ЛАБОРАТОРНАЯ РАБОТА №5

ПРОЦЕДУРЫ, ФУНКЦИИ, ТРИГГЕРЫ В POSTGRESQL

Цель работы: овладеть практическими создания и использования процедур, функций и триггеров в базе данных PostgreSQL.

Оборудование: компьютерный класс.

Программное обеспечение: СУБД PostgreSQL, SQL Shell (psql).

Практическое задание (min - 6 баллов, max - 10 баллов, доп. баллы - 3):

1. Создать 3 процедуры для индивидуальной БД согласно варианту (часть 4 ЛР 2). Допустимо использование IN/OUT параметров. Допустимо создать авторские процедуры. (3 балла)
2. Создать триггеры для индивидуальной БД согласно варианту:
   Вариант 2.1. 3 триггера - 3 балла (min). Допустимо использовать триггеры логирования из практического занятия по функциям и триггерам.
   Вариант 2.2. 7 оригинальных триггеров - 7 баллов (max).

Дополнительные баллы - 3:

Модифицировать триггер (триггерную функцию) на проверку корректности входа и выхода сотрудника (см. Практическое задание 1 Лабораторного практикума (Приложение)) с максимальным учетом «узких» мест некорректных данных по входу и выходу).

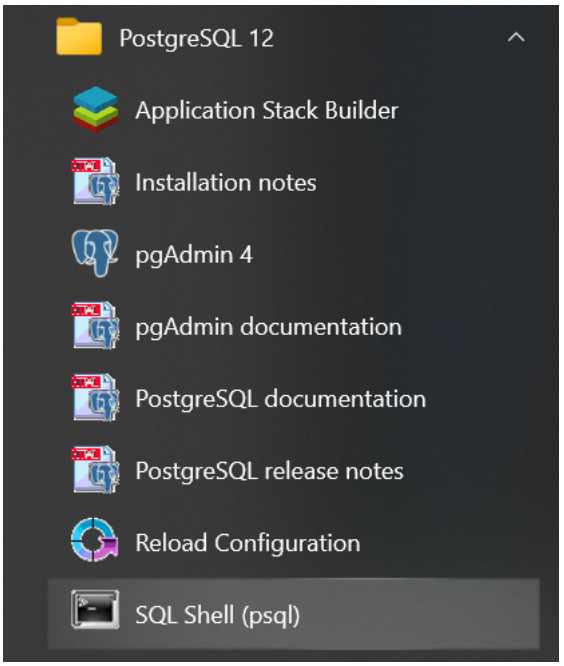
Указание. Работа выполняется в консоли SQL Shell (psql).

Технология выполнения работы:

Работа в SQL SHELL (psql)

psql – это консольный клиент, который дает еще один из способов взаимодействия с PostgreSQL.

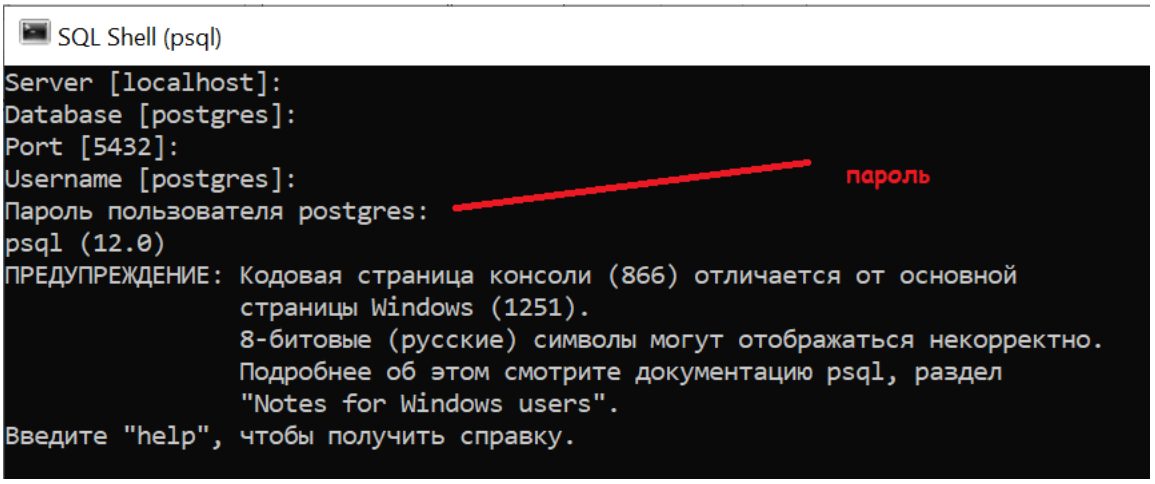
Он позволяет интерактивно вводить запросы, передавать их в PostgreSQL и видеть результаты. Также запросы могут быть получены из файла или из аргументов командной строки. Кроме того, psql предоставляет ряд метакоманд и различные возможности, подобные тем, что имеются у командных оболочек, для облегчения написания скриптов и автоматизации широкого спектра задач.



