PostgreSQL 9.4

Chapter 40. PL/pgSQL - процедурный язык SQL

40.1. Обзор

PL/pgSQL это процедурный язык для СУБД PostgreSQL. Целью проектирования PL/pgSQL было создание загружаемого процедурного языка, который:

- используется для создания функций и триггеров,
- добавляет управляющие структуры к языку SQL,
- может выполнять сложные вычисления,
- наследует все пользовательские типы, функции и операторы,
- может быть определен как доверенный язык,
- прост в использовании.

Функции PL/pgSQL могут использоваться везде, где допустимы встроенные функции. Например, можно создать функции со сложными вычислениями и условной логикой, а затем использовать их при определении операторов или в индексных выражениях.

В версии PostgreSQL 9.0 и выше, PL/pgSQL устанавливается по умолчанию. Тем не менее, это по-прежнему загружаемый модуль и администраторы, особо заботящиеся о безопасности, могут удалить его при необходимости.

40.1.1. Преимущества использования PL/pgSQL

PostgreSQL и большинство других СУБД используют SQL в качестве языка запросов. SQL хорошо переносим и прост в изучении. Однако каждый оператор SQL выполняется индивидуально на сервере базы данных.

Это значит, что ваше клиентское приложение должно каждый запрос отправлять на сервер, ждать пока он будет обработан, получать результат, делать некоторые вычисления, затем отправлять последующие запросы на сервер. Всё это требует межпроцессного взаимодействия, а также несет нагрузку на сеть, если клиент и сервер базы данных расположены на разных компьютерах.

PL/pgSQL позволяет сгруппировать блок вычислений и последовательность запросов *внутри* сервера базы данных, таким образом, мы получаем силу процедурного языка и простоту использования SQL при значительной экономии накладных расходов на клиент-серверное

взаимодействие.

- Исключаются дополнительные обращения между клиентом и сервером
- Промежуточные ненужные результаты не передаются между сервером и клиентом
- Есть возможность избежать многочисленных разборов одного запроса

В результате это приводит к значительному увеличению производительности по сравнению с приложением, которое не использует хранимых функций.

Кроме того, PL/pgSQL позволяет использовать все типы данных, операторы и функции SQL.

40.1.2. Поддерживаемые типы данных аргументов и возвращаемых значений

PL/pgSQL функции могут принимать в качестве аргументов все поддерживаемые сервером скалярные типы данных или массивы и возвращать в качестве результата любой из этих типов. Они могут принимать и возвращать именованные составные типы (строковый тип). Также есть возможность объявить PL/pgSQL функцию, возвращающую `record`, это означает, что результатом является строковый тип, чьи столбцы будут определены в спецификации вызывающего запроса, как описано в Разд. 7.2.1.4.

Использование маркера `VARIADIC` позволяет объявлять PL/pgSQL функции с переменным числом аргументов. Это работает точно так же, как и для SQL функций, как описано в Разд. 35.4.5.

PL/pgSQL функции могут принимать и возвращать полиморфные типы `anyelement`, `anyarray`, `anynonarray`, `anyenum` и `anyrange`. В таких случаях фактические типы данных могут меняться от вызова к вызову, как описано в Разд. 35.2.5. Пример показан в Разд. 40.3.1.

PL/pgSQL функции могут возвращать "set" (или таблицу) любого типа, который может быть возвращен в качестве одного экземпляра. Такие функции генерируют вывод, выполняя команду `RETURN NEXT` для каждого элемента результирующего набора или `RETURN QUERY` для вывода результата запроса.

Наконец, при отсутствии полезного возвращаемого значения PL/pgSQL функция может возвращать `void`.

PL/pgSQL функции можно объявить с выходными параметрами вместо явного задания типа возвращаемого значения. Это не добавляет никаких фундаментальных возможностей языку, но часто бывает удобно, особенно для возвращения нескольких значений. Нотация `RETURNS TABLE` может использоваться вместо `RETURNS SETOF`.

Конкретные примеры рассматриваются в Разд. 40.3.1 и Разд. 40.6.1.

40.2. Структура PL/pgSQL

PL/pgSQL это блочно-структурированный язык. Текст определения функции должен быть *блоком*. Структура блока:

```
[ <<метка>> ]
[ DECLARE
    объявления ]
BEGIN
    операторы
END [ метка ];
```

Каждое объявление и каждый оператор в блоке должны завершаться символом ";"(точка с запятой). Блок, вложенный в другой блок, должен иметь точку с запятой после `END`, как показано выше. Однако финальный `END`, завершающий тело функции, не требует точки с запятой.

> **Подсказка:** Распространенной ошибкой является добавление точки с запятой сразу после `BEGIN`. Это неправильно и приведет к синтаксической ошибке.

`Метка` требуется только тогда, когда нужно идентифицировать блок в операторе `EXIT`, или квалифицировать имена переменных, объявленных в этом блоке. Если метка указана после `END`, то она должна совпадать с меткой в начале блока.

Ключевые слова не чувствительны к регистру символов. Как и в обычных SQL командах, идентификаторы неявно преобразуются к нижнему регистру, если они не взяты в двойные кавычки.

Комментарии в PL/pgSQL коде работают так же, как и в обычном SQL. Двойное тире (`--`) начинает комментарий, который завершается в конце строки. Блочный комментарий начинается с `/*` и завершается `*/`. Блочные комментарии могут быть вложенными.

Любой оператор в выполняемой секции блока может быть *вложенным блоком*. Вложенные блоки используются для логической группировки нескольких операторов или локализации области действия переменных для группы операторов. Во время выполнения вложенного блока переменные, объявленные в нем, скрывают переменные внешних блоков с такими же именами. Чтобы получить доступ к внешним переменным, нужно квалифицировать их имена меткой блока. Например:

```
CREATE FUNCTION somefunc() RETURNS integer AS $$
<< outerblock >>
DECLARE
    quantity integer := 30;
BEGIN
    RAISE NOTICE 'Сейчас quantity = %', quantity;  -- Выводится 30
    quantity := 50;
    --
    -- Создаем вложенный блок
    --
    DECLARE
        quantity integer := 80;
```

```
    BEGIN
        RAISE NOTICE 'Сейчас quantity = %', quantity;  -- Выводится 80
        RAISE NOTICE 'Во внешнем блоке quantity = %', outerblock.quantity;  --
Выводится 50
    END;

    RAISE NOTICE 'Сейчас quantity = %', quantity;  -- Выводится 50

    RETURN quantity;
END;
$$ LANGUAGE plpgsql;
```

> **Замечание:** Существует скрытый "внешний блок", окружающий тело каждой PL/pgSQL функции. Этот блок содержит объявления параметров функции (если они есть), а также некоторые специальные переменные, такие как `FOUND` (смотри Разд. 40.5.5). Этот блок имеет метку, совпадающую с именем функции, таким образом, параметры и специальные переменные могут быть квалифицированы именем функции.

Важно не путать использование `BEGIN`/`END` для группировки операторов в PL/pgSQL с одноименными SQL командами для управления транзакциями. `BEGIN`/`END` в PL/pgSQL служат только для группировки предложений; они не начинают и не заканчивают транзакции. Функции и триггерные процедуры всегда выполняются в рамках транзакции, начатой во внешнем запросе - они не могут начать или завершить эту транзакцию, т.к. у них внутри нет контекста для выполнения таких действий. Однако блок содержащий секцию `EXCEPTION` создает вложенную транзакцию, которая может быть отменена, не затрагивая внешней транзакции. Дополнительно об этом смотри Разд. 40.6.6.

40.3. Объявления

Все переменные, используемые в блоке, должны быть определены в секции объявления. (За исключением переменной-счетчика цикла `FOR`, которая объявляется автоматически. Для цикла по диапазону чисел автоматически объявляется целочисленная переменная, а для цикла по результатам курсора - переменная типа `record`.)

PL/pgSQL переменные могут иметь любой тип данных SQL, такой как `integer`, `varchar`, `char`.

Примеры объявления переменных:

```
user_id integer;
quantity numeric(5);
url varchar;
myrow tablename%ROWTYPE;
myfield tablename.columnname%TYPE;
arow RECORD;
```

Общий синтаксис объявления переменной:

```
name [ CONSTANT ] type [ COLLATE collation_name ] [ NOT NULL ] [ { DEFAULT | :=
| = } expression ];
```

Фраза DEFAULT, если присутствует, задает начальное значение, которое присваивается переменной при входе в блок. Если отсутствует, то переменная инициализируется SQL значением NULL. Опция CONSTANT предотвращает изменение значения переменной после инициализации, таким образом, значение остается постоянным в течение всего блока. Опция COLLATE определяет схему упорядочения, которая будет использоваться для этой переменной (смотри Разд. 40.3.6). Если указано NOT NULL, то попытка присвоить NULL во время выполнения приведет к ошибке. Все переменные, объявленные как NOT NULL, должны иметь не пустые значения по умолчанию. Можно использовать знак равенства (=) вместо совместимого с PL/SQL :=.

Значение по умолчанию вычисляется и присваивается переменной каждый раз при входе в блок (не только при первом вызове функции). Так, например, если переменная типа timestamp имеет функцию now() в качестве значения по умолчанию, это приведет к тому, что переменная всегда будет содержать время текущего вызова функции, а не время, когда функция была предварительно скомпилирована.

Примеры:

```
quantity integer DEFAULT 32;
url varchar := 'http://mysite.com';
user_id CONSTANT integer := 10;
```

40.3.1. Объявление параметров функции

Переданные в функцию параметры именуются идентификаторами $1, $2, и т.д. Дополнительно, для улучшения читаемости, можно объявить псевдонимы для параметров $*n*. Либо псевдоним, либо цифровой идентификатор используются для обозначения параметра.

Создать псевдоним можно двумя способами. Предпочтительный способ это дать имя параметру в команде CREATE FUNCTION, например:

```
CREATE FUNCTION sales_tax(subtotal real) RETURNS real AS $$
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

Другой способ это явное объявление псевдонима при помощи синтаксиса:

```
name ALIAS FOR $n;
```

Предыдущий пример для этого стиля выглядит так:

```
CREATE FUNCTION sales_tax(real) RETURNS real AS $$
DECLARE
    subtotal ALIAS FOR $1;
BEGIN
    RETURN subtotal * 0.06;
END;
```

```
$$ LANGUAGE plpgsql;
```

> **Замечание:** Эти два примера не полностью эквивалентны. В первом случае, на `subtotal` можно ссылаться как `sales_tax.subtotal`, а во втором случае такая ссылка невозможна. (Если бы к внутреннему блоку была добавлена метка, то `subtotal` можно было бы квалифицировать этой меткой.)

Еще несколько примеров:

```
CREATE FUNCTION instr(varchar, integer) RETURNS integer AS $$
DECLARE
    v_string ALIAS FOR $1;
    index ALIAS FOR $2;
BEGIN
    -- здесь вычисления, использующие v_string и index
END;
$$ LANGUAGE plpgsql;


CREATE FUNCTION concat_selected_fields(in_t sometablename) RETURNS text AS $$
BEGIN
    RETURN in_t.f1 || in_t.f3 || in_t.f5 || in_t.f7;
END;
$$ LANGUAGE plpgsql;
```

Когда PL/pgSQL функция объявляется с выходными параметрами, им выдаются цифровые идентификаторы $*n* и для них можно создавать псевдонимы точно таким же способом, как и для обычных входных параметров. Выходной параметр это фактически переменная, стартующая с NULL и которой присваивается значение во время выполнения функции. Возвращается последнее присвоенное значение. Например, функция sales_tax может быть переписана так:

```
CREATE FUNCTION sales_tax(subtotal real, OUT tax real) AS $$
BEGIN
    tax := subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

Обратите внимание, что мы опустили `RETURNS real` — хотя можно было и включить, но это было бы излишним.

Выходные параметры наиболее полезны для возвращения нескольких значений. Простейший пример:

```
CREATE FUNCTION sum_n_product(x int, y int, OUT sum int, OUT prod int) AS $$
BEGIN
    sum := x + y;
    prod := x * y;
END;
$$ LANGUAGE plpgsql;
```

Как обсуждалось в Разд. 35.4.4, здесь фактически создается анонимный тип `record` для возвращения результата функции. Если используется фраза `RETURNS`, то она должна выглядеть как `RETURNS record`.

Есть еще способ объявить PL/pgSQL функцию с использованием RETURNS TABLE, например:

```
CREATE FUNCTION extended_sales(p_itemno int)
RETURNS TABLE(quantity int, total numeric) AS $$
BEGIN
    RETURN QUERY SELECT s.quantity, s.quantity * s.price FROM sales s
                 WHERE s.itemno = p_itemno;
END;
$$ LANGUAGE plpgsql;
```

Это в точности соответствует объявлению одного или нескольких OUT параметров и указанию RETURNS SETOF *sometype*.

Для PL/pgSQL функции, возвращающей полиморфный тип (anyelement, anyarray, anynonarray, anyenum, anyrange), создается специальный параметр $0. Его тип данных соответствует типу, фактически возвращаемому функцией, и который устанавливается на основании фактических типов входных параметров (смотри Разд. 35.2.5). Это позволяет функции получить доступ к фактически возвращаемому типу данных, как показано в Разд. 40.3.3. Параметр $0 инициализируется в NULL и его можно изменять внутри функции. Таким образом, его можно использовать для хранения возвращаемого значения, хотя это необязательно. Параметру $0 можно дать псевдоним. В следующем примере функция работает с любым типом данных, поддерживающим оператор +:

```
CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement)
RETURNS anyelement AS $$
DECLARE
    result ALIAS FOR $0;
BEGIN
    result := v1 + v2 + v3;
    RETURN result;
END;
$$ LANGUAGE plpgsql;
```

Такой же эффект получается при объявлении одного или нескольких выходных параметров полиморфного типа. При этом $0 не создается; выходные параметры сами используются для этой цели. Например:

```
CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement,
                                 OUT sum anyelement)
AS $$
BEGIN
    sum := v1 + v2 + v3;
END;
$$ LANGUAGE plpgsql;
```

40.3.2. Псевдонимы (ALIAS)

newname ALIAS FOR *oldname*;

Синтаксис ALIAS более общий, чем предполагалось в предыдущем разделе: псевдонимы

можно объявлять для любых переменных, а не только для параметров функции. Основная практическая польза в том, чтобы назначить другие имена переменным с предопределенными названиями, таким как NEW или OLD в триггерной процедуре.

Примеры:

```
DECLARE
  prior ALIAS FOR old;
  updated ALIAS FOR new;
```

Поскольку ALIAS дает два различных способа именования одних и тех же объектов, то его неограниченное использование может привести к путанице. Лучше всего использовать ALIAS для переименования предопределенных имен.

40.3.3. Наследование типов данных

variable%TYPE

Конструкция %TYPE предоставляет тип данных переменной или столбца таблицы. Её можно использовать для объявления переменных, содержащих значения из базы данных. Например, для объявления переменной с таким же типом, как и столбец user_id в таблице users нужно написать:

```
user_id users.user_id%TYPE;
```

Используя %TYPE не нужно знать тип данных структуры, на которую ссылаетесь. И самое главное, если в будущем тип данных изменится (например: тип данных для user_id поменяется с integer на real), то вам может не понадобиться изменять определение функции.

Использование %TYPE особенно полезно в полиморфных функциях, поскольку типы данных, необходимые для внутренних переменных, могут меняться от одного вызова к другому. Соответствующие переменные могут быть созданы с применением %TYPE к аргументам и возвращаемому значению функции.

40.3.4. Строковый тип

name table_name%ROWTYPE;
name composite_type_name;

Переменная составного типа называется строковой (*row*) переменной или переменной строкового типа (*row-type*). Значением такой переменной может быть целая строка, полученная в результате выполнения запроса SELECT или FOR, при условии, что набор столбцов запроса соответствует заявленному типу переменной. Доступ к отдельным значениям полей строковой переменной осуществляется, как обычно, через точку, например rowvar.field.

Строковая переменная может быть объявлена с таким же типом, как и строка в

существующей таблице или представлении, используя нотацию *table_name*%ROWTYPE; или при объявлении указывается имя составного типа. (Поскольку каждая таблица имеет соответствующий составной тип с таким же именем, то на самом деле, в PostgreSQL не имеет значения, пишете ли вы %ROWTYPE или нет. Но использование %ROWTYPE более переносимо.)

Параметры функции могут быть составного типа (строки таблицы). В этом случае соответствующий идентификатор $*n* будет строковой переменной, поля которой можно выбирать, например $1.user_id.

Только определенные пользователем столбцы таблицы доступны в переменной строкового типа, но не OID или другие системные столбцы (потому что это может быть строка представления). Поля строкового типа наследуют размер и точность от типов данных столбцов таблицы, таких как char(*n*).

Ниже приведен пример использования составных типов. table1 и table2 это существующие таблицы, имеющие, по меньшей мере, перечисленные столбцы:

```
CREATE FUNCTION merge_fields(t_row table1) RETURNS text AS $$
DECLARE
    t2_row table2%ROWTYPE;
BEGIN
    SELECT * INTO t2_row FROM table2 WHERE ... ;
    RETURN t_row.f1 || t2_row.f3 || t_row.f5 || t2_row.f7;
END;
$$ LANGUAGE plpgsql;

SELECT merge_fields(t.*) FROM table1 t WHERE ... ;
```

40.3.5. Тип record

name RECORD;

Переменные типа record похожи на переменные строкового типа, но они не имеют предопределенной структуры. Они приобретают фактическую структуру от строки, которая им присваивается командами SELECT или FOR. Структура переменной типа record может меняться каждый раз при присвоении значения. Следствием этого является то, что пока значение не присвоено первый раз, переменная типа record не имеет структуры и любая попытка получить доступ к отдельному полю приведет к ошибке во время исполнения.

