

Севастопольский государственный университет
Кафедра «Информационные системы»

Управление данными

курс лекций

лектор:
ст. преподаватель кафедры ИС Абрамович А.Ю.



Лекция 2

Язык SQL.

Триггеры.

Функции и хранимые процедуры.

Триггерные функции

ТРИГГЕРЫ

подпрограммы, которые всегда выполняются автоматически на стороне сервера, в ответ на изменение данных в таблицах БД. Это методы, с помощью которых разработчик может обеспечить целостность БД.

Триггер активизируется при попытке изменения данных в таблице, для которой **определен**. SQL выполняет эту процедуру при операциях добавления, обновления и удаления (**INSERT, UPDATE, DELETE**) в данной таблице.

Наиболее общее применение триггера – поддержка целостности в базах данных. Триггеры незначительно влияют на производительность сервера и часто используются для усиления предложений, выполняющих многокаскадные операции в таблицах и строках.

Триггер может выполняться в трех фазах изменения данных: до (**before**) какого-то события, после (**after**) него или вместо операции (**instead of**).

ПОСТРОЧНЫЙ ТРИГГЕР -

триггерная функция вызывается **один раз для каждой строки**, затронутой оператором, запустившим триггер.

Триггер уровня строк

ОПЕРАТОРНЫЙ ТРИГГЕР

вызывается **только один раз** при выполнении соответствующего оператора, **независимо от количества строк**, которые он затрагивает.

Триггер уровня оператора

- **Триггеры BEFORE уровня оператора** срабатывают **до того**, как оператор начинает делать что-либо, тогда как **триггеры AFTER уровня оператора** срабатывают **в самом конце работы оператора**. Эти типы триггеров могут быть определены **для таблиц, представлений или сторонних таблиц**.
- **Триггеры BEFORE уровня строки** срабатывают непосредственно **перед обработкой конкретной строки**, в то время как **триггеры AFTER уровня строки** срабатывают **в конце работы всего оператора** (но до любого из триггеров AFTER уровня оператора). Эти типы триггеров могут определяться **только для таблиц, в том числе сторонних, но не для представлений**.
- **Триггеры INSTEAD OF** могут определяться **только для представлений и только на уровне строк**: они срабатывают **для каждой строки сразу после того как строка представления идентифицирована как подлежащая обработке**.

```
CREATE [ OR REPLACE ] [ CONSTRAINT ] TRIGGER name
{ BEFORE | AFTER | INSTEAD OF } { event [ OR ... ] }
  ON table_name
  [ FROM referenced_table_name ]
  [ NOT DEFERRABLE | [ DEFERRABLE ]
[ INITIALLY IMMEDIATE | INITIALLY DEFERRED ] ]
  [ REFERENCE { { OLD | NEW } TABLE [ AS ] transition_relation_name }
[... ] ]
  [ FOR [ EACH ] { ROW | STATEMENT } ]
  [ WHEN ( condition ) ]
  EXECUTE { FUNCTION | PROCEDURE } function_name ( arguments )
```

где **событие (event)** может быть одним из следующих:

```
INSERT
UPDATE [ OF column_name [, ... ] ]
DELETE
TRUNCATE
```

CREATE TRIGGER создаёт **новый** триггер, а **CREATE OR REPLACE TRIGGER** создаёт **новый триггер или заменяет существующий**. Триггер будет связан с указанной таблицей, представлением или сторонней таблицей и будет выполнять заданную функцию **function_name** при определённых операциях с этой таблицей.

Триггер с пометкой **FOR EACH ROW** вызывается **один раз для каждой строки**, изменяемой в процессе операции. Триггер с пометкой **FOR EACH STATEMENT** вызывается **только один раз для конкретной операции**, вне зависимости от того, как много строк она изменила (при выполнении операции, изменяющей ноль строк, всё равно будут вызваны все триггеры FOR EACH STATEMENT).

Триггеры, срабатывающие в режиме **INSTEAD OF**, должны быть помечены **FOR EACH ROW** и могут быть определены только для представлений. Триггеры **BEFORE** и **AFTER** для **представлений** должны быть помечены **FOR EACH STATEMENT**.

В определении триггера **можно указать логическое условие WHEN**, которое определит, **вызывать триггер или нет**. В триггерах на уровне строк условия WHEN могут **проверять старые и/или новые значения столбцов в строке**. Триггеры на уровне оператора так же могут содержать условие WHEN, **хотя для них это не имеет смысла**, так как в этом условии нельзя ссылаться на какие-либо значения в таблице.

ФУНКЦИИ И ХРАНИМЫЕ ПРОЦЕДУРЫ В SQL

Функции и хранимые процедуры в SQL, обеспечивают возможность повторного использования и гибкость. Представляют собой блок кода или запросов, хранящихся в базе данных, которые можно использовать снова и снова. Что касается гибкости, в момент, когда происходит изменение логики запросов, можно передавать новый параметр функциям и хранимым процедурам.

Хранимая процедура (ХП) — это программный модуль, который может быть вызван с клиента, из другой процедуры, функции, выполнимого блока (executable block) или триггера. **Хранимые процедуры могут принимать и возвращать множество параметров.**

Функция является программой, хранящейся в области метаданных базы данных и выполняющейся на стороне сервера. К хранимой функции могут обращаться хранимые процедуры, хранимые функции (в том числе и сама к себе), триггеры и клиентские программы. **В отличие от хранимых процедур хранимые функции всегда возвращают одно скалярное значение.** Для возврата значения из хранимой функции **используется оператор RETURN**, который немедленно прекращает выполнение функции.

ФУНКЦИИ

- Функция **имеет возвращаемый тип** и **возвращает значение**
- Использование DML (insert, update, delete) запросов внутри функции невозможно. **В функциях разрешены только SELECT-запросы**
- Функция **не имеет выходных аргументов**
- **Вызов хранимой процедуры из функции невозможно**
- **Вызов функции** внутри **SELECT** запросов **возможен**

ХРАНИМЫЕ ПРОЦЕДУРЫ

- Хранимая процедура **не имеет возвращаемого типа, но имеет выходные аргументы**
- **Использование DML-запросов** (insert, update, delete) **возможно** в хранимой процедуре.
- Хранимая процедура **имеет и входные, и выходные аргументы**
- **Использование или управление транзакциями возможно** в хранимой процедуре
- **Вызов хранимой процедуры** из **SELECT** запросов **невозможно**

Оператор **CREATE FUNCTION** создаёт новую хранимую функцию. Имя хранимой функции должно быть уникальным среди имён всех хранимых функций и внешних функций.

```
CREATE Имя хранимой функции. Может содержать до 63 символов. [or REPLACE] FUNCTION funcname Описание входного параметра. [(<inparam>)]  
  Тип данных, который возвращает функция. RETURNS <type> [COLLATE collation]  
  LANGUAGE plpgsql  
  AS  
  $$  
  DECLARE -- variable declaration  
  BEGIN -- logic  
  END;  
  $$
```

CREATE FUNCTION является составным оператором, состоящий из заголовка и тела. **Заголовок** определяет имя хранимой функции, объявляет входные параметры и тип возвращаемого значения. **Тело функции** состоит из необязательных объявлений локальных переменных, подпрограмм и именованных курсоров, и одного или нескольких операторов, или блоков операторов, заключённых во внешний блок, который начинается с ключевого слова BEGIN, и завершается ключевым словом END.

ВХОДНЫЕ ПАРАМЕТРЫ

```
CREATE FUNCTION ADD_INT(A INT, B INT DEFAULT 0)
RETURNS INT DETERMINISTIC LANGUAGE plpgsql
AS
$$
BEGIN RETURN A+B;
END
$$
```

Входные параметры заключаются в скобки после имени функции. Они передаются в функцию **по значению** (любые изменения входных параметров внутри функции никак не повлияет на значения этих параметров в вызывающей программе).

У каждого параметра указывается тип данных (для параметра можно указать ограничение NOT NULL, тем самым запретив передавать в него значение NULL). Для параметра **строкового типа существует возможность задать порядок сортировки** с помощью предложения COLLATE.

Входные параметры **могут иметь значение по умолчанию**. Параметры, для которых заданы значения, должны располагаться в конце списка параметров.

Использование доменов при объявлении параметров

В качестве типа параметра можно указать имя домена. В этом случае **параметр будет наследовать все характеристики домена.**

Если перед названием домена дополнительно используется предложение **TYPE OF**, то используется только тип данных домена — не проверяется (не используется) его ограничение (если оно есть в домене) на NOT NULL, CHECK ограничения и/или значения по умолчанию.

Использование типа столбца при объявлении параметров

Входные и выходные параметры можно объявлять, используя тип данных столбцов существующих таблиц и представлений. Для этого используется предложение **TYPE OF COLUMN**, после которого указывается имя таблицы или представления и через точку имя столбца.

