Пользовательские функции

Функция — это подпрограмма, которая принимает входные параметры, выполняет набор операций и возвращает одно значение или набор данных.

В пользовательских функциях набор операций прописывает сам пользователь.

Подпрограмма — это **именованный/идентифицированный фрагмент кода**, который выполняет определённый набор действий и **может быть вызван** из разных частей программы многократно **по своему имени/идентификатору.**
В подпрограмму так же входит понятие "процедура"

Процедура — подобна функции, но имеет некоторые отличия:

- Процедуры, в отличие от функций, не возвращают значение; поэтому в `CREATE PROCEDURE` отсутствует предложение `RETURNS`. Однако процедуры могут выдавать данные в вызывающий код через выходные параметры.
- Функции вызываются как часть запроса или команды DML, а процедуры вызываются отдельно командой CALL.
- 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.
- Некоторые атрибуты функций (например, `STRICT`) неприменимы к процедурам. Эти атрибуты влияют на вызов функций в запросах и не имеют отношения к процедурам.

Процедурная логика - это подход к программированию, при котором выполнение программы представляет собой **последовательность команд**, где разработчик явно указывает как достичь результата, контролируя порядок и способ выполнения операций.
Т.е. подпрограмма выполняет несколько команд. Следующая команда может зависеть от предыдущей, либо они могут быть независимы, но соблюдается строгая последовательность выполнения, которую продумал пользователь.

Функции делятся на:

- **Встроенные** - предопределенные функции, которые доступны в языке программирования по умолчанию, без необходимости их дополнительного подключения или определения. *Т.е. разработчик языка уже прописал набор операций и придумал имя/идентификатор и подал это в готовом виде для использования. Например AVG() - мы пишем имя и в скобках указываем входной параметр, средне

каких данных нужно посчитать, но не прописываем всю формулу - сложить все значения и разделить на количество.

- **Пользовательские** - блок кода, который программист создает и именует сам для выполнения определенной задачи.

Пользовательские функции в Postgres бывают:

- на языке запросов (функции, написанные на SQL)
- функции на процедурных языках (функции, написанные, например, на PL/pgSQL или PL/Tcl) - В них процедурная логика
- внутренние функции
- функции на языке C

Стандартная конструкция пользовательской функции

Простая функция:

1. `CREATE OR REPLACE FUNCTION get_user_count()`
   Создаём или изменяем функцию, даём ей название
2. `RETURNS INTEGER AS`
   Возвращаем значение , указываем какой тим данных оно имеет
3. `$$`
   Знаки для отграничения тела функции
4. `SELECT COUNT(*) FROM users;`
   Команда
5. `$$ LANGUAGE SQL;`
   Знаки для закрытия закрытия тела функции, указание языка, на котором команда написана
   ``

Функция с процедурной логикой:

1. `CREATE OR REPLACE FUNCTION имя_функции(входной параметр1 тип данных,...)`
   Создаём или изменяем функцию, даём ей название,
   прописываем параметры/аргументы и их тип данных
2. `RETURNS возвращаемый_тип AS`
   Возвращаем значение , указываем какой тим данных оно имеет
3. `$$`
   Знаки для отграничения тела функции

4. `DECLARE`
   `переменная 1 тип данных`
   `переменная 2 тип данных;`
   Объявление переменных при необходимости
5. `BEGIN – начало тела функции`
   Начало блока последовательных команд

   `Тут запросы, вычисления и все операции`

   `RETURN название_переменной;`
   После всех операций мы получаем значение, и этому значению присваиваем переменную. Далее RETURN передаёт ее за пределы последовательности к RETURNS, который уже присваивает ей тип данных
6. `END;`
   Конец блока последовательных команд
7. `$$ LANGUAGE plpgsql;`
   Знаки для закрытия закрытия тела функции, указание языка, на котором команда написана

Функции на процедурных языках

Расширение - это пакет SQL-объектов (типы данных, функции, операторы), управляющий файл и опционально динамически загружаемая библиотека, предназначенный для расширения функциональности базы данных. Расширения позволяют легко добавлять поддержку специфических типов данных, упрощать выполнение сложных задач или интегрироваться с другими системами. Управлять ими можно с помощью команд `CREATE EXTENSION` и `ALTER EXTENSION`, а просмотреть список установленных расширений — с помощью команды `\dx`.

Процедурные языки(PL, Procedural Languages) - расширения, которые позволяют создавать пользовательские функции, хранимые процедуры и триггеры с более сложной логикой, чем может обеспечить стандартный язык SQL.
PostgreSQL поставляется со встроенной поддержкой нескольких процедурных языков, таких как PL/pgSQL,PL/Tcl, PL/Perl и PL/Python.

PL/pgSQL - используется в системе управления базами данных PostgreSQL для написания функций, триггеров и процедур. Оно добавляет к стандартному SQL управляющие конструкции, такие как циклы и условные операторы, позволяя создавать более сложную логику работы с данными, чем это возможно при использовании одного только SQL.

Триггер - это указание, что база данных должна автоматически выполнить заданную функцию(**триггерную функцию**), всякий раз когда выполнен определённый тип

операции. *Условие, при котором запускается выполнение функции.*

Условные операторы:

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

- `IF ... THEN ... END IF`
- `IF ... THEN ... ELSE ... END IF`
- `IF ... THEN ... ELSIF ... THEN ... ELSE ... END IF`

и две формы `CASE`:

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

Триггерная функция - 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).
Сначала создаётся сама функция, затем триггер. Она должна быть объявлена без аргументов и возвращать тип `trigger`.

При срабатывании триггерной функции в памяти временно сохраняется несколько данных :

1. `NEW record`
   Новая строка базы данных(новое значение) для команд `INSERT` / `UPDATE` в триггерах уровня строки. В триггерах уровня оператора и для команды `DELETE` эта переменная имеет значение NULL.
2. `OLD record`
   Старая строка базы данных для команд `UPDATE` / `DELETE` в триггерах уровня строки. В триггерах уровня оператора и для команды `INSERT` эта переменная имеет значение NULL.
3. `TG_NAME name`
   Имя сработавшего триггера.
4. `TG_WHEN text`
   Когда срабатывает триггер?
   `BEFORE` (до изменений в таблице) `</span> <span style="color:`
   `orange;">AFTER` (после изменений в таблице)
   `INSTEAD OF` (работает для представлений(views), триггер срабатывает ВМЕСТО стандартного действия (`INSERT` , `UPDATE` , `DELETE`) для обновления представления)

5. `TG_LEVEL` `text`
   Уровень, на котором работает триггер.
   ROW - на уровне строки. Значит срабатывать будет для каждой строки отдельно
   STATEMENT - на уровне всей операции. Значит сработает один раз для операции, независимо от количества строк
6. `TG_OP` `text`
   Операция, для которой сработал триггер: `INSERT`, `UPDATE`, `DELETE` или `TRUNCATE`.
7. `TG_RELID` `oid` (ссылается на `pg_class.oid`)
   OID таблицы, для которой сработал триггер.
   OID таблицы — это уникальный идентификатор (часто числовой) для каждой таблицы в базе данных
8. `TG_RELNAME` `name`
   Таблица, для которой сработал триггер. Эта переменная устарела и может стать недоступной в будущих релизах. Вместо неё нужно использовать `TG_TABLE_NAME`.
9. `TG_TABLE_NAME` `name`
   Таблица, для которой сработал триггер.
10. `TG_TABLE_SCHEMA` `name`
    Схема таблицы, для которой сработал триггер.
11. `TG_NARGS` `integer`
    Число аргументов в команде `CREATE TRIGGER`, которые передаются в триггерную функцию.
12. `TG_ARGV` `text[]`
    Аргументы от оператора `CREATE TRIGGER`, сохранённые в массиве ARRAY. Индекс массива начинается с 0. Для недопустимых значений индекса (меньше 0 или больше или равно `tg_nargs`) возвращается NULL.

Эти данные мы можем использовать в условных операторах, циклах, проверках и пр.

