**Тема 7. Язык запросов SQL. Курсоры и триггеры. Транзакции.**

**Курсоры**

Курсор в SQL – это временная выборка записей в процессе выполнения функции, над которой могут выполняться необходимые Вам действия. Данная выборка является указателем на область памяти. Курсоры могут быть очень полезны, например, если Вам в функции необходимо выполнять определенные действия с каким то набором строк, при этом до начала выполнения функции Вы даже не знаете, сколько строк будет при обработке той или иной записи. Если проще курсор — это просто запрос, который запускается в процессе выполнения функции. Например, у Вас есть определенный запрос (набор записей), над каждой строкой которого необходимо выполнять какие то хитрые действия, которые нужно запомнить для операции над следующей строкой этого же запроса.

О том, что курсоры могут быть полезны, мы поговорили, но когда их лучше использовать? А использовать их лучше всего только тогда, когда у Вас нет другого выхода! Потому что курсор является очень ресурсоемким решением. Если Вы будете выполнять операции над небольшим количеством записей, то это приемлемо, а если необходимо обработать большой объем данных, то Вы можете очень долго ждать, пока будет выполняться Ваша функция, а как Вы знаете быстрота в нашем деле чуть ли не главный фактор.

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.

Общий синтаксис курсора выглядит следующим образом:

select public.cursorfuc()

CREATE OR REPLACE FUNCTION название функции(типы переменных)

RETURNS тип возвращаемого значения AS

$BODY$

DECLARE

объявление переменных

объявление курсора

BEGIN

открытие курсора

перебор данных и операции над ними

закрытие курсора

RETURN возвращение значения;

END;

$BODY$

LANGUAGE 'plpgsql' VOLATILE

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:

имя [ [ NO ] SCROLL ] CURSOR [ ( аргументы ) ] FOR запрос;

С указанием SCROLL 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 аргументы, то они должны представлять собой пары имя тип\_данных, разделённые через запятую. Эти пары определяют имена, которые будут заменены значениями параметров в данном запросе. Фактические значения для замены этих имён 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 заменён целочисленным значением параметра при открытии курсора.) Про переменную curs1 говорят, что она является несвязанной (unbound), так как к ней не привязан никакой запрос.

Прежде чем получать строки из курсора, его нужно открыть.В PL/pgSQL есть несколько форм оператора OPEN, которых используются для связанных и несвязанных курсорных переменных.

OPEN несвязанная\_переменная\_курсора [[NO] SCROLL] FOR запрос;

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

ОPEN связанная\_переменная\_курсора [( [имя\_аргумента :=] значение\_аргумента [, ...] )];

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.

OPEN curs3(key := 42);

Пример:

CREATE OR REPLACE FUNCTION public.cursorfuc() RETURNS refcursor AS

$BODY$

DECLARE

curs1 refcursor;

curs2 CURSOR FOR SELECT \* FROM tenk1;

curs3 CURSOR (key integer) FOR SELECT \* FROM tenk1 WHERE unique1 = key;

BEGIN

OPEN curs1 FOR SELECT \* FROm t1;

OPEN curs3 (key := 42);

return curs1;

END;

$BODY$

LANGUAGE plpgsql;

После того как курсор будет открыт, с ним можно работать при помощи операторов про которые мы сегодня поговорим. Работать с курсором не обязательно в той же функции, где он был открыт. Из функции можно вернуть значение с типом refcursor, что позволит вызывающему продолжить работу с курсором.

Команда FETCH:

FETCH [направление { FROM | IN }] курсор INTO цель;

FETCH извлекает следующую строку из курсора в цель. В качестве цели может быть строковая переменная, переменная типа record, или разделённый запятыми список простых переменных, как и в SELECT INTO. Если следующей строки нет, цели присваивается NULL.

FETCH curs1 INTO rowvar;

FETCH curs2 INTO foo, bar, baz;

FETCH LAST FROM curs3 INTO x, y;

FETCH RELATIVE -2 FROM curs4 INTO x;

Команда MOVE:

MOVE [направление { FROM | IN }] курсор;

MOVE перемещает курсор без извлечения данных. MOVE работает точно так же как и FETCH, но при этом только перемещает курсор и не извлекает строку, к которой переместился.

MOVE curs1;

MOVE LAST FROM curs3;

MOVE RELATIVE -2 FROM curs4;

MOVE FORWARD 2 FROM curs4;

Команда UPDATE/DELETE WHERE CURRENT OF:

UPDATE таблица SET ... WHERE CURRENT OF курсор;

DELETE FROM таблица WHERE CURRENT OF курсор;

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).