После запуска программ предлагает ввести названия:

- сервера,
- базы данных,
- порта.
- пользователя.

Их можно «пропустить», нажав Enter, так как для них будут использоваться значения по умолчанию (для сервера - localhost, для базы данных - postgres, для порта - 5432, в качестве пользователя - суперпользователь postres). Далее нужно ввести пароль для пользователя (по умолчанию пользователя postgres):



После удачного подключения можно будет отправлять серверу команды через psql.

Для создания БД используйте команду `create database`.

В программе psql есть множество внутренних команд, которые не являются SQL-операторами. Они начинаются с обратной косой черты, «\».

Справку по различным SQL-командам PostgreSQL можно получить, введя `\h`.

Перечень внутренних команд psql можно получить по команде `\?`.

Чтобы выйти из psql, нужно ввести `\q`.

Работа с БД

Чтобы корректно отображалась кириллица, можно применить команду `\! chcp 1251`:

```
postgres=# \! chcp 1251
Текущая кодовая страница: 1251
```

Чтобы просмотреть список активных БД используется команда `\list` (SQL: `select * from pg_database`):

```
postgres=# \list
                                                    ╥яшёюъ срч фрээѵї
    ╚ь      | ъырфхыхЎ | ╧юфшЁютър |     LC_COLLATE      |      LC_CTYPE       |     ╧Ёртр фюёЄєяр
-----------+----------+-----------+---------------------+---------------------+-----------------------
 emp        | postgres | UTF8      | Russian_Russia.1251 | Russian_Russia.1251 |
 ira        | postgres | UTF8      | Russian_Russia.1251 | Russian_Russia.1251 |
 kitdb      | kitadmin | UTF8      | Russian_Russia.1251 | Russian_Russia.1251 |
 mmm1       | postgres | UTF8      | Russian_Russia.1251 | Russian_Russia.1251 |
 postgres   | postgres | UTF8      | Russian_Russia.1251 | Russian_Russia.1251 |
 template0  | postgres | UTF8      | Russian_Russia.1251 | Russian_Russia.1251 | =c/postgres          +
            |          |           |                     |                     | postgres=CTc/postgres
 template1  | postgres | UTF8      | Russian_Russia.1251 | Russian_Russia.1251 | =c/postgres          +
            |          |           |                     |                     | postgres=CTc/postgres
 test       | postgres | UTF8      | Russian_Russia.1251 | Russian_Russia.1251 |
 vadim      | postgres | UTF8      | Russian_Russia.1251 | Russian_Russia.1251 |
(9 ёЄЁюъ)
```