Обратите внимание, что RECORD это не подлинный тип данных, а только лишь заполнитель. Также следует понимать, что PL/pgSQL функция, имеющая тип возвращаемого значения record, это не то же самое, что и переменная типа record, хотя такая функция может использовать переменную типа record для хранения своего результата. В обоих случаях фактическая структура строки неизвестна во время создания функции, но для функции, возвращающей record, фактическая структура определяется во время разбора вызывающего запроса, в то время как переменная типа record может менять свою

структуру на лету.

40.3.6. Упорядочение PL/pgSQL переменных

При каждом вызове PL/pgSQL функции определяется схема упорядочения, которая зависит от соответствующих схем каждого фактического аргумента, как описано в Разд. 22.2. Если схема упорядочения успешно определена (т.е. среди аргументов нет конфликтов между неявными схемами упорядочения), то все соответствующие параметры неявно трактуются как имеющие эту схему упорядочения. Внутри функции это будет влиять на поведение операторов, зависящих от используемой схемы упорядочения. Рассмотрим пример:

```
CREATE FUNCTION less_than(a text, b text) RETURNS boolean AS $$
BEGIN
    RETURN a < b;
END;
$$ LANGUAGE plpgsql;

SELECT less_than(text_field_1, text_field_2) FROM table1;
SELECT less_than(text_field_1, text_field_2 COLLATE "C") FROM table1;
```

В первом случае `less_than` будет использовать для сравнения общую схему упорядочения для `text_field_1` и `text_field_2`, в то время как во втором случае будет использоваться схема `C`.

Кроме того, определенная для вызова функции схема упорядочения также будет использоваться для любых локальных переменных соответствующего типа. Таким образом, функция не станет работать по-другому, если её переписать так:

```
CREATE FUNCTION less_than(a text, b text) RETURNS boolean AS $$
DECLARE
    local_a text := a;
    local_b text := b;
BEGIN
    RETURN local_a < local_b;
END;
$$ LANGUAGE plpgsql;
```

Если нет параметров с типами данных, требующих схемы упорядочения, или для параметров невозможно определить общую схему, тогда параметры и локальные переменные используют схемы по умолчанию их типа данных (которые обычно совпадают со схемой по умолчанию базы данных, но могут отличаться для переменных доменных типов).

Локальная переменная может иметь схему упорядочения отличную от схемы по умолчанию. Для этого используется опция `COLLATE` в объявлении переменной, например:

```
DECLARE
    local_a text COLLATE "en_US";
```

Эта опция переопределяет схему упорядочения, которую получила бы переменная в соответствии с вышеуказанными правилами.

И, конечно же, можно явно указывать опцию `COLLATE` для конкретных операций внутри

функции, если к ним требуется применить конкретную схему упорядочения. Например:

```
CREATE FUNCTION less_than_c(a text, b text) RETURNS boolean AS $$
BEGIN
    RETURN a < b COLLATE "C";
END;
$$ LANGUAGE plpgsql;
```

Как и в обычной SQL команде, это переопределяет схемы упорядочения, связанные с полями таблицы, параметрами и локальными переменными, которые используются в данном выражении.

40.4. Выражения

Все выражения, используемые в PL/pgSQL операторах, обрабатываются основной SQL машиной сервера. Например, для вычисления такого выражения:

```
IF expression THEN ...
```

PL/pgSQL отправит следующий запрос SQL машине:

```
SELECT expression
```

При формировании команды SELECT все вхождения имен PL/pgSQL переменных заменяются параметрами, как подробно описано в Разд. 40.10.1. Это позволяет один раз подготовить план выполнения команды SELECT и повторно использовать его в последующих вычислениях с различными значениями переменных. Таким образом, при первом использовании выражения, по сути происходит выполнение команды PREPARE. Например, если мы объявили две целочисленные переменные x и y, и написали:

```
IF x < y THEN ...
```

то, что реально происходит за сценой, эквивалентно:

```
PREPARE statement_name(integer, integer) AS SELECT $1 < $2;
```

и затем, эта подготовленная команда исполняется (EXECUTE) для каждого оператора IF с текущими значениями PL/pgSQL переменных, переданных как значения параметров. Обычно эти детали не важны для пользователей PL/pgSQL, но их полезно знать при диагностировании проблем. Более подробная информация появится в Разд. 40.10.2.

40.5. Основные операторы

В этом и последующих разделах описаны все типы операторов, которые понимает PL/pgSQL. Все, что не признается в качестве одного из этих типов операторов, считается командой SQL и отправляется для исполнения в основную машину базы данных, как описано в Разд. 40.5.2 и Разд. 40.5.3.

40.5.1. Присваивания

Присвоение значения PL/pgSQL переменной записывается в виде:

```
variable { := | = } expression;
```

Как описывалось ранее, выражение в таком операторе вычисляется с помощью SQL команды SELECT, посылаемой в основную машину базы данных. Выражение должно получить одно значение (возможно, значение строки, если переменная строкового типа или типа record). Целевая переменная может быть простой переменной (опционально квалифицированной именем блока), полем в переменной строкового типа или записи; или элементом массива, который является простой переменной или полем. Для присвоения можно использовать знак равенства (=) вместо совместимого с PL/SQL :=.

Если результирующий тип данных выражения не соответствует типу данных переменной, или переменная имеет определенный размер/точность (как char(20)), то результирующее значение будет неявно преобразовано интерпретатором PL/pgSQL, используя функцию вывода типа данных результата и функцию ввода типа данных переменной. Заметим, что это потенциально может привести к ошибкам времени выполнения в функции ввода, если формат строки результирующего значения не допустим для функции ввода.

Примеры:

```
tax := subtotal * 0.06;
my_record.user_id := 20;
```

40.5.2. Выполнение команды, не возвращающей результат

В PL/pgSQL функции можно выполнить любую команду SQL, не возвращающую строк, просто написав эту команду (например INSERT без фразы RETURNING).

Имя любой PL/pgSQL переменной в тексте команды рассматривается как параметр, а затем текущее значение переменной подставляется в качестве значения параметра во время выполнения. Это в точности совпадает с описанной ранее обработкой для выражений; подробнее смотри Разд. 40.10.1.

При выполнении SQL команды таким образом, PL/pgSQL может кэшировать и повторно использовать план выполнения команды, как обсуждается в Разд. 40.10.2.

Иногда бывает полезно вычислить значение выражения или запроса SELECT, но отказаться от результата, например, при вызове функции, у которой есть побочные эффекты, но нет полезного результата. Для этого в PL/pgSQL, используется оператор PERFORM:

```
PERFORM query;
```

Эта команда выполняет *query* и отбрасывает результат. Запросы (*query*) пишутся таким же образом, как и в команде SQL SELECT, но ключевое слово SELECT заменяется на PERFORM. Для запросов WITH после PERFORM нужно поместить запрос в скобки. (В этом случае запрос

может вернуть только одну строку.) PL/pgSQL переменные будут подставлены в запрос так же, как и в команду, не возвращающую результат, план запроса также кэшируется. Кроме того, специальная переменная FOUND устанавливается в истину, если запрос возвращает, по крайней мере, одну строку, или ложь, если не возвращает ни одной строки (смотри Разд. 40.5.5).

> **Замечание:** Можно предположить, что такой же результат получается непосредственно командой SELECT, но в настоящее время использование PERFORM является единственным способом. Команда SQL, которая может возвращать строки, например SELECT, будет отклонена с ошибкой, если не имеет фразы INTO, как описано в следующем разделе.

Пример:

```
PERFORM create_mv('cs_session_page_requests_mv', my_query);
```

40.5.3. Выполнение запроса, возвращающего одну строку

Результат SQL команды, возвращающей одну строку (возможно из нескольких столбцов), может быть присвоен переменной типа record, переменной строкового типа или списку скалярных переменных. Для этого нужно к основной команде SQL добавить фразу INTO. Так, например:

```
SELECT select_expressions INTO [STRICT] target FROM ...;
INSERT ... RETURNING expressions INTO [STRICT] target;
UPDATE ... RETURNING expressions INTO [STRICT] target;
DELETE ... RETURNING expressions INTO [STRICT] target;
```

где *target* может быть переменной типа record, строковой переменной или разделенным через запятую списком скалярных переменных, полей записи/строки. PL/pgSQL переменные подставляются в оставшуюся часть запроса, план выполнения кэшируется, так же, как было описано выше для команд, не возвращающих строки. Это работает для команд SELECT, INSERT/UPDATE/DELETE с фразой RETURNING и утилит, возвращающих результат в виде набора строк (таких, как EXPLAIN). За исключением фразы INTO, это те же SQL команды, как их можно написать вне PL/pgSQL.

> **Подсказка:** Обратите внимание, что данная интерпретация SELECT с INTO полностью отличается от PostgreSQL команды SELECT INTO, где в INTO указывается вновь создаваемая таблица. Если вы хотите в PL/pgSQL функции создать таблицу, основанную на результате команды SELECT, используйте синтаксис CREATE TABLE ... AS SELECT.

Если результат запроса присваивается переменной строкового типа или списку переменных, то они должны в точности соответствовать по количеству и типам данных столбцам результата, иначе произойдет ошибка во время выполнения. Если используется переменная типа record, то она автоматически приводится к строковому типу результата запроса.

Фраза INTO может появиться практически в любом месте SQL команды. Обычно её записывают непосредственно перед или сразу после списка *select_expressions* в SELECT или в конце команды для команд других типов. Рекомендуется следовать этому соглашению на случай, если парсер PL/pgSQL станет более строгим в будущих версиях.

Если опция STRICT не указана во фразе INTO, то в *target* присваивается первая строка возвращенная запросом; или NULL, если запрос не вернул строк. (Заметим, что понятие "первая строка" не четко определено без использования ORDER BY.) Все остальные строки результата после первой отбрасываются. Можно проверить специальную переменную FOUND (смотри Разд. 40.5.5), чтобы определить была ли возвращена запись:

```
SELECT * INTO myrec FROM emp WHERE empname = myname;
IF NOT FOUND THEN
    RAISE EXCEPTION 'Сотрудник % не найден', myname;
END IF;
```

Если опция STRICT указана, то запрос должен вернуть ровно одну строку или произойдет ошибка во время выполнения: либо NO_DATA_FOUND (нет строк), либо TOO_MANY_ROWS (более одной строки). Можно использовать секцию исключений в блоке для обработки ошибок, например:

```
BEGIN
    SELECT * INTO STRICT myrec FROM emp WHERE empname = myname;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            RAISE EXCEPTION 'Сотрудник % не найден', myname;
        WHEN TOO_MANY_ROWS THEN
            RAISE EXCEPTION 'Сотрудник % уже существует', myname;
END;
```

После успешного выполнения команды с опцией STRICT, значение переменной FOUND всегда устанавливается в истину.

Для INSERT/UPDATE/DELETE с RETURNING, PL/pgSQL возвращает ошибку, если выбрано более одной строки, даже в том случае, когда опция STRICT не указана. Так происходит потому, что у этих команд нет возможности, типа ORDER BY, указать какая из задействованных строк должна быть возвращена.

Если для функции доступна опция print_strict_params, то при возникновении ошибки, связанной с нарушением условия STRICT, в детальную (DETAIL) часть сообщения об ошибке будет включена информация о параметрах переданных запросу. Изменить значение print_strict_params можно установкой параметра plpgsql.print_strict_params. Но это повлияет только функции скомпилированные после изменения. Для конкретной функции можно использовать опцию компилятора, например:

```
CREATE FUNCTION get_userid(username text) RETURNS int
AS $$
#print_strict_params on
DECLARE
```

```
    userid int;
BEGIN
    SELECT users.userid INTO STRICT userid
        FROM users WHERE users.username = get_userid.username;
    RETURN userid;
END
$$ LANGUAGE plpgsql;
```

В случае сбоя, будет сформировано примерно такое сообщение об ошибке

```
ERROR:  query returned no rows
DETAIL:  parameters: $1 = 'nosuchuser'
CONTEXT:  PL/pgSQL function get_userid(text) line 6 at SQL statement
```

> **Замечание:** Опция `STRICT` совпадает с поведением `SELECT INTO` и соответствующих операторов в Oracle PL/SQL.

Как действовать в случаях, когда требуется обработать несколько строк результата смотри в Разд. 40.6.4.

40.5.4. Выполнение динамически формируемых команд

Часто требуется динамически формировать команды внутри PL/pgSQL функций, то есть такие команды, в которых при каждом выполнении могут использоваться разные таблицы или типы данных. Обычная практика PL/pgSQL кэшировать планы выполнения (как описано в Разд. 40.10.2), в случае с динамическими командами, работать не будет. Для исполнения динамических команд предусмотрен оператор `EXECUTE`:

```
EXECUTE command-string [ INTO [STRICT] target ] [ USING expression [, ... ] ];
```

где *`command-string`* это выражение, формирующее строку (типа `text`) с текстом команды, которую нужно выполнить. Необязательный *`target`*, куда присваивается результат команды, может быть переменной типа `record`, строковой переменной или разделенным через запятую списком скалярных переменных, полей записи/строки. Необязательные выражения во фразе `USING` формируют значения, которые будут вставлены в команду.

В сформированном тексте команды замена имен переменных PL/pgSQL на их значения проводиться не будет. Все необходимые значения переменных должны быть вставлены в командную строку при её построении, либо нужно использовать параметры, как описано ниже.

Также, нет никакого плана кэширования для команд, выполняемых с помощью `EXECUTE`. Вместо этого план создается каждый раз при выполнении. Таким образом, строка команды может динамически создаваться внутри функции для выполнения действий с различными таблицами и столбцами.

Фраза `INTO` указывает, куда должны быть присвоены результаты SQL команды, возвращающей строки. Если используется переменная строкового типа или список переменных, то они должны в точности соответствовать структуре результата запроса (когда

используется переменная типа record, она автоматически приводится к строковому типу результата запроса). Если возвращается несколько строк, то только первая будет присвоена переменной(-ым) в INTO. Если не возвращается ни одной строки, то присваивается NULL. При отсутствии фразы INTO результаты запроса отбрасываются.

С опцией STRICT запрос должен вернуть ровно одну строку, иначе выдается сообщение об ошибке.

В тексте команды можно использовать значения параметров, ссылки на параметры обозначаются как $1, $2 и т.д. Эти символы указывают на значения, находящиеся во фразе USING. Такой метод зачастую предпочтительнее, чем вставка значений в команду в виде текста: это позволяет избегать во время исполнения дополнительных расходов на преобразования значений в текст и обратно, и менее подвержено атакам при помощи SQL-инъекций, так как не требуется заключать текст в кавычки и экранировать символы. Пример:

```
EXECUTE 'SELECT count(*) FROM mytable WHERE inserted_by = $1 AND inserted <= $2'
   INTO c
   USING checked_user, checked_date;
```

Обратите внимание, что символы параметров могут быть использованы только в местах, где должны быть значения. Если требуется динамически формировать имена таблиц или столбцов, то их необходимо вставлять в виде текста. Например, если в предыдущем запросе необходимо динамически задавать имя таблицы, можно сделать следующее:

```
EXECUTE 'SELECT count(*) FROM '
    || tabname::regclass
    || ' WHERE inserted_by = $1 AND inserted <= $2'
   INTO c
   USING checked_user, checked_date;
```

Еще одно ограничение состоит в том, что символы параметров могут использоваться только в командах SELECT, INSERT, UPDATE и DELETE. В других типах операторов (обычно называемых утилитами), необходимо вставлять значения в виде текста даже там, где требуются просто значения.

Команда EXECUTE с неизменяемым текстом и параметрами во фразе USING (как в первом примере выше), функционально эквивалентна команде записанной напрямую в PL/pgSQL, в которой переменные PL/pgSQL автоматически заменяются значениями. Важное отличие в том, что EXECUTE при каждом исполнении заново строит план команды с учетом текущих значений параметров, тогда как PL/pgSQL строит общий план выполнения и кэширует его при повторном использовании. В тех случаях, когда наилучший план выполнения сильно зависит от значений параметров, может быть полезно использовать EXECUTE для гарантии того, что не будет выбран общий план.

В настоящее время команда SELECT INTO не поддерживается в EXECUTE, вместо этого нужно выполнять обычный SELECT и использовать фразу INTO самой команды EXECUTE.

Замечание: Оператор EXECUTE в PL/pgSQL не имеет отношения к одноименному

SQL оператору сервера PostgreSQL. Серверный EXECUTE не может напрямую использоваться в PL/pgSQL функциях (и в этом нет необходимости).

Пример 40-1. Использование кавычек в динамических запросах

При работе с динамическими командами часто приходится иметь дело с экранированием одинарных кавычек. Рекомендуемым методом для взятия текста в кавычки в теле функции является экранирование знаками доллара. (Если имеется унаследованный код, не использующий этот метод, пожалуйста, обратитесь к обзору в Разд. 40.11.1, это поможет сэкономить усилия при переводе кода к более приемлемому виду.)