Пример:

UPDATE foo SET dataval = myval WHERE CURRENT OF curs1;

Команда CLOSE курсор;

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

Пример:

CLOSE curs1;

Рассмотрим конкретный пример работы с курсором. Для начала давайте определим, с какими данными мы будем работать, допустим, у нас есть таблица, в которой хранятся записи о сотрудниках, которые постоянно находятся в командировках, или просто у них есть служебный транспорт, да это и не важно, важно то, что им выделяют деньги на расходы, а они отчитываются по этим расходам.

id\_per – период по которому идет отчет;

id\_user – идентификатор сотрудника;

rashod – сумма расходов за этот период;

summa – сумма, которая выделялась на расходы;

pr – возможная премия, на погашение задолженности в прошлом периоде.

CREATE TABLE testTable (

id\_per int,

id\_user int,

rashod int,

summa int,

pr int);

INSERT INTO testTable VALUES(1, 1, 100, 100, 0);

INSERT INTO testTable VALUES(1, 2, 100, 110, 0);

INSERT INTO testTable VALUES(2, 1, 100, 90, 0);

INSERT INTO testTable VALUES(2, 2, 100, 90, 0);

INSERT INTO testTable VALUES(3, 1, 110, 100, 0);

INSERT INTO testTable VALUES(3, 2, 100, 100, 10);

Стоит следующая задача: нам необходимо определить первый период у сотрудников, в котором они расходовали средств больше, чем им выдали, в случае если в следующем периоде эти средства не были возмещены. Рассмотрим функцию, реализующую требуемый функционал при помощи курсоров:

CREATE OR REPLACE FUNCTION public.my\_fun(numeric) RETURNS numeric AS

$BODY$

DECLARE

\_id\_user ALIAS FOR $1;

--объявляем курсор

crs\_my CURSOR FOR select id\_per, rashod, summa from public.testTable where

id\_user = \_id\_user order by id\_per;

--объявляем нужные нам переменные

\_id\_per numeric;

\_rashod numeric;

\_summa numeric;

\_pr numeric;

\_var numeric;

\_rezult numeric;

BEGIN

\_pr := 0;

OPEN crs\_my;--открываем курсор

LOOP --начинаем цикл по курсору

--извлекаем данные из строки и записываем их в переменные

FETCH crs\_my INTO \_id\_per, \_rashod, \_summa;

--если такого периода и не возникнет, то мы выходим

IF NOT FOUND THEN EXIT; END IF;

--ищем сумму возмещения, если она была

select into \_pr pr from public.testTable where id\_user =\_id\_user and id\_per = \_id\_per+1;

RAISE NOTICE 'var= %', \_var;

\_var = \_rashod - \_summa;

if \_var > 0 then

\_var = \_var - \_pr;

if(\_var > 0) then

\_rezult =\_id\_per;

End if;

End if;

--если \_var даже после возмещения больше нуля, то выходим и возвращаем период

EXIT when \_var > 0;

END LOOP;--заканчиваем цикл по курсору

CLOSE crs\_my; --закрываем курсор

RETURN \_rezult;--возвращаем результат

END;

$BODY$

LANGUAGE 'plpgsql' VOLATILE

SELECT public.my\_fun(1)

**Триггеры**

В PL/pgSQL можно создавать триггерные процедуры, которые будут вызываться при изменениях данных или событиях в базе данных. Триггерная процедура создаётся командой CREATE FUNCTION, при этом у функции не должно быть аргументов, а типом возвращаемого значения должен быть trigger (для триггеров, срабатывающих при изменениях данных) или event\_trigger (для триггеров, срабатывающих при событиях в базе). Для триггеров автоматически определяются специальные локальные переменные с именами вида TG\_имя, описывающие условие, повлёкшее вызов триггера.

Когда функция на PL/pgSQL 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\_OP**. Тип данных text. Строка, содержащая INSERT, UPDATE, DELETE или TRUNCATE, в зависимости от того, для какой операции сработал триггер.
* **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.

Рассмотрим конкретную задачу - реализовать самую простую систему логирования пользователей. Она будет следить за изменениями в таблице пользователей и при изменениях добавлять текстовые действия в таблицу логов.