Чтобы просмотреть список активных список пользователей используется команда:
`select * from pg_shadow;`
`\l+` позволяет получить расширенную информацию об активных БД:

```
SQL Shell (psql)
emp=# \dt+
                    Список отношений
 Схема |    Имя     |  Тип  | Владелец |   Размер   | Описание
-------+------------+-------+----------+------------+---------
 public | employee   | таблица | postgres | 16 kB      |
 public | logs       | таблица | postgres | 16 kB      |
 public | time_punch | таблица | postgres | 8192 bytes |
(3 строки)


emp=# \l+
                                                                                   Список баз данных
   Имя    | Владелец | Кодировка |     LC_COLLATE      |      LC_CTYPE       |      Права доступа      | Размер | Табл. пространство |                Описание
----------+----------+-----------+---------------------+---------------------+-------------------------+--------+--------------------+----------------------------------------
------
 emp       | postgres | UTF8      | Russian_Russia.1251 | Russian_Russia.1251 |                         | 8097 kB | pg_default         |
 ira       | postgres | UTF8      | Russian_Russia.1251 | Russian_Russia.1251 |                         | 8193 kB | pg_default         |
 kitdb     | kitadmin | UTF8      | Russian_Russia.1251 | Russian_Russia.1251 |                         | 9129 kB | pg_default         |
 mmm1      | postgres | UTF8      | Russian_Russia.1251 | Russian_Russia.1251 |                         | 7881 kB | pg_default         |
 postgres  | postgres | UTF8      | Russian_Russia.1251 | Russian_Russia.1251 |                         | 8081 kB | pg_default         | default administrative connection dat
abase
 template0 | postgres | UTF8      | Russian_Russia.1251 | Russian_Russia.1251 | =c/postgres          +| 7769 kB | pg_default         | unmodifiable empty database
           |          |           |                     |                     | postgres=CTc/postgres   |        |                    |
 template1 | postgres | UTF8      | Russian_Russia.1251 | Russian_Russia.1251 | =c/postgres          +| 7769 kB | pg_default         | default template for new databases
           |          |           |                     |                     | postgres=CTc/postgres   |        |                    |
 test      | postgres | UTF8      | Russian_Russia.1251 | Russian_Russia.1251 |                         | 7809 kB | pg_default         |
 vadim     | postgres | UTF8      | Russian_Russia.1251 | Russian_Russia.1251 |                         | 9073 kB | pg_default         |
(9 строк)

-- Далее  --
```

По умолчанию консоль в Windows поддерживает кодировку CP866, тогда как базы данных могут работать совсем с другой кодировкой, например, 1251. И даже сам клиент psql выводит соответствующие сообщения. Кроме того, при получении данных, при выводе информации о базах данных, таблицы и т.д. некоторая информация может отображаться некорректно. В этом случае перед запуском psql надо установить нужную кодировку и затем из консоли запустить программу psql.

Для создания БД используйте команду `create database`.
Для подключения к БД применяется команда `\c name_db`:

```
postgres=# \c emp
Вы подключены к базе данных "emp" как пользователь "postgres".
```

Список таблиц текущей БД можно получить по команде \dt:

```
emp=# \dt
            Список отношений
 Схема  |   Имя    |   Тип   | Владелец
--------+----------+---------+----------
 public | employee | таблица | postgres
(1 строка)
```

\dt+ позволяет получить информацию о размере и описании таблиц:

```
emp=# \dt+
                        Список отношений
 Схема  |    Имя     |   Тип   | Владелец |   Размер   | Описание
--------+------------+---------+----------+------------+----------
 public | employee   | таблица | postgres | 16 kB      |
 public | logs       | таблица | postgres | 16 kB      |
 public | time_punch | таблица | postgres | 8192 bytes |
(3 строки)
```

Для создания таблиц БД используйте команду `create table`.

Процедуры. Функции

Хранимая процедура (procedure)— это объект базы данных, представляющий собой набор SQL- инструкций, который компилируется один раз и хранится на сервере.

Хранимая функция (function) отличается от хранимой процедуры, тем что хранимая функция всегда возвращает значение.

Процедуры:

1. Не возвращают значение. Однако могут выдавать данные в вызывающий код через выходные параметры.
2. Вызываются отдельно командой CALL.
3. Процедура, в отличие от функции, может фиксировать или откатывать транзакции во время её выполнения (а затем автоматически начинать новую транзакцию), если вызывающая команда CALL находится не в явном блоке транзакции.
4. Некоторые атрибуты функций (например, STRICT) неприменимы к процедурам. Эти атрибуты влияют на вызов функций в запросах и не имеют отношения к процедурам.