Конструкция триггерной функции и триггера

Триггерная функция:

1. `CREATE OR REPLACE FUNCTION имя_триггерной_функции()`
2. `RETURNS TRIGGER AS`
   В триггерной функции всегда возврат типа trigger
3. `$$`
   Знаки для отграничения тела функции
4. `DECLARE`
   `переменная 1 тип данных`
   `переменная 2 тип данных;`
   Объявление переменных при необходимости

5. `BEGIN - начало тела функции`
   Начало блока последовательных команд
6. `IF OLD.поле IS DISTINCT FROM NEW.поле THEN`
   `INSERT INTO таблица_аудита... (OLD.поле, NEW.поле);`
   `END IF;`
   Пример условного оператора: если старое поле уникально по отношению к новому полю, то внести в таблицу аудита старое поле и новое поле
7. `RETURN NEW;`
   Обязательно возвращаем NEW, OLD или NULL
   Выбор между RETURN NEW и RETURN OLD зависит от типа операции и времени срабатывания триггера.
   Для INSERT всегда NEW
   Для UPDATE. Если NEW - данные изменятся после сохранения. Если OLD - данные останутся прежние, изменения будут отменены, исходная таблица будет такой же, но в таблице логирования появится запись, на что хотели поменять
   Для DELETE - всегда OLD
   Для TRUNCATE - всегда NULL
8. `END;`
   Конец блока последовательных команд
9. `$$ LANGUAGE plpgsql;`
   Знаки для закрытия закрытия тела функции, указание языка, на котором команда написана

Триггер:

1. `CREATE TRIGGER имя_триггера`
2. `[BEFORE | AFTER | INSTEAD OF]`
   Когда он срабатывает по последовательности действий
3. `[INSERT | UPDATE | DELETE | TRUNCATE]`
   На какую операцию реагирует
4. `ON имя_таблицы`
   Операция над какой таблицей?
5. `[FOR EACH ROW | FOR EACH STATEMENT]`
   Уровень срабатывания - для каждой строки, или для всей операции
6. `[WHEN (условие,если есть)]`
7. `EXECUTE FUNCTION имя_триггерной_функции();`
   Ссылка на триггерную функцию, которую он запускает

pg_cron

pg_cron - это расширение PostgreSQL, которое позволяет выполнять SQL команды по расписанию прямо внутри базы данных (аналог cron в Linux, но для БД).
Оно не встроено, его нужно устанавливать. Устанавливается вместе с контейнеров в докер, для установки в Dockerfile и docker-compose прописываются нужные инструкции.

Расширение нужно активировать командой:
`CREATE EXTENSION IF NOT EXISTS pg_cron;`

После активации создаются системные таблицы:

- `cron.job` — список запланированных задач
- `cron.job_run_details` — история выполнения

После чего можно создавать задачу для выполнения по расписанию.
Задача создаётся при помощи функции `cron.schedule` :
```
SELECT cron.schedule( 'расписание', -- ==cron выражение== 'SQL_команда', -- ==команда для выполнения== 'имя_задачи', -- опционально: имя задачи 'описание', -- опционально: описание 'база_данных', -- опционально: база данных 'пользователь', -- опционально: пользователь 'порт' -- опционально: порт );
```

cron выражение - формат расписания для планировщика. Имеет структуру
минута час день*месяца месяц день*недели

Поле	Допустимые значения	Спецсимволы
минута	0-59	`*` `,` `-` `/`
час	0-23	`*` `,` `-` `/`
день месяца	1-31	`*` `,` `-` `/` `?` `L` `W`
месяц	1-12 или JAN-DEC	`*` `,` `-` `/`
день недели	0-7 или SUN-SAT (0 и 7 = воскресенье)	`*` `,` `-` `/` `?` `L` `#`

Специальные символы:

1. `*` - любое значение
2. `,` - перечисление значений одного поля, если несколько вариантов часов, минут или дней и тп.
3. `-` - диапазон значений
4. `/` - шаг (интервал)

Команда для выполнения - sql-команда, запуск функций, процедур и т.д.