Создадим две таблицы:

CREATE TABLE users

(

"name" text

)

CREATE TABLE logs

(

"text" text,

"added" timestamp without time zone

)

При изменении данных в таблице users должен срабатывать триггер, который будет заполнять данные в таблице logs:

CREATE TRIGGER t\_user

AFTER INSERT OR UPDATE OR DELETE ON users FOR EACH ROW EXECUTE PROCEDURE add\_to\_log ();

CREATE OR REPLACE FUNCTION add\_to\_log() RETURNS TRIGGER AS $$

DECLARE

mstr varchar(30);

astr varchar(100);

retstr varchar(254);

BEGIN

IF TG\_OP = 'INSERT' THEN

astr = NEW.name;

mstr := 'Add new user ';

retstr := mstr || astr;

INSERT INTO logs(text,added) values (retstr,NOW());

RETURN NEW;

ELSIF TG\_OP = 'UPDATE' THEN

astr = NEW.name;

mstr := 'Update user ';

retstr := mstr || OLD.name || ' ON ' || astr;

INSERT INTO logs(text,added) values (retstr,NOW());

RETURN NEW;

ELSIF TG\_OP = 'DELETE' THEN

astr = OLD.name;

mstr := 'Remove user ';

retstr := mstr || astr;

INSERT INTO logs(text,added) values (retstr,NOW());

RETURN OLD;

END IF;

END;

$$ LANGUAGE plpgsql;

Что тут происходит ? Собственно определяется новая функция, без входящих параметров, возвращает специальный тип TRIGGER. Для внутреннего использования определяем 3-и переменные в разделе DECLARE. В самой процедуре смотрим переменную TG\_OP (внутренняя переменная триггера которая определяет с какой операцией была вызвана процедура). В зависимости от операции определяем переменную mstr, собираем строку retstr которая будет записана в базу данных (обратите внимание как производится конкатенация строк в pgsql, через ||) и собственно делаем запись в таблицу логов (INSERT INTO). Надо пояснить что такое переменные NEW и OLD. Это собственно строки которые обрабатывает триггер. В случае INSERT переменная NEW будет содержать новую строку, а OLD будет пустая, в случае UPDATE обе переменные будут определены (соответствующими данными), а в случае DELETE переменная NEW будет пустая, OLD содержать удаляемую строку.

**Транзакции**

Транзакции — это фундаментальное понятие во всех СУБД. Суть транзакции в том, что она объединяет последовательность действий в одну операцию "всё или ничего". Промежуточные состояния внутри последовательности не видны другим транзакциям, и если что-то помешает успешно завершить транзакцию, ни один из результатов этих действий не сохранится в базе данных.

Рассмотрим пример банковской операции:

CREATE TABLE Account (

id int,

amount int

)

INSERT INTO Account VALUES(1, 100);

INSERT INTO Account VALUES(2, 200);

UPDATE Account SET amount = amount + 10 WHERE id = 2;

UPDATE Account SET amount = amount - 10 WHERE id = 1;

Точное содержание команд здесь не важно, важно лишь то, что для выполнения этой довольно простой операции потребовалось несколько отдельных действий. При этом с точки зрения банка необходимо, чтобы все эти действия выполнились вместе, либо не выполнились совсем. Если 1 получит 10 долларов, но они не будут списаны со счёта 2, объяснить это сбоем системы определённо не удастся. И наоборот, 2 вряд ли будет доволен, если переведет деньги, а до 1 они не дойдут. Нам нужна гарантия, что если что-то помешает выполнить операцию до конца, ни одно из действий не оставит следа в базе данных. И мы получаем эту гарантию, объединяя действия в одну транзакцию. Говорят, что транзакция атомарна: с точки зрения других транзакций она либо выполняется и фиксируется полностью, либо не фиксируется совсем.

Для объединения операций в транзакции используют следующий синтаксис:

BEGIN;

UPDATE Account SET amount = amount + 10 WHERE id = 2;

UPDATE Account SET amount = amount - 10 WHERE id = 1;

COMMIT;

Теперь, если какой-либо из скриптов не выполнится, результаты выполнения остальных скриптов не будут сохранены в базу данных.

Нам также нужна гарантия, что после завершения и подтверждения транзакции системой баз данных, её результаты в самом деле сохраняются и не будут потеряны, даже если вскоре произойдёт авария. Например, если мы списали сумму и выдали её 1, мы должны исключить возможность того, что сумма на его счёте восстановится, как только он выйдет за двери банка. Транзакционная база данных гарантирует, что все изменения записываются в постоянное хранилище (например, на диск) до того, как транзакция будет считаться завершенной.

Другая важная характеристика транзакционных баз данных тесно связана с атомарностью изменений: когда одновременно выполняется множество транзакций, каждая из них не видит незавершённые изменения, произведённые другими. Например, если одна транзакция подсчитывает баланс по отделениям, будет неправильно, если она посчитает расход в отделении 2, но не учтёт приход в отделении 1, или наоборот. Поэтому свойство транзакций "всё или ничего" должно определять не только, как изменения сохраняются в базе данных, но и как они видны в процессе работы. Изменения, производимые открытой транзакцией, невидимы для других транзакций, пока она не будет завершена, а затем они становятся видны всем и сразу.