

# лаба 3

## Нормализация. Формы

### Проблемы Плохого Дизайна и Аномалии

Зачем нам вообще нужна нормализация, если мы можем просто создать одну большую таблицу со всеми нужными данными? Рассмотрим таблицу `STUDENTS` с информацией о студентах, их группах и кураторах (GrMentor - Куратор Группы).

#### Таблица STUDENTS (Ненормализованная)

StudID	StudName	Group	GrMentor
1	Ivan Petrov	P3100	Egor Kirov
3	Vasily Ivanov	P3101	Roman Ivov
34	Gleb Anisimov	P3100	Egor Kirov

#### Проблемы:

1. **Избыточность Данных:** Информация о кураторе (GrMentor) повторяется для каждого студента из одной и той же группы (Egor Kirov для группы P3100). Это ведет к неэффективному использованию памяти.
2. **Аномалии Обновления (Update Anomalies):** Если куратор группы P3100 сменится, нам придется обновить поле GrMentor во *всех* строках, где Group = 'P3100'. Если мы обновим не все строки, данные станут противоречивыми (у одной группы окажется несколько кураторов).

```
-- Попытка сменить куратора только для одного студента
UPDATE STUDENTS
SET GrMentor = 'Eugene Lomov'
WHERE StudName = 'Ivan Petrov';
```

#### Результат (Несогласованность):

StudID	StudName	Group	GrMentor
1	Ivan Petrov	P3100	Eugene Lomov (Это новый куратор)
3	Vasily Ivanov	P3101	Roman Ivov
34	Gleb Anisimov	P3100	Egor Kirov (А тут остался старый)

#### 3. Аномалии Вставки (Insertion Anomalies):

- Мы не можем добавить информацию о новой группе и ее кураторе, пока в этой группе нет хотя бы одного студента (потому что StudID, вероятно, часть первичного ключа или просто идентификатор студента, который не может быть NULL для информации о группе).
- При добавлении нового студента в существующую группу, мы должны *правильно* указать ее куратора. Ошибка при вводе имени куратора приведет к несогласованности

```
-- Попытка добавить студентов с разными написаниями куратора
INSERT INTO STUDENTS VALUES(57, 'Nina Simonova', 'P3100', 'E. Kirov');
INSERT INTO STUDENTS VALUES(58, 'Petr Uvarov', 'P3100', 'Egor Lomov');
```

4. **Аномалии Удаления (Deletion Anomalies):** Если мы удалим последнего студента из какой-либо группы (например, Василия Иванова из Р3101), мы потеряем информацию о самой группе Р3101 и ее кураторе Романе Ивове, даже если эта информация нам еще нужна.

**DELETE FROM STUDENTS**

**WHERE** StudName = 'Vasily Ivanov';

*После этого запроса информация о группе Р3101 и ее кураторе исчезнет из таблицы.*

Все эти проблемы возникают из-за того, что в одной таблице смешаны данные о разных сущностях (студенты и группы/кураторы) и существуют “неправильные” зависимости между атрибутами. Нормализация помогает выявить и устранить эти проблемы.

**Нормализация** - формальный метод для проверки/доработки модели на основе функциональных зависимостей.

Процесс нормализации заключается в последовательном приведении таблиц к нормальным формам более высокого порядка. Каждая следующая нормальная форма накладывает более строгие ограничения.

**Ненормализованная Форма (UNF / ONF):** Таблица содержит неатомарные значения (повторяющиеся группы, списки в ячейках). Это нарушает основные принципы реляционной модели.

Пример, в котором значения неатомарны (у одного студента несколько экзаменов, дат и преподавателей)

StudID	StudName	ExamID	ExamName	ExDate	ProfID	ProfName
123	Ivan Ivanov	34	OPD	14.01.19	55	Rebrov A.
		78	DBMS	29.12.20	789	Uvarov S.
345	Egor Kirov	34	OPD	14.01.19	55	Rebrov A.
		87	History	25.01.19	342	Serov G.

## Первая нормальная форма (1NF)

Правило: отношение находится в 1NF, если все его атрибуты содержат только атомарные (неделимые) значения. На пересечении строки и столбца - ровно 1 значение

**Как можно достичь 1NF:**

**Хороший способ:** разделить таблицу на несколько, вынеся повторяющиеся группы в отдельную таблицу со связью через внешний ключ (декомпозиция)

Разделяем одну таблицу, в которой были повторяющиеся значения на две: STUDENTS и EXAMS

**Таблица STUDENTS**

StudID (PK)	StudName
123	Ivan Ivanov
345	Egor Kirov

**Таблица EXAMS**

StudID (FK)	ExamID	ExamName	ExDate	ProfID	ProfName
123	34	OPD	14.01.19	55	Rebrov A.

123	78	DBMS	29.12.20	789	Uvarov S.
345	34	OPD	14.01.19	55	Rebrov A.
345	87	History	25.01.19	342	Serov G.

#### Обе таблицы в 1NF

**Плохой способ:** создать отдельную строку для каждого значения из повторяющейся группы. Это приводит к сильной избыточности (в итоге значения атомарны, но они дублируются)

StudID	StudName	ExamID	ExamName	ExDate	ProfID	ProfName
123	Ivan Ivanov	34	OPD	14.01.19	55	Rebrov A.
123	Ivan Ivanov	78	DBMS	29.12.20	789	Uvarov S.
345	Egor Kirov	34	OPD	14.01.19	55	Rebrov A.
345	Egor Kirov	87	History	25.01.19	342	Serov G.

#### Вторая Нормальная Форма (2NF):

- **Правило:** Отношение находится в 2НФ, если оно находится в 1НФ и все **неключевые атрибуты полностью функционально зависят** от *каждого* потенциального ключа. (Нет частичных зависимостей от ключа).
- **Актуально для таблиц с составными первичными ключами.**
- **Цель:** Устранить избыточность, возникающую из-за того, что неключевой атрибут зависит только от части составного ключа.
- **Как достичь:** Вынести частично зависимые атрибуты и ту часть ключа, от которой они зависят, в отдельную таблицу.

#### Пример (используем "расплюснутую" таблицу из 1НФ):

- Первичный ключ: {StudID, ExamID}.
- Неключевые атрибуты: StudName, ExamName, ExDate, ProfID, ProfName.
- **Частичные зависимости:**
  - StudID → StudName (Зависит только от части ключа - StudID).
  - ExamID → ExamName (Зависит только от части ключа - ExamID).
- **Полные зависимости:**
  - {StudID, ExamID} → ExDate (Предполагаем, что дата зависит от студента и экзамена).
  - {StudID, ExamID} → ProfID (Предполагаем, что преподаватель зависит от студента и экзамена).
- **Другие зависимости:**
  - ProfID → ProfName (Имя преподавателя зависит от его ID).

Таблица не в 2НФ из-за частичных зависимостей StudName и ExamName.

### Декомпозиция для 2НФ:

1. Вносим StudID → StudName:

- Новая таблица STUDENTS: {StudID (PK), StudName}.
- Старая таблица (переименуем в EXAMS\_PARTICIPATION): {StudID (FK), ExamID (PK), ExamName, ExDate, ProfID, ProfName}. (StudName удален).

2. Вносим ExamID → ExamName из EXAMS\_PARTICIPATION:

- Новая таблица EXAMS: {ExamID (PK), ExamName}.
- Таблица EXAMS\_PARTICIPATION: {StudID (FK), ExamID (FK), ExDate, ProfID, ProfName}. (ExamName удален). Первичный ключ теперь {StudID, ExamID}.

#### Таблица STUDENTS

StudID (PK)	StudName
123	Ivan Ivanov
345	Egor Kirov

#### Таблица EXAMS

ExamID (PK)	ExamName
34	OPD
78	DBMS
87	History

#### Таблица EXAMS\_PARTICIPATION

StudID (FK, PK)	ExamID (FK, PK)	ExDate	ProfID	ProfName
123	34	14.01.19	55	Rebrov A.
123	78	29.12.20	789	Uvarov S.
345	34	14.01.19	55	Rebrov A.
345	87	25.01.19	342	Serov G.

Вот пример таблицы, где также не соблюдена 2NF, так как атрибут Bank Location больше зависит от сущности BANK, нежели чем от ACCOUNT

## Second Normal Form (2NF)

An attribute must be dependent on its entity's entire unique identifier.



The Bank Location attribute is dependent on BANK rather than on ACCOUNT. Therefore, this is not in 2NF. Move the attribute to the BANK entity.

### Третья Нормальная Форма (3NF):

- **Правило:** Отношение находится в 3НФ, если оно находится в 2НФ и все **неключевые атрибуты нетранзитивно зависят** от *каждого* потенциального ключа. (Нет транзитивных зависимостей неключевых атрибутов от ключа через другие неключевые атрибуты).
- **Цель:** Устранить избыточность, возникающую из-за того, что неключевой атрибут зависит от другого неключевого атрибута.
- **Как достичь:** Вынести транзитивно зависимые атрибуты и их непосредственный детерминант в отдельную таблицу.

#### Пример (используем таблицы из 2НФ):

- В таблицах STUDENTS и EXAMS нет транзитивных зависимостей (там просто ключ и один неключевой атрибут).
- Рассмотрим EXAMS\_PARTICIPATION:
  - Первичный ключ: {StudID, ExamID}.
  - Неключевые атрибуты: ExDate, ProfID, ProfName.
  - **Зависимости:**
    - {StudID, ExamID} → ExDate (Полная)
    - {StudID, ExamID} → ProfID (Полная)
    - ProfID → ProfName (Зависимость между неключевыми атрибутами!)
  - **Транзитивная зависимость:** {StudID, ExamID} → ProfID и ProfID → ProfName, следовательно, ProfName *транзитивно* зависит от первичного ключа через ProfID.

Таблица EXAMS\_PARTICIPATION не в 3НФ.

#### Декомпозиция для 3НФ:

1. Выносим ProfID → ProfName из EXAMS\_PARTICIPATION:
  - Новая таблица PROFS: {ProfID (PK), ProfName}.

- Таблица EXAMS\_PARTICIPATION: {StudID (FK, PK), ExamID (FK, PK), ExDate, ProfID (FK)}. (ProfName удален, ProfID остался как внешний ключ к PROFS).

Таблицы STUDENTS и EXAMS остаются

#### Таблица PROFS

ProfID (PK)	ProfName
55	Rebrov A.
789	Uvarov S.
342	Serov G.

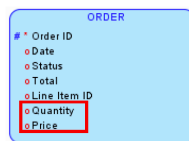
#### Таблица EXAMS\_PARTICIPATION

StudID (FK, PK)	ExamID (FK, PK)	ExDate	ProfID (FK)
123	34	14.01.19	55
123	78	29.12.20	789
345	34	14.01.19	55
345	87	25.01.19	342

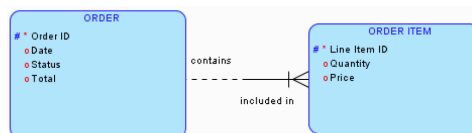
Ещё пример нарушения 3NF:

### Third Normal Form (3NF)

Each attribute depends only on the UID of its entity.



The Quantity and Price attributes are dependent on the Order ID (UID) and Line Item ID (non-UID). Therefore, this is not in 3NF.



Create a new ORDER ITEM entity. Move the Line Item ID, Quantity, and Price attributes to the new entity, and then create an identifying relationship.

ORACLE

11 - 5

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

В ORDER есть Quantity и Price, которые зависят от Order ID и от Line Item ID, то есть есть транзитивная зависимость, проходящая через неключевой атрибут Line Item ID

Order ID → Line Item ID

Line Item ID → Quantity, Price

Тогда Order ID → Quantity, Price

### Нормальная Форма Бойса-Кодда (НФБК / BCNF):

- Более строгая версия ЗНФ.
- **Правило:** Отношение находится в НФБК, если для *каждой* нетривиальной функциональной зависимости  $A \rightarrow B$  детерминант  $A$  является **суперключом** (т.е. содержит в себе потенциальный ключ).
- Большинство таблиц в ЗНФ также находятся и в НФБК. Проблемы могут возникнуть при наличии нескольких перекрывающихся потенциальных ключей.
- Часто является конечной целью нормализации на практике.

## Функциональные зависимости. Виды

**Функциональная зависимость** – ключевое понятие для нормализации.

- **Функциональная зависимость (Functional Dependency, FD)** описывает смысловую связь между атрибутами *внутри одного отношения (таблицы)*.
- Говорят, что атрибут (или набор атрибутов)  $B$  **функционально зависит** от атрибута (или набора атрибутов)  $A$ , если для *каждого* возможного значения  $A$  существует *ровно одно* соответствующее значение  $B$ .
- **Обозначение:**  $A \rightarrow B$  (читается “ $A$  функционально определяет  $B$ ” или “ $B$  функционально зависит от  $A$ ”).
- $A$  называется **детерминантом**.

**Важно:** ФЗ определяются **смыслом данных (семантикой)** предметной области, а не текущими данными в таблице!

**Примеры ФЗ для таблицы STUDENTS (StudID, StudName, Group, GrMentor):**

- $\text{StudID} \rightarrow \text{StudName}$  (По ID студента однозначно определяется его имя).
- $\text{StudID} \rightarrow \text{Group}$  (По ID студента однозначно определяется его группа).
- $\text{StudID} \rightarrow \text{GrMentor}$  (По ID студента можно узнать его группу, а по группе – куратора, т.е. косвенно ID определяет куратора).
- $\text{Group} \rightarrow \text{GrMentor}$  (По номеру группы однозначно определяется ее куратор. Предполагаем, что у группы только один куратор).
- $\text{StudID}, \text{Group} \rightarrow \text{GrMentor}$  (Тоже верно, но избыточно, т.к. Group уже зависит от StudID).

**Примеры НЕ ФЗ:**

- $\text{Group} \rightarrow \text{StudID}$  (Неверно, т.к. в одной группе много студентов).
- $\text{StudName} \rightarrow \text{StudID}$  (Неверно, т.к. могут быть тезки).

**Типы ФЗ:**

- **Тривиальная ФЗ:** Зависимость вида  $A \rightarrow B$ , где  $B$  является подмножеством  $A$ . Например,  $\{\text{StudID}, \text{StudName}\} \rightarrow \text{StudName}$ . Такие зависимости выполняются всегда и не несут полезной информации для нормализации. Обычно рассматривают только **нетривиальные** ФЗ.
- **Полная ФЗ:** Зависимость  $A \rightarrow B$ , где  $A$  – составной детерминант (несколько атрибутов), и  $B$  не зависит ни от какого *подмножества*  $A$ . Мы уже обсуждали это в контексте 2НФ.
- **Частичная ФЗ:** Зависимость  $A \rightarrow B$ , где  $A$  – составной детерминант, и  $B$  зависит от *части*  $A$ . (Пример:  $\{\text{StudID}, \text{ExamID}\} \rightarrow \text{StudName}$ , но при этом  $\text{StudID} \rightarrow \text{StudName}$ . Здесь  $\text{StudName}$  частично зависит от ключа).

- **Транзитивная ФЗ:** Зависимость  $A \rightarrow C$ , которая существует только через промежуточный атрибут  $B$ , такой что  $A \rightarrow B$  и  $B \rightarrow C$ , при этом  $B$  не зависит от  $A$  ( $B \rightarrow A$  неверно) и  $B$  не является частью ключа  $A$ . (Пример:  $StudID \rightarrow Group$  и  $Group \rightarrow GrMentor$ , следовательно,  $StudID \rightarrow GrMentor$  транзитивно через  $Group$ ).

**Аксиомы Армстронга:** Формальные правила для вывода новых ФЗ из существующих:

1. **Рефлексивность:** Если  $B \subseteq A$ , то  $A \rightarrow B$ . (Тривиальная зависимость).
2. **Дополнение (Augmentation):** Если  $A \rightarrow B$ , то  $A, C \rightarrow B, C$ . (Добавление атрибута  $C$  к обеим частям).
3. **Транзитивность:** Если  $A \rightarrow B$  и  $B \rightarrow C$ , то  $A \rightarrow C$ .

## Денормализация

Иногда, после приведения базы данных к высокой нормальной форме (например, 3НФ или НФБК), оказывается, что для выполнения частых запросов требуется слишком много операций соединения (**JOIN**) между таблицами. Это может снижать производительность.

В таких случаях иногда прибегают к **денормализации** — процессу **осознанного** нарушения некоторых правил нормализации для повышения производительности запросов.

- **Прием:** Объединение нескольких таблиц в одну, добавление избыточных данных.
- **Плюсы:**
  - Уменьшение количества соединений в запросах.
  - Потенциальное ускорение выполнения частых запросов на чтение.
- **Минусы:**
  - Увеличение избыточности данных (занимает больше места).
  - Повышенный риск аномалий (вставки, обновления, удаления).
  - Требуется больше усилий для поддержания целостности данных (например, с помощью триггеров или на уровне приложения).

Денормализацию следует применять **осторожно**, только после тщательного анализа производительности и понимания всех рисков. Обычно она является оправданной в системах с преобладанием операций чтения (например, в хранилищах данных для аналитики), где скорость выборки критически важна.

## Язык 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.

### Функции

```
CREATE [ OR REPLACE ] FUNCTION имя_функции ( [ [имя_arg1] тип_arg1, ...] )
RETURNS тип_возвращаемого_значения
-- Или RETURNS TABLE(...) для возврата таблицы,
-- или VOID если ничего не возвращает AS $$ --
-- Или AS '...' - тело функции в $$ или в 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.

### Примеры SQL-функций

```
-- 1 вариант (обращаемся к переменным по номерам)
CREATE FUNCTION updateStudentGroup (int, int) -- указали типы аргументов
RETURNS void
AS '
UPDATE student
SET group_id = $2 -- $2 -- второй аргумент
WHERE student_id = $1 -- $1 - первый аргумент
'
```

```

LANGUAGE SQL;
SELECT updateStudentGroup(100, 2); -- вызываем функцию
-- обновление группы для студента 100 на группу 2

CREATE FUNCTION add_em(int, int) RETURNS integer AS $$
    SELECT $1 + $2;
$$ LANGUAGE SQL;

SELECT add_em(1, 2) AS answer;

-----
-- 2 вариант (используем именованные параметры)
CREATE FUNCTION updateStudentGroup (p_stud_id INT, p_group_id INT)
RETURNS void
AS $$
    UPDATE student
    SET group_id = p_group_id
    WHERE student_id = p_stud_id
$$ LANGUAGE SQL;
-- можно вызывать по позиции
SELECT updateStudentGroup(100, 2);
-- или по именам
SELECT updateStudentGroup(p_stud_id := 100, p_group_id := 2);

```

## Процедуры

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_arg1] тип_arg1, ...] )
AS $$
    -- Тело процедуры (обычно PL/pgSQL)
    $$ LANGUAGE язык;

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

```

## Основы PL/pgSQL

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

```

[ <<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**

```
CREATE FUNCTION somefunc() RETURNS integer AS $$ -- SQL command
<< outerblock »                               -- pl/pgSQL
DECLARE
    quantity integer := 30;
BEGIN
    -- Создаем вложенный блок
    DECLARE
        quantity integer := 80;
    BEGIN
        RAISE NOTICE 'Inner quantity = %', quantity; -- Выводится 80
        RAISE NOTICE 'Outer quantity = %', outerblock.quantity; -- Выводится 30
    END;
    RAISE NOTICE 'Сейчас quantity = %', quantity; -- Выводится 30
    RETURN quantity;
END;
$$ LANGUAGE plpgsql;
```



- Сначала выводится 80, так как 80 - переменная внутреннего блока
- Далее выводится 30, так как 30 - переменная внешнего блока, мы обратились к ней через **outerblock.quantity (метка\_блока.имя\_переменной)**
- END - конец вложенного блока
- Второй раз выводится 30, это переменная внешнего блока, и вышли из вложенного блока
- Тогда в RETURN quantity вернется 30

## Анонимные блоки ( DO )

Позволяют выполнить блок PL/pgSQL кода без создания функции или процедуры.

Удобно для одноразовых задач или скриптов.

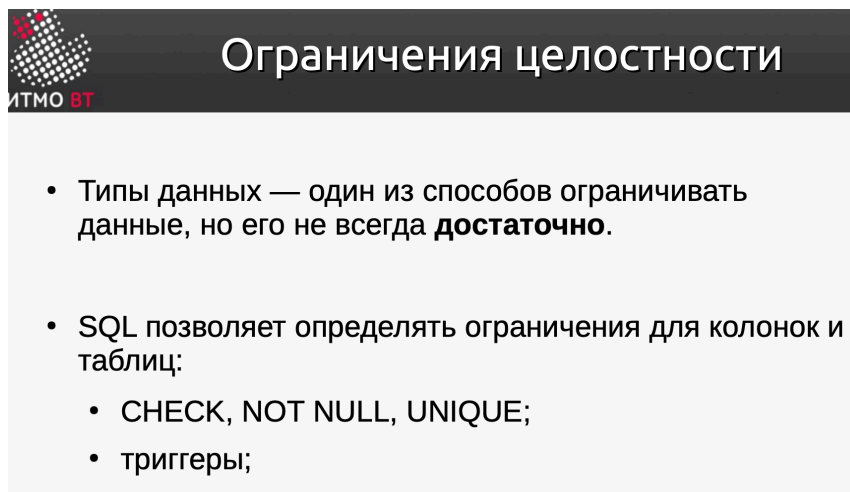
### Структура:

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

### Пример

```
DO $$  
<<studentBlock>>  
DECLARE  
    studCount integer := 0;  
BEGIN  
    SELECT COUNT(*)  
    INTO studCount -- сохраняем результат в переменную studCount  
    FROM student;  
    RAISE NOTICE 'Students: %', studCount; -- выводим результат  
END studentBlock $$;
```

## Триггеры



**Ограничения целостности**

- Типы данных — один из способов ограничивать данные, но его не всегда **достаточно**.
- SQL позволяет определять ограничения для колонок и таблиц:
  - CHECK, NOT NULL, UNIQUE;
  - триггеры;

**Триггеры** — это специальные процедуры, которые **автоматически** выполняются (срабатывают) в ответ на определенные события, происходящие с таблицей (обычно это операции DML: **INSERT**, **UPDATE**, **DELETE**). Могут быть объявлены как для таблицы из пользовательской БД, так и для представления (PostgreSQL).



# Триггеры

Триггеры дают возможность:

- Реализовывать **сложные ограничения целостности** данных, которые невозможно реализовать через ограничения, устанавливаемые при создании таблицы (*CHECK* ...).
- Контролировать информацию, хранимую в таблице, посредством **регистрации вносимых изменений** и пользователей, производящих эти изменения.

Для чего 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.


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**.

## Триггерная функция в PostgreSQL

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** ).



## Переменные триггерной процедуры

Когда PL/pgSQL функция вызывается триггером, создается ряд переменных:

- **NEW** — (RECORD) содержит новую строку базы данных для команд INSERT/UPDATE в строковых триггерах.
- **OLD** — (RECORD) содержит старую строку базы данных для команд UPDATE/DELETE в строковых триггерах.
- **TG\_NAME** — имя сработавшего триггера.
- **TG\_WHEN** — BEFORE, AFTER или INSTEAD OF, в зависимости от определения триггера.

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

```
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 из презентации

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

```
-- создание таблиц
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
);

-- 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;

-- создание триггера
CREATE TRIGGER employee_audit_insert
AFTER INSERT ON employee -- срабатывает после вставки
FOR EACH ROW -- для каждой вставляемой строки
EXECUTE PROCEDURE auditfunc(); -- вызов функции
```

```
-- 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

- 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:

```
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
```

**Для удаления триггера:** `DROP TRIGGER` имя\_триггера **ON** имя\_таблицы;

## 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 сбоя 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.

```
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';  
  
-- Допустим 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 SAVEPOINT savepoint1), все точки, созданные после\* нее, автоматически удаляются.