Лабораторная работа 4

Цели лабораторной работы:

* Знакомство с языком PL/pgSQL
* Создание процедур, функций и триггеров, обеспечивающих удобство работы с базой и обеспечение целостности
* Использование транзакция для обеспечения целостности

**Содержание**

 [[убрать](http://wiki.cs.hse.ru/%D0%91%D0%B0%D0%B7%D1%8B_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85/%D0%9B%D0%B0%D0%B1%D0%BE%D1%80%D0%B0%D1%82%D0%BE%D1%80%D0%BD%D0%B0%D1%8F_%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%B0_4)]

* 1 Зачем использовать PL/pgSQL
* 2 Процедуры и функции
  + 2.1 Определение переменных
  + 2.2 Управляющие конструкции
  + 2.3 Использование курсоров
  + 2.4 Логирование и обработка ошибок
  + 2.5 Пример полезной для администрирования процедуры
* 3 Триггеры
  + 3.1 Типы триггеров
* 4 Транзакции
  + 4.1 Создание таблицы истории
* 5 Задания лабораторной работы
* 6 Защита лабораторной работы
* 7 Дополнительно

Зачем использовать PL/pgSQL

PL/pgSQL - процедурный язык программирования, позволяющий выполнять несложные скрипты. Язык PL/pgSQL является расширенной версией стандарта языка PL/SQL. В PostgreSQL есть также возможность расширить синтаксис встроенного языка за счет интерпретаторов популярных языков: Python, Perl, Java, Lua, R и тд.

Основное преимущество использования PL/pgSQL состоит в том, что скрипты выполняются непосредственно на сервере СУБД в отличии от любых типов взаимодействия, которые предполагают, что скрипт выполняется на стороне клиента, лишь через драйвер взаимодействуя с сервером, передавая ему запросы и получая ответы. Таким образом, PL/pgSQL востребован для операций с интенсивным вводом/выводом информации.

К наиболее типичным задачам, реализуемым на PL/SQL относятся:

* Выполнение и контроль целостности операций с высокими требованиями к скорости исполнения (например, финансовые операции, событиями систем мониторинга)
* Аудит изменения данных: сохранение истории изменения значений кортежей наиболее важных отношений в системе.
* Генерация разовых периодических отчетов, в которых анализируется большой объем данных (для экономии времени пересылки данных между клиентом и сервером).

Также к преимуществам использования PL/SQL является использование одного языка программирования для выполнения запросов и для выполнения инструкций.

Как правило, язык PL/pgSQL не используют в задачах, где некритичны его преимущества из-за относительно сложного процесса поддержки и нагрузки на сервер СУБД.

Процедуры и функции

Отличие процедур и функций состоит лишь в том, что процедуры не возвращают никаких значений, в отличии от функций. И процедуры и функции могут принимать входные параметры.

Но в PostgreSQL нет различий в синтаксисе создания функций и процедур, обе создаются с помощью ключевых слов create function.

Пример создания процедуры:

CREATE OR REPLACE FUNCTION add\_movie(title VARCHAR(70), production\_year INTEGER)

RETURNS void AS $$

DECLARE phonetic\_code VARCHAR(70);

BEGIN

SELECT soundex(title) INTO phonetic\_code;

RAISE NOTICE 'The phonetic code for % is %', title, phonetic\_code;

INSERT INTO title (title, production\_year, phonetic\_code, kind\_id) VALUES (title, production\_year, phonetic\_code, 1);

COMMIT;

END;

$$ LANGUAGE plpgsql;

Тело процедуры находится между BEGIN и END. При этом внутри тела могут быть транзакции, которые также будут обернуты в BEGIN-END.

В примере также используется soundex, которая доступна после подключения модуля (выполните до создания процедуры):

imdb=# create extension fuzzystrmatch;

Чтобы выполнить созданную процедуру, выберите ее:

select add\_movie('How I will pass all the exams', 2016);

Этот пример сейчас скорее всего не сработает с ERROR: duplicate key value violates unique constraint "title\_pkey". В конце этой части есть процедура, которая поможет вам добавить фильм.