Необходимо соблюдать осторожность при вставке динамических значений в конструируемый текст запроса, так как они могут сами содержать кавычки. Пример (предполагается, что тело функции экранируется знаками доллара, поэтому кавычки не нужно дублировать):

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(colname)
        || ' = '
        || quote_literal(newvalue)
        || ' WHERE key = '
        || quote_literal(keyvalue);
```

Этот пример демонстрирует использование функций quote_ident и quote_literal (смотри Разд. 9.4). Для надежности, выражения, содержащие идентификаторы столбцов и таблиц должны использовать функцию quote_ident при добавлении в текст запроса. А для выражений со значениями, которые должны быть обычными строками, используется функция quote_literal. Эти функции выполняют соответствующие шаги, чтобы вернуть текст, по ситуации заключенный в двойные или одинарные кавычки и с правильно экранированными специальными символами.

Так как функция quote_literal помечена как STRICT, то она всегда возвращает NULL, если переданный ей аргумент имеет значение NULL. В приведенном выше примере, если newvalue или keyvalue были NULL, вся строка с текстом запроса станет NULL, что приведет к ошибке в EXECUTE. Для избегания этой проблемы используйте функцию quote_nullable, которая работает так же, как quote_literal за исключением того, что при вызове с пустым аргументом возвращает строку 'NULL'. Например:

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(colname)
        || ' = '
        || quote_nullable(newvalue)
        || ' WHERE key = '
        || quote_nullable(keyvalue);
```

Если вы имеете дело со значениями, которые могут быть пустыми, то, как правило, нужно использовать quote_nullable вместо quote_literal.

Как обычно, необходимо убедиться, что пустые значения в запросе не принесут неожиданных результатов. Например, следующая фраза WHERE

```
'WHERE key = ' || quote_nullable(keyvalue)
```

никогда не вернет истину, если keyvalue - NULL, так как применение = с операндом, имеющим значение NULL, всегда дает NULL. Если требуется, чтобы NULL обрабатывалось как обычное значение, то фразу выше нужно переписать так:

```
'WHERE key IS NOT DISTINCT FROM ' || quote_nullable(keyvalue)
```

(В настоящее время IS NOT DISTINCT FROM работает менее эффективно чем =, так что используйте этот способ, если это действительно необходимо. Смотри Разд. 9.2 для более подробной информации о работе с NULL и IS DISTINCT.)

Обратите внимание, что использование знака $ полезно только для взятия в кавычки фиксированного текста. Плохая идея написать этот пример так:

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(colname)
        || ' = $$'
        || newvalue
        || '$$ WHERE key = '
        || quote_literal(keyvalue);
```

потому что newvalue может также содержать $$. Эта же проблема может возникнуть и с любым другим разделителем, используемым после знака $. Поэтому, чтобы безопасно заключить заранее неизвестный текст в кавычки, *нужно* использовать соответствующие функции: quote_literal, quote_nullable, или quote_ident.

Динамические операторы SQL также могут быть безопасно сконструированы при помощи функции format (смотри Разд. 9.4).Так, например:

```
EXECUTE format('UPDATE tbl SET %I = %L WHERE key = %L', colname, newvalue,
keyvalue);
```

Функцию format можно использовать в сочетании с фразой USING:

```
EXECUTE format('UPDATE tbl SET %I = $1 WHERE key = $2', colname)
   USING newvalue, keyvalue;
```

Это более эффективная форма, так как параметры newvalue и keyvalue не преобразуются в текст.

Более объемный пример использования динамической команды и EXECUTE можно увидеть в Прим. 40-9. В нем создается и динамически выполняется команда CREATE FUNCTION для определения новой функции.

40.5.5. Статус выполнения команды

Существует несколько способов определить статус выполнения команды. Первый способ заключается в использовании команды GET DIAGNOSTICS, которая имеет следующий вид:

```
GET [ CURRENT ] DIAGNOSTICS variable { = | := } item [ , ... ];
```

Эта команда позволяет получить значения индикаторов состояния системы. Каждый *item* является ключевым словом, идентифицирующим значение состояния, которое будет присвоено указанной переменной. Переменная должна быть соответствующего типа данных. В настоящий момент доступны следующие индикаторы: ROW_COUNT - количество строк, обработанных последней командой SQL; RESULT_OID - OID последней строки, вставленной последней выполненной командой SQL. Обратите внимание, что получать RESULT_OID имеет смысл только после вставки (INSERT) записей в таблицу, содержащую OID. Для GET DIAGNOSTICS можно использовать двоеточие-равно (:=) вместо = в стандарте SQL.

Пример:

```
GET DIAGNOSTICS integer_var = ROW_COUNT;
```

Второй способ определения статуса выполнения команды заключается в проверке значения специальной переменной FOUND, имеющей тип boolean. При вызове PL/pgSQL функции, переменная FOUND инициализируется в ложь. Далее, значение переменной изменяется следующими операторами:

- SELECT INTO устанавливает FOUND в истину, если строка присвоена, или в ложь, если строки не выбраны.
- PERFORM устанавливает FOUND в истину если строки выбраны (затем они отбрасываются), или в ложь, если строки не выбраны.
- UPDATE, INSERT и DELETE устанавливают FOUND в истину, если при их выполнении была задействована хотя бы одна строка, или в ложь, если ни одна строка не была задействована.
- FETCH устанавливают FOUND в истину, если команда вернула строку, или ложь, если строка не выбрана.
- MOVE устанавливает FOUND в истину при успешном перемещении курсора, в противном случае - в ложь.
- FOR, как и FOREACH, устанавливают FOUND в истину, если была произведена хотя бы одна итерация цикла, в противном случае - в ложь. При этом значение FOUND будет установлено только после выхода из цикла. Пока цикл выполняется, оператор цикла не изменяет значение переменной. Но другие операторы внутри цикла могут менять значение FOUND.
- RETURN QUERY и RETURN QUERY EXECUTE устанавливают FOUND в истину, если запрос вернул хотя бы одну строку, или в ложь, если строки не выбраны.

Другие операторы PL/pgSQL не меняют значение FOUND. Помните в частности, что EXECUTE изменяет вывод GET DIAGNOSTICS, но не меняет FOUND.

FOUND является локальной переменной в каждой функции PL/pgSQL и любые её изменения,

влияют только на текущую функцию.

40.5.6. Не делать ничего

Иногда бывает полезен оператор, который не делает ничего. Например, он может показывать, что одна из ветвей if/then/else сознательно оставлена пустой. Для этих целей используется `NULL`:

```
NULL;
```

В следующем примере два фрагмента кода эквивалентны:

```
BEGIN
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN
        NULL;  -- игнорируем ошибку
END;

BEGIN
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN  -- игнорируем ошибку
END;
```

Какой вариант выбрать - дело вкуса.

> **Замечание:** В Oracle PL/SQL оператор не может отсутствовать, поэтому `NULL` *обязателен* в подобных ситуациях. В PL/pgSQL разрешается не писать ничего.

40.6. Управляющие структуры

Управляющие структуры, вероятно, наиболее полезная и важная часть PL/pgSQL. С их помощью можно очень гибко и эффективно манипулировать данными PostgreSQL.

40.6.1. Команды для возврата значения из функции

Две команды позволяют вернуть данные из функции: `RETURN` и `RETURN NEXT`.

40.6.1.1. RETURN

```
RETURN expression;
```

`RETURN` с последующим выражением прекращает выполнение функции и возвращает значение выражения в вызывающую программу. Эта форма используется для функций PL/pgSQL, которые не возвращают набор строк.

В функции, возвращающей скалярный тип, результирующее выражение автоматически приводится к типу возвращаемого значения. Однако, чтобы вернуть составной тип (строку), возвращаемое выражение должно в точности содержать требуемый набор столбцов. При этом

может потребоваться явное приведение типов.

Для функции с выходными параметрами просто используйте RETURN без выражения. Будут возвращены текущие значения выходных параметров.

Для функции, возвращающей void, RETURN можно использовать в любом месте, но без выражения после RETURN.

Возвращаемое значение функции не может остаться не определенным. Если достигнут конец блока верхнего уровня, а оператор RETURN так и не встретился, происходит ошибка времени исполнения. Это не касается функций с выходными параметрами и функций, возвращающих void. Для них оператор RETURN выполняется автоматически по окончании блока верхнего уровня.

Несколько примеров:

```
-- Функции, возвращающие скалярный тип данных
RETURN 1 + 2;
RETURN scalar_var;

-- Функции, возвращающие составной тип данных
RETURN composite_type_var;
RETURN (1, 2, 'three'::text);  -- требуется приведение типов
```

40.6.1.2. RETURN NEXT и RETURN QUERY

```
RETURN NEXT expression;
RETURN QUERY query;
RETURN QUERY EXECUTE command-string [ USING expression [, ... ] ];
```

Для PL/pgSQL функций, возвращающих SETOF *sometype*, нужно действовать несколько по-иному. Отдельные элементы возвращаемого значения формируются командами RETURN NEXT или RETURN QUERY, а финальная команда RETURN без аргументов завершает выполнение функции. RETURN NEXT используется как со скалярными, так и с составными типами данных. Для составного типа результат функции возвращается в виде таблицы. RETURN QUERY добавляет результат выполнения запроса к результату функции. RETURN NEXT и RETURN QUERY можно свободно смешивать в теле функции, в этом случае их результаты будут объединены.

RETURN NEXT и RETURN QUERY не выполняют возврат из функции. Они просто добавляют строки в результирующее множество. Затем выполнение продолжается со следующего оператора в функции. Успешное выполнение RETURN NEXT и RETURN QUERY формирует множество строк результата. Для выхода из функции используется RETURN, обязательно без аргументов (или можно просто дождаться окончания выполнения функции).

RETURN QUERY имеет разновидность RETURN QUERY EXECUTE, предназначенную для динамического выполнения запроса. В тексте запроса можно использовать параметры, используя фразу USING, также как и в команде EXECUTE.

Для функции с выходными параметрами просто используйте RETURN NEXT без аргументов. При каждом исполнении RETURN NEXT текущие значения выходных параметров сохраняются для последующего возврата в качестве строки результата. Обратите внимание, что если функция с выходными параметрами должна возвращать множество значений, то при объявлении нужно указывать RETURNS SETOF. При этом если выходных параметров несколько, то используется RETURNS SETOF record, а если только один с типом *sometype*, то RETURNS SETOF *sometype*.

Пример использования RETURN NEXT:

```
CREATE TABLE foo (fooid INT, foosubid INT, fooname TEXT);
INSERT INTO foo VALUES (1, 2, 'three');
INSERT INTO foo VALUES (4, 5, 'six');

CREATE OR REPLACE FUNCTION get_all_foo() RETURNS SETOF foo AS
$BODY$
DECLARE
    r foo%rowtype;
BEGIN
    FOR r IN
        SELECT * FROM foo WHERE fooid > 0
    LOOP
        -- здесь возможна обработка данных
        RETURN NEXT r; -- добавляет текущую строку запроса к возвращаемому
результату
    END LOOP;
    RETURN;
END
$BODY$
LANGUAGE plpgsql;

SELECT * FROM get_all_foo();
```

Пример использования RETURN QUERY:

```
CREATE FUNCTION get_available_flightid(date) RETURNS SETOF integer AS
$BODY$
BEGIN
    RETURN QUERY SELECT flightid
                   FROM flight
                  WHERE flightdate >= $1
                    AND flightdate < ($1 + 1);

    -- Т.к. выполнение еще не закончено, можно проверить были ли возвращены
строки
    -- Если нет, то вызываем исключение
    IF NOT FOUND THEN
        RAISE EXCEPTION 'Нет рейсов на дату: %.', $1;
    END IF;

    RETURN;
 END
$BODY$
LANGUAGE plpgsql;

-- Возвращает доступные рейсы, либо вызывает исключение
SELECT * FROM get_available_flightid(CURRENT_DATE);
```

Замечание: В текущей реализации RETURN NEXT и RETURN QUERY результирующее множество накапливается целиком, прежде чем будет возвращено из функции. Если множество очень большое, то это может отрицательно сказаться на производительности, т.к. при нехватке оперативной памяти данные записываются на диск. В следующих версиях PL/pgSQL это ограничение будет снято. В настоящее время управлять количеством оперативной памяти в подобных случаях можно параметром конфигурации work_mem. При наличии свободной памяти администраторы должны рассмотреть возможность увеличения значения данного параметра.

40.6.2. Условные операторы

Операторы IF и CASE позволяют выполнять команды в зависимости от определенных условий. PL/pgSQL поддерживает три формы IF:

- IF ... THEN
- IF ... THEN ... ELSE
- IF ... THEN ... ELSIF ... THEN ... ELSE

и две формы CASE:

- CASE ... WHEN ... THEN ... ELSE ... END CASE
- CASE WHEN ... THEN ... ELSE ... END CASE

40.6.2.1. IF-THEN

```
IF boolean-expression THEN
    statements
END IF;
```

IF-THEN это простейшая форма IF. Операторы между THEN и END IF выполняются, если условие (*boolean-expression*) истинно. В противном случае они опускаются.

Пример:

```
IF v_user_id <> 0 THEN
    UPDATE users SET email = v_email WHERE user_id = v_user_id;
END IF;
```

40.6.2.2. IF-THEN-ELSE

```
IF boolean-expression THEN
    statements
ELSE
    statements
END IF;
```

IF-THEN-ELSE добавляет к IF-THEN возможность указать альтернативный набор операторов, которые будут выполнены, если условие не истинно (в том числе, если условие

NULL).

Примеры:

```
IF parentid IS NULL OR parentid = ''
THEN
    RETURN fullname;
ELSE
    RETURN hp_true_filename(parentid) || '/' || fullname;
END IF;
```

```
IF v_count > 0 THEN
    INSERT INTO users_count (count) VALUES (v_count);
    RETURN 't';
ELSE
    RETURN 'f';
END IF;
```

40.6.2.3. IF-THEN-ELSIF

```
IF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
    ...]]
[ ELSE
    statements ]
END IF;
```

В некоторых случаях двух альтернатив недостаточно. IF-THEN-ELSIF обеспечивает удобный способ проверки нескольких вариантов по очереди. Условия в IF последовательно проверяются до тех пор, пока не будет найдено первое истинное. После этого операторы, относящиеся к этому условию, выполняются, и управление переходит к следующей после END IF команде. (Все последующие условия не проверяются.) Если ни одно из условий IF не является истинным, то выполняется блок ELSE (если присутствует).

Пример:

```
IF number = 0 THEN
    result := 'zero';
ELSIF number > 0 THEN
    result := 'positive';
ELSIF number < 0 THEN
    result := 'negative';
ELSE
    -- раз мы здесь, значит значение number не определено (NULL)
    result := 'NULL';
END IF;
```

Вместо ключевого слова ELSIF можно использовать ELSEIF.

Другой вариант сделать то же самое, это использование вложенных операторов IF-THEN-ELSE, как в следующем примере:

```
IF demo_row.sex = 'm' THEN
    pretty_sex := 'man';
ELSE
    IF demo_row.sex = 'f' THEN
        pretty_sex := 'woman';
    END IF;
END IF;
```

Однако это требует написания соответствующих `END IF` для каждого `IF`, что при наличии нескольких альтернатив делает код более громоздким, чем использование `ELSIF`.

40.6.2.4. Простой CASE

```
CASE search-expression
    WHEN expression [, expression [ ... ]] THEN
      statements
  [ WHEN expression [, expression [ ... ]] THEN
      statements
    ... ]
  [ ELSE
      statements ]
END CASE;
```

Простая форма `CASE` реализует условное выполнение на основе сравнения операндов. *`search-expression`* вычисляется (один раз) и последовательно сравнивается с каждым *`expression`* во фразах `WHEN`. Если совпадение найдено, то выполняются соответствующие *`statements`* и управление переходит к следующей после `END CASE` команде. (Все последующие выражения `WHEN` не проверяются.) Если совпадение не было найдено, то выполняются `ELSE` *`statements`*. Но если `ELSE` нет, то вызывается исключение `CASE_NOT_FOUND`.

Пример:

```
CASE x
    WHEN 1, 2 THEN
        msg := 'один или два';
    ELSE
        msg := 'значение, отличное от один или два';
END CASE;
```

40.6.2.5. CASE с перебором условий

```
CASE
    WHEN boolean-expression THEN
      statements
  [ WHEN boolean-expression THEN
      statements
    ... ]
  [ ELSE
      statements ]
END CASE;
```

Эта форма `CASE` реализует условное выполнение, основываясь на истинности логических условий. Каждое выражение *`boolean-expression`* во фразе `WHEN` вычисляется по

порядку до тех пор, пока не будет найдено истинное. Затем выполняются соответствующие *statements* и управление переходит к следующей после END CASE команде. (Все последующие выражения WHEN не проверяются.) Если ни одно из условий не окажется истинным, то выполняются ELSE *statements*. Но если ELSE нет, то вызывается исключение CASE_NOT_FOUND.

Пример:

```
CASE
    WHEN x BETWEEN 0 AND 10 THEN
        msg := 'значение в диапазоне между 0 и 10';
    WHEN x BETWEEN 11 AND 20 THEN
        msg := 'значение в диапазоне между 11 и 20';
END CASE;
```

Эта форма CASE полностью эквивалента IF-THEN-ELSIF, за исключением того, что при невыполнении всех условий и отсутствии ELSE, IF-THEN-ELSIF ничего не делает, а CASE вызывает ошибку.

40.6.3. Простые циклы

Операторы LOOP, EXIT, CONTINUE, WHILE, FOR и FOREACH позволяют повторить серию команд в PL/pgSQL функции.

40.6.3.1. LOOP

```
[ <<label>> ]
LOOP
    statements
END LOOP [ label ];
```

LOOP организует безусловный цикл, который повторяется до бесконечности, пока не будет прекращен операторами EXIT или RETURN. Для вложенных циклов можно использовать *label* в операторах EXIT и CONTINUE, чтобы указать к какому циклу эти операторы относятся.

40.6.3.2. EXIT