Создание процедуры определяется командой:

```
CREATE [ OR REPLACE ] PROCEDURE
имя ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
      [ { DEFAULT | = } выражение_по_умолчанию ] [, ...] ] )
  { LANGUAGE имя_языка
| TRANSFORM { FOR TYPE имя_типа } [, ... ]
| [ EXTERNAL ] SECURITY INVOKER
| [ EXTERNAL ] SECURITY DEFINER
| SET параметр_конфигурации { TO значение | = значение | FROM
CURRENT }
| AS 'определение' | AS 'объектный_файл', 'объектный_символ'
} ...
```

Описание параметров: https://postgrespro.ru/docs/postgresql/11/sql-createprocedure

Функции:

1. В PostgreSQL представлены функции четырёх видов (https://postgrespro.ru/docs/postgresql/11/xfunc):
   - функции на языке запросов (функции, написанные на SQL) (https://postgrespro.ru/docs/postgresql/11/xfunc-sql);
   - функции на процедурных языках (функции, написанные, например, на PL/pgSQL или PL/Tcl) ;
   - внутренние функции;
   - функции на языке C.

2. Функции любых видов могут принимать в качестве аргументов (параметров) базовые типы, составные типы или их сочетания (https://postgrespro.ru/docs/postgresql/11/extend-type-system). Кроме того, любые функции могут возвращать значения базового или составного типа. Также можно определить функции, возвращающие наборы базовых или составных значений.

3. Функции многих видов могут также принимать или возвращать определённые псевдотипы (например, полиморфные типы), но доступные средства для работы с ними различаются.

Создание функции определяется командой:

```
CREATE [ OR REPLACE ] FUNCTION имя ( [ [ режим_аргумента ] [
   имя_аргумента ] тип_аргумента [ { DEFAULT | = }
   выражение_по_умолчанию ] [, ...] ] )
 [ RETURNS тип_результата
 | RETURNS TABLE ( имя_столбца тип_столбца [, ...] ) ]
 { LANGUAGE имя_языка
 | TRANSFORM { FOR TYPE имя_типа } [, ... ]
 | WINDOW
 | IMMUTABLE | STABLE | VOLATILE | [ NOT ] LEAKPROOF
 | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
 | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY
 DEFINER
 | PARALLEL { UNSAFE | RESTRICTED | SAFE }
 | COST стоимость_выполнения
 | ROWS строк_в_результате
 | SET параметр_конфигурации { TO значение | = значение | FROM
 CURRENT }
 | AS 'определение'
 | AS 'объектный_файл', 'объектный_символ'
} ...
```

Управляющие структуры в процедурах и функциях

- RETURN
- RETURN NEXT, RETURN QUERY
- Условные операторы:

  IF – THEN

  IF – THEN – ELSE

  IF – THEN – ELSIF

  Простой CASE

  CASE с перебором условий

```
●Простые циклы:
  LOOP
  EXIT
  CONTINUE
  WHILE
  FOR (целочисленный вариант)
●Цикл по результатам запроса
●Цикл по элементам массива
```

Основные способы возврата данных

См. материалы лекции: https://clck.ru/346MXt

Удаление хранимых процедур

Удалить функцию (https://postgrespro.ru/docs/postgresql/11/sql-dropfunction):

```
DROP FUNCTION [ IF EXISTS ] имя [ ( [ [ режим_аргумента ]
  [ имя_аргумента ] тип_аргумента [, ...] ] ) ] [, ...]
  [ CASCADE | RESTRICT ]
```

Удалить функцию (https://postgrespro.ru/docs/postgresql/11/sql-dropprocedure):

```
DROP PROCEDURE [ IF EXISTS ] имя [ ( [ [ режим_аргумента ]
  [ имя_аргумента ] тип_аргумента [, ...] ] ) ] [, ...]
  [ CASCADE | RESTRICT ]
```

Триггеры

Триггер (trigger) — это хранимая процедура, которую пользователь не вызывает непосредственно, а исполнение которой обусловлено действием по модификации данных: добавлением INSERT, удалением DELETE строки в заданной таблице, или изменением UPDATE данных в определенном столбце заданной таблицы реляционной базы данных.

