Мадока ВТ Вики

БД:Теория:Глава 5

В этой главе мы углубимся в команды для изменения данных (DML), рассмотрим мощный инструмент SQL — представления (Views), научимся создавать собственные функции и процедуры с использованием процедурного расширения PL/pgSQL, разберемся с триггерами для реализации сложной логики и автоматизации, а также подробно рассмотрим механизм транзакций для обеспечения целостности данных. Эта глава основана на материалах Лекции 5: “PL/pgSQL” и частично затрагивает DML из Лекции 4/5.

Часть 1: Манипулирование Данными (DML)

DML (Data Manipulation Language) — это часть SQL, отвечающая за добавление, изменение и удаление данных *внутри* таблиц.

1. **`INSERT`**: Добавляет одну или несколько новых строк в таблицу.

   - **Синтаксис 1 (Указание столбцов):**

     ```
     INSERT INTO имя_таблицы (столбец1, столбец2, ...)
     VALUES (значение1, значение2, ...);
     ```

     Порядок значений должен соответствовать порядку указанных столбцов.

   - **Синтаксис 2 (Без указания столбцов):**

     ```
     INSERT INTO имя_таблицы VALUES (значение1, значение2, ...);
     ```

     Значения должны быть перечислены в том же порядке, в каком столбцы определены в таблице. Менее надежный способ, т.к. зависит от порядка столбцов.

   - **Синтаксис 3 (Вставка из `SELECT`):**

     ```
     INSERT INTO имя_таблицы (столбец1, столбец2, ...)
     SELECT выражение1, выражение2, ...
     FROM другая_таблица
     WHERE условие;
     ```

     Позволяет вставить в таблицу результат другого запроса. Количество и типы столбцов в `INSERT` и `SELECT` должны совпадать.

Пример (из слайдов):

```sql
-- Пример 1 и 2: Добавление студента
INSERT INTO student VALUES (123, 'Vasya', 345);
INSERT INTO student (StudID, GroupID, Name) VALUES (123, 345, 'Vasya');

-- Пример 3: Добавление студентов из временной таблицы по условию
INSERT INTO student (StudID, Name, GroupID) -- Указываем порядок для INSERT
SELECT ID, Name, 345 -- Указываем порядок в SELECT
FROM tmp_person
WHERE Exam > 70;
```

2. **`UPDATE`**: Изменяет существующие строки в таблице.

- **Синтаксис:**

```sql
UPDATE имя_таблицы
SET столбец1 = новое_значение1,
    столбец2 = новое_значение2,
    ...
[ WHERE условие ]; -- Если WHERE отсутствует, обновятся ВСЕ строки!
```

Условие `WHERE` определяет, какие строки нужно обновить.

Пример (из слайдов):

```sql
-- Установить всем студентам GroupID = 578
UPDATE student SET GroupID = 578;

-- Установить GroupID = 34 для студента с именем 'Ivan'
UPDATE student SET GroupID = 34 WHERE Name = 'Ivan';
```

3. **`DELETE`**: Удаляет существующие строки из таблицы.

- **Синтаксис:**

```sql
DELETE FROM имя_таблицы
[ WHERE условие ]; -- Если WHERE отсутствует, удалятся ВСЕ строки!
```

Условие `WHERE` определяет, какие строки нужно удалить.

Пример (из слайдов):

```sql
-- Удалить студентов с GroupID = 34
DELETE FROM student WHERE GroupID = 34;

-- Удалить ВСЕХ студентов
DELETE FROM student;
```

4. **`TRUNCATE`**: Быстро удаляет **все** строки из таблицы (или нескольких таблиц).