```
EXIT [ label ] [ WHEN boolean-expression ];
```

Если *label* не указана, то завершается самый внутренний цикл, далее выполняется оператор, следующий за END LOOP. Если *label* указана, то она должна относиться к текущему или внешнему циклу, или это может быть метка блока. При этом в именованном цикле/блоке выполнение прекращается, а управление переходит к следующему оператору после соответствующего END.

При наличии WHEN цикл прекращается, только если *boolean-expression* истинно. В противном случае управление переходит к оператору следующему за EXIT.

EXIT можно использовать со всеми типами циклов, не только с безусловным.

Когда EXIT используется для выхода из блока, управление переходит к следующему оператору после окончания блока. Обратите внимание, что для выхода из блока нужно обязательно указывать *label*. EXIT без *label* не позволяет прекратить работу блока. (Это изменение по сравнению с версиями PostgreSQL до 8.4, в которых разрешалось использовать EXIT без *label* для прекращения работы текущего блока.)

Примеры:

```
LOOP
    -- здесь вычисления
    IF count > 0 THEN
        EXIT;  -- выход из цикла
    END IF;
END LOOP;

LOOP
    -- здесь вычисления
    EXIT WHEN count > 0;  -- аналогично предыдущему примеру
END LOOP;

<<ablock>>
BEGIN
    -- здесь вычисления
    IF stocks > 100000 THEN
        EXIT ablock;  -- выход из блока ablock
    END IF;
    -- вычисления не будут выполнены, если stocks > 100000
END;
```

40.6.3.3. CONTINUE

```
CONTINUE [ label ] [ WHEN boolean-expression ];
```

Если *label* не указана, то начинается следующая итерация самого внутреннего цикла. То есть все оставшиеся в цикле операторы пропускаются, и управление переходит к управляющему выражению цикла (если есть) для определения, нужна ли еще одна итерация цикла. Если *label* присутствует, то она указывает на метку цикла, выполнение которого будет продолжено.

При наличии WHEN следующая итерация цикла начинается только тогда, когда *boolean-expression* истинно. В противном случае управление переходит к оператору, следующему за CONTINUE.

CONTINUE можно использовать со всеми типами циклов, не только с безусловным.

Пример:

```
LOOP
    -- здесь вычисления
    EXIT WHEN count > 100;
    CONTINUE WHEN count < 50;
    -- Вычисления для count в диапазоне 50 .. 100
```

```
END LOOP;
```

40.6.3.4. WHILE

```
[ <<label>> ]
WHILE boolean-expression LOOP
    statements
END LOOP [ label ];
```

WHILE выполняет серию команд до тех пор, пока истинно выражение *boolean-expression*. Выражение проверяется непосредственно перед каждым входом в тело цикла.

Пример:

```
WHILE amount_owed > 0 AND gift_certificate_balance > 0 LOOP
    -- здесь вычисления
END LOOP;

WHILE NOT done LOOP
    -- здесь вычисления
END LOOP;
```

40.6.3.5. FOR (целочисленный вариант)

```
[ <<label>> ]
FOR name IN [ REVERSE ] expression .. expression [ BY expression ] LOOP
    statements
END LOOP [ label ];
```

В этой форме цикла FOR итерации выполняются по диапазону целых чисел. Переменная *name* автоматически определяется с типом integer и существует только внутри цикла (если уже существует переменная с таким именем, то внутри цикла она будет игнорироваться). Выражения для нижней и верхней границы диапазона чисел вычисляются один раз при входе в цикл. Если не указана фраза BY, то шаг итерации 1, в противном случае используется значение в BY, которое вычисляется, опять же, один раз при входе в цикл. Если указано REVERSE, то после каждой итерации величина шага вычитается, а не добавляется.

Примеры целочисленного FOR:

```
FOR i IN 1..10 LOOP
    -- внутри цикла переменная i будет иметь значения 1,2,3,4,5,6,7,8,9,10
END LOOP;

FOR i IN REVERSE 10..1 LOOP
    -- внутри цикла переменная i будет иметь значения 10,9,8,7,6,5,4,3,2,1
END LOOP;

FOR i IN REVERSE 10..1 BY 2 LOOP
    -- внутри цикла переменная i будет иметь значения 10,8,6,4,2
END LOOP;
```

Если нижняя граница цикла больше верхней границы (или меньше, в случае REVERSE), то тело цикла не выполняется вообще. При этом ошибка не возникает.

Если у цикла есть метка, то к переменной цикла можно обращаться по имени, квалифицированному меткой.

40.6.4. Цикл по результатам запроса

Другой вариант FOR позволяет организовать цикл по результатам запроса. Синтаксис:

```
[ <<label>> ]
FOR target IN query LOOP
    statements
END LOOP [ label ];
```

Переменная *target* может быть строковой переменной, переменной типа record или разделенным запятыми списком скалярных переменных. В переменную *target* последовательно присваиваются строки результата запроса, и для каждой строки выполняется тело цикла. Пример:

```
CREATE FUNCTION cs_refresh_mviews() RETURNS integer AS $$
DECLARE
    mviews RECORD;
BEGIN
    RAISE NOTICE 'Обновление материализованных представлений...';

    FOR mviews IN SELECT * FROM cs_materialized_views ORDER BY sort_key LOOP

        -- Сейчас "mviews" содержит одну запись из cs_materialized_views

        RAISE NOTICE 'Обновляется мат. представление %s ...',
quote_ident(mviews.mv_name);
        EXECUTE 'TRUNCATE TABLE ' || quote_ident(mviews.mv_name);
        EXECUTE 'INSERT INTO '
                  || quote_ident(mviews.mv_name) || ' '
                  || mviews.mv_query;
    END LOOP;

    RAISE NOTICE 'Закончено обновление материализованных представлений.';
    RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

Если цикл завершается по команде EXIT, то последняя присвоенная строка доступна и после цикла.

В цикле FOR можно использовать любые SQL команды, возвращающие строки. Чаще всего это SELECT, но могут быть и INSERT, UPDATE, DELETE с фразой RETURNING. А также некоторые утилиты, например EXPLAIN.

Для PL/pgSQL переменных в тексте запроса выполняется подстановка значений, план запроса кэшируется для возможного повторного использования, как подробно описано в Разд. 40.10.1 и Разд. 40.10.2.

Еще одна разновидность этого типа цикла FOR-IN-EXECUTE:

```
[ <<label>> ]
```

```
FOR target IN EXECUTE text_expression [ USING expression [, ... ] ] LOOP
    statements
END LOOP [ label ];
```

Она похожа на предыдущую форму, за исключением того, что текст запроса указывается в виде строкового выражения. Текст запроса формируется и для него строится план выполнения при каждом входе в цикл. Это дает программисту выбор между скоростью предварительно разобранного запроса и гибкостью динамического запроса, так же, как и в случае с обычным оператором EXECUTE. Как и в EXECUTE, значения параметров могут быть добавлены в команду с использованием USING.

Еще один способ организовать цикл по результатам запроса это объявить курсор. Описание в Разд. 40.7.4.

40.6.5. Цикл по элементам массива

Цикл FOREACH очень похож на FOR. Отличие в том, что вместо перебора строк SQL запроса происходит перебор элементов массива. (В целом, FOREACH предназначен для перебора выражений составного типа. Варианты реализации цикла для работы с прочими составными выражениями помимо массивов могут быть добавлены в будущем.) Синтаксис цикла FOREACH:

```
[ <<label>> ]
FOREACH target [ SLICE number ] IN ARRAY expression LOOP
    statements
END LOOP [ label ];
```

Без указания SLICE, или если SLICE равен 0, цикл выполняется по всем элементам массива, полученного из *expression*. В переменную *target* последовательно присваивается каждый элемент массива и для него выполняется тело цикла. Пример цикла по элементам целочисленного массива:

```
CREATE FUNCTION sum(int[]) RETURNS int8 AS $$
DECLARE
  s int8 := 0;
  x int;
BEGIN
  FOREACH x IN ARRAY $1
  LOOP
    s := s + x;
  END LOOP;
  RETURN s;
END;
$$ LANGUAGE plpgsql;
```

Обход элементов проводится в том порядке, в котором они сохранялись, независимо от размерности массива. Как правило, *target* это одиночная переменная, но может быть и списком переменных, когда элементы массива имеют составной тип (записи). В этом случае переменным присваиваются значения из последовательных столбцов составного элемента массива.

При положительном значении SLICE FOREACH выполняет итерации по срезам массива, а не по отдельным элементам. Значение SLICE должно быть целым числом, не превышающим размерности массива. Переменная *target* должна быть массивом, который получает последовательные срезы исходного массива, где размерность каждого среза задается значением SLICE. Пример цикла по одномерным срезам:

```
CREATE FUNCTION scan_rows(int[]) RETURNS void AS $$
DECLARE
  x int[];
BEGIN
  FOREACH x SLICE 1 IN ARRAY $1
  LOOP
    RAISE NOTICE 'row = %', x;
  END LOOP;
END;
$$ LANGUAGE plpgsql;

SELECT scan_rows(ARRAY[[1,2,3],[4,5,6],[7,8,9],[10,11,12]]);

NOTICE:  row = {1,2,3}
NOTICE:  row = {4,5,6}
NOTICE:  row = {7,8,9}
NOTICE:  row = {10,11,12}
```

40.6.6. Обработка ошибок

По умолчанию любая возникающая ошибка прерывает выполнение PL/pgSQL функции, а также транзакцию, относящуюся к этой функции. Использование в блоке секции EXCEPTION позволяет перехватывать и обрабатывать ошибки. Синтаксис секции EXCEPTION дополняет синтаксис обычного блока:

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
EXCEPTION
    WHEN condition [ OR condition ... ] THEN
        handler_statements
    [ WHEN condition [ OR condition ... ] THEN
          handler_statements
      ... ]
END;
```

Если ошибок не было, то выполняются все *statements* блока и управление переходит к следующему оператору после END. Но если при выполнении *statements* происходит ошибка, то дальнейшая обработка прекращается и управление переходит к списку исключений в секции EXCEPTION. В этом списке ищется первое исключение, условие которого соответствует ошибке. Если исключение найдено, то выполняются соответствующие *handler_statements* и управление переходит к следующему оператору после END. Если исключение не найдено, то ошибка передается наружу, как будто секции

EXCEPTION не было. При этом ошибку можно перехватить в секции EXCEPTION внешнего блока. Если ошибка так и не была перехвачена, то обработка функции прекращается.

Допустимые имена *condition* перечислены в Прил. А. Специальное имя OTHERS соответствует любой ошибке, за исключением QUERY_CANCELED. Можно явно обработать QUERY_CANCELED, но зачастую это неразумно. Имена исключений не чувствительны к регистру. Кроме того, *condition* можно указать через соответствующий SQLSTATE код. В следующем примере обе строки эквивалентны:

```
WHEN division_by_zero THEN ...
WHEN SQLSTATE '22012' THEN ...
```

Если при выполнении *handler_statements* возникнет новая ошибка, то она не может быть перехвачена в этой секции EXCEPTION. Ошибка передается наружу и её можно перехватить в секции EXCEPTION внешнего блока.

При выполнении команд в секции EXCEPTION локальные переменные PL/pgSQL функции сохраняют те значения, которые были на момент возникновения ошибки. Однако, будут отменены все изменения в базе данных, выполненные в блоке. В качестве примера рассмотрим следующий фрагмент:

```
INSERT INTO mytab(firstname, lastname) VALUES('Tom', 'Jones');
BEGIN
    UPDATE mytab SET firstname = 'Joe' WHERE lastname = 'Jones';
    x := x + 1;
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'перехватили ошибку division_by_zero';
        RETURN x;
END;
```

При присвоении значения переменной y произойдет ошибка division_by_zero. Она будет перехвачена в секции EXCEPTION. Оператор RETURN вернет значение x, увеличенное на единицу, но изменения сделанные командой UPDATE будут отменены. Изменения, выполненные командой INSERT, которая предшествует блоку, не будут отменены. В результате, база данных будет содержать Tom Jones, а не Joe Jones.

> **Подсказка:** Наличие секции EXCEPTION значительно увеличивает накладные расходы на вход/выход из блока. Поэтому не используйте EXCEPTION без надобности.

Пример 40-2. Обработка исключений для команд UPDATE/INSERT

В этом примере обработка исключений используется для того, чтобы определить какую команду выполнить UPDATE или INSERT:

```
CREATE TABLE db (a INT PRIMARY KEY, b TEXT);

CREATE FUNCTION merge_db(key INT, data TEXT) RETURNS VOID AS
$$
```

```
BEGIN
    LOOP
        -- для начала UPDATE записи по значению ключа
        UPDATE db SET b = data WHERE a = key;
        IF found THEN
            RETURN;
        END IF;
        -- записи с таким ключом нет, поэтому попытаемся её вставить
        -- если параллельно с нами кто-то еще пытается вставить запись с таким
же ключом,
        -- то мы получим ошибку уникальности
        BEGIN
            INSERT INTO db(a,b) VALUES (key, data);
            RETURN;
        EXCEPTION WHEN unique_violation THEN
            -- Здесь ничего не делаем и идем на следующую итерацию цикла,
            -- чтобы повторно попытаться сделать UPDATE.
        END;
    END LOOP;
END;
$$
LANGUAGE plpgsql;

SELECT merge_db(1, 'david');
SELECT merge_db(1, 'dennis');
```

В примере предполагается, что ошибка `unique_violation` вызвана командой `INSERT` текущего блока, а не, скажем, `INSERT` из триггерной функции таблицы. Здесь также не учтено, что у таблицы может быть несколько уникальных ключей, поэтому попытка повторить операцию будет предприниматься вне зависимости от того, уникальность какого ключа привела к ошибке. Далее будут рассмотрены возможности, позволяющие убедиться, что мы обрабатываем именно ту ошибку, которую хотели.

40.6.6.1. Получение информации об ошибке

При обработке исключений часто бывает необходимым получить детальную информацию о произошедшей ошибке. Для этого в PL/pgSQL есть два способа: использование специальных переменных и команда `GET STACKED DIAGNOSTICS`.

Внутри секции `EXCEPTION` специальная переменная `SQLSTATE` содержит код ошибки, для которой было вызвано исключение (смотри список возможных кодов ошибок в Табл. A-1). Специальная переменная `SQLERRM` содержит сообщение об ошибке, связанное с исключением. Эти переменные являются неопределенными вне секции `EXCEPTION`.

Также, при обработке исключений, дополнительную информацию можно получить командой `GET STACKED DIAGNOSTICS`, которая имеет вид:

```
GET STACKED DIAGNOSTICS variable { = | := } item [ , ... ];
```

Каждый *`item`* является ключевым словом, идентифицирующим значение состояния, которое будет присвоено указанной переменной. Переменная должна быть соответствующего типа данных. Идентификаторы, доступные в настоящее время, приведены в Табл. 40-1.

Таблица 40-1. Диагностические коды ошибок

Название	Тип	Описание
RETURNED_SQLSTATE	text	код исключения, возвращаемый SQLSTATE
COLUMN_NAME	text	имя столбца, относящегося к исключению
CONSTRAINT_NAME	text	имя ограничения целостности, относящегося к исключению
PG_DATATYPE_NAME	text	имя типа данных, относящегося к исключению
MESSAGE_TEXT	text	текст основного сообщения исключения
TABLE_NAME	text	имя таблицы, относящейся к исключению
SCHEMA_NAME	text	имя схемы, относящейся к исключению
PG_EXCEPTION_DETAIL	text	текст детального сообщения исключения (если есть)
PG_EXCEPTION_HINT	text	текст подсказки к исключению (если есть)
PG_EXCEPTION_CONTEXT	text	строка(или строки) с описанием стека вызова

Если исключение не устанавливает значение для идентификатора, то возвращается пустая строка.

Пример:

```
DECLARE
  text_var1 text;
  text_var2 text;
  text_var3 text;
BEGIN
  -- здесь происходит обработка, которая может вызвать исключение
  ...
EXCEPTION WHEN OTHERS THEN
  GET STACKED DIAGNOSTICS text_var1 = MESSAGE_TEXT,
                          text_var2 = PG_EXCEPTION_DETAIL,
                          text_var3 = PG_EXCEPTION_HINT;
END;
```

40.6.7. Получение информации о выполнении в текущий момент

Команда `GET [ CURRENT ] DIAGNOSTICS` извлекает информацию о текущем состоянии выполнения (в то время как обсуждавшаяся выше команда `GET STACKED DIAGNOSTICS` выдает информацию о состоянии выполнения на момент предыдущей ошибки). Команда имеет следующий вид:

```
GET [ CURRENT ] DIAGNOSTICS variable { = | := } item [ , ... ];
```

В настоящий момент поддерживается только один *`item`* `PG_CONTEXT`, который возвращает значение типа `text`, содержащее стек вызова. Стек вызова может состоять из нескольких строк, первая строка относится к выполняемый в текущий момент команде `GET DIAGNOSTICS` в текущей функции. Вторая и последующие строки относятся к следующим функциям далее вверх по стеку вызова. Например:

```
CREATE OR REPLACE FUNCTION outer_func() RETURNS integer AS $$
```

```
BEGIN
  RETURN inner_func();
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION inner_func() RETURNS integer AS $$
DECLARE
  stack text;
BEGIN
  GET DIAGNOSTICS stack = PG_CONTEXT;
  RAISE NOTICE E'--- Стек вызова ---\n%', stack;
  RETURN 1;
END;
$$ LANGUAGE plpgsql;

SELECT outer_func();

