**1. Введение**

С постоянным развитием информационных технологий и ростом объемов данных появляется всё большая потребность в мощных и гибких базах данных для обеспечения эффективной работы приложений и хранения информации. В этом контексте реляционные базы данных, такие как PostgreSQL, играют важную роль, обеспечивая надежность, масштабируемость и множество возможностей для хранения и обработки данных.

PostgreSQL – это распределенная объектно-реляционная система управления базами данных с открытым исходным кодом, основанная на

POSTGRES. У неё есть свои «плюсы» и «минусы».

Преимуществами данной СУБД можно считать:

* полную SQL-совместимость;
* постоянная поддержка, активное сообщество PostgreSQL поможет найти решение любой проблемы, связанной с СУБД, в любое время суток;
* возможность программного расширения за счет хранимых процедур;
* объектно-ориентированность;
* бесплатное использование.

Недостатками, которых становится всё меньше и меньше со временем из-за развития PostgreSQL, являются:

* производительность: в некоторых ситуациях производительность PostgreSQL ниже, чем у, например, MySQL;
* невысокая популярность.

В рамках курса «Программирование баз данных» мы изучили много возможностей PostgreSQL, которые позволяют осуществлять основные манипуляции с данные, хранящимися в базе данных. Однако функционал PostgreSQL не ограничивается только тем, что мы узнали на парах, поэтому эта курсовая работа посвящена тем аспектам данной СУБД, о которых важно знать и которые нужно понимать для успешной деятельности в области программирования, но которые мы, к сожалению, не прошли на занятиях в силу ограниченности времени.

1. **Процедуры и функции**

Виды пользовательских функций PostgreSQL:

* функции на языке запросов (функции, написанные на SQL);
* функции на процедурных языках (функции, написанные, например, на PL/pgSQL или PL/Tcl);
* внутренние функции;
* функции на языке C.

Мы подробно разберем функции на языке SQL и в дальнейшем их связь с триггерами.

**Функции**

SQL-функции выполняют произвольный список операторов SQL и возвращают результат последнего запроса в списке. Функции выполняют схожие с функциями из языков программирования таких как C, C++, Python, и т.д. действия. Функции и процедуры обеспечивают возможность повторного использования и гибкость. Функции и хранимые процедуры представляют собой блок кода или запросов, хранящихся в базе данных, которые можно использовать снова и снова. Вместо того чтобы писать одни и те же запросы, удобнее сгруппировать все запросы и сохранить их, чтобы можно было использовать их много раз. А также они плотно интегрированы в саму структуру PostgreSQL, например, с помощью них строятся триггеры, о которых будет рассказано позже.

**Создание функции**

Функции любых видов могут принимать в качестве аргументов базовые типы (integer, text и т.д.), составные типы (массивы) или их сочетания. Кроме того, любые функции могут возвращать значения базового или составного типа. Также можно определить функции, возвращающие наборы базовых или составных значений.

Следует упомянуть, что при создании функции большое число аргументов нужны для определения параметров для языка, на котором описана функция, эти аргументы мы рассмотрим кратко, и остановимся на описании функции с помощью SQL языка.

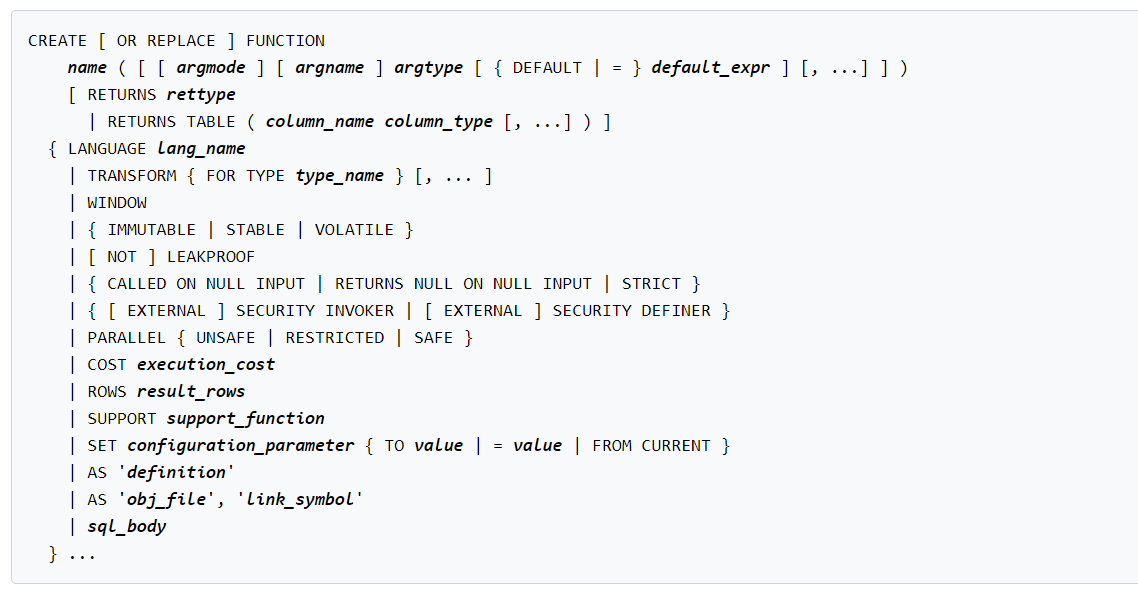


Рисунок 1. Синтаксис функции

Разберем поподробнее аргументы:

* OR REPLACE – заменяет уже существующую функцию вновь созданной. Но не позволяет изменить аргументы и тип возвращаемого объекта функции.
* name – имя аргумента.
* argmode - Режим аргумента: IN (входной), OUT (выходной), INOUT (входной и выходной).
* argtype - Тип данных аргумента функции, при наличии аргументов. Тип аргументов может быть базовым, составным или доменным, либо это может быть ссылка на столбец таблицы.
* default\_expr – возвращаемое выражение по умолчанию, если аргумент не задан явно.
* rettype - тип возвращаемых данных (возможно, дополненный схемой). Это может быть базовый, составной или доменный тип, либо ссылка на тип столбца таблицы. Если функция не должна возвращать значение, в качестве типа результата указывается void. В случае наличия параметров OUT или INOUT, предложение RETURNS можно опустить.
* lang\_name - имя языка, на котором реализована функция. Это может быть SQL, c, internal, либо имя процедурного языка, определённого пользователем, например, PL/pgSQL. Если присутствует тело\_sql, подразумевается язык SQL.
* WINDOW – установка этого флага показывает, что прописывается оконная функция.
* sql\_body – прописывает код функции на SQL. Код SQL можно заключить в кавычки, либо в «$$», второй способ является предпочтительным, так как позволяет избавиться от необходимости экранировать кавычки внутри блока кода.
* TRANSFORM - устанавливает список преобразований типов, применяющихся при вызове процедуры (преобразования между типами SQL и типами данных, специфичными для языков).

**Обзор работы функции**

В единственном случае будет возвращена первая строка результата последнего запроса. Если последний запрос вообще не вернёт строки, будет возвращено значение NULL. Кроме того, можно объявить SQL-функцию как возвращающую множество (то есть, несколько строк), указав в качестве возвращаемого типа функции SETOF ***некий\_тип***, либо объявив её с указанием RETURNS TABLE(***столбцы***). В этом случае будут возвращены все строки результата последнего запроса. Если надо определить функцию SQL, выполняющую действия, ничего не возвращающую – ставим тип void.

**Содержание функций**

Тело SQL-функции должно представлять собой список SQL-операторов, разделённых точкой с запятой. Точка с запятой после последнего оператора может отсутствовать. Если только функция не объявлена как возвращающая void, последним оператором должен быть SELECT, либо INSERT, UPDATE или DELETE с предложением RETURNING. Пример:



Рисунок 2. Таблица products

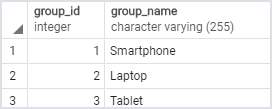


Рисунок 3. Таблица product\_groups

Пользовательская функция:

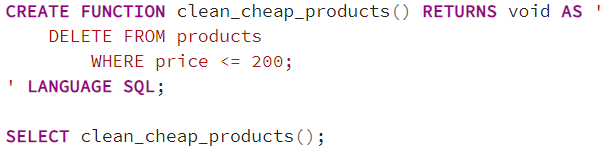


Рисунок 4. Пользовательская функция изменения цены в таблице products

После её выполнения из products удалятся 3 строки, и она вернет пустую таблицу:

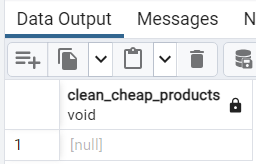


Рисунок 5. Результат пользовательской функции изменения цены в таблице products

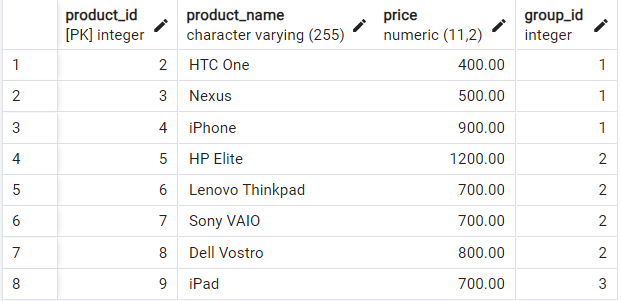


Рисунок 6. Таблица products в результате выполнения функции изменнеия цены

Важное замечание касательно пользовательских функций: прежде чем начинается выполнение команд, разбирается всё тело SQL-функции. Когда SQL-функция содержит команды, модифицирующие системные каталоги (например, CREATE TABLE), действие таких команд не будет проявляться на стадии анализа последующих команд этой функции. Так, например, команды CREATE TABLE foo (...); INSERT INTO foo VALUES(...); не будут работать, как ожидается, если их упаковать в одну SQL-функцию, так как foo не будет существовать к моменту разбору команды INSERT. В подобных ситуациях вместо SQL-функции рекомендуется использовать другой язык: PL/pgSQL.

**Аргументы SQL-функций**

К аргументам функций можно обращаться в теле функции по номерам: «*$n*» (нумерация идет с единицы) или именам «*имя\_аргумента*». Причем если в текущей функции PostgreSQL используется одинаковое имя столбца и аргумента, то столбец будет «перекрывать» аргумент, и обратиться к аргументу можно с помощью вида «*имя\_функции.имя\_аргумента*». Если у аргумента составной тип, то обращаться к полям можно так: «*аргумент.поле*» или «*$1.поле*».

Аргументы могут быть использованы только в качестве данных, но не в качестве идентификатора: например, такое сработает:



Рисунок 7. Сравнение вставки элементов с помощью аргументов функций, вставка значений через аргумент

А такое нет:



Рисунок 8. Сравнение вставки элементов с помощью аргументов функций, попытка обращения через аргумент как через идентификатор

**Работа функций SQL с базовыми типами**

Пример функции сложения двух чисел integer, возвращает integer обозначенный за answer.

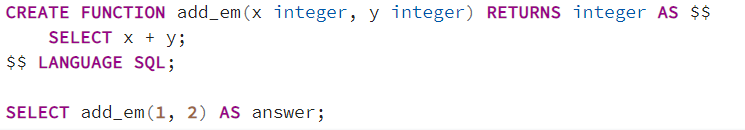


Рисунок 9. Пример функции сложения двух чисел integer

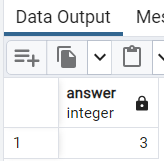


Рисунок 10. Результат выполнения функции сложения двух чисел integer

Если не переобозначать название вывода, оно будет именем функции, в данном случае add\_em.

Работа функций со сложными типами в качестве аргументов и выходных параметров.

В качестве аргументов функций можно задавать составные типы данных, например тут показано как мы редактируем строку формата таблицы products:

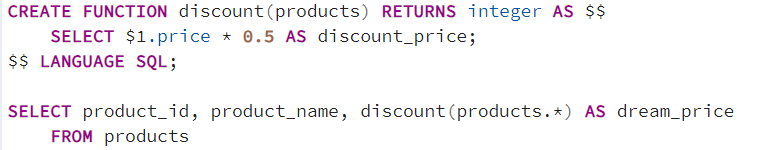


Рисунок 11. Пример использования составных типов данных в функции

Выходные данные будут следующими (с измененными в 2 раза ценами):

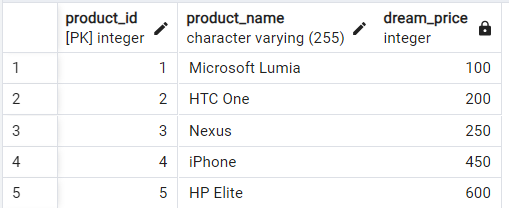


Рисунок 12. Выходные данные функции, использующей составные типы данных.

Или, например, функция, возвращающая вновь созданный продукт (возвращаемая строка составная):

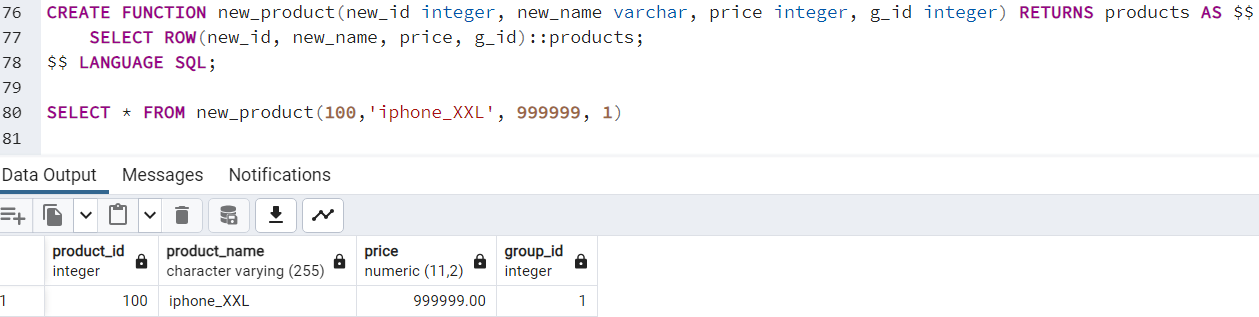


Рисунок 13. Пример функции возвращение составного типа данных

Вывод функции будет следующим:

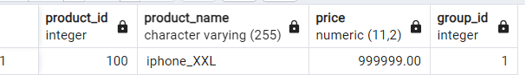


Рисунок 14. Вывод функции, возвращающей составной тип данных

**Функции с выходными параметрами**

Альтернативный способ описать вывод функции – в аргументах прописать OUT. Таким образом можно обозначать возвращаемые параметры. Также можно указывать их значение по умолчанию, либо делать произвольное число параметров.