Создание триггера определяется командой:

```
CREATE TRIGGER триггер
BEFORE | AFTER } { событие [ OR событие ] } ON таблица
FOR EACH { ROW |  STATEMENT }
EXECUTE PROCEDURE функция ( аргументы )
```

Параметры создания триггера:

- **CREATE TRIGGER триггер**. В аргументе *триггер* указывается произвольное имя создаваемого триггера. Имя может совпадать с именем триггера, уже существующего в базе данных при условии, что этот триггер установлен для другой таблицы. Кроме того, по аналогии с большинством других несистемных объектов баз данных, имя триггера (в сочетании с таблицей, для которой он устанавливается) должно быть уникальным лишь в контексте базы данных, в которой он создается **{ BEFORE | AFTER }.** Ключевое слово BEFORE означает, что функция должна выполняться *перед* попыткой выполнения операции, включая все встроенные проверки ограничений данных, реализуемые при выполнении команд INSERT и DELETE. Ключевое слово AFTER означает, что функция вызывается после завершения операции, приводящей в действие триггер.
- **{ *событие* [OR событие ...] }.** События SQL, поддерживаемые в PostgreSQL: INSERT, UPDATE или DELETE. При перечислении нескольких событий в качестве разделителя используется ключевое слово OR.

- **ON таблица.** Имя таблицы, модификация которой заданным событием приводит к срабатыванию триггера.
- **FOR EACH { ROW | STATEMENT }.** Ключевое слово, следующее за конструкцией FOR EACH и определяющее количество вызовов функции при наступлении указанного события. Ключевое слово ROW означает, что функция вызывается *для каждой* модифицируемой записи. Если функция должна вызываться всего один раз для всей команды, используется ключевое слово STATEMENT.
- **EXECUTE PROCEDURE функция (*аргументы*).** Имя вызываемой функции с аргументами. На практике аргументы при вызове триггерных функций не используются.

Определение триггерной функции:

```
CREATE FUNCTION функция () RETURNS trigger AS '
DECLARE
        объявления ;
BEGIN
команды;
END; '
LANGUAGE  plpgsql;
```

Специальные переменные, доступные в триггерных функциях:

Имя	Тип	Описание
NEW	RECORD	Новые значения полей записи базы данных, созданной командой INSERT или обновленной командой UPDATE, при срабатывании триггера уровня записи (ROW). Переменная используется для модификации новых записей. **Внимание !!!** Переменная NEW доступна только при операциях INSERT и UPDATE. Поля записи NEW могут быть изменены триггером.
OLD	RECORD	Старые значения полей записи базы данных, содержавшиеся в записи перед выполнением команды DELETE или UPDATE при срабатывании триггера уровня записи (ROW) **Внимание !!!** Переменная OLD доступна только при операциях DELETE и UPDATE. Поля записи OLD можно использовать только для чтения, изменять нельзя.
TG_NAME	name	Имя сработавшего триггера.
TG_WHEN	text	Строка BEFORE или AFTER в зависимости от момента срабатывания триггера, указанного в определении (до или после операции).

Имя	Тип	Описание
TG_LEVEL	text	Строка ROW или STATEMENT в зависимости от уровня триггера, указанного в определении.
TG_OP	text	Строка INSERT, UPDATE или DELETE в зависимости от операции, вызвавшей срабатывание триггера.
TG_RELID	oid	Идентификатор объекта таблицы, в которой сработал триггер.
TG_RELNAME	name	name Имя таблицы, в которой сработал триггер.

Удаление триггера определяется командой:

DROP TRIGGER [IF EXISTS] *имя* ON *имя_таблицы*
[CASCADE | RESTRICT]

Содержание отчета:

1. Цель работы.
2. Практическое задание.
3. Выполнение:

Вариант 1

- скрипты кода разработанных объектов (процедур/функций и триггера на логирование действий) и подтверждающие скриншоты работы и результатов в psql согласно индивидуальному заданию (часть 4 и 5).

Вариант 2

- скрипты кода разработанных объектов (процедур/функций) и подтверждающие скриншоты работы и результатов в psql согласно индивидуальному заданию (часть 4 и 5);
- *вариант 2.1*: скрипт кода модифицированного триггера (триггерной функции) на проверку корректности входа и выхода сотрудника (см. Практическое задание 1 Лабораторного практикума) с максимальным учетом «узких» мест некорректных данных по входу и выходу и подтверждающие скриншоты работы и результатов в psql;
- *вариант 2.2*: скрипт кода авторского триггера по варианту индивидуального задания и подтверждающие скриншоты работы и результатов в psql.

4. Выводы.

