**1.Преобразование типов в PostgreSQL**

Типизация данных — важный аспект проектирования баз данных, поскольку неправильный выбор типа данных может негативно сказаться на скорости исполнения запросов, целостности данных и удобстве разработки приложений. PostgreSQL предлагает богатый набор инструментов для управления типами данных, включая возможность явного и неявного преобразования типов.

Основные концепции преобразования типов

Типы данных и их классификация

Типы данных классифицируются следующим образом:

* **Простые типы данных**: целые числа (INT, SMALLINT), вещественные числа (FLOAT, REAL), символьные строки (VARCHAR, TEXT), булевские значения (BOOLEAN), временные метки (TIMESTAMP, DATE), денежные единицы (MONEY).
* **Сложные типы данных**: массивы, диапазоны, составные структуры, JSONB, геометрия и другие специализированные типы.

Каждый тип имеет свою семантику, представление и ограничения. Например, целочисленные типы имеют ограничение по размеру, строковые типы поддерживают разные кодировки и режимы сортировки.

Языковые конструкции для преобразования типов

PostgreSQL предоставляет следующие механизмы для явного преобразования типов:

* **Оператор CAST**: универсальная конструкция для приведения любого выражения к нужному типу данных.
* **Каст-шорткат**: альтернативный синтаксический сахар для кастинга, записываемый через двойное двоеточие (::).
* **Специальные функции**: специфичные способы преобразования для отдельных случаев, например, TO\_DATE(), TO\_TIMESTAMP(), TO\_CHAR(), TO\_NUMBER().

Примеры:

-- Приведение строки к целому числу с помощью CAST

SELECT CAST('123' AS INT);

-- Приведение строки к целому числу с помощью ::

SELECT '123'::INT;

-- Специфичная функция для даты

SELECT TO\_DATE('2023-01-01', 'YYYY-MM-DD');

Неявное преобразование типов

Иногда PostgreSQL автоматически выполняет преобразование типов в ситуациях, когда это допустимо. Такие случаи возникают чаще всего при арифметических операциях или сравнениях.

Например, рассмотрим простую операцию сложения двух выражений разных типов:

SELECT '123' + 456;

PostgreSQL попытается преобразовать строку '123' в целое число и успешно выполнит операцию. Но если строка не может быть представлена числом, возникает ошибка:

SELECT '123abc' + 456; -- ERROR!

Таким образом, неявное преобразование удобно, но требует осторожности, особенно в больших системах.

Особенности некоторых операций

Конвертация строковых данных в числовые

Функция TO\_NUMBER() помогает преобразовывать строки в числа даже в тех случаях, когда строка содержит посторонние символы.

Пример:

SELECT TO\_NUMBER('1234фывапр', 'FM9999'); -- вернёт 1234

Формат FM9999 означает, что все лишние символы после найденных цифр отбрасываются. Такой подход полезен, если приходится обрабатывать некритично загрязнённые данные.

Но стоит помнить, что злоупотребление такими функциями может скрыть реальные проблемы качества данных.

Каверзные вопросы по теме преобразования типов

1. **Вопрос:** Что произойдёт при попытке сложить два целых числа с разным знаком (-123 и +456)?**Ответ:** Результат будет обычным арифметическим сложением: −123+456=333−123+456=333.
2. **Вопрос:** Какое отличие между операторами CAST и сокращённым вариантом (::)?**Ответ:** Никаких отличий! Оба варианта выполняют одну и ту же операцию преобразования типов.
3. **Вопрос:** Почему операция SELECT 'abc' + 123 вызывает ошибку?**Ответ:** Потому что PostgreSQL не может автоматически перевести буквенную строку в число. Необходимо сначала привести строку к числовому типу явно или исправить входные данные.
4. **Вопрос:** Есть ли разница между конструкциями CAST(x AS y) и (x)::y?**Ответ:** Нет разницы, обе формы идентичны и служат одной цели.
5. **Вопрос:** Возможна ли ситуация, когда неявное преобразование замедляет работу приложения?**Ответ:** Да, если база данных вынуждена часто производить дополнительные преобразования, производительность может снизиться. Рекомендуется избегать неявных преобразований там, где возможно, и проверять индексы на предмет возможности эффективных преобразований.
6. **Вопрос:** Зачем нужны специальные функции вроде TO\_DATE() или TO\_NUMBER()?**Ответ:** Эти функции позволяют гибко контролировать формат исходных данных и обработку исключительных ситуаций. Они полезны, когда стандартные операторы преобразования недостаточно выразительны.
7. **Вопрос:** Будет ли работать такая конструкция: SELECT ('123'::INT + '456'::INT)?**Ответ:** Да, сработает отлично, потому что оба аргумента приводятся явно к целым числам.
8. **Вопрос:** Могут ли возникать ошибки при неявном преобразовании типов?**Ответ:** Конечно, если автоматическое преобразование невозможно (например, строка не соответствует формату нужного типа). Следует аккуратно относиться к таким случаям.
9. **Вопрос:** Чем отличается работа функции TO\_NUMBER() от простого приведения строки к числу с помощью CAST()?**Ответ:** TO\_NUMBER() позволяет задать маску формата и обработать строки, содержащие лишние символы. Обычный CAST() выдаст ошибку, если строка содержит нечисловые символы.
10. **Вопрос:** Возможно ли принудительно заставить PostgreSQL произвести преобразование, которое кажется невозможным?**Ответ:** Иногда да, если создать специальную кастомную функцию или использовать расширения. Но обычно разумнее нормализовать данные заранее.

2.Подробный разбор команды UPDATE в PostgreSQL с примерами и вопросами

I. Введение в команду UPDATE

Команда UPDATE в PostgreSQL позволяет изменять существующие данные в таблицах базы данных. Это крайне полезная команда, широко применяемая в повседневной деятельности администраторов баз данных и разработчиков приложений. Давайте детально изучим, как работает команда UPDATE, какими возможностями обладает и какие нюансы важно учитывать при её использовании.

II. Основная структура команды UPDATE

Наиболее общий вид команды UPDATE выглядит так:

UPDATE имя\_таблицы

SET столбец1 = значение1,

столбец2 = значение2,

...

[WHERE условие];

Компоненты команды:

* **имя\_таблицы**: Имя таблицы, в которой находятся изменяемые данные.
* **SET**: Устанавливает новые значения для указанного списка столбцов.
* **WHERE**: Необязательная часть, позволяющая задать критерий отбора строк для обновления. Если условие не указано, обновляются все строки таблицы.

Также можно использовать дополнительную секцию RETURNING, которая позволяет получить обновлённые строки после выполнения команды.

III. Алгоритм выполнения команды UPDATE

1. **Выборка строк**. Если указано условие WHERE, выбираются только те строки, которые соответствуют этому условию. Если условие не указано, обновляются все строки таблицы.
2. **Рассчет новых значений**. Новые значения для столбцов рассчитываются согласно правилам, установленным в секции SET.
3. **Замена старых значений**. Старые значения заменяются на новые.
4. **Завершение транзакции**. Если никаких ошибок не возникло, обновление фиксируется.

IV. Примеры практических применений команды UPDATE

Пример 1: Увеличение зарплаты всем сотрудникам на 10%

UPDATE employees

SET salary = salary \* 1.1;

Запрос умножает текущую зарплату каждого сотрудника на коэффициент 1.1, тем самым повысив её на 10%.

Пример 2: Перевод сотрудников отдела Sales в отдел Marketing

UPDATE employees

SET department = 'Marketing'

WHERE department = 'Sales';

Этот запрос переводит всех сотрудников из отдела Sales в отдел Marketing.

Пример 3: Интересный случай: update employees set department = salary

Рассмотрим нестандартный запрос:

UPDATE employees

SET department = salary;

Данный запрос ставит знак равенства между столбцом department и столбцом salary. Поскольку столбец department обычно хранит строки (названия отделов), а столбец salary — числовое значение (заработная плата), после выполнения этого запроса значения станут выглядеть примерно так:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **id** | **first\_name** | **last\_name** | **department** | **salary** |
| 1 | John | Doe | 50000 | 50000 |
| 2 | Jane | Smith | 60000 | 60000 |
| 3 | Alice | Brown | 70000 | 70000 |

## Таким образом, после выполнения такого запроса столбец department потеряет осмысленность, так как вместо названий отделов будут отображаться числовые значения зарплат.

## Результат выполнения запроса и необходимая структура таблицы

Рассмотрим подробнее ситуацию с запросом:

UPDATE employees

SET department = salary;

Анализ запроса

* Данный запрос присваивает столбцу department значение столбца salary.
* Поле department обычно хранит наименование отдела ("Sales", "HR", "Engineering"), а поле salary содержит численную величину заработка сотрудника.
* Присваивание строковому полю значения из численного поля приведёт к замене наименования отдела численными значениями.

Пример структуры таблицы до выполнения запроса

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **id** | **first\_name** | **last\_name** | **department** | **salary** |
| 1 | John | Doe | Sales | 50000 |
| 2 | Jane | Smith | HR | 60000 |
| 3 | Alice | Brown | Engineering | 70000 |

Результаты выполнения запроса

После выполнения запроса структура таблицы будет выглядеть так:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **id** | **first\_name** | **last\_name** | **department** | **salary** |
| 1 | John | Doe | 50000 | 50000 |
| 2 | Jane | Smith | 60000 | 60000 |
| 3 | Alice | Brown | 70000 | 70000 |

Объяснение результата

1. Поле department, раньше содержащее текст («Sales», «HR»), стало содержать численные значения из поля salary.
2. Это привело к потере смысла поля department, так как теперь в нём указаны не названия отделов, а суммы зарплаты.
3. Такой запрос вряд ли окажется полезным в реальной жизни, так как нарушает логичность структуры данных.

Ответ на вопрос

**Итоговый результат выполнения запроса:**

* Столбец department заполнится числами из столбца salary, уничтожив первоначальную информацию о наименовании отдела.

**Необходимая структура таблицы:**

Таблица должна иметь минимум четыре столбца:

* id (идентификатор сотрудника, уникальный)
* first\_name (имя сотрудника)
* last\_name (фамилия сотрудника)
* department (текстовое поле, содержащее название отдела)
* salary (числовое поле, содержащее зарплату сотрудника)

V. Тонкости и особенности команды UPDATE

1. **Неявное преобразование типов**: PostgreSQL автоматически преобразовывает типы данных, если это необходимо. Например, если вы устанавливаете значение строки в числовой столбец, система попытается преобразовать строку в число. Но помните, что не каждое преобразование возможно.
2. **Одновременное обновление нескольких столбцов**: За один запрос можно обновить сразу несколько столбцов, указывая их через запятую.
3. **Возврат обновленных строк**: Используя предложение RETURNING, можно получить обновленные строки непосредственно после выполнения команды:

UPDATE employees

SET salary = salary \* 1.1

RETURNING \*;

1. **Возможность ошибки**: Если условие WHERE не ограничено должным образом, вы рискуете обновить большее количество строк, чем ожидалось. Всегда внимательно проверяйте условие.

VI. Десять каверзных вопросов с ответами

1. **Вопрос:** Что произойдет, если не указать условие WHERE в запросе UPDATE?**Ответ:** Без условия WHERE запрос обновит все строки в таблице.
2. **Вопрос:** Что покажет команда UPDATE employees SET department = salary RETURNING \*;?**Ответ:** Вернет полный набор обновленных строк, где столбец department принял значение столбца salary.
3. **Вопрос:** Можно ли обновить столбец, зависящий от другого столбца?**Ответ:** Да, это разрешено, например: UPDATE employees SET salary = salary \* 1.1;.
4. **Вопрос:** Верно ли утверждение, что команда UPDATE всегда выполняется в рамках транзакции?**Ответ:** Да, верно. Любая команда UPDATE завершается фиксацией транзакции, если она выполнена успешно.
5. **Вопрос:** Какой результат даст запрос UPDATE employees SET salary = salary / 0;?**Ответ:** Возникнет ошибка деления на ноль.
6. **Вопрос:** Как узнать, какие строки были обновлены?**Ответ:** Используйте предложение RETURNING, например: UPDATE employees SET salary = salary \* 1.1 RETURNING \*;.
7. **Вопрос:** Может ли команда UPDATE повредить данные?**Ответ:** Да, неправильное использование (например, пропуск условия WHERE) может привести к повреждению данных.
8. **Вопрос:** Можно ли в команде UPDATE комбинировать данные из разных таблиц?**Ответ:** Да, можно, используя вложенные запросы или соединения (JOINs).
9. **Вопрос:** Что произойдет, если в запросе указана несуществующая колонка?**Ответ:** Запрос завершится с ошибкой, уведомляющей о ненайденном столбце.
10. **Вопрос:** Насколько велика нагрузка на сервер при массовой операции UPDATE?**Ответ:** Объем нагрузки зависит от размера таблицы и сложности условий. Большие объемы данных могут значительно увеличить нагрузку на сервер.

VII. Аналогичные команды в PostgreSQL

* **DELETE**: Удаляет строки из таблицы.
* **INSERT**: Вставляет новые строки в таблицу.
* **MERGE**: Объединяет вставку и обновление в одном запросе (PostgreSQL поддерживает MERGE начиная с версии 15).

Каждая из этих команд решает схожие задачи, но отличается своим назначением и особенностями применения.

**3.Создание копии таблицы по структуре без заполнения через команду CREATE TABLE**

Что такое создание пустой копии структуры таблицы?

При работе с базами данных PostgreSQL часто возникает необходимость создать копию существующей таблицы, но без её содержимого — то есть пустую таблицу с такой же структурой полей, ограничений, индексов и триггеров. Для этого используется специальная конструкция команды CREATE TABLE с указанием ключевых слов LIKE.

Как работает команда LIKE

Команду LIKE используют совместно с командой CREATE TABLE, чтобы новая таблица унаследовала структуру исходной таблицы. Вот общий синтаксис:

CREATE TABLE new\_table\_name (

LIKE existing\_table\_name INCLUDING ALL

);

Здесь:

* new\_table\_name: имя новой создаваемой таблицы.
* existing\_table\_name: имя существующей таблицы, структура которой копируется.
* INCLUDING ALL: ключевое слово, которое включает все ограничения, индексы, комментарии и значения по умолчанию исходной таблицы.

Также существуют дополнительные опции для настройки наследования структуры:

* INCLUDING DEFAULTS: включить значения по умолчанию.
* INCLUDING CONSTRAINTS: включить ограничения целостности (NOT NULL, UNIQUE, CHECK, внешние ключи).
* INCLUDING INDEXES: включить существующие индексы.
* INCLUDING STORAGE: сохранить настройку хранения столбцов.
* INCLUDING COMMENTS: скопировать комментарии на уровне таблиц и колонок.
* INCLUDING IDENTITY: сохранить свойства идентификаторов (например, автоинкремент).
* INCLUDING STATISTICS: включить статистику распределения значений.

Примеры SQL-запросов с разными вариантами использования конструкции LIKE:

**Пример №1:** Создать новую таблицу employees\_copy, полную копию таблицы employees включая все объекты:

CREATE TABLE employees\_copy (

LIKE employees INCLUDING ALL

);

**Пример №2:** Скопировать только поля и ограничения, исключив индексы и другие объекты:

CREATE TABLE employees\_copy (

LIKE employees INCLUDING DEFAULTS INCLUDING CONSTRAINTS

);

**Пример №3:** Копирование всех объектов кроме внешних ключей:

CREATE TABLE employees\_copy (

LIKE employees INCLUDING ALL EXCLUDING FOREIGN KEYS

);

Теория о процессе наследования структур таблиц

Создание пустой копии позволяет быстро организовать новое хранилище данных, основываясь на существующем шаблоне. Например, это полезно для организации временных хранилищ, архивирования данных, тестирования новых версий приложений, разработки аналитических запросов без риска повредить рабочие данные.

При создании копий важно учитывать следующие аспекты:

* Если существует внешний ключ, ссылающийся на оригинальную таблицу, этот ключ автоматически включается в копию, но ссылка останется указывать на оригинал, если явно не указано иначе.
* Индексы позволяют ускорить запросы, но также увеличивают объем памяти базы данных. Поэтому выбор индексации влияет на производительность операций вставки/обновления и выборки.
* Ограничения помогают поддерживать целостность данных, гарантируют отсутствие ошибок ввода неверных данных.

Таким образом, используя конструкцию LIKE, мы можем эффективно управлять дублированием метаданных, поддерживая оптимальную работу нашей базы данных.

Почему не использовать SELECT INTO?

Часто разработчики пытаются сделать копию таблицы с использованием команды SELECT INTO. Однако такая техника имеет ряд недостатков:

* Команда SELECT INTO создает таблицу динамически, копируя строки и поля одновременно. Это занимает больше ресурсов и увеличивает нагрузку на базу данных.
* Структура полученной таблицы зависит от результатов выборки, а не полной схемы оригинальной таблицы.
* Нельзя задать индексирование, ограничение, значения по умолчанию или любые другие специфические настройки заранее.

Использование CREATE TABLE … LIKE решает эти проблемы, предоставляя полный контроль над процессом создания копии структуры.

Практический пример на таблице Employees

Допустим, у вас есть таблица сотрудников с названием employees, содержащая такие поля:

|  |  |
| --- | --- |
| **Column Name** | **Data Type** |
| employee\_id | SERIAL PRIMARY KEY |
| first\_name | VARCHAR(50) NOT NULL |
| last\_name | VARCHAR(50) NOT NULL |
| email | VARCHAR(100) UNIQUE |
| department | VARCHAR(50) |
| hire\_date | DATE DEFAULT CURRENT\_DATE |

Создаем точную копию этой таблицы без записей, включающую всю схему и поведение оригинала:

-- Полная копия структуры, включая все элементы

CREATE TABLE employees\_backup (

LIKE employees INCLUDING ALL

);

Теперь наша база данных содержит две абсолютно одинаковые таблицы по структуре: employees и employees\_backup, но вторая пока пустая.

### Способ №2: Использование CREATE TABLE AS SELECT

Этот метод создаёт новую таблицу на основе выборки из исходной таблицы, но сам запрос специально строят так, чтобы выборка была пустой.

#### Пример:

CREATE TABLE employees\_copy AS

SELECT \* FROM employees WHERE FALSE;

Или альтернативный вариант:

CREATE TABLE employees\_copy AS

SELECT \* FROM employees LIMIT 0;

Оба способа создают точную копию структуры таблицы employees, но записи в новую таблицу не переносятся.

### Отличия двух методов:

* **Метод с LIKE**: Копирует всю структуру, включая индексы, ограничения и комментарии (при использовании соответствующих ключевых слов вроде INCLUDING ALL).
* **Метод с AS SELECT**: Создает простую копию структуры (без ограничений, индексов и комментариев), но быстро генерирует новые таблицы.

Используйте первый способ, если важна полная копия структуры, включая дополнительную метаинформацию, а второй способ подойдет, если достаточно базовой схемы таблицы.

Каверзные вопросы с ответами

1. Чем отличается использование LIKE от команды SELECT INTO?*Ответ:* Конструкция LIKE создаёт только структуру таблицы, тогда как SELECT INTO создаёт таблицу вместе с заполнением её данными.
2. Нужно ли создавать вручную индексы и ограничения при использовании LIKE INCLUDING ALL?*Ответ:* Нет, всё необходимое создаётся автоматически согласно оригиналу.
3. Может ли новая таблица содержать внешнюю ссылку на старую таблицу, если указана опция INCLUDING FOREIGN KEYS?*Ответ:* Да, эта связь сохраняется и переопределять её отдельно не требуется.
4. Какие типы данных можно передать в новом поле таблицы при создании копии?*Ответ:* Типы данных будут такими же, как в исходной таблице, даже если поле AUTO\_INCREMENT или генерируемый серийный номер.
5. Можно ли изменить порядок следования столбцов при копировании структуры таблицы?*Ответ:* Порядок столбцов сохранится таким же, как в исходной таблице, если не задано специально другое расположение.
6. Включаются ли вычисляемые поля при выполнении операции LIKE INCLUDING ALL?*Ответ:* Да, вычисляемые поля и виртуальные столбцы сохраняются.
7. Возможна ли передача комментариев на уровнях таблиц и столбцов при использовании конструкции LIKE INCLUDING COMMENTS?*Ответ:* Да, комментарий передаётся автоматически, обеспечивая перенос документации между объектами.
8. Какой оператор позволит исключить включение определённых элементов структуры?*Ответ:* Оператор EXCLUDING, например, EXCLUDING FOREIGN KEYS.
9. Поддерживает ли PostgreSQL передачу типов данных, используемых в полях ENUM при помощи оператора LIKE?*Ответ:* Да, типы ENUM передаются автоматически, сохраняя своё определение.
10. Зачем нужны временные таблицы, созданные командой LIKE?*Ответ:* Временные таблицы используются для промежуточных вычислений, быстрого теста сценариев обработки данных или временного хранения промежуточных результатов без влияния на основную рабочую среду.

**4.** Различия операторов TRUNCATE и DELETE в PostgreSQL

- \*\*DELETE\*\*: Удаляет отдельные строки на основании условия (оператор `WHERE`), поддерживая транзакции и триггеры. Операция журналируется и занимает больше ресурсов, поскольку выполняется построчно.

```sql

DELETE FROM table\_name WHERE condition;

```

- \*\*TRUNCATE\*\*: Быстро удаляет ВСЕ записи из таблицы, не сохраняя никаких журналов изменений. Команда не поддерживает условие удаления отдельных строк и работает быстрее, потому что очищает всю таблицу целиком. Эта команда не запускает триггеры и сразу освобождает пространство.

```sql

TRUNCATE TABLE table\_name;

```

Таким образом, если нужно быстро удалить все записи без сохранения истории изменений, предпочтительнее использовать `TRUNCATE`. Но если важна целостность данных и нужны гарантии безопасности (триггеры, ограничения целостности), лучше выбрать `DELETE`.

Различия операторов TRUNCATE и DELETE в PostgreSQL

Операторы TRUNCATE и DELETE выполняют схожую задачу — удаление строк из таблицы, однако делают это совершенно разными способами, каждый из которых подходит для разных ситуаций. Рассмотрим подробнее различия между ними.

1. Механизм удаления данных

* **DELETE**: Удаляет строки одну за другой, проверяя каждую строку на наличие условий (WHERE), применяя правила безопасности транзакций и триггеры.
* **TRUNCATE**: Быстро удаляет сразу все строки таблицы, минуя проверку индивидуальных условий, триггеры и блокировки отдельных строк.

2. Скорость исполнения

* **DELETE**: Медленнее, поскольку выполняется построчно, проверяется каждое условие, срабатывают триггеры и журналируются изменения в журнале транзакций.
* **TRUNCATE**: Значительно быстрее, потому что операция производится одномоментно и целиком на уровне таблицы, без проверки каждой отдельной записи.

3. Триггеры и события

* **DELETE**: Запускает триггеры, привязанные к событию удаления строк. Это значит, что любой код внутри триггеров сработает при каждом удалении строки.
* **TRUNCATE**: Игнорирует триггеры, предназначенные для событий удаления строк (BEFORE ROW TRIGGER, AFTER ROW TRIGGER). Только специальные триггеры уровня таблицы типа FOR EACH STATEMENT могут обработать событие.

4. Возврат значений и возвращаемые наборы

* **DELETE**: Возвращает количество удаленных строк, можно вернуть набор удаленных данных с использованием конструкции RETURNING.
* **TRUNCATE**: Не возвращает никаких данных и не поддерживает механизм возвращения значений.

5. Транзакционная поддержка

* **DELETE**: Является частью транзакции и откатывается, если транзакция прерывается или отменяется.
* **TRUNCATE**: Так же поддерживается в рамках транзакций, но требует особой осторожности, так как сам по себе не допускает возврат удаленной информации обратно (кроме случаев отката транзакции).

6. Возможность выбора условия удаления

* **DELETE**: Позволяет выбрать конкретные строки для удаления с помощью выражения WHERE.
* **TRUNCATE**: Всегда удаляет ВСЕ строки одной таблицы или нескольких таблиц разом, нельзя применить условие фильтрации.

7. Автоматическая очистка журналов

* **DELETE**: Журнал изменений записывается индивидуально для каждой удаляемой строки, увеличивая объём журнала транзакций.
* **TRUNCATE**: Журналирование минимально, лишь фиксируются факты изменения самой таблицы, а не каждой конкретной строки.

Когда использовать?

* Используйте **DELETE**, если вам необходимо удалить отдельные строки, соблюдая бизнес-правила и триггеры, или хотите вернуть удалённые данные.
* Применяйте **TRUNCATE**, если нужно мгновенно очистить всю таблицу, сэкономить ресурсы и минимизировать журнал транзакций.

Подытожим ключевые отличия:

|  |  |  |
| --- | --- | --- |
| **Отличие** | **DELETE** | **TRUNCATE** |
| Способ удаления | По одной строке | Все строки сразу |
| Проверка условий | Возможно через WHERE | Невозможно |
| Триггеры | Работают | Отключены (за исключением особых) |
| Время исполнения | Дольше | Быстрее |
| Возврат данных | Возможно RETURNING | Невозможен |
| Транзакционность | Учитывается | Учитывается |
| Фиксация в журнале | Каждая запись фиксируется | Минимальное журналирование |

Эти знания позволят грамотно выбирать нужный оператор в зависимости от ситуации и избегать неоправданных затрат ресурсов.

**5.В PostgreSQL можно добавлять различные виды ограничений (constraints)** для контроля качества и достоверности данных, хранящихся в таблице. Эти ограничения обеспечивают соблюдение правил целостности данных и предотвращают ввод некорректных или противоречащих друг другу сведений. Давайте рассмотрим наиболее распространённые типы ограничений, доступные в PostgreSQL.

Ограничения используются для поддержания целостности данных в таблицах. Основные виды ограничений включают:

- \*\*NOT NULL\*\*: Столбец обязательно должен иметь значение.

- \*\*UNIQUE\*\*: Значение в данном столбце уникально среди всех записей.

- \*\*PRIMARY KEY\*\*: Уникальное и обязательное значение, используемое для идентификации каждой строки.

- \*\*FOREIGN KEY\*\*: Связь с другим полем в другой таблице (обеспечивает ссылочную целостность).

- \*\*CHECK\*\*: Проверка соблюдения определённых правил (условий). Например, проверка диапазона допустимых значений.

- \*\*DEFAULT\*\*: Присваивание стандартного значения полю при вставке новых записей.

Эти ограничения задаются при создании таблицы или позже посредством команды ALTER TABLE

### Основные типы ограничений в PostgreSQL

1. **NOT NULL**Обеспечивает обязательное наличие значения в столбце. То есть, при попытке записать значение NULL система выдаст ошибку.

ALTER TABLE employees ADD COLUMN phone\_number TEXT NOT NULL;

Пример:

* + Телефон сотрудника обязательно должен быть указан.

1. **UNIQUE**Гарантирует уникальность значений в данном столбце или группе столбцов. Этот тип ограничения предотвращает повторение одних и тех же данных.

ALTER TABLE employees ADD CONSTRAINT unique\_email UNIQUE(email);

Пример:

* + Электронная почта должна быть уникальной среди всех сотрудников.

1. **PRIMARY KEY**Определяет уникальный идентификатор строки. Первичный ключ обязан иметь уникальное значение и не может быть пустым (NULL).

ALTER TABLE employees ADD PRIMARY KEY(employee\_id);

Пример:

* + Каждый сотрудник должен иметь уникальный ID.

1. **FOREIGN KEY**Устанавливает отношение между двумя таблицами. Внешний ключ гарантирует, что значение, указанное в дочерней таблице, должно существовать в родительской таблице.

ALTER TABLE orders ADD FOREIGN KEY(customer\_id) REFERENCES customers(id);

Пример:

* + Заказ должен относиться к реально существующему клиенту.

1. **CHECK**Использует выражение, определяющее допустимые значения для столбца или группы столбцов. Любые попытки ввести недопустимое значение приведут к ошибке.

ALTER TABLE employees ADD CHECK(age >= 18 AND age <= 65);

Пример:

* + Возраст сотрудника должен находиться в диапазоне от 18 до 65 лет.

1. **DEFAULT**Задаёт стандартное значение для столбца, которое применяется, если значение явно не указано при вставке данных.

ALTER TABLE employees ALTER COLUMN hire\_date SET DEFAULT NOW();

Пример:

* + Дата приёма сотрудника автоматически устанавливается равной сегодняшнему дню.

1. **EXCLUDE**Используется для предотвращения перекрытия диапазонов или областей в значениях одного или нескольких столбцов. Часто применяется для защиты от конфликтов между интервалами дат, временными периодами и подобными элементами.
2. CREATE TABLE reservations (
3. room\_id INT,
4. period TSTZRANGE,
5. EXCLUDE USING gist (room\_id WITH =, period WITH &&)

);

Пример:

* + Бронирование номера отеля не может пересекаться с другим бронированием того же номера.

### Дополнительные возможности и нюансы

Помимо базовых видов ограничений, PostgreSQL предлагает возможность объединения различных ограничивающих механизмов:

* Несколько уникальных индексов могут накладываться на разные комбинации столбцов.
* Одно и то же ограничение можно наложить несколькими способами, выбирая оптимальный для конкретного случая.
* Ограничения можно устанавливать как на этапе создания таблицы, так и позже с помощью команды ALTER TABLE.

Кроме того, PostgreSQL позволяет настроить режим проверки ограничений:

* **Deferrable constraints**: Некоторые ограничения можно проверять не немедленно, а отложить проверку до конца транзакции.

**6**. В postgres какие объекты бд есть

Объектами базы данных в PostgreSQL называются структурированные элементы, составляющие базу данных и обеспечивающие хранение, обработку и управление информацией. Эти объекты делятся на разные категории и выполняют конкретные задачи. Давайте рассмотрим подробнее, какие бывают объекты в PostgreSQL и какую роль они играют.

### Основные объекты PostgreSQL:

1. **Таблицы (Tables)**:
   * Основное средство хранения данных. Таблица состоит из набора колонок (fields) и строк (records). Колонки определяют структуру данных, а строки содержат фактические данные.
2. **Представления (Views)**:
   * Виртуальная таблица, основанная на одном или нескольких запросах. Представления обеспечивают удобные способы просмотра и анализа данных из одной или нескольких таблиц. Данные физически не хранятся в представлении, оно динамически формируется при каждом обращении.
3. **Индексы (Indexes)**:
   * Структуры данных, предназначенные для повышения производительности запросов путём быстрого нахождения нужных записей. Индексы создаются на основе конкретных колонок или выражений и ускоряют операции поиска и сортировки.
4. **Последовательности (Sequences)**:
   * Специальные объекты, генерирующие последовательность уникальных целых чисел. Часто применяются для формирования уникального первичного ключа (auto-increment).
5. **Функции (Functions)**:
   * Подпрограммы, написанные на встроенном языке PL/pgSQL или другом поддерживаемом языке (Python, Perl, C и т.д.). Используются для реализации бизнес-логики, расчетов и других действий.
6. **Триггеры (Triggers)**:
   * Процедуры, автоматически запускающиеся при выполнении определённых событий (insert, update, delete). Например, триггер может автоматически создавать резервную копию удаляемого элемента.
7. **Материалы представления (Materialized views)**:
   * Фиксированная версия виртуальной таблицы, которая сохраняет результат заранее рассчитанного запроса. Используется для повышения производительности сложных запросов путём предварительной подготовки результатов.
8. **Права доступа (Privileges)**:
   * Определённые разрешения на выполнение действий (чтение, запись, модификация и удаление данных) для отдельных пользователей или ролей.
9. **Роли (Roles)**:
   * Пользователи и группы пользователей, которым назначаются права доступа. Роли позволяют удобно организовать систему безопасности и разделить ответственность среди администраторов и конечных пользователей.
10. **Правила (Rules)**:
    * Средства переопределения поведения запросов. Можно настроить правило так, чтобы запрос направлялся в другую таблицу или обрабатывался особым образом.
11. **Расширения (Extensions)**:
    * Добавочные модули, увеличивающие возможности PostgreSQL. К примеру, расширение PostGIS добавляет поддержку географических данных, pg\_trgm улучшает полнотекстовый поиск.
12. **Типы данных (Data types)**:
    * Определяют форматы хранимых данных. Стандартные типы включают целые числа, строки, даты, временные отметки, денежные суммы и многое другое. Возможны кастомные типы данных.

### Как работают объекты?

Каждый объект имеет своё назначение и влияет на производительность и удобство работы с базой данных. Например:

* **Таблицы** хранят основную массу данных, обеспечивая организацию информации.
* **Индексы** улучшают скорость поиска данных.
* **Представления** упрощают работу аналитиков и разработчиков, предоставляя удобную форму отображения данных.
* **Функции и триггеры** позволяют автоматизировать рутинные задачи и повысить безопасность системы.

Управление объектами осуществляется с помощью специальных команд SQL, таких как:

* CREATE TABLE — создание таблицы,
* DROP VIEW — удаление представления,
* GRANT PRIVILEGES — предоставление прав доступа,
* REVOKE PRIVILEGE — отзыв прав доступа,
* CREATE FUNCTION — создание пользовательской функции.

### 7.Основные моменты о последовательностях

Последовательности (Sequence) — это объекты базы данных PostgreSQL, предназначенные для автоматического создания уникальных числовых значений (идентификаторов). Они часто используются для задания первичных ключей таблиц или генерации номеров заказов/документов.

### Создание последовательности (CREATE SEQUENCE)

Команда CREATE SEQUENCE служит для создания новой последовательности. Рассмотрим синтаксис и доступные параметры:

CREATE SEQUENCE [IF NOT EXISTS] имя\_последовательности

[INCREMENT BY increment]

[MINVALUE minvalue | NO MINVALUE]

[MAXVALUE maxvalue | NO MAXVALUE]

[START WITH start]

[CACHE cache\_size]

[CYCLE | NO CYCLE];

Где:

* **INCREMENT BY increment**: Размер шага (обычно положительное число для возрастания, отрицательное — для убывания).
* **MINVALUE minvalue**: Минимально возможное значение последовательности.
* **MAXVALUE maxvalue**: Максимально возможное значение последовательности.
* **START WITH start**: Первое значение, выдаваемое последовательностью.
* **CACHE cache\_size**: Количество предварительно зарезервированных значений (для ускорения выдачи последующих номеров).
* **CYCLE**: После достижения максимального значения последовательность снова начинается сначала.
* **NO CYCLE**: По достижении предела выдача новых значений прекращается (это поведение по умолчанию).

Примеры:

-- Простая последовательность, начинающаяся с 1 и увеличивающаяся на 1

CREATE SEQUENCE simple\_seq START WITH 1 INCREMENT BY 1;

-- Последовательность с циклом

CREATE SEQUENCE cyclic\_seq START WITH 1 INCREMENT BY 1 MAXVALUE 100 CYCLE;

### Изменение свойств последовательности (ALTER SEQUENCE)

Команду ALTER SEQUENCE используют для изменения настроек уже существующей последовательности:

ALTER SEQUENCE имя\_последовательности

[RESTART [WITH restart]]

[INCREMENT BY increment]

[MINVALUE minvalue | NO MINVALUE]

[MAXVALUE maxvalue | NO MAXVALUE]

[RESTART]

[OWNED BY { table.column | NONE }]

[RENAME TO new\_name]

[OWNER TO новый\_владелец]

[SET SCHEMA новое\_пространство\_имен];

Основные опции:

* **RESTART**: Перезапуск последовательности с начальным значением.
* **OWNED BY**: Связывание последовательности с определенной колонкой таблицы (часто для автогенерации значений).
* **RENAME TO**: Переименование последовательности.
* **OWNER TO**: Назначение владельца последовательности.

Пример:

-- Увеличить размер шага и перезапустить последовательность

ALTER SEQUENCE simple\_seq INCREMENT BY 2 RESTART WITH 1;

### Удаление последовательности (DROP SEQUENCE)

Команда DROP SEQUENCE предназначена для удаления последовательности из базы данных:

DROP SEQUENCE [IF EXISTS] имя\_последовательности;

Пример:

-- Удалить последовательность, если она существует

DROP SEQUENCE IF EXISTS simple\_seq;

### Работа с последовательностью

Следующие функции позволяют взаимодействовать с последовательностью непосредственно:

* **nextval(sequence)**: Возвращает следующее значение из последовательности.
* **currval(sequence)**: Показывает последнее выданное значение последовательности.
* **setval(sequence, value)**: Устанавливает заданное значение как последующее в последовательности.

Примеры:

-- Получить первое значение последовательности

SELECT nextval('simple\_seq');

-- Проверить текущее значение

SELECT currval('simple\_seq');

-- Установить стартовое значение

SELECT setval('simple\_seq', 10);

#### Что такое последовательность?

Последовательность — это объект базы данных, который автоматически генерирует уникальные числа. Последовательности полезны там, где требуется автоматическое присвоение уникального идентификатора каждой записи таблицы.

#### Как создать последовательность?

Создание последовательности выполняется командой CREATE SEQUENCE. Например:

CREATE SEQUENCE my\_sequence START WITH 1 INCREMENT BY 1 NO MINVALUE NO MAXVALUE CACHE 1;

Здесь:

* **CREATE SEQUENCE**: Команда для создания новой последовательности.
* **START WITH 1**: Начальное значение последовательности.
* **INCREMENT BY 1**: Шаг увеличения последовательности (каждый раз прибавляет единицу).
* **NO MINVALUE / NO MAXVALUE**: Нет минимальной и максимальной границы, последовательность может расти бесконечно.
* **CACHE 1**: Кэширование одного значения (используется для улучшения производительности при работе с последовательностью).

#### Использование последовательности

Для извлечения следующего значения из последовательности используется команда nextval():

SELECT nextval('my\_sequence');

Эта команда вернёт следующее число из последовательности каждый раз, когда её вызывают.

#### Связывание столбца с последовательностью

Можно связать столбец таблицы непосредственно с последовательностью. Обычно это делается при создании таблицы следующим образом:

CREATE TABLE example\_table (

id SERIAL PRIMARY KEY,

name TEXT NOT NULL

);

SERIAL — это псевдоним типа поля, означающий создание целочисленного столбца, связанный с новой последовательностью, которая автоматически генерирует уникальное значение для каждого нового вставляемого элемента.

Это эквивалентно следующему:

CREATE SEQUENCE example\_table\_id\_seq;

CREATE TABLE example\_table (

id INTEGER DEFAULT nextval('example\_table\_id\_seq') PRIMARY KEY,

name TEXT NOT NULL

);

ALTER SEQUENCE example\_table\_id\_seq OWNED BY example\_table.id;

#### Примеры управления последовательностями

Помимо команды nextval(), также существуют другие полезные функции:

* **currval()**: возвращает последнее использованное значение последовательности.
* **setval()**: устанавливает новое значение последовательности вручную.

Например:

-- Установка текущего значения последовательности вручную

SELECT setval('my\_sequence', 100);

-- Получаем текущее значение

SELECT currval('my\_sequence');

### Зачем нужны последовательности?

Последовательности позволяют легко управлять уникальным идентификационным номером записей в таблице, гарантируют отсутствие дубликатов и упрощают управление ключами.

## Заключение

Таким образом, последовательности в PostgreSQL предоставляют удобный способ автоматической генерации уникальных чисел, что особенно полезно при работе с большими объёмами данных и первичной настройке структуры базы данных.