NOTICE:  --- Стек вызова ---
PL/pgSQL function inner_func() line 5 at GET DIAGNOSTICS
PL/pgSQL function outer_func() line 3 at RETURN
CONTEXT:  PL/pgSQL function outer_func() line 3 at RETURN
 outer_func
 ------------
           1
(1 row)
```

40.7. Курсоры

Вместо того чтобы сразу выполнять весь запрос, есть возможность настроить курсор, инкапсулирующий запрос, и затем получать результат запроса по нескольку строк за раз. Одна из причин так делать заключается в том, чтобы избежать переполнения памяти, когда результат содержит большое количество строк. (Пользователям PL/pgSQL не нужно об этом беспокоиться, так как циклы `FOR` автоматически используют курсоры, чтобы избежать проблем с памятью.) Более интересным вариантом использования является возврат из функции ссылки на курсор, что позволяет вызывающему получать строки запроса. Это эффективный способ получать большие наборы строк из функций.

40.7.1. Объявление курсорных переменных

Доступ к курсорам в PL/pgSQL осуществляется через курсорные переменные, которые всегда имеют специальный тип данных `refcursor`. Один из способов создать курсорную переменную, просто объявить её как переменную типа `refcursor`. Другой способ заключается в использовании синтаксиса объявления курсора, который в общем виде выглядит так:

`name` `[ [ NO ] SCROLL ] CURSOR [ ( ` *`arguments`* ` ) ] FOR` *`query`*`;`

(Для совместимости с Oracle, `FOR` можно заменять на `IS`.) Если указана опция `SCROLL`, то курсор можно будет прокручивать назад. При `NO SCROLL` прокрутка назад не разрешается. Если ничего не указано, то возможность прокрутки назад зависит от запроса. Если указаны

arguments, то они должны представлять собой пары *name datatype*, разделенные через запятую. Эти пары определяют имена, которые будут заменены значениями параметров в данном запросе. Фактические значения для замены этих имен появятся позже, при открытии курсора.

Примеры:

```
DECLARE
    curs1 refcursor;
    curs2 CURSOR FOR SELECT * FROM tenk1;
    curs3 CURSOR (key integer) FOR SELECT * FROM tenk1 WHERE unique1 = key;
```

Все три переменные имеют тип данных refcursor. Первая может быть использована с любым запросом, вторая связана (bound) с полностью сформированным запросом, а последняя связана с параметризованным запросом. (key будет заменен целочисленным значением параметра при открытии курсора.) Про переменную curs1 говорят, что она является несвязанной (unbound), т.к. к ней не привязан никакой запрос.

40.7.2. Открытие курсора

Прежде чем получать строки из курсора, его нужно открыть. (Это эквивалентно действию SQL команды DECLARE CURSOR.) В PL/pgSQL есть три формы оператора OPEN, две из которых используются для несвязанных курсорных переменных, а третья для связанных.

> **Замечание:** Связанные курсорные переменные можно использовать с циклом FOR без явного открытия курсора, как описано в Разд. 40.7.4.

40.7.2.1. OPEN FOR *query*

```
OPEN unbound_cursorvar [ [ NO ] SCROLL ] FOR query;
```

Курсорная переменная открывается и получает конкретный запрос для выполнения. Курсор не может уже быть открытым, а курсорная переменная обязана быть несвязанной (то есть просто переменной типа refcursor). Запрос должен быть командой SELECT или любой другой, которая возвращает строки (к примеру EXPLAIN). Запрос обрабатывается так же, как и другие команды SQL в PL/pgSQL: имена PL/pgSQL переменных заменяются на значения, план запроса кэшируется для повторного использования. Подстановка значений PL/pgSQL переменных проводится при открытии курсора командой OPEN, последующие изменения значений переменных не влияют на работу курсора. SCROLL и NO SCROLL имеют тот же смысл, что и для связанного курсора.

Пример:

```
OPEN curs1 FOR SELECT * FROM foo WHERE key = mykey;
```

40.7.2.2. OPEN FOR EXECUTE

```
OPEN unbound_cursorvar [ [ NO ] SCROLL ] FOR EXECUTE query_string
```

```
[ USING expression [, ... ] ];
```

Курсорная переменная открывается и получает конкретный запрос для выполнения. Курсор не может уже быть открытым, а курсорная переменная обязана быть несвязанной (то есть просто переменной типа `refcursor`). Запрос задается строковым выражением, так же, как в команде `EXECUTE`. Как обычно, это дает гибкость, план запроса от раза к разу может меняться (смотри Разд. 40.10.2). Это также означает, что замена переменных выполняется не в командной строке. Как и в `EXECUTE`, для вставки в динамическую команду значений параметров используется `USING`. `SCROLL` и `NO SCROLL` имеют тот же смысл, что и для связанного курсора.

Пример:

```
OPEN curs1 FOR EXECUTE 'SELECT * FROM ' || quote_ident(tabname)
                                        || ' WHERE col1 = $1' USING keyvalue;
```

В этом примере имя таблицы вставляется в текст запроса, поэтому рекомендуется использовать функцию `quote_ident()` для защиты от SQL инъекций. Значение для сравнения с `col1` вставляется через `USING`, поэтому его не требуется заключать в кавычки.

40.7.2.3. Открытие связанного курсора

```
OPEN bound_cursorvar [ ( [ argument_name := ] argument_value [, ...] ) ];
```

Эта форма `OPEN` используется для открытия курсорной переменной, которая была связана с запросом при объявлении. Курсор не может уже быть открытым. Список фактических значений аргументов должен присутствовать только в том случае, если курсор объявлялся с параметрами. Эти значения будут подставлены в запрос.

План запроса для связанного курсора всегда считается кэшируемым. В этом случае, нет эквивалента `EXECUTE`. Обратите внимание, что `SCROLL` и `NO SCROLL` не могут быть указаны в этой форме `OPEN`, возможность прокрутки назад была определена при объявлении курсора.

При передаче значений аргументов можно использовать позиционную или именную нотацию. В позиционной нотации все аргументы указываются по порядку. В именной нотации имя каждого аргумента отделяется от выражения аргумента с помощью `:=`. Это подобно вызову функций, описанному в Разд. 4.3. Также разрешается смешивать позиционную и именную нотации.

Примеры (здесь используются ранее объявленные курсоры):

```
OPEN curs2;
OPEN curs3(42);
OPEN curs3(key := 42);
```

Так как для связанного курсора выполняется подстановка значений переменных, то, на самом деле, существует два способа передать значения в курсор. Либо использовать явные

аргументы в OPEN, либо неявно, ссылаясь на PL/pgSQL переменные в запросе. В связанном курсоре можно ссылаться только на те переменные, которые были объявлены до самого курсора. В любом случае, значение переменной для подстановки в запрос будет определяться на момент выполнения OPEN. Вот еще один способ получить тот же результат с curs3, как в примере выше:

```
DECLARE
    key integer;
    curs4 CURSOR FOR SELECT * FROM tenk1 WHERE unique1 = key;
BEGIN
    key := 42;
    OPEN curs4;
```

40.7.3. Использование курсоров

После того, как курсор был открыт, с ним можно работать при помощи описанных здесь операторов.

Работать с курсором необязательно в той же функции, где он был открыт. Из функции можно вернуть значение с типом refcursor, что позволит вызывающему продолжить работу с курсором. (Внутри refcursor представляет собой обычное строковое имя так называемого портала, содержащего активный запрос курсора. Это имя можно передавать, присваивать другим переменным с типом refcursor и так далее, при этом портал не нарушается.)

Все порталы неявно закрываются в конце транзакции. Поэтому значение refcursor можно использовать для ссылки на открытый курсор только до конца транзакции.

40.7.3.1. FETCH

```
FETCH [ direction { FROM | IN } ] cursor INTO target;
```

FETCH извлекает следующую строку из курсора в *target*. В качестве *target* может быть строковая переменная, переменная типа record, или разделенный запятыми список простых переменных, как и в SELECT INTO. Если следующей строки нет, в *target* присваивается NULL. Как и в SELECT INTO, проверить была ли получена запись можно при помощи специальной переменной FOUND.

Значение *direction* может быть любым допустимым в SQL команде FETCH вариантом, кроме тех, что извлекают более одной строки. А именно: NEXT, PRIOR, FIRST, LAST, ABSOLUTE *count*, RELATIVE *count*, FORWARD или BACKWARD. Без указания *direction* используется значение NEXT. Значения *direction*, которые требуют перемещения назад, приведут к ошибке, если курсор не был объявлен или открыт с опцией SCROLL.

cursor это переменная с типом refcursor, которая ссылается на открытый портал курсора.

Примеры:

```
FETCH curs1 INTO rowvar;
FETCH curs2 INTO foo, bar, baz;
FETCH LAST FROM curs3 INTO x, y;
FETCH RELATIVE -2 FROM curs4 INTO x;
```

40.7.3.2. MOVE

```
MOVE [ direction { FROM | IN } ] cursor;
```

MOVE перемещает курсор без извлечения данных. MOVE работает точно также как и FETCH, но при этом только перемещает курсор и не извлекает строку, к которой переместился. Как и в SELECT INTO, проверить успешность перемещения можно с помощью специальной переменной FOUND.

Значение *direction* может быть любым допустимым в SQL команде FETCH вариантом, а именно: NEXT, PRIOR, FIRST, LAST, ABSOLUTE *count*, RELATIVE *count*, ALL, FORWARD [*count* | ALL] или BACKWARD [*count* | ALL]. Без указания *direction* используется значение NEXT. Значения *direction*, которые требуют перемещения назад, приведут к ошибке, если курсор не был объявлен или открыт с опцией SCROLL.

Примеры:

```
MOVE curs1;
MOVE LAST FROM curs3;
MOVE RELATIVE -2 FROM curs4;
MOVE FORWARD 2 FROM curs4;
```

40.7.3.3. UPDATE/DELETE WHERE CURRENT OF

```
UPDATE table SET ... WHERE CURRENT OF cursor;
DELETE FROM table WHERE CURRENT OF cursor;
```

Когда курсор позиционирован на строку таблицы, эту строку можно изменить или удалить при помощи курсора. Есть ограничения на то, каким может быть запрос курсора (в частности, не должно быть группировок), и крайне желательно использовать фразу FOR UPDATE. Для дополнительной информации, смотри справку по DECLARE.

Пример:

```
UPDATE foo SET dataval = myval WHERE CURRENT OF curs1;
```

40.7.3.4. CLOSE

```
CLOSE cursor;
```

CLOSE закрывает связанный с курсором портал. Используется для того, чтобы освободить ресурсы раньше, чем закончится транзакция, или чтобы освободить курсорную переменную для повторного открытия.

Пример:

```
CLOSE curs1;
```

40.7.3.5. Возврат курсора из функции

Курсоры можно возвращать из PL/pgSQL функции. Это полезно, когда нужно вернуть множество строк и столбцов, особенно если выборки очень большие. Для этого, в функции открывается курсор и его имя возвращается вызывающему (или просто открывается курсор, используя указанное имя портала, каким-либо образом известное вызывающему). Вызывающий затем может извлекать строки из курсора. Курсор может быть закрыт вызывающим или он будет автоматически закрыт при завершении транзакции.

Имя портала, используемое для курсора, может быть указано разработчиком или будет генерироваться автоматически. Чтобы указать имя портала, нужно просто присвоить строку в переменную `refcursor` перед его открытием. Значение строки переменной `refcursor` будет использоваться командой `OPEN` как имя портала. Однако, если переменная `refcursor` имеет значение NULL, `OPEN` автоматически генерирует имя, которое не конфликтует с любым существующим порталом и присваивает его переменной `refcursor`.

> **Замечание:** Связанная курсорная переменная инициализируется в строковое значение, представляющее собой имя самой переменной. Таким образом, имя портала совпадает с именем курсорной переменной, кроме случаев, когда разработчик переопределил имя, присвоив новое значение перед открытием курсора. Несвязанная курсорная переменная инициализируется в NULL и получит автоматически сгенерированное уникальное имя, если не будет переопределена.

Следующий пример показывает один из способов передачи имени курсора вызывающему:

```
CREATE TABLE test (col text);
INSERT INTO test VALUES ('123');

CREATE FUNCTION reffunc(refcursor) RETURNS refcursor AS '
BEGIN
    OPEN $1 FOR SELECT col FROM test;
    RETURN $1;
END;
' LANGUAGE plpgsql;

BEGIN;
SELECT reffunc('funccursor');
FETCH ALL IN funccursor;
COMMIT;
```

В следующем примере используется автоматическая генерация имени курсора:

```
CREATE FUNCTION reffunc2() RETURNS refcursor AS '
DECLARE
    ref refcursor;
BEGIN
    OPEN ref FOR SELECT col FROM test;
    RETURN ref;
END;
' LANGUAGE plpgsql;
```

```
-- для использования курсоров, необходимо начать транзакцию
BEGIN;
SELECT reffunc2();

      reffunc2
--------------------
 <unnamed cursor 1>
(1 row)

FETCH ALL IN "<unnamed cursor 1>";
COMMIT;
```

В следующем примере показан один из способов вернуть несколько курсоров из одной функции:

```
CREATE FUNCTION myfunc(refcursor, refcursor) RETURNS SETOF refcursor AS $$
BEGIN
    OPEN $1 FOR SELECT * FROM table_1;
    RETURN NEXT $1;
    OPEN $2 FOR SELECT * FROM table_2;
    RETURN NEXT $2;
END;
$$ LANGUAGE plpgsql;

-- для использования курсоров необходимо начать транзакцию
BEGIN;

SELECT * FROM myfunc('a', 'b');

FETCH ALL FROM a;
FETCH ALL FROM b;
COMMIT;
```

40.7.4. Обработка курсора в цикле

Один из вариантов цикла FOR позволяет перебирать строки, возвращенные курсором. Вот его синтаксис:

```
[ <<label>> ]
FOR recordvar IN bound_cursorvar [ ( [ argument_name := ] argument_value [, ...]
) ] LOOP
    statements
END LOOP [ label ];
```

Курсорная переменная должна быть связана с запросом при объявлении. Курсор *не может* быть открытым. Команда FOR автоматически открывает курсор и автоматически закрывает при завершении цикла. Список фактических значений аргументов должен присутствовать только в том случае, если курсор объявлялся с параметрами. Эти значения будут подставлены в запрос, также как и при выполнении OPEN (смотри Разд. 40.7.2.3).

Переменная *recordvar* автоматически определяется как переменная типа record и существует только внутри цикла (другие объявленные переменные с таким именем игнорируется в цикле). Каждая возвращаемая курсором строка последовательно присваивается этой переменной и выполняется тело цикла.

40.8. Сообщения и ошибки

Команда RAISE предназначена для вывода сообщений и вызова ошибок.

```
RAISE [ level ] 'format' [, expression [, ... ]] [ USING option = expression
[, ... ] ];
RAISE [ level ] condition_name [ USING option = expression [, ... ] ];
RAISE [ level ] SQLSTATE 'sqlstate' [ USING option = expression [, ... ] ];
RAISE [ level ] USING option = expression [, ... ];
RAISE ;
```

level задает уровень серьезности ошибки. Возможные значения: DEBUG, LOG, INFO, NOTICE, WARNING и EXCEPTION. По умолчанию используется EXCEPTION. EXCEPTION вызывает ошибку (что обычно прерывает текущую транзакцию), остальные значения *level* только генерируют сообщения с различными уровнями приоритета. Будут ли сообщения конкретного приоритета переданы клиенту, или записаны в лог сервера, или и то и другое, зависит от конфигурационных переменных log_min_messages и client_min_messages. Смотри Гл. 18 для дополнительной информации.

После *level* можно записать *format* (это должна быть простая символьная строка, не являющаяся выражением). Строка *format* задает текст сообщения об ошибке. После *format* может следовать список необязательных аргументов выражений для вставки в текст сообщения. Внутри строки *format* символ % заменяется на текстовое представление следующего необязательного значения аргумента. Чтобы вставить символ %, используйте %%.

В следующем примере символ % будет заменен на значение v_job_id:

```
RAISE NOTICE 'Вызов функции cs_create_job(%)', v_job_id;
```

При помощи фразы USING и последующих элементов *option* = *expression* можно добавить дополнительную информацию к отчету об ошибке. Все *expression* представляют собой строковые выражения. Возможные ключевые слова для *option* следующие:

MESSAGE

Устанавливает текст сообщения об ошибке. Эта опция не может использоваться, если в команде RAISE присутствует *format* перед USING.

DETAIL

Предоставляет детальное сообщение об ошибке.

HINT

Предоставляет подсказку по вызванной ошибке.

ERRCODE

Устанавливает код ошибки (SQLSTATE). Код ошибки задается либо по имени, как

показано в Прил. А, или напрямую, пятисимвольный код SQLSTATE.

COLUMN
CONSTRAINT
DATATYPE
TABLE
SCHEMA

Предоставляет имя соответствующего объекта, связанного с ошибкой.

Этот пример прерывает транзакцию и устанавливает сообщение об ошибке с подсказкой:

```
RAISE EXCEPTION 'Несуществующий ID --> %', user_id
      USING HINT = 'Проверьте ваш пользовательский ID';
```

Следующие два примера демонстрируют эквивалентные способы задания SQLSTATE:

```
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = 'unique_violation';
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = '23505';
```

У команды RAISE есть и другой синтаксис, в котором в качестве главного аргумента используется имя или код SQLSTATE ошибки. Например:

```
RAISE division_by_zero;
RAISE SQLSTATE '22012';
```

Фразу USING в этом синтаксисе можно использовать для того, чтобы переопределить стандартное сообщение об ошибке, детальное сообщение, подсказку. Еще один вариант предыдущего примера:

```
RAISE unique_violation USING MESSAGE = 'ID пользователя уже существует: ' ||
user_id;
```

Еще один вариант - использовать RAISE USING или RAISE *level* USING, а всё остальное прописать в опциях USING.

И заключительный вариант, в котором RAISE не имеет параметров вообще. Эта форма может использоваться только в секции EXCEPTION блока и предназначена для того, чтобы повторно вызвать ошибку, которая сейчас перехвачена и обрабатывается.

> **Замечание:** До версии PostgreSQL 9.1 команда RAISE без параметров всегда вызывала ошибку с выходом из блока, содержащего активную секцию EXCEPTION. Эту ошибку нельзя было перехватить, даже если RAISE в секции EXCEPTION поместить во вложенный блок со своей секцией EXCEPTION. Это было сочтено удивительным и не совместимым с Oracle PL/SQL.

Если в команде RAISE EXCEPTION не задано ни имя, ни SQLSTATE код, то по умолчанию используются RAISE_EXCEPTION (P0001). В качестве текста сообщения об ошибке (если не задан) используется имя или SQLSTATE код.

> **Замечание:** При задании SQLSTATE кода необязательно использовать только

список предопределенных кодов ошибок. В качестве кода ошибки может быть любое пятисимвольное значение, состоящее из цифр и/или ASCII символов в верхнем регистре, кроме `00000`. Не рекомендуется использовать коды ошибок, которые заканчиваются на `000`, потому что так обозначаются коды категорий. И чтобы их перехватить, нужно перехватывать целую категорию.

40.9. Триггерные процедуры

40.9.1. Триггеры на изменение данных

В PL/pgSQL можно создавать триггерные процедуры. Триггерная процедура создается командой `CREATE FUNCTION`, при этом у функции не должно быть аргументов, а тип возвращаемого значения должен быть `trigger`. Обратите внимание, что функция создается без аргументов, даже если ей нужно получить аргументы, указанные в команде `CREATE TRIGGER`. Аргументы триггера передаются через массив `TG_ARGV`, как будет показано ниже.

Когда PL/pgSQL функция срабатывает как триггер, в блоке верхнего уровня автоматически создаются несколько специальных переменных:

`NEW`

Тип данных `RECORD`. Переменная содержит новую строку базы данных для команд `INSERT`/`UPDATE` в триггерах уровня строки. В триггерах уровня оператора и для команды `DELETE` этой переменной значение не присваивается.

`OLD`

Тип данных `RECORD`. Переменная содержит старую строку базы данных для команд `UPDATE`/`DELETE` в триггерах уровня строки. В триггерах уровня оператора и для команды `INSERT` этой переменной значение не присваивается.

`TG_NAME`

Тип данных `name`. Переменная содержит имя сработавшего триггера.

`TG_WHEN`

Тип данных `text`. Строка, содержащая `BEFORE`, `AFTER` или `INSTEAD OF`, в зависимости от определения триггера.

`TG_LEVEL`

Тип данных `text`. Строка, содержащая `ROW` или `STATEMENT`, в зависимости от определения триггера.

`TG_OP`

Тип данных text. Строка, содержащая INSERT, UPDATE, DELETE или TRUNCATE, в зависимости от того, для какой операции сработал триггер.

TG_RELID

Тип данных oid. OID таблицы, для которой сработал триггер.

TG_RELNAME

Тип данных name. Имя таблицы, для которой сработал триггер. Эта переменная устарела и может стать недоступной в будущих релизах. Вместо неё нужно использовать TG_TABLE_NAME.

TG_TABLE_NAME

Тип данных name. Имя таблицы, для которой сработал триггер.

TG_TABLE_SCHEMA

Тип данных name. Имя схемы, содержащей таблицу, для которой сработал триггер.

TG_NARGS

Тип данных integer. Число аргументов в команде CREATE TRIGGER, которые передаются в триггерную процедуру.

TG_ARGV[]

Тип данных массив text. Аргументы от оператора CREATE TRIGGER. Индекс массива начинается с 0. Для недопустимых значений индекса (< 0 или >= tg_nargs) возвращается NULL.

Триггерная функция должна вернуть либо NULL, либо запись/строку, соответствующую структуре таблице, для которой сработал триггер.

Если BEFORE триггер уровня строки возвращает NULL, то все дальнейшие действия с этой строкой прекращаются (т.е. не срабатывают последующие триггера, команда INSERT/UPDATE/DELETE для этой строки не выполняется). Если возвращается не NULL, то дальнейшая обработка продолжается именно с этой строкой. Возвращение строки отличной от начальной NEW, изменяет строку, которая будет вставлена или изменена. Поэтому, если в триггерной функции нужно выполнить некоторые действия и не менять саму строку, то нужно возвратить переменную NEW (или её эквивалент). Для того чтобы изменить сохраняемую строку, можно поменять отдельные значения в переменной NEW и затем её вернуть. Либо создать и вернуть полностью новую переменную. В случае строчного триггера BEFORE для команды DELETE само возвращаемое значение не имеет прямого эффекта, но оно должно быть отличным от NULL, чтобы не прерывать обработку строки. Обратите внимание, что переменная NEW всегда NULL в триггерах на DELETE, поэтому возвращать её

не имеет смысла. Традиционной идиомой для триггеров DELETE является возврат переменной OLD.

Триггеры INSTEAD OF могут создаваться только как триггеры уровня строки и только для представлений. Если INSTEAD OF триггер возвращает NULL, то это значит, что он не произвел никаких изменений и дальнейшая обработка этой строки не требуется (т.е. для соответствующей команды INSERT/UPDATE/DELETE не срабатывают последующие триггера и не увеличивается счетчик обработанных строк). В остальных случаях должно возвращаться значение, отличное от NULL, что означает, что триггер выполнил требуемые действия. Команды INSERT и UPDATE должны возвращать NEW (которая может быть изменена в триггерной функции) для корректной работы INSERT RETURNING и UPDATE RETURNING. Возвращаемое значение также влияет на значение строки, которое будет передано в последующие триггеры. Для команды DELETE возвращаемое значение должно быть OLD.

Возвращаемое значение для строчного триггера AFTER и триггеров уровня оператора (BEFORE или AFTER) всегда игнорируется. Это может быть и NULL. Однако, в этих триггерах по-прежнему можно прервать вызвавшую их команду, для этого нужно явно вызвать ошибку.

Прим. 40-3 показывает пример триггерной процедуры в PL/pgSQL.

Пример 40-3. Триггерная процедура PL/pgSQL

Триггер гарантирует, что всякий раз, когда в таблице добавляется или изменяется запись, в этой записи сохраняется информация о текущем пользователе и временной метке. Также контролируется, что имя сотрудника указано и размер зарплаты выше нуля.

```
CREATE TABLE emp (
    empname text,
    salary integer,
    last_date timestamp,
    last_user text
);

CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
    BEGIN
        -- Проверим, что указаны имя сотрудника и зарплата
        IF NEW.empname IS NULL THEN
            RAISE EXCEPTION 'Не указано имя сотрудника';
        END IF;
        IF NEW.salary IS NULL THEN
            RAISE EXCEPTION 'Не указана зарплата для %', NEW.empname;
        END IF;

        -- Зачем работать, если за это еще нужно платить?
        IF NEW.salary < 0 THEN
            RAISE EXCEPTION 'У % не должна быть отрицательная зарплата',
NEW.empname;
        END IF;

        -- Запомним кто и когда изменил запись
```

```
        NEW.last_date := current_timestamp;
        NEW.last_user := current_user;
        RETURN NEW;
    END;
$emp_stamp$ LANGUAGE plpgsql;

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE emp_stamp();
```

Другой вариант вести журнал изменений для таблицы предполагает создание новой таблицы, которая будет содержать отдельную запись для каждой выполненной команды INSERT, UPDATE, DELETE. Этот подход можно рассматривать как аудирование изменений таблицы. Прим. 40-4 показывает реализацию триггерной процедуры для аудита в PL/pgSQL.

Пример 40-4. Триггерная процедура для аудита в PL/pgSQL

Триггер гарантирует, что любая команда на вставку, изменение или удаление строки в таблице `emp` будет записана (аудирована) в таблице `emp_audit`. Также записывается информация о пользователе, выполнившем операцию, временной метке и типе операции.

```
CREATE TABLE emp (
    empname           text NOT NULL,
    salary            integer
);

CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    stamp             timestamp NOT NULL,
    userid            text      NOT NULL,
    empname           text      NOT NULL,
    salary integer
);

CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER AS $emp_audit$
    BEGIN
        --
        -- Создаем строку в emp_audit, которая отражает выполненную операцию.
        -- Воспользуемся переменной TG_OP для определения типа операции.
        --
        IF (TG_OP = 'DELETE') THEN
            INSERT INTO emp_audit SELECT 'D', now(), user, OLD.*;
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            INSERT INTO emp_audit SELECT 'U', now(), user, NEW.*;
            RETURN NEW;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp_audit SELECT 'I', now(), user, NEW.*;
            RETURN NEW;
        END IF;
        RETURN NULL; -- возвращаемое значение для AFTER триггера не имеет
значения
    END;
$emp_audit$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit
AFTER INSERT OR UPDATE OR DELETE ON emp
    FOR EACH ROW EXECUTE PROCEDURE process_emp_audit();
```

У предыдущего примера есть разновидность, которая использует представление, соединяющее основную таблицу и таблицу аудита, для отображения даты последнего изменения каждой строки. При этом подходе по-прежнему ведется полный журнал аудита в отдельной таблице, но также имеется представление с упрощенным аудиторским следом. Это представление содержит временную метку, которая вычисляется для каждой строки из данных аудиторской таблицы. Прим. 40-5 показывает пример триггера на представление для аудита в PL/pgSQL.

Пример 40-5. Триггер на представление для аудита в PL/pgSQL

Триггер на представление используется для того, чтобы сделать это представление изменяемым и гарантировать, что любая команда на вставку, изменение или удаление строки в представлении будет записана (т.е. аудирована) в таблице `emp_audit`. Также записываются временная метка, имя пользователя и тип выполняемой операции. Представление показывает дату последнего изменения для каждой строки.

```
CREATE TABLE emp (
    empname           text PRIMARY KEY,
    salary            integer
);

CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    userid            text      NOT NULL,
    empname           text      NOT NULL,
    salary            integer,
    stamp             timestamp NOT NULL
);

CREATE VIEW emp_view AS
    SELECT e.empname,
           e.salary,
           max(ea.stamp) AS last_updated
      FROM emp e
      LEFT JOIN emp_audit ea ON ea.empname = e.empname
     GROUP BY 1, 2;

CREATE OR REPLACE FUNCTION update_emp_view() RETURNS TRIGGER AS $$
    BEGIN
        --
        -- Выполняем требуемую операцию в emp и создаем строку в emp_audit,
        -- которая отражает сделанную операцию.
        --
        IF (TG_OP = 'DELETE') THEN
            DELETE FROM emp WHERE empname = OLD.empname;
            IF NOT FOUND THEN RETURN NULL; END IF;

            OLD.last_updated = now();
            INSERT INTO emp_audit VALUES('D', user, OLD.*);
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            UPDATE emp SET salary = NEW.salary WHERE empname = OLD.empname;
            IF NOT FOUND THEN RETURN NULL; END IF;

            NEW.last_updated = now();
            INSERT INTO emp_audit VALUES('U', user, NEW.*);
```

```
            RETURN NEW;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp VALUES(NEW.empname, NEW.salary);

            NEW.last_updated = now();
            INSERT INTO emp_audit VALUES('I', user, NEW.*);
            RETURN NEW;
        END IF;
    END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit
INSTEAD OF INSERT OR UPDATE OR DELETE ON emp_view
    FOR EACH ROW EXECUTE PROCEDURE update_emp_view();
```

Один из вариантов использования триггеров это поддержание в актуальном состоянии отдельной таблицы итогов для некоторой таблицы. В некоторых случаях отдельная таблица с итогами может использоваться в запросах вместо основной таблицы. При этом зачастую время выполнения запросов значительно сокращается. Эта техника широко используется в хранилищах данных, где таблицы фактов могут быть очень большими. Прим. 40-6 показывает триггерную процедуру в PL/pgSQL, которая поддерживает таблицу итогов для таблицы фактов в хранилище данных.

Пример 40-6. Триггерная процедура в PL/pgSQL для поддержки таблицы итогов

Представленная здесь схема данных частично основана на примере *Grocery Store* из книги *The Data Warehouse Toolkit* (автор Ralph Kimball).

```
--
-- Основные таблицы: таблица измерений временных периодов и таблица фактов
продаж
--
CREATE TABLE time_dimension (
    time_key                    integer NOT NULL,
    day_of_week                 integer NOT NULL,
    day_of_month                integer NOT NULL,
    month                       integer NOT NULL,
    quarter                     integer NOT NULL,
    year                        integer NOT NULL
);
CREATE UNIQUE INDEX time_dimension_key ON time_dimension(time_key);

CREATE TABLE sales_fact (
    time_key                    integer NOT NULL,
    product_key                 integer NOT NULL,
    store_key                   integer NOT NULL,
    amount_sold                 numeric(12,2) NOT NULL,
    units_sold                  integer NOT NULL,
    amount_cost                 numeric(12,2) NOT NULL
);
CREATE INDEX sales_fact_time ON sales_fact(time_key);

--
-- Таблица с итогами продаж по периодам
--
CREATE TABLE sales_summary_bytime (
    time_key                    integer NOT NULL,
    amount_sold                 numeric(15,2) NOT NULL,
```

```
    units_sold                  numeric(12) NOT NULL,
    amount_cost                 numeric(15,2) NOT NULL
);
CREATE UNIQUE INDEX sales_summary_bytime_key ON sales_summary_bytime(time_key);

--
-- Функция и триггер, обновляющие столбцы с итоговыми значениями при выполнении
-- команд INSERT, UPDATE, DELETE
--
CREATE OR REPLACE FUNCTION maint_sales_summary_bytime() RETURNS TRIGGER
AS $maint_sales_summary_bytime$
    DECLARE
        delta_time_key          integer;
        delta_amount_sold       numeric(15,2);
        delta_units_sold        numeric(12);
        delta_amount_cost       numeric(15,2);
    BEGIN

        -- определим на сколько произошло увеличение/уменьшение количеств
        IF (TG_OP = 'DELETE') THEN

            delta_time_key = OLD.time_key;
            delta_amount_sold = -1 * OLD.amount_sold;
            delta_units_sold = -1 * OLD.units_sold;
            delta_amount_cost = -1 * OLD.amount_cost;

        ELSIF (TG_OP = 'UPDATE') THEN

            -- запрещаем изменять time_key
            -- для таких изменений больше подходит DELETE + INSERT
            IF ( OLD.time_key != NEW.time_key) THEN
                RAISE EXCEPTION 'Запрещено изменение time_key : % -> %',
                                                        OLD.time_key,
NEW.time_key;
            END IF;

            delta_time_key = OLD.time_key;
            delta_amount_sold = NEW.amount_sold - OLD.amount_sold;
            delta_units_sold = NEW.units_sold - OLD.units_sold;
            delta_amount_cost = NEW.amount_cost - OLD.amount_cost;

        ELSIF (TG_OP = 'INSERT') THEN

            delta_time_key = NEW.time_key;
            delta_amount_sold = NEW.amount_sold;
            delta_units_sold = NEW.units_sold;
            delta_amount_cost = NEW.amount_cost;

        END IF;


        -- вставляем или обновляем строку в таблице итогов.
        <<insert_update>>
        LOOP
            UPDATE sales_summary_bytime
                SET amount_sold = amount_sold + delta_amount_sold,
                    units_sold = units_sold + delta_units_sold,
                    amount_cost = amount_cost + delta_amount_cost
                WHERE time_key = delta_time_key;

            EXIT insert_update WHEN found;
```

```
            BEGIN
                INSERT INTO sales_summary_bytime (
                            time_key,
                            amount_sold,
                            units_sold,
                            amount_cost)
                    VALUES (
                            delta_time_key,
                            delta_amount_sold,
                            delta_units_sold,
                            delta_amount_cost
                           );

                EXIT insert_update;

            EXCEPTION
                WHEN UNIQUE_VIOLATION THEN
                    -- ничего не делаем
            END;
        END LOOP insert_update;

        RETURN NULL;

    END;
$maint_sales_summary_bytime$ LANGUAGE plpgsql;

CREATE TRIGGER maint_sales_summary_bytime
AFTER INSERT OR UPDATE OR DELETE ON sales_fact
    FOR EACH ROW EXECUTE PROCEDURE maint_sales_summary_bytime();

INSERT INTO sales_fact VALUES(1,1,1,10,3,15);
INSERT INTO sales_fact VALUES(1,2,1,20,5,35);
INSERT INTO sales_fact VALUES(2,2,1,40,15,135);
INSERT INTO sales_fact VALUES(2,3,1,10,1,13);
SELECT * FROM sales_summary_bytime;
DELETE FROM sales_fact WHERE product_key = 1;
SELECT * FROM sales_summary_bytime;
UPDATE sales_fact SET units_sold = units_sold * 2;
SELECT * FROM sales_summary_bytime;
```

40.9.2. Триггеры событий

В PL/pgSQL можно создавать триггеры событий. PostgreSQL требует, чтобы процедура, которая вызывается как триггер события, была объявлена без аргументов и имела тип возвращаемого значения `event_trigger`.

Когда PL/pgSQL функция срабатывает как триггер события, в блоке верхнего уровня автоматически создаются несколько специальных переменных:

`TG_EVENT`

Тип данных `text`. Строка, содержащая событие, по которому сработал триггер.

`TG_TAG`

Тип данных `text`. Переменная, содержащая тэг команды, для которой сработал триггер.

Прим. 40-7 показывает пример процедуры триггера события в PL/pgSQL.

Пример 40-7. Процедура триггера события в PL/pgSQL

Триггер просто выдает сообщение всякий раз, когда выполняется поддерживаемая команда.