Список источников

1. https://postgrespro.ru/docs/postgresql/12/app-psql
2. https://postgrespro.ru/docs/postgresql/12/tutorial-accessdb

3. https://postgrespro.ru/docs/postgresql/11/sql-createprocedure
4. https://postgrespro.ru/docs/postgresql/11/sql-createfunction
5. https://postgrespro.ru/docs/postgresql/9.6/plpgsql-trigger
6. https://postgrespro.ru/docs/postgresql/11/sql-droptrigger
7. https://postgrespro.ru/docs/postgresql/9.6/plpgsql-trigger#plpgsql-trigger-audit-example
8. Материалы лекции «процедуры. Функции. Триггеры» https://postgrespro.ru/docs/postgresql/11/sql-createfunction

Приложение

ЛАБОРАТОРНЫЙ ПРАКТИКУМ №3

Работа с триггерами и функциями в PostgreSQL

Что такое триггер?

Триггеры SQL, также называемые триггерами баз данных, позволяют сказать движку SQL (например PostgreSQL) выполнить часть кода при наступлении некоторого события, или даже перед наступлением события.

В PostgreSQL исполняемый код описывается посредством создания функции, которая возвращает значение типа trigger. В некоторых других движках, например, MySQL блок кода является частью триггера, и находится внутри него.

Как создать триггер SQL: синтаксис PostgreSQL

Составляющие создания триггера для базы данных:

- тип события триггера;
- до или после события;
- воздействие триггера.

Типы событий триггера

События SQL, поддерживаемые PostgreSQL: `INSERT`, `UPDATE` или `DELETE`. При перечислении нескольких событий в качестве разделителя используется ключевое слово `OR`.

Триггер BEFORE (до) или AFTER (после)

Триггер может выполняться либо до, либо после события.

Если вы хотите заблокировать событие типа INSERT, вы захотите выполнять действие до (BEFORE). Если вы хотите быть уверенным, что событие действительно произойдет, идеальный вариант - после (AFTER).

Воздействие триггера

Триггер может выполняться либо на строку, либо на оператор. Например, если выполняется один оператор UPDATE, который изменяет 5 строк в таблице.

Если указать в триггере FOR EACH ROW, тогда триггер выполнится 5 раз. Если указать FOR EACH STATEMENT, тогда он выполнится только раз.

И, конечно, нельзя забыть о фактическом коде, который выполняется при срабатывании триггера. В PostgreSQL он помещается в функцию и отделен от триггера. Разделение триггера и кода, который он выполняет, создает более чистый код и позволяет нескольким триггерам выполнять один и тот же код.

Пример 1. Таймер для учета работы сотрудников

В организации необходимо фиксировать время прихода и ухода сотрудников на работу, вычисление общего отработанного времени.

Рассмотрим пример таймера и посмотрим, каким образом можно использовать триггеры для предотвращения ввода сотрудниками неверных данных.

Настройка схемы БД

Структура схемы предполагает каждый вход и выход отдельными событиями. Каждое событие – это строка в таблице `time_punch`. Как альтернативу вы можете также сделать каждую «рабочую смену» сотрудника событием и хранить время как входа, так и выхода в одной строке. Информация о сотруднике хранится в таблице `employee`.

Задание 1:

1. Создайте БД `emp_time`. Используйте консоль psql.
2. Проверьте список активных БД.
3. Подключитесь к БД `emp_time`.
4. Создайте таблицы БД со следующей структурой:

   Таблица `employee`
   `id` (тип `serial`, первичный ключ)
   `username` (тип `varchar`)

   Таблица `time_punch`
   `id` (тип `serial` , первичный ключ)
   `employee_id` (тип `int`, обязательное, внешний ключ: соответствует ключу `employee.id`)
   `is_out_punch` (тип `boolean`, обязательное, значение по умолчанию `false`)
   `punch_time` (тип `timestamp`, обязательное, значение по умолчанию `now()`).

Вставьте данные в таблицы:

1. Вставьте в таблицу сотрудников строку для сотрудника с именем Михаил.
2. Вставьте в таблицу учета времени две строки: вход Михаил в 10.00 2021-01-01 и выход в 11.30 2021-01-01.
3. Проверьте содержимое таблиц.