**Определение переменных**

Определение переменных происходит с помощью команды DECLARE, после которой нужно указать имя и тип. Также можно сразу присвоить значение:

DECLARE production\_year INTEGER := 2016;

Определять переменные можно между BEGIN и END - в этом случае они действуют только, или сразу после определения процедуры (перед BEGIN как в примере выше), тогда переменная будет глобальной и доступна во всей процедуре.

Чтобы записать в переменную кортеж или одно значение, нужно использовать SELECT ... INTO var:

SELECT count(1) INTO actors\_num from cast\_info WHERE ...

**Управляющие конструкции**

При выборке данных в переменную, можно написать разные сценарии в зависимости от количества кортежей, которые возвращает запрос:

BEGIN

SELECT \* INTO movie FROM title WHERE title like '%'||movie\_name||'%';

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RAISE EXCEPTION 'movie % not found', movie\_name;

WHEN TOO\_MANY\_ROWS THEN

RAISE EXCEPTION 'movie % not unique', movie\_name;

END;

В этом примере также есть конструкция title like '%'||movie\_name||'%', в которой movie\_name - переменная, хранящая часть название фильма. Конкатенация строк выполняется оператором ||.

Также доступны циклы и условия:

FOR movie\_row IN SELECT \* FROM title limit 10 LOOP

RAISE NOTICE 'Currently working on movie %s ...', quote\_ident(movie\_row.title);

IF movie\_row.production\_year > 2000 THEN

RAISE NOTICE 'This movie is of this century';

END IF;

END LOOP;

Эта конструкция проходит по кортежам, возвращаемым запросом SELECT \* FROM title limit 10, для каждого выводит его название и, если фильм выпущен после 2000 года, то сообщение о его новизне.

**Использование курсоров**

Курсоры позволяют не выбирать в память сразу много данных, а использовать указатель на записи и переходить к следующей или предыдущей записи, считывая в память только ее.

Определения курсора:

DECLARE

curs1 refcursor; -- будет указан в будущем

curs2 CURSOR FOR SELECT \* FROM cast\_info; -- сразу же указывает на запрос (запрос не выполняется в момент определения)

Чтобы начать пользоваться курсором, его нужно открыть:

OPEN curs1 FOR SELECT \* FROM cast\_info; -- если курсор не был привязан к запросу ранее

OPEN curs2; -- для запросов, привязанных к запросу

После этого навигация курсора может быть следующей:

FETCH curs1 INTO rowvar; -- получить следующий кортеж запроса

MOVE curs1; -- то же самое

MOVE LAST FROM curs3; -- получить последний кортеж запроса

MOVE RELATIVE -2 FROM curs4; -- сдвинуть относительно текущего положения запроса на 2 кортежа назад

MOVE FORWARD 2 FROM curs4; -- сдвинуть курсор на 2 кортежа вперед.

Когда курсор не нужен или требуется его переоткрыть, то используется команда:

CLOSE curs1;

**Логирование и обработка ошибок**

Вывод информационного сообщения (при этом процедура продолжит работу дальше):

RAISE NOTICE 'Data processing...'

Команда завершающая процедуру или транзакцию с ошибкой:

RAISE EXCEPTION 'Something went wrong'

**Пример полезной для администрирования процедуры**

Попробуем для всех отношений, у которых есть последовательности генерации первичных ключей установить значения максимальных идентификаторов, чтобы эти последовательности далее работали корректно. Для этого понадобится

* выбрать все последовательности, связать их с таблицами
* для каждой пары выбрать подходящее значение
* установить это значение как текущее для последовательности.

CREATE OR REPLACE FUNCTION fix\_sequences()

RETURNS void AS $$

DECLARE max\_id INTEGER;

DECLARE t RECORD; -- переменная для обхода кортежей выборки

BEGIN

FOR t IN (select table\_name from pg\_class s

join information\_schema.tables on tables.table\_schema = 'public' and table\_name=replace(relname, '\_id\_seq', '' )

where s.relkind='S') LOOP

EXECUTE 'select max(id) from '||t.table\_name into max\_id; -- выполняет запрос, сформированный в строке и передает результат в max\_id

raise notice 'max id for % is %', t.table\_name, max\_id;

if max\_id is not null then

-- выполняет запрос увеличения текущего значения последовательности

EXECUTE 'SELECT setval('''||t.table\_name||'\_id\_seq'', '||max\_id||', true)';

end if;

END LOOP;

END;

$$ LANGUAGE plpgsql;

select fix\_sequences(); -- вызов процедуры

Триггеры

Триггеры - это процедуры, которые срабатывают в случае наступления в системе какого-либо события. В триггерах доступны переменные, связанные с происходящим событием.

