[Операции INSERT, UPDATE и DELETE с помощью оператора "MERGE". Синтаксис и основные элементы 2](#_Toc197424219)

[Операторы ветвления логики. "DECODE", "CASE WHEN"(2 способа). Синтаксис и основные элементы. 2](#_Toc197424220)

[Иерархические запросы с помощью WITH RECURSIVE + UNION ALL 2](#_Toc197424221)

[Оконные/аналитические функции. "LEAD()", "LAG()", "FIRST\_VALUES()", "LAST\_VALUE()", "RANK()", "DENSE\_RANK()", "ROW\_NUMBER()" 2](#_Toc197424222)

[Использование оконных функций для расчёта накопительной суммы, скользящего среднего, центрированного среднего. Конструкции вида "SUM (\*) / AVG (\*) / MIN(\*) / MAX(\*) OVER (PARTITION BY \* ORDER BY)". Синтаксис. Разница работы агрегационных функций с выражением "order by" и без 2](#_Toc197424223)

[Расширенные опции group by "GROUPING SETS", "CUBE" и "ROLLUP" 2](#_Toc197424224)

[Функция CROSSTAB().  Принцип работы, синтаксис и основные элементы 2](#_Toc197424225)

[Группирование нескольких строк в одну с помощью оператора "STRING\_AGG()". Принцип работы, синтаксис и основные элементы 2](#_Toc197424226)

[Создание таблиц. Добавление внешних ключей, значений столбцов по умолчанию, уникальных ограничений целостности. Удаление таблиц и его параметры. Переименование таблиц. Truncate таблиц. Чем Truncate отличается от Delete. 2](#_Toc197424227)

[Создание представлений (View) 2](#_Toc197424228)

[Создание материализованных представлений. Инкрементальное обновление материализованных представлений 2](#_Toc197424229)

[Создание последовательностей (Sequence). Использование Sequence как Default значение при создание таблицы. Реинициализация Sequence. 2](#_Toc197424230)

[Создание индексов. Различные типы индексов 2](#_Toc197424231)

[Обзор структуры (Блоков) PL/pgSQL 2](#_Toc197424232)

[Переменные и типы PL/pgSQL 2](#_Toc197424233)

[Условные операторы (Логика выполнения). Операторы "IF", "THEN", "ELSE", "ELSIF" и "END IF" PL/pgSQL 3](#_Toc197424234)

[Циклы (Петли, Loops). "WHILE loops", "FOR loops"  PL/pgSQL 3](#_Toc197424235)

[Курсоры (Cursors). Использование курсоров вместе с циклами PL/pgSQL 3](#_Toc197424236)

[Обработка исключений (Exceptions) PL/pgSQL 3](#_Toc197424237)

[Процедуры (Procedure). Создание, удаление и вызов. Просмотр информации о процедурах PL/pgSQL 3](#_Toc197424238)

[Функции (Function). Создание, удаление и вызов. Просмотр информации о процедурах PL/pgSQL 3](#_Toc197424239)

[Пакеты (Package). Создание, удаление и вызов процедур и функций из пакета. Просмотр информации о процедурах PL/pgSQL 3](#_Toc197424240)

[Триггеры. Типы триггеров. Создание, удаление, отключение (Disabling), включение (Enabling). Варианты срабатывания триггеров. Просмотр информации о триггерах PL/pgSQL 3](#_Toc197424241)

## **Операции INSERT, UPDATE и DELETE с помощью оператора "MERGE". Синтаксис и основные элементы**

Ранее для достижения схожего эффекта приходилось вручную писать длинные и сложные последовательности инструкций (INSERT, UPDATE, DELETE), что могло приводить к проблемам с производительностью и усложнению кода. Теперь оператор **MERGE** объединяет всё это в единый декларативный запрос.

### **Зачем нужен оператор MERGE?**

Цель оператора **MERGE** — объединить данные из одной таблицы с другой, автоматически решая, какие строки следует вставить, обновить или удалить. Это чрезвычайно полезно в ситуациях, когда необходимо периодически синхронизировать две таблицы (например, при загрузке свежих данных из стороннего источника).

Например, представьте себе таблицу сотрудников (employees), в которую регулярно поступают свежие данные о зарплате и адресе проживания. Чтобы обеспечить синхронность, нужно проверять каждую строку и проводить соответствующие операции:

* Если сотрудник уже присутствует в таблице, но его зарплата изменилась — обновить его запись.
* Если новый сотрудник появляется впервые — вставить его запись.
* Если сотрудник ушел из компании и его данные отсутствуют в новом источнике — удалить его запись.

Всё это легко реализуется с помощью единого оператора **MERGE**.

### **Синтаксис оператора MERGE**

Простейший шаблон выглядит так:

MERGE INTO target\_table t

USING source\_table s

ON match\_condition

WHEN MATCHED THEN

UPDATE SET t.column1 = s.column1, t.column2 = s.column2, ...

WHEN NOT MATCHED THEN

INSERT (column1, column2, ...) VALUES (s.column1, s.column2, ...);

Описание элементов:

* **target\_table**: Цель, куда происходят изменения (обычно ваша рабочая таблица).
* **source\_table**: Источник данных, сравниваемый с целью.
* **match\_condition**: Критерий, по которому происходит сопоставление (обычно уникальное поле или группа полей).
* **WHEN MATCHED THEN**: Блок, выполняющийся, если строки совпадают (чаще всего здесь выполняют обновление).
* **WHEN NOT MATCHED THEN**: Блок, запускающийся, если строка из источника не соответствует ни одной строке в цели (используют для вставки новых данных).

### **Пример на основе таблицы Employees**

Создадим простую демонстрационную таблицу сотрудников:

CREATE TABLE employees (

employee\_id SERIAL PRIMARY KEY,

first\_name VARCHAR(50),

last\_name VARCHAR(50),

salary DECIMAL(10, 2)

);

INSERT INTO employees (first\_name, last\_name, salary)

VALUES ('Иван', 'Иванов', 80000),

('Пётр', 'Петров', 70000),

('Анна', 'Иванова', 60000);

Затем создадим источник данных (например, свежий импорт CSV-файлов):

CREATE TABLE new\_data (

employee\_id INT,

first\_name VARCHAR(50),

last\_name VARCHAR(50),

salary DECIMAL(10, 2)

);

INSERT INTO new\_data (employee\_id, first\_name, last\_name, salary)

VALUES (1, 'Иван', 'Иванов', 85000), -- Изменилась зарплата

(2, 'Пётр', 'Петров', 70000), -- Осталась прежней

(4, 'Дмитрий', 'Смирнов', 90000); -- Новый сотрудник

Теперь воспользуемся оператором **MERGE** для синхронизации:

MERGE INTO employees e

USING new\_data n

ON e.employee\_id = n.employee\_id

WHEN MATCHED THEN

UPDATE SET salary = n.salary

WHEN NOT MATCHED THEN

INSERT (first\_name, last\_name, salary)

VALUES (n.first\_name, n.last\_name, n.salary);

**Объяснение:**

* Если запись совпадает по employee\_id, обновляем зарплату.
* Если совпадения нет (новый сотрудник), вставляем новую запись.

### **Расширенная версия с поддержкой удаления**

Допустим, сотрудник уходит из компании, и соответствующая запись исчезает из свежего источника данных. В таком случае её нужно удалить из рабочей таблицы. Вот как это реализовать:

MERGE INTO employees e

USING new\_data n

ON e.employee\_id = n.employee\_id

WHEN MATCHED THEN

UPDATE SET salary = n.salary

WHEN NOT MATCHED THEN

INSERT (first\_name, last\_name, salary)

VALUES (n.first\_name, n.last\_name, n.salary)

WHEN NOT MATCHED BY SOURCE THEN

DELETE;

Новый блок **WHEN NOT MATCHED BY SOURCE THEN DELETE** инициирует удаление записей, отсутствующих в источнике.

### **Вопросы и ответы по теме**

1. **Вопрос:** Что произойдёт, если не указать условие **ON** в запросе **MERGE**?**Ответ:** Запрос не сможет сопоставить строки между источником и целью, что сделает невозможным правильное применение блоков MATCHED и NOT MATCHED.
2. **Вопрос:** Можно ли одновременно производить и вставку, и обновление в одном запросе **MERGE**?**Ответ:** Да, именно в этом и состоит суть оператора **MERGE** — он позволяет вставлять новые строки и обновлять существующие в едином запросе.
3. **Вопрос:** Верно ли утверждение, что **MERGE** эффективнее, чем раздельные инструкции **INSERT**, **UPDATE** и **DELETE**?**Ответ:** Да, в большинстве случаев **MERGE** оптимизирует нагрузку на базу данных, сокращая количество обращений к индексам и минимизируя количество проходов по данным.
4. **Вопрос:** Может ли оператор **MERGE** вызывать конфликты данных?**Ответ:** Да, если нарушается уникальность ключей или наложены ограничения на уровне базы данных (например, внешний ключ).
5. **Вопрос:** Поддерживается ли конструкция **MERGE** в предыдущих версиях PostgreSQL?**Ответ:** Официально поддержка появилась только в PostgreSQL 14. Ранее использовали обходные пути, такие как PL/PgSQL-процедуры или отдельные запросы.
6. **Вопрос:** Какие действия выполняются первым: проверка соответствия или обработка блока **WHEN MATCHED**?**Ответ:** Первым делом выполняется проверка соответствия строк по условию **ON**, и только потом активируется нужный блок (MATCHED или NOT MATCHED).
7. **Вопрос:** Сколько раз можно использовать блок **WHEN MATCHED** в одном запросе **MERGE**?**Ответ:** Один запрос допускает только один блок **WHEN MATCHED** и один блок **WHEN NOT MATCHED**.
8. **Вопрос:** Можно ли отказаться от выполнения какого-то действия (например, вставки), если оно кажется неуместным?**Ответ:** Да, для этого используйте блок **DO NOTHING** вместо **INSERT** или **UPDATE**.
9. **Вопрос:** Поддерживаются ли операции удаления данных в рамках оператора **MERGE**?**Ответ:** Да, с помощью блока **WHEN NOT MATCHED BY SOURCE THEN DELETE** можно реализовывать удаление строк, отсутствующих в источнике.
10. **Вопрос:** Должны ли обе таблицы иметь одинаковую структуру?**Ответ:** Нет, важна только возможность сопоставления по ключевым полям, а сами таблицы могут различаться по количеству и типу столбцов.

### **Сравнения с другими аналогичными командами**

До появления **MERGE** пользователи PostgreSQL вынуждены были прибегать к нескольким разным операторам:

* **INSERT**: Для вставки новых строк.
* **UPDATE**: Для обновления существующих строк.
* **DELETE**: Для удаления устаревших или несуществующих строк.

## **Операторы ветвления логики. "DECODE", "CASE WHEN"(2 способа). Синтаксис и основные элементы.**

Decode в оракле!

## **CASE WHEN в SQL**

### **Теоретическое введение**

Операторы ветвления логики в SQL предназначены для изменения порядка выполнения инструкций в зависимости от определенных условий. Они обеспечивают возможность динамического выбора результата в зависимости от состояния данных. Наиболее распространенные операторы ветвления — это DECODE и CASE WHEN, используемые для разветвления логики исполнения запросов и формирования отчетов.

Эти операторы широко применяются в аналитике и обработке данных, помогая структурировать данные и создавать удобные отчёты. Несмотря на внешнюю схожесть, они имеют ряд отличительных черт, которые влияют на выбор оптимального инструмента для конкретной задачи.

### **2. Оператор CASE WHEN (два способа)**

**Назначение:**CASE WHEN — универсальный оператор, поддерживающий как простое сравнение (подобно DECODE), так и проверку сложных условий. Существуют два основных подхода к использованию оператора CASE: простая форма и расширенная форма.

#### a) Простая форма CASE

**Особенности:**

* Позволяет сравнивать значения одной колонки с несколькими возможными вариантами.
* Идеально подходит для стандартных случаев замены одних значений другими.

**Синтаксис:**

CASE expression

WHEN value1 THEN result1

WHEN value2 THEN result2

ELSE default\_result

END

**Элементы синтаксиса:**

* **expression**: проверяемое выражение (обычно это одна колонка или агрегат).
* **valueN**: возможные значения, с которыми сравнивается выражение.
* **resultN**: результат, возвращаемый, если выражение равно указанному значению.
* **default\_result**: результат, возвращаемый, если не было найдено совпадений.

**Практика:**Возвращаемся к таблице сотрудников (employees). Теперь мы хотим определить квалификацию сотрудников на основании их опыта работы (years\_of\_service). Применим простую форму оператора CASE:

SELECT employee\_id,

first\_name,

last\_name,

CASE years\_of\_service

WHEN 0 THEN 'Новичок'

WHEN BETWEEN 1 AND 3 THEN 'Начинающий специалист'

WHEN BETWEEN 4 AND 7 THEN 'Средний специалист'

ELSE 'Старший специалист'

END AS qualification

FROM employees;

**Что делает эта команда:**Команда просматривает столбец years\_of\_service и, в зависимости от количества лет службы, назначает соответствующую квалификационную категорию.

#### b) Сложная форма CASE

**Особенности:**

* Позволяет проводить проверку на основе сложных условий (булевых выражений, арифметических операций и т.д.).
* Позволяет комбинировать разнообразные условия и производить точные вычисления.

**Синтаксис:**

CASE

WHEN condition1 THEN result1

WHEN condition2 THEN result2

ELSE default\_result

END

**Элементы синтаксиса:**

* **conditionN**: любое булево выражение, которое может вернуть TRUE или FALSE.
* **resultN**: результат, возвращаемый, если указанное условие истинно.
* **default\_result**: результат, возвращаемый, если не выполнено ни одно из условий.

**Практика:**Теперь давайте усложним нашу задачу. Предположим, мы хотим оценить зарплату сотрудников относительно средней зарплаты по всей компании. Мы можем сравнить индивидуальные заработные платы с определенным пороговым значением, используя сложную форму оператора CASE:

SELECT employee\_id,

first\_name,

last\_name,

salary,

CASE

WHEN salary < 50000 THEN 'Зарплата ниже среднего уровня'

WHEN salary >= 50000 AND salary <= 100000 THEN 'Средняя зарплата'

ELSE 'Высокая зарплата'

END AS salary\_evaluation

FROM employees;

**Что делает эта команда:**Мы создаем новое поле (salary\_evaluation), где определяется оценка заработной платы сотрудника на основе его индивидуального дохода.

### **Отличительные особенности и различия**

Несмотря на сходство назначения, между операторами DECODE и двумя видами оператора CASE существует ряд ключевых различий:

* **Простота использования:**
  + DECODE ограничивается работой с простыми равенствами, что упрощает создание базовых запросов.
  + Простая форма CASE близка по удобству использования к DECODE, но обладает большей гибкостью благодаря поддержке BETWEEN и LIKE.
  + Сложная форма CASE предлагает полную свободу построения логики, но требует большего понимания SQL.
* **Производительность:**По производительности существенных различий нет, но простота запросов с DECODE делает их быстрее в разработке и чтении, особенно при малых объемах данных.
* **Универсальность:**Оператор CASE гораздо универсальнее и применим практически ко всем ситуациям, включая агрегаты и сложные запросы. Он идеально подходит для замены логики в приложениях и расширения функционала запросов.

### **Практические советы и рекомендации**

* Используйте оператор DECODE, если ваше приложение должно обрабатывать большое количество одинаковых простых запросов.
* Переходите к простой форме оператора CASE, если нужно добавить немного дополнительной логики.
* Перейдите к сложной форме оператора CASE, если требуется выполнение множества сложных условий.

### **10 каверзных вопросов с ответами**

1. В чём принципиальное отличие между оператором DECODE и оператором CASE?

*Ответ: Оператор DECODE способен проверять только точное равенство значений, тогда как оператор CASE позволяет использовать любые условия (включая арифметические выражения и сравнения по диапазонам).*

1. Возможно ли реализовать сложную логику с помощью оператора DECODE?

*Ответ: Нет, оператор DECODE предназначается только для точной замены значений, не поддерживая проверку условий.*

1. Могут ли использоваться подзапросы в качестве части условия в сложном варианте оператора CASE?

*Ответ: Да, подзапросы могут быть частью сложного условия в операторе CASE.*

1. Нужно ли указывать блок ELSE в операторах ветвления?

*Ответ: Блок ELSE не обязателен, но рекомендуется включать его для предотвращения ошибок, связанных с отсутствием обработанного случая.*

1. Влияют ли индексы на производительность операторов ветвления?

*Ответ: Индексация существенно влияет на скорость выполнения запросов, содержащих операторы ветвления, так как индексация ускоряет доступ к данным.*

1. Является ли порядок условий важным фактором в операторах ветвления?

*Ответ: Порядок условий важен, потому что первая встретившаяся истина останавливает дальнейшую проверку последующих условий.*

1. Есть ли ограничение на глубину вложения операторов ветвления?

*Ограничений на количество вложений операторов ветвления формально нет, но чрезмерно глубокие конструкции снижают производительность и ухудшают читаемость кода.*

1. Может ли оператор CASE возвращать сразу несколько значений?

*Ответ: Один экземпляр оператора CASE может вернуть только одно значение, но несколько экземпляров могут существовать параллельно в разных частях запроса.*

1. Какие дополнительные аргументы можно передавать в оператор DECODE?