- **Синтаксис:**

```sql
TRUNCATE [TABLE] имя_таблицы1 [, имя_таблицы2 ...];
```

- **Отличия от `DELETE` без `WHERE`:**
  - Обычно **намного быстрее** для больших таблиц, так как не сканирует таблицу построчно и не записывает удаление каждой строки в лог транзакций так же подробно.
  - Не запускает триггеры `DELETE` на уровне строк (`FOR EACH ROW`).
  - Сбрасывает счетчики последовательностей (`SERIAL` и т.п.), связанных с таблицей (в PostgreSQL, если указано `RESTART IDENTITY`).
  - Не возвращает количество удаленных строк.
- Используйте `TRUNCATE`, когда нужно быстро и полностью очистить таблицу.

Пример (из слайдов):

```
-- Быстро очистить таблицу student
TRUNCATE TABLE student;

-- Быстро очистить несколько таблиц
TRUNCATE TABLE student, groups, exams;
```

Часть 2: Выполнение Скриптов

Часто SQL-код (особенно DDL для создания схемы или DML для наполнения данными) хранится в файлах (`.sql`). Такие файлы называются **скриптами**. Выполнить скрипт в PostgreSQL можно несколькими способами:

1. **Через утилиту `psql` при запуске:**

```
psql -U имя_пользователя -d имя_базы -f путь/к/файлу/script.sql
```

   - `-U`: Имя пользователя PostgreSQL.
   - `-d`: Имя базы данных, к которой подключаемся.
   - `-f`: Путь к файлу скрипта.

2. **Внутри интерактивной сессии `psql`:** Используются мета-команды (начинаются с `\`):

```
-- Подключаемся к базе
psql -U имя_пользователя -d имя_базы

-- Выполняем скрипт из файла
\i путь/к/файлу/script.sql
-- или эквивалентная команда
\include путь/к/файлу/script.sql
```

Это удобно для выполнения небольших скриптов или частей кода во время работы с БД.

Часть 3: Представления (Views)

Мы уже кратко упоминали **виртуальные таблицы**. Основной их вид — это **представления**.

- **Представление (View)** — это сохраненный в базе данных **именованный SQL-запрос** (`SELECT`).
- С представлением можно работать почти так же, как с обычной таблицей (выполнять `SELECT`, иногда `INSERT`, `UPDATE`, `DELETE`, если представление обновляемое).
- Представление **не хранит данные** само по себе (за исключением материализованных представлений). Каждый раз при обращении к представлению СУБД выполняет лежащий в его основе `SELECT`-запрос к базовым таблицам.

Зачем нужны представления?

1. **Упрощение сложных запросов:** Длинный и сложный запрос можно “спрятать” за простым именем представления.
2. **Сокрытие структуры данных:** Можно предоставить пользователям доступ только к представлению, которое показывает лишь необходимые им столбцы и строки, скрывая сложность или конфиденциальность базовых таблиц.
3. **Логическая независимость данных:** Если структура базовых таблиц меняется, можно изменить только определение представления, а запросы пользователей, обращающиеся к представлению, останутся прежними.
4. **Обеспечение обратной совместимости:** При рефакторинге схемы можно создать представления, имитирующие старую структуру.

Создание Представления (`CREATE VIEW`)

```
CREATE [ OR REPLACE ] VIEW имя_представления [ (имя_колонки1, имя_колонки2, ...) ]
AS
SELECT_запрос; -- Запрос, определяющий представление
```

- `OR REPLACE`: Если представление с таким именем уже существует, оно будет заменено.
- `[ (имя_колонки1, ...) ]`: Необязательный список имен для столбцов представления. Если опущен, используются имена столбцов из `SELECT_запрос`.