**Типы триггеров**

Триггер может быть привязан к командам UPDATE, INSERT и DELETE конкретных отношений. Также триггер может быть запущен как до выполнения указанной операции, так и после. Например:

CREATE TRIGGER cool\_trigger\_name BEFORE INSERT OR UPDATE ON movie\_info

...

Этот триггер выполнится перед операциями добавления и обновления данных в таблице movie\_info.

В триггерах можно обращаться к изменяемым данным. также триггер может обрабатывать сразу все изменяемые кортежи в цикле. Обычно в каждой итерации вызывают процедуру, которая не принимает аргументов (в ней доступны переменные триггера) и возвращает объект триггера:

CREATE OR REPLACE FUNCTION check\_rows() RETURNS trigger AS $$

BEGIN

IF NEW.title is null THEN

RAISE EXCEPTION 'Invalid title';

END IF;

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER before\_insert\_check BEFORE INSERT OR UPDATE ON title

FOR EACH ROW EXECUTE PROCEDURE check\_rows();

Старые и новые значения данных доступны соответственно через OLD.\* и NEW.\* для каждого кортежа.

Транзакции

Транзакции нужны для того, чтобы проводить сложные операции атомарно: в случае успеха фиксировать все изменения, в случае неуспеха не фиксировать ни одно изменение транзакции.

Управление транзакцией выполняется с помощью команд BEGIN, SAVEPOINT, ROLLBACK и COMMIT. Пример сценария сложной транзакции:

BEGIN; -- начало транзакции

-- выполнение операций

SAVEPOINT my\_savepoint; -- сохранить точку, на которую можно потом вернуться,

-- но изменения в базу все еще не вносятся

-- выполнение операций

ROLLBACK TO my\_savepoint; -- после сохранения произошла ошибка, откат на состояние my\_savepoint

-- выполнение другого сценария

IF check\_ok() THEN:

COMMIT; -- фиксация результата

ELSE

ROLLBACK;

END IF;

**Создание таблицы истории**

Для задания понадобится таблица истории, сделать ее проще всего унаследовав от существующей, а затем убрав наследование и добавив колонку для времени изменения:

CREATE TABLE tablename\_history () INHERITS (tablename); -- копирует структуру таблицы без записей

ALTER TABLE tablename\_history NO INHERIT tablename;

Задания лабораторной работы

1. Сделать таблицу с историей изменений company\_name, в которую при обновлении через триггер записываются прежние значения и дата окончания их действий (дата обновления). Сделайте запрос, который показывает значения для конкретного кортежа на заданный момент времени.

2. Создать функцию с входным параметром - имя актера. Если актер с таким именем не найден, функция должна вернуть 0. Если найден, то вычислить через person\_info birth date (info\_type\_id=21) и death date (info\_type\_id=23) возраст актера. Если актер еще не умер, то вычислить, сколько сейчас лет. Функция должна вернуть целое число прожитых лет.

3. Используя функцию, созданную ранее, создайте процедуру, которая по имени актера выводит в лог текст следующего содержания:

Name: ...

Nicknames: nickname1, nickname2, ... строка выводится только если у человека есть клички, выводятся все клички через запятую

Age: ... вычисляется из функции

First appear: ... фильм и год выхода, в котором актер впервые снялся.

Если актер не найден или не удалось определить его первую роль, то процедура должна завершиться с ошибкой и вывести в лог "Invalid data"

Защита лабораторной работы

* Показать код созданных триггера, функции и процедуры
* Продемонстрировать их работу
* По просьбе преподавателя прокомментировать ход выполнения кода.

Дополнительно

* <http://www.postgresql.org/docs/9.3/static/plpgsql.html>
* <http://postgres.cz/wiki/PL/pgSQL_%28en%29> - примеры процедур, функций и триггеров
* <http://www.postgresql.org/docs/9.3/static/tutorial-transactions.html>
* <http://www.postgresql.org/docs/9.3/static/plpython.html> - позволяет писать хранимые процедуры на питоне, нужно включить расширение