Пример функции с выходными параметрами:

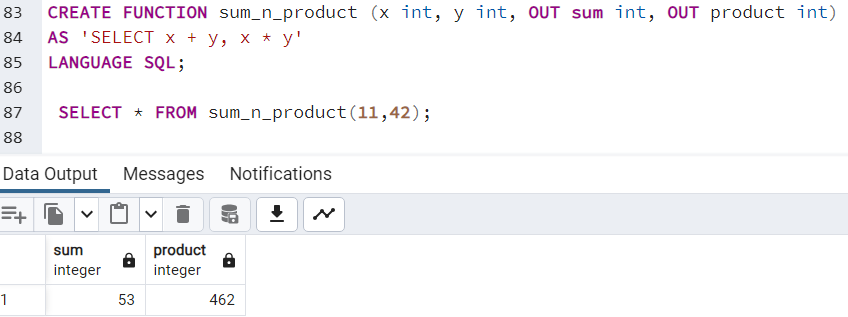


Рисунок 15. Пример функции с выходными параметрами



Рисунок 16. Вывод функции с выходными параметрами

Пример функции с заданными по умолчанию параметрами:

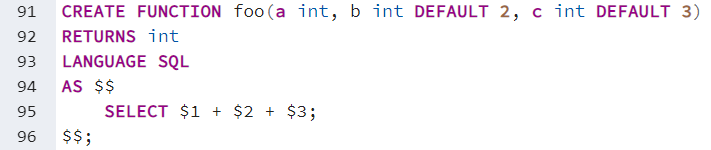


Рисунок 17. Пример функции с заданными по умолчанию параметрами

Тогда вывод будет в зависимости от аргументов разный:

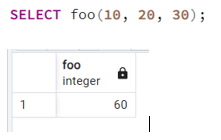


Рисунок 18. Результат выполнения функции с заданными по умолчанию параметрами, через явную передачу всех параметров

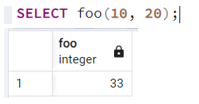


Рисунок 19. Результат выполнения фнукции с заданными по умолчанию параметрами, без явной передачи всех параметров

**Пользовательские процедуры**

Отличие процедур от функций:

* 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 (SELECT, INSERT, UPDATE, DELETE), 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.

**Создание процедуры**

При определении ключевых выражений внутри создания процедуры пропустим те, которые уже были определены в создании функций, так как они выполняют ровно те же самые действия.

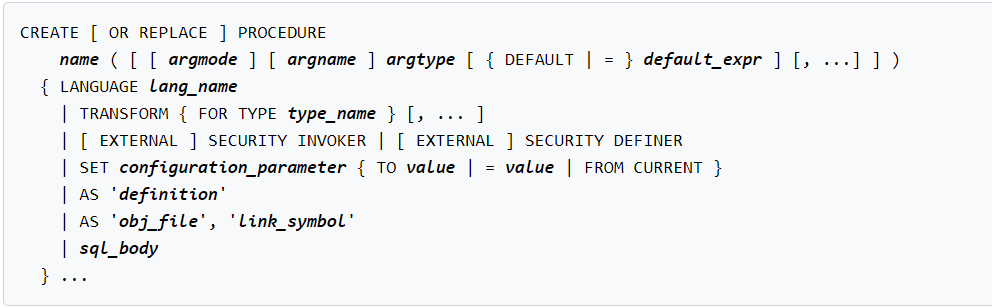


Рисунок 20. Документация функции создания процедуры

Как и при создании функций, вызов OR REPLACE не дает изменить входные и выходные аргументы. Однако если попытаться нарушить это правило – будет создана новая отдельная процедура.

* SET value - при вызове процедуры указанный параметр конфигурации configuration\_parameter будет установлен в значение value, а после выхода из процедуры – вернется к изначальному значению. Если установлено SET FROM CURRENT, то будет использовано значение, которое имеет параметр, на этапе выполнения CREATE PROCEDURE
* definition – строка-константа определение самого кода реализации процедуры. Её значение зависит от языка. Это может быть имя внутренней процедуры, путь к объектному файлу, команда SQL или код на процедурном языке.
* obj\_file, link\_symbol – нужны для загрузки процедуры из скомпилированного файла C.
* sql\_body – код процедуры в стиле языка SQL, подобно definition, но работает только в стиле SQL.

**Примеры процедур**

Различие процедуры от функции можно проследить на следующем примере возвращаемых параметров. Для процедуры они должны быть включены в аргументы через OUT:

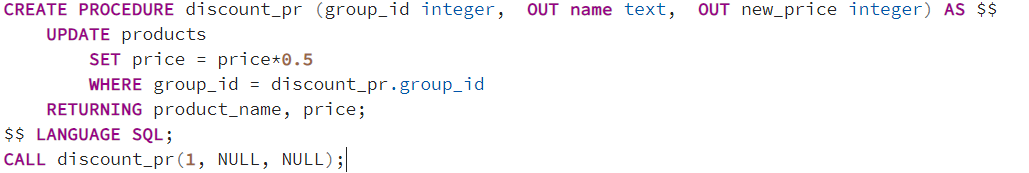


Рисунок 21. Процедура, прописанная через возвращение параметров через OUT

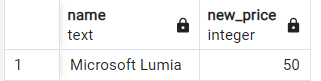


Рисунок 22. Результат выполнения процедуры с передачей параметров через OUT

Пример процедуры вставки данных в таблицу products.