Пример (из слайдов): Создать представление, показывающее студентов факультета ПИКТ (предполагаем, что это группы, начинающиеся с ‘Р3’).

```
-- Создаем представление
CREATE VIEW PICTStudents AS
SELECT * -- Выбираем все колонки из student
FROM student
WHERE GroupID IN ( -- Выбираем студентов, чья группа в списке групп ПИКТ
    SELECT GroupID
    FROM groups
    WHERE GroupName LIKE 'P3%' -- Находим группы, чье имя начинается с 'P3'
);

-- Используем представление как обычную таблицу
SELECT st_name, GroupID FROM PICTStudents WHERE st_id > 100;
```

Пример (с переименованием колонок):

```
CREATE VIEW PICTStudents2 (PICTId, pSurname) AS -- Задаем свои имена колонок
SELECT StudentID, Surname -- Выбираем нужные колонки из student
FROM student
WHERE GroupID IN (
    SELECT GroupID FROM groups WHERE GroupName LIKE 'P3%'
);
```

```sql
-- Используем представление с новыми именами
SELECT PICTId FROM PICTStudents2 WHERE pSurname LIKE 'Иван%';
```

Материализованные Представления (`MATERIALIZED VIEW`)

- В отличие от обычных представлений, материализованное представление **хранит результат** своего запроса физически в базе данных (как кэш).
- Обращение к материализованному представлению происходит очень быстро, так как не требует выполнения основного запроса каждый раз.
- **Проблема:** Данные в материализованном представлении могут **устареть**, если данные в базовых таблицах изменились.
- **Обновление:** Данные нужно обновлять **вручную** (или по расписанию) с помощью команды `REFRESH MATERIALIZED VIEW`.

Создание и Обновление:

```sql
-- Создаем материализованное представление
CREATE MATERIALIZED VIEW PICTStudents_MV (PICTId, pSurname) AS -- Добавляем MATERIALIZED
SELECT StudentID, Surname
FROM student
WHERE GroupID IN (
    SELECT GroupID FROM groups WHERE GroupName LIKE 'P3%'
);

-- Используем (быстро, читает сохраненные данные)
SELECT * FROM PICTStudents_MV;

-- Обновляем данные (выполняет основной SELECT и перезаписывает сохраненные данные)
REFRESH MATERIALIZED VIEW PICTStudents_MV;
```

Материализованные представления полезны для сложных, ресурсоемких запросов, результаты которых нужны часто, а актуальность данных с точностью до секунды не критична (например, для отчетов).

Часть 4: Введение в PL/pgSQL

SQL — декларативный язык. Иногда нам нужно выполнить последовательность действий, использовать циклы, условия, переменные — то есть написать *процедурный* код. Для этого в PostgreSQL (и других СУБД) существуют процедурные расширения языка.

- **PL/pgSQL (Procedural Language / PostgreSQL SQL)** — стандартный, блочно-структурированный процедурный язык для PostgreSQL. Его синтаксис во многом основан на языке Ada.

Зачем нужен PL/pgSQL?

- Создание **пользовательских функций (UDF)** и **хранимых процедур**.
- Реализация сложной бизнес-логики непосредственно в базе данных.
- Создание **триггеров** для автоматизации действий при изменении данных.
- Повышение производительности за счет уменьшения обмена данными между приложением и СУБД (логика выполняется на сервере БД).

Типы Пользовательских Функций:

В PostgreSQL можно создавать функции на разных языках:

1. **SQL-функции:** Тело функции состоит из одного или нескольких SQL-запросов. Выполняются быстро, но возможности ограничены самим SQL.
2. **PL/pgSQL-функции:** Тело функции написано на языке PL/pgSQL, позволяет использовать переменные, циклы, условия и т.д. Самый распространенный вариант для сложной логики.
3. **Функции на других языках (C, Python, Perl, Tcl и др.):** Требуют установки соответствующих расширений (`CREATE EXTENSION plpython3u;`). Позволяют использовать возможности и библиотеки этих языков внутри БД.