*Ответ: В дополнение к основным аргументам, аргумент DEFAULT может быть использован для определения результата, если ни одно из предыдущих условий не сработало.*

1. Какой оператор удобнее использовать для быстрой замены небольшого набора значений?

*Ответ: Лучше выбрать оператор DECODE, так как он оптимизирован именно для такой задачи и его синтаксис более лаконичен.*

## **Иерархические запросы с помощью WITH RECURSIVE + UNION ALL**

Иерархические запросы используются для извлечения данных, связанных отношением типа «родитель-потомок». Классический пример такой структуры — организация сотрудников фирмы, файловая система компьютера или схема веб-каталога. Рассмотрим подробнее возможности PostgreSQL для решения таких задач.

### **Понятие рекурсивных запросов**

Одним из ключевых инструментов PostgreSQL для обработки иерархической информации являются рекурсивные запросы. Они позволяют пошагово проходить всю глубину дерева и собирать нужные сведения, следуя принципу «итерация за итерацией».

#### **Как работает рекурсия в SQL?**

Подход основан на механизме Common Table Expressions (CTE), специально адаптированном для рекурсивных операций. Конструкции выглядят следующим образом:

WITH RECURSIVE cte\_name AS (

-- базовый случай (начальное условие рекурсии)

initial\_query

UNION ALL

-- рекурсивный шаг (продвижение вниз по дереву)

recursive\_query

)

SELECT \* FROM cte\_name;

#### **Важность конструкции WITH RECURSIVE**

Конструктор WITH RECURSIVE предназначен именно для того, чтобы удобно решать проблемы, возникающие при работе с иерархиями. Его главная особенность заключается в способности последовательно повторять шаги рекурсии до исчерпания возможных переходов между уровнями дерева.

##### **Почему используется UNION ALL?**

Операция UNION ALL соединяет результаты начальной и последующих рекурсивных частей. Важно отметить, что:

* Без параметра ALL итоговый набор мог бы содержать дублирующиеся записи.
* Параметр ALL сохраняет порядок следования записей и предотвращает удаление повторений, обеспечивая правильное построение результата.

#### **Другие важные аспекты рекурсивных запросов**

1. **Определение базовых условий**: Первая часть рекурсивного запроса задаёт стартовую точку. Именно отсюда начинается движение по дереву. Если эта часть неверна, то дальнейшие вычисления тоже окажутся некорректными.
2. **Выбор условия перехода**: Вторая часть описывает правило продвижения по дереву (рекурсивный шаг). Здесь определяется логика нахождения дочерних элементов относительно текущих родителей.
3. **Предотвращение бесконечной рекурсии**: Следует внимательно следить за условиями остановки рекурсии. Для этого важно правильно определить критерий завершения процесса.
4. **Производительность**: Несмотря на мощь рекурсивных запросов, чрезмерно глубокое дерево или большое количество параллельных запросов могут замедлить выполнение. Рекомендуется оптимизировать индексы и учитывать специфику конкретной базы данных.

### **Примеры ситуаций, где полезны рекурсивные запросы**

Вот некоторые распространённые случаи использования рекурсивных запросов:

* Определение полного списка подчинённых конкретного менеджера.
* Подъём по файловой системе от файла к директории.
* Обработка категорий продуктов с неограниченной глубиной вложенности.
* Сбор статистики продаж с учётом региональных подразделений.

### **Типичные команды и операторы в рекурсивных выражениях**

1. **CTE (Common Table Expression)** — временная виртуальная таблица, созданная на этапе выполнения запроса.
2. **JOIN / LEFT JOIN** — объединение текущей итерации с результатом предыдущей для дальнейшего прохождения по уровню иерархии.
3. **WHERE** — фильтрация данных на каждом уровне рекурсии, необходимая для точного выбора нужных записей.
4. **ORDER BY** — сортировка результирующего набора, иногда критичная для понимания порядка вывода записей.
5. **LIMIT / OFFSET** — ограничение числа возвращаемых записей, полезно для больших деревьев.
6. **DISTINCT** — удаляет дублирующие записи, хотя чаще всего это делается автоматически с помощью правильного построения запроса.
7. **GROUP BY / HAVING** — агрегация данных на уровнях иерархии, полезна при анализе совокупных показателей (суммы, средние значения и др.).
8. **PARTITION BY** — разделение данных на сегменты согласно определённому ключу (полезно для оконных функций).

### **Оптимизация производительности рекурсивных запросов**

При обработке больших объёмов данных производительность становится ключевым фактором. Вот несколько рекомендаций:

* Создавайте индексы по столбцам, участвующим в связях между уровнями иерархии.
* Ограничивайте число проходов по дереву, задавая пределы глубины.
* Используйте агрегатные функции только там, где это действительно необходимо.
* Регулярно проверяйте наличие избыточных обращений к данным.

### **Плюсы и минусы рекурсивных запросов**

**Преимущества:**

* Простота реализации: одно выражение покрывает весь диапазон иерархического анализа.
* Высокая читаемость: конструкция понятна даже начинающим разработчикам.
* Гибкость: поддержка произвольной глубины дерева и множественных точек входа.

**Недостатки:**

* Потенциальная низкая производительность при плохо настроенном индексе или глубоком дереве.
* Сложность диагностики ошибок, особенно в многомерных структурах.
* Необходимость чёткого понимания механизма рекурсии для эффективного использования.

### **Реализация примера на практике**

Теперь рассмотрим конкретный пример, основанный на таблице сотрудников, показывающий применение рекурсивных запросов в реальной ситуации.

Допустим, имеется следующая таблица сотрудников с иерархическим уровнем управления:

CREATE TABLE employees (

id SERIAL PRIMARY KEY,

first\_name VARCHAR(50),

last\_name VARCHAR(50),

manager\_id INT REFERENCES employees(id)

);

Данные в таблице могут выглядеть примерно так:

|  |  |  |  |
| --- | --- | --- | --- |
| **id** | **first\_name** | **last\_name** | **manager\_id** |
| 1 | Петр | Иванов | NULL |
| 2 | Анна | Петрова | 1 |
| 3 | Сергей | Сидоров | 1 |
| 4 | Ольга | Кузнецова | 2 |
| 5 | Алексей | Васильев | 3 |

Задача: составить список всех подчинённых Петра Иванова (включая самого Ивана).

Реализуем её с помощью рекурсивного запроса:

WITH RECURSIVE employee\_tree AS (

-- Начало рекурсии: берем запись самого Петрова

SELECT id, first\_name || ' ' || last\_name AS full\_name, manager\_id

FROM employees

WHERE id = 1

UNION ALL

-- Рекурсивный шаг: ищем подчинённых найденных ранее менеджеров

SELECT e.id, e.first\_name || ' ' || e.last\_name AS full\_name, e.manager\_id

FROM employees e

JOIN employee\_tree et ON e.manager\_id = et.id

)

SELECT \* FROM employee\_tree;

Результат выполнения:

|  |  |  |
| --- | --- | --- |
| **id** | **full\_name** | **manager\_id** |
| 1 | Петр Иванов | NULL |
| 2 | Анна Петрова | 1 |
| 3 | Сергей Сидоров | 1 |
| 4 | Ольга Кузнецова | 2 |
| 5 | Алексей Васильев | 3 |

Этот запрос показывает полную картину организации сотрудников вокруг Петра Иванова.

### **Итоги**

Иерархические запросы с применением конструкций WITH RECURSIVE предоставляют мощный инструмент для работы с древесными структурами данных. Их преимущества очевидны при создании аналитических отчётов, управлении персоналом и проектировании информационных систем, где важна прозрачность связей между элементами данных.

## **Оконные/аналитические функции. "LEAD()", "LAG()", "FIRST\_VALUES()", "LAST\_VALUE()", "RANK()", "DENSE\_RANK()", "ROW\_NUMBER()"**

Аналитические функции предназначены для анализа оконных данных — позволяют обращаться к ряду записей рядом с текущей записью и создавать ранги, последовательности номеров и прочее.

### **Самые популярные аналитические функции:**

* **LEAD()**: Забегает вперед и смотрит следующую запись.Пример:
* SELECT employee\_id, salary, LEAD(salary) OVER (ORDER BY employee\_id) AS next\_salary

FROM employees;

* **LAG()**: Смотрит назад на предыдущую запись.Пример:
* SELECT employee\_id, salary, LAG(salary) OVER (ORDER BY employee\_id) AS prev\_salary

FROM employees;

* **FIRST\_VALUE()**: Возвращает первое значение окна.Пример:
* SELECT employee\_id, salary, FIRST\_VALUE(salary) OVER (PARTITION BY department ORDER BY salary DESC) AS top\_salary\_in\_dept

FROM employees;

* **LAST\_VALUE()**: Возвращает последнее значение окна.
* **ROW\_NUMBER()**: Присваивает уникальный номер каждой строке в окне.
* **RANK()**: Назначает ранг строкам с пропусками (при равных значениях пропадает ранг следующего ряда).
* **DENSE\_RANK()**: Ранг без пропуска позиций (каждый последующий ранг отличается на единицу).

Аналитические функции в PostgreSQL помогают производить расчёты над группами данных и предоставляют возможность анализировать ряды данных последовательно. Они называются оконными (window) функциями, потому что работают с окнами данных, сформированными по указанным правилам группировки.

Вот подробное описание каждой аналитической функции:

Современные СУБД предлагают богатый инструментарий для аналитики и обработки данных, который позволяет делать сложные расчеты и группировки буквально одной командой. Особую роль в арсенале аналитиков занимают **оконные функции** (*window functions*). Они позволяют выполнять операции над группами строк (так называемыми окнами), рассматривая строки в определенном порядке или диапазоне.

Ниже представлен глубокий разбор каждой оконной функции, примеры на основе таблицы сотрудников (employees), подробное объяснение того, что делает каждая команда, а также рассмотрены альтернативы и отличия в PostgreSQL.

### **ТАБЛИЦА СОТРУДНИКОВ (EMPLOYEES)**

Примем следующую структуру таблицы:

### ОБЩИЕ ОСОБЕННОСТИ ОКОННЫХ ФУНКЦИЙ

* **OVER():** Все оконные функции используют предложение OVER, чтобы указать окно (группу строк), в пределах которого действует функция.
* **PARTITION BY:** Задает разделение окна на группы (например, по департаментам).
* **ORDER BY:** Определяет порядок обработки строк в окне.
* **ROWS/RANGE:** Определяет размер окна (для функций, работающих с диапазонами).

### 

#### LEAD() — смотрим вперед

Позволяет посмотреть на значение следующей строки (или N-й последующей строки) в пределах окна.

**СИНТАКСИС:**

LEAD(column, offset, default\_value) OVER (...)

**ЗНАЧЕНИЯ:**

* column: Столбец, значение которого будет получено.
* offset: Смещение (количество строк вперёд).
* default\_value: Значение, если впереди нет нужной строки.

**ЗАДАЧА:** Узнать оклад следующего сотрудника в своем отделе.

SELECT employee\_id, department\_id, salary,

LEAD(salary, 1) OVER (PARTITION BY department\_id ORDER BY hire\_date) AS next\_salary

FROM employees;

**ПОЯСНЕНИЕ:**

* Мы разбиваем сотрудников по департаментам (PARTITION BY department\_id).
* Сортируем по дате приема на работу (ORDER BY hire\_date).
* Функция LEAD() смотрит на следующую строку, показывая оклад следующего сотрудника в своём подразделении.

#### LAG() — смотрим назад

То же самое, что и LEAD(), но обратно — смотреть на предыдущее значение.

**ЗАДАЧА:** Найти разницу в зарплате между сотрудником и его предшественником в отделе.

SELECT employee\_id, department\_id, salary,

salary - LAG(salary, 1) OVER (PARTITION BY department\_id ORDER BY hire\_date) AS salary\_diff

FROM employees;

**ПОЯСНЕНИЕ:**

* Функцией LAG() мы берем предыдущую зарплату сотрудника из своего департамента.
* Вычтем её из текущей зарплаты, получив прирост.

#### FIRST\_VALUE() — первое значение в окне

Вернуть первое значение окна, соответствующего вашему запросу.

**ЗАДАЧА:** Показать первую зарплату в вашем департаменте.

SELECT employee\_id, department\_id, salary,

FIRST\_VALUE(salary) OVER (PARTITION BY department\_id ORDER BY hire\_date) AS first\_dept\_salary

FROM employees;

**ПОЯСНЕНИЕ:**

* Группируем сотрудников по департаментам (PARTITION BY department\_id).
* Первым значением будет самая ранняя зарплата в каждом департаменте.

#### LAST\_VALUE() — последнее значение в окне

Показывает последнее значение окна.

**ЗАДАЧА:** Определить последнюю зарплату вашего департамента.

SELECT employee\_id, department\_id, salary,

LAST\_VALUE(salary) OVER (PARTITION BY department\_id ORDER BY hire\_date ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS last\_dept\_salary

FROM employees;

**ПОЯСНЕНИЕ:**

* Используя ROWS BETWEEN, охватываем всё окно целиком.
* Берём последнее значение в каждом департаменте.

#### RANK() — рейтинг с пропусками рангов

Присваивает уникальный ранг каждой строке, пропуская порядковые номера, если есть одинаковые значения.

**ЗАДАЧА:** Назначить рейтинги сотрудникам по зарплатам в департаменте.

SELECT employee\_id, department\_id, salary,

RANK() OVER (PARTITION BY department\_id ORDER BY salary DESC) AS rank\_in\_department

FROM employees;

**ПОЯСНЕНИЕ:**

* Сотрудник с наибольшей зарплатой получит ранг 1.
* Следующий сотрудник, имеющий меньшую зарплату, получит ранг 2.
* Но если двое сотрудников получают одинаковую зарплату, следующий сотрудник после них получит ранг, превышающий пропущенный.

#### DENSE\_RANK() — рейтинг без пропуска рангов

Похоже на RANK(), но присваивает соседствующие номера подряд, без пропусков.

**ЗАДАЧА:** Назначить непрерывные рейтинги сотрудникам по зарплатам в департаменте.

SELECT employee\_id, department\_id, salary,

DENSE\_RANK() OVER (PARTITION BY department\_id ORDER BY salary DESC) AS dense\_rank\_in\_department

FROM employees;

**ПОЯСНЕНИЕ:**

* Сотрудники с одинаковой зарплатой получат один и тот же ранг.
* После них идет сотрудник с номером ранга подряд, без пропусков.

#### ROW\_NUMBER() — уникальный номер строки

Присваивает уникальным образом строки номер по порядку.

**ЗАДАЧА:** Присвоить уникальные номера сотрудникам в порядке поступления на работу.

SELECT employee\_id, department\_id, hire\_date,

ROW\_NUMBER() OVER (ORDER BY hire\_date) AS row\_number

FROM employees;

**ПОЯСНЕНИЕ:**

* Нумеруются строки последовательно, независимо от департаментов.
* Первая строка (по дате приёма) получит номер 1, вторая — 2 и так далее.

### **10 КАВЕРЗНЫХ ВОПРОСОВ С ОТВЕТАМИ**

1. **Вопрос:** Что выдаст функция LEAD() при отсутствии смещения?

***Ответ:*** *Вернется значение следующей строки в данном окне.*

1. **Вопрос:** Будет ли результат функции LAG() отличаться от LEAD() при отрицательном смещении?

***Ответ:*** *Нет, функции взаимозаменяемы при изменении знака смещения.*

1. **Вопрос:** Чем отличается RANK() от DENSE\_RANK()?

***Ответ:*** *RANK() пропускает номера при появлении одинаковых значений, а DENSE\_RANK() продолжает выдавать последовательные номера.*

1. **Вопрос:** Для чего нужна фраза ROWS BETWEEN?

***Ответ:*** *Чтобы указать границы окна, в пределах которого рассчитываются значения оконных функций.*

1. **Вопрос:** Может ли LAST\_VALUE() показывать неправильное значение, если забыть указать границы окна?

***Ответ:*** *Да, если не указана граница окна, функция покажет некорректное значение.*

1. **Вопрос:** В чем особенность функции ROW\_NUMBER()?

***Ответ:*** *Она гарантированно выдает уникальные номера строк, даже если значения ключей совпадают.*

1. **Вопрос:** Можно ли сгруппировать результаты оконных функций?

***Ответ:*** *Да, для этого используется предложение PARTITION BY.*

1. **Вопрос:** Зачем использовать предложение ORDER BY в оконных функциях?

***Ответ:*** *Оно упорядочивает строки в окне, влияя на порядок расчета оконных функций.*

1. **Вопрос:** Какие преимущества у оконных функций перед обычными агрегатами?

***Ответ:*** *Окна позволяют выполнять расчет над отдельными группами строк, сохранять оригинальную структуру данных и выполнять сложные аналитические задачи.*

1. **Вопрос:** Почему функция FIRST\_VALUE() полезна?

***Ответ:*** *Она позволяет быстро получить первое значение из окна, что бывает крайне востребовано в аналитических задачах.*

### **ЧТО ЕЩЁ МОЖНО СДЕЛАТЬ В POSTGRESQL?**

* **Альтернативы оконным функциям:** Многие аналитические задачи можно решить с помощью традиционных агрегатов и JOIN'ов, но оконные функции делают решение намного короче и выразительнее.
* **Другие полезные функции:** Помимо рассмотренных, PostgreSQL предлагает массу других полезных оконных функций, таких как NTILE(), PERCENT\_RANK(), CUME\_DIST() и другие.

## **Использование оконных функций для расчёта накопительной суммы, скользящего среднего, центрированного среднего. Конструкции вида "SUM (\*) / AVG (\*) / MIN(\*) / MAX(\*) OVER (PARTITION BY \* ORDER BY)". Синтаксис. Разница работы агрегационных функций с выражением "order by" и без**

## **Подробный разбор оконных функций для расчетов накопительной суммы, скользящего среднего и центрированного среднего**

Оконные функции (*Window Functions*) в SQL позволяют выполнять различные вычисления, применимые к определенной группе строк (окну), выбранных из основной таблицы. Это дает огромные возможности для аналитических задач, таких как расчет накопительных сумм, средних значений и максимальных/максимальных показателей в группах.

### **Общий синтаксис оконных функций**

FUNCTION\_NAME([col]) OVER ([PARTITION BY col1[, col2,...]]

[ORDER BY col3 [ASC/DESC]]

[[ROWS|RANGE] {BETWEEN boundary\_start AND boundary\_end}

| {CURRENT ROW}] )

* **FUNCTION\_NAME**: Имя агрегирующей функции (например, SUM, AVG, MIN, MAX).
* **PARTITION BY**: Определяет группу строк, внутри которой будут производиться вычисления.
* **ORDER BY**: Упорядочивает строки внутри окна.
* **ROWS/RANGE**: Задает рамки окна, относительно текущей строки (например, текущая строка плюс предыдущие X строк).

### **Основные сценарии использования оконных функций**

1. **Накопительная сумма (Running Total)** — сумма нарастающим итогом.
2. **Скользящее среднее (Rolling Average)** — усредненное значение на основе текущих и предыдущих строк.
3. **Центрированное среднее (Centered Moving Average)** — среднее значение вокруг текущей строки.

### **НАКОПИТЕЛЬНАЯ СУММА (RUNNING TOTAL)**

#### Пример: Нахождение общей накопительной суммы зарплат по департаментам

Пусть у нас есть таблица сотрудников (employees) с полями:

|  |  |  |
| --- | --- | --- |
| **employee\_id** | **department\_id** | **salary** |
| 1 | 1 | 50000 |
| 2 | 1 | 60000 |
| 3 | 2 | 55000 |
| 4 | 2 | 70000 |
| 5 | 1 | 65000 |

Нам нужно рассчитать сумму зарплат сотрудников по департаментам, накопительно.

SELECT employee\_id, department\_id, salary,

SUM(salary) OVER (PARTITION BY department\_id ORDER BY employee\_id) AS running\_total

FROM employees;

**⭐ Что делает каждая команда:**

* **SUM(salary)**: Агрегатная функция, считающая сумму.
* **PARTITION BY department\_id**: Разделение по департаментам, чтобы считать суммы раздельно для каждого департамента.
* **ORDER BY employee\_id**: Определение порядка строк в расчете, суммируя зарплаты последовательно.

### **СКОЛЬЗЯЩЕЕ СРЕДНЕЕ (ROLLING AVERAGE)**

#### Пример: Скользящий средний заработок сотрудников по месяцам

Дополнительно предположим, что у нас есть таблица с ежемесячным доходом сотрудников:

|  |  |  |
| --- | --- | --- |
| **month** | **employee\_id** | **salary** |
| Jan | 1 | 50000 |
| Feb | 1 | 55000 |
| Mar | 1 | 60000 |
| Apr | 1 | 65000 |

Рассчитаем скользящее среднее заработка за последние 3 месяца для каждого сотрудника.

SELECT month, employee\_id, salary,

AVG(salary) OVER (PARTITION BY employee\_id ORDER BY month

ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS rolling\_avg

FROM salaries;

**Что делает каждая команда:**

* **AVG(salary)**: Средняя величина.
* **PARTITION BY employee\_id**: Отделяем данные по сотрудникам.
* **ORDER BY month**: Порядок по месяцам.
* **ROWS BETWEEN 2 PRECEDING AND CURRENT ROW**: Рамки окна включают текущую строку и 2 предшествующих ей строки.

### **ЦЕНТРИРОВАННОЕ СРЕДНЕЕ (CENTERED MOVING AVERAGE)**

#### Пример: Центрированный средний заработок за последние 3 месяца

Для нахождения центрационного среднего нам понадобится учитывать среднюю величину около текущей строки (текущую и соседей слева и справа).

SELECT month, employee\_id, salary,

AVG(salary) OVER (PARTITION BY employee\_id ORDER BY month

ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS centered\_avg

FROM salaries;

**Что делает каждая команда:**

* **ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING**: Окно охватывает текущую строку и соседние строки слева и справа.

### **Работа агрегационных функций с предложением ORDER BY**

Предложение ORDER BY играет ключевую роль в поведении оконных функций:

* Без ORDER BY: Окно рассматривается глобально (все строки считаются вместе).
* С ORDER BY: Строки упорядочиваются, и агрегация происходит в установленном порядке.

Например, возьмем команду:

SUM(salary) OVER ()

Она вернет одну общую сумму по всем строкам.

Но:

SUM(salary) OVER (ORDER BY employee\_id)

Уже вернет нарастающий итог (running total) по мере роста employee\_id.

### **10 каверзных вопросов с ответами**

1. **Вопрос:** Чем отличается накопительная сумма от обычной суммы?

***Ответ:*** *Накопительная сумма учитывает все предыдущие строки вплоть до текущей, а обычная сумма берет только отдельные группы данных.*

1. **Вопрос:** Зачем нужна рамка окна (ROWS BETWEEN)?

***Ответ:*** *Рамка окна определяет, какие строки входят в область вычисления функции (например, только текущая строка или диапазон соседних строк).*

1. **Вопрос:** Что получится, если убрать PARTITION BY?

***Ответ:*** *Тогда агрегация будет выполнена по всей таблице, без разделения на группы.*

1. **Вопрос:** Может ли ORDER BY повлиять на результат?

***Ответ:*** *Да, ORDER BY меняет порядок строк, следовательно, повлияет на результат вычислений, особенно для накопительных сумм и скользящих средних.*

1. **Вопрос:** В чем разница между окном по строкам (ROWS) и интервалу (RANGE)?

***Ответ:*** *ROWS считает количество строк, а RANGE учитывает интервал значений (может давать непредсказуемые результаты с числами).*

1. **Вопрос:** Можно ли найти максимум зарплаты за квартал с помощью оконных функций?

***Ответ:*** *Да, с помощью функции MAX() и правильной настройки границ окна.*

1. **Вопрос:** Для чего нужна функция LAG()?

***Ответ:*** *Функция LAG() позволяет получить значение предыдущей строки, что полезно для вычисления изменений.*

1. **Вопрос:** Как работает рамка окна "BOTH"?

***Ответ:*** *Режим "BOTH" захватывает строки как до, так и после текущей строки, симметрично располагая окно.*

1. **Вопрос:** Какой смысл у окна с параметром "CURRENT ROW"?

***Ответ:*** *Окно "CURRENT ROW" рассматривает только текущую строку, не включая никаких соседних строк.*

1. **Вопрос:** Почему окно ROWS BETWEEN лучше подойдет для скользящей средней, чем RANGE BETWEEN?

***Ответ:*** *Потому что режим ROWS контролирует четкую границу строк, в то время как RANGE может интерпретировать значения неоднозначно.*

### **Другие аналогичные команды в PostgreSQL**

В PostgreSQL аналогичны базовые оконные функции, такие как SUM, AVG, COUNT, LAG, LEAD. Отличием является возможность использования специфических функций, например:

* **CUME\_DIST()**: Доля текущих значений относительно всех значений в окне.
* **PERCENT\_RANK()**: Процентный ранг текущего значения.
* **NTILE(n)**: Распределение строк на n сегментов.

## **Расширенные опции group by "GROUPING SETS", "CUBE" и "ROLLUP"**

В классическом SQL-запросе с использованием GROUP BY данные собираются в группы по единственному набору критериев. Однако в ряде задач возникает необходимость формировать несколько группировок за один запрос, чтобы обеспечить множественную комбинаторику данных. В таком случае выручают расширенные опции группировки: **GROUPING SETS**, **CUBE** и **ROLLUP**.

### **Группа данных с помощью GROUP BY: классический подход**

Классический запрос с использованием GROUP BY собирает данные по единственной комбинации группировочных полей. Например, мы можем подсчитать сумму зарплат сотрудников по отделам:

SELECT department\_id, SUM(salary)

FROM employees

GROUP BY department\_id;

Здесь данные группируются по одному признаку (department\_id), и итоговая таблица будет содержать сумму зарплат по каждому отделу.

### **Расширенные опции группировки: зачем они нужны?**

Во многих случаях недостаточно группировать данные по одному признаку. Требуется выполнить несколько различных группировок за один запрос, получить сводные данные различного уровня детализации. Именно тут приходят на помощь продвинутые опции:

* **GROUPING SETS**: Позволяют задать произвольные наборы группировок.
* **CUBE**: Создает все возможные комбинации группировок.
* **ROLLUP**: Генерирует иерархическую группировку с постепенной деградацией уровня детализации.

Эти опции значительно сокращают количество необходимых запросов и повышают продуктивность анализа данных.

### **Пример таблицы EMPLOYEES**

Для примеров используем гипотетическую таблицу сотрудников (employees):

|  |  |  |  |
| --- | --- | --- | --- |
| **employee\_id** | **department\_id** | **region\_id** | **salary** |
| 1 | 1 | 1 | 50000 |
| 2 | 1 | 1 | 60000 |
| 3 | 2 | 2 | 55000 |
| 4 | 2 | 2 | 70000 |
| 5 | 1 | 1 | 65000 |

### **Расширенные опции группировки**

#### GROUPING SETS

Позволяет явно задать требуемые комбинации группировок. Команда принимает набор наборов, для каждого из которых создается отдельная группа.

**СИНТАКСИС:**

GROUP BY GROUPING SETS ((group\_col1, group\_col2), (group\_col3))

**ЗАДАЧА:** Получить суммарную зарплату по регионам и отделам, а также суммарную зарплату по каждому региону отдельно.

SELECT department\_id, region\_id, SUM(salary)

FROM employees

GROUP BY GROUPING SETS((department\_id, region\_id), (region\_id));

**РЕЗУЛЬТАТ:**

|  |  |  |
| --- | --- | --- |
| **department\_id** | **region\_id** | **sum\_salary** |
| 1 | 1 | 175000 |
| 2 | 2 | 125000 |
| NULL | 1 | 175000 |
| NULL | 2 | 125000 |

**ПОЯСНЕНИЕ:**

* Первый набор группировок (department\_id, region\_id) создает стандартные группы по отделам и регионам.
* Второй набор (region\_id) создает дополнительную группу только по регионам.

#### CUBE

Генерирует **ВСЕ ВОЗМОЖНЫЕ КОМБИНАЦИИ** группировок. Это означает, что для каждого возможного сочетания признаков формируются группы.

**СИНТАКСИС:**

GROUP BY CUBE(group\_col1, group\_col2, ...)

**ЗАДАЧА:** Получить все возможные комбинации суммарных зарплат по отделам и регионам.

SELECT department\_id, region\_id, SUM(salary)

FROM employees

GROUP BY CUBE(department\_id, region\_id);

**РЕЗУЛЬТАТ:**

|  |  |  |
| --- | --- | --- |
| **department\_id** | **region\_id** | **sum\_salary** |
| 1 | 1 | 175000 |
| 2 | 2 | 125000 |
| 1 | NULL | 175000 |
| 2 | NULL | 125000 |
| NULL | 1 | 175000 |
| NULL | 2 | 125000 |
| NULL | NULL | 300000 |

**ПОЯСНЕНИЕ:**

* Каждая комбинация отдел+регион генерируется отдельно.
* Дополнительно создаются частичные группы только по отделам, только по регионам и общая сумма по всем сотрудникам.

#### ROLLUP

Создает **ИЕРАРХИЧЕСКУЮ ГРУППИРОВКУ**, начинающуюся с полной детализации и постепенно снижающей степень детализации.

**СИНТАКСИС:**

GROUP BY ROLLUP(group\_col1, group\_col2, ...)

**ЗАДАЧА:** Получить сумму зарплат сначала по отделам и регионам, затем по отделам, затем общую сумму.

SELECT department\_id, region\_id, SUM(salary)

FROM employees

GROUP BY ROLLUP(department\_id, region\_id);

**РЕЗУЛЬТАТ:**

|  |  |  |
| --- | --- | --- |
| **department\_id** | **region\_id** | **sum\_salary** |
| 1 | 1 | 175000 |
| 2 | 2 | 125000 |
| 1 | NULL | 175000 |
| 2 | NULL | 125000 |
| NULL | NULL | 300000 |

**ПОЯСНЕНИЕ:**

* Вначале формируется полная детализация (отдел+регион).
* Далее подгруппы складываются по отделам.
* Завершается общей суммой по всем сотрудникам.

### **KAVERSHIE VOPROSY I OTVETI**

1. **Вопрос:** Чем отличается GROUPING SETS от классического GROUP BY?

***Ответ:*** *Классический GROUP BY создает группы по единой комбинации полей, тогда как GROUPING SETS позволяет задать несколько независимых наборов группировок.*

1. **Вопрос:** Что выведет запрос с CUBE(department\_id)?

***Ответ:*** *Запрос создаст группы по отделам и отдельную общую группу (NULL).*

1. **Вопрос:** Чем отличается ROLLUP от CUBE?

***Ответ:*** *ROLLUP создает группы по убывающей степени детализации (от самой детализированной до общей), тогда как CUBE создает ВСЕ возможные комбинации группировок.*

1. **Вопрос:** Как определить, что данная строка принадлежит итоговой группе?

***Ответ:*** *Итоговые строки содержат NULL в полях группирования.*

1. **Вопрос:** Какую операцию выполняет GROUPING SETS((A),(B))?

***Ответ:*** *Формирует две независимые группы: одну по полю A, вторую по полю B.*

1. **Вопрос:** Как отличить обычную группу от итоговой строки?

***Ответ:*** *Итоговые строки выделяются наличием NULL в полях группирования.*

1. **Вопрос:** Чем полезен оператор ROLLUP?

***Ответ:*** *Позволяет получить иерархические сводные данные с постепенным понижением уровня детализации.*

1. **Вопрос:** В чем отличие результата GROUP BY department\_id от GROUP BY ROLLUP(department\_id)?

***Ответ:*** *GROUP BY department\_id дает обычные группы по отделам, а ROLLUP дополнительно создает итоговую строку с общей суммой.*

1. **Вопрос:** Какая комбинация ключей будет создана в результате CUBE(A,B,C)?

***Ответ:*** *Будут созданы все возможные комбинации группировок: (A,B,C), (A,B), (A,C), (B,C), (A), (B), (C) и общая группа.*

1. **Вопрос:** Какая специфика использования оператора ROLLUP?

***Ответ:*** *Он формирует иерархические данные, выстраивая группы по уменьшению уровня детализации: сперва группы по полному набору признаков, затем по отдельным признакам, затем итоговая строка.*

### **PostgeSQL-аналоги и особенности**

PostgreSQL полностью поддерживает указанные выше операторы: GROUPING SETS, CUBE и ROLLUP. Они ведут себя одинаково, как и в других стандартах SQL. Основная ценность таких функций — сокращение числа запросов и повышение скорости анализа данных, что актуально для крупных проектов и BI-решений.

## **Функция CROSSTAB().  Принцип работы, синтаксис и основные элементы**

## **Теория работы функции CROSSTAB()**

### **Определение и назначение**

CROSSTAB() — специализированная функция в PostgreSQL, созданная для трансформации обычной двумерной таблицы (нормализованной формы хранения данных) в специальную трехмерную форму, называемую *перекрестной таблицей*. Этот термин появился благодаря принципу организации данных, когда некоторые поля используются для формирования осей X и Y, а остальные служат значением в ячейках.

Использование функции CROSSTAB() целесообразно в ситуациях, когда:

* Требуется визуально разделить данные на две пересекающиеся оси.
* Есть необходимость быстро увидеть распределение данных по нескольким измерениям.
* Облегчается задача принятия управленческих решений и ведения аналитической деятельности.

### **Концептуальная основа**

Основная идея заключается в следующем: исходные данные хранятся в стандартном табличном формате (строки и столбцы). Функция берёт исходные данные и перестраивает их так, чтобы конкретные характеристики (получаемые по одному полю) стали самостоятельными столбцами итоговой таблицы, а другое поле служило меткой строки.

Рассмотрим конкретный пример на базе нашей таблицы сотрудников (employees):

Пусть у нас есть сотрудники с указанием должности, отдела и города проживания. Мы хотим получить статистику по количеству сотрудников в каждом отделе отдельно по каждому городу.

Исходная таблица сотрудников:

|  |  |  |  |
| --- | --- | --- | --- |
| **EmployeeID** | **Name** | **Department** | **City** |
| 1 | Иванова | IT | Москва |
| 2 | Петрова | Sales | Петербург |
| 3 | Сергеев | HR | Самара |
| 4 | Александров | IT | Петербург |
| 5 | Павлов | Sales | Москва |

Обычная выборка в SQL выдаст нечто подобное:

SELECT Department, City, COUNT(\*) AS CountEmployees

FROM Employees

GROUP BY Department, City;

Результат обычного запроса:

|  |  |  |
| --- | --- | --- |
| **Department** | **City** | **CountEmployees** |
| IT | Москва | 1 |
| IT | Петербург | 1 |
| Sales | Москва | 1 |
| Sales | Петербург | 1 |
| HR | Самара | 1 |

Такое представление сложно анализировать и сравнивать цифры непосредственно. Здесь на помощь приходит функция CROSSTAB(), способная превратить этот результат в удобный формат:

| Department | Москва | Петербург | Самара |

|------------|--------|-----------|--------|

| IT | 1 | 1 | 0 |

| Sales | 1 | 1 | 0 |

| HR | 0 | 0 | 1 |

Такой формат значительно облегчает понимание распределения сотрудников по городам.

### **Внутренняя логика работы функции**

Функция CROSSTAB() получает два ключевых компонента:

1. **Основной запрос (Main Query)**. Обычно это сложный запрос с группировками и условиями фильтрации, результатом которого являются три важные части:
   * Первая колонка (которая станет ключом строки итоговой таблицы).
   * Вторая колонка (категория, которая превратится в имя столбца итоговой таблицы).
   * Третья колонка (числовое значение, которое попадёт в ячейку пересечения первой и второй колонок).
2. **Категории (Categories Query)**. Запрос, возвращающий список уникальных категорий (тех самых будущих столбцов итоговой таблицы).

Принцип прост: в итоговой таблице строками будут служить уникальные значения первой колонки основного запроса, а столбцами — уникальные значения второго запроса. Пересечение строк и столбцов формирует ячейки итоговой таблицы, куда попадают значения третьей колонки основного запроса.

### **Важные моменты и нюансы**

1. **Тип данных итоговой таблицы**. При создании описания итоговой таблицы необходимо заранее определять типы данных, которые будут возвращаться функцией. Типы данных столбцов итоговой таблицы определяются структурой данных исходных запросов.
2. **Порядок следования данных**. Порядок, в котором возвращаются данные основным запросом и запросом на получение категорий, имеет огромное значение. Изменение порядка приведет к изменению положения данных в итоговой таблице.
3. **Поддерживаемые версии PostgreSQL**. Функция доступна начиная с версий PostgreSQL 8.4 и выше. Более ранние версии не поддерживают данную возможность.
4. **Производительность**. Чем больше объем исходных данных, тем медленнее работают запросы с применением CROSSTAB(). Поэтому важно оптимизировать индексы и избегать ненужных операций над данными.
5. **Агрегирование данных**. Часто в основной запрос включаются операции агрегирования, такие как SUM, AVG, MIN, MAX, COUNT и др., чтобы получать итоговые значения по определенным критериям.
6. **Отсутствие значений**. Если какой-то регион или сотрудник отсутствуют в исходных данных, соответствующая ячейка в итоговой таблице останется пустой (NULL). Эту проблему можно решить с помощью конструктора COALESCE.
7. **Расширяемость**. Хотя базовая версия CROSSTAB() поддерживает всего три колонки (ключ строки, категория и значение), существует расширение pg\_crosstab, которое позволяет расширить возможности функции до большего числа колонок.

### **Параллели с другими инструментами и решениями**

Помимо функции CROSSTAB(), в PostgreSQL есть несколько способов достижения схожего эффекта:

* **Pivot Table Extension**: Расширение, предоставляющее инструменты для поворота данных, аналогичные тем, что предлагает функция CROSSTAB(). Оно доступно в отдельных дистрибутивах PostgreSQL и позволяет динамически создавать pivot-таблицы.
* **CASE Expressions**: Использование конструкций CASE WHEN в сочетании с агрегатными функциями также способно создавать аналоги перекрестных таблиц, хотя реализация сложнее и менее интуитивна.
* **CTE (Common Table Expression)**: Иногда применяется для предварительной обработки данных перед передачей их в CROSSTAB(), обеспечивая более детальную настройку процесса.

## **Группирование нескольких строк в одну с помощью оператора "STRING\_AGG()". Принцип работы, синтаксис и основные элементы**

## **Постгрес: оператор STRING\_AGG()**

Одним из мощнейших инструментов PostgreSQL для работы с группами данных является оператор STRING\_AGG(). Данный оператор предназначен для объединения строк из одного или нескольких столбцов в единую группу. Он особенно полезен, когда необходимо вывести объединённые значения ряда связанных записей в виде единой строки.

Эта операция часто используется для подготовки данных для аналитических отчётов, экспорта данных или создания динамического контента. Рассмотрим детально, как работает данная функция, её синтаксис и возможные варианты применения.

### **Понятие группировки строк**

Часто возникают ситуации, когда группа данных должна быть представлена единым целым, несмотря на различие конкретных значений. Например, представьте себе таблицу сотрудников, где у каждого сотрудника указана должность, отдел и адрес электронной почты. Задача состоит в том, чтобы вывести всю электронную почту работников конкретного отдела одним списком.

Оператор STRING\_AGG() идеально справляется с подобной задачей, собирая все электронные адреса сотрудников одного отдела в единый блок.

### **Работа оператора STRING\_AGG()**

Оператор STRING\_AGG() объединяет значения строк, относящихся к группе, задаваемой условием, и помещает их в одно значение строки. Его синтаксис следующий:

STRING\_AGG(expression, delimiter)

или с добавлением условия упорядочивания:

STRING\_AGG(expression, delimiter ORDER BY sort\_expression)

Здесь:

* **expression** — выражение, которое представляет собой строку, подлежащую конкатенации.
* **delimiter** — символ-разделитель, используемый для разделения элементов в итоговой строке.
* **sort\_expression** — необязательное условие сортировки, если требуется упорядочить объединение.

Этот оператор действует как агрегатная функция и применяется вместе с командой GROUP BY.

### **Пример использования**

Представим таблицу сотрудников (employees):

|  |  |  |  |
| --- | --- | --- | --- |
| **employee\_id** | **full\_name** | **email** | **department** |
| 1 | Алексей | alex@company.ru | IT |
| 2 | Мария | maria@company.ru | Marketing |
| 3 | Сергей | sergey@company.ru | IT |
| 4 | Екатерина | kate@company.ru | Finance |
| 5 | Дмитрий | dima@company.ru | Marketing |

Нам нужно собрать электронные письма всех сотрудников, работающих в одном отделе, в единую строку. Используем команду:

SELECT department, STRING\_AGG(email, ', ')

FROM employees

GROUP BY department;

Результат выполнения:

|  |  |
| --- | --- |
| **department** | **emails** |
| IT | alex@company.ru, sergey@company.ru |
| Marketing | maria@company.ru, dima@company.ru |
| Finance | kate@company.ru |

Мы видим, что для каждого отдела все почтовые адреса собраны в единственную строку, разделённую запятыми.

### **Дополнительные опции**

* **Упорядочивание** (ORDER BY):Если требуется задать определенный порядок объединения, используйте параметр ORDER BY. Например, отсортируем адреса сотрудников по алфавиту:
* SELECT department, STRING\_AGG(email, ', ' ORDER BY email ASC)
* FROM employees

GROUP BY department;

* **Применение условий**:Команда WHERE может применяться совместно с оператором STRING\_AGG() для отбора нужных строк. Допустим, мы хотим включить только тех сотрудников, чьи имена начинаются с буквы 'A':
* SELECT department, STRING\_AGG(email, ', ')
* FROM employees
* WHERE full\_name LIKE 'A%'

GROUP BY department;

### **Практическое применение**

Вот несколько примеров реального использования оператора STRING\_AGG():

1. Объединение телефонных номеров клиентов в одной записи:
2. SELECT customer\_id, STRING\_AGG(phone\_number, '; ') AS phones
3. FROM customers\_phones

GROUP BY customer\_id;

1. Создание списка имен студентов на одном факультете:
2. SELECT faculty, STRING\_AGG(student\_name, ', ')
3. FROM students

GROUP BY faculty;

1. Получение полного списка товаров заказа:
2. SELECT order\_id, STRING\_AGG(product\_name, ', ')
3. FROM orders\_products

GROUP BY order\_id;

### **Аналогичные операторы в PostgreSQL**

Помимо STRING\_AGG(), в PostgreSQL есть и другие способы объединения строк:

* **ARRAY\_AGG()**: Собирает данные в массив, что полезно для дальнейшей обработки:

ARRAY\_AGG(value)

* **JSON\_AGG()**: Формирует JSON-массив из значений:

JSON\_AGG(value)

* **XML\_AGG()**: Создает XML-документ из записей:

XML\_AGG(XMLCONCAT('<item>', value, '</item>'))

Каждый из этих операторов ориентирован на различные сценарии и потребности пользователей.

### **Производительность и ограничения**

Несмотря на свою полезность, оператор STRING\_AGG() обладает некоторыми особенностями, влияющими на производительность:

* Объединение большого объёма данных требует значительных ресурсов памяти и процессора.
* Следует осторожно относиться к использованию сложных выражений внутри STRING\_AGG(), так как это увеличивает нагрузку на сервер.
* Большие наборы данных могут приводить к снижению скорости выполнения запросов.

При необходимости повышения производительности можно рассмотреть индексирование и разбиение крупных наборов данных на меньшие порции.

### **10 каверзных вопросов и ответов**

1. **Вопрос:** Является ли STRING\_AGG() функцией или оператором?
   * **Ответ:** Это агрегатная функция PostgreSQL, применяемая для объединения строк.
2. **Вопрос:** Обязательно ли использовать GROUP BY с оператором STRING\_AGG()?
   * **Ответ:** Да, оператор всегда используется совместно с командой GROUP BY, иначе возникнет ошибка.
3. **Вопрос:** Возможно ли изменить порядок объединения строк?
   * **Ответ:** Да, используйте конструкцию ORDER BY внутри STRING\_AGG().
4. **Вопрос:** Будет ли выполняться запрос быстрее, если сортировка не задана?
   * **Ответ:** Нет, сортировка не влияет на скорость выполнения, но улучшает читаемость и структурированность результата.
5. **Вопрос:** Существуют ли ограничения на длину итоговой строки?
   * **Ответ:** Длина итоговой строки ограничивается размером переменной VARCHAR или TEXT в PostgreSQL.
6. **Вопрос:** Можем ли мы передавать произвольные символы в качестве разделителя?
   * **Ответ:** Да, любой допустимый символ или последовательность символов может выступать в роли разделителя.
7. **Вопрос:** Какая разница между STRING\_AGG() и ARRAY\_AGG()?
   * **Ответ:** STRING\_AGG() собирает строки в единую строку, а ARRAY\_AGG() создаёт массив значений.
8. **Вопрос:** Возможно ли применить фильтры внутри самого STRING\_AGG()?
   * **Ответ:** Фильтры применяются вне оператора, через конструкцию WHERE перед GROUP BY.
9. **Вопрос:** Что произойдет, если не передать разделитель?
   * **Ответ:** Возникнет ошибка, так как разделитель обязателен.
10. **Вопрос:** Применяется ли оператор STRING\_AGG() только для электронных писем?
    * **Ответ:** Нет, его можно использовать для любых текстовых данных, требующих объединения в группы.

## **Создание таблиц. Добавление внешних ключей, значений столбцов по умолчанию, уникальных ограничений целостности. Удаление таблиц и его параметры. Переименование таблиц. Truncate таблиц. Чем Truncate отличается от Delete.**

### **Основные команды и понятия**

**Создание таблиц:**PostgreSQL позволяет создавать структуры таблиц, используя команду CREATE TABLE. Таблица состоит из строк и столбцов, каждый из которых имеет определенный тип данных. Типы данных могут варьироваться от простых типов (integer, varchar) до сложных структурированных типов (jsonb).

Пример простого создания таблицы сотрудников:

CREATE TABLE employees (

id SERIAL PRIMARY KEY,

first\_name VARCHAR(50),

last\_name VARCHAR(50),

email VARCHAR(100) UNIQUE NOT NULL,

hire\_date DATE DEFAULT CURRENT\_DATE,

salary NUMERIC CHECK(salary > 0)

);

Здесь мы видим несколько важных моментов:

* **SERIAL**: специальный тип данных, автоматически создающий последовательность чисел. Когда вставляется новая запись, этот столбец получает уникальный номер.
* **PRIMARY KEY**: определяет первичный ключ таблицы, обеспечивающий уникальность каждой строки.
* **UNIQUE**: гарантирует, что значения в данном столбце будут уникальны.
* **NOT NULL**: означает, что данное поле обязательно должно иметь значение.
* **DEFAULT**: устанавливает значение по умолчанию, которое используется, если оно явно не задано при вставке новой записи.
* **CHECK**: ограничивает допустимые значения в колонке, проверяя условие перед записью.

### **Внешние ключи**

Внешний ключ связывает одну таблицу с другой. Это механизм обеспечения согласованности между двумя таблицами. Например, рассмотрим следующую структуру базы данных:

Таблица департаментов:

CREATE TABLE departments (

dept\_id SERIAL PRIMARY KEY,

name VARCHAR(50) NOT NULL

);

Теперь свяжем нашу таблицу сотрудников с департаментами через внешний ключ:

ALTER TABLE employees ADD COLUMN department\_id INT REFERENCES departments(dept\_id);

или же сразу при создании таблицы:

CREATE TABLE employees (

...

department\_id INT REFERENCES departments(dept\_id)

);

Команда REFERENCES создает связь между значениями поля department\_id и полем dept\_id в таблице departments.

#### Особенности внешних ключей:

* Если удалить строку из родительской таблицы (например, удалив отдел), SQL-сервер откажется удалять её, пока есть зависимые данные в дочерней таблице. Для разрешения этого ограничения используют каскадирование действий:

FOREIGN KEY (department\_id) REFERENCES departments(dept\_id) ON DELETE CASCADE

Это приведет к автоматическому удалению всех записей сотрудников, связанных с данным отделом.

### **Ограничения целостности**

Ограничения целостности гарантируют достоверность и правильность хранимых данных. Мы уже видели ограничение типа UNIQUE, но есть ещё другие важные типы:

* **PRIMARY KEY**: Уникальное значение, используемое для идентификации строки.
* **FOREIGN KEY**: Связь между двумя таблицами.
* **CHECK**: Проверка условий на уровне отдельной ячейки.
* **UNIQUE**: Гарантирует уникальность данных в указанном столбце.
* **NOT NULL**: Обязательное наличие значения.

Примеры:

-- Чек на возраст сотрудника старше 18 лет

CHECK(age >= 18)

-- Строгое имя отдела длиной менее 50 символов

CONSTRAINT valid\_department\_length CHECK(length(name) <= 50)

### **Значения по умолчанию**

Значения по умолчанию устанавливаются командой DEFAULT. Они задаются для полей, которым нужны начальные значения, если никаких иных значений не предоставляется при вставке данных.

Например:

hire\_date DATE DEFAULT CURRENT\_DATE

salary NUMERIC DEFAULT 0

is\_active BOOLEAN DEFAULT TRUE

Таким образом, при отсутствии указанного значения сервер сам подставит нужное значение.

### **Изменение и удаление таблиц**

Для изменения существующей таблицы существует команда ALTER TABLE. Она помогает добавлять новые столбцы, изменять существующие или даже переименовывать саму таблицу.

#### Примеры изменений:

1. **Переименовать таблицу**

ALTER TABLE employees RENAME TO staff;

1. **Изменить название столбца**

ALTER TABLE employees RENAME COLUMN first\_name TO given\_name;

1. **Удалить существующий столбец**

ALTER TABLE employees DROP COLUMN middle\_name;

1. **Создать новый индекс для повышения производительности запросов**

CREATE INDEX idx\_email ON employees(email);

### **Удаление таблиц**

Для удаления таблиц используется команда DROP TABLE. Эта операция необратима и должна использоваться осторожно.

Простой пример удаления таблицы:

DROP TABLE employees;

Однако, если таблица связана внешними ключами с другими таблицами, такая простая команда выдаст ошибку. Поэтому используются дополнительные опции:

* **CASCADE**: принудительно удаляет таблицу вместе со всеми зависимыми объектами (например, внешние ключи):

DROP TABLE employees CASCADE;

* **RESTRICT**: запрещает удаление, если существуют зависимости (это поведение по умолчанию):

DROP TABLE employees RESTRICT;

### **Команда TRUNCATE**

Команда TRUNCATE очищает всю таблицу, сохраняя её структуру, но уничтожая все данные внутри неё. Она работает быстрее обычного удаления (DELETE), поскольку действует непосредственно на низкоуровневые страницы хранения данных, минуя ряд проверок.

Использование:

TRUNCATE TABLE employees RESTART IDENTITY;

Параметр RESTART IDENTITY сбрасывает счётчик последовательности обратно к начальному значению (полезно для колонок типа SERIAL).

Основные отличия между TRUNCATE и DELETE:

|  |  |  |
| --- | --- | --- |
| **Отличия** | **TRUNCATE** | **DELETE** |
| Скорость | Очень быстрая | Медленнее |
| Транзакции | Требует отдельного подтверждения | Работает в рамках транзакций |
| Возврат данных | Невозможно восстановить | Данные восстанавливаются при откате |
| Последовательности | Может сбросить автоинкремент | Оставляет автоинкремент неизменённым |
| Записи журнала аудита | Нет | Да |

### **Каверзные вопросы и ответы:**

1. **Вопрос:** Что произойдёт, если попытаться создать таблицу с именем уже существующей?**Ответ:** Будет ошибка с сообщением "relation already exists".
2. **Вопрос:** Как добавить внешний ключ после создания таблицы?**Ответ:** Используем команду ALTER TABLE table\_name ADD FOREIGN KEY (column\_name) REFERENCES other\_table(other\_column);
3. **Вопрос:** Можно ли изменить порядок столбцов в таблице?**Ответ:** Прямого способа нет, но можно переписать таблицу заново либо воспользоваться расширением pg\_reorg.
4. **Вопрос:** Зачем нужен RESTART IDENTITY при выполнении TRUNCATE?**Ответ:** Чтобы перезагрузить автоинкрементируемый счётчик (например, столбец типа SERIAL).
5. **Вопрос:** Какие последствия удаления внешнего ключа из одной таблицы?**Ответ:** Все зависящие связи больше не защищены от нарушения целостности данных.
6. **Вопрос:** Как правильно создать уникальную комбинацию двух столбцов?**Ответ:** Используйте команду UNIQUE(column1, column2) при создании таблицы или позже добавьте ALTER TABLE ADD CONSTRAINT unique\_constraint UNIQUE(column1, column2);
7. **Вопрос:** Какой синтаксис сброса всей БД к первоначальному состоянию?**Ответ:** Выполнять полное удаление базы данных: DROP DATABASE dbname; Затем воссоздавать новую базу.
8. **Вопрос:** Какова разница между ON UPDATE CASCADE и ON DELETE CASCADE?**Ответ:** Первый обновляет связанные данные при изменении родительского ключа, второй удаляет дочернюю запись при удалении родителя.
9. **Вопрос:** Почему лучше избегать команды DROP TABLE CASCADE?**Ответ:** Потому что она автоматически удаляет все зависимые объекты, включая индексы, триггеры и внешние ключи, потенциально нарушая целостность схемы.
10. **Вопрос:** Есть ли возможность восстановить данные после выполнения TRUNCATE?**Ответ:** Нет, восстановление невозможно, так как эта команда немедленно освобождает пространство диска.

### **Заключение**

Работа с созданием, изменением и удалением таблиц является фундаментальной частью администрирования баз данных. Команды вроде CREATE TABLE, ALTER TABLE, TRUNCATE и DROP TABLE предоставляют мощные инструменты для управления структурой данных и поддержания их целостностью. Понимание различий между ними позволит эффективно проектировать и поддерживать ваши приложения.

## **Создание представлений (View)**

## **Представления (Views) в PostgreSQL**

Представления являются виртуальными таблицами, построенными на основе одного или нескольких реальных таблиц. Они позволяют упростить запросы, абстрагируя сложную логику и скрывая внутренние механизмы обработки данных. Основное преимущество представления заключается в возможности сократить повторяющиеся операции и повысить безопасность, предоставляя пользователям доступ лишь к необходимым данным.

### **Теория и принципы работы представлений**

Представления не хранят сами данные — они содержат только определение запроса, который выполняется динамически при обращении к представлению. Таким образом, представление служит своего рода интерфейсом для взаимодействия с базой данных.

Преимущества представлений:

* Упрощают сложные запросы.
* Повышают уровень безопасности путем сокрытия внутренних механизмов работы с таблицами.
* Обеспечивают гибкость и упрощают изменение структуры базы данных без затрагивания клиентских приложений.

Типичные сценарии использования представлений включают агрегированные отчёты, разделение прав доступа пользователей, объединение данных из разных источников и обеспечение прозрачного доступа к частям больших таблиц.

### **Создание представлений**

Команду для создания представления называют CREATE VIEW. Формат выглядит следующим образом:

CREATE VIEW view\_name AS SELECT \* FROM source\_table WHERE condition;

Рассмотрим подробнее каждую часть команды:

* **view\_name**: Имя создаваемого представления.
* **AS**: Ключевое слово, обозначающее начало определения представления.
* **SELECT**: Стандартная инструкция выборки данных.
* **FROM**: Указывает исходную таблицу, на которой основано представление.
* **WHERE**: Опциональный фильтр для отбора нужных данных.

Допустим, у вас есть таблица сотрудников employees, содержащая следующие поля:

CREATE TABLE employees (

employee\_id SERIAL PRIMARY KEY,

first\_name TEXT NOT NULL,

last\_name TEXT NOT NULL,

department TEXT,

salary NUMERIC

);

Чтобы создать простое представление, показывающее список сотрудников определённого департамента, выполните следующий код:

CREATE VIEW sales\_employees AS

SELECT \*

FROM employees

WHERE department = 'Sales';

Теперь любое обращение к этому представлению вернёт данные только тех сотрудников, кто относится к департаменту продаж.

### **Использование представлений**

После создания представления обращаться к нему можно точно так же, как и к обычной таблице:

SELECT \* FROM sales\_employees;

Эта команда вернет всех сотрудников отдела продаж, хотя фактически физически хранится только одна таблица — employees.

### **Модификация представлений**

Иногда возникает необходимость обновить определение представления. Это делается с помощью команды CREATE OR REPLACE VIEW:

CREATE OR REPLACE VIEW sales\_employees AS

SELECT \*

FROM employees

WHERE department IN ('Sales', 'Marketing');

Данная команда заменит предыдущее определение представления новым.

### **Удаление представлений**

При необходимости представление можно удалить с помощью команды DROP VIEW:

DROP VIEW IF EXISTS sales\_employees;

Эта команда удалит представление sales\_employees, если оно существует.

### **Параметры создания представлений**

Помимо стандартного создания представлений, PostgreSQL поддерживает различные модификации поведения представлений:

* **WITH LOCAL CHECK OPTION**: Используется для проверки условия фильтра при попытке обновления данных через представление. Позволяет убедиться, что данные соответствуют условиям фильтрации представления.
* **WITH CASCADED CHECK OPTION**: Распространяет проверку условий на вложенные представления.

Пример:

CREATE VIEW low\_salary\_employees AS

SELECT \*

FROM employees

WHERE salary < 50000

WITH LOCAL CHECK OPTION;

Если теперь попробовать обновить зарплату сотрудника через это представление, попытка установить зарплату выше порога (50,00050,000) приведёт к ошибке.

### **Материалы представления (Materialized Views)**

Материалы представления представляют собой особый вид представлений, где результаты сохраняются физически в базе данных. Это ускоряет выполнение запросов, так как результаты предварительно вычислены и хранятся отдельно.

Формат создания материала представления:

CREATE MATERIALIZED VIEW materialized\_view\_name AS

SELECT \* FROM source\_table WHERE condition;

Эти представления полезны, когда необходимо часто получать одни и те же данные и агрегационные показатели, особенно если такие запросы выполняются медленно.

Для обновления материалов представлений вручную применяется команда:

REFRESH MATERIALIZED VIEW materialized\_view\_name;

### **Вопросы и ответы**

1. **Что произойдет, если изменить схему основной таблицы, на которой основано представление?Ответ:** Необходимо повторно применить команду CREATE OR REPLACE VIEW, чтобы обновить определение представления.
2. **Можно ли сделать представление изменяемым (updatable)?Ответ:** Да, некоторые представления допускают обновление через себя, однако PostgreSQL требует соблюдения строгих правил соответствия полей и ограничений.
3. **Какие преимущества даёт использование материализованных представлений?Ответ:** Материализованные представления улучшают производительность запросов благодаря предварительному сохранению результатов.
4. **Что такое секционированные представления?Ответ:** Хотя термин "секционированные представления" специфичен для некоторых СУБД (например, Microsoft SQL Server), PostgreSQL предлагает альтернативы в виде разбиения таблиц на секции (partitioned tables). Но концепция представления остаётся схожей — отображается подмножество данных.
5. **Почему важно учитывать права доступа при работе с представлениями?Ответ:** Права доступа помогают защитить конфиденциальные данные, позволяя ограничить круг лиц, имеющих доступ к определенной части информации.
6. **Какой эффект оказывает команда WITH LOCAL CHECK OPTION?Ответ:** Она заставляет проверять условие представления при любом обновлении данных через это представление.
7. **Что случится, если попытаться обновить данные через материализованное представление?Ответ:** Без специальной настройки попытки обновить данные через материализованное представление завершатся неудачей, так как оно представляет статический снимок данных.
8. **Как правильно определить порядок сортировки в представлении?Ответ:** Порядок сортировки определяется стандартным оператором ORDER BY в определении представления.
9. **Может ли одно представление зависеть от другого представления?Ответ:** Да, представления могут быть построены друг на друга, образуя цепочку вложенных представлений.
10. **Нужно ли постоянно обновлять материалы представления вручную?Ответ:** Нет, можно настроить периодическое автоматическое обновление с использованием заданий cron или специальных функций PostgreSQL.

Отличия между обычными представлениями (VIEW) и материализованными представлениями (MATERIАЛIZED VIEW) довольно существенные, хотя внешне они выглядят одинаково. Давайте сравним их по ключевым характеристикам:

### **🔹** View (простое представление)

📌 **Описание**: Представление — это виртуальная таблица, созданная на основе SQL-запроса. Оно не хранит никаких данных, а фактически перезапускает запрос всякий раз, когда вы обращаетесь к этому представлению.

📌 **Хранение данных**: Представление не содержит физической копии данных. Всякий раз, когда вы отправляете запрос к представлению, запускается лежащий в его основе SQL-запрос, и данные собираются динамически из базовой таблицы.

📌 **Скорость исполнения**: Поскольку данные генерируются динамически, выполнение запросов к представлению зависит от сложности базового запроса и объема данных в связанных таблицах. Если запрос тяжелый, обращения к представлению могут занимать значительное время.

📌 **Обновляемость**: Представления мгновенно отражают любые изменения в базовых таблицах, так как они постоянно пересчитываются при любом обращении.

📌 **Поддерживаемые операции**: Представления можно использовать почти так же, как физические таблицы, однако некоторые операции (например, модификация данных) возможны только при особых обстоятельствах.

### **🔸** Materialized View (материализованное представление)

📌 **Описание**: Материализованное представление — это физическая копия данных, полученных в результате выполнения SQL-запроса. Эти данные сохраняются на диске и обновляются периодически, либо вручную, либо автоматически.

📌 **Хранение данных**: Материализованное представление реально хранит данные, полученные из запроса, на физическом носителе. Именно поэтому оно называется "материализованным": данные реально присутствуют и доступны сразу.

📌 **Скорость исполнения**: Так как данные уже готовы и лежат на диске, запросы к материализованному представлению исполняются гораздо быстрее, чем к обычному представлению. Нет необходимости многократно выполнять исходный запрос.

📌 **Обновляемость**: Материализованные представления требуют обновления. Пользователи могут принудительно обновлять их вручную или настроить автоматическое обновление (через задание cron или PL/pgSQL процедуры). Таким образом, данные в материализованном представлении не всегда актуальны в режиме реального времени.

📌 **Поддерживаемые операции**: Материализованные представления тоже ведут себя подобно физическим таблицам, однако у них ограничено прямое редактирование данных. В основном они используются для чтения данных.

### Итоговые отличия**:**

* **Скорость**: Материализованные представления работают быстрее, так как данные уже собраны и ждут обращений.
* **Актуальность данных**: Простые представления всегда показывают актуальные данные, в то время как материализованные представления могут отставать, так как нуждаются в регулярном обновлении.
* **Место на диске**: Материализованные представления занимают дополнительное пространство на диске, так как сохраняют физическую копию данных.
* **Применение**: Простые представления хорошо подходят для мелких запросов и ситуаций, где важны свежие данные. Материализованные представления идеальны для тяжёлых аналитических запросов, где высокая частота обращений важнее мгновенной актуальности данных.

Выбор между этими видами представлений зависит от характера вашего проекта и требований к быстродействию и свежести данных.

## **Создание материализованных представлений. Инкрементальное обновление материализованных представлений**

## **Материализированные представления (Materialized Views) в PostgreSQL**

Материализированные представления представляют собой особую разновидность обычных представлений, где результаты заранее рассчитываются и сохраняются физически на диске. Такой подход обеспечивает значительное повышение производительности при обработке часто используемых данных, особенно в ситуациях, когда база данных подвергается большому количеству аналитических запросов или сложных агрегатов.

### **Преимущества материализованных представлений:**

* **Производительность**: Результаты предварительно рассчитаны и сохранены, что снижает нагрузку на систему при каждом новом запросе.
* **Кэширование**: Улучшение масштабируемости и снижение нагрузки на CPU и память сервера.
* **Ускорение анализа**: Полезны для аналитики и отчетов, требующих значительных объемов обработанных данных.
* **Снижение блокировки**: Минимизирует конфликты блокировок между читающими и записывающими процессами.

### **Недостатки материализованных представлений:**

* **Обновляемость**: Нужно периодически обновлять представление вручную или автоматически, иначе информация устаревает.
* **Дополнительное хранилище**: Хранятся дублирующие копии данных, занимающие дополнительное место на диске.
* **Зависимость от основной таблицы**: Любые изменения в структуре основной таблицы требуют переопределения материализованного представления.

### **Создание материализованного представления**

Создание материализованного представления осуществляется командой CREATE MATERIALIZED VIEW. Вот простой пример на основе таблицы сотрудников (employees):

CREATE MATERIALIZED VIEW mv\_employee\_salary\_summary AS

SELECT department, AVG(salary) as avg\_salary, COUNT(\*) as num\_employees

FROM employees

GROUP BY department;

Давайте разберём детально, что делает каждая часть команды:

* **CREATE MATERIALIZED VIEW**: Создаем новое материализованное представление.
* **mv\_employee\_salary\_summary**: Название нашего нового представления.
* **AS**: Указываем источник данных.
* **SELECT**: Обычный оператор выбора данных.
* **AVG**, **COUNT**: Агрегаты, применяемые к результатам выборки.
* **GROUP BY**: Группа результата по департаментам.

Теперь любые обращения к представлению возвращают готовые сводные данные, сформированные заранее:

SELECT \* FROM mv\_employee\_salary\_summary;

### **Инкрементальное обновление материализованных представлений**

По умолчанию материализованные представления не обновляются автоматически при изменениях в базовых таблицах. Однако PostgreSQL позволяет организовать инкрементальное обновление через регулярные задания (cron jobs) или ручное выполнение оператора REFRESH MATERIALIZED VIEW.

Инкрементальное обновление подразумевает сохранение состояния предыдущих расчетов и применение изменений, произошедших с момента последнего обновления. Рассмотрим два подхода:

1. **Полный рефреш**: Перерасчет всего представления целиком:

REFRESH MATERIALIZED VIEW mv\_employee\_salary\_summary;

Этот способ подходит, если объем данных невелик или требуется полная точность расчета. Однако, он затратен по ресурсам и времени.

1. **Частичное (инкрементальное) обновление**: Здесь расчет ведется только для новых или изменённых данных. Этот метод полезен, если известно, что количество изменений относительно общего объема данных минимально.

Инкрементальное обновление требует дополнительного программирования, так как Postgres не поддерживает нативную функциональность автоматического инкрементального обновления. Обычно применяются методы отслеживания изменений с помощью триггеров, временных таблиц или вспомогательных журналов.

Вот пример реализации простейшего механизма с временными таблицами:

BEGIN TRANSACTION;

-- Сохраняем предыдущие данные

CREATE TEMPORARY TABLE temp\_mv\_employee\_salary\_summary AS

SELECT \* FROM mv\_employee\_salary\_summary;

-- Очищаем старое представление

TRUNCATE TABLE mv\_employee\_salary\_summary;

-- Пересчитываем суммарные данные заново

INSERT INTO mv\_employee\_salary\_summary

SELECT department, AVG(salary) as avg\_salary, COUNT(\*) as num\_employees

FROM employees

GROUP BY department;

COMMIT;

Этот сценарий реализует полный пересчёт данных, но его легко адаптировать для инкрементальных изменений, обрабатывая только недавно изменённые строки.

### **Другие аналогичные конструкции в PostgreSQL**

1. **Common Table Expressions (CTE)**: Альтернатива простым представлениям, используемая для временных наборов данных в пределах одного запроса. CTE работают быстрее и занимают меньше места, но не поддерживают кэширование.
2. **Temporary Tables**: Временные таблицы также служат промежуточным этапом для быстрого хранения промежуточных данных. Однако временные таблицы живут только в течение сессии подключения.
3. **Functions with RETURNING**: Функции с возвратом данных похожи на представления, но отличаются возможностью сохранять состояние и проводить расчеты внутри самой функции.

### **Часто задаваемые вопросы (FAQ)**

1. **Вопрос:** Возможно ли избежать полного перерасчета при изменении отдельных строк в базовой таблице?**Ответ:** Инкрементальное обновление возможно путём сохранения текущего состояния и добавления только изменившихся данных. Реализуется самостоятельно, так как встроенной поддержки нет.
2. **Вопрос:** Как управлять временем жизни материализованного представления?**Ответ:** Время жизни контролируется настройкой автопересчёта (через cron job или запланированную задачу).
3. **Вопрос:** Поддерживается ли автоматическая индексация в материализованном представлении?**Ответ:** Индексация возможна после создания самого представления, подобно обычным таблицам.
4. **Вопрос:** Можно ли создать материализованное представление поверх другого материализованного представления?**Ответ:** Да, но следует помнить, что такой подход увеличивает накладные расходы на хранение и обслуживание.
5. **Вопрос:** Подходят ли материализованные представления для OLTP-систем?**Ответ:** В основном материализованные представления ориентированы на аналитические системы (OLAP), но могут применяться и в OLTP-системах с осторожностью, учитывая высокую стоимость пересчетов.
6. **Вопрос:** Обязательно ли регулярно очищать старые версии представления?**Ответ:** Нет, старая версия замещается новой при каждом обновлении. Тем не менее рекомендуется контролировать размер дискового пространства.
7. **Вопрос:** Могут ли материализованные представления содержать JOIN'ы?**Ответ:** Да, материализованные представления могут включать сложные операторы объединения (JOIN) и агрегаты.
8. **Вопрос:** Насколько сложно поддерживать материализованные представления при высокой нагрузке?**Ответ:** Высокая нагрузка на обработку данных повышает требования к производительности, но грамотное проектирование и регулярное обновление делают их эффективными инструментами.
9. **Вопрос:** Нуждается ли материализованное представление в индексации?**Ответ:** Индексирование полезно для улучшения скорости чтения, особенно если представления большие.
10. **Вопрос:** Всегда ли требуется вручную запускать REFRESH MATERIALIZED VIEW?**Ответ:** Только при полном пересчёте. Частично обновляемые представления могут требовать меньшего вмешательства администратора.

**View (обычное представление)**:

* Генерирует данные динамически при каждом запросе.
* Не хранит физические данные на диске.
* Всегда показывает самые актуальные данные.
* Менее производительное при частых запросах к крупным наборам данных.

**Materialized View (материализованное представление)**:

* Хранимая физическая копия данных, созданная однократно.
* Быстрое получение данных при последующем обращении.
* Требует регулярного обновления для актуализации данных.
* Более высокая производительность при аналитике и больших объёмах данных.

## **Создание последовательностей (Sequence). Использование Sequence как Default значение при создание таблицы. Реинициализация Sequence.**

### **Основные положения:**

Последовательности (sequences) — это объекты базы данных, предназначенные для автоматического генерации уникальных последовательных целых чисел. Очень удобно использовать их в качестве значений по умолчанию для первичных ключей (primary keys).

### **Зачем нужны последовательности:**

* Генерация уникальных идентификаторов (ID).
* Устранение риска дубликатов при ручном назначении идентификаторов.
* Увеличение надежности базы данных за счёт автоматизированного подхода к формированию значений.

### **Пример использования последовательности:**

Пусть у вас есть таблица сотрудников, и вы хотите автоматически генерировать уникальные идентификационные номера:

CREATE SEQUENCE employee\_id\_seq START WITH 1 INCREMENT BY 1 NO MAXVALUE;

CREATE TABLE employees (

employee\_id BIGINT DEFAULT nextval('employee\_id\_seq'),

first\_name TEXT,

last\_name TEXT

);

### **Механизм работы:**

* nextval('sequence\_name') — это функция, которая возвращает следующее значение из последовательности.
* Параметр DEFAULT гарантирует, что при вставке данных в таблицу, если не указано конкретное значение для поля employee\_id, будет автоматически применяться значение из последовательности.

### **Что такое последовательность (Sequence)?**

Последовательность (sequence) — это объект базы данных, предназначенный для автоматической генерации уникальных числовых значений. Её удобно использовать там, где требуется обеспечить уникальную нумерацию записей, например, для первичного ключа (id) в таблице.

Основное назначение последовательности — давать уникальные целые числа (обычно положительные), начинающиеся с какого-то значения и далее последовательно увеличивающиеся на указанный шаг.

### **Пример реальной ситуации**

Допустим, у тебя есть таблица сотрудников:

CREATE TABLE employees (

employee\_id INTEGER PRIMARY KEY,

first\_name VARCHAR(50),

last\_name VARCHAR(50)

);

Когда мы добавляем нового сотрудника, нам надо придумать какое-нибудь уникальное значение для поля employee\_id. Вместо того чтобы придумывать это самим, мы можем поручить эту задачу последовательности. То есть каждый раз, когда добавляется сотрудник, последовательность сама даст новое значение.

### **Пример создания последовательности**

Сначала создадим последовательность:

CREATE SEQUENCE seq\_employee\_id

START WITH 1000 -- Начинать с тысячи

INCREMENT BY 1 -- Каждое последующее число увеличивается на единицу

NO MINVALUE -- Никаких ограничений снизу

NO MAXVALUE -- Никаких ограничений сверху

CACHE 10 -- Заранее зарезервировать 10 чисел для быстроты выдачи

NO CYCLE; -- После достижения максимального значения перестаёт считать дальше

Теперь назначим её нашей таблице:

CREATE TABLE employees (

employee\_id INTEGER PRIMARY KEY DEFAULT nextval('seq\_employee\_id'), -- Авто присваивание из последовательности

first\_name VARCHAR(50),

last\_name VARCHAR(50)

);

Так что при добавлении новых сотрудников последовательность автоматически присвоит уникальное значение полю employee\_id:

INSERT INTO employees(first\_name, last\_name) VALUES ('Иван', 'Иванов');

INSERT INTO employees(first\_name, last\_name) VALUES ('Петя', 'Петров');

### **Дополнительные полезные команды**

1. **Получить следующее значение из последовательности**:

SELECT nextval('seq\_employee\_id'); -- Возвращает следующее число

1. **Посмотреть текущее значение последовательности**:

SELECT currval('seq\_employee\_id'); -- Показывает последний выданный номер

1. **Реинициализировать последовательность** (сбросить к исходному значению):

ALTER SEQUENCE seq\_employee\_id RESTART WITH 1000; -- Вернуть начало последовательности

### **Ещё немного теории**

Последовательности могут:

* Увеличивать значения постепенно (например, каждые пять единиц: INCREMENT BY 5)
* Установить минимальное и максимальное значения (например, не ниже 100 и не выше 1000000)
* Буферизоваться (опция CACHE n): выдавать значения пакетами, ускоряя их генерацию
* Циклироваться (циклически возвращаться к началу после исчерпания максимума: CYCLE)

### **Аналогичные вещи в других СУБД**

* В MySQL аналогом последовательности является тип столбца AUTO\_INCREMENT.
* В Oracle тоже есть понятие последовательности (создаются командой CREATE SEQUENCE).

### **10 Каверзных вопросов по данной теме:**

1. **Вопрос:** Как вернуть последовательность к начальному значению?**Ответ:** Через команду ALTER SEQUENCE … RESTART WITH X.
2. **Вопрос:** Что возвращает функция nextval()?**Ответ:** Следующее уникальное значение из последовательности.
3. **Вопрос:** Нужно ли вручную вызывать nextval() при вставке в таблицу?**Ответ:** Нет, если последовательность назначена как дефолтное значение для поля, PostgreSQL автоматически вызывает nextval().
4. **Вопрос:** Влияет ли параметр CACHE N на расход памяти?**Ответ:** Да, увеличивает расход памяти, но ускоряет выдачу значений.
5. **Вопрос:** Можно ли уменьшить значение последовательности после его роста?**Ответ:** Нет, уменьшение номера доступно только через реинициализацию (перезапуск последовательности).
6. **Вопрос:** Может ли две таблицы использовать одну и ту же последовательность?**Ответ:** Да, но тогда значения могут пересекаться между разными таблицами.
7. **Вопрос:** Почему последовательность полезна в многопользовательском режиме?**Ответ:** Гарантирует уникальные значения даже при одновременных вставках.
8. **Вопрос:** Что делает опция NO CYCLE?**Ответ:** Предотвращает повторение значений после достижения верхнего лимита.
9. **Вопрос:** Важно ли выбирать правильный размер шага (INCREMENT BY)?**Ответ:** Да, чрезмерно большой шаг может приводить к быстрому росту размеров идентификатора, маленький шаг замедлит рост, но увеличит частоту обращения к базе данных.
10. **Вопрос:** Можно ли применять последовательности для столбцов, отличных от первичного ключа?**Ответ:** Да, это возможно, хотя чаще всего их применяют именно для первичных ключей.

## **Создание индексов. Различные типы индексов**

## **Индексы в PostgreSQL: теория, практика и примеры**

Индексы — это структуры данных, используемые для оптимизации поиска и извлечения информации из таблиц. Индексы увеличивают производительность запросов, делая доступ к данным более быстрым и эффективным. Основная цель индексов — минимизировать количество сканирований строк и ускорить операции поиска, сортировки и группировки.

### **Важность индексов**

Без индексов большинство запросов требует полного просмотра всех строк таблицы (так называемое полностраничное сканирование), что становится неэффективным с ростом объёма данных. Индексы решают эту проблему, создавая специальную структуру, похожую на оглавление в книге, что позволяет быстро находить нужные данные.

### **Принцип работы индексов**

Индексы создают упорядоченную структуру для данных в выбранных столбцах таблицы. Каждый индекс привязывается к одному или нескольким столбцам и строится на основе выбранной стратегии организации данных (типа индекса). При поиске конкретных данных PostgreSQL сначала обращается к индексу, находит местоположение нужной строки в таблице и извлекает необходимые данные.

Индексы имеют огромное влияние на производительность, но злоупотребление ими может негативно сказываться на скорости вставки и обновления данных, так как каждое изменение данных в таблице требует соответствующего обновления индексов.

### **Типы индексов в PostgreSQL**

PostgreSQL поддерживает несколько различных типов индексов, каждый из которых предназначен для определенных целей и оптимизирован под конкретные сценарии использования.

#### B-tree-индексы

B-tree (Binary Tree) — самый распространенный тип индекса. Это сбалансированное дерево, поддерживающее быструю сортировку и эффективный диапазон поисковых запросов. Применяется практически повсеместно, особенно для стандартных сравнительных операторов (равенство, неравенство, интервалы).

Пример создания B-tree индекса:

CREATE INDEX emp\_last\_name\_idx ON employees(last\_name);

Здесь создается индекс по столбцу last\_name, который будет использоваться для ускорения запросов по фамилии сотрудников.

#### Hash-индексы

Hash-индексы эффективны исключительно для точного равенства. Такие индексы строятся на хеш-функциях, которые преобразуют данные в короткие ключевые коды, обеспечивая быстрый поиск совпадений. Однако hash-индексы не поддерживают диапазонные запросы или сортировку.

Пример создания hash-индекса:

CREATE INDEX emp\_hash\_first\_name\_idx ON employees USING HASH(first\_name);

#### GiST-индексы

GiST (Generalized Search Tree) — обобщенное дерево поиска, предназначено для многокритериальных запросов, пространственных данных, полнотекстового поиска и сложных запросов. GiST хорошо подходит для работы с геоданными, геометрическими фигурами и другими специализированными типами данных.

Пример создания GiST-индекса:

CREATE INDEX emp\_location\_gist\_idx ON employees USING GIST(location);

#### SP-GiST-индексы

SP-GiST (Space-Partitioned Generalized Search Tree) — улучшенная версия GiST, специально разработанная для пространственного разделения данных. SP-GiST лучше справляется с запросами по большим объемам распределенных данных, таких как точки на карте или другие географические координаты.

Пример создания SP-GiST-индекса:

CREATE INDEX emp\_spatial\_idx ON employees USING SP\_GIST(location);

#### GIN-индексы

GIN (Generalized Inverted Index) — обратные индексы, отлично подходящие для высокочастотных множественных данных, таких как массивы или текстовые поля. Особенно полезен для полнотекстового поиска.

Пример создания GIN-индекса:

CREATE INDEX emp\_tags\_gin\_idx ON employees USING GIN(tags);

#### BRIN-индексы

BRIN (Block Range Index) — блоки диапазонов индексов, предназначены для огромных таблиц, где распределение данных линейно или предсказуемо. Хорошо подходят для архивных данных или исторических регистров.

Пример создания BRIN-индекса:

CREATE INDEX emp\_hire\_date\_brin\_idx ON employees(hire\_date) USING BRIN;

### **Практика использования индексов**

Вернемся к примеру с сотрудниками (employees). Допустим, у нас есть таблица сотрудников:

CREATE TABLE employees (

employee\_id SERIAL PRIMARY KEY,

first\_name VARCHAR(50),

last\_name VARCHAR(50),

email VARCHAR(100),

hire\_date DATE,

location GEOMETRY(Point, 4326),

tags TEXT[]

);

Мы хотим ускорить выборку сотрудников по различным критериям:

1. Поиск сотрудников по фамилии:

CREATE INDEX emp\_last\_name\_idx ON employees(last\_name);

1. Быстрая проверка наличия электронных адресов:

CREATE INDEX emp\_email\_hash\_idx ON employees USING HASH(email);

1. Геопространственный поиск сотрудников поблизости:

CREATE INDEX emp\_location\_gist\_idx ON employees USING GIST(location);

1. Полнотекстовый поиск по тегам:

CREATE INDEX emp\_tags\_gin\_idx ON employees USING GIN(tags);

### **Советы по использованию индексов**

1. Избегайте избыточных индексов. Каждый дополнительный индекс замедляет операции вставки и обновления.
2. Регулярно оценивайте эффективность индексов с помощью инструментов мониторинга, таких как EXPLAIN ANALYZE.
3. Рассмотрите сочетание разных типов индексов для решения комплексных задач.
4. Анализируйте потребности ваших запросов и создавайте индексы только там, где они действительно необходимы.

### **10 каверзных вопросов по теме**

1. **Вопрос:** Почему индекс не всегда ускоряет запрос?**Ответ:** Иногда небольшие таблицы или неквалифицированный запрос могут оказаться медленнее при использовании индекса из-за дополнительных издержек на обход дерева.
2. **Вопрос:** Можно ли создавать индексы на нескольких столбцах?**Ответ:** Да, мультиколоные индексы создаются командой CREATE INDEX ON table(col1, col2).
3. **Вопрос:** Является ли B-tree единственным типом индекса в PostgreSQL?**Ответ:** Нет, кроме B-tree, PostgreSQL поддерживает GIN, GiST, SP-GiST, hash и BRIN.
4. **Вопрос:** Что произойдет, если включить параметр CONCURRENTLY при создании индекса?**Ответ:** Процесс создания индекса станет безопасным для параллельной работы, позволяя другим операциям продолжаться параллельно созданию индекса.
5. **Вопрос:** Как проверить, какой индекс используется для запроса?**Ответ:** Используя команду EXPLAIN ANALYZE, которая покажет план выполнения запроса.
6. **Вопрос:** Можно ли индексировать JSON-данные?**Ответ:** Да, индексы могут создаваться и для JSON-столбцов с помощью GIN или GIST.
7. **Вопрос:** Всегда ли следует создавать индексы для первичных ключей?**Ответ:** Первичные ключи автоматически получают индексы, так что создавать их дополнительно не требуется.
8. **Вопрос:** Каковы недостатки использования индексов?**Ответ:** Замедляют операции вставки и обновления, занимают дополнительное дисковое пространство.
9. **Вопрос:** Можно ли отменить создание индекса после начала процесса?**Ответ:** Нельзя прервать создание индекса без потери произведённой работы.
10. **Вопрос:** Что значит параметр USING BTREE при создании индекса?**Ответ:** Указывает на использование метода построения индекса B-tree, который подходит для большинства случаев и является индексом по умолчанию.

## **Обзор структуры (Блоков) PL/pgSQL**

## Обзор структуры (блоков) PL/pgSQL

PL/pgSQL — процедурный язык программирования PostgreSQL, используемый для написания хранимых функций, триггеров и скриптов, выполняемых внутри базы данных. Структура PL/pgSQL напоминает традиционные языки программирования вроде Pascal или SQL Server T-SQL. Этот обзор поможет разобраться в ключевых блоках и структурах, используемых в PL/pgSQL, а также понять различия между различными элементами синтаксиса.

### **Основные блоки PL/pgSQL**

#### Блок объявления переменных (DECLARE)

Перед выполнением основной части процедуры часто возникает необходимость объявить локальные переменные. Это делается в секции DECLARE. Переменные могут хранить данные различных типов (например, строки, числа, массивы).

CREATE FUNCTION example\_function() RETURNS void AS $$

DECLARE

v\_employee\_id integer := 1;

BEGIN

-- Основная логика программы

END;

$$ LANGUAGE plpgsql;

**Что делает этот блок:**Определяет область видимости переменной и её начальное значение. Все переменные существуют только в пределах блока, в котором они определены.

#### Основной исполняемый блок (BEGIN...END)

Основной код хранится именно здесь. Внутри этого блока размещаются команды SQL, условные операторы, циклы и другие конструкции.

CREATE FUNCTION update\_salary(employee\_id integer, new\_salary numeric)

RETURNS void AS $$

BEGIN

UPDATE employees SET salary = new\_salary WHERE id = employee\_id;

END;

$$ LANGUAGE plpgsql;

**Что делает этот блок:**Выполняет основную бизнес-логику хранимой процедуры. Здесь мы можем разместить любые инструкции PL/pgSQL или обычные SQL-запросы.

#### Условные выражения (IF...THEN, CASE)

Эти конструкции позволяют изменять поведение программы в зависимости от условий.

Пример IF THEN ELSE:

CREATE FUNCTION calculate\_bonus(employee\_id integer)

RETURNS numeric AS $$

DECLARE

bonus numeric;

BEGIN

SELECT INTO bonus salary \* 0.1 FROM employees WHERE id = employee\_id;

IF bonus > 1000 THEN

RAISE NOTICE 'Высокий бонус';

ELSIF bonus BETWEEN 500 AND 1000 THEN

RAISE NOTICE 'Средний бонус';

ELSE

RAISE NOTICE 'Низкий бонус';

END IF;

RETURN bonus;

END;

$$ LANGUAGE plpgsql;

**Что делает эта конструкция:**Позволяет динамически выбирать разные пути исполнения в зависимости от значений переменных или результатов запросов.

#### Циклические конструкции (FOR, WHILE, LOOP)

Для повторяющихся действий используются циклы. Рассмотрим пример цикла FOR:

CREATE FUNCTION print\_employees()

RETURNS void AS $$

DECLARE

r record;

BEGIN

FOR r IN SELECT \* FROM employees LOOP

RAISE NOTICE '%', r.name;

END LOOP;

END;

$$ LANGUAGE plpgsql;

$$

**Что делает цикл:**Перебирает записи из выборки и позволяет последовательно обрабатывать каждую строку.

#### Исключение ошибок (EXCEPTION)

При возникновении исключений в процессе выполнения можно перехватывать ошибки и реагировать на них соответствующим образом.

CREATE FUNCTION safe\_divide(numerator numeric, denominator numeric)

RETURNS numeric AS $$

BEGIN

RETURN numerator / denominator;

EXCEPTION WHEN division\_by\_zero THEN

RAISE NOTICE 'Ошибка деления на ноль!';

RETURN NULL;

END;

$$ LANGUAGE plpgsql;

**Что делает исключение:**Защищает программу от аварийного завершения при возникновении неожиданных ситуаций (например, деление на ноль). Позволяет продолжить выполнение или вернуть разумное значение.

### **Краткое сравнение с другими языками баз данных**

|  |  |  |
| --- | --- | --- |
| **Конструкция** | **PL/pgSQL** | **MySQL Stored Procedures** |
| Объявление переменных | DECLARE | DECLARE |
| Исполнение | BEGIN...END | BEGIN...END |
| Условия | IF...THEN | IF...THEN |
| Циклы | FOR, WHILE | REPEAT, WHILE |
| Исключения | EXCEPTION | HANDLER |

MySQL немного отличается по стилю реализации конструкций (например, использование отдельного оператора обработки исключений), однако общая концепция схожа.

### **Примеры и практика на таблице employees**

Рассмотрим таблицу сотрудников:

CREATE TABLE employees (

id SERIAL PRIMARY KEY,

name VARCHAR(100),

department VARCHAR(50),

salary NUMERIC

);

INSERT INTO employees VALUES (1, 'Иван Иванов', 'IT', 80000);

INSERT INTO employees VALUES (2, 'Марья Петрова', 'Sales', 70000);

Теперь напишем простейшую процедуру для вывода зарплаты сотрудника:

CREATE OR REPLACE FUNCTION show\_salary(employee\_id integer)

RETURNS numeric AS $$

DECLARE

emp\_salary numeric;

BEGIN

SELECT salary INTO emp\_salary FROM employees WHERE id = employee\_id;

RETURN emp\_salary;

END;

$$ LANGUAGE plpgsql;

Эта процедура принимает ID сотрудника и возвращает его зарплату.

### **Вопросы с подвохом (каверзные):**

1. Что произойдет, если в процедуре используется оператор SELECT без ключевого слова INTO?
   * Ответ: Запрос будет выполнен, но результат потеряется. Оператор SELECT вернет набор строк, но не сохранит его ни в одной переменной.
2. Какая разница между операторами RETURNING и обычным оператором RETURN?
   * Ответ: RETURNING используется для возврата записей непосредственно из операции DML (UPDATE, INSERT, DELETE), тогда как обычный RETURN предназначен для возвращения значения из функции.
3. Чем отличается цикл FOR от цикла LOOP?
   * Ответ: Цикл FOR автоматически перебирает элементы списка или результата запроса, а цикл LOOP требует ручного управления условиями выхода.
4. Можно ли вызвать хранимую процедуру внутри другой хранимой процедуры?
   * Ответ: Да, конечно! Одна процедура может вызывать другую таким же способом, как обычная функция вызывает другую функцию.
5. Почему лучше избегать чрезмерного использования вложенных блоков в PL/pgSQL?
   * Ответ: Чрезмерная вложенность затрудняет чтение и поддержку кода, увеличивает вероятность появления трудноуловимых ошибок и снижает производительность.
6. Когда целесообразно использовать конструкцию EXCEPTION?
   * Ответ: Для защиты от фатальных ошибок и возможности корректного завершения работы процедуры даже при возникновении непредвиденных обстоятельств.
7. Может ли один оператор BEGIN охватывать несколько операторов SQL?
   * Ответ: Да, один блок BEGIN может содержать любое количество операторов SQL, включая другие блоки BEGIN…END.
8. Зачем нужен оператор PERFORM?
   * Ответ: Используется для выполнения анонимных SQL-выражений, которые возвращают результат, но сами по себе не требуют присвоения результата какой-то переменной.
9. Могут ли две процедуры иметь одинаковые имена?
   * Ответ: Нет, имена процедур уникальны для каждой схемы. Однако возможно создание перегруженных функций, различающихся аргументами.
10. Какие ограничения накладывает PL/pgSQL на типы данных, передаваемых в качестве аргументов?
    * Ответ: Аргументы могут быть любого допустимого типа данных PostgreSQL, включая составные типы и домены.

## **Переменные и типы PL/pgSQL**

### **Основные сведения:**

PL/pgSQL — это процедурный язык расширения PostgreSQL, который позволяет писать хранимые процедуры и функции, содержащие переменные, условия, циклы и многое другое.

### **Что такое переменные?**

Переменные — это контейнеры для хранения данных внутри блока программы (процедуры или функции). Переменные можно объявлять с разными типами данных, чтобы хранить необходимые значения.

### **Типы переменных:**

В PL/pgSQL используются различные типы переменных, характерные для SQL и PostgreSQL:

* **INTEGER**, **BIGINT**, **NUMERIC**: Целые и вещественные числа.
* **TEXT**, **VARCHAR**, **CHARACTER**: Строковые данные разной длины.
* **BOOLEAN**: Булевский тип (TRUE, FALSE).
* **RECORD**: Специальный тип, позволяющий хранить целые записи (строки) из таблиц.
* **ANYELEMENT**: Обобщающий тип для передачи неизвестных заранее данных.

### **Пример объявления переменных:**

DECLARE

user\_id INTEGER := 101; -- объявление целочисленной переменной с инициализацией

full\_name TEXT; -- объявление строковой переменной

is\_active BOOLEAN := TRUE; -- булевская переменная

BEGIN

-- тело блока программы

END;

## **Переменные и типы PL/pgSQL**

PL/pgSQL — процедурный язык программирования PostgreSQL, расширяющий возможности SQL баз данных функциями, условиями, циклами и переменными. Рассмотрим подробнее работу с переменными и типами данных в PL/pgSQL.

### **Что такое переменная?**

Переменная в PL/pgSQL представляет собой именованную область памяти, предназначенную для хранения значений различных типов данных. Как и в любом другом языке программирования, переменные позволяют хранить промежуточные значения внутри функций, блоков BEGIN...END или триггеров.

#### Объявление переменных

Объявлять переменные в PL/pgSQL можно несколькими способами:

DECLARE

var\_name type DEFAULT value;

или

DECLARE

var\_name ALIAS FOR column\_name;

Здесь var\_name — имя переменной, type — её тип данных, а DEFAULT value задаёт начальное значение переменной.

**Пример объявления переменной:**

CREATE OR REPLACE FUNCTION example\_function()

RETURNS void AS $$

DECLARE

employee\_id integer := 100; -- объявление переменной типа integer

BEGIN

RAISE NOTICE 'Employee ID is %', employee\_id;

END;

$$ LANGUAGE plpgsql;

**Использование переменных:**Вы можете присвоить новое значение переменной следующим образом:

employee\_id := new\_value;

Также возможно использование оператора SELECT INTO для присваивания результата запроса переменным:

SELECT salary INTO emp\_salary FROM employees WHERE id = employee\_id;

#### Типы данных PL/pgSQL

Типы данных в PL/pgSQL совпадают с теми, что используются в самом PostgreSQL. Вот наиболее распространённые:

* **Integer**: целые числа (например, INT, SMALLINT, BIGINT).
* **Numeric**: десятичные дроби с фиксированной точностью (NUMERIC(precision, scale)).
* **Character types**: строки символов (CHAR(n), VARCHAR(n), TEXT).
* **Boolean**: булевские значения (true, false, null).
* **Date/time**: датировки и временные метки (DATE, TIMESTAMP, TIME).
* **Array**: массивы любых базовых типов.
* **Composite Types**: составные типы данных, состоящие из нескольких полей (аналог структуры).

Примеры задания разных типов переменных:

-- объявляем строковую переменную

first\_name text := 'John';

-- числовая переменная

salary numeric(8,2);

-- временная переменная

hire\_date timestamp with time zone;

#### Классификация переменных

Переменные делятся на два основных вида:

* **Скалярные переменные**: хранят одно значение определённого типа (число, строку, дату и т.п.).
* **Составные переменные**: включают другие переменные (например, записи базы данных, объекты структурированных типов). Такие переменные называются также RECORDS или ROWTYPES.

Например, запись из таблицы сотрудников:

emp\_record employees%ROWTYPE;

Это позволяет обращаться к полям такой записи как к отдельным переменным:

emp\_record.id := 100;

emp\_record.name := 'Alex';

### **Примеры работы с переменными на основе таблицы employees**

Предположим, мы имеем таблицу employees следующего формата:

|  |  |
| --- | --- |
| **Column** | **Type** |
| id | INTEGER |
| name | TEXT |
| position | TEXT |
| hire\_date | DATE |
| salary | NUMERIC |

**Задача 1: Напишите функцию, которая возвращает сумму зарплат всех сотрудников старше заданного возраста.**

Для начала создадим примерную функцию, использующую переменные:

CREATE OR REPLACE FUNCTION total\_salary\_by\_age(age\_in\_years integer)

RETURNS numeric AS $$

DECLARE

total\_salary numeric := 0;

BEGIN

SELECT SUM(salary) INTO total\_salary

FROM employees

WHERE EXTRACT(YEAR FROM AGE(hire\_date)) >= age\_in\_years;

RETURN total\_salary;

END;

$$ LANGUAGE plpgsql;

**Что делает этот код:**Мы объявили переменную total\_salary типа numeric. Затем используем агрегатную функцию SUM() для подсчёта суммы зарплаты всех сотрудников, чей стаж превышает указанный возраст. Результат записывается в переменную и возвращается функцией.

**Задача 2: Найти сотрудника с самой высокой зарплатой среди тех, кто работает дольше указанного количества месяцев.**

Реализуем функцию, возвращающую идентификатор самого высокооплачиваемого сотрудника, проработавшего больше N месяцев:

CREATE OR REPLACE FUNCTION find\_highest\_paid\_employee(min\_months integer)

RETURNS integer AS $$

DECLARE

highest\_paid\_emp\_id integer;

BEGIN

SELECT id INTO highest\_paid\_emp\_id

FROM employees

ORDER BY salary DESC, extract(year from current\_timestamp) - extract(year from hire\_date) DESC

LIMIT 1

OFFSET (SELECT COUNT(\*) FILTER (WHERE extract(month from age(current\_timestamp, hire\_date)) > min\_months));

RETURN highest\_paid\_emp\_id;

END;

$$ LANGUAGE plpgsql;

**Что делает эта функция:**Используя оператор ORDER BY, сортируем сотрудников сначала по зарплате, потом по стажу. Ограничение LIMIT 1 выводит первого сотрудника, а условие фильтра отсекает тех, кто работал меньше указанных месяцев.

### **Вопросы с подвохом по работе с переменными и типами PL/pgSQL**

1. **Вопрос:** Почему нельзя инициализировать переменную одного типа значением другого типа?**Ответ:** Каждый тип данных имеет свою структуру представления в памяти и операции над ним. Неправильное преобразование может привести к потере точности или ошибкам выполнения программы.
2. **Вопрос:** Зачем нужны составные переменные (RECORD)?**Ответ:** Составные переменные удобны для работы с записями таблиц базы данных, позволяя обратиться сразу ко всей строке целиком либо отдельно к каждому полю.
3. **Вопрос:** Можно ли изменить размер массива динамически в PL/pgSQL?**Ответ:** Да, это возможно. Например, используя команду array\_append().
4. **Вопрос:** Какие проблемы возникают при попытке сохранения NULL-значений в переменных BOOLEAN?**Ответ:** NULL-значение нарушает принцип двухвалентности логики, приводя к неопределённости при сравнении или вычислениях.
5. **Вопрос:** Может ли переменная иметь тип JSONB?**Ответ:** Да, начиная с версии PostgreSQL 9.4, поддерживаются JSON и JSONB как полноценные типы данных, включая возможность их хранения в переменных.
6. **Вопрос:** Какой метод используется для преобразования между разными типами данных в PL/pgSQL?**Ответ:** Оператор CAST, например: CAST(var\_name AS target\_type) или синтаксический сахар с использованием двойных двоеточий: var\_name::target\_type.
7. **Вопрос:** Чем отличаются составные переменные от обычных скалярных?**Ответ:** Скалярные переменные содержат одиночное значение конкретного типа, тогда как составные представляют объект с множеством полей, каждый из которых может быть отдельного типа.
8. **Вопрос:** Возможно ли создание временной переменной внутри блока PL/pgSQL?**Ответ:** Нет, переменные объявляются в блоке DECLARE и существуют только в пределах текущего блока или функции.
9. **Вопрос:** Есть ли ограничения на длину имени переменной?**Ответ:** Имя переменной должно начинаться с буквы и содержать максимум 63 символа.
10. **Вопрос:** В чём разница между типами BOOLEAN и BIT(1)? **Ответ:** BOOLEAN предназначен исключительно для операций true/false/null, тогда как BIT(1) — это битовая маска длиной один бит, принимающая значения '0' или '1'.

### **Другие команды PL/pgSQL**

Помимо управления переменными, PL/pgSQL поддерживает условные конструкции IF-THEN-ELSE, циклы LOOP, WHILE, цикл FOREACH для обхода элементов коллекций и ряд других возможностей. Все они дополняют инструментарий программиста при написании сложных хранимых процедур и функций.

#### Условные операторы:

IF condition THEN

-- выполнение действий

ELSIF another\_condition THEN

-- альтернативные действия

ELSE

-- финальное действие

END IF;

#### Циклы:

LOOP

-- тело цикла

EXIT WHEN condition;

END LOOP;

Таким образом, PL/pgSQL предлагает богатый набор инструментов для обработки и манипулирования данными, предоставляя гибкость и удобство разработки серверных

## **Условные операторы (Логика выполнения). Операторы "IF", "THEN", "ELSE", "ELSIF" и "END IF" PL/pgSQL**

PL/pgSQL предлагает полноценную поддержку условных конструкций, аналогичных операторам if–else в традиционных языках программирования.

### **Основные формы:**

* **IF THEN**: Простое условие.
* **IF THEN ELSE**: Выбор между двумя альтернативами.
* **IF THEN ELSIF THEN ELSE**: Несколько условий подряд.

## **Условные операторы (Логика выполнения). Операторы "IF", "THEN", "ELSE", "ELSIF" и "END IF"**

Условные операторы позволяют изменять последовательность выполнения программы в зависимости от определенных условий. Одним из важнейших аспектов программирования является способность организовать поток выполнения таким образом, чтобы решение могло варьироваться в зависимости от конкретных обстоятельств. В языке PL/pgSQL это реализуется посредством операторов IF, THEN, ELSE, ELSIF и завершающего блока END IF.

Давайте разберемся подробнее, как работают эти операторы и какие возможности они предоставляют.

### **Базовые понятия**

#### 1. IF

Ключевое слово IF запускает проверку условия. Если указанное после него условие истинно, выполняются последующие инструкции.

#### 2. THEN

Инструкция THEN обозначает начало блока инструкций, которые выполняются, если предыдущее условие истинно.

#### 3. ELSE

Ключевое слово ELSE определяет действия, которые будут выполнены, если первоначальное условие не выполняется.

#### 4. ELSIF

Если нужно проверить дополнительные условия, применяется оператор ELSIF. Это своего рода последовательная альтернатива ELSE IF, характерная для многих языков программирования.

#### 5. END IF

Завершающий блок END IF обозначает завершение секции условного оператора.

### **Общая структура условных операторов**

Вот общая схема условных операторов:

IF condition THEN

-- Выполняются инструкции, если условие истинно

ELSIF another\_condition THEN

-- Выполняются инструкции, если первое условие ложно, а второе истинно

ELSE

-- Выполняются инструкции, если ни одно из условий не было удовлетворено

END IF;

Эта структура напоминает аналогичные конструкции в других языках программирования, таких как JavaScript, Python или C++, но с небольшими отличиями, обусловленными спецификой PostgreSQL.

### **Основные моменты и правила**

#### Порядок проверки условий

Условия проверяются сверху вниз, и как только находится подходящее условие, дальнейшие проверки прекращаются. Поэтому важно располагать условия в правильном порядке.

#### Булевые значения

Любое условие в PL/pgSQL обязательно принимает булевое значение: TRUE, FALSE или NULL. Если условие возвращает NULL, поведение зависит от контекста:

* в операторе IF: ничего не произойдет (условие считается ложным),
* в операторе CASE: поведенческие особенности немного отличаются.

#### Работа с переменными

Значения переменных могут использоваться непосредственно в условиях. Например:

IF variable IS NOT NULL THEN ...

### **Детальные объяснения каждого элемента**

#### 1. Ключевое слово IF

Оператор IF инициирует проверку условия. После этого ключевого слова идет любое допустимое логическое выражение, результатом которого является булево значение.

**Пример простого условия:**

IF some\_variable > 10 THEN

RAISE NOTICE 'Переменная больше 10';

END IF;

Здесь условие some\_variable > 10 проверяется на истину. Если условие истинно, исполняется следующий блок инструкций.

#### 2. Блок THEN

Блок THEN отделяет само условие от последующих инструкций, которые будут исполнены, если условие выполнено успешно. Здесь размещают любую требуемую логику, такую как изменение переменных, запросы к БД, отправка уведомлений и т.д.

**Пример:**

IF some\_variable > 10 THEN

UPDATE table SET field = 'new\_value';

END IF;

Если условие истинно, обновление поля field будет произведено.

#### 3. Дополнительные проверки: ELSIF

Часто возникает необходимость проверить не одно, а несколько условий подряд. Для этого в PL/pgSQL предусмотрен оператор ELSIF, позволяющий добавлять дополнительные проверки последовательно.

**Пример:**

IF some\_variable > 10 THEN

RAISE NOTICE 'Переменная больше 10';

ELSIF some\_variable < 5 THEN

RAISE NOTICE 'Переменная меньше 5';

ELSE

RAISE NOTICE 'Переменная от 5 до 10';

END IF;

При каждом последующем применении оператора ELSIF проверяется очередное условие. Первое истинное условие останавливает дальнейшее тестирование остальных условий.

#### 4. Запасной выход: ELSE

Блок ELSE гарантирует, что хотя бы какая-то логика будет выполнена, даже если ни одно из предшествующих условий не соответствует действительности.

**Пример:**

IF some\_variable > 10 THEN

RAISE NOTICE 'Переменная больше 10';

ELSE

RAISE NOTICE 'Переменная меньше или равна 10';

END IF;

Если первая проверка провалилась, будет выполнена вторая часть блока ELSE.

#### 5. Завершительный блок: END IF

Каждый оператор IF должен заканчиваться соответствующим блоком END IF, который обозначает окончание условного оператора. Без этого окончания возникнет ошибка компиляции.

### **Важные нюансы и тонкости**

#### 1. Логические операторы

PL/pgSQL поддерживает стандартные логические операторы: AND, OR, NOT. Их применение упрощает формирование сложных условий:

IF some\_variable > 10 AND other\_variable IS NOT NULL THEN

-- сложное условие

END IF;

#### 2. Использование переменных

Переменные могут быть использованы прямо в теле условия:

IF employee\_salary > average\_salary THEN

RAISE NOTICE 'Зарплата выше средней';

END IF;

#### 3. Поведение с NULL

NULL ведет себя особым образом в логике условий. Например, вот простой пример:

IF null\_variable IS NULL THEN

RAISE NOTICE 'Переменная пустая';

END IF;

NULL рассматривается как отсутствующее значение, и любой тест на равенство с ним автоматически даёт ложь.

#### 4. Подсказка по оптимизации

Чтобы повысить производительность ваших скриптов, постарайтесь минимизировать количество обращений к данным и оптимально распределять нагрузки между условиями.

### **Практическая демонстрация**

Рассмотрим практическую реализацию с примером. Предположим, у вас есть таблица сотрудников (employees) с такими столбцами:

* id (идентификатор сотрудника)
* name (имя сотрудника)
* position (должность)
* salary (зарплата)
* hire\_date (дата найма)

#### Задача: Нужно определить категорию сотрудника исходя из стажа и зарплаты.

##### **Шаг 1: Определяем логику**

Категории сотрудников могут зависеть от следующих факторов:

* Зарплаты: выше среднего — опытный сотрудник, ниже среднего — младший сотрудник.
* Стажа: стаж более трех лет — стабильный сотрудник, менее трёх лет — новый сотрудник.

##### **Шаг 2: Реализация на PL/pgSQL**

DO $$

DECLARE

employee\_salary numeric := 50000; -- текущая зарплата

average\_salary numeric := 45000; -- средняя зарплата по организации

hire\_date date := CURRENT\_DATE - interval '2 years'; -- дата приема на работу

BEGIN

IF employee\_salary > average\_salary THEN

RAISE NOTICE 'Сотрудник опытный (%)', employee\_salary;

ELSE

RAISE NOTICE 'Сотрудник начинающий (%)', employee\_salary;

END IF;

IF extract(year from age(CURRENT\_DATE, hire\_date)) > 3 THEN

RAISE NOTICE 'Стабильный сотрудник';

ELSE

RAISE NOTICE 'Новый сотрудник';

END IF;

END $$;

### **Разбор примера:**

1. Сначала проверяется, выше ли зарплата сотрудника средней зарплаты. Если да, то выдается уведомление, что сотрудник опытный.
2. Далее определяется стаж сотрудника путём расчета разницы между сегодняшним днем и датой приёма на работу. Если стаж превышает три года, сотрудник помечается как стабильный.

### **Советы и лучшие практики**

1. Избегайте чрезмерного усложнения условий, поскольку длинные цепочки IF-ELSIF трудно читать и поддерживать.
2. Старайтесь выделять крупные блоки условных операторов в отдельные процедуры или функции, чтобы облегчить понимание и поддержку кода.
3. Всегда следите за правильной структурой вложенности операторов, особенно при наличии нескольких уровней вложенности.
4. Помните о важности комментариев для пояснений вашего кода другим разработчикам.

## **Циклы (Петли, Loops). "WHILE loops", "FOR loops"  PL/pgSQL**

### **Циклические конструкции:**

Циклы — это конструкции, позволяющие повторять определенный участок кода определенное количество раз или пока выполняется некоторое условие.

### **Два типа циклов:**

* **WHILE**: Повторяет цикл, пока истинно заданное условие.
* **FOR**: Проходит по списку значений (целых чисел, записей таблицы и т.п.) и выполняет заданные инструкции.

### **Пример цикла WHILE:**

DO $$

DECLARE

i INTEGER := 1;

BEGIN

WHILE i <= 5 LOOP

RAISE NOTICE 'Шаг %', i;

i := i + 1;

END LOOP;

END $$ LANGUAGE plpgsql;

**Пример цикла FOR:**

DO $$

DECLARE

i INTEGER;

BEGIN

FOR i IN 1..5 LOOP

RAISE NOTICE 'Элемент %', i;

END LOOP;

END $$ LANGUAGE plpgsql;

## **Петли и циклы в PL/pgSQL**

Циклические конструкции позволяют многократно повторять одни и те же фрагменты кода до тех пор, пока определенное условие остается истинным. Эти конструкции широко применяются в процедурах и функциях для автоматизации процессов и выполнения однотипных операций. PL/pgSQL предоставляет две основные разновидности циклов: цикл WHILE и цикл FOR.

### **Основная концепция циклов**

Циклы — это способ повторного выполнения одной или нескольких инструкций. Каждый цикл состоит из условия завершения и тела цикла, содержащего необходимые инструкции. Во время выполнения программа переходит к началу цикла снова и снова, пока условие не станет ложным.

В отличие от простых конструкций вроде IF, циклы продолжают свое выполнение до тех пор, пока определённое условие сохраняется.

### **Основы работы с циклом WHILE**

Цикл WHILE проверяет условие перед каждым новым проходом и продолжает выполнение, пока условие остаётся истинным. Его общий вид выглядит так:

WHILE condition LOOP

-- тело цикла

END LOOP;

**Порядок выполнения:**

1. Перед началом итерации проверяется условие (condition).
2. Если условие истинно, выполняется тело цикла.
3. По завершении выполнения тела вновь проверяется условие.
4. Процесс продолжается до тех пор, пока условие не перестанет быть истинным.

#### Пример использования цикла WHILE

Представим себе задачу: необходимо увеличивать зарплату сотрудника на 5% ежегодно, пока она не достигнет определенного предела. Используем таблицу employees с колонками:

* id (идентификатор сотрудника)
* name (имя сотрудника)
* salary (текущая зарплата)

Посчитаем новую зарплату сотрудника с увеличением ежегодного дохода до достижения максимальной границы в 100,000100,000.

DO $$

DECLARE

current\_salary numeric := 50000; -- Начальная зарплата

increase\_rate numeric := 0.05; -- Ежегодный процент увеличения

BEGIN

WHILE current\_salary < 100000 LOOP

current\_salary := current\_salary + (current\_salary \* increase\_rate); -- Увеличение зарплаты на 5%

RAISE NOTICE 'Новая зарплата: %. Текущая сумма: %', increase\_rate\*100 || '%', current\_salary;

END LOOP;

END $$;

**Что происходит:**

* До начала цикла устанавливается начальная зарплата равная 50,00050,000 долларов.
* Внутри цикла производится увеличение зарплаты на 5%.
* Цикл продолжается до тех пор, пока зарплата не превысит 100,000100,000.

### **Особенности цикла WHILE**

1. **Проверка условия**: В цикле WHILE условие проверяется **до входа в тело цикла**. Если изначально условие окажется ложным, тело цикла вообще не запустится.
2. **Риск бесконечного цикла**: Будьте осторожны, убедитесь, что ваше условие рано или поздно станет ложным, иначе вы получите бесконечный цикл.
3. **Изменение переменных**: Часто условием выхода из цикла становится изменение какой-нибудь переменной внутри тела цикла. В нашем примере новая зарплата постоянно меняется, приближаясь к границе.

### **Цикл FOR**

Другой распространенный тип цикла в PL/pgSQL — это цикл FOR. Этот цикл удобно использовать, когда заранее известно число итераций. Общий вид цикла FOR выглядит так:

FOR counter IN start..stop LOOP

-- тело цикла

END LOOP;

Где:

* counter — счётчик цикла (переменная, которая изменяется автоматически на каждой итерации).
* start — стартовое значение счётчика.
* stop — конечное значение счётчика.

Цикл выполняется ровно столько раз, сколько чисел содержится в диапазоне от start до stop.

#### Пример использования цикла FOR

Допустим, вам нужно увеличить заработную плату всех сотрудников на определенный процент, но сделать это постепенно, поэтапно увеличивая каждый год. Используя цикл FOR, мы можем легко реализовать подобное постепенное повышение.

DO $$

DECLARE

i integer; -- переменная счётчика

BEGIN

FOR i IN 1..5 LOOP -- пять этапов увеличения

UPDATE employees

SET salary = salary \* 1.05; -- увеличение на 5%

COMMIT; -- фиксация изменений

RAISE NOTICE 'Прошёл этап #%: Новая зарплата установлена.', i;

END LOOP;

END $$;

**Что происходит:**

* Задавая диапазон от 1 до 5, цикл пройдёт ровно 5 раз.
* На каждой итерации выполняется операция обновления заработной платы для всех сотрудников.
* После каждого этапа фиксируется состояние транзакции.

### **Особенности цикла FOR**

1. **Автоматизация счётчика**: В цикле FOR переменная-счётчик управляется системой автоматически. Вам не нужно вручную менять её значение.
2. **Интервал шага**: По умолчанию шаг изменения счётчика равен единице. Однако можно задать произвольный шаг, используя ключевое слово BY:

FOR i IN 1..10 BY 2 LOOP

-- Тело цикла

END LOOP;

Здесь цикл пробежит по чётным числам от 1 до 10.

1. **Регистрация состояний**: Удобно выводить промежуточные данные на каждом этапе цикла, чтобы отслеживать процесс выполнения.

### **Оптимизация и полезные советы**

1. **Оптимизируйте ваши циклы**: Хотя PL/pgSQL отлично справляется с большими объёмами данных, старайтесь минимизировать влияние тяжёлых операций внутри циклов.
2. **Комментарии и документация**: Добавляйте комментарии в ваш код, объясняющие назначение каждой части цикла. Особенно полезно это для больших проектов, работающих с критическими данными.
3. **Мониторинг производительности**: Используйте профилировщики и отчёты PostgreSQL для анализа затрат ресурсов и скорости выполнения ваших циклов.

### **Каверзные вопросы по теме циклов**

1. **Вопрос:** Что произойдёт, если условие цикла WHILE изначально ложно?**Ответ:** Тело цикла вообще не выполнится ни разу.
2. **Вопрос:** Что означает фраза «бесконечный цикл», и почему это опасно?**Ответ:** Бесконечным называется цикл, условие которого никогда не станет ложным. Такой цикл вызывает зависание программы, что требует особого внимания при разработке.
3. **Вопрос:** Можно ли изменить значение счётчика в цикле FOR вручную?**Ответ:** Нельзя. Значение счётчика управляется автоматически и защищено от ручных модификаций.
4. **Вопрос:** В каком месте цикла FOR задаётся шаг изменения счётчика?**Ответ:** Шаг указывается в конце определения диапазона с ключевым словом BY.
5. **Вопрос:** Можно ли запустить цикл без использования индекса (счётчика)?**Ответ:** В циклах PL/pgSQL индекс обязателен, так как цикл сам управляет изменением счётчика.
6. **Вопрос:** Существуют ли в PL/pgSQL встроенные механизмы прерывания цикла досрочно?**Ответ:** Да, механизм досрочного прекращения цикла доступен через команды EXIT или CONTINUE, позволяющие выйти из цикла или перейти к следующей итерации соответственно.
7. **Вопрос:** В чём принципиальное различие между циклами WHILE и FOR?**Ответ:** Основное различие заключается в том, что цикл WHILE ориентирован на выполнение до тех пор, пока условие истинно, а цикл FOR ограничен известным числом шагов.
8. **Вопрос:** Каково максимальное число итераций в цикле FOR?**Ответ:** Число итераций ограничено размером целочисленного типа, используемого для счётчика. Обычно это около миллиарда итераций, что вполне достаточно для большинства практических целей.
9. **Вопрос:** Что произойдёт, если поменять местами значения границ диапазона в цикле FOR?**Ответ:** Цикл просто не начнёт своё выполнение, потому что меньшее значение ставится слева, а большее справа.
10. **Вопрос:** Можно ли объединить несколько циклов в одном блоке кода?**Ответ:** Да, это абсолютно нормально и делается довольно часто. Главное следить за вложенностью и правильным закрытием циклов.

## **Курсоры (Cursors). Использование курсоров вместе с циклами PL/pgSQL**

### **Что такое курсор?**

Курсор — это объект, который позволяет построчно перебирать содержимое таблицы или запроса. Курсор напоминает указатель на конкретные строки, которые можно прочитать или обработать.

### **Типы курсоров:**

* **Implicit cursor**: Неявный курсор, созданный системой автоматически при выполнении инструкций SELECT.
* **Explicit cursor**: Явный курсор, который программист объявляет самостоятельно и управляет его поведением.

## Курсоры (Cursors)

### **Что такое курсор?**

**Курсор —** это объект базы данных PostgreSQL, позволяющий эффективно обрабатывать большие наборы строк выборки SQL-запросов последовательно, одну строку за другой. Курсор представляет собой своего рода указатель на набор результатов, возвращаемых запросом, и позволяет получать строки выборки порциями, уменьшая нагрузку на память сервера баз данных и клиента.

### **Зачем нужны курсоры?**

При работе с большими наборами данных обычная практика возврата всех записей сразу часто оказывается неэффективной, поскольку требует значительного объема памяти для хранения результата. Особенно критично это становится, когда обработчик данных имеет ограниченный объем оперативной памяти, например, клиентские приложения или веб-сервисы. Чтобы избежать такой проблемы, используются курсоры.

#### Основные преимущества курсоров:

* Позволяют пошагово обходить большой набор строк без необходимости загружать весь результат сразу.
* Обеспечивают экономию ресурсов серверной стороны, особенно если используется частичный доступ к данным.
* Помогают управлять ресурсами, обеспечивая возможность параллельной обработки больших объемов данных.

### **Типы курсоров**

В PostgreSQL поддерживаются два основных типа курсоров:

1. **Простые (нескроллируемые) курсоры**: позволяют перемещаться только вперед от начала набора данных, извлекая данные построчно или небольшими блоками.

DECLARE cursor\_name CURSOR FOR SELECT \* FROM employees ORDER BY employee\_id ASC;

1. **Скроллируемые курсоры**: предоставляют гибкость перемещения назад-вперед внутри набора данных, позволяя возвращаться к ранее извлеченным строкам.Скроллируемый курсор объявляется следующим образом:

DECLARE scroll\_cursor CURSOR SCROLL FOR SELECT \* FROM employees ORDER BY employee\_id DESC;

Кроме того, курсоры делятся на следующие категории по типу поведения:

* **Read-only (только чтение)**: курсор предназначