Лабораторная работа № 4

SQL

Изменение прав доступа к объектам системы управления базами данных

Для изменения прав доступа роли к объектам системы управления баз данных (базам, схемам, таблицам, столбцам и т. д.) используются два оператора:

- `GRANT` - оператор назначает прав доступа;
- `REVOKE` - оператор лишает прав доступа.

Назначение прав доступа осуществляется следующим образом:

```
GRANT ${PRIVILEGES} ON ${OBJECT} TO ${ROLE};
```

Лишение прав осуществляется следующей командой:

```
REVOKE ${PRIVILEGES} ON ${OBJECT} FROM ${ROLE};
```

Где:

- `${PRIVILEGES}` - список прав (краткий список):
- `SELECT`;
- `CONNECT`;
- `INSERT`;
- `UPDATE`;
- `DELETE`;
- `TRUNCATE`;
- `CREATE`;
- `ALL PRIVILEGES` - все возможные права.

Пример. Предоставление всех прав на таблицу `TEST` для роли `admin`:

```
GRANT ALL PRIVILEGES ON TEST TO admin;
```

Пример. Лишение права `удаления` на таблицу `TEST` для роли `admin`:

```
REVOKE DELETE ON TEST FROM admin;
```

Хранимые функции в PostgreSQL. Серверное программирование

Функции (хранимые процедуры) - объекты базы данных, которые компилируются и размещаются в системе управления базами данных.

`PostgreSQL` поддерживает несколько языков программирования для создания функций. Одним из таких языков является Python: для объявления функций в `PostgreSQL` использует процедурный язык `PL/Python`.

Функции на языке `PL/Python` являются объектами базы данных, для их создания необходимо подключить специальное расширение `PostgreSQL`:

1. Необходимо установить дополнительный пакет (`${VERSION}` - версия `PostgreSQL`):

```
apt install postgresql-plpython3-${VERSION}
```

2. Создать объект базы данных (создаётся для конкретной базы):

```
CREATE EXTENSION plpython3u;
```

После подключения расширения можно создавать функции, для этого используется выражение `CREATE FUNCTION`

```
CREATE OR REPLACE FUNCTION ${FUNCTION_NAME} (
        ${ARGS}
) RETURNS ${TYPE} AS $$
        ${BODY}
$$ LANGUAGE plpython3u;
```

Где:

- `${FUNCTION_NAME}` - имя функции.
- `${ARGS}` - список аргументов. Аргументы записываются в формате: `${ARG_NAME} ${TYPE}`;
- `${TYPE}` - тип возвращаемого значения, поддерживаемый `PostgreSQL`;
- `${BODY}` - тело функции, содержащее код на языке `PL/Python`.

Вызов функции производится с помощью оператора `SELECT`:

```
SELECT ${FUNCTION_NAME}(${VALUES});
```

Где:

- `${FUNCTION_NAME}` - имя функции;
- `${VALUES}` - список значений аргументов.

Пример 1. Функция возвращает текущий год:

```
CREATE OR REPLACE FUNCTION current_year () RETURNS INTEGER
AS $$
from datetime import datetime
return datetime.now().year
$$ LANGUAGE plpython3u;

-- вызов функции
SELECT current_year();
```

Пример 2. Функция возвращает текст в верхнем регистре:

```
CREATE OR REPLACE FUNCTION upper (
        msg TEXT
) RETURNS TEXT AS $$
return msg.upper()
$$ LANGUAGE plpython3u;

-- вызов функции
SELECT upper('test message')
```

Статические и глобальные данные

Каждая функция может использовать статические данные. Статические данные позволяют хранить информацию между вызовами функции.
Статические данные функции сохраняются в словарь `SD`:

```
CREATE OR REPLACE FUNCTION cache (
        msg TEXT
) RETURNS TEXT AS $$

key = "key"

if SD.get(key) is None:
        SD[key] = msg

return SD[key]
$$ LANGUAGE plpython3u;

SELECT cache('1st message');
SELECT cache('2nd message');
SELECT cache('3rd message');
```

Глобальные данные сохраняются в словарь `GD` и доступны из любых функций `PL/Python`:

```
CREATE OR REPLACE FUNCTION write_to_cache (
        msg TEXT
) RETURNS VOID AS $$
GD["cache"] = msg
$$ LANGUAGE plpython3u;

CREATE OR REPLACE FUNCTION read_from_cache() RETURNS TEXT
AS $$

msg = "" if GD.get("cache") is None else GD["cache"]

return msg
$$ LANGUAGE plpython3u;

SELECT write_to_cache('this is a message');
SELECT read_from_cache();
SELECT write_to_cache('this is another message');
SELECT read_from_cache();
```

Использование `SQL` запросов

Использование `SQL` запроса внутри функции выполняется посредством объекта `plpy`. При этом, используются два метода:

- `prepare` - метод подготавливает запрос и создает план - специально подготовленный объект запроса;
- `execute` - метод выполняет запрос.

Планы поддерживают метод `execute`.