Часть 5: Пользовательские Функции и Процедуры

Создание Функции (`CREATE FUNCTION`)

```
CREATE [ OR REPLACE ] FUNCTION имя_функции ( [ [имя_арг1] тип_арг1, [имя_арг2] тип_арг2, ...] )
RETURNS тип_возвращаемого_значения -- Или RETURNS TABLE(...) для возврата таблицы, или VOID если ничего
не возвращает
AS $$ -- Или AS '...' - тело функции в $$ или одинарных кавычках
    -- Тело функции (SQL или PL/pgSQL код)
$$ LANGUAGE язык; -- язык: sql, plpgsql, plpython3u и т.д.
```

- `OR REPLACE`: Заменяет существующую функцию с тем же именем и типами аргументов.
- **Аргументы:**
  - В старых версиях и в SQL-функциях часто используются позиционные параметры (`$1`, `$2`, ...).
  - В PL/pgSQL и современных SQL-функциях можно (и рекомендуется) использовать именованные аргументы.
- `RETURNS`: Указывает тип возвращаемого значения. `VOID` означает, что функция ничего не возвращает (похоже на процедуру, но это все еще функция).
- `AS $$ ... $$`: Тело функции. Использование `$$` (долларовое квотирование) предпочтительнее одинарных кавычек (`'...'`), так как позволяет легко использовать одинарные кавычки внутри тела функции без экранирования (`).
- `LANGUAGE`: Язык, на котором написано тело функции.

Пример SQL-функции (позиционные параметры):

```
-- Функция для изменения группы студента
CREATE FUNCTION updateStudentGroup_v1(int, int) -- Аргументы: stud_id, group_id
RETURNS void -- Ничего не возвращает
AS '
    UPDATE student
    SET GroupID = $2 -- $2 - второй аргумент (group_id)
    WHERE StudID = $1; -- $1 - первый аргумент (stud_id)
' LANGUAGE sql;

-- Вызов функции
SELECT updateStudentGroup_v1(101, 2); -- Обновить группу для студента 101 на группу 2
```

Пример SQL-функции (именованные параметры, PostgreSQL 9.2+):

```
CREATE FUNCTION updateStudentGroup_v2(
    p_stud_id INT, -- Имя аргумента p_stud_id, тип INT
    p_group_id INT -- Имя аргумента p_group_id, тип INT
)
RETURNS void
AS $$
    UPDATE student
    SET GroupID = p_group_id -- Используем имя параметра
    WHERE StudID = p_stud_id; -- Используем имя параметра
```

```
$$ LANGUAGE sql;

-- Вызов функции (можно по именам или по позиции)
SELECT updateStudentGroup_v2(p_stud_id := 102, p_group_id := 1);
SELECT updateStudentGroup_v2(103, 1);
```

Процедуры (`CREATE PROCEDURE`, PostgreSQL 11+)

Процедуры похожи на функции, возвращающие `VOID`, но имеют ключевое отличие: внутри процедур **можно управлять транзакциями** (`COMMIT`, `ROLLBACK`), а внутри функций — нельзя.

```
CREATE [ OR REPLACE ] PROCEDURE имя_процедуры ( [ [имя_арг1] тип_арг1, ...] )
AS $$
    -- Тело процедуры (обычно PL/pgSQL)
$$ LANGUAGE язык;

-- Вызов процедуры
CALL имя_процедуры(значение1, ...);
```

Пример:

```
CREATE PROCEDURE transfer_money(sender_acc INT, receiver_acc INT, amount NUMERIC)
AS $$
BEGIN
    UPDATE accounts SET balance = balance - amount WHERE acc_id = sender_acc;
    UPDATE accounts SET balance = balance + amount WHERE acc_id = receiver_acc;
    -- Здесь можно было бы добавить COMMIT или ROLLBACK, если бы это не было частью бОльшей транзакции
END;
$$ LANGUAGE plpgsql;

-- Вызов
CALL transfer_money(1, 2, 100.00);
```

Часть 6: Основы PL/pgSQL

PL/pgSQL — **блочно-структурированный** язык. Основной элемент — блок кода.

