Модуль 2. Управление данными в базах данных

Лекция 8. Хранимые процедуры и триггеры.
(Процедурное обеспечение целостности данных).

Преподаватель:
Семенов Геннадий Николаевич, к.т.н., доцент

2023

Основные объекты БД

1. Таблицы;
2. Индексы;
3. Представления;
4. **Хранимые Процедуры И Функции;**
5. **Триггеры;**
6. Курсоры;
7. Системный Словарь, Содержащий Метаданные.

Хранимая процедура

- В приложениях баз данных часто возникает необходимость выполнения одних и тех же или сходных подзадач, включающих в себя одинаковое манипулирование(управления) данными, хранимыми в базе данных.
- Как и в процедурных языках, в языке SQL возникла концепция процедур, отдельно описанных, повторно используемых последовательностей операций, выполняющих, как правило, определенную прикладную функцию.
- В современных приложения баз данных эта концепция развилась в концепцию ***хранимых процедур***.
- **Хранимая процедура** представляет собой процедуру, находящуюся и выполняющуюся на сервере базы данных и содержащую операторы SQL и операторы процедурной логики (операторы процедурного SQL или другого языка программирования).
- В современных информационных технологиях перенос как можно большей части логики предметной области в хранимые процедуры базы данных является стратегической концепцией.

Назначение и преимущества хранимых процедур

- Хранимые процедуры позволяют повысить производительность, расширяют возможности программирования и поддерживают функции безопасности данных.
- Вместо хранения часто используемого запроса, клиенты могут ссылаться на соответствующую хранимую процедуру. При вызове хранимой процедуры её содержимое сразу же обрабатывается сервером.
- Кроме собственно выполнения запроса, хранимые процедуры позволяют также производить **вычисления** и манипуляцию данными — **изменение**, **удаление**, **выполнять** DDL-операторы (не во всех СУБД!) и **вызывать** другие хранимые процедуры, выполнять сложную транзакционную логику.
- Один-единственный оператор позволяет вызвать сложный сценарий, который содержится в хранимой процедуре, что позволяет избежать пересылки через сеть сотен команд и, в особенности, необходимости передачи больших объёмов данных с клиента на сервер.

Хранимые процедуры и функции

- Хранимые процедуры пишутся на специальном встроенном языке программирования, они могут включать любые операторы SQL, а также включают некоторый набор операторов, управляющих ходом выполнения программ

- Хранимые процедуры являются объектами БД.
- Каждая хранимая процедура компилируется при первом выполнении
- В процессе компиляции строится оптимальный план выполнения процедуры.
- Описание процедуры совместно с планом ее выполнения хранится в системных таблицах БД.
- В процедурах можно выполнять практически любые действия - обновление, удаление, выборку и т.д. Чаще всего используется выборка данных и их последующая обработка.

Синтаксис PostgreSQL

Функции, написанные на PL/pgSQL, определяются на сервере командами CREATE FUNCTION. Такая команда обычно выглядит, например, так:

```
CREATE FUNCTION somefunc(integer, text)
 RETURNS integer AS 'тело функции'
 LANGUAGE plpgsql;
```

PL/pgSQL это блочно-структурированный язык. Текст тела функции должен быть *блоком*. Структура блока:

```
[ <<метка>> ]
[ DECLARE
    объявления ]
BEGIN
    операторы
END [ метка ];
```

Каждое объявление и каждый оператор в блоке должны завершаться символом «;» (точка с запятой). Блок, вложенный в другой блок, должен иметь точку с запятой после END, как показано выше. Однако финальный END, завершающий тело функции, не требует точки с запятой.

Синтаксис PostgreSQL

Все переменные, используемые в блоке, должны быть определены в секции объявления.

Переменные PL/pgSQL могут иметь любой тип данных SQL, такой как integer, varchar, char.