При использовании TYPE OF COLUMN наследуется только тип данных, а в случае строковых типов ещё и набор символов, и порядок сортировки. **Ограничения и значения по умолчанию столбца никогда не используются.**

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

```
CREATE FUNCTION ADD_INT(A INT, B INT DEFAULT 0)
RETURNS INT DETERMINISTIC LANGUAGE plpgsql
$$
AS
BEGIN RETURN A+B;
END
$$
```

Предложение RETURNS задаёт тип возвращаемого значения хранимой функции.

Если функция возвращает значение строкового типа, то существует возможность задать порядок сортировки с помощью предложения COLLATE. **В качестве типа выходного значения можно указать имя домена, ссылку на его тип (с помощью предложения TYPE OF) или ссылку на тип столбца таблицы (с помощью предложения TYPE OF COLUMN).**

Детерминированные функции

Необязательное предложение **DETERMINISTIC** указывает, что функция детерминированная.

Детерминированные функции каждый раз возвращают один и тот же результат, если предоставлять им один и тот же набор входных значений. Недетерминированные функции могут возвращать каждый раз разные результаты, даже если предоставлять им один и тот же набор входных значений.

Если для функции указано, что она является **детерминированной**, то такая функция не вычисляется заново, если она уже была вычислена однажды с данным набором входных аргументов, а берет свои значения из кэша метаданных (если они там есть).

В текущей версии Firebird, не существует кэша хранимых функций с маппингом входных аргументов на выходные значения. Указание инструкции **DETERMINISTIC** на самом деле нечто вроде «обещания», что код функции будет возвращать одно и то же. В данный момент детерминистическая функция считается инвариантом и работает по тем же принципам, что и другие инварианты (вычисляется и кэшируется на уровне текущего выполнения данного запроса).

ТЕЛО ФУНКЦИИ

```
CREATE FUNCTION ADD_INT(A INT, B INT DEFAULT 0)
RETURNS INT DETERMINISTIC LANGUAGE plpgsql
AS
$$
BEGIN RETURN A+B;END
$$
```

После ключевого слова **AS** следует тело хранимой функции.

В необязательной секции **<declarations>** описаны локальные переменные функции, именованные курсоры и подпрограммы (подпроцедуры и подфункции). Локальные переменные подчиняются тем же правилам, что и входные параметры функции в отношении спецификации типа данных.

После необязательной секции деклараций обязательно следует **составной оператор**. Составной оператор состоит из одного или нескольких операторов, заключенных между **ключевыми словами BEGIN и END**. Составной оператор может содержать один или несколько других составных операторов. **Вложенность ограничена 512 уровнями**. Любой из **BEGIN ... END** блоков может быть пустым, в том числе и главный блок

Кто может создать функцию?

Выполнить оператор CREATE FUNCTION могут:

- администраторы;
- пользователи с привилегией CREATE FUNCTION.

Пользователь, создавший функцию, **становится её владельцем**.

ПРИМЕР: запросить имя пользователя и его самые дорогие покупки.

```
sbrmvch=# select * from purchases;
```

id	name	cost	user_id
1	M1 MacBook Air	1300.99	1
2	Iphone 14	1200.00	2
3	Iphon 10	700.00	3
4	Iphone 13	800.00	1
5	Intel Core i5	500.00	4
6	M1 MacBook Pro	1500.00	5
7	IMAC	2500.00	7
8	ASUS VIVOBOOK	899.99	6
9	Lenovo	1232.99	1
10	Galaxy S21	999.99	2
11	XIAMI REDMIBOOK 14	742.99	4
12	M1 MacBook Air	1299.99	8
13	ACER	799.99	7

(13 rows)

```
sbrmvch=# select * from users;
```

id	name	profession
1	Bob	QA
2	Camilo	Front End developer
3	Billy	Backend Developer
4	Alice	Mobile Developer
5	Kate	QA
6	Wayne	DevOps
7	Tim	Mobile Developer
8	Amigos	QA

(8 rows)

```

CREATE OR REPLACE FUNCTION findMostExpensivePurchase(customer_id int)
    RETURNS numeric(10, 2)
    LANGUAGE plpgsql
AS
$$
DECLARE
    itemCost numeric
begin
    SELECT MAX(cost)
    FROM purchases
    WHERE user_id = customer_id
    RETURN itemCost;
end;
$$;

```

sbrmvch=# CREATE OR REPLACE FUNCTION findMostExpensivePurchase(customer_id int)
sbrmvch=# RETURNS numeric(10, 2)
sbrmvch=# LANGUAGE plpgsql
sbrmvch=# AS
sbrmvch=# \$\$
sbrmvch\$# DECLARE
sbrmvch\$# itemCost numeric(10, 2);
sbrmvch\$# begin
sbrmvch\$# SELECT MAX(cost)
sbrmvch\$# INTO itemCost
sbrmvch\$# FROM purchases
sbrmvch\$# WHERE user_id = customer_id;
sbrmvch\$# RETURN itemCost;
sbrmvch\$# end;
sbrmvch\$# \$\$;
CREATE FUNCTION

Чтобы вызвать функцию — необходимо выполнить следующую команду:

```
SELECT findMostExpensivePurchase(1) as mostExpensivePurchase;
```

```
[sbrmvch=# SELECT findMostExpensivePurchase(1) as mostExpensivePurchase;
```

```
mostexpensivepurchase
```

```
-----
```

```
1300.99
```

```
(1 row)
```

```
SELECT name, findMostExpensivePurchase(id) as purchase
       from users;
```

```
[sbrmvch=# SELECT name, findMostExpensivePurchase(id) as purchase from users;
```

```
name | purchase
```

```
-----+-----
```

```
Bob   | 1300.99
```

```
Camilo | 1200.00
```

```
Billy | 700.00
```

```
Alice | 742.99
```

```
Kate  | 1500.00
```

```
Wayne | 899.99
```

```
Tim   | 2500.00
```

```
Amigos | 1299.99
```

```
(8 rows)
```

ALTER FUNCTION

Оператор **ALTER FUNCTION** позволяет **изменять состав и характеристики входных параметров**, типа выходного значения, локальных переменных, именованных курсоров, подпрограмм и тело хранимой функции.

```
ALTER FUNCTION funcname [(<inparam> [, <inparam> ...])]
```

```
RETURNS <type> [COLLATE collation]
```

```
[DETERMINISTIC]
```

```
<routine-body>
```

Будьте осторожны при изменении количества и типов входных параметров хранимых функций. Существующий код приложения может стать неработоспособным из-за того, что формат вызова функции несовместим с новым описанием параметров.

Выполнить оператор ALTER FUNCTION могут:

- администраторы;
- владелец хранимой функции;
- пользователи с привилегией ALTER ANY FUNCTION.

DROP FUNCTION

```
DROP FUNCTION funcname
```

Оператор **DROP FUNCTION** **удаляет существующую хранимую функцию**. Если от хранимой функции существуют зависимости, то при попытке удаления такой функции будет выдана соответствующая ошибка.

Оператор **CREATE PROCEDURE** создаёт новую **хранимую процедуру**. Имя хранимой процедуры должно быть уникальным среди имён всех хранимых процедур, таблиц и представлений базы данных.

```
CREATE [OR REPLACE] PROCEDURE procedure_name(parameter_list)
LANGUAGE language_name
AS $
    stored_procedure_body;
$;
```

Создание хранимой процедуры, почти такое же, как создание функции с небольшим отличием — **в ней нет return**. Остальное почти идентично.

CREATE PROCEDURE является составным оператором, состоящий из **заголовка и тела**. Заголовок определяет имя хранимой процедуры и объявляет входные параметры.

Тело процедуры состоит из **необязательных объявлений** локальных переменных, подпрограмм и именованных курсоров, и одного или нескольких операторов, или блоков операторов, заключённых во внешний блок, который начинается с ключевого слова **BEGIN**, и завершается ключевым словом **END**.

Функции позволяют выполнять только SELECT-запросы, а хранимые процедуры позволяют выполнять INSERT, UPDATE, DELETE операции. **Хранимые процедуры очень удобны при работе со случаями, когда необходимы операции INSERT, UPDATE ИЛИ DELETE.**

ПРИМЕР: банковские переводы.

```
[sbrmvch=# select * from accounts;
```

id	balance	user_id
1	1500	1
2	1100	2
3	2300	3
4	7500	5
5	6500	4

(5 rows)

```
[sbrmvch=# select * from users;
```

id	name	profession
1	Bob	QA
2	Camilo	Front End developer
3	Billy	Backend Developer
4	Alice	Mobile Developer
5	Kate	QA
6	Wayne	DevOps
7	Tim	Mobile Developer
8	Amigos	QA

(8 rows)

```

CREATE OR REPLACE PROCEDURE transfer(sourceAccountId bigInt,
destinationAccount bigInt, amount Integer)
language plpgsql
as $$
begin
    update accounts
    set balance = balance - amount
    where id = sourceAccountId;

    update accounts
    set balance = balance + amount
    where id = destinationAccountId;

    commit;
end;
$$;

```

В приведенном примере показано **создание процедуры — transfer()**, которая принимает три параметра. Сразу после имени процедуры передаются аргументы с соответствующими типами данных — sourceAccountId, destinationAccountId, сумма. **Процедура вычитает переданную сумму из одного account и добавляет ее к другому account.**

Для вызова хранимой процедуры используется — `call procedure_name()` .

```
call transfer (5, 4, 2000);
```

```
[sbrmvch=# call transfer (5, 4, 2000);
```

```
[sbrmvch=# select * from accounts;
```

id	balance	user_id
1	1500	1
2	1100	2
3	2300	3
4	7500	5
5	6500	4

(5 rows)



```
[sbrmvch=# select * from accounts;
```

id	balance	user_id
1	1500	1
2	1100	2
3	2300	3
5	4500	4
4	9500	5

(5 rows)

ТРИГГЕРНЫЕ ФУНКЦИИ

ТРИГГЕРНЫЕ ФУНКЦИИ, ВЫЗЫВАЮТСЯ ПРИ ИЗМЕНЕНИЯХ ДАННЫХ ИЛИ СОБЫТИЯХ В БАЗЕ ДАННЫХ.

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). Для триггеров автоматически определяются специальные локальные переменные с именами вида `TG_имя`, описывающие условие, повлёкшее вызов триггера.

NEW

Тип данных `RECORD`. Переменная содержит **новую строку базы данных** для команд **INSERT/UPDATE** в триггерах уровня строки. В триггерах уровня оператора и для команды **DELETE** эта переменная **имеет значение null**.

OLD

Тип данных `RECORD`. Переменная **содержит старую строку базы данных** для команд **UPDATE/DELETE** в триггерах уровня строки. В триггерах уровня оператора и для команды **INSERT** эта переменная **имеет значение null**.

ТРИГГЕРНАЯ ФУНКЦИЯ ДОЛЖНА ВЕРНУТЬ ЛИБО NULL, ЛИБО ЗАПИСЬ/СТРОКУ, СООТВЕТСТВУЮЩУЮ СТРУКТУРЕ ТАБЛИЦЕ, ДЛЯ КОТОРОЙ СРАБОТАЛ ТРИГГЕР.

Если **BEFORE** триггер уровня строки возвращает **NULL**, то все дальнейшие действия с этой строкой прекращаются (не срабатывают последующие триггеры, команда **INSERT/UPDATE/DELETE** для этой строки не выполняется). Если **возвращается не NULL**, то **дальнейшая обработка продолжается именно с этой строкой**. Возвращение строки отличной от начальной **NEW**, изменяет строку, которая будет вставлена или изменена. Традиционно для триггеров **DELETE** возвращается переменная **OLD**.

Возвращаемое значение для строчного триггера **AFTER** и триггеров уровня оператора (**BEFORE** или **AFTER**) всегда игнорируется. Это может быть и **NULL**. В этих триггерах по-прежнему можно прервать вызвавшую их команду, для этого нужно явно вызвать ошибку.

СУЩНОСТЬ «ПОЛЬЗОВАТЕЛИ»

```
[sbrmvch=# select * from users;
```

id	name	profession
1	Bob	QA
2	Camilo	Front End developer
3	Billy	Backend Developer
4	Alice	Mobile Developer
5	Kate	QA
6	Wayne	DevOps
7	Tim	Mobile Developer
8	Amigos	QA
9	Sima	QA
10	Kim	QA
11	Kate	QA

(11 rows)

СУЩНОСТЬ «АУДИТ»

```
CREATE TABLE Audit
```

```
(  
  UsersId INT NOT NULL,  
  Name VARCHAR(20) NOT NULL,  
  AuditUserName VARCHAR(20) NOT NULL,  
  UserAdditionTime VARCHAR(20) NOT NULL  
);
```

ТРИГГЕРНАЯ ФУНКЦИЯ

```
CREATE OR REPLACE FUNCTION user_insert_trigger_fnc()  
RETURNS trigger AS  
$$  
BEGIN  
INSERT INTO Audit (UsersId, Name, AuditUserName, UserAdditionTime)  
VALUES (NEW.id,NEW.name,current_user,current_date);  
RETURN NEW;  
END;  
$$  
LANGUAGE 'plpgsql';
```

ТРИГГЕР

```
CREATE TRIGGER user_insert_trigger
AFTER INSERT
ON users
FOR EACH ROW
EXECUTE PROCEDURE user_insert_trigger_fnc();
```

Как только выполнится описанный INSERT в «Users», триггер добавит одну новую запись в «Audit» со следующими данными:

```
insert into users values (12, 'Simona', 'Mobile Dev');
```

```
sbrmvch=# select * from audit;
 usersid |  name  | auditusername | useradditiontime
-----+-----+-----+-----
      12 | Simona | sbrmvch       | 2023-09-21
```