Вычисление рабочего времени

Запрос позволяет на каждый выход найти соответствующий вход:

```
select tp1.punch_time - tp2.punch_time as time_worked
     from time_punch tp1 join time_punch tp2
        on tp2.id = (select tps.id from time_punch tps
                where tps.id < tp1.id and
                    tps.employee_id = tp1.employee_id and
                       not tps.is_out_punch
                       order by tps.id desc limit 1
                     )
     where tp1.employee_id = 1 and tp1.is_out_punch;
```

Задание 2:

Проверьте работу запроса.

Примечание.

Одна из проблем в этой схеме состоит в том, что возможно вставить несколько "входов" или "выходов" подряд. С созданным запросом это приведет к неоднозначности, которая может привести к неточным расчетам и зарплате сотрудников – больше или меньше, чем они должны были бы получить.

Триггер INSERT BEFORE: сохранение целостности данных

Требуется то, что не позволит нарушить шаблон вход/выход. К сожалению, ограничения check только отслеживают вставляемую или обновляемую строку и не могут учитывать данные из других строк.

Создадим триггер для предотвращения события INSERT, которое нарушает шаблон. Сначала создадим "триггерную функцию". Эта функция есть то, что будет выполнять триггер при наступлении события.

Триггерная функция создается как обычная функция PostgreSQL за тем исключением, что возвращает *триггер*.

```
create or replace function fn_check_time_punch() returns
trigger as $psql$
    begin
        if new.is_out_punch = (
          select tps.is_out_punch
          from time_punch tps
          where tps.employee_id = new.employee_id
          order by tps.id desc limit 1
        ) then
            return null;
        end if;
        return new;
    end;
$psql$ language plpgsql;
```

Ключевое слово `new` представляет значения вставляемой строки. Это также объект, который можно вернуть, чтобы позволить продолжиться вставке. Напротив, возвращение `null` остановит вставку.

Этот запрос сначала находит в `time_punch` предыдущее значение и гарантирует, что это значение входа/выхода не совпадает с вставляемым значением. Если значения совпадают, то триггер возвращает `null`, и `time_punch` не записывается. В противном случае, триггер возвращает `new` и оператор `insert` продолжается.

Теперь привяжем функцию в качестве триггера к таблице `time_punch`. `BEFORE` здесь ключевой момент. Если выполним этот триггер как триггер `AFTER`, он будет выполнен слишком поздно, чтобы остановить вставку.

```
create trigger check_time_punch before insert on time_punch
    for each row execute procedure fn_check_time_punch();
```

Задание 3:

1. Проверьте работу триггера на корректных и некорректных данных по входу и выходу сотрудника.
2. Найдите “узкие” места в работе триггера.

3. Выполните запрос на просмотр данных в таблице учета времени.

Пример 2. Аудит изменения данных

Вариант 1

Задача: реализовать самую простую систему логирования сотрудников. Она будет следить за изменениями в таблице сотрудников и при изменениях добавлять текстовые экшены в таблицу логов.

Задание 4:

Создайте таблицу для хранения логов `logs` следующей структурой:
`text` (тип `text`),
`added` (тип `timestamp without time zone`)

Триггер будет обрабатывать операции `Update`, `Insert` и `Delete` для таблицы employee, будет выполняться после (`after`) для каждой строки (`for each row`) вызывая функцию `add_to_log()`:

```
CREATE TRIGGER t_employee AFTER INSERT OR UPDATE OR DELETE ON
    employee FOR EACH ROW EXECUTE PROCEDURE add_to_log ();
```

Но до создания триггера нужно сначала создать функцию триггера `add_to_log()`:

```
CREATE OR REPLACE FUNCTION add_to_log() RETURNS TRIGGER AS $$
DECLARE
    mstr varchar(30);
    astr varchar(100);
    retstr varchar(254);
BEGIN
    IF    TG_OP = 'INSERT' THEN
        astr = NEW.id;
        mstr := 'Add new user ';
        retstr := mstr||astr;
        INSERT INTO logs(text,added) values (retstr,NOW());
        RETURN NEW;
    ELSIF TG_OP = 'UPDATE' THEN
        astr = NEW.id;
        mstr := 'Update user ';
        retstr := mstr||astr;
        INSERT INTO logs(text,added) values (retstr,NOW());
        RETURN NEW;
    ELSIF TG_OP = 'DELETE' THEN
        astr = OLD.id;
        mstr := 'Remove user ';
        retstr := mstr || astr;
        INSERT INTO logs(text,added) values (retstr,NOW());
        RETURN OLD;
    END IF;
END;
$$ LANGUAGE plpgsql;
```

Пояснение:

Определяется новая функция, без входящих параметров, возвращает специальный тип TRIGGER.