Структура блока:

```
[ <<метка_блока>> ] -- Необязательная метка
[ DECLARE
    -- Объявление переменных
    имя_переменной тип_данных [ := начальное_значение ];
    ...
]
BEGIN
    -- Исполняемые операторы (SQL-запросы, присваивания, циклы, условия и т.д.)
    ...
    [ RETURN значение; ] -- Только для функций, возвращающих значение
END [ метка_блока ]; -- Метка конца блока (необязательно)
```

- Секция `DECLARE` необязательна, если переменные не нужны.
- `BEGIN`/`END` обязательны.
- Блоки могут быть вложенными.
- Метки используются для разрешения имен переменных во вложенных блоках (обращение к переменной внешнего блока: `метка_блока.имя_переменной`).
- Присваивание значения: `переменная := выражение;`.

- Выполнение SQL-запросов: Просто пишете SQL-запрос. Чтобы сохранить результат `SELECT` в переменную, используется `SELECT ... INTO переменная ...`.
- Вывод отладочной информации: `RAISE NOTICE 'Сообщение: %', переменная;` (`%` - место для подстановки значения переменной).

Пример блока (из слайда 33):

```
CREATE FUNCTION somefunc() RETURNS integer AS $$ -- SQL command
<< outerblock >> -- Метка внешнего блока (pl/pgSQL)
DECLARE
    quantity integer := 30; -- Переменная внешнего блока
BEGIN
    -- Создаем вложенный блок
    DECLARE
        quantity integer := 80; -- Переменная внутреннего блока (то же имя!)
    BEGIN
        RAISE NOTICE 'Inner quantity = %', quantity; -- Выведет 80 (переменная внутреннего блока)
        RAISE NOTICE 'Outer quantity = %', outerblock.quantity; -- Выведет 30 (обращение к переменной
внешнего блока по метке)
    END; -- Конец вложенного блока

    RAISE NOTICE 'Сейчас quantity = %', quantity; -- Выведет 30 (переменная внешнего блока, т.к. вышли из
вложенного)
    RETURN quantity; -- Вернет 30
END;
$$ LANGUAGE plpgsql;

-- Вызов функции
SELECT somefunc();
```

Вывод в `NOTICE` при вызове:

```
NOTICE:  Inner quantity = 80
NOTICE:  Outer quantity = 30
NOTICE:  Сейчас quantity = 30
```

Результат `SELECT`:

```
 somefunc
----------
       30
(1 row)
```

Анонимные блоки (`DO`)

Позволяют выполнить блок PL/pgSQL кода без создания функции или процедуры. Удобно для одноразовых задач или скриптов.

```
DO $$
[ DECLARE ...]
BEGIN
    -- PL/pgSQL код
END;
$$ LANGUAGE plpgsql; -- Язык можно опустить, если используется plpgsql по умолчанию
```

Пример (из слайда 34): Посчитать и вывести количество студентов.

```
DO $$
<<studentBlock>>
DECLARE
    studCount integer := 0;
BEGIN
    SELECT COUNT(*)
```

```
    INTO studCount -- Сохраняем результат COUNT(*) в переменную studCount
    FROM student;

    RAISE NOTICE 'Students: %', studCount; -- Выводим результат
END studentBlock $$;
```

Вывод в `NOTICE`: (Зависит от количества студентов в таблице)

```
NOTICE:  Students: 15
DO
```

Часть 7: Триггеры

Триггеры — это специальные процедуры, которые **автоматически** выполняются (срабатывают) в ответ на определенные события, происходящие с таблицей (обычно это операции DML: `INSERT`, `UPDATE`, `DELETE`).

Зачем нужны триггеры?

1. **Реализация сложных ограничений целостности:** Правила, которые сложно или невозможно выразить стандартными `CHECK`, `FOREIGN KEY` (например, проверка баланса перед списанием, сложные зависимости между таблицами).
2. **Аудит изменений:** Автоматическая запись информации о том, кто, когда и какие данные изменил, в отдельную таблицу логов.
3. **Автоматическое обновление связанных данных:** Например, пересчет итоговых сумм при изменении деталей заказа.
4. **Репликация данных** (в некоторых случаях).