Примеры объявления переменных:

user_id integer;

quantity numeric(5);

url varchar;

Общий синтаксис объявления переменной:

имя [CONSTANT] ***тип*** [COLLATE ***имя_правила_сортировки***] [NOT NULL] [{ DEFAULT | := | = } ***выражение***];

Предложение DEFAULT, если присутствует, задаёт начальное значение, которое присваивается переменной при входе в блок. Если отсутствует, то переменная инициализируется SQL-значением NULL.

Указание CONSTANT предотвращает изменение значения переменной после инициализации, таким образом, значение остаётся постоянным в течение всего блока.

Параметр COLLATE определяет правило сортировки, которое будет использоваться для этой переменной (см. документацию)

Синтаксис PostgreSQL

Значение по умолчанию вычисляется и присваивается переменной каждый раз при входе в блок (не только при первом вызове функции).

Примеры:

```
quantity integer DEFAULT 32;
url varchar := 'http://mysite.com';
user_id CONSTANT integer := 10;
```

Объявление параметров функции

Переданные в функцию параметры именуются идентификаторами $1, $2 и т. д. Дополнительно, для улучшения читаемости, можно объявить псевдонимы для параметров $***n***. Либо псевдоним, либо цифровой идентификатор используются для обозначения параметра.

Создать псевдоним можно двумя способами.

Предпочтительный способ это дать имя параметру в команде CREATE FUNCTION, например:

```
CREATE FUNCTION sales_tax(subtotal real) RETURNS real AS $$
BEGIN
        RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

Синтаксис PostgreSQL

Другой способ это явное объявление псевдонима при помощи синтаксиса:

имя ALIAS FOR $***n***;

Предыдущий пример для этого стиля выглядит так:

```
CREATE FUNCTION sales_tax(real) RETURNS real AS $$
DECLARE
        subtotal ALIAS FOR $1;
BEGIN
        RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

Синтаксис PostgreSQL

Когда функция на PL/pgSQL объявляется с выходными параметрами, им выдаются цифровые идентификаторы $n и для них можно создавать псевдонимы точно таким же способом, как и для обычных входных параметров.

Выходной параметр это фактически переменная, стартующая с NULL и которой присваивается значение во время выполнения функции. Возвращается последнее присвоенное значение.

Например, функция sales_tax может быть переписана так:

```
CREATE FUNCTION sales_tax(subtotal real, OUT tax real) AS $$
BEGIN
        tax := subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

Синтаксис PostgreSQL

Выходные параметры наиболее полезны для возвращения нескольких значений.

Простейший пример:

```
CREATE FUNCTION sum_n_product(x int, y int, OUT sum int, OUT prod
    int) AS $$
BEGIN
    sum := x + y;
    prod := x * y;
END;
$$ LANGUAGE plpgsql;
```

Хранимые процедуры PostgreSQL

Выходные параметры поддерживаются и в процедурах,

например:

```
CREATE PROCEDURE sum_n_product(x int, y int, OUT sum int, OUT prod
    int) AS $$
BEGIN
    sum := x + y;
    prod := x * y;
 END; $$
LANGUAGE plpgsql;
```

При вызове процедуры нужно указать все параметры. Вызывая процедуру на чистом SQL, вместо выходных параметров можно указать NULL:

```
CALL sum_n_product(2, 4, NULL, NULL);
```

Операторы в процедурах

Условный оператор:

```
begin
        IF(условие) THEN
                запрос 1;
        ELSEIF(условие)
                запрос 2;
        ELSE
                запрос 3;
        END IF;
end //
```

Оператор цикла WHILE:

```
WHILE условие DO
  запрос
END WHILE
```

Оператор цикла **REPEAT**:
Условие цикла проверяется не в начале, как в цикле WHILE, а в конце, т.е. хотя бы один раз, но цикл выполняется. Сам же цикл выполняется, пока условие ложно.