Пример 1. Выполнение простого запроса:

```
CREATE OR REPLACE FUNCTION wrap_version() RETURNS TEXT AS
$$

res = plpy.execute("SELECT version()")

if len(res):
```

```
        version = res[0]["version"]
else:
        version = "unknown"
return version
$$ LANGUAGE plpython3u;
```

Пример 2. Подготовка и выполнение плана (база library):

```
CREATE OR REPLACE FUNCTION get_books_by_author_name(
        name TEXT
) RETURNS JSON AS $$
import json

KEY = "QUERY"

if SD.get(KEY) is None:
        SD[KEY] = plpy.prepare("""SELECT book.name FROM
book
INNER JOIN (
        (SELECT * FROM author WHERE name = $1) AS T
        INNER JOIN author_has_book ON T.id =
author_has_book.author_id
) AS T2 ON T2.book_id = book.id;
""", ["text"])

res = SD[KEY].execute([name])
books = list(map(lambda rec: rec["name"], res))

return json.dumps(books, ensure_ascii=False)
$$ LANGUAGE plpython3u;
```

Триггеры в PostgreSQL

Триггер - определённый вид хранимых процедур, связанный с некоторой функцией-обработчиком, которая вызывается всякий раз при наступлении некоторого события модификации объекта или его

данных. Функции-обработчики могут срабатывать по следующим схемам:

- `BEFORE` - до наступления какого-то события;
- `AFTER` - после наступления события;
- `INSTEAD OF` - функция будет вызвана вместо события.

Создание триггера осуществляется командой `CREATE TRIGGER`:

```
CREATE OR REPLACE TRIGGER ${NAME}
        BEFORE | AFTER | INSTEAD OF
        ${EVENT} [OR ${ANOTHER_EVENT}]
        ON ${TABLE_NAME}
        EXECUTE FUNCTION ${FN_NAME}(${ARGS});
```

Список возможных событий:

- `INSERT`;
- `UPDATE`;
- `DELETE`;
- `TRUNCATE`.

Для функции-обработчика на `PL/Python` предусмотрен специальный словарь `TD`.
Некоторые полезные значения:
`TD["event"]` - название события (`INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`);
`TD["when"]` - `BEFORE`, `AFTER` или `INSTEAD OF`;
`TD["name"]` - имя триггера;
`TD["table_name"]` - имя таблицы;
`TD["args"]` - список аргументов функции.

Отметим, что функция-обработчик должна возвращать тип `TRIGGER`.
Пример. Создание функции-обработчика и триггера (база library):

```
-- create function
CREATE OR REPLACE FUNCTION handler() RETURNS TRIGGER AS $$
```

```
import json

plpy.notice(json.dumps(TD))
with open("/tmp/LOG", "w") as fp:
        json.dump(TD, fp)
$$ LANGUAGE plpython3u;

-- create trigger
CREATE TRIGGER test_trigger
        AFTER
        INSERT
        ON author
        EXECUTE FUNCTION handler();
```

Удаление триггера осуществляется командой `DROP TRIGGER`:

```
DROP TRIGGER IF EXISTS ${NAME} ON ${TABLE_NAME};
```

Пример. Удаление триггера и функции-обработчика:

```
DROP TRIGGER IF EXISTS test_trigger ON author;
DROP FUNCTION IF EXISTS handler;
```

Использование `psycopg2`. Клиентское программирование

`psycopg2` - библиотека `Python`, которая позволяет использовать `API` `PostgreSQL`: осуществлять подключение и выполнять `SQL` запросы.

Для использования `psycopg2` необходимо установить одноименный пакет:

```
pip install psycopg2-binary
# ИЛИ
pip install psycopg2 # требуется библиотека libpq
```

После установки `psycopg2` можно использовать в коде `Python`.

Пример. Получение версии `PostgreSQL` с помощью `psycopg2`:

```
#! /usr/bin/env python3
import psycopg2
import time

closeable = []

try:
        connection = psycopg2.connect(
                user="admin",
                password="1234",
                host="127.0.0.1",
                port="5432",
                database="postgres",
        )
        closeable.append(connection)

        print("connection has been opened")

        cursor = connection.cursor()
        closeable.append(cursor)

        print("cursor has been created")

        cursor.execute("SELECT version();")

        record = cursor.fetchone()

        print("response has been received: ", record)
except Exception as err:
        print("Error: ", err)
finally:
        if bool(closeable):
                _ = [*map(lambda o: o.close(), closeable)]

print("connect has been closed")
```

Практическая часть

Требования:

1. Все задания должны быть выполнены с помощью сценариев на языке `Python` с применением `psycopg2`.
2. Конфигурационные параметры, используемые для подключения к базе данных должны быть записаны в файл конфигурации `conf.json` и зачитываться из него при запуске сценариев `Python`.
3. В качестве базы данных необходимо использовать базу из лабораторной работы №3.

Задания:

1. Реализовать функцию `PostgreSQL` (функция сервера) `reverse`, которая разворачивает строку в обратном порядке: функция принимает строку и возвращает строку, символы в возвращаемой строке должны быть расположены в обратном порядке относительно строки-аргумента.
2. Реализуйте функцию `PostgreSQL` (функция сервера), которая принимает название должности (строка) и возвращает минимум и максимум заработной платы для данной должности. Тип возвращаемого значения - строка в формате `JSON`. Если должность существует:

```
{
        "ok": {"salary": {"min": "${VALUE}" , "max":
"${VALUE}"}},
        "error": null
}
```

Если должность не найдена:

```
{
        "ok": null,
```

```
        "error": "data not found"
}
```

3. Реализовать триггер и функцию-обработчик: триггер должен срабатывать после вставки заработной платы для определенного сотрудника и записывать (добавлять) в файл `notification` сообщение вида (каждое сообщение на отдельной строке):

```
salary: ${SURNAME} ${NAME} (номер: ${EMPLOYEE_ID},
должность: ${POSITION}): начислена заработная плата за
${DATE} в размере ${SALARY}
```

4. Реализовать триггер и функцию-обработчик, триггер должен срабатывать:

- при добавлении новой должности;
- `[*]` при освобождении должности;

При срабатывании триггера, в файл `notification` должна добавляться запись вида:

```
job: открыта вакансия ${NAME} в отделе ${DEPARTMENT};
заработная плата ${VALUE}
```