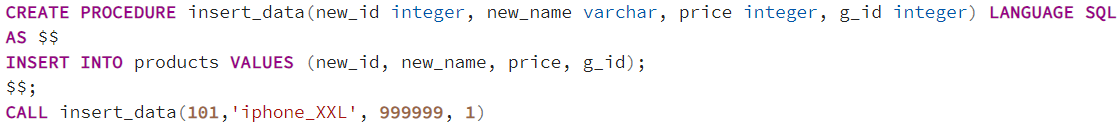


Рисунок 23. Процедура вставки новых данных в таблицу products.

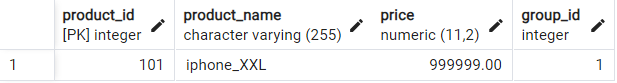
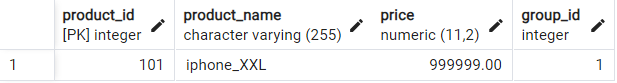


Рисунок 24. Результат выполнения процедуры вставки новых данных в таблицу products



25. Результат выполнения процедуры вставки новых данных в таблицу products

**3. Работа с триггерами**

Триггер определяет операцию, которая должна выполняться при наступлении некоторого события в базе данных. Триггеры срабатывают при выполнении с таблицей команды SQL INSERT, UPDATE или DELETE.

В PostgreSQL триггеры создаются на основе существующих функции или процедур, т.е. сначала командой CREATE FUNCTION определяется триггерная функция, затем на ее основе командой CREATE TRIGGER определяется собственно триггер.

Полный синтаксис запроса создания триггера следующим образом:

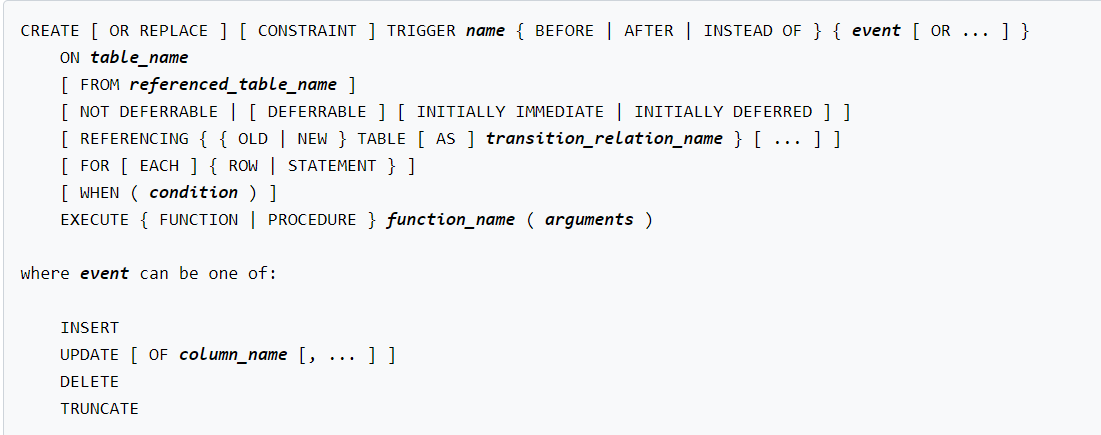


Рисунок 26. Синтаксис триггера

Разберём поподробнее, что означают некоторые основные слова в данном запросе:

* CREATE TRIGGER name. В аргументе name 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.
* REPLACE – заменит уже существующий триггер.
* {BEFORE | AFTER | INSTEAD OF}. 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. Ключевое слово INSTEAD OF указывает выполнить действия из триггера вместо операции, на которую он сработал. Работает это ключевое слово в случае вставки, обновления или удаления в представлении.
* {event [OR …]} – указываются события, на которые срабатывает триггер. Такими событиями могут быть INSERT, UPDATE [OF column\_name [, ... ]].
* ON table\_name – имя таблицы, к которой применяется операция, на которую срабатывает триггер.
* FROM referenced\_table\_name. Имя другой таблицы, на которую ссылается ограничение. Эта опция используется для ограничений внешнего ключа и не рекомендуется для общего использования. Данную опцию можно указать только для триггеров ограничений.
* REFERENCING – далее указывается имена таблиц – переходных отношений появляющихся при выполнении целевого оператора (OLD TABLE, NEW TABLE).
* transition\_relation\_name – имя для переходного отношения.
* Если триггер помечен опцией FOR EACH ROW, тогда функция вызывается для каждой строки, которая изменяется в результате события. Например, если сделать UPDATE для 10 строк, триггерная функция UPDATE будет вызываться 10 раз, по одному разу для каждой обновлённой строки. Опция FOR EACH STATEMENT вызовет функцию только один раз для каждого оператора, независимо от количества изменяемых строк.
* condition – выражение, если True – функция триггера будет выполняться. Может ссылаться на OLD|NEW TABLE, если установлено FOR EACH ROW. Триггеры INSTEAD OF не поддерживают условия WHEN.
* EXECUTE function\_name – функция определенная как функция без аргументов, возвращающая integer.
* arguments – опциональные предоставленные пользователем аргументы, которые будут переданы функции при срабатывании триггера.