```
REPEAT
        запрос
UNTIL  условие
END REPEAT
```

Оператор цикла LOOP

Этот цикл вообще не имеет условий, поэтому обязательно должен иметь оператор LEAVE:

```
LOOP
        запрос
END LOOP
```

Функции и хранимые процедуры

- SQL
- PL/pgSQL
- PL/Perl
- PL/Tcl
- PL/Python
- PL/R
- PL/Java
- plPHP
- plRuby
- всегда можно добавить что-то своё

Пример функции

SQL:

```
CREATE FUNCTION add(integer, integer) RETURNS integer
AS 'select $1 + $2;‘
LANGUAGE SQL IMMUTABLE RETURNS NULL ON NULL INPUT;
```

pl/pgSQL:

```
CREATE OR REPLACE FUNCTION increment(i integer) RETURNS
    integer
AS  $$
    BEGIN RETURN i + 1;
    END;
$$ LANGUAGE plpgsql;
```

Оператор CASE

Оператор **CASE** может быть использован в одной из двух синтаксических форм записи:

1-я форма:

```
CASE <проверяемое выражение>
    WHEN <сравниваемое выражение 1> THEN <возвращаемое значение 1>
    …
    WHEN <сравниваемое выражение N> THEN <возвращаемое значение N>
    [ELSE <возвращаемое значение>]
END
```

2-я форма:

```
CASE
        WHEN <предикат 1> THEN <возвращаемое значение 1>
        …
        WHEN <предикат N> THEN <возвращаемое значение N>
        [ELSE <возвращаемое значение>]
END
```

Примеры использования оператора CASE

1) Посчитать количество студентов дневной и вечерней формы обучения:

```
CREATE VIEW students_number (DEPARTMENT, YEAR,
DAY_FORM, EVENING_FORM) as
SELECT gr.department, gr.year,
    count(case when gr.study='ДНЕВНАЯ'   then 1 else null end)
form1,
    count(case when gr.study='ВЕЧЕРНЯЯ' then 1 else null end)
form2
  FROM groups gr,   students st
    WHERE gr.group_code = st.group_code
    GROUP BY gr.department, gr.year, gr.study
    ORDER BY gr.department, gr.year asc;
```

Примеры использования оператора CASE

2) Вывести все имеющиеся модели ПК с указанием цены. Отметить самые дорогие и самые дешевые модели.

