию без тела. После того, как спецификация откомпилирована, хранимые подпрограммы, обращающиеся к пакету, также могут быть откомпилированы. Вы не обязаны полностью определять тела пакетов до тех пор, пока не будете готовы к реализации деталей приложения.

Скрытие информации

С помощью пакетов вы можете указывать, какие типы, объекты и подпрограммы являются общими (видимыми и доступными) или личными (скрытыми и недоступными). Например, если пакет содержит четыре подпрограммы, то три из них могут быть общими, а одна личной. Пакет скрывает определение личной подпрограммы, так что лишь этот пакет, а не ваши приложения, будет затронут, если это определение изменится. Это упрощает сопровождение и развитие. Кроме того, скрывая детали реализации от пользователей, вы защищаете целостность вашей базы данных.

Расширенная функциональность

Пакетированные общие переменные и курсоры продолжают существовать в течение всей сессии. Поэтому они могут совместно использоваться всеми процедурами, выполняющимися в данном окружении. Кроме того, через них можно передавать данные между транзакциями без необходимости записывать такие данные в базу данных.

Улучшенная производительность

Когда вы вызываете пакетированную подпрограмму первый раз, в память загружается весь пакет. Поэтому последующие вызовы других подпрограмм этого пакета не требуют операций ввода-вывода. Помимо этого, пакеты останавливают каскадные зависимости, и тем самым избегают излишних перекомпиляций. Например, когда вы изменяете определение независимой функции, ORACLE должен перекомпилировать все хранимые подпрограммы, которые вызывают эту функцию. Однако, когда вы изменяете определение пакетированной функции, перекомпиляция вызывающих подпрограмм не требуется, потому что они не зависят от тела пакета.

Применения пакетов

Пакеты используются для определения взаимосвязанных процедур, переменных и курсоров. С помощью пакетов часто достигаются преимущества в следующих областях:

- инкапсуляция связанных процедур и переменных;

- разделение спецификации пакета и тела этого пакета;
- объявление общих и личных процедур, переменных, констант и курсоров;
- определение переменных, сохраняющих значение между вызовами;
- улучшение производительности.

▼ Подробнее

Инкапсуляция

Пакеты позволяют вам инкапсулировать логически связанные типы данных, хранимые процедуры, переменные и т.д. в единую поименованную единицу, хранящуюся в базе данных. Каждый пакет легко понять, а интерфейсы между пакетами просты, ясны и хорошо определены. Это облегчает разработку приложений. Инкапсуляция процедурных конструкторов в пакет также облегчает управление привилегиями. Назначение привилегии на использование пакета делает все конструкторы в этом пакете доступными тому, кто получает привилегию.

Общие и личные данные и процедуры

С помощью пакетов вы можете указывать, какие типы, объекты и подпрограммы являются:

- общими (Непосредственно видимыми пользователю пакета);
- личными (Скрытыми от пользователя пакета).

Например, пакет может содержать десять процедур. Однако этот пакет может быть определен так, что лишь три процедуры являются общими, и потому доступными для выполнения пользователю пакета; остальные процедуры являются личными, и могут использоваться лишь процедурами внутри самого пакета.

Сравнение личных и общих объектов

Если в теле пакета объявляется переменная, то она может использоваться только внутри пакета. Следовательно, код PL/SQL вне пакета не может обращаться к этой переменной. Такие элементы называются ЛИЧНЫМИ. Однако элементы, объявленные в спецификации пакета являются видимыми вне пакета. Следовательно, код PL/SQL вне пакета может обращаться к таким элементам. Такие элементы называются ОБЩИМИ. Если вам надо поддерживать какие-нибудь элементы на протяжении всей сессии или между транзакциями, помещайте их в декларативную часть тела пакета. Если вы должны сделать какие-нибудь элементы общими, помещайте их в спецификацию пакета.

Разделение спецификации и тела пакета

Пакет создается из двух частей: спецификации и тела.

Спецификация пакета объявляет все общие конструкторы этого пакета, а тело определяет все

конструкты (как общие, так и личные). Это разделение на две части предоставляет следующие преимущества:

- Определяя спецификацию пакета отдельно от тела пакета, разработчик получает большую гибкость в цикле разработки. После создания спецификации к общим процедурам можно обращаться еще до того, как создано тело пакета.

- Тела процедур, содержащиеся в теле пакета, можно изменять независимо от из публично объявленных спецификаций в спецификации пакета. Пока не изменяется сама спецификация процедуры, объекты, которые обращаются к изменяющимся процедурам, никогда не помечаются как недействительные; иными словами, они никогда не требуют перекомпиляции.

Улучшение производительности

Использование пакетов вместо независимых хранимых процедур дает следующие преимущества:

- Когда вы вызываете пакетированную подпрограмму первый раз, в память загружается весь пакет. Эта загрузка выполняется за одну операцию, в отличие от отдельных загрузок, требуемых для независимых процедур. Поэтому последующие вызовы других подпрограмм этого пакета не требуют операций ввода-вывода.

- Тело пакета можно заменять и перекомпилировать, не затрагивая его спецификацию. Как следствие, объекты, которые обращаются к конструктам пакета (всегда через его спецификацию), никогда не требуют перекомпиляции, пока не будет изменена также спецификация пакета. Благодаря пакетам, количество перекомпиляций может быть минимизировано, что улучшает общую производительность базы данных.

Как ORACLE хранит процедуры и пакеты

Хранимые процедуры, определенные вне контекста пакета, называются независимыми процедурами. Процедуры, определенные внутри пакета (пакетированные), рассматриваются как часть пакета.

Когда вы создаете процедуру или пакет, ORACLE автоматически выполняет следующие шаги:

- а) Компилирует процедуру или пакет.
- б) Сохраняет откомпилированный код в основной памяти.
- в) Сохраняет процедуру или пакет в базе данных.

▼ Подробнее

Компиляция процедур и пакетов

Компиляцию исходного кода осуществляет компилятор PL/SQL. Этот компилятор является частью процессора PL/SQL, содержащегося в ORACLE. Если по время компиляции происходит ошибка, то возвращается сообщение об ошибке.

Сохранение откомпилированного кода в памяти

ORACLE кэширует откомпилированную процедуру или пакет в разделяемом пуле в области SGA. Это позволяет быстро исполнять откомпилированный код и использовать его совместно всеми пользователями. Откомпилированная версия процедуры или пакета остается в разделяемом пуле столько, сколько это позволит алгоритм LRU, даже если первоначальный создатель процедуры закончит свою сессию.

Сохранение процедур и пакетов в базе данных

Во время создания и компиляции процедуры или пакета ORACLE автоматически сохраняет в базе данных следующую информацию: имя объекта ORACLE использует это имя для идентификации процедуры или пакета. Вы задаете это имя в предложениях CREATE PROCEDURE, CREATE FUNCTION, CREATE PACKAGE и CREATE PACKAGE BODY. Исходный код Компилятор PL/SQL разбирает исходный код и и дерево генерирует разобранное представление исходного разбора кода, называемое ДЕРЕВОМ РАЗБОРА. псевдокод Компилятор PL/SQL генерирует ПСЕВДОКОД, или (P-код) P-код, на основе разобранного кода. Процессор PL/SQL исполняет этот код при вызове процедуры или пакета. сообщения ORACLE может генерировать ошибки во время об ошибках компиляции процедуры или пакета. Чтобы избежать излишних перекомпиляций процедуры или пакета, в базе данных сохраняются как дерево разбора, так и псевдокод объекта. Это позволяет процессору PL/SQL при обращении к процедуре или пакету прочитать откомпилированную версию в разделяемый пул в SGA, если его еще нет в SGA. Дерево разбора используется при компиляции кода, вызывающего процедуру. Все части процедур базы данных сохраняются в табличном пространстве SYSTEM (словаре данных) соответствующей базы данных. Администратор базы данных должен планировать размер табличного пространства SYSTEM, имея в виду, что все хранимые процедуры требуют места в этом табличном пространстве.

Как ORACLE исполняет процедуры и пакеты

Когда вызывается независимая или пакетированная процедура, ORACLE выполняет следующие шаги:

- а) Проверяет права пользователя.
- б) Проверяет действительность процедуры.
- в) Исполняет процедуру.

▼ Подробнее

Проверка прав пользователя

ORACLE проверяет, что вызывающий пользователь владеет вызываемой процедурой или имеет привилегию EXECUTE для процедуры или объемлющего ее пакета. Пользователю, вызывающему процедуру, не требуется привилегий доступа к процедурам или объектам, к которым есть обращения в данной процедуре; такие привилегии требуются лишь для владельца процедуры или пакета.

Проверка действительности процедуры

ORACLE проверяет состояние процедуры или пакета в словаре данных на действительность. Объект (процедура или пакет) помечается как НЕДЕЙСТВИТЕЛЬНЫЙ, если после последней компиляции этого объекта произошло одно из следующих событий:

- Хотя бы один из объектов (таблиц, обзоров, других процедур), к которым обращается процедура, был изменен (ALTER) или удален (DROP). (Например, вы добавили в таблицу столбец.)
- Системная привилегия была отобрана у PUBLIC или у владельца процедуры или пакета.
- Привилегия объекта для одного или нескольких объектов, к которым обращается процедура или пакет, была отобрана у PUBLIC или у владельца процедуры или пакета. Процедура ДЕЙСТВИТЕЛЬНА, если она не была отмечена как недействительная по одной из перечисленных выше причин. Если вызывается действительная независимая или пакетированная процедура, то исполняется ее откомпилированный код. Если вызывается недействительная независимая или пакетированная процедура, то она автоматически перекомпилируется перед исполнением.

Исполнение процедуры

Процессор PL/SQL исполняет процедуру или пакет различными методами, в зависимости от ситуации:

- Если процедура действительна и сейчас находится в памяти, то процессор PL/SQL просто исполняет ее псевдокод.
- Если процедура действительна, но сейчас не находится в памяти, то процессор PL/SQL загружает с диска в память ее откомпилированный псевдокод и исполняет его. В случае пакета, загружаются все конструкторы пакета (т.е. все процедуры, переменные и т.п., которые были откомпилированы как единица кода). Процессор PL/SQL обрабатывает процедуру предложение за предложением, исполняя все процедурные предложения самостоятельно и передавая предложения SQL исполнителю предложений SQL.

Пакетированные курсоры

При помещении курсора в пакет вы можете отделить спецификацию курсора от его тела,

используя фразу RETURN, как показывает следующий пример:

```
CREATE PACKAGE audit_actions AS
/* Объявить спецификацию курсоров */
CURSOR c1 RETURN goods%ROWTYPE;
CURSOR c2 RETURN warehouses%ROWTYPE;
CURSOR c3 RETURN storages%ROWTYPE;
...
END audit_actions;
CREATE PACKAGE BODY audit_actions AS
/* Определить тела курсоров */
CURSOR c1 RETURN goods%ROWTYPE SELECT * FROM goods WHERE goods_id > 100;
CURSOR c2 RETURN warehouses%ROWTYPE SELECT * FROM warehouses WHERE ware_id > 10;
CURSOR c3 RETURN storages%ROWTYPE SELECT * FROM storages WHERE storage_id > 1000;
...
END audit_actions;
```

Это позволит вам изменять тело курсора, не затрагивая его спецификацию. Так, в последнем примере можно было бы изменить фразу WHERE :

```
CURSOR c1 RETURN goods%ROWTYPE SELECT * FROM goods WHERE goods_id =115;
```

В спецификации курсора отсутствует предложение SELECT, потому что фразы RETURN достаточно, чтобы определить тип данных результирующего значения. Вы можете использовать в фразе RETURN атрибут %ROWTYPE, чтобы указать тип записи, которая будет представлять строку в таблице базы данных. Вы также можете использовать в фразе RETURN атрибут %TYPE, чтобы указать тип данных переменной, константы или столбца базы данных. Тело курсора должно содержать предложение SELECT, а также точно такую же фразу RETURN, как и в спецификации курсора. Более того, количество и типы данных элементов списка в предложении SELECT должны соответствовать фразе RETURN.

Перекрытие имен

PL/SQL позволяет двум или нескольким пакетированным подпрограммам иметь одно и то же имя. Эта возможность полезна, когда вы хотите, чтобы подпрограмма могла принимать параметры, принадлежащие разным типам данных.

Например, следующий пакет определяет две процедуры с именем date _ storages :

```
PACKAGE storages_date IS
PROCEDURE date_storages (stor_id NUMBER, stor_date CHAR);
PROCEDURE date_storages (stor_id NUMBER, stor_date NUMBER);
END storages_date;
PACKAGE BODY storages_date IS
PROCEDURE date_storages (stor_id NUMBER, stor_date CHAR) IS
```



```
BEGIN
UPDATE storages SET begin_time= TO_DATE(stor_date, 'DD-MON-YYYY')
WHERE srorage_id=stor_id;
END date_storages;
PROCEDURE date_storages (stor_id NUMBER, stor_date NUMBER) IS
BEGIN
UPDATE storages SET begin_time= TO_DATE(stor_date, 'J')
WHERE srorage_id=stor_id;
END date_storages;
END storages_date;
```

Первая процедура принимает дату stor _ date как строку символов, тогда как вторая - как число (юлианский день). Тем не менее, обе процедуры обрабатывают эти данные как следует.

Вызов пакетированных подпрограмм

Пакетированные подпрограммы должны вызываться с помощью квалифицированной ссылки, как показывает следующий пример:

```
storages_date.date_storages (4, '12-MAY-2003');
```

Это указывает компилятору PL / SQL , что date _ storages находится в пакете storages _ date. Вы можете вызывать пакетированные подпрограммы из триггера базы данных, другой хранимой подпрограммы, приложения прекомпилятора ORACLE, приложения OCI, или из инструмента ORACLE, такого как SQL*Plus.

▼ Подробнее

Из другой хранимой подпрограммы

Хранимая подпрограмма может вызывать пакетированную подпрограмму. Например, в независимой подпрограмме может появиться следующий вызов пакетированной процедуры date _ storages :

```
storages_date.date_storages (4, '12-MAY-2003');
```

Из инструмента ORACLE

Вы можете вызывать пакетированные подпрограммы интерактивно из инструментов ORACLE, таких как SQL*Plus, SQL*Forms или SQL*DBA. Например, из SQL*Plus вы могли бы вызвать пакетированную процедуру hire_employee следующим образом:

```
SQL> EXECUTE storages_date.date_storages (4, '12-MAY-2003');
```

Удаленный доступ

Для вызова пакетированных подпрограмм, хранимых в удаленной базе данных ORACLE, используйте следующий синтаксис:

имя_пакета.имя_подпрограммы@связьБД(параметр1, параметр2, ...);

В следующем примере вызывается пакетированная процедура `date _ storages`, определенная в пакете `storages _ date` в базе данных `newyork`: `storages_date.date_storages@newyork(4, '12-MAY-2003');`

Ограничение

Тело пакетированной подпрограммы может содержать любое предложение SQL или PL/SQL. Однако подпрограммы, участвующие в распределенной транзакции, триггерах базы данных и приложениях SQL*Forms, не могут вызывать пакетированных подпрограмм, содержащих предложения COMMIT, ROLLBACK или SAVEPOINT.

Состояния пакетов и зависимости

Спецификация пакета всегда находится в одном из двух состояний: действительна или недействительна.

Спецификация пакета **ДЕЙСТВИТЕЛЬНА**, если ни ее исходный код, ни любой из объектов, к которым она обращается, не был ни удален (DROP), ни заменен (REPLACE), ни изменен (ALTER) с момента последней компиляции этой спецификации.

С другой стороны, спецификация пакета **НЕДЕЙСТВИТЕЛЬНА**, если ее исходный код или любой из объектов, к которым она обращается, был либо удален (DROP), либо заменен (REPLACE), либо изменен (ALTER) с момента последней компиляции этой спецификации.

Когда ORACLE помечает спецификацию пакета как недействительную, он также помечает как недействительные все объекты, обращающиеся к этому пакету. К телу пакета применяются те же правила, с той разницей, что ORACLE может перекомпилировать тело пакета, НЕ помечая его спецификацию как недействительную. Эта возможность позволяет ограничить каскад перекомпиляций, вызываемых зависимостями.

Зависимости

Когда спецификация пакета перекомпилируется, ORACLE помечает зависимые объекты как недействительные. К таким объектам относятся независимые или пакетированные подпрограммы, которые вызывают или обращаются к объектам, объявленным в перекомпилируемой спецификации пакета.

Когда вы вызовете или обратитесь к зависимому, но еще не перекомпилированному, объекту, ORACLE автоматически перекомпилирует его во время выполнения. Когда перекомпилируется тело пакета, ORACLE определяет, действительны ли объекты, от которых зависит тело пакета. К

таким объектам относятся независимые подпрограммы и спецификации пакетов, к которым существуют обращения из процедур или курсоров, определенных в перекомпилируемом теле пакета.

Если среди таких объектов есть недействительные, ORACLE перекомпилирует их, прежде чем перекомпилировать тело пакета. Если все перекомпиляции успешны, тело пакета становится действительным. В противном случае ORACLE возвращает ошибку выполнения, а тело пакета остается недействительным.

Ошибки компиляции сохраняются в словаре данных вместе с пакетом. ORACLE хранит спецификацию и тело пакета отдельно в словаре данных. Другие объекты, которые вызывают глобальные объекты данного пакета или обращаются к ним, зависят только от спецификации пакета. Поэтому вы можете переопределять объекты в теле пакета (что приведет к перекомпиляции тела), не нарушая действительности зависимых объектов.

Триггеры объектов

ORACLE позволяет определять процедуры, которые неявно выполняются, когда для ассоциированной таблицы выдается предложение INSERT, UPDATE или DELETE. Такие процедуры называются *триггерами базы данных*.

Триггеры аналогичны хранимым процедурам. Триггер может состоять из предложений SQL и PL/SQL, исполняемым как единица, и может вызывать другие хранимые процедуры. Однако процедуры и триггеры различаются по способу их вызова. В то время как процедура явно вызывается пользователем или приложением, триггер неявно ВОЗБУЖДАЕТСЯ (исполняется) ORACLE, когда выдается предложение INSERT, UPDATE или DELETE, независимо от того, какой пользователь сейчас подключен или какое приложение используется.

Триггеры создаются с помощью команды CREATE TRIGGER. Эту команду можно использовать в любом интерактивном инструменте (таком как SQL*Plus или SQL*DBA); при использовании в таких инструментах, одиночная наклонная черта ("/"), вводимая как последняя строка, обозначает конец предложения CREATE TRIGGER.

Предложение CREATE собьется, если в блоке PL/SQL будут обнаружены ошибки.

Имена триггеров должны быть уникальными среди всех триггеров в той же схеме. Имена триггеров не обязаны быть уникальными по отношению к другим объектам схемы (таким как таблицы, обзоры, процедуры); например, таблица и триггер могут иметь одно и то же имя (хотя, во избежание путаницы, это не рекомендуется).

Например, рисунок 11 показывает приложение базы данных с некоторыми предложениями SQL, которые неявно возбуждают несколько триггеров, хранящихся в базе данных.

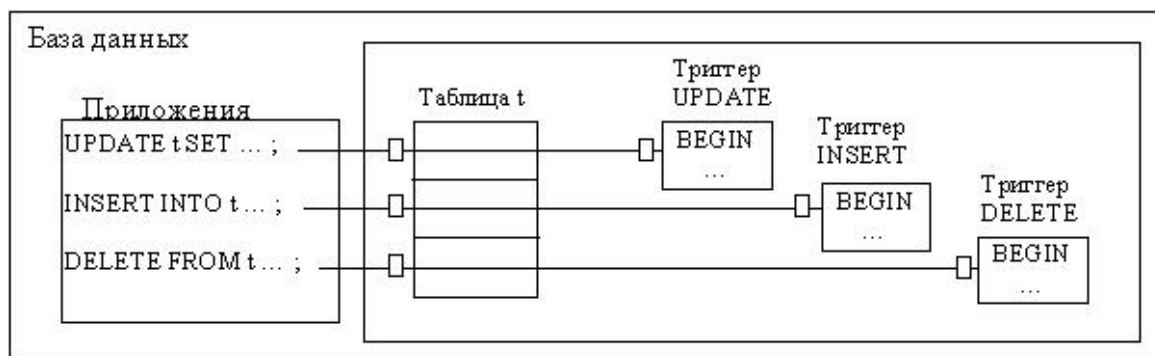


Рисунок 11 - Вызов триггеров из приложения базы данных

Необходимо заметить, что триггеры хранятся в базе данных отдельно от таблиц, с которыми они ассоциированы.

Триггеры можно определять только на таблицах (до появления Oracle 9), но не на обзорах. Однако при выдаче предложения INSERT, UPDATE или DELETE для обзора будут возбуждены триггеры для базовых таблиц этого обзора, если они были определены.

В данном курсе рассмотрим триггеры объектов на примере триггеров таблиц. Триггеры, определенные на представлениях аналогичны триггерам таблиц, но при их создании используется опция **INSTEAD OF**, показывающая, что будут затронуты только данные обзора, например:

```
CREATE OR REPLACE TRIGGER T1 INSTEAD OF INSERT ON VIEW1
```

Синтаксис команды создания триггера

Синтаксис команды создания триггера следующий:

```
CREATE TRIGGER имя триггера
BEFORE/AFTER INSERT/UPDATE/DELETE ON имя таблицы -- предложение триггера
[ FOR EACH ROW ] --предложение триггера
[ WHEN условие ограничения триггера] -- ограничение триггера
BEGIN
-- действие триггера
END;
```

Таким образом, триггер имеет три основные части:

а) событие, или предложение, триггера;

б) ограничение триггера;

в) действие триггера.

Опции BEFORE/AFTER

Либо опция BEFORE, либо опция AFTER должна быть указана в предложении CREATE TRIGGER, чтобы точно специфицировать, когда должно исполняться тело триггера по отношению к исполнению предложения триггера. В предложении CREATE TRIGGER опция BEFORE или AFTER задается непосредственно перед ключевым словом, обозначающим предложение триггера. Например, триггер DUMMY, который был определен в предыдущем примере, является триггером BEFORE.

Триггеры строк AFTER несколько более эффективны, чем триггеры строк BEFORE. При триггерах строк BEFORE, затрагиваемые блоки данных должны быть считаны (логической, а не физической, операцией чтения) один раз для триггера и еще один раз для предложения триггера. Альтернативно, при триггерах строк AFTER, затрагиваемые блоки данных должны быть считаны лишь один раз, сразу для предложения триггера и для самого триггера.

Событие, или предложение, триггера

Предложение триггера специфицирует:

- Тип предложения SQL, которое возбуждает тело триггера. Допустимыми возможностями являются DELETE, INSERT и UPDATE. В спецификацию предложения триггера могут быть включены одна, две или все три этих опции.

- Таблицу, ассоциированную с триггером. Заметьте, что в предложении триггера может быть специфицирована ровно одна таблица (но не обзор).

Событие триггера, или предложение триггера, - это предложение SQL, которое заставляет триггер выполняться. Событием триггера может быть предложение INSERT, UPDATE или DELETE для конкретной таблицы.

Например, для триггера

```
TRIGGER update_stor
```

```
BEFORE UPDATE OF Quantity ON Storages
```

```
FOR EACH ROW
```

```
WHEN (new.quantity<10)
```

предложением триггера является :

... UPDATE OF Quantity ON Storages ...

которое означает, что триггер возбуждается при обновлении столбца Quantity в строке таблицы Storages .

Заметьте , что , когда событием триггера является предложение UPDATE, вы можете включить список столбцов, чтобы указать, обновление каких столбцов возбуждает триггер; поскольку предложения INSERT и DELETE действуют на целые строки данных, то для таких предложений нельзя специфицировать список столбцов.

Если предложение триггера специфицирует UPDATE, то в эту спецификацию может быть включен необязательный список столбцов. Если включается список столбцов, то данный триггер возбуждается по предложению UPDATE лишь тогда, когда это предложение обновляет один из перечисленных столбцов. Если пользователь опускает список столбцов, то триггер возбуждается при обновлении любого столбца ассоциированной таблицы. Список столбцов не может быть специфицирован для предложений триггера INSERT или DELETE.

Событие триггера может специфицировать несколько предложений DML, например:

INSERT OR UPDATE OR DELETE OF Storages ...

что означает, что триггер должен возбуждаться при выдаче предложения INSERT, UPDATE или DELETE для таблицы Storages .

Когда триггер может возбуждаться несколькими предложениями DML, внутри триггера можно использовать предикаты условий для определения типа предложения, возбудившего триггер; поэтому можно создать один триггер, который будет выполнять разные коды в зависимости от того, какой тип предложения возбудил триггер.

Опция FOR EACH ROW

Присутствие или отсутствие опции FOR EACH ROW определяет, является ли этот триггер триггером предложения или триггером строки. Если эта опция включена, она указывает, что тело триггера возбуждается отдельно для каждой строки таблицы, затрагиваемой предложением триггера. Отсутствие опции FOR EACH ROW указывает, что данный триггер должен возбуждаться лишь один раз для предложения триггера.

Ограничение триггера

Ограничение триггера задает булевское (логическое) выражение, которое должно быть истинным (TRUE) для того, чтобы триггер возбуждился. Действие триггера не выполняется, если ограничение триггера дает ложь (FALSE).

Ограничение триггера - это необязательная возможность, которая используется в триггерах,

возбуждаемых по каждой строке. Цель ограничения - наложить условие на выполнение триггера. Ограничение триггера задается с помощью фразы WHEN.

В определение триггера строки может быть включено необязательное ограничение триггера, путем специфицирования булевского выражения SQL в фразе WHEN (фраза WHEN не может быть включена в определение триггера предложения). Выражение в фразе WHEN, если эта фраза присутствует, вычисляется для каждой строки, затрагиваемой триггером. Если результат выражения дает TRUE для строки, то тело триггера исполняется для этой строки. Однако, если это выражение вычисляется для строки как FALSE или NULL, то тело триггера не исполняется для этой строки. Вычисление условия фразы WHEN не влияет на исполнение самого предложения триггера (т.е. предложение триггера НЕ откатывается, если выражение в фразе WHERE вычисляется как FALSE).

Например, триггер update _ stor имеет ограничение триггера. Этот триггер возбуждается предложением UPDATE, действующим на столбец quantity таблицы Storages , но выполнен этот триггер будет лишь при следующем условии:

```
new.quantity<10)
```

Действие триггера

Действие триггера - это процедура (блок PL/SQL), содержащая предложения SQL и PL/SQL, которые будут выполнены, если выдано предложение триггера, а ограничение триггера вычислено как TRUE. Для триггеров строк тело триггера имеет некоторые специальные конструкции, которые могут быть включены в код этого блока PL/SQL:

- а) корреляционные имена ;
- б) опцию REFERENCING ;
- в) условные предикаты INSERTING, DELETING и UPDATING.

Эти предложения исполняются тогда, когда выдано предложение триггера, и ограничение триггера (если оно есть) вычислено как TRUE.

Аналогично хранимым процедурам, действие триггера может содержать предложения как SQL, так и PL/SQL; может определять языковые конструкции PL/SQL (переменные, константы, курсоры, исключения и т.п.); и может вызывать хранимые процедуры. Кроме того, для триггеров строки, предложения в действии триггера имеют доступ как к новым, так и к старым значениям столбцов текущей строки, обрабатываемой триггером (см. ниже).

Тело триггера может содержать любые предложения DML, включая предложения SELECT (только SELECT ... INTO или предложения SELECT в определениях курсоров), INSERT, UPDATE и DELETE; в теле триггера не допускаются предложения DDL. Нельзя также управлять транзакциями в контексте триггера. Поэтому внутри контекста тела триггера не допускаются следующие предложения: ROLLBACK, COMMIT и SAVEPOINT.

Процедура, вызываемая из триггера, также не может выполнять перечисленных выше предложений управления транзакциями, ибо такая процедура исполняется внутри контекста тела триггера. Предложения внутри триггера могут адресоваться к удаленным объектам. Требуется особое внимание при вызове удаленных процедур из локального триггера; если во время выполнения триггера будет обнаружено несовпадение отметок времени, то удаленная процедура не выполняется, а триггер станет недействительным.

Типы триггеров

Рассмотрим разные типы триггеров.

Опция FOR EACH ROW

При определении триггера, можно указать, сколько раз должно исполняться действие триггера: один раз для каждой строки, обрабатываемой предложением триггера (как, например, для предложения UPDATE, обновляющего несколько строк), или один раз на все предложение триггера, независимо от того, сколько строк оно обрабатывает.

Триггер строки возбуждается каждый раз, когда предложение триггера действует на таблицу. Например, если предложение UPDATE обновляет несколько строк таблицы, то триггер строки возбуждается один раз для каждой строки, обновляемой предложением UPDATE. Если предложение триггера не затрагивает ни одной строки, то триггер строки вообще не возбуждается.

Триггеры строки полезны, когда действие триггера зависит от данных, предоставляемых либо предложением триггера, либо строкой, на которую воздействует это предложение.

Триггер предложения возбуждается один раз на предложение триггера, независимо от того, сколько строк таблицы затрагивается этим предложением (даже если не затрагивается ни одной строки). Например, если предложение DELETE удаляет из таблицы несколько строк, то триггер DELETE на уровне предложения возбуждается лишь один раз.

Триггеры предложения полезны, если действие триггера не зависит от данных, предоставляемых предложением триггера, или от данных строк, обрабатываемых этим предложением. Например, если триггер осуществляет комплексную проверку защиты для текущего момента времени или текущего пользователя, или если триггер генерирует одиночную запись аудитинга на основе типа предложения триггера, то используется триггер предложения.

При определении триггера вы можете указать МОМЕНТ ТРИГГЕРА, то есть специфицировать, когда должно выполняться действие триггера по отношению к предложению триггера: перед (BEFORE) или после (AFTER) выполнения предложения триггера. BEFORE и AFTER применимы как к триггерам предложения, так и к триггерам строки.

Триггеры BEFORE выполняют действие триггера перед предложением триггера. Этот тип триггера обычно используется в следующих ситуациях:

- если действие триггера должно определять, можно ли разрешать выполнение предложения триггера. Используя для этой цели триггер BEFORE, вы можете избежать ненужной обработки предложения триггера и его последующего отката в случаях, когда в действии триггера возбуждается исключение;

- для вычисления специфических значений столбцов перед выполнением предложений INSERT или UPDATE.

Триггеры AFTER выполняют действие триггера после предложения триггера. Этот тип триггера используется в следующих ситуациях:

- если вы хотите, чтобы предложение триггера было завершено до выполнения действия триггера;

- если триггер BEFORE тоже присутствует, то триггер AFTER может выполнять другие действия для того же самого предложения триггера.

Используя возможности, указанные выше, вы можете создавать четыре типа триггеров:

- триггер предложения BEFORE - действие триггера выполняется перед выполнением предложения триггера.

- триггер строки BEFORE - действие триггера выполняется перед модификацией каждой строки, затрагиваемой предложением триггера, и перед проверкой соответствующих ограничений целостности, если ограничение триггера вычисляется как TRUE или не включено.

- триггер предложения AFTER - действие триггера выполняется после выполнения предложения триггера и применения любых отложенных ограничений целостности.

- триггер строки AFTER - действие триггера выполняется после модификации каждой строки, затрагиваемой предложением триггера, и после проверки соответствующих ограничений целостности, если ограничение триггера вычисляется как TRUE или не включено. В отличие от триггеров строки BEFORE, триггеры строки AFTER выполняются в состоянии, когда строки таблицы заблокированы.

Доступ к значениям столбцов в триггерах строки

Внутри тела тригга строк, код PL/SQL и предложения SQL имеют доступ как к старым, так и к новым значениям столбцов текущей строки, затрагиваемой предложением триггера. Для каждого столбца модифицируемой таблицы определены два **корреляционных имени**: одно для старого

(old), другое - для нового значения столбца (new). В зависимости от типа предложения триггера, то или иное корреляционное имя может быть лишено смысла:

- Триггер, возбужденный предложением INSERT, имеет осмысленный доступ лишь к новым значениям столбцов. Поскольку строка создается предложением INSERT, старые значения столбцов пусты (NULL).

- Триггер, возбужденный предложением UPDATE, имеет доступ как к старым, так и к новым значениям столбцов для обоих возможных типов триггера (BEFORE или AFTER).

- Триггер, возбужденный предложением DELETE, имеет осмысленный доступ лишь к старым значениям столбцов. Поскольку строка перестает существовать после ее удаления, новые значения столбцов пусты (NULL).

Например, если предложение триггера ассоциировано с таблицей Storages , содержащей столбцы Quantity , Ware _ id и т.д., то вы можете включить в тело триггера предложения, подобные следующим:

```
IF :new.ware_id =15 . . .
```

```
IF :new.quantity < :old.quantity . . .
```

Старые и новые значения доступны как в триггерах BEFORE, так и в триггерах AFTER. Назначать новое значение столбца можно в триггере строк BEFORE, но не в триггере строк AFTER (потому что предложение триггера уже выполнено, прежде чем триггер AFTER получает управление). Если триггер строк BEFORE изменяет значение NEW для столбца, то триггер AFTER, возбужденный тем же самым предложением, видит значение, которое было назначено триггером BEFORE.

Корреляционные имена могут также использоваться в булевском выражении фразы WHEN. Следует заметить, что перед квалификаторами OLD и NEW должно кодироваться двоеточие, когда они используются в теле триггера, но двоеточие не допускается, когда эти квалификаторы используются в фразе WHEN или опции REFERENCING.

Опция REFERENCING может специфицироваться в теле триггера строк для того, чтобы избежать конфликтов между корреляционными именами и именами таблиц, в случае, если таблица имеет имя "OLD" или "NEW". Поскольку такая ситуация редка, эта опция почти никогда не применяется.

Например, предположим, что у вас есть таблица с именем NEW, содержащая столбцы FIELD1 (числовой) и FIELD2 (символьный). Следующее предложение CREATE TRIGGER показывает триггер, ассоциированный с таблицей NEW, который использует опцию REFERENCING, чтобы избежать конфликтов между корреляционными именами и именем таблицы:

```
CREATE TRIGGER dummy  
BEFORE UPDATE ON new REFERENCING new AS newest  
FOR EACH ROW
```

```
BEGIN
:newest.field2 := TO_CHAR (:newest.field1);
END;
```

Заметьте, как квалификатор NEW переименован в NEWEST с помощью опции REFERENCING, а затем использован в теле триггера.

Условные предикаты

Если триггер может быть возбужден более чем одним типом предложения DML (например, "INSERT OR DELETE OR UPDATE OF emp "), то в теле триггера можно использовать условные предикаты INSERTING, DELETING и UPDATING, для того чтобы выполнять различные участки кода в зависимости от типа предложения, возбудившего триггер. Предположим, что предложение триггера определено следующим образом:

```
INSERT OR UPDATE ON emp
```

В коде внутри тела триггера вы можете использовать следующие условия:

```
IF INSERTING THEN
...
END IF;
IF UPDATING THEN
...
END IF ;
```

Первое условие будет вычисляться как TRUE лишь в тех случаях, когда триггер был возбужден предложением INSERT; второе условие будет вычисляться как TRUE лишь в тех случаях, когда триггер был возбужден предложением UPDATE.

Кроме того, в триггере UPDATE условный предикат UPDATING можно специфицировать перед именем столбца, чтобы определять, обновляется ли этот столбец текущим предложением, возбудившим триггер.

Например, предположим, что триггер определен следующим образом:

```
CREATE TRIGGER ...
... UPDATE OF quantity, ware_id ON storages ...
BEGIN
...
IF UPDATING ('quantity') THEN
...
END IF;
END;
```

Код в фразе THEN выполняется лишь в том случае, если предложение UPDATE, возбудившее

триггер, обновляет столбец Quantity .

Например, следующее предложение возбudit показанный выше триггер и заставит условный предикат вычислиться как TRUE:

```
UPDATE storages SET quantity = quantiti -20;
```

Дополнительные аспекты работы с триггерами

Доступ к данным для триггеров

▼ Подробнее

Когда триггер возбуждается, таблицы, к которым обращается его действие, могут в данный момент подвергаться изменениям со стороны предложений SQL в транзакциях других пользователей. Во всех случаях, предложения SQL, выполняемые внутри триггеров, следуют общим правилам, используемым для независимых предложений SQL. В частности, если неподтвержденная транзакция модифицировала те значения, которые необходимы возбужденному триггеру либо для чтения (запрос), либо для записи (обновление), то для предложений SQL в теле возбуждаемого триггера используются следующие правила:

- запросы видят текущий согласованный по чтению снимок всех необходимых данных с учетом изменений, осуществленных текущей транзакцией;
- обновления ожидают освобождения существующих блокировок данных, прежде чем могут продолжить выполнение.

Условия ошибок и исключения в теле триггера

▼ Подробнее

Если во время исполнения тела триггера возникает условие предопределенной или определенной пользователем ошибки (исключение), то все действия как тела триггера, так и предложения, возбудившего триггер, откатываются (если это исключение не обрабатывается специально). Поэтому тело триггера может сознательно воспрепятствовать исполнению предложения триггера путем возбуждения исключения. Обычно в триггерах используются определяемые пользователем исключения, которые реализуют комплексные проверки полномочий или ограничения целостности.

Проектирование триггеров

▼ Подробнее

Используйте следующие рекомендации при проектировании триггеров:

- Используйте триггеры для того, чтобы гарантировать, что при выполнении определенной операции будут выполнены связанные с ней действия.
- Используйте триггеры базы данных только для глобальных, централизованных операций, которые должны быть выполнены для соответствующего предложения (предложения триггера), независимо от того, какой пользователь или приложение базы данных выдает это предложение.
- Не определяйте триггеров, дублирующих возможности, уже встроенные в ORACLE. Например, не определяйте триггеров для ввода в действие правил целостности данных, которые могут быть легко реализованы посредством декларативных ограничений целостности.

Каскады триггеров

▼ Подробнее

Когда триггер возбуждается, предложение SQL внутри его действия потенциально может возбуждать другие триггеры, как показано на рисунке 12. Когда предложение в теле триггера возбуждает другой триггер, это называется каскадом.

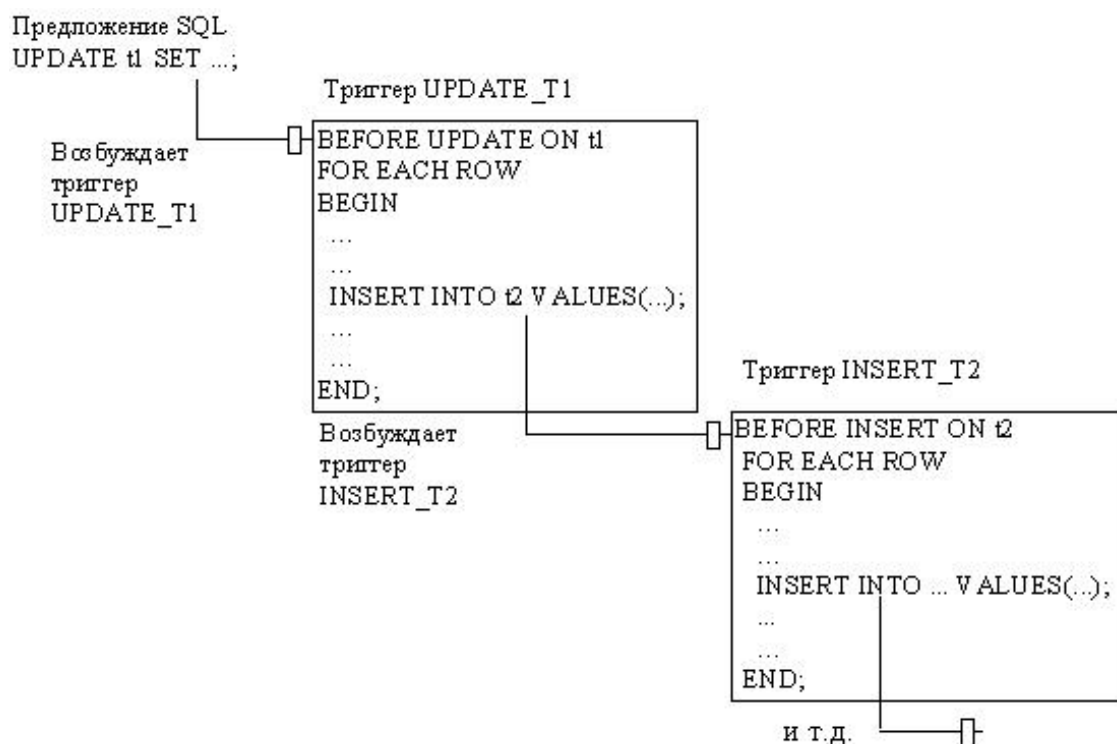


Рисунок 12 - Каскад триггеров

Выполнение триггера

▼ Подробнее

Триггер может находиться в одном из двух различных состояний (режимов):

- включен. ВКЛЮЧЕННЫЙ триггер выполняет свое действие, если выдано предложение триггера, а ограничение триггера (если есть) вычисляется как TRUE;
- выключен. ВЫКЛЮЧЕННЫЙ триггер не выполняет свое действие, даже если выдано предложение триггера, а ограничение триггера (если есть) вычисляется как TRUE.

Для включенных триггеров ORACLE автоматически выполняет следующие действия:

- выполняет триггеры в спланированной последовательности, если одно предложение SQL возбуждает более одного триггера;
- выполняет проверку ограничений целостности в момент времени, предписанный для каждого типа триггера, и обеспечивает, чтобы триггеры не нарушали ограничений целостности;
- предоставляет согласованные по чтению представления данных для запросов и ограничений;
- управляет зависимостями между триггерами и объектами, к которым есть обращения в действии триггера;
- использует двухфазное подтверждение, если триггер обновляет удаленные таблицы.

Модель исполнения для триггеров и проверок ограничений целостности

▼ Подробнее

Одно предложение SQL может потенциально возбудить до четырех триггеров: триггер строки BEFORE, триггер предложения BEFORE, триггер строки AFTER и триггер предложения AFTER. Как предложение триггера, так и любое предложение внутри действия триггера может вызвать проверку одного или нескольких ограничений целостности. Кроме того, триггеры могут содержать предложения, возбуждающие другие триггеры (каскад триггеров). ORACLE использует следующую модель, чтобы поддерживать должный порядок возбуждения множественных триггеров и проверки ограничений.

- а) выполнить триггер предложения BEFORE;
- б) цикл для каждой строки, на которую действует предложение SQL;
 - 1) выполнить триггер строки BEFORE;
 - 2) заблокировать и изменить строку, выполнить проверку ограничений целостности. (Эта блокировка не будет освобождена до конца транзакции);
 - 3) Выполнить триггер строки AFTER;

- в) выполнить проверку отложенных ограничений целостности;
- г) выполнить триггер предложения AFTER.

Определение этой модели рекурсивно. Например, некоторое предложение SQL может возбудить триггер строки BEFORE и проверку ограничения целостности. Этот триггер строки BEFORE, в свою очередь, может выполнять обновление, которое вынудит проверку ограничения целостности и возбудит триггер предложения AFTER. Этот триггер предложения AFTER вызовет проверку ограничения целостности. В этом случае показанная модель будет выполнять шаги рекурсивно:

- а) Выдается первоначальное предложение SQL;
- б) Возбуждается триггер строки BEFORE;
 - 1) предложение UPDATE в триггере строки BEFORE возбуждает триггер предложения AFTER;
 - 2) выполняются предложения в действии триггера предложения AFTER;
 - 3) проверяются ограничения целостности по тем таблицам, которые изменены триггером предложения AFTER;
- в) Выполняются предложения в триггере строки BEFORE;
- г) Проверяются ограничения целостности по тем таблицам, которые изменены триггером строки BEFORE;
- д) Выполняется первоначальное предложение SQL;
- е) Проверяется ограничение целостности для предложения SQL.

Важным свойством описанной модели исполнения является то, что все действия и проверки, осуществляемые как результат предложения SQL, должны быть выполнены успешно. Если внутри триггера возбуждается исключение, и это исключение явно не перехватывается, то все действия, выполненные как результат первоначального предложения SQL, включая те действия, которые были выполнены возбужденными триггерами, подвергаются откату. Поэтому ограничения целостности не могут быть нарушены триггерами. Модель исполнения принимает во внимание ограничения целостности и не позволяет выполняться триггерам, которые нарушили бы декларативные ограничения целостности. Например, в описанном выше сценарии, предположим, что шаги с первого по восьмой выполнены успешно; однако в шаге (е) обнаружилось нарушение ограничения целостности. Как следствие этого нарушения, все изменения, которые были сделаны предложением SQL (шаг д), триггером строки BEFORE (шаг в) и триггером предложения AFTER (шаг 2), подвергаются откату.

Ограничения целостности и триггеры

▼ Подробнее

Несмотря на то, что большинство аспектов целостности данных могут быть определены и задействованы через декларативные ограничения целостности, некоторые сложные организационные правила, не определяемые через декларативные ограничения целостности, могут быть реализованы посредством триггеров. Например, триггеры могут применяться в следующих случаях:

- чтобы задействовать ссылочную целостность, когда нужное правило ссылочной целостности не может быть введено в действие через ограничения целостности: обновление CASCADE, обновление и удаление SET NULL, обновление и удаление SET DEFAULT 4

- чтобы задействовать ссылочную целостность, когда зависимая и родительская таблицы находятся на разных узлах распределенной базы данных ;

- чтобы задействовать комплексные организационные правила, которые не могут быть определены через выражения, допустимые в ограничениях CHECK .

Привилегии, требуемые для создания триггеров**▼ Подробнее**

Чтобы создать триггер в своей схеме, вы должны иметь системную привилегию CREATE TRIGGER, а также одно из:

- владеть таблицей, специфицированной в предложении триггера, или
- иметь привилегию ALTER для таблицы, специфицированной в предложении триггера, или
- иметь системную привилегию ALTER ANY TABLE .

Чтобы создать триггер в схеме другого пользователя, вы должны иметь системную привилегию CREATE ANY TRIGGER. Эта привилегия позволяет создать триггер в любой схеме и ассоциировать его с таблицей любого пользователя.

Привилегии для объектов схем, адресуемых в теле триггера

Как и для хранимых процедур, владелец триггера должен иметь объектные привилегии для объектов, адресуемых в теле триггера, причем эти привилегии должны быть получены им явно (не через роли). Предложения в теле триггера оперируют под доменом защиты владельца триггера, а не того пользователя, который выдает предложение, возбуждающее триггер.

Изменение триггеров**▼ Подробнее**

Нельзя явно изменить триггер; его необходимо заменить новым определением триггера. Если

вы используете текстовый редактор для создания ваших триггеров, вы можете просто отредактировать соответствующий текстовый файл и выполнить предложение CREATE TRIGGER, чтобы определить новую версию кода.

Заменяя триггер, вы должны включить в предложение CREATE TRIGGER опцию OR REPLACE. Опция OR REPLACE позволяет заменить существующий триггер новой версией, не затрагивая никаких грантов, которые были выданы для первоначальной версии этого триггера.

Альтернативно, триггер можно удалить и создать заново. Однако все гранты, которые были выданы для удаляемого триггера, также удаляются, и должны быть выданы заново после создания новой версии триггера.

Включение и выключение триггеров

▼ Подробнее

Триггер может находиться в одном из двух различных режимов:

- включен: Включенный триггер выполняет свое тело, если выдано предложение триггера, и ограничение триггера (если есть) вычисляется как TRUE.

- выключен: Выключенный триггер не выполняет свое тело, даже если выдано предложение триггера, и ограничение триггера (если есть) вычисляется как TRUE.

Вы можете временно выключить триггер, если имеет место одно из следующих условий:

- Объект, к которому обращается триггер, недоступен.
- Вы должны выполнить массовую загрузку данных, и хотите осуществить ее быстро, не возбуждая триггеров.
- Вы загружаете данные в таблицу, к которой применяется триггер.

По умолчанию, триггер включается в момент его создания. Чтобы отключить триггер, используйте команду ALTER TRIGGER с опцией DISABLE. Например, следующее предложение отключает триггер REORDER по таблице INVENTORY:

```
ALTER TRIGGER reorder DISABLE;
```

Вы можете одновременно отключить все триггеры, ассоциированные с таблицей, с помощью команды ALTER TABLE с опциями DISABLE и ALL TRIGGERS. Например, следующее предложение отключает все триггеры, определенные для таблицы INVENTORY:

```
ALTER TABLE inventory DISABLE ALL TRIGGERS;
```

По умолчанию, триггер автоматически включается в момент его создания; однако позже он может быть выключен. Закончив задачу, для которой потребовалось выключать триггер, вы

можете снова включить его.

Чтобы включить триггер, используйте команду ALTER TRIGGER с опцией ENABLE. Например, следующее предложение включает триггер REORDER по таблице INVENTORY :

```
ALTER TRIGGER reorder ENABLE;
```

Вы можете одновременно включить все триггеры, ассоциированные с таблицей, с помощью команды ALTER TABLE с опциями ENABLE и ALL TRIGGERS. Например, следующее предложение включает все триггеры, определенные для таблицы INVENTORY:

```
ALTER TABLE inventory ENABLE ALL TRIGGERS;
```

Привилегии, требуемые для включения и выключения триггеров

Для включения и выключения триггеров с помощью команды ALTER TABLE, вы должны либо владеть таблицей, либо иметь объектную привилегию ALTER TABLE для таблицы или системную привилегию ALTER ANY TABLE. Для включения или выключения индивидуального триггера с помощью команды ALTER TRIGGER, вы должны либо владеть триггером, либо иметь системную привилегию ALTER ANY TRIGGER.

Вывод информации о триггерах

▼ Подробнее

Следующие обзоры словаря данных раскрывают информацию о триггерах:

- USER_TRIGGERS ;
- ALL_TRIGGERS ;
- DBA_TRIGGERS .

Например, следующие два запроса возвращают информацию о триггере REORDER:

```
SELECT type, triggering_statement, table_name FROM user_triggers WHERE name = 'REORDER';
```

```
SELECT trigger_body FROM user_triggers WHERE name = 'REORDER';
```

Удаление триггеров

▼ Подробнее

Для удаления триггера из базы данных используется команда DROP TRIGGER.

Например, чтобы удалить триггер с именем REORDER, требуется ввести следующее предложение:

```
DROP TRIGGER Trigger _ name ;
```

Чтобы удалить триггер, пользователь должен иметь его в своей схеме, либо иметь системную привилегию DROP ANY TRIGGER.

Сравнение триггеров и декларативных ограничений целостности

▼ Подробнее

Как триггеры, так и декларативные ограничения целостности можно использовать для ограничений на входные данные. Однако триггеры и декларативные ограничения целостности имеют существенные различия.

Декларативное ограничение целостности - это утверждение о базе данных, которое всегда истинно. Ограничение применяется к существующим данным в таблице и к любому предложению, которое манипулирует этой таблицей.

Триггеры налагают ограничения на то, что могут делать транзакции. Триггер не применяется к данным, которые были загружены до того, как триггер был определен; поэтому триггер не гарантирует, что все данные в таблице удовлетворяют правилам, установленным ассоциированным триггером.

Триггер вводит в действие переходные ограничения; иными словами, триггер вводит в действие ограничение, действующее на момент изменения данных. Поэтому такое ограничение, как "гарантировать, что дата доставки отстоит по меньшей мере на семь дней от сегодняшней даты", должно реализовываться через триггер, а не через декларативное ограничение целостности.

При вычислении триггеров, которые содержат функции SQL, имеющие в качестве аргументов параметры средства поддержки национальных языков NLS (например, TO_CHAR, TO_DATE, TO_NUMBER), умалчиваемые значения таких параметров берутся из параметров NLS, действующих для сессии в текущий момент. Вы можете перекрывать умалчиваемые значения для параметров NLS, явно специфицируя параметры NLS в таких функциях при написании триггера.

С помощью триггеров могут быть введены в действие многие варианты ссылочной целостности. Однако применяйте триггеры лишь тогда, когда вы хотите реализовать ссылочные действия UPDATE и DELETE SET NULL, UPDATE и DELETE SET DEFAULT, или если вы хотите задействовать ссылочную целостность, когда зависимая и родительская таблицы находятся на разных узлах распределенной базы данных.

Применяя триггеры для поддержания ссылочной целостности, объявите ограничение PRIMARY KEY или UNIQUE по родительской таблице. Если ссылочная целостность должна поддерживаться между родительской и порожденной таблицами в одной и той же базе данных, то вы можете также объявить внешний ключ в порожденной таблице, но отключить его; это предотвратит возможность удаления соответствующего ограничения PRIMARY KEY (если только

ограничение PRIMARY KEY не удаляется явно опцией CASCADE).

Чтобы поддерживать ссылочное ограничение с помощью триггеров:

Для порожденной таблицы должен быть определен триггер, гарантирующий, что значения, вставляемые или обновляемые во внешнем ключе, соответствуют значениям родительского ключа. Один или несколько триггеров должны быть определены для родительской таблицы. Эти триггеры гарантируют выполнение нужного ссылочного действия (RESTRICT, CASCADE или SET NULL) при обновлении или удалении значения родительского ключа. При вставках в родительскую таблицу не требуется никаких ссылочных действий, ибо еще не существует зависимых внешних ключей.

Мутирующие и ограничивающие таблицы

▼ Подробнее

Мутирующая таблица - это таблица, модифицируемая в данный момент предложением UPDATE, DELETE или INSERT, или таблица, которая может потребовать обновления в результате действия декларативного ссылочного ограничения целостности DELETE CASCADE.

Ограничивающая таблица - это таблица, которую предложение, возбуждающее триггер или ограничение, должно читать, - либо явно (для предложения SQL), либо неявно (для декларативного ограничения ссылочной целостности).

Мутирующая или ограничивающая таблица является таковой лишь для той сессии, которая выдает предложение, возбуждающее триггер или ограничение.

Для триггеров предложений таблица не рассматривается как мутирующая или как ограничивающая; однако для триггеров строк имеют место два важных ограничения, касающихся мутирующих и ограничивающих таблиц:

- Предложения в триггере строк не могут читать или модифицировать мутирующую таблицу предложения триггера. Это ограничение не позволяет триггеру строк иметь дело с несогласованным множеством данных.

- Предложения в триггере строк не могут изменять столбцов первичного, внешнего или уникального ключа ограничивающей таблицы предложения триггера. Это ограничение введено с тем, чтобы успех предложения внутри триггера не зависел от порядка обработки строк или от индекса.

Для этого правила существует одно исключение: триггер BEFORE ROW, возбужденный предложением INSERT, вставляющим одиночную строку в таблицу с внешним ключом, может модифицировать любые столбцы первичной таблицы, пока эта модификация не нарушает никаких ограничений целостности.

Если вам необходимо обновить мутирующую или ограничивающую таблицу, вам следует

использовать временную таблицу, таблицу PL/SQL или пакетированную переменную, чтобы обойти эти ограничения.

Аудитинг с помощью триггеров

▼ Подробнее

Типичное применение триггеров - дополнять встроенные средства аудиторинга ORACLE. Хотя можно писать триггеры, которые будут записывать информацию, аналогичную той, что регистрируется командой AUDIT, триггеры следует применять лишь в том случае, если вам требуется более детальная аудиторская информация. Например с помощью триггеров можно реализовать отслеживание на уровне значений столбцов в строках таблиц.

Иногда команда ORACLE AUDIT рассматривается как средство аудита ЗАЩИТЫ, тогда как триггеры могут обеспечить средства ФИНАНСОВОГО аудита.

Принимая решение о создании триггера для отслеживания операций в базе данных, рассматривайте те возможности, которые могут предоставить средства аудиторинга ORACLE, и сравнивайте их с аудитингом, который может быть реализован триггерами.

Применяя триггеры для изолированного аудиторинга, обычно используют триггеры AFTER. За счет использования триггеров AFTER, вы регистрируете аудиторскую информацию после того, как предложение триггера подверглось воздействию всех возможных ограничений целостности, и избегаете выполнения излишней аудиторской работы в тех случаях, когда предложения подвергаются откату из-за нарушения ограничений целостности.

Использовать ли триггеры строк или триггеры предложений, зависит от того, какую информацию вы отслеживаете. Например, триггеры строк обеспечивают отслеживание значений по строкам таблицы. Триггеры могут также позволить пользователю предоставлять "код причины", по которой выдается отслеживаемое предложение, что может быть полезным при аудитинге как на уровне строк, так и на уровне предложений.

Триггеры и вычисляемые значения столбцов

▼ Подробнее

Триггеры могут автоматически вычислять значения столбцов, базируясь на значениях, которые поставляются предложением INSERT или UPDATE. Такой тип триггера полезен для принудительной установки значений определенных столбцов, зависящих от значений других столбцов в той же самой строке. Для такого типа операций необходимы триггеры строк BEFORE, ибо:

- Зависимые значения должны быть вычислены перед тем, как произойдет вставка или обновление, так, чтобы предложение триггера могло использовать вычисленные значения.

- Триггер должен возбуждаться для каждой строки, которую затрагивает возбуждающее

триггер предложение INSERT или UPDATE.

Для закрепления на практике изученного теоретического материала рекомендуется выполнить практическое занятие 3 и лабораторную работу 3.

Практика

Практика 2. Использование подпрограмм в языках программирования 4-го поколения

Работа со встроенными функциями

Данные поля name (таблицы Goods) перевести в верхний регистр.



```
SELECT UPPER(NAME) FROM GOODS;
```

Для произвольной строки вывести подстроку прописными символами. Подстрока длиной LEN и начинается с позиции POS .



```
CREATE OR REPLACE FUNCTION F2  
(STR IN VARCHAR2, POS NUMBER, LEN NUMBER) RETURN VARCHAR2 AS  
STR_2 VARCHAR2 (20);  
BEGIN  
STR_2:=UPPER( SUBSTR (STR, POS, LEN));  
RETURN STR_2 ;  
END ;
```

Вызов функции F2 может быть такой:

```
SQL> SELECT F2('qwertyu', 2, 4);
```

Рассмотрим пример вызова в SQL*Plus встроенной функции преобразования типов данных (TO _ CHAR)



```
SQL> SELECT TO_CHAR(SYSDATE, YEAR MONTH DAY DD') FROM DUAL;
```

Функция первый параметр получает с помощью другой встроенной функции SYSDATE , а затем полученное значение приводит к формату YEAR MONTH DAY DD '. Таким образом, показанные вызов функции позволяет вернуть текущую дату в формате: год прописью, название месяца, название дня недели и число.

Создание пользовательских функций

Написать функцию, которая возвращает значение определенной статистики по некоторому столбцу некоторой таблицы. Например, для таблицы Goods подсчитать количество строк со значением поля PRICE меньше среднего.



```
CREATE OR REPLACE FUNCTION LESS_THAN_AVG RETURN INTEGER AS  
ROW_COUNT INTEGER;  
BEGIN  
SELECT COUNT(*) INTO ROW_COUNT FROM GOODS  
WHERE PRICE <  
(SELECT AVG(PRICE) FROM GOODS);  
RETURN ROW_COUNT;  
END LESS_THAN_AVG;
```

Определить функцию, которая получает в качестве параметра цену и возвращает количество записей в таблице Goods со значением поля PRICE меньше заданного.



```
CREATE OR REPLACE FUNCTION count_goods (new_price IN Goods.price%TYPE)  
RETURN INTEGER  
IS temp_count Goods.Quantity%TYPE;  
BEGIN  
SELECT COUNT(*) INTO temp_count FROM Goods  
WHERE price < new_price;  
RETURN(temp_count);  
END count_goods;
```

Создание и работа с сохраняемыми процедурами

Написать процедуру, которая добавляла бы новые отделы в таблицу Departments.



Для этого понадобится последовательность, с помощью которой мы можем задавать первичный ключ таблицы (последовательность должна начинаться с номера, следующего за

последним номером, который вы использовали для добавления отделов в таблицу).

```
CREATE SEQUENCE dept_seq START WITH 4;
```

```
CREATE PROCEDURE insert_dept (new_name VARCHAR2, new_info VARCHAR2 DEFAULT New  
department ) AS
```

```
BEGIN
```

```
INSERT INTO Departments VALUES (dept_seq.NEXTVAL, new_name, new_info);
```

```
END insert_dept;
```

Вызов процедуры производится с помощью команды EXEC:

```
EXEC insert_dept ('SWEETS');
```

Написать процедуру, которая для товара с заданным номером изменяет цену на заданную. При вызове этой процедуры использовать позиционную и именную нотации



```
CREATE PROCEDURE change _ price ( good _ id 1 NUMBER , price 1 NUMBER ) IS  
BEGIN  
UPDATE goods SET price = price1 WHERE good_id = good_id1;  
END change _ price ;
```

При вызове подпрограммы можно записывать фактические параметры, используя позиционную или именную нотацию. Иными словами, можно указывать соответствие между фактическими и формальными параметрами через позиции этих параметров или через их имена.

Можно вызвать процедуру change _ price четырьмя логически эквивалентными способами:

```
DECLARE  
change_goods NUMBER(4):=2;  
new_price NUMBER(8,2):=10;  
BEGIN  
change_price(change_goods, new_price); -- позиционная нотация  
change_price(change_goods=> goods_id1, new_price=> price1); -- именная нотация  
change_price(new_price=> price1, change_goods=> goods_id1); -- именная нотация  
change_price(change_goods, new_price=> price1); -- смешанная нотация  
END;
```

Написать процедуру покупки товара - занесение данных в таблицу Sales и изменение количества товаров, имеющихся в наличии. В случае недостачи товара - покупку не фиксировать и выводить сообщение не «хватает X единиц товара», где X - недостающее количество. Для этого понадобится последовательность, с помощью которой задавать первичный ключ таблицы Sales



```
CREATE SEQUENCE Sales_seq START WITH 4;
```

```
CREATE OR REPLACE PROCEDURE ins_sale (new_check NUMBER, new_good_id NUMBER,  
new_date DATE DEFAULT SYSDATE, need_quantity INTEGER DEFAULT 1) AS
```

```
quantity1 INTEGER;
```

```
BEGIN
```

```
SELECT quantity INTO quantity1 FROM Goods WHERE good_id=new_good_id;
```

```
IF quantity1 < need_quantity THEN
```

```
DBMS_OUTPUT.put_line ( 'Не хватает || (need_quantity-quantity1) || единиц товара');
```

```
/* Для вывода на экран используем процедуру put_line пакета вывода DBMS_OUTPUT */
```

```
ELSIF quantity1 = need_quantity THEN
```

```
DELETE FROM Goods WHERE good_id=new_good_id;
```

```
INSERT INTO Sales VALUES (Sales_seq.nextval, new_check, new_good_id, new_date,  
need_quantity);
```

```
COMMIT;
```

```
ELSE
```

```
UPDATE Goods SET quantity = quantity - need_quantity WHERE good_id=new_good_id;
```

```
INSERT INTO Sales VALUES (Sales_seq.nextval, new_check, new_good_id, new_date,  
need_quantity);
```

```
COMMIT;
```

```
END IF;
```

```
END ins_sale
```

Для вывода информации с использованием стандартных функций необходимо включить режим вывода:

```
SET SERVEROUT ON;
```

Следует обратить внимание на то, что при удалении информации о товаре, количество которого равно заданному (тому, что желает приобрести покупатель) может возникнуть ошибка выполнения программы, связанная с наличием порожденных записей в таблице Sales (предварительные операции покупки этого же товара). Необходимо переделать процедуру с учетом такой ситуации.

Использование явных курсоров

Необходимо сформировать таблицу-описание товаров, хранящихся во всех отделах, в виде показанном в таблице 7.



Таблица 7 - Вид результирующей таблицы

Dept_id	Name	List_of_goods: goods/quantity/price;
1	VEGETABLES	APPLE/7.6/12; PINEAPPLE/14.99/9;
2

Для ведения данной таблицы понадобится последовательность:

```
CREATE SEQUENCE list_seq;
```

Команда создания таблицы-описания приведена ниже:

```
CREATE TABLE list_dept (  
Dept_id NUMBER(3) NOT NULL PRIMARY KEY,  
Name VARCHAR2(20),  
list_of_goods VARCHAR2(2000));
```

Приведем пример реализации этой процедуры с помощью явного описания курсоров и обращения к атрибутам явных курсоров:

```
CREATE OR REPLACE PROCEDURE list_goods AS  
  
-- курсор для получения всего перечня отделов  
  
CURSOR list_dept IS  
  
SELECT dept_id, name FROM Departments;
```

/* курсор для получения информации о товарах, находящихся в заданном отделе */

CURSOR I_goods (dept_n1 NUMBER) IS

SELECT Name, Price, Quantity FROM Goods

WHERE Dept_id=dept_n1;

Dept_id1 Departments.Dept_id%Type;

Name1 Departments.Name%Type;

Name_G Goods.Name%Type;

Quantity1 Goods.Quantity%Type;

Price1 Goods.Price%Type;

list_goods VARCHAR2(200) := '';

BEGIN

DELETE FROM list_dept;

IF List_dept%isopen THEN Close List_dept;

End if;

OPEN list_dept;

LOOP

FETCH list_dept INTO Dept_id1, Name1;

EXIT WHEN list_dept%NOTFOUND;

OPEN I_goods(Dept_id1);

list_goods:= '';

LOOP

FETCH I_goods INTO Name_G, Price1, Quantity1;

EXIT WHEN I_goods%NOTFOUND;

list_goods:= list_goods|| Name_G||' '|| Price1 || ' '|| Quantity1 || ' ';

END LOOP;

```
CLOSE l_goods;  
  
INSERT INTO list_dept VALUES (Dept_id1, Name1, list_goods);  
  
END LOOP;  
  
CLOSE list_Dept;  
  
COMMIT;  
  
END;
```

Вызов и результат выполнения процедуры приведены на рис. 13

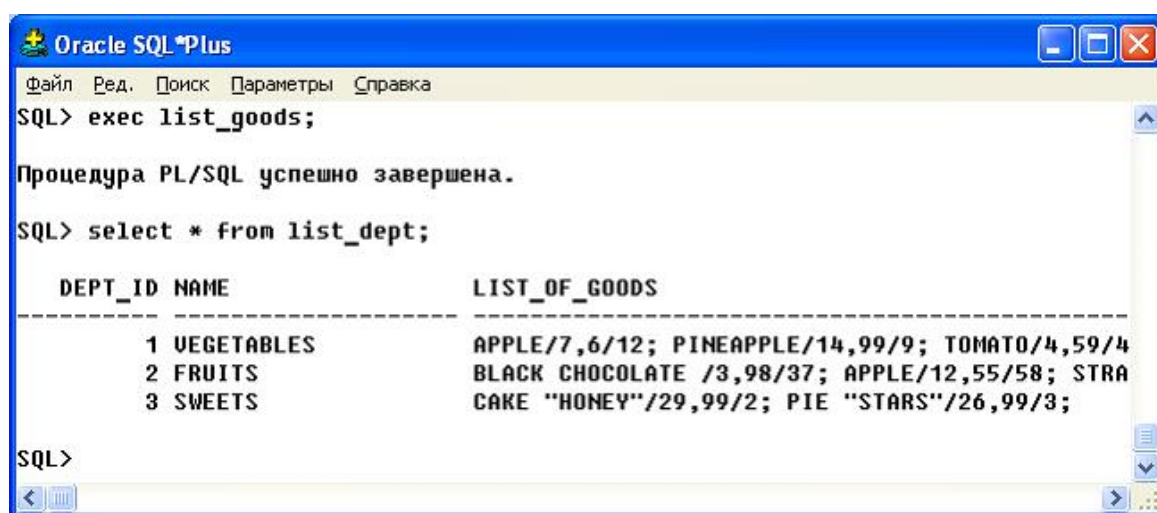


Рисунок 13 - Вызов и результат работы процедуры

Самостоятельно разработайте процедуру заполнения аналогичной таблицы с использованием курсорных циклов.



Обработка исключений

Разработайте процедуру снижения цены на заданные товары на заданную сумму. Следует предусмотреть такие исключительные ситуации:

- если количество товара не превышает 10, а новая цена составляет мене чем 50% от старой цены, то ошибка «Большая скидка»;
- если количество товара более 100, а новая стоимость составляет более чем 90% от старой цены, то ошибка «Маленькая скидка»;
- если не найдено такого товара, необходимо вывести предупреждение: «Такого товара не существует».



```
CREATE OR REPLACE PROCEDURE Discount (G_name VARCHAR2, Price1 NUMBER) AS

Good_Id1 Goods.Good_id%type;

Price_g Goods.Price%type;

Quantity_g Goods.Quantity%type;

Low_disc EXCEPTION;

Big_disc EXCEPTION;

BEGIN

SELECT Good_id, Price, Quantity INTO Good_id1, Price_g, Quantity_g

FROM Goods

WHERE Upper(Name) like '%'||Upper(G_name)||'%' AND ROWNUM=1;

IF ((Price_G*0.5)>(price_g-price1) and Quantity_g<10) THEN RAISE Big_Disc;

ELSIF (Price_G*0.9<(price_g-price1) and Quantity_g>100) THEN RAISE Low_Disc;

END IF;

UPDATE Goods SET Price=Price-Price1 WHERE Good_id=Good_id1;

Commit;

EXCEPTION

WHEN Low_disc THEN

    Raise_application_error (-20002, 'Low discount');

WHEN Big_disc THEN

    Raise_application_error (-20003, 'Big discount');

WHEN NO_DATA_FOUND THEN

    Raise_application_error (-20001, 'No such good');

END;

/
```

Задачи для самостоятельного выполнения

Разработайте функцию обработки данных из таблиц. Примеры заданий:

- разработайте функцию, которая по названию отдела (входной параметр) возвращает суммарную стоимость товаров этого отдела;
- разработайте функцию, которая по названию отдела (входной параметр) возвращает наименование товара, количество которого является максимальной;
- разработайте функцию, которая по названию изготовителя (входной параметр) возвращает наименование его дешевого товара (любого).



Разработайте процедуру, которая выполняет действия модифицирования данных таблиц, созданных в предыдущей лабораторной работе. Примеры заданий:

- для товаров из заданного отдела, которые дороже заданной цены, в поле Description написать «Дорогой товар», а для остальных его товаров- «Дешевый товар»;
- провести снижение цены товаров, которые находятся в заданном отделе на заданное число процентов;
- для заданного отдела в поле Info записать текущую дату.



Разработайте функцию формирования выходной строки через запятую. Примеры заданий:

- входной параметр - название отдела, результат - список наименований товаров, которые находятся в заданном отделе;
- входные параметры - диапазон цен (m i n), результат - список товаров с ценой в заданном диапазоне;
- список номеров чеков, по которым было приобретено товаров на сумму меньше заданной как входной параметр.



Разработайте процедуру с использованием курсоров . Примеры заданий:

- процедура закупки - заполнить таблицу закупки (поля: производитель, название товара, цена, размер закупки) такими данными: если price>1000, то нужно 10 единиц товара, если 500<price<1000, то нужно 20 единиц, если price<500, то - 30 единиц;
- выбрать и записать в таблицу по два самых дешевых товара отдела (поля: производитель, товар, цена);
- процедура разбиения товара на ценовые группы (входной параметр - количество групп). Сформировать таблицу с такими полями: номер группы, минимальная цена в группе, максимальная цена в группе, список товаров группы через запятую.



Далее следует выполнить лабораторную работу 2

Лабораторная работа 2 Подпрограммы как средство создания

серверных частей информационных систем на примере СУБД Oracle

Цель работы

Ознакомление с основными возможностями языка PL/SQL. Закрепление теоретических и приобретение практических навыков создания хранимых пользовательских функций и процедур, работы с курсорами, исключениями, пакетами.

Методические указания к выполнению работы

Для выполнения лабораторной работы необходимо повторить теоретические материалы изложенные в лекционном и практическом материале этого раздела.

Следует рассмотреть вопросы, касающиеся создания и выполнения хранимых процедур и функций, использования курсоров (необходимо разобраться с особенностями двух видов курсоров: неявных и явных). Также нужно разобраться с различиями внутренних (предопределенных) исключений и пользовательских исключений. Изучить способы возбуждения пользовательских исключений. Кроме того, следует рассмотреть вопросы создания спецификации пакета и тела пакета, особенности пакетирования различных объектов языка PL / SQL : процедур и функций, курсоров, типов данных.

Ход работы

Написать спецификацию пакета для своей серверной части информационной системы (CREATE package...) с подробными комментариями о назначении функций и процедур пакета.

Пакет должен содержать 8-10 процедур и функций, среди которых должны быть:

- Процедуры на реальное добавление, редактирование, удаление информации в БД (с использованием последовательностей);
- Процедуры или функции получения статистики по БД;
- Хотя бы 2 процедуры или функции с курсорами;
- Обработка хотя бы 3 исключительных ситуаций (предопределенных и пользовательских).

Написать тело пакета.

Написать команды вызова процедур и функций из пакета.



Вопросы для самоконтроля:

- а) Приведите синтаксис задания функции.
- б) Приведите синтаксис задания процедуры.
- в) С какими параметрами могут быть использованы функции?
- д) Перечислите преимущества хранимых процедур и функций в PL/SQL;
- е) Проведите сравнительный анализ явных и неявных курсоров, приведите примеры их использования;
- ж) Расскажите об особенностях использования различных управляющих предложений в PL/SQL;
- з) Приведите синтаксис объявления курсоров.
- и) Перечислите преимущества использования пакетов и пакетированных процедур и функций;
- к) Расскажите об особенностях вызова пакетированных процедур и функций;
- л) Приведите синтаксис создания спецификации и тела пакета.

Указания к оформлению отчета

Отчеты по лабораторной работе можно оформлять и представлять преподавателю в электронном виде или в твердой копии.

Отчет должен содержать:

- название работы;
- цель работы;
- спецификацию и тело пакета с комментариями к процедурам и функциям;
- экранные формы утилиты SQL * Plus , демонстрирующие вызов процедур и функций пакета и результаты их выполнения;
- выводы по работе.

Практика 3. Создание и работа с триггерами объектов и событий

Во многих случаях триггеры дополняют стандартные возможности ORACLE, способствуя созданию гибко настраиваемой системы управления базой данных.

Например, триггер может позволять операции DML по таблице лишь тогда, когда они

предпринимаются в нормальные рабочие часы. Итак, стандартные средства защиты ORACLE, роли и привилегии, контролируют, какие пользователи могут выдавать предложения DML для таблицы; в дополнение к этому, триггер еще более ограничивает операции DML, следя за тем, чтобы они выполнялись в определенное время. Это лишь один из способов, как вы можете использовать триггеры для настройки управления информацией в базе данных ORACLE.

Помимо этого, триггеры обычно используются для:

- автоматической генерации значений вычисляемых столбцов;
- предотвращения незаконных транзакций;
- ввода в действие комплексных правил защиты;
- обеспечения ссылочной целостности между узлами в распределенной базе данных;
- реализации сложных организационных правил;
- прозрачной регистрации событий;
- изощренного аудита;
- поддержания синхронных дублирований таблиц;
- сбора статистики по обращениям к таблице.

Каскадные триггеры

Создать триггер каскадного изменения записей.

Таблица DEPARTMENTS и GOODS (а также таблица WORKERS) связаны между собой по полю Dept_id. При изменении данных этого поля в главной таблице (DEPARTMENTS) необходимо соответственно изменять данные в подчиненных таблицах (GOODS и WORKERS)



```
CREATE TRIGGER CASCADE_MODIFY  
AFTER UPDATE DEPT_ID ON DEPARTMENTS  
FOR EACH ROW  
BEGIN  
UPDATE GOODS SET DEPT_ID=:NEW.DEPT_ID  
WHERE DEPT_ID=:OLD.DEPT_ID;  
UPDATE WORKERS SET DEPT_ID=:NEW.DEPT_ID
```

```
WHERE DEPT_ID=:OLD.DEPT_ID;END;
```

Создать триггер каскадного удаления записей.



```
CREATE OR REPLACE TRIGGER CASCADE_DELETE
BEFORE DELETE ON DEPARTMENTS
FOR EACH ROW
BEGIN
DELETE FROM GOODS
WHERE DEPT_ID=:OLD.DEPT_ID;
DELETE FROM WORKERS
WHERE DEPT_ID=:OLD.DEPT_ID;
END;
```

Обратите внимание на необходимость самостоятельно создать аналогичные триггеры (изъятия и модификации) для таблицы Goods, потому что таблицы Goods и Sales связаны между собой с использованием связи «многие-к-одному» по полю Good_id и при наличии информации о продаже товаров в таблице Sales невозможно проверить работу триггеров каскадного модифицирования и удаления.



Результат работы триггера каскадного изменения показаны на рисунке 14

Oracle SQL*Plus

Файл Ред. Поиск Параметры Справка

SQL> UPDATE Departments SET dept_id=11 WHERE dept_id=1;

1 строка обновлена.

SQL> select * from goods;

GOOD_ID	NAME	PRICE	QUANTITY	PRODUCER	DEPT_ID	DESC
1	APPLE	7,4	12	UKRAINE	11	
2	PINEAPPLE	14,99	9	EGYPT	11	
3	BLACK CHOCOLATE	3,98	107	UKRAINE	2	78%
7	CAKE "HONEY"	29,99	2	AVK	3	
6	APPLE	12,55	128	SPAIN	2	
4	TOMATO	7,4	112	UKRAINE	11	
5	STRAWBERRY	39,99	4	TURKEY	2	
8	PIE "STARS"	26,00	3	RISANTT	3	

Рисунок 14 - Результат работы триггера каскадного изменения связанных строк

Просмотр информации о существующих триггерах

Такие представления словаря данных раскрывают информацию о триггерах:

- USER_TRIGGERS;
- ALL_TRIGGERS;
- DBA_TRIGGERS.

Если известно имя триггера (может быть получен выборкой из словаря данных), то можно использовать такие запросы, которые возвращают информацию о триггере TR1:

```
SELECT type, triggering_statement, table_name  
FROM user_triggers  
WHERE name = 'TR1';
```

Данный запрос возвращает информацию из представления USER_TRIGGERS о типе триггера (табличный триггер или триггер строки и его действия), о предложении триггера и названии таблицы, которой принадлежит триггер TR1.

```
SELECT trigger_body  
FROM user_triggers  
WHERE trigger_name = 'CASCADE_DELETE';
```

С помощью запроса, показанного на рисунке 15, можно просмотреть тело триггера.

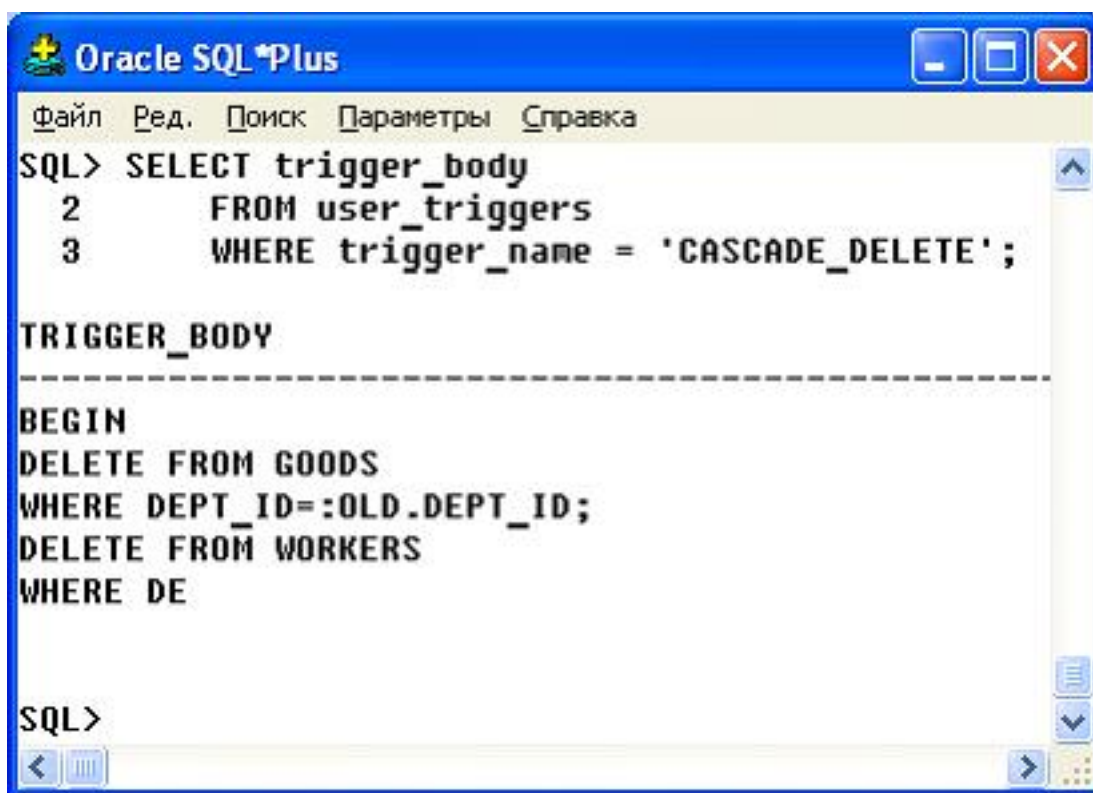


Рисунок 15 - Просмотр из словаря данных

Триггеры и комплексные проверки полномочий

Триггеры часто используются для реализации сложных проверок защиты данных таблицы. Используйте триггеры только для таких проверок полномочий, которые нельзя выполнить с помощью стандартных средств защиты базы данных в ORACLE.

Лучше всего для комплексной проверки полномочий использовать триггер предложения BEFORE. Это дает следующие преимущества:

- Контроль осуществляется до исполнения предложения триггера, так что не придется отменять выполненную работу, если предложение будет подвергнуто откату.
- Контроль осуществляется лишь один раз для предложения триггера, а не по каждой строке, затрагиваемой этим предложением.

Написать триггер запрета проведения операций по торговле во время выходных дней, а также в нерабочие часы.



```
CREATE or REPLACE TRIGGER goods_work
```

```
BEFORE INSERT OR DELETE OR UPDATE ON goods

DECLARE

not_on_weekends EXCEPTION;

non_working_hours EXCEPTION;

BEGIN

/* проверить на выходные дни */

IF ((To_char(sysdate, 'DY') = 'SAT') OR (To_char(sysdate, 'DY') = 'SUN')) THEN

RAISE not_on_weekends;

END IF;

/* Проверить на рабочие часы (8am .. 6pm) */

IF ((To_char(sysdate, 'HH24') < 8) OR (To_char(sysdate, 'HH24') > 18)) THEN

RAISE non_working_hours;

END IF;

EXCEPTION

WHEN not_on_weekends THEN

raise_application_error (-20324, 'May not change quantity of goods during the weekend');

WHEN non_working_hours THEN

raise_application_error (-20326, 'May not change quantity of goods during non-working hours');

END;

/
```

Результат работы триггера показан на рисунке 16

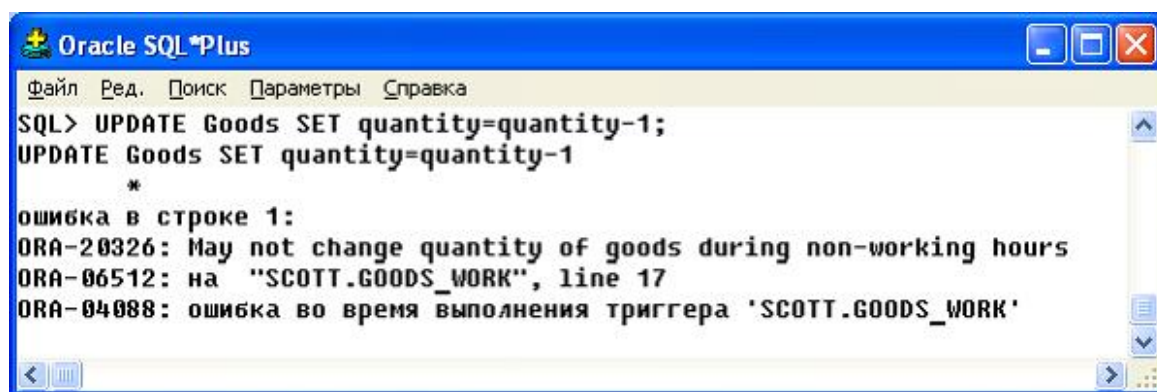


Рисунок 16 - Результат работы триггера

Аудит с помощью триггеров

Нам необходима информация, когда обращаются к таблице Goods , и какие типы запросов выдаются. Следующий пакет и триггер отслеживает эту информацию по часам и типам действий (например, UPDATE, DELETE или INSERT) по таблице Goods . Глобальная переменная сессии ITOG .ROW_ OPER инициализируется нулевым значением триггером предложения BEFORE, затем наращивается при каждом выполнении триггера строки, и, наконец, накопленная статистика сохраняется в таблице ITOG _TAB триггером предложения AFTER.



```
CREATE TABLE itog_tab
(oper_type VARCHAR2(10),
row_oper INTEGER,
Time_oper VARCHAR2(20));

CREATE OR REPLACE PACKAGE itog IS

Row_oper INTEGER;

END;

/

CREATE TRIGGER T1
BEFORE UPDATE OR DELETE OR INSERT ON Goods

BEGIN

itog.row_oper:= 0;
```

```
END;

/

CREATE TRIGGER T2
BEFORE UPDATE OR DELETE OR INSERT ON Goods
FOR EACH ROW
BEGIN
    itog.row_oper:= itog.row_oper + 1;
END;

/

CREATE TRIGGER T3
AFTER UPDATE OR DELETE OR INSERT ON Goods
DECLARE
    typ VARCHAR2(10);
    hour VARCHAR2(20);
BEGIN
    IF updating THEN
        typ := 'update';
    END IF;
    IF deleting THEN
        typ := 'delete';
    END IF;
    IF inserting THEN
        typ := 'insert';
    END IF;
    hour := TO_CHAR(SYSDATE, DD:MM:YY - HH:MIN:SS');
```



```
INSERT INTO itog_tab VALUES (itog.row_oper, typ, hour);
```

```
END;
```

```
/
```

Задачи для самостоятельного выполнения

Разработайте триггер копирования каждой второй удаляемой строки.



Разработайте триггер копирования каждой третьей изменяемой строк.



Разработайте триггер, с помощью которого можно при покупке оставлять хотя бы один товар (уменьшить количество товаров до 0 нельзя).



Разработайте триггер ограничения изменения цены, например, с увеличением цены более чем на 50%, применить лишь 50% увеличения цены.



С помощью триггера запретить добавление товара, если в том же отделе уже есть товары общей стоимостью более чем 10000.



Далее следует выполнить лабораторную работу 3

Лабораторная работа 3. Триггеры объектов и событий базы данных

Цель работы

Приобретение практических навыков и закрепление теоретических сведений о создании триггеров баз данных. Разработка триггеров различных типов для пользовательских таблиц и изучение основные особенности работы с триггерами.

Методические указания к выполнению работы

Для выполнения работы необходимо изучить материалы лекции этого раздела. Обратите внимание на отличия в создании и использовании триггеров строки и триггеров предложения, возможности доступа к данным из триггеров строки. Рассмотрите различные области применения триггеров. Для всех написанных триггеров написать команды, демонстрирующие их работу.

Ход работы

Написать триггер аудита для своей базы данных.



Написать триггеры каскадного обновления и удаления для тех таблиц своей базы данных, для которых это целесообразно.



Написать триггер для вычисляемого поля одной из таблиц своей базы данных.



Написать триггер проверки полномочий для своей базы данных.



Вопросы для самоконтроля:

- а) Что такое триггеры базы данных?
- б) Какие типы триггеров существуют?
- в) Приведите пример для обработки исключения, возникающего в триггерах.
- г) Расскажите о возможностях применения триггеров и ограничениях, возникающих при их применении.
- д) Как вызвать триггер базы данных?



Указания к оформлению отчета

Отчеты по лабораторной работе можно оформлять и представлять преподавателю в электронном виде или в твердой копии.

Отчет должен содержать:

- название работы;
- цель работы;
- формулировку заданий для самостоятельного выполнения;
- результаты выполненных заданий, включающие тексты самостоятельно разработанных триггеров, а также экранные формы утилиты SQL * Plus , демонстрирующие работу триггеров;
- выводы по работе.

Текущий контроль знаний

Управляющие структуры

Управляющие структуры

Какой даст результат выполнение оператора при значении $x=null$:

If $x=2$ then DBMS_OUTPUT.put_line(Значение переменной x равно 2);

else DBMS_OUTPUT.put_line(Значение переменной x не равно 2); end if;

- ☐ Печать строки «Значение переменной x равно 2»
- ☐ Печать строки «Значение переменной x не равно 2»
- ☐ Печать строки «Значение не определено»
- ☐ Управление передастся следующим операторам без результатов.

Управляющие структуры

Какие управляющие структуры не являются циклами (PL/SQL)?

- ☐ For ...loop ... end loop;
- ☐ Loop ... end loop;
- ☐ While ...loop ... end loop;
- ☐ Repeat ... loop ... end loop;

Управляющие структуры

Какие из приведенных операторов корректно выполняют следующую задачу: увеличить зарплату на 200 всем начальникам отделов (head of department), всем программистам - на 150, а остальному персоналу - на 100.

☐ If position=Head of department then update workers set salary=salary+200;

elseif position=Software engineer then update workers set salary=salary+150;

else update workers set salary=salary+100;

end if;

☐ Case

when position=Head of department update workers set salary=salary+200;

when position=Software engineer update workers set salary=salary+150;

else update workers set salary=salary+100;

end case;

☐ Case

when position=Head of department then update workers set salary=salary+200;

when position=Software engineer then update workers set salary=salary+150;

else update workers set salary=salary+100;

end case;

☐ If position=Head of department then update workers set salary=salary+200;

elseif position=Software engineer then update workers set salary=salary+150;

else update workers set salary=salary+100;

end if;

☐ If position=Head of department then update workers set salary=salary+200;

else if position=Software engineer

then update workers set salary=salary+150;

```
else update workers set salary=salary+100;  
end if;
```

Управляющие структуры

Каким образом описывается переменная цикла For...Loop?

- ☐ Как глобальная переменная целого типа.
- ☐ Как локальная переменная целого типа.
- ☐ Не описывается.
- ☐ Переменная типа Number.

Управляющие структуры

Какой из предложенных вариантов корректно напечатает каждый символ строки Example1 на отдельной строке.

- ☐ For i in 1 .. length(Example1)

loop

```
DBMS_OUTPUT.put_line(Substr(Example1,i,1));
```

end loop;

- ☐ For i :=1 to length(Example1)

loop

```
DBMS_OUTPUT.put_line(Substr(Example1,i,1));
```

end loop;

- ☐ i:=0

```
loop  
  
exit when i<length(Example1)  
  
DBMS_OUTPUT.put_line(Substr (Example1,1));  
  
end loop;
```

Управляющие структуры

Какие операторы выполняют выход из цикла?

- ☐ exit;
- ☐ goto end loop;
- ☐ break;
- ☐ close;
- ☐ exit when;

Управляющие структуры

Выберете верные утверждения

- ☐ На счетчик цикла можно ссылаться извне.
- ☐ Нельзя менять диапазон цикла внутри самого цикла.
- ☐ Можно уменьшать значения счетчика от большего к меньшему.
- ☐ Выражения, используемые при определении значений диапазона счетчика цикла вычисляются один раз.
- ☐ Переменная цикла должна быть определена пользователем как локальная переменная, имеющая тип Integer.
- ☐ Можно изменять значения счетчика произвольно.

Управляющие структуры

Какие фрагменты процедуры выполнения вычислений с использованием функции `calc_function` только для четных чисел (в диапазоне от 1 до 100) являются корректными?

☐ `for l_index in 1 .. 100`

`loop`

`if mod (l_index, 2)=0 then res:= calc_function(l_index);`

`end if;`

`end loop;`

☐ `for l_index in 1 .. 100`

`loop`

`if mod (l_index, 2)=1 then i:=i+1; end if;`

`res:= calc_function(l_index);`

`end loop;`

☐ `for l_index in 1 .. 100 step 2`

`loop`

`res:= calc_function(l_index);`

`end loop;`

☐ `for l_index in 1 .. 50`

`loop`

`res:= calc_function(l_index*2);`

`end loop;`

Управляющие структуры

Выберете корректный способ описания метки цикла

☐ `<<label_name>>`

`LOOP ...`

```
END LOOP label_name;
```

☐ --label_name--

```
LOOP ...
```

```
END LOOP label_name;
```

☐ DECLARE

```
label_name LABEL;
```

```
BEGIN
```

```
label_name
```

```
LOOP ...
```

```
END LOOP label_name;
```

```
END;
```

Управляющие структуры.

Каждая фраза в предложении IF должны содержать хотя бы одно выполнимое предложение.

☐ Верно ☐ Неверно

Подпрограммы языка PL/SQL

Подпрограммы

Какие из приведенных вызовов функции (function f1 (job1 varchar2) return number) являются корректными?

☐ select f1 (Manager) from dual;

☐ select f1 (job1=>Manager) from dual;

- ☐ select f1 from dual;
- ☐ select f1 (Manager) from workers;
- ☐ select f1 Manager from dual;
- ☐ select f1 (Manager);

Подпрограммы

Какая из приведенных подпрограмм может быть использована для добавления записи в таблицу Workers (w_id number(6) primary key, name varchar2(20), job varchar2(20), salary number (6,2)). Имеется последовательность s_workers.

- ☐ procedure p1 (name1 varchar2, job1 varchar2, salary1 number) is
begin
 insert into workers values (s_workers.nextval, name1, job1, salary1);
 commit;
end;
- ☐ function f1 (name1 varchar2, job1 varchar2, salary1 number) return row is
begin
 insert into workers values (s_workers.nextval, name1, job1, salary1);
 commit;
end;
- ☐ procedure p1 (name1 varchar2(20), job1 varchar2(20), salary1 number(6,2)) is
begin
 insert into workers values (s_workers.nextval, name1, job1, salary1);
 commit;
end;

```
☐ function f1 (name varchar2, job varchar2, salary number) return row is  
  
begin  
  
    insert into workers values (s_workers.nextval, name, job, salary);  
  
    commit;  
  
end;
```

Подпрограммы

Процедуры отличаются от функций тем, что:

- ☐ Процедуры имеют только выходные параметры, но не входные.
- ☐ Процедуры имеют входные параметры, а функции - нет.
- ☐ Функции имеют выходной параметр, а процедуры - нет.

подпрограммы

Создание какой из приведенных функций позволит посчитать количество всех сотрудников, находящихся на определенной должности. Используется таблица Workers (w_id number(6) primary key, name varchar2(20), job varchar2(20), salary number (6,2))?

```
☐ CREATE OR REPLACE FUNCTION f1 (job1 varchar2) return number IS  
  
res number(4)  
  
BEGIN  
  
    select count (*) into res from workers where job=job1;  
  
    RETURN res;  
  
END;
```

❑ CREATE OR REPLACE FUNCTION f1 (job1 varchar2(20)) return number IS

res number(4)

BEGIN

select count (*) into res from workers where job=job1;

RETURN res;

END;

❑ CREATE OR REPLACE FUNCTION f1 (job1 varchar2) return varchar2 IS

res number(4)

BEGIN

select count (job1) into res from workers;

RETURN res;

END;

❑ CREATE FUNCTION f1 (job1 varchar2) return number IS

res number(4)

BEGIN

res:=select count (distinct job) from workers where job=job1;

RETURN res;

END;

Подпрограммы

При компиляции какой из приведенных функций будет выведено сообщение об ошибке «То_many_rows»? Используется таблица Workers (w_id number(6) primary key, name varchar2(20), job varchar2(20), salary number (6,2)).

❑ function f1 (job1 varchar2, salary1 number) return varchar IS

```
res varchar2(20)

begin

    select name into res where job=job1 and salary>salary1;

    return res;

end;
```

☐ function f1 (job1 varchar2, salary1 number) return varchar IS

```
res varchar2(20)

begin

    select name into res where job=job1 and salary>salary1 and rownumber=1;

    return res;

end;
```

☐ function f1 (job1 varchar2, salary1 number) return number IS

```
res number(6)

begin

    select count(name) into res where job=job1 and salary>salary1;

    return res;

end;
```

☐ function f1 (job1 varchar2) return varchar IS

```
res varchar2(20)

begin

    select name into res where job=job1 and salary>salary1;

    return res;

end;
```

Подпрограммы

Какая из приведенных процедур корректно заполняет таблицу Workers (w_id number(6) primary key, name varchar2(20), job varchar2(20), salary number (6,2))? Имеется последовательность s_workers.

☐ create procedure p1 (name1 varchar2, job1 varchar2, salary1 number) IS BEGIN insert into Workers values (s_workers.nextval, name1, job1, salary1);

COMMIT;

END;

☐ create procedure p1 (name varchar2, job varchar2, salary number) IS BEGIN insert into Workers values (s_workers.nextval, name, job, salary);

COMMIT;

END;

☐ create procedure p1 (name1 varchar2, job1 varchar2, salary1 number) IS BEGIN insert into Workers values (name1, job1, salary1);

COMMIT;

END;

☐ create procedure p1 (name1 varchar2, job1 varchar2, salary1 number) IS BEGIN insert into Workers values (s_workers.nextval, job1, name1, salary1);

COMMIT;

END;

Подпрограммы

Какие из приведенных вызовов процедуры (procedure p1 (name1 varchar2, job1 varchar2, salary1 number)) являются корректными?

☐ exec p1 (Smith, Manager, 800);

- ☐ select p1 ("Smith", Manager, 800) from dual;
- ☐ select p1 (Smith, Manager, 800);
- ☐ exec p1 (800, Smith, Manager);
- ☐ exec p1 (name1=>Smith, job1=>Manager, salary1=>800);
- ☐ exec p1 (Smith, job1=>Manager, 800);

Подпрограммы

Имеется таблица Workers (w_id number(6) primary key, name varchar2(20), job varchar2(20), salary number (6,2)). и процедура P1:

Procedure P1 (job1 varchar2) IS

BEGIN

update Workers set salary=salary+salary*0.1 where job=job1;

COMMIT;

nd;

Что произойдет при вызове процедуры P1 : exec p1(Manager);

- ☐ Для всех сотрудников произойдет увеличение зарплаты на 10%.
- ☐ Добавится поле для всех сотрудников, которые работают менеджерами, в котором будет записана новая зарплата.
- ☐ Для всех сотрудников, которые работают менеджерами, произойдет увеличение зарплаты на 10%.

Подпрограммы

Создание каких из приведенных функций позволит выяснить максимальную зарплату сотрудников, занимающих определенную должность.

Используется таблица Workers (w_id number(6) primary key, name varchar2(20), job varchar2(20), salary number (6,2))?

❑ CREATE OR REPLACE FUNCTION f1 (job1 varchar2) return workers.salary%type IS

res workers.salary%type

BEGIN

select max(salary) into res from workers where job=job1;

RETURN res;

END;

❑ CREATE OR REPLACE FUNCTION f1 (job1 varchar2(20)) return number IS

res workers.salary%type

BEGIN

res:=select max(salary) from workers where job=job1;

RETURN res;

END;

❑ CREATE FUNCTION f1 (job1 varchar2) IS res number(6)

BEGIN

select max(salary) into res from workers where job=job1;

RETURN res;

END;

❑ CREATE OR REPLACE FUNCTION f1 (job varchar2) return workers.salary%type IS

res workers.salary%type

BEGIN

return select max(salary) from workers where job=job;

END;

Попрограммы

Какие из приведенных вызовов процедуры (procedure p1 (name1 varchar2, job1 varchar2, salary1 number)) являются корректными?

☐ exec p1 (Smith, Manager, 800);

☐ select p1 (Smith, Manager, 800) from dual;

☐ /*Процедуры p1 и p2 находятся в одном пакете*/

procedure p2 is begin

exec p1 (name1=>Smith, job1=>Manager, salary1=>800); end;

☐ /*Процедуры p1 и p2 находятся в одном пакете*/

procedure p2 is begin

p1 (name1=>Smith, job1=>Manager, salary1=>800); ... end;

☐ /*Процедуры p1 и p2 не находятся в одном пакете*/

procedure p2 is begin

p1 (name1=>Smith, job1=>Manager, salary1=>800); ... end;

☐ function f2 return number is begin

p1 (name1=>Smith, job1=>Manager, salary1=>800); ...end;

Курсоры

Курсоры

Какой атрибут курсора не возвращает исключение INVALID_CURSOR при обращении к еще не открытому курсору?

☐ %FOUND

☐ %NOTFOUND

☐ %ISOPEN

- ☐ %ROWCOUNT

Курсоры

Использованием какой конструкции можно заблокировать строки, возвращаемые запросом?

- ☐ CURSOR ... FOR UPDATE
- ☐ SELECT ... FOR UPDATE
- ☐ FETCH
- ☐ SELECT ... INTO

Курсоры

Какому термину соответствует следующее определение «Объявленная программистом переменная, указывающая на объект типа курсора в базе данных»?

- ☐ Явный курсор.
- ☐ Неявный курсор.
- ☐ Курсорная переменная.
- ☐ Динамическая переменная.

Курсоры

Какие конструкции позволяют полностью организовать работу с курсором?

- ☐ Цикл FOR с курсором

- ☐ Операторы OPEN, FETCH
- ☐ Атрибуты %NOTFOUND, %ISOPEN

Курсоры

Курсор - это:

- ☐ Временная область памяти для PL/SQL.
- ☐ Выполняемый модуль кода.
- ☐ Рабочая область для операторов SQL, возвращающих более одной строки.
- ☐ Рабочая область и область хранения для PL/SQL.

Курсоры

Неявные курсоры отличаются от явных тем, что:

- ☐ Инструкция SELECT, определяющая неявный курсор определяется в исполняемом блоке, а не в разделе объявлений, как явные курсоры.
- ☐ Все операции (открытие, выборки и закрытия) для неявного курсора выполняются автоматически, а для явных - задаются программистом.
- ☐ Все операции (открытие, выборки и закрытия) для явного курсора выполняются автоматически, а для неявных - задаются программистом.
- ☐ Если строка не найдена явный курсор вызывает завершение блока, а неявный курсор инициирует исключение.
- ☐ Если строка не найдена неявный курсор вызывает завершение блока, а явный курсор инициирует исключение.

Курсоры

Укажите правильную последовательность операций с запросами и курсорами.

☐ 1) синтаксический анализ;

2) привязка;

3) открытие;

4) выполнение;

5) выборка;

6) закрытие.

☐ 1) привязка;

2) открытие;

3) синтаксический анализ;

4) выборка;

5) выполнение;

6) закрытие.

☐ 1) синтаксический анализ;

2) открытие;

3) выполнение;

4) выборка;

5) закрытие.

☐ 1) открытие;

2) синтаксический анализ;

3) привязка;

4) выполнение;

5) выборка;

6) закрытие.

Курсоры

Какие утверждения верны?

- ☐ Курсорная переменная не может быть объявлена в пакете.
- ☐ Курсорную переменную можно передавать между серверами с помощью удаленных вызовов процедуры.
- ☐ Курсорную переменную нельзя с помощью операторов сравнения проверять на равенство или неравенство какому-либо значению.
- ☐ Курсорную переменную нельзя передавать в качестве аргумента процедуре или функции.
- ☐ Значение курсорной переменной может храниться в столбце базы данных.

Курсоры

Какая из приведенных процедур заполняет таблицу Low_salary, в которой отображаются данные в следующем формате «Должность, минимальная зарплата, средняя зарплата», для должностей, сотрудники, находящиеся на которых получают зарплату меньше средней по предприятию.

Имеется таблица Workers (w_id number(6) primary key, name varchar2(20), job varchar2(20), salary number(6,2)).

☐ procedure p1 is

job1 workers.job%type;

min1 workers.salary%type;

avg1 workers.salary%type;

cursor C1 is select job, min(salary), avg(salary) from workers group by job having avg(salary)< (select avg(salary) from workers);

begin

if not C1%ISOPEN then open C1;

```
        end if;

    loop

        fetch C1 into job1, min1, avg1;

        exit when C1%NOTFOUND ;

        insert into Low_salary values (job1, min1, avg1);

    end loop;

    close C1;

    commit;

end;
```

□ procedure p1 is

```
job1 workers.job%type;

min1 workers.salary%type,

avg1 workers.salary%type,

cursor C1 is select job, min(salary), avg(salary) from workers group by job having avg(salary)< (select
avg(salary) from workers);

begin

    if C1%ISNOTOPEN then open C1;

    end if;

    loop

        fetch C1 into job1, min1, avg1;

        exit when C1%NOTFOUND ;

        insert into Low_salary values (job1, min1, avg1);

    end loop;

    close C1;

    commit;
```

```
end;
```

```
□ procedure p1 is
```

```
job1 workers.job%type;
```

```
min1 workers.salary%type;
```

```
avg1 workers.salary%type;
```

```
cursor C1 is select job, min(salary), avg(salary) from workers group by job having avg(salary)< (select  
avg(salary) from workers);
```

```
begin
```

```
loop
```

```
    exit when C1%NOTFOUND;
```

```
    fetch C1 into job1, min1, avg1;
```

```
    insert into Low_salary values (job1, min1, avg1);
```

```
    close C1;
```

```
end loop;
```

```
commit;
```

```
end;
```

```
□ procedure p1 is
```

```
job1 workers.job%type,
```

```
min1 workers.salary%type,
```

```
avg1 workers.salary%type,
```

```
cursor C1 is select job, min(salary), avg(salary) from workers group by job having avg(salary)< (select  
avg(salary) from workers),
```

```
begin
```

```
    open C1;
```

```
    loop
```

```
        exit when C1%NOTFOUND

    fetch C1 into job1, min1, avg1;

    insert into Low_salary values (job1, min1, avg1);

    end loop;

close C1;

commit;

end;
```

□ procedure p1 is

cursor C1 is select job, min(salary), avg(salary) from workers group by job having avg(salary)< (select avg(salary) from workers);

```
begin

    if C1%ISNOTOPEN then open C1;

    end if;

    fetch C1 into job1, min1, avg1;

    insert into Low_salary values (job1, min1, avg1);

    close C1;

    commit;

end;
```

Курсоры

Процедура, приведенная ниже, выведет на экран данные о сотрудниках с максимальной зарплатой в формате «<ФИО> - работает <должность>, зарплата - <salary>» (Имеется таблица Workers (w_id number(6) primary key, name varchar2(20), job varchar2(20), salary number (6,2))). Так ли это?

.

Procedure p1 is

```
cursor C1 is name, job, salary from workers where salary>= (select max(salary) from workers);  
  
name1 workers.name%type,  
job1 workers.job%type,  
salary1 workers.salary%type;  
  
begin  
    open C1;  
  
loop  
    fetch C1 into name1, job1, salary1;  
    exit when C1%NOTFOUND;  
    BMS_OUTPUT.put_line (ФИО ||name1|| - работает ||job1|| , зарплата - || salary1);  
  
end loop;  
  
close C1;  
  
end;
```

☐ Верно ☐ Неверно

Исключительные ситуации

Исключительные ситуации

Каким образом можно перейти в обработчик исключительных ситуаций?

- ☐ Оператором GOTO
- ☐ Оператором EXIT
- ☐ Автоматически при возникновении исключительной ситуации.
- ☐ Оператором RAISE

Исключительные ситуации

Какие атрибуты курсора доступны при возбуждении исключительной ситуации внутри курсорного цикла?

- ☐ %FOUND, %NOTFOUND.
- ☐ %ROWCOUNT.
- ☐ Все атрибуты курсора доступны.
- ☐ Все атрибуты курсора недоступны.

Исключительные ситуации

Условие блока обработки ошибки WHEN_OTHERS может применяться в блоке EXCEPTIONS:

- ☐ Как первое условие.
- ☐ Как последнее условие.
- ☐ В любом месте, порядок расположения условий не имеет значения.

Исключительные ситуации

Если ошибочная ситуация не обработана в блоке PL/SQL, то она:

- ☐ Игнорируется.
- ☐ Распространяется на вызывающую программу.
- ☐ Вызывает ошибку в блоке PL/SQL.
- ☐ Обрабатывается в соответствии с правилами, применяемыми в языке PL/SQL по умолчанию.

Исключительные ситуации

Какие из приведенных утверждений верны?

- ☐ В PL/SQL отсутствует структурное различие между внутренними ошибками и ошибками, специфическими для приложений.
- ☐ Имена исключений по своему формату отличаются от имен логических переменных.
- ☐ Обработать исключение - значит перехватить ошибку, передав управление обработчику исключений.
- ☐ Из обработчика исключений нельзя вернуться в процедуру, которая сгенерировала исключительную ситуацию.
- ☐ Нельзя переопределить стандартные исключения.

Исключительные ситуации

Какие из приведенных утверждений верны?

- ☐ Инициировать и обрабатывать именованные системные исключения можно в любом блоке.
- ☐ Областью действия именованного исключения, определенного программистом, определенного в спецификации пакета, являются те программы, владельцы которых имеют на этот пакет привилегию EXECUTE
- ☐ Инициировать и обрабатывать именованные исключения, определенные программистом, можно только в том исполнительном разделе и разделе исключений, которые входят в состав блока, где объявлено данное исключение.
- ☐ Анонимное системное исключение можно обрабатывать только после переопределения программистом.
- ☐ Областью действия именованного исключения, определенного программистом, определенного в спецификации пакета, являются только те программы, которые входят в состав пакета.

Исключительные ситуации

Каждая ошибка ORACLE имеет номер, по которым они и обрабатываются

☐ Верно ☐ Неверно

Исключительные ситуации

При помощи исключений можно обрабатывать ошибки, не кодируя многочисленных проверок

☐ Верно ☐ Неверно

Пакеты

Пакеты

Что для пакета является обязательной составляющей?

☐ Спецификация пакета.

☐ Тело пакета.

☐ Функции и процедуры.

☐ Курсоры.

Пакеты

Какие утверждения верны?

☐ Внешняя, по отношению к пакету программа, не может обращаться к элементам, определенным внутри пакета.

☐ Для того чтобы личный объект (по отношению к пакету) стал общим (доступным для любых программ), достаточно добавить этот объект в спецификацию пакета.

☐ Тело пакета не может содержать элементы, не описанные в спецификации пакета.

☐ Если элемент определяется в спецификации пакета, то для ссылки на него из внешней

программы необходимо использовать точный синтаксис: имя_пакета.имя_элемента.

- ☐ Раздел исключений может присутствовать в пакете только тогда, когда в пакете имеется инициализационный раздел.
- ☐ Все правила и ограничения, используемые при объявлении структур данных уровня пакета в его спецификации, не относятся к телу пакета.

Пакеты

Укажите правильный порядок инициализации пакета.

- ☐ 1. Присваиваются переменным и константам значения по умолчанию, указанные в их объявлениях.
2. Выполняется блок кода, содержащийся в инициализационном разделе.
3. Обрабатываются данные уровня пакета (значения переменных и констант).
- ☐ 1. Присваиваются переменным и константам значения по умолчанию, указанные в их объявлениях.
2. Обрабатываются данные уровня пакета (значения переменных и констант).
3. Выполняется блок кода, содержащийся в инициализационном разделе.
- ☐ 1. Обрабатываются данные уровня пакета (значения переменных и констант).
2. Выполняется блок кода, содержащийся в инициализационном разделе.
3. Присваиваются переменным и константам значения по умолчанию, указанные в их объявлениях.
- ☐ 1. Обрабатываются данные уровня пакета (значения переменных и констант).
2. Присваиваются переменным и константам значения по умолчанию, указанные в их объявлениях.
3. Выполняется блок кода, содержащийся в инициализационном разделе.

Пакеты

Какие утверждения верны?

- ☐ В теле пакета нельзя объявлять курсорные переменные.
- ☐ Если ссылка на элемент осуществляется в самом пакете, то необходимо использовать точный синтаксис: имя_пакета.имя_элемента.
- ☐ Если в результате присвоения значения по умолчанию инициируется исключение, то оно перехватывается в пакете и может быть обработано.
- ☐ Переменная уровня пакета, получившая значение при выполнении программы, сохраняет свое значение в течении всего сеанса, даже если выполнение присвоившей его программы завершается.
- ☐ В теле пакета находится исполняемый раздел, который выполняется только один раз для инициализации пакета.

Пакеты

Видимыми и доступными для приложений являются лишь объявления в спецификации пакета.

- ☐ Верно ☐ Неверно

Пакеты

Имеется пакет, спецификация которого приведена ниже. Выберите корректный вызов элементов пакета.

```
PACKAGE PACK1 IS
```

```
procedure p1 (n number, v varchar2);
```

```
function f1 (n number, v varchar2) return number;
```

```
END PACK1;
```

- ☐ EXEC p1(23,'example 1');
- ☐ SELECT f1 (12, 'rxample 2') from pack1;

☐ SELECT f1 (12, 'example 2') from dual;

☐ EXEC pack1.p1(23,'example 1');

Пакеты

Установите правильную последовательность выполнения независимых или пакетированных процедур.

	Проверка прав пользователя.
	Исполнение процедуры.
	Проверка действительности процедуры.

Пакеты

При помещении курсора в пакет можно отделить спецификацию курсора от его тела, используя фразу RETURN

☐ Верно ☐ Неверно

Пакеты

Хранимая подпрограмма не может вызывать пакетированную подпрограмму

☐ Верно ☐ Неверно

Триггеры

Триггеры

Какие утверждения верны?

- ☐ Триггер уровня строки не может считывать или записывать данные таблицы, с которой он связан.
- ☐ Триггер уровня предложения не может считывать или записывать данные таблицы, с которой он связан.
- ☐ Если триггер выполняется как автономная транзакция, тогда в нем можно запрашивать содержимое его таблицы.
- ☐ Если триггер выполняется как автономная транзакция, тогда в нем можно модифицировать таблицу, с которой он связан.

Триггеры

Какие из приведенных типов триггеров событий базы данных могут быть созданы?

- ☐ BEFORE STARTUP
- ☐ AFTER SERVERERROR
- ☐ BEFORE LOGON
- ☐ BEFORE SHUTDOWN
- ☐ AFTER LOGOFF

Триггеры

Приведите порядок выполнения предложения SQL и триггеров.

	Триггер строки BEFORE.
	Триггер строки AFTER.
	Триггер предложения AFTER.
	Предложение SQL, инициирующее триггер.
	Триггер предложения BEFORE.

Триггеры

Оператор `:new.inv:=:old.inv` имеет следующее назначение:

- ☐ Позволяет исключить внесение изменений в содержимое поля `inv`.
- ☐ Устанавливает новое значение поля `inv`.
- ☐ Позволяет изменить значение поля `inv` другим процессам.

Триггеры

Какой из приведенных заголовков триггера соответствует задаче «Запрещать ввод данных в таблицу, если количество записей, соответствующих определенному условию достигло предела». Имеется таблица `Children (child_id number(6) primary key, name varchar2(20), group varchar2(10));`

- ☐ `CREATE OR REPLACE TRIGGER TR1 before insert on Children for each row...`
- ☐ `CREATE OR REPLACE TRIGGER TR1 after insert on Children for each row...`
- ☐ `CREATE OR REPLACE TRIGGER TR1 before insert on Children ..`
- ☐ `CREATE OR REPLACE TRIGGER TR1 after insert on Children ...`

Триггеры

Перед квалификаторами `OLD` и `NEW` должно кодироваться двоеточие, когда они используются в теле триггера, но двоеточие не допускается, когда эти квалификаторы используются в фразе `WHEN` или опции `REFERENCING`.

- ☐ Верно ☐ Неверно

Триггеры

Какой из приведенных триггеров корректно выполняет заказ товара (запись требуемого товара в

таблицу Orders), если количество товара стало меньше 20 единиц. Имеется таблица Storages (st_id number (6) primary key, goods varchar2(20), quantity number (6)).

☐ CREATE TRIGGER reorder

AFTER DELETE OR UPDATE OF quantity ON storages

FOR EACH ROW

WHEN (new.quantity < 20)

BEGIN

INSERT INTO orders VALUES (:new.storages_id, :new.goods, SYSDATE);

END;

☐ CREATE TRIGGER reorder

BEFORE DELETE OR UPDATE OF quantity ON storages

WHEN (new.quantity < 20)

BEGIN

INSERT INTO orders VALUES (:new.storages_id, :new.goods, SYSDATE);

END;

☐ CREATE TRIGGER reorder

AFTER DELETE OR UPDATE OF quantity ON storages

WHEN (new.quantity < 20)

BEGIN

INSERT INTO orders VALUES (:new.storages_id, :new.goods, SYSDATE);

END;

☐ CREATE TRIGGER reorder

AFTER DELETE OR UPDATE OF quantity ON storages

FOR EACH ROW

BEGIN

if :new.quantity <20 then INSERT INTO orders VALUES (:new.storages_id, :new.goods, SYSDATE);

END;

Триггеры

Какие утверждения верны?

- ☐ Триггер уровня инструкции выполняется для отдельной SQL-инструкции, которая может обрабатывать только одну запись.
- ☐ Псевдозапись OLD обладает такими же свойствами как и запись PL/SQL. Она доступна внутри триггеров обновления и вставки.
- ☐ Псевдозапись NEW обладает такими же свойствами как и запись PL/SQL. Она доступна внутри триггеров обновления и вставки.
- ☐ В триггере DML можно выполнять инструкции COMMIT и ROLLBACK.
- ☐ Если триггер инициирует исключение будет выполнен откат соответствующей транзакции, из которой запущен триггер.

Триггеры

Триггеры, используемые для поддержания целостности базы данных могут заменять декларативные ограничения целостности.

☐ Верно ☐ Неверно

Триггеры

Какое предложение SQL инициирует действие триггера, имеющего следующий заголовок:

TRIGGER control_quantity

BEFORE UPDATE OR INSERT ON storages

Имеется таблица storages (storage_id, date_in, price, quantity).

- ☐ EXEC T1;
- ☐ UPDATE storages SET price=price-10;
- ☐ UPDATE storages SET quantity=quantity-10;
- ☐ INSERT INTO storages VALUES (123, sysdate, 12.5, 45)
- ☐ DELETE FROM storages;

Какое предложение SQL инициирует действие триггера, имеющего следующий заголовок:

TRIGGER control_quantity

BEFORE UPDATE OF quantity ON storages

FOR EACH ROW

WHEN new.quantity<0;

Имеется таблица storages (storage_id, date_in, price, quantity).

- ☐ UPDATE storages SET price=price-10;
- ☐ EXEC T1;
- ☐ UPDATE storages SET quantity=quantity-10;

Словарь терминов