Важным является замечание, что для создания функции для триггера, необходимо будет прописать её реализацию на одном из языков программирования (не SQL).

**Триггерные функции**

PostgreSQL, хоть триггеры и подразумевают написание пользовательских триггерных функций, также содержит и встроенные функции-триггеры.

* suppress\_redundant\_updates\_trigger() - предотвращает изменения, не меняющие данные.
* tsvector\_update\_trigger() - автоматически обновляет содержимое столбца tsvector из связанных столбцов с обычным текстовым содержимым. Конфигурация текстового поиска, которая будет использоваться, задаётся по имени в аргументе триггера.
* tsvector\_update\_trigger\_column() - автоматически обновляет содержимое столбца tsvector из связанных столбцов с обычным текстовым содержимым. Конфигурация текстового поиска, которая будет использоваться, определяется содержимым столбца regconfig целевой таблицы.

Рассмотрим подробнее одну из них - suppress\_redundant\_updates\_trigger(). Данная функция, применяемая в качестве триггера BEFORE UPDATE на уровне строк, предотвратит внесение изменений, при которых данные в строке фактически не меняются. Тем самым переопределяется обычное поведение, когда изменение физической строки происходит вне зависимости от того, были ли изменены данные. (Обычное поведение не предполагает сравнения данных, поэтому операции изменения выполняются быстрее, и в ряде случаев именно это поведение желательно.)

В идеале следует избегать операций изменения, которые фактически не меняют данные в записях. Подобные ненужные изменения могут обходиться дорого, особенно когда требуется обновлять множество индексов, к тому же впоследствии базу данных придётся очищать от «мёртвых» строк. Однако выявить такие изменения в клиентском коде бывает сложно, если вообще возможно, а при составлении соответствующих проверочных выражений легко допустить ошибку. Использовать данную следует с осторожностью. Данный триггер выполняется для каждой записи довольно быстро, но всё же не мгновенно, так что, если большинство затронутых записей фактически изменяется, с этим триггером операция изменения в среднем замедлится.

**Примеры использования триггеров**

Разберём на конкретном примере функцию suppress\_redundant\_updates\_trigger(). Имеется таблица:

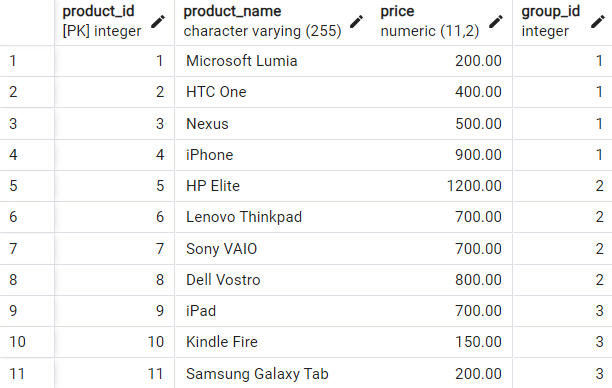


Рисунок 27.Таблица products

В магазине началась акция и некоторые устройства изменили свою цену, поэтому необходимо изменить цену в таблице. Однако мы знаем, что большинство цен остались прежними, а в таблице много строк, поэтому мы можем написать такой триггер для изменения данной таблицы:

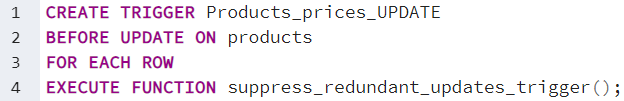


Рисунок 28. Создание триггера №1

Теперь при очередной акции мы быстрее обновим все цены на продукты в магазине.

Пример триггера с триггерной функцией, который устанавливает дату создания продукта с помощью функции NOW():

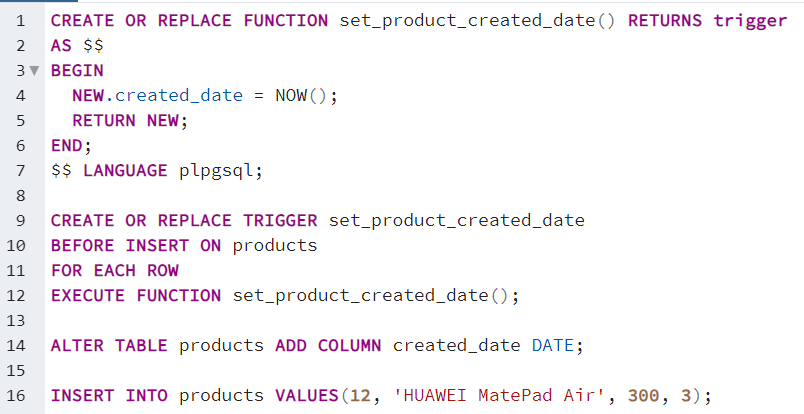


Рисунок 29. Создание триггера №2

NEW обозначает новую строку, которую мы хотим вставить в таблицу. Пришлось также добавить новый столбец в исходную таблицу. После запуска запросов выше получаем вот такой результат работы:

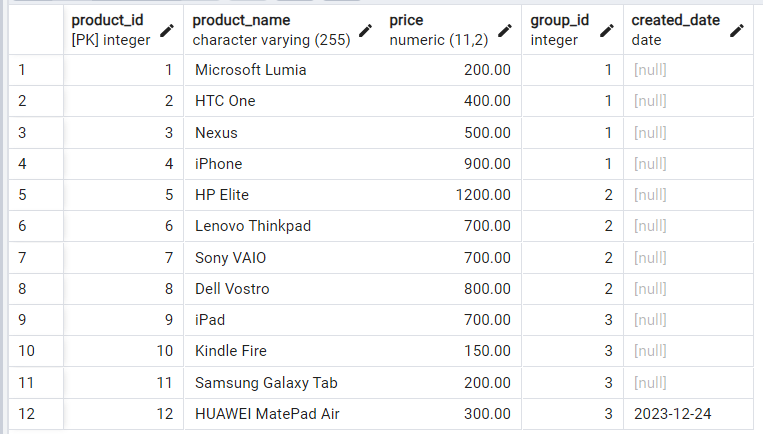


Рисунок 30. Результат работы триггера №2

Имеется также вторая таблица product\_groups, которая изначально имеется данный вид:

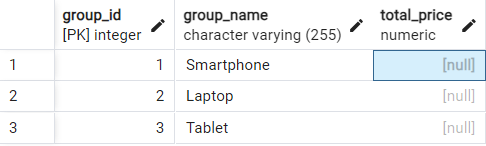


Рисунок 31. Таблица product\_groups

На неё ссылается первая таблица по столбцу group\_id. Добавим к данной таблице ещё один столбец total\_price, который будет показывать суммарную стоимость устройств из первой таблице в группе, и создадим триггер с функцией, который будет этот столбец обновлять при изменении данных в строке или самих строк:

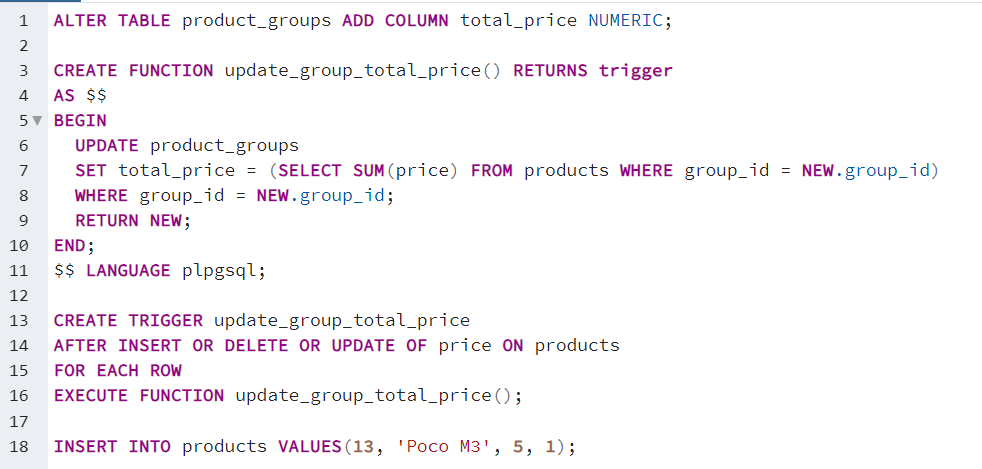


Рисунок 32. Создание триггера №3

То есть функция в триггере просто пересчитывает значение суммы цен из тех строк, в которых group\_id равно group\_id изменяемой строки. После выполнения всех запросов сверху, таблица product\_groups будет выглядеть следующим образом:

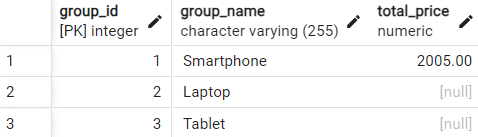


Рисунок 33.Таблица product\_groups после запроса №3

Предположим, что у нас нет связи между двумя таблицами из примеров выше в виде внешнего ключа, но мы всё равно не хотим удалять те строчки из таблицы product\_groups, group\_id которых присутствует и во второй таблице в какой-либо строке. Для этого мы можем создать триггер на удаление данных из таблицы product\_groups, который будет прерывать транзакцию и выводить сообщение об ошибке с помощью RAISE EXCEPTION:

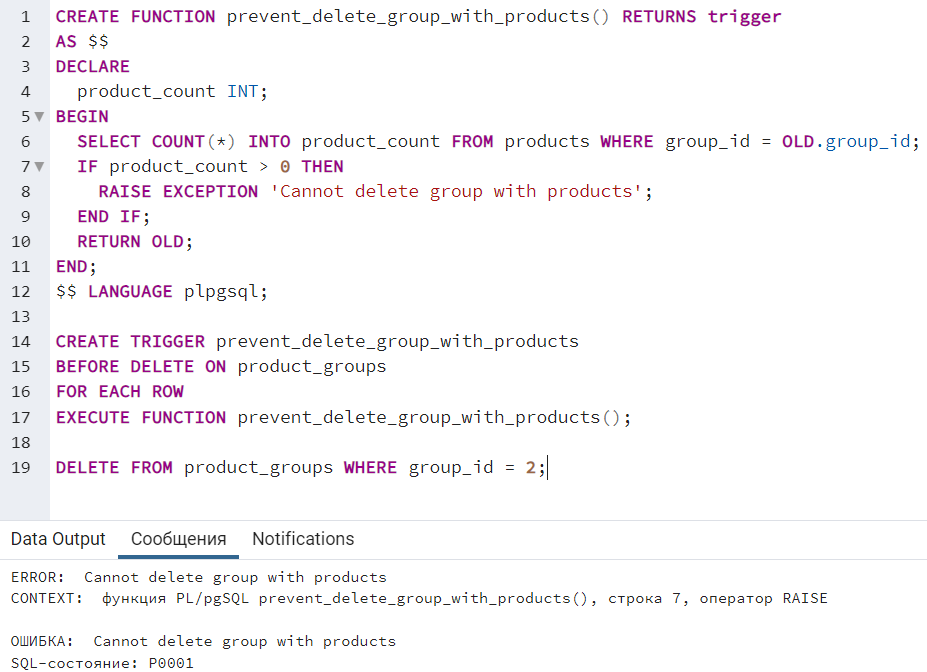


Рисунок 34. Создание триггера №4

Если же мы теперь добавить новую строку с новым group\_id, с которым нет строчек в другой таблице, то мы сможем успешно её удалить:

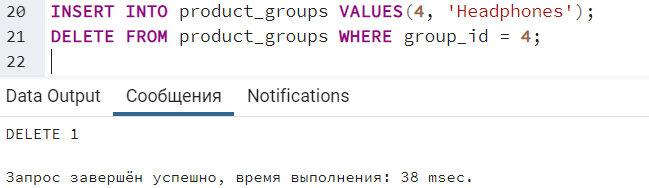


Рисунок 35. Результат работы запросы с триггером №4

**4. Заключение**

Таким образом, мы рассмотрели функционал процедур, функций, триггеров и триггерных функций и можем сделать вывод, что использование функций и процедур заметно упрощает работу с базой данных, поскольку дает возможность не только сократить и сделать более удобным использование SQL, позволяя реализовать продвинутую логику базы данных, которые не всегда удобно и вообще возможно реализовать с помощью SQL (в случае использования языков программирования, как, например, PL/pgSQL).

Также использование функций и процедур позволяет реализовывать более продвинутые способы работы с базой данных, а именно триггеры. Использование триггеров оптимизирует работу с базой данных, так как позволять выполнять меньше операций вручную, может обеспечить целостность данных, отслеживает изменения базы данных, но иногда триггеры могут и замедлить скорость выполнения запросов на больших объёмах данных, поэтому надо подходить к их использованию с умом.

Также разобрана работа с полнотекстовым видом данных, а также функционал полезных функций взаимодействия с ним: ts\_rewrite и ts\_stat, которые предоставляют преимущества в обработке текстовых данных, а именно: более гибкий поиск, анализ запросов и т.д.

**6. Список использованных источников**

1. Компания Postgres Professional [Электронный ресурс]. URL: postgrespro.ru (Дата обращения: 24.11.2023).
2. Официальный сайт PostgreSQL[Электронный ресурс]. URL: https://www.postgresql.org (Дата обращения: 24.11.2023).
3. Папилина Т.М., Прядко С.А., Фомочкина А.С. Основы построения запросов на языке SQL / Папилина Т.М., Прядко С.А., Фомочкина А.С. — М. РГУ нефти и газа (НИУ) имени И.М. Губкина, 2021. — 171 с., 1,67 Мб (Библиотека РГУ нефти и газа) [Электронный ресурс] URL: http://elib.gubkin.ru/content/24806 (режим доступа: по паролю)
4. Создание триггеров в PostgreSql [Электронный ресурс]. URL: https://ciu.nstu.ru/kaf/persons/1914/study/baz\_dannh/trigger (Дата обращения: 24.11.2023).
5. Функции и хранимые процедуры в SQL: зачем нужны и как применять в реальных примерах [Электронный ресурс]. URL: https://proglib.io/p/funkcii-i-hranimye-procedury-v-sql-zachem-nuzhny-i-kak-primenyat-v-realnyh-primerah-2022-10-07 (Дата обращения: 24.11.2023).