```
SELECT DISTINCT model, price,
  CASE price
    WHEN (SELECT MAX(price) FROM PC)
      THEN 'Самый дорогой'
    WHEN (SELECT MIN(price) FROM PC)
      THEN 'Самый дешевый'
    ELSE 'Средняя цена'
  END comment
FROM PC
 ORDER BY price;
```

model	price	comment
1232	350.0	Самый дешевый
1260	350.0	Самый дешевый
1232	400.0	Средняя цена
1232	600.0	Средняя цена
1233	600.0	Средняя цена
1121	850.0	Средняя цена
1233	950.0	Средняя цена
1233	980.0	Самый дорогой

Триггеры

- Фактически триггер — это специальный вид хранимой процедуры, которую SQL Server вызывает при выполнении операций модификации соответствующих таблиц.
- Триггер автоматически активизируется при выполнении операции, с которой он связан.
- Триггеры связываются с одной или несколькими операциями модификации над одной таблицей.

Цели использования триггеров

- проверка корректности введенных данных и выполнение сложных ограничений целостности данных, которые трудно, если вообще возможно, поддерживать с помощью ограничений целостности, установленных для таблицы;
- выдача предупреждений, напоминающих о необходимости выполнения некоторых действий при обновлении таблицы, реализованных определенным образом;
- накопление аудиторской информации посредством фиксации сведений о внесенных изменениях и тех лицах, которые их выполнили;
- поддержка репликации.

При условии правильного использования триггеры могут стать очень мощным механизмом. Основное их преимущество заключается в том, что стандартные функции сохраняются внутри базы данных и согласованно активизируются при каждом ее обновлении. Это может существенно упростить приложения.

Недостатки триггеров:

- **сложность**: при перемещении некоторых функций в базу данных усложняются задачи ее проектирования, реализации и администрирования;
- **скрытая функциональность**: перенос части функций в базу данных и сохранение их в виде одного или нескольких триггеров иногда приводит к сокрытию от пользователя некоторых функциональных возможностей.

  Хотя это упрощает работу пользователя, но может стать причиной незапланированных, потенциально нежелательных и вредных побочных эффектов, поскольку в этом случае пользователь не в состоянии контролировать все процессы, происходящие в базе данных;
- **влияние на производительность**: перед выполнением каждой команды по изменению состояния базы данных СУБД должна проверить триггерное условие с целью выяснения необходимости запуска триггера для этой команды.

В моменты пиковой нагрузки ее снижение производительности может стать особенно заметным. при возрастании количества триггеров увеличиваются и накладные расходы, связанные с такими операциями.

Триггеры

Триггеры могут быть назначены для операторов:

1. INSERT
2. UPDATE
3. DELETE
4. TRUNCATE

CREATE TRIGGER *name* { BEFORE | AFTER } { *event* [OR ...] }
    ON *table* [FOR [EACH] { ROW | STATEMENT }]
EXECUTE PROCEDURE *funcname* (*arguments*)

Создание триггера

CREATE TRIGGER имя_триггера
BEFORE | **AFTER** <триггерное_событие>
ON <имя_таблицы>
[REFERENCING
<список_старых_или_новых_псевдонимов>]
[FOR EACH { ROW | STATEMENT}]
[WHEN(условие_триггера)]
<**тело_триггера**>

Время запуска триггера определяется с помощью ключевых слов **BEFORE** (триггер запускается до выполнения связанных с ним событий) или **AFTER** (после их выполнения).

Выполняемые триггером действия задаются для каждой строки (**FOR EACH ROW**), охваченной данным событием, или только один раз для каждого события (**FOR EACH STATEMENT**).

Обозначение <**список_старых_или_новых_псевдонимов**> относится к таким компонентам, как старая или новая строка (OLD / NEW) либо старая или новая таблица (OLD TABLE / NEW TABLE). Это значит: старые значения не применимы для событий вставки, а новые – для событий удаления.

Триггеры в MySQL

CREATE [DEFINER = { user | CURRENT_USER }]
TRIGGER trigger_name trigger_time trigger_event
ON tbl_name FOR EACH ROW trigger_stmt

Триггер связывается с таблицей с именем **tbl_name**, которая должна быть постоянной таблицей. Триггер не может быть создан для временной таблицы или представления.

Выражение **DEFINER** определяет права, которые получает триггер при активизации

trigger_time - время срабатывания триггера. Возможны значения **BEFORE** или **AFTER**, которые определяют время выполнения триггера (до или после события вызвавшего триггер).

trigger_event – определяет какое событие активизирует триггер:

INSERT: активизируется всякий раз когда в таблицу добавляется новая строка. Например, при выполнении операторов: **INSERT**, **LOAD DATA**, и **REPLACE**.

INSERT: активизируется всякий раз когда в таблицу добавляется новая строка. Например, при выполнении операторов: **INSERT**, **LOAD DATA**, и **REPLACE**.

Триггеры в PostgreSQL

Триггеры также классифицируются в соответствии с тем, срабатывают ли они до, после или вместо операции.

Они называются триггерами **BEFORE**, **AFTER** и **INSTEAD OF**, соответственно.

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.

Триггеры в PostgreSQL

- Если триггер **AFTER** — это *триггер ограничения*, его выполнение может быть отложено не до конца работы оператора, а до конца транзакции. В любом случае триггер выполняется в рамках той же транзакции, к которой относится вызвавший его оператор, поэтому если или оператор, или триггер вызывает ошибку, оба действия отменяются.
- Оператор, нацеленный на родительскую таблицу в иерархии наследования или секционирования, не вызывает срабатывания триггеров уровня оператора для задействованных дочерних таблиц; срабатывать будут только такие триггеры для родительской таблицы.
- Однако если для этих дочерних таблиц установлены триггеры уровня строк, они будут срабатывать.

Триггеры в PostgreSQL

CREATE [OR REPLACE] [CONSTRAINT] TRIGGER ***имя*** { BEFORE | AFTER | INSTEAD OF } { ***событие*** [OR ...] }
ON ***имя_таблицы***
[FROM ***ссылающаяся_таблица***]
[NOT DEFERRABLE | [DEFERRABLE] [INITIALLY IMMEDIATE | INITIALLY DEFERRED]]
[REFERENCING { { OLD | NEW } TABLE [AS] ***имя_переходного_отношения*** } [...]]
[FOR [EACH] { ROW | STATEMENT }]
[WHEN (***условие***)]
EXECUTE { FUNCTION | PROCEDURE } ***имя_функции*** (***аргументы***)

допускается ***событие***:

INSERT
UPDATE [OF ***имя_столбца*** [, ...]]
DELETE
TRUNCATE

События:

UPDATE: активизируется всякий раз когда какая либо строка таблицы изменяется. Например, при выполнении оператора **UPDATE**.

DELETE: активизируется всякий раз когда из таблицы удаляется строка. Например, при выполнении операторов **DELETE** и **REPLACE**. Однако операторы **DROP TABLE** и **TRUNCATE** не активизируют триггер удаления, потому, что эти операторы не используют удаление строк.

Важно понимать, что события **trigger_event** не являются буквальными ссылками на операторы SQL, которые активизируют триггер. Например, **INSERT** триггер (триггер добавления) срабатывает не только по оператору **INSERT**, но и по оператору **LOAD DATA**, поскольку оба эти оператора используют вставку строк в таблицу.

trigger_stmt – это оператор исполняемый при активизации триггера. Для использования нескольких оператором необходимо использовать конструкцию **BEGIN** ... **END**

Пример использования оператора

INSERT INTO ... ON DUPLICATE KEY UPDATE ...: BEFORE INSERT

- Триггер будет срабатывать при каждом добавлении строки, а триггеры AFTER INSERT или BEFORE UPDATE и AFTER UPDATE будут активизироваться в зависимости от возникновения ситуации дублирования ключа (DUPLICATE KEY).
- При дублировании ключа триггер AFTER INSERT не активизируется, но активизируются триггеры BEFORE UPDATE и AFTER UPDATE

Для таблицы не может быть создано два триггера, у которых совпадают и время (trigger_time) и событие (trigger_event).

- Например, нельзя создать два BEFORE UPDATE триггера для одной и той же таблицы, но можно создать BEFORE UPDATE и BEFORE INSERT триггеры, или BEFORE UPDATE и AFTER UPDATE триггеры.

Следующие пример демонстрирует создание таблицы и INSERT триггера (триггера добавления). Триггер суммирует значения одного из добавляемых в таблицу столбцов.

- CREATE TABLE account (acct_num INT, amount DECIMAL(10,2))
- CREATE TRIGGER ins_sum BEFORE INSERT ON account FOR EACH ROW SET @sum = @sum + NEW.amount;

Ограничения

- Нельзя использовать в теле триггера операции создания объектов БД (новой БД, новой таблицы, нового индекса, новой хранимой процедуры, нового триггера, новых индексов, новых представлений),
- Нельзя использовать в триггере команду удаления объектов **DROP** для всех типов базовых объектов БД.
- Нельзя использовать в теле триггера команды изменения базовых объектов **ALTER TABLE**, **ALTER DATABASE**.
- Нельзя изменять права доступа к объектам БД, то есть выполнять команду **GRAND** или **REVOKE**.
- Нельзя создать триггер для представления (**VIEW**).
- В отличие от хранимых процедур, триггер не может возвращать никаких значений, он запускается автоматически сервером и не может связаться самостоятельно ни с одним клиентом.