Для внутреннего использования определяются 3-и переменные в разделе DECLARE.

В самой процедуре внутренняя переменная триггера TG_OP определяет, с какой операцией была вызвана процедура.

В зависимости от операции определяется переменная mstr, собирается строка retstr, которая будет записана в базу данных (обратите внимание, как производится контаминация строк в pgsql, через ||), и делается запись в таблицу логов (INSERT INTO).

Переменные NEW и OLD: Это 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.

Если все правильно сделано, то теперь при любой работе с таблицей employee без нашего участия будет записываться лог действий с таблицей.

Задание 5:

1. Протестируйте работу триггера, выполнив операции вставки, удаления и редактирования данных в таблице сотрудников.
2. Проверьте содержание таблицы логов после выполнения операций.

Вариант 2

Пусть компания хочет воссоздавать в хронологии состояние таблицы на случай обнаружения нарушений.

Таблица аудита выполняет эту роль, отслеживая каждое изменение основной таблицы. Когда в главной таблице обновляется строка, в таблицу аудита вставляется строка в ее предыдущем состоянии.

Используем таблицу time_punch для демонстрации создания и автоматического обновления таблицы аудита с помощью триггеров.

Задание 6:

Создайте таблицу аудита следующей командой:

```
create table time_punch_audit (
  id serial primary key,
  change_time timestamp not null default now(),
  change_employee_id int not null references employee(id),
  time_punch_id int not null references time_punch(id),
  punch_time timestamp not null
);
```

В эту таблицу записывается:

- время обновления прохождения,
- сотрудник, который выполнил обновление,
- ID прохода, который был изменен,
- время прохода до того, как было сделано обновление.

Прежде чем создавать триггер, сначала нужно добавить столбец change_employee_id в таблицу time_punch. Тогда триггер будет знать, какой сотрудник сделал каждое изменение в таблице time_punch.

```
alter table time_punch
add column change_employee_id int null references employee(id);
```

Далее создадим функцию, которая и запишет `OLD` (старое) значение времени прохода в нашу таблицу аудита.

```
create or replace function fn_change_time_punch_audit() returns
trigger as $psql$
  begin
    insert into time_punch_audit (change_time,
       change_employee_id, time_punch_id, punch_time)
       values (now(), new.change_employee_id, new.id,
       old.punch_time);
    return new;
  end;
$psql$ language plpgsql;
```

Триггер для обеспечения аудита:

```
create trigger change_time_punch_audit after update on
time_punch for each row execute procedure
fn_change_time_punch_audit();
```

Функция NOW() возвращает текущую дату и время с точки зрения сервера SQL. Если бы это было привязано к настоящему приложению, можно передавать точное время, когда пользователь фактически сделал запрос, чтобы избежать расхождения из-за задержки.

Для триггера на обновление объект NEW представляет те значения, которые будут содержаться в строке при успешном обновлении. Можно использовать триггер для "перехвата" вставки или обновления простым присвоением своих собственных значений в объект NEW. Объект OLD содержит значения строки до обновления.

<u>Задание 7:</u>

1. Добавьте нового сотрудника в БД.
2. Пусть он будет редактором времени прохода Михаила.
3. Пусть необходимо отредактировать время выхода в 11:30:

```
emp=# select punch_time
emp-#
emp-# from time_punch
emp-#
emp-# where id=2;
     punch_time
---------------------
 2020-01-01 11:30:00
(1 строка)
```

4. Выполните дважды запрос для имитации 2 редакций, которые увеличивают время на 5 минут.

```
update time_punch set punch_time = punch_time + interval '5
minute', change_employee_id = 2 where id = 2;
```

5. Проверьте содержимое таблицы аудита `time_punch_audit`.

Источник: Daniel Lifflander. A Guide to SQL Triggers: Setting up Database Tracking in PostgreSQL