Как работает триггер?

1. **Событие:** Происходит операция DML (`INSERT`, `UPDATE`, `DELETE`) или DDL (для событийных триггеров) с таблицей, для которой определен триггер.
2. **Срабатывание:** СУБД проверяет, есть ли триггеры, связанные с этим событием и таблицей.
3. **Выполнение:** Если триггер найден, выполняется связанная с ним **триггерная функция**.

Создание триггера в PostgreSQL:

Процесс состоит из двух шагов:

1. **Создание триггерной функции:** Это обычная функция PL/pgSQL (или на другом языке), но со специфическими особенностями:
   - Она **не принимает аргументов**.
   - Она должна возвращать специальный тип **`TRIGGER`** (для DML триггеров) или `EVENT_TRIGGER` (для DDL триггеров).
   - Внутри функции доступны специальные переменные (`NEW`, `OLD`, `TG_OP`, `TG_WHEN` и т.д.), содержащие информацию о событии и изменяемых данных.
   - Возвращаемое значение функции имеет значение (особенно для `BEFORE ROW` триггеров):
     - Возврат `NEW` (или измененной строки `NEW`): Операция продолжается с этой (возможно измененной) строкой.

- Возврат `OLD` (для `UPDATE`/`DELETE`): Операция продолжается со старой строкой (редко используется).
- Возврат `NULL`: **Операция для данной строки отменяется**, последующие триггеры для этой строки не срабатывают. Позволяет "запретить" изменение.
- Для `AFTER` триггеров возвращаемое значение игнорируется (операция уже произошла), но рекомендуется возвращать `NULL` или ту же запись (`NEW` или `OLD`).

2. **Создание самого триггера:** Связывает триггерную функцию с конкретной таблицей и событием.

Синтаксис `CREATE TRIGGER`:

```
CREATE TRIGGER имя_триггера
    { BEFORE | AFTER | INSTEAD OF } -- Когда срабатывать
    { event [ OR ... ] } -- На какое событие(я) (INSERT, UPDATE, DELETE, TRUNCATE)
    ON имя_таблицы
    [ FOR [ EACH ] { ROW | STATEMENT } ] -- Уровень срабатывания
    [ WHEN ( условие ) ] -- Дополнительное условие срабатывания
    EXECUTE PROCEDURE имя_триггерной_функции(); -- Какую функцию вызвать
```

- **`BEFORE | AFTER | INSTEAD OF`**:
  - `BEFORE`: Функция выполняется *перед* выполнением операции DML и перед проверкой ограничений. Позволяет изменить данные (`NEW`) или отменить операцию (вернув `NULL`).
  - `AFTER`: Функция выполняется *после* выполнения операции DML и проверки ограничений. Не может изменить данные (операция уже прошла) или отменить ее. Используется для аудита, обновления связанных данных.
  - `INSTEAD OF`: Специальный тип для **представлений (Views)**. Функция выполняется *вместо* операции DML над представлением, позволяя реализовать логику обновления базовых таблиц.
- **`event`**: `INSERT`, `UPDATE [ OF column1, ... ]`, `DELETE`, `TRUNCATE`. Можно указать несколько через `OR`. `UPDATE OF` срабатывает только при изменении указанных колонок.
- **`FOR EACH ROW | STATEMENT`**:
  - `ROW`: Функция выполняется **для каждой строки**, затронутой операцией DML. Внутри доступны переменные `NEW` (для `INSERT`/`UPDATE`) и `OLD` (для `UPDATE`/`DELETE`).
  - `STATEMENT` (По умолчанию): Функция выполняется **один раз на всю операцию DML**, независимо от количества затронутых строк. Переменные `NEW` и `OLD` недоступны.
- **`WHEN (условие)`**: Дополнительное условие (использующее значения `NEW`/`OLD`), которое проверяется перед вызовом функции (только для `ROW` триггеров). Если условие ложно, функция не вызывается.

Пример: Аудит таблицы `EMPLOYEE` (из слайдов)

Задача: При добавлении нового сотрудника в таблицу `EMPLOYEE` автоматически записывать ID сотрудника и время добавления в таблицу `AUDIT`.

1. Таблицы:

```
CREATE TABLE employee (
    id      INT PRIMARY KEY,
```

```
    name   TEXT NOT NULL,
    addr   CHAR(50),
    salary REAL
);

CREATE TABLE audit (
    emp_id     INT NOT NULL,
    entry_date TEXT NOT NULL -- Лучше использовать TIMESTAMPTZ
);
```

2. Триггерная функция:

```
CREATE OR REPLACE FUNCTION auditfunc()
RETURNS TRIGGER AS $$
BEGIN
    -- Вставляем запись в таблицу аудита
    -- NEW.id - это ID из строки, которая сейчас вставляется в employee
    -- current_timestamp - встроенная функция PostgreSQL, возвращает текущее время
    INSERT INTO audit(emp_id, entry_date) VALUES (NEW.id, current_timestamp);

    -- Возвращаем NEW, чтобы операция INSERT продолжилась успешно
    -- Для AFTER триггера возвращаемое значение игнорируется, но хорошей практикой
    -- является возврат соответствующей строки или NULL.
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

3. Создание триггера:

```
CREATE TRIGGER employee_audit_insert
    AFTER INSERT ON employee -- Срабатывать ПОСЛЕ вставки
    FOR EACH ROW -- Для каждой вставляемой строки
    EXECUTE PROCEDURE auditfunc(); -- Вызвать нашу функцию
```

Проверка:

```
INSERT INTO employee (id, name, salary) VALUES (1, 'Иван', 1000);
-- После этой команды в таблице audit появится запись: (1, <текущее время>)

INSERT INTO employee (id, name, salary) VALUES (2, 'Петр', 1500);
-- После этой команды в таблице audit появится вторая запись: (2, <текущее время>)
```

Событийные триггеры (Event Triggers):

- Срабатывают на DDL-команды (`CREATE TABLE`, `ALTER TABLE`, `DROP TABLE` и т.д.).
- Используются для аудита изменений схемы, запрета определенных DDL-операций и т.п.
- Триггерная функция должна возвращать `EVENT_TRIGGER`.
- Доступны специальные переменные (например, `tg_event`, `tg_tag`).

Пример (из слайдов): Логгирование DDL команд.

```
-- Функция
CREATE OR REPLACE FUNCTION eventtest()
RETURNS event_trigger AS $$
BEGIN
    RAISE NOTICE 'DDL Event: %, Command Tag: %', tg_event, tg_tag; -- tg_tag содержит текст команды
END;
$$ LANGUAGE plpgsql;

-- Триггер
CREATE EVENT TRIGGER eventtest_trigger
    ON ddl_command_start -- Срабатывать перед началом любой DDL команды
    EXECUTE PROCEDURE eventtest();
```

```
-- Проверка
CREATE TABLE some_table (id int);
```

Вывод в `NOTICE`:

```
NOTICE:  DDL Event: ddl_command_start, Command Tag: CREATE TABLE
CREATE TABLE
```

Удаление триггера:

```
DROP TRIGGER имя_триггера ON имя_таблицы;
```

Часть 8: Транзакции

Транзакция — это **логическая единица работы**, состоящая из одной или нескольких операций SQL, которая должна быть выполнена **атомарно**.

Свойства ACID:

Транзакции в реляционных СУБД обычно гарантируют свойства ACID:

1. **Atomicity (Атомарность):** Либо *все* операции внутри транзакции успешно выполняются и фиксируются, либо *ни одна* из них не оказывает влияния на базу данных (все изменения отменяются). “Все или ничего”.
2. **Consistency (Согласованность):** Транзакция переводит базу данных из одного *согласованного* (целостного) состояния в другое *согласованное* состояние. Во время выполнения транзакции целостность может временно нарушаться, но к моменту фиксации она должна быть восстановлена.
3. **Isolation (Изолированность):** Параллельно выполняющиеся транзакции не должны мешать друг другу. Каждая транзакция должна выполняться так, как будто других транзакций в системе нет. (На практике существуют разные *уровни изоляции*, которые допускают те или иные аномалии для повышения производительности).
4. **Durability (Долговечность/Устойчивость):** Если транзакция успешно завершена (зафиксирована), ее результаты должны быть сохранены постоянно и не должны быть потеряны даже в случае сбоев системы (например, отключения питания). Обычно достигается за счет записи изменений в **журналы транзакций (WAL - Write-Ahead Log)** перед применением их к основным файлам данных.

Управление транзакциями в SQL:

- **`BEGIN`** или **`START TRANSACTION`**: Начинает новую транзакцию. В PostgreSQL многие команды DDL (как `CREATE TABLE`) не могут выполняться внутри явного блока `BEGIN...COMMIT`, они сами по себе являются транзакциями. DML команды (`INSERT`, `UPDATE`, `DELETE`) могут быть сгруппированы. Если `BEGIN` не вызван явно, каждая отдельная DML команда часто выполняется в своей собственной неявной транзакции (режим autocommit, зависит от настроек клиента/СУБД).
- **`COMMIT`**: Успешно завершает текущую транзакцию, делая все ее изменения видимыми для других транзакций и постоянными.
- **`ROLLBACK`**: Отменяет все изменения, сделанные в текущей транзакции с момента ее начала (или с последней точки сохранения), и завершает транзакцию.

Пример (Банковский перевод):

```
BEGIN; -- Начать транзакцию

-- Снять деньги со счета Алекса
UPDATE account SET balance = balance - 50.00 WHERE name = 'Alex';

-- Добавить деньги на счет Ивана
UPDATE account SET balance = balance + 50.00 WHERE name = 'Ivan';

-- Если обе операции прошли успешно:
COMMIT; -- Зафиксировать изменения

-- Если на каком-то этапе произошла ошибка (например, недостаточно средств),
-- нужно выполнить ROLLBACK вместо COMMIT:
-- ROLLBACK; -- Отменить все изменения с момента BEGIN
```

Точки сохранения (`SAVEPOINT`):

Позволяют установить “закладку” внутри транзакции, к которой можно будет откатиться, не отменяя всю транзакцию.

- **`SAVEPOINT имя_точки;`**: Устанавливает точку сохранения.
- **`ROLLBACK TO SAVEPOINT имя_точки;`**: Отменяет все изменения, сделанные *после* указанной точки сохранения. Сама точка сохранения остается активной.
- **`RELEASE SAVEPOINT имя_точки;`**: Удаляет точку сохранения, но *не* отменяет изменения, сделанные после нее.

Пример с `SAVEPOINT`:

```
BEGIN;

UPDATE account SET balance = balance - 50.00 WHERE name = 'Alex';

SAVEPOINT savepoint1; -- Установить точку сохранения

UPDATE account SET balance = balance + 50.00 WHERE name = 'Ivan';

-- Предположим, здесь возникла проблема или нужно отменить перевод Ивану
ROLLBACK TO SAVEPOINT savepoint1; -- Откат к состоянию после списания у Алекса

-- Теперь можно попробовать перевести деньги кому-то другому
UPDATE account SET balance = balance + 50.00 WHERE name = 'Ivan2';

COMMIT; -- Фиксируем результат (списание у Алекса и зачисление Ivan2)
```

При откате к точке сохранения (`ROLLBACK TO`), все точки, созданные после* нее, автоматически удаляются.*

Источник — https://xn--b1amah.xn--80aalyho.xn--d1acj3b/mediawiki/index.php?title=БД:Теория:Глава_5&oldid=40