```
CREATE OR REPLACE FUNCTION snitch() RETURNS event_trigger AS $$
BEGIN
    RAISE NOTICE 'Произошло событие: % %', tg_event, tg_tag;
END;
$$ LANGUAGE plpgsql;

CREATE EVENT TRIGGER snitch ON ddl_command_start EXECUTE PROCEDURE snitch();
```

40.10. PL/pgSQL изнутри

В этом разделе обсуждаются некоторые детали реализации, которые пользователям PL/pgSQL важно знать.

40.10.1. Подстановка переменных

SQL операторы и выражения внутри PL/pgSQL функции могут ссылаться на переменные и параметры этой функции. За кулисами PL/pgSQL заменяет параметры запросов для таких ссылок. Параметры будут заменены только в местах, где параметр или ссылка на столбец синтаксически допустимы. Как крайний случай, рассмотрим следующий пример плохого стиля программирования:

```
INSERT INTO foo (foo) VALUES (foo);
```

Первый раз `foo` появляется на том месте, где синтаксически должно быть имя таблицы, поэтому замены не будет, даже если функция имеет переменную `foo`. Второй раз `foo` встречается там, где должно быть имя столбца таблицы, поэтому замены не будет и здесь. Только третье вхождение `foo` является кандидатом на то, чтобы быть ссылкой на переменную функции.

> **Замечание:** Версии PostgreSQL до 9.0 пытаются заменить переменную во всех трех случаях, что приводит к синтаксической ошибке.

Если имена переменных синтаксически не отличаются от названий столбцов таблицы, то возможна двусмысленность и в ссылках на таблицы. Является ли данное имя ссылкой на столбец таблицы или ссылкой на переменную? Изменим предыдущий пример:

```
INSERT INTO dest (col) SELECT foo + bar FROM src;
```

Здесь `dest` и `src` должны быть именами таблиц, `col` должен быть столбцом `dest`. Однако, `foo` и `bar` могут быть как переменными функции, так и столбцами `src`.

По умолчанию, PL/pgSQL выдаст ошибку, если имя в операторе SQL может относиться как к переменной, так и к столбцу таблицы. Ситуацию можно исправить переименованием переменной, переименованием столбца, точной квалификацией неоднозначной ссылки или указанием PL/pgSQL машине, какую интерпретацию предпочесть.

Самое простое решение - переименовать переменную или столбец. Общее правило кодирования предполагает использование различных соглашений о наименовании для переменных PL/pgSQL и столбцов таблиц. Например, если имена переменных всегда имеют вид `v_`*`something`*, а имена столбцов никогда не начинаются на `v_`, то конфликты исключены.

В качестве альтернативы можно квалифицировать имена неоднозначных ссылок, чтобы сделать их точными. В приведенном выше примере `src.foo` однозначно бы определялась, как ссылка на столбец таблицы. Чтобы сделать однозначный ссылку на переменную, переменная должна быть объявлена в блоке с меткой, и далее нужно использовать эту метку. (смотри Разд. 40.2). Например:

```
<<block>>
DECLARE
    foo int;
BEGIN
    foo := ...;
    INSERT INTO dest (col) SELECT block.foo + bar FROM src;
```

Здесь `block.foo` ссылается на переменную, даже если в таблице `src` есть колонка `foo`. Параметры функции, а также специальные переменные, такие как `FOUND`, могут быть квалифицированы по имени функции, потому что они неявно объявлены во внешнем блоке, метка которого совпадает с именем функции.

Иногда может быть не очень практичным исправлять таким способом все неоднозначные ссылки в большом куске PL/pgSQL кода. В таких случаях можно указать, чтобы PL/pgSQL разрешал неоднозначные ссылки в пользу переменных (это совместимо с PL/pgSQL до версии PostgreSQL 9.0), или в пользу столбцов таблицы (совместимо с некоторыми другими системами, такими как Oracle).

На уровне всей системы поведение PL/pgSQL регулируется установкой конфигурационного параметра `plpgsql.variable_conflict`, имеющего значения: `error`, `use_variable` или `use_column` (`error` устанавливается по умолчанию при установке системы). Изменение этого параметра влияет на все последующие компиляции операторов в PL/pgSQL функциях, но не на операторы уже скомпилированные в текущей сессии. Так как изменение этого параметра может привести к неожиданным изменениям в поведении PL/pgSQL функций, он может быть изменен только суперпользователем.

Поведение PL/pgSQL можно изменять для каждой отдельной функции, если добавить в начало функции одну из этих специальных команд:

```
#variable_conflict error
#variable_conflict use_variable
```

```
#variable_conflict use_column
```

Эти команды влияют только на функцию, в которой они записаны и перекрывают действие `plpgsql.variable_conflict`. Пример:

```
CREATE FUNCTION stamp_user(id int, comment text) RETURNS void AS $$
    #variable_conflict use_variable
    DECLARE
        curtime timestamp := now();
    BEGIN
        UPDATE users SET last_modified = curtime, comment = comment
          WHERE users.id = id;
    END;
$$ LANGUAGE plpgsql;
```

В команде `UPDATE`, `curtime`, `comment` и `id` будут ссылаться на переменные и параметры функции вне зависимости от того, есть ли столбцы с такими именами в таблице `users`. Обратите внимание, что нужно квалифицировать именем таблицы ссылку на `users.id` в предложении `WHERE`, чтобы она ссылалась на столбец таблицы. При этом необязательно квалифицировать ссылку на `comment` в левой части списка `UPDATE`, т.к. синтаксически в этом месте должно быть имя столбца таблицы `users`. Эту функцию можно было бы записать и без зависимости от значения `variable_conflict`:

```
CREATE FUNCTION stamp_user(id int, comment text) RETURNS void AS $$
    <<fn>>
    DECLARE
        curtime timestamp := now();
    BEGIN
        UPDATE users SET last_modified = fn.curtime, comment =
stamp_user.comment
          WHERE users.id = stamp_user.id;
    END;
$$ LANGUAGE plpgsql;
```

Замена переменных не происходит в строке, исполняемой командой `EXECUTE` или её вариантом. Если нужно вставлять изменяющиеся значения в такую команду, то это делается либо при построении самой командной строки или с использованием `USING`, как показано в Разд. 40.5.4.

Замена переменных в настоящее время работает только в командах `SELECT`, `INSERT`, `UPDATE` и `DELETE`, потому что основная SQL машина допускает использование параметров запроса только в этих командах. Чтобы использовать изменяемые имена или значения в других типах операторов (обычно называются утилиты), необходимо построить текст команды в виде строки и выполнить её в `EXECUTE`.

40.10.2. Кэширование плана

Интерпретатор PL/pgSQL анализирует исходный текст функции и строит внутреннее бинарное дерево инструкций при первом вызове функции (для каждой сессии). В дерево инструкций полностью переводится вся структура операторов PL/pgSQL, но для выражений

и команд SQL, используемых в функции, это происходит не сразу.

При первом выполнении в функции каждого выражения или команды SQL интерпретатор PL/pgSQL разбирает и анализирует команду для создания подготовленного к выполнению оператора с помощью функции `SPI_prepare` менеджера интерфейса программирования сервера. Последующие обращения к этому выражению или команде повторно используют подготовленный к выполнению оператор. Таким образом, SQL команды, находящиеся в редко посещаемой ветке кода условного оператора, не несут накладных расходов на разбор команд, если они так и не будут выполнены в текущей сессии. Здесь есть недостаток, заключающийся в том, что ошибки в определенном выражении или команде не могут быть обнаружены, пока выполнение не дойдет до этой части функции. (Тривиальные синтаксические ошибки обнаружатся в ходе первоначального разбора, но ничего более серьезного не будет обнаружено до исполнения.)

Кроме того, PL/pgSQL (точнее, менеджер интерфейса программирования сервера) будет пытаться кэшировать план выполнения для любого подготовленного к исполнению оператора. При каждом вызове оператора, если не используется план из кэша, генерируется новый план выполнения, и текущие значения параметров (то есть значения переменных PL/pgSQL) могут быть использованы для оптимизации нового плана. Если оператор не имеет параметров или выполняется много раз, менеджер интерфейса программирования сервера рассмотрит вопрос о создании и кэшировании (для повторного использования) общего плана, не зависящего от значений параметров. Как правило, это происходит в тех случаях, когда план выполнения не очень чувствителен к имеющимся ссылкам на значения переменных PL/pgSQL. В противном случае, выгоднее каждый раз формировать новый план. Смотри PREPARE для более подробной информации об операторах, подготовленных к выполнению.

Чтобы PL/pgSQL мог сохранять подготовленные операторы и планы выполнения, команды SQL, находящиеся в PL/pgSQL функции, должны использовать одни и те же таблицы и столбцы при каждом исполнении. А это значит, что в SQL командах нельзя использовать названия таблиц и столбцов в качестве параметров. Чтобы обойти это ограничение, нужно построить динамическую команду для PL/pgSQL оператора `EXECUTE` — ценой будет разбор и построение нового плана выполнения при каждом вызове.

Изменчивая природа переменных типа `record` представляет еще одну проблему в этой связи. Когда поля переменной типа `record` используются в выражениях или операторах, типы данных полей не должны меняться от одного вызова функции к другому, так как при анализе каждого выражения будет использоваться тот тип данных, который присутствовал при первом вызове. При необходимости можно использовать `EXECUTE` для решения этой проблемы.

Если функция используется в качестве триггера более чем для одной таблицы, PL/pgSQL независимо подготавливает и кэширует операторы для каждой такой таблицы. То есть создается кэш для каждой комбинации триггерная функция + таблица, а не только для каждой функции. Это устраняет некоторые проблемы, связанные с различными типами

данных. Например, триггерная функция сможет успешно работать со столбцом key, даже если в разных таблицах этот столбец имеет разные типы данных.

Таким же образом, функции с полиморфными типами аргументов имеют отдельный кэш для каждой комбинации фактических типов аргументов, так что различия типов данных не вызывают неожиданных сбоев.

Кэширование операторов иногда приводит к неожиданным эффектам при интерпретации чувствительных ко времени значений. Например, есть разница между тем, что делают эти две функции:

```
CREATE FUNCTION logfunc1(logtxt text) RETURNS void AS $$
    BEGIN
        INSERT INTO logtable VALUES (logtxt, 'now');
    END;
$$ LANGUAGE plpgsql;
```

и

```
CREATE FUNCTION logfunc2(logtxt text) RETURNS void AS $$
    DECLARE
        curtime timestamp;
    BEGIN
        curtime := 'now';
        INSERT INTO logtable VALUES (logtxt, curtime);
    END;
$$ LANGUAGE plpgsql;
```

В случае logfunc1, при анализе INSERT, основной парсер PostgreSQL знает, что строку 'now' следует толковать как timestamp, потому что целевой столбец таблицы logtable имеет такой тип данных. Таким образом, 'now' будет преобразовано в константу timestamp при анализе INSERT, а затем эта константа будет использоваться в последующих вызовах logfunc1 в течение всей сессии. Разумеется, это не то, что хотел программист. Лучше было бы использовать функцию now() или current_timestamp.

В случае logfunc2, основной парсер PostgreSQL не знает, какого типа будет 'now' и поэтому возвращает значение типа text, содержащее строку now. При последующем присвоении локальной переменной curtime интерпретатор PL/pgSQL преобразовывает эту строку к типу timestamp, вызывая функции text_out и timestamp_in. Таким образом, метка времени будет обновляться при каждом выполнении, как и ожидается программистом. И хотя всё работает как ожидалось, это ужасно неэффективно, поэтому использование функции now() по-прежнему значительно лучше.

40.11. Советы по разработке на PL/pgSQL

Хороший способ разрабатывать на PL/pgSQL заключается в том, чтобы в одном окне с текстовым редактором по выбору создавать тексты функций, а в другом окне с psql загружать и тестировать эти функции. В таком случае удобно записывать функцию, используя CREATE

OR REPLACE FUNCTION. Таким образом, можно легко загрузить файл для обновления определения функции. Например:

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $$
          ....
$$ LANGUAGE plpgsql;
```

В psql, можно загрузить или перезагрузить такой файл определения функции, выполнив:

```
\i filename.sql
```

а затем сразу выполнять команды SQL для тестирования функции.

Еще один хороший способ разрабатывать на PL/pgSQL связан с использованием GUI инструментов, облегчающих разработку на процедурном языке. Один из примеров такого инструмента pgAdmin, хотя есть и другие. Такие инструменты часто предоставляют удобные возможности, такие как экранирование одинарных кавычек, отладка и повторное создание функций.

40.11.1. Обработка кавычек

Код PL/pgSQL функции указывается в команде CREATE FUNCTION в виде строки. Если писать строку в обычном порядке, внутри одинарных кавычек, то любой символ одинарной кавычки должен быть удвоен, также как и должен быть удвоен каждый знак обратной косой черты (если используется синтаксис с экранированием в строках). Удвоение кавычек в лучшем случае утомительно, а в более сложных случаях код может стать совершенно непонятным, так как легко может потребоваться полудюжина или более кавычек идущих подряд. Вместо этого при создании тела функции рекомендуется использовать знаки доллара в качестве кавычек (смотри Разд. 4.1.2.4). При таком подходе никогда не потребуется дублировать кавычки, но придется позаботиться о том, чтобы иметь разные долларовые разделители для каждого уровня вложенности. Например, команду CREATE FUNCTION можно записать так:

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $PROC$
          ....
$PROC$ LANGUAGE plpgsql;
```

Внутри можно использовать кавычки для простых текстовых строк и $$ для разграничения фрагментов SQL команды, собираемой из отдельных строк. Если нужно взять в кавычки текст, который включает $$, можно использовать Q, и так далее.

Следующая таблица показывает, как применяются знаки кавычек, если не используется экранирование долларами. Это может быть полезно при переводе кода, не использующего экранирование знаками доллара, в нечто более понятное.

1 кавычка

В начале и конце тела функции, например:

```
CREATE FUNCTION foo() RETURNS integer AS '
        ....
' LANGUAGE plpgsql;
```

Внутри такой функции любая кавычка *должна* дублироваться.

2 кавычки

Для строковых литералов внутри тела функции, например:

```
a_output := ''Blah'';
SELECT * FROM users WHERE f_name=''foobar'';
```

При использовании знаков доллара можно просто написать:

```
a_output := 'Blah';
SELECT * FROM users WHERE f_name='foobar';
```

и это именно то, что нужно PL/pgSQL парсеру.

4 кавычки

Когда нужны одинарные кавычки в строковой константе внутри тела функции, например:

```
a_output := a_output || '' AND name LIKE ''''foobar'''' AND xyz''
```

К `a_output` будет добавлено: `AND name LIKE 'foobar' AND xyz`

При использовании знаков доллара это записывается так:

```
a_output := a_output || $$ AND name LIKE 'foobar' AND xyz$$
```

будьте внимательны, при этом не должно быть внешнего долларового разделителя `$$`.

6 кавычек

Когда нужны одинарные кавычки в строковой константе внутри тела функции, при этом кавычки находятся в конце строковой константы. Например:

```
a_output := a_output || '' AND name LIKE ''''foobar''''''
```

К `a_output` будет добавлено: `AND name LIKE 'foobar'`.

При использовании знаков доллара это записывается так:

```
a_output := a_output || $$ AND name LIKE 'foobar'$$
```

10 кавычек

Когда нужны две одиночные кавычки в строковой константе (это уже 8 кавычек),

примыкающие к концу строковой константы (еще 2). Вероятно, такое может понадобиться при разработке функции, которая генерирует другие функции, как в Прим. 40-9. Например:

```
a_output := a_output || '' if v_'' ||
    referrer_keys.kind || '' like ''''''''''
    || referrer_keys.key_string || ''''''''''
    then return ''''''  || referrer_keys.referrer_type
    || ''''''; end if;'';
```

Значение `a_output` затем будет:

```
if v_... like ''...'' then return ''...''; end if;
```

При использовании знаков доллара:

```
a_output := a_output || $$ if v_$$ || referrer_keys.kind || $$ like '$$
    || referrer_keys.key_string || $$'
    then return '$$  || referrer_keys.referrer_type
    || $$'; end if;$$;
```

где предполагается, что нужны только одиночные кавычки в `a_output`, так как потребуется повторное взятие в кавычки перед использованием.

40.11.2. Дополнительные проверки во время компиляции

Чтобы помочь найти и предупредить простые, но часто встречающиеся проблемы, PL/PgSQL предоставляет дополнительные проверки *checks*. Если они включены в конфигурации, то во время компиляции функций будут выдаваться дополнительные сообщения `WARNING` или ошибки `ERROR`. Функция, при компиляции которой выдавалось `WARNING`, при последующем выполнении не будет выдавать это сообщение и её можно протестировать в отдельной среде разработки.

Для включения этих проверок используются параметры конфигурации `plpgsql.extra_warnings` для предупреждений и `plpgsql.extra_errors` для ошибок. Каждому из параметров можно присвоить список значений, разделенный через запятую, значение `"none"` или `"all"`. По умолчанию используется `"none"`. В настоящий момент доступна только одна проверка:

`shadowed_variables`

Проверяет, что объявление новой переменной не скрывает ранее объявленную переменую.

Следующий пример показывает эффект от установки `plpgsql.extra_warnings` в значение `shadowed_variables`:

```
SET plpgsql.extra_warnings TO 'shadowed_variables';

CREATE FUNCTION foo(f1 int) RETURNS int AS $$
```

```
DECLARE
f1 int;
BEGIN
RETURN f1;
END
$$ LANGUAGE plpgsql;
WARNING:  variable "f1" shadows a previously defined variable
LINE 3: f1 int;
        ^
CREATE FUNCTION
```

40.12. Портирование из Oracle PL/SQL

В этом разделе рассматриваются различия между языками PostgreSQL PL/pgSQL и Oracle PL/SQL, чтобы помочь разработчикам, портирующим приложения из Oracle® в PostgreSQL.

PL/pgSQL во многих аспектах похож на PL/SQL . Это блочно-структурированный, императивный язык, в котором все переменные должны объявляться. Присвоения, циклы, условные операторы в обоих языках похожи. Основные отличия, которые необходимо иметь в виду при портировании с PL/SQL в PL/pgSQL, следующие:

- Если имя, используемое в SQL команде, может быть как именем столбца таблицы, так и ссылкой на переменную функции, то PL/SQL считает, что это имя столбца таблицы. Это соответствует поведению PL/pgSQL при `plpgsql.variable_conflict = use_column`, что не является значением по умолчанию, как описано в Разд. 40.10.1. В первую очередь, было бы правильно избегать таких двусмысленностей, но если требуется портировать большое количество кода, зависящее от данного поведения, то установка переменной `variable_conflict` может быть лучшим решением.

- В PostgreSQL тело функции должно быть записано в виде строки. Поэтому нужно использовать знак доллара в качестве кавычек или экранировать одиночные кавычки в теле функции. (Смотри Разд. 40.11.1.)

- Для группировки функций вместо пакетов используются схемы.

- Так как пакетов нет, нет и пакетных переменных. Это несколько раздражает. Вместо этого можно хранить состояние каждого сеанса во временных таблицах.

- Целочисленные циклы `FOR` с опцией `REVERSE` работают по разному. В PL/SQL значение счетчика уменьшается от второго числа к первому, в то время как в PL/pgSQL счетчик уменьшается от первого ко второму. Поэтому при портировании нужно менять местами границы цикла. Это печально, но вряд ли будет изменено. (Смотри Разд. 40.6.3.5.)

- Циклы `FOR` по запросам (не курсорам) также работают по разному. Переменная цикла должна быть объявлена, в то время как в PL/SQL она объявляется неявно. Преимущество в том, что значения переменных доступны и после выхода из цикла.

- Существуют некоторые отличия в нотации при использовании курсорных переменных.

40.12.1. Примеры портирования

Прим. 40-8 показывает, как портировать простую функцию из PL/SQL в PL/pgSQL.

Пример 40-8. Портирование простой функции из PL/SQL в PL/pgSQL

Функция Oracle PL/SQL:

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name varchar,
                                                  v_version varchar)
RETURN varchar IS
BEGIN
    IF v_version IS NULL THEN
        RETURN v_name;
    END IF;
    RETURN v_name || '/' || v_version;
END;
/
show errors;
```

Пройдемся по этой функции и посмотрим различия по сравнению с PL/pgSQL:

- Ключевое слово `RETURN` в прототипе функции (не в теле функции) заменяется на `RETURNS` в PostgreSQL. Кроме того, `IS` становится `AS`, и нужно добавить фразу `LANGUAGE`, потому что PL/pgSQL не единственный возможный язык.
- В PostgreSQL тело функции является строкой, поэтому нужно использовать кавычки или знаки доллара. Это заменяет завершающий `/` в подходе Oracle.
- Команда `show errors` не существует в PostgreSQL и не требуется, так как ошибки будут выводиться автоматически.

Вот как эта функция будет выглядеть после портирования в PostgreSQL:

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name varchar,
                                                  v_version varchar)
RETURNS varchar AS $$
BEGIN
    IF v_version IS NULL THEN
        RETURN v_name;
    END IF;
    RETURN v_name || '/' || v_version;
END;
$$ LANGUAGE plpgsql;
```

Прим. 40-9 показывает, как портировать функцию, которая создает другую функцию, и как обрабатывать проблемы с кавычками.

Пример 40-9. Портирование из PL/SQL to PL/pgSQL функции создающей другую функцию

Следующая процедура получает строки из SELECT и строит большую функцию, в целях эффективности возвращающую результат в операторах IF.

Версия Oracle:

```
CREATE OR REPLACE PROCEDURE cs_update_referrer_type_proc IS
    CURSOR referrer_keys IS
        SELECT * FROM cs_referrer_keys
        ORDER BY try_order;
    func_cmd VARCHAR(4000);
BEGIN
    func_cmd := 'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host IN
VARCHAR,
                v_domain IN VARCHAR, v_url IN VARCHAR) RETURN VARCHAR IS
BEGIN';

    FOR referrer_key IN referrer_keys LOOP
        func_cmd := func_cmd ||
          ' IF v_' || referrer_key.kind
          || ' LIKE ''' || referrer_key.key_string
          || ''' THEN RETURN ''' || referrer_key.referrer_type
          || '''; END IF;';
    END LOOP;

    func_cmd := func_cmd || ' RETURN NULL; END;';

    EXECUTE IMMEDIATE func_cmd;
END;
/
show errors;
```

В конечном итоге в PostgreSQL эта функция может выглядеть так:

```
CREATE OR REPLACE FUNCTION cs_update_referrer_type_proc() RETURNS void AS $func$
DECLARE
    referrer_keys CURSOR IS
        SELECT * FROM cs_referrer_keys
        ORDER BY try_order;
    func_body text;
    func_cmd text;
BEGIN
    func_body := 'BEGIN';

    FOR referrer_key IN referrer_keys LOOP
        func_body := func_body ||
          ' IF v_' || referrer_key.kind
          || ' LIKE ' || quote_literal(referrer_key.key_string)
          || ' THEN RETURN ' || quote_literal(referrer_key.referrer_type)
          || '; END IF;' ;
    END LOOP;

    func_body := func_body || ' RETURN NULL; END;';

    func_cmd :=
      'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host varchar,
                                                        v_domain varchar,
                                                        v_url varchar)
        RETURNS varchar AS '
      || quote_literal(func_body)
      || ' LANGUAGE plpgsql;' ;
```

```
    EXECUTE func_cmd;
END;
$func$ LANGUAGE plpgsql;
```

Обратите внимание, что тело функции строится отдельно, с использованием `quote_literal` для удвоения кавычек. Эта техника необходима, потому что мы не можем безопасно использовать знаки доллара при определении новой функции: мы не знаем наверняка, какие строки будут вставлены из `referrer_key.key_string`. (Мы предполагаем, что `referrer_key.kind` всегда имеет значение из списка: `host`, `domain` или `url`, но `referrer_key.key_string` может быть чем угодно, в частности, может содержать знаки доллара.) На самом деле, в этой функций есть улучшение по сравнению с оригиналом Oracle, потому что не будет генерироваться неправильный код, когда `referrer_key.key_string` или `referrer_key.referrer_type` содержат кавычки.

Прим. 40-10 показывает, как портировать функцию с `OUT` параметрами и манипулирующую строками. PostgreSQL не имеет встроенной функции `instr`, но её можно создать, используя комбинацию других функций. Разд. 40.12.3 содержит реализацию `instr` в PL/pgSQL, которую можно использовать для облегчения портирования.

Пример 40-10. Портирование из PL/SQL в PL/pgSQL процедуры, которая манипулирует строками и содержит `OUT` параметры

Следующая Oracle PL/SQL процедура используется для разбора URL и возвращения несколько элементов (хост, путь и запрос).

Версия Oracle:

```
CREATE OR REPLACE PROCEDURE cs_parse_url(
    v_url IN VARCHAR,
    v_host OUT VARCHAR,  -- Возвращается обратно
    v_path OUT VARCHAR,  -- И это возвращается
    v_query OUT VARCHAR) -- И это
IS
    a_pos1 INTEGER;
    a_pos2 INTEGER;
BEGIN
    v_host := NULL;
    v_path := NULL;
    v_query := NULL;
    a_pos1 := instr(v_url, '//');

    IF a_pos1 = 0 THEN
        RETURN;
    END IF;
    a_pos2 := instr(v_url, '/', a_pos1 + 2);
    IF a_pos2 = 0 THEN
        v_host := substr(v_url, a_pos1 + 2);
        v_path := '/';
        RETURN;
    END IF;

    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
    a_pos1 := instr(v_url, '?', a_pos2 + 1);
```

```
    IF a_pos1 = 0 THEN
        v_path := substr(v_url, a_pos2);
        RETURN;
    END IF;

    v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
    v_query := substr(v_url, a_pos1 + 1);
END;
/
show errors;
```

Вот возможная трансляция в PL/pgSQL:

```
CREATE OR REPLACE FUNCTION cs_parse_url(
    v_url IN VARCHAR,
    v_host OUT VARCHAR,  -- Возвращается обратно
    v_path OUT VARCHAR,  -- И это возвращается
    v_query OUT VARCHAR) -- И это
AS $$
DECLARE
    a_pos1 INTEGER;
    a_pos2 INTEGER;
BEGIN
    v_host := NULL;
    v_path := NULL;
    v_query := NULL;
    a_pos1 := instr(v_url, '//');

    IF a_pos1 = 0 THEN
        RETURN;
    END IF;
    a_pos2 := instr(v_url, '/', a_pos1 + 2);
    IF a_pos2 = 0 THEN
        v_host := substr(v_url, a_pos1 + 2);
        v_path := '/';
        RETURN;
    END IF;

    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
    a_pos1 := instr(v_url, '?', a_pos2 + 1);

    IF a_pos1 = 0 THEN
        v_path := substr(v_url, a_pos2);
        RETURN;
    END IF;

    v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
    v_query := substr(v_url, a_pos1 + 1);
END;
$$ LANGUAGE plpgsql;
```

Эту функцию можно использовать так:

```
SELECT * FROM cs_parse_url('http://foobar.com/query.cgi?baz');
```

Прим. 40-11 показывает, как портировать процедуру, использующую большое количество специфических для Oracle возможностей.

Пример 40-11. Портирование процедуры из PL/SQL в PL/pgSQL

Версия Oracle:

```
CREATE OR REPLACE PROCEDURE cs_create_job(v_job_id IN INTEGER) IS
    a_running_job_count INTEGER;
    PRAGMA AUTONOMOUS_TRANSACTION;(1)
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE;(2)

    SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS
NULL;

    IF a_running_job_count > 0 THEN
        COMMIT; -- free lock(3)
        raise_application_error(-20000,
                'Не могу создать новое задание. Задание сейчас выполняется.');
    END IF;

    DELETE FROM cs_active_job;
    INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

    BEGIN
        INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, sysdate);
    EXCEPTION
        WHEN dup_val_on_index THEN NULL; -- don't worry if it already exists
    END;
    COMMIT;
END;
/
show errors
```

Подобные процедуры легко конвертируются в функции PostgreSQL, возвращающие `void`. На примере этой процедуры можно научиться следующему:

(1)
: В PostgreSQL нет оператора `PRAGMA`.

(2)
: Если выполнить `LOCK TABLE` в PL/pgSQL, блокировка не будет снята, пока не завершится вызывающая транзакция.

(3)
: В PL/pgSQL функции нельзя использовать `COMMIT`. Функция работает в рамках некоторой внешней транзакции, и поэтому `COMMIT` будет означать прекращение выполнения функции. Однако, в данном конкретном случае, в этом нет необходимости, потому что блокировка, полученная командой `LOCK TABLE`, будет снята при вызове ошибки.

В PL/pgSQL эту процедуру можно портировать так:

```
CREATE OR REPLACE FUNCTION cs_create_job(v_job_id integer) RETURNS void AS $$
DECLARE
    a_running_job_count integer;
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE;

    SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS
NULL;

    IF a_running_job_count > 0 THEN
```

```
        RAISE EXCEPTION 'Не могу создать новое задание. Задание сейчас
выполняется.';(1)
    END IF;

    DELETE FROM cs_active_job;
    INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

    BEGIN
        INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, now());
    EXCEPTION
        WHEN unique_violation THEN (2)
            -- don't worry if it already exists
    END;
END;
$$ LANGUAGE plpgsql;
```

(1)

Синтаксис `RAISE` существенно отличается от Oracle, хотя основной вариант `RAISE` *`exception_name`* работает похоже.

(2)

Имена исключений, поддерживаемые PL/pgSQL, отличаются от исключений в Oracle. Количество встроенных имен исключений значительно больше (смотри Прил. A). В настоящее время нет способа задать пользовательское имя исключения, хотя вместо этого можно вызывать ошибку с заданным пользователем значением SQLSTATE.

Основное функциональное отличие между этой процедурой и Oracle эквивалента в том, что монопольная блокировка таблицы `cs_jobs` будет продолжаться до окончания вызывающей транзакции. Кроме того, если в последствии работа вызывающей программы прервется (например из-за ошибки), произойдет откат всех действий, выполненных в этой процедуре.

40.12.2. На что еще обратить внимание

В этом разделе рассматриваются еще несколько вещей, на которые нужно обращать внимание при портировании функций из Oracle PL/SQL в PostgreSQL.

40.12.2.1. Неявный откат изменений после возникновения исключения

В PL/pgSQL при перехвате исключения в секции `EXCEPTION` все изменения в базе данных с начала блока автоматически откатываются. В Oracle это эквивалентно следующему:

```
BEGIN
    SAVEPOINT s1;
    ... здесь код ...
EXCEPTION
    WHEN ... THEN
        ROLLBACK TO s1;
        ... здесь код ...
    WHEN ... THEN
        ROLLBACK TO s1;
        ... здесь код ...
END;
```

При портировании процедуры Oracle, которая использует `SAVEPOINT` и `ROLLBACK TO` в

таком же стиле, задача простая: достаточно убрать операторы SAVEPOINT и ROLLBACK TO. Если же SAVEPOINT и ROLLBACK TO используются по-другому, то придется подумать.

40.12.2.2. EXECUTE

PL/pgSQL версия EXECUTE работает аналогично версии в PL/SQL, но нужно помнить об использовании quote_literal и quote_ident, как описано в Разд. 40.5.4. Без использования этих функций конструкции типа EXECUTE 'SELECT * FROM $1'; будут работать ненадежно.

40.12.2.3. Оптимизация PL/pgSQL функций

Для оптимизации исполнения PostgreSQL предоставляет два модификатора при создании функции: "волатильность" (будет ли функция всегда возвращать тот же результат при тех же аргументах) и "строгость" (возвращает ли функция NULL, если хотя бы один из аргументов NULL). Для получения подробной информации обратитесь к справочной странице CREATE FUNCTION.

При использовании этих атрибутов оптимизации оператор CREATE FUNCTION может выглядеть примерно так:

```
CREATE FUNCTION foo(...) RETURNS integer AS $$
...
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```

40.12.3. Приложение

Этот раздел содержит код для совместимых с Oracle функций instr, которые можно использовать для упрощения портирования.

```
--
-- instr functions that mimic Oracle's counterpart
-- Syntax: instr(string1, string2, [n], [m]) where [] denotes optional
parameters.
--
-- Searches string1 beginning at the nth character for the mth occurrence
-- of string2.  If n is negative, search backwards.  If m is not passed,
-- assume 1 (search starts at first character).
--

CREATE FUNCTION instr(varchar, varchar) RETURNS integer AS $$
DECLARE
    pos integer;
BEGIN
    pos:= instr($1, $2, 1);
    RETURN pos;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;


CREATE FUNCTION instr(string varchar, string_to_search varchar, beg_index
integer)
RETURNS integer AS $$
```

```
DECLARE
    pos integer NOT NULL DEFAULT 0;
    temp_str varchar;
    beg integer;
    length integer;
    ss_length integer;
BEGIN
    IF beg_index > 0 THEN
        temp_str := substring(string FROM beg_index);
        pos := position(string_to_search IN temp_str);

        IF pos = 0 THEN
            RETURN 0;
        ELSE
            RETURN pos + beg_index - 1;
        END IF;
    ELSIF beg_index < 0 THEN
        ss_length := char_length(string_to_search);
        length := char_length(string);
        beg := length + beg_index - ss_length + 2;

        WHILE beg > 0 LOOP
            temp_str := substring(string FROM beg FOR ss_length);
            pos := position(string_to_search IN temp_str);

            IF pos > 0 THEN
                RETURN beg;
            END IF;

            beg := beg - 1;
        END LOOP;

        RETURN 0;
    ELSE
        RETURN 0;
    END IF;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;


CREATE FUNCTION instr(string varchar, string_to_search varchar,
                      beg_index integer, occur_index integer)
RETURNS integer AS $$
DECLARE
    pos integer NOT NULL DEFAULT 0;
    occur_number integer NOT NULL DEFAULT 0;
    temp_str varchar;
    beg integer;
    i integer;
    length integer;
    ss_length integer;
BEGIN
    IF beg_index > 0 THEN
        beg := beg_index;
        temp_str := substring(string FROM beg_index);

        FOR i IN 1..occur_index LOOP
            pos := position(string_to_search IN temp_str);

            IF i = 1 THEN
                beg := beg + pos - 1;
```

```
            ELSE
                beg := beg + pos;
            END IF;

            temp_str := substring(string FROM beg + 1);
        END LOOP;

        IF pos = 0 THEN
            RETURN 0;
        ELSE
            RETURN beg;
        END IF;
    ELSIF beg_index < 0 THEN
        ss_length := char_length(string_to_search);
        length := char_length(string);
        beg := length + beg_index - ss_length + 2;

        WHILE beg > 0 LOOP
            temp_str := substring(string FROM beg FOR ss_length);
            pos := position(string_to_search IN temp_str);

            IF pos > 0 THEN
                occur_number := occur_number + 1;

                IF occur_number = occur_index THEN
                    RETURN beg;
                END IF;
            END IF;

            beg := beg - 1;
        END LOOP;

        RETURN 0;
    ELSE
        RETURN 0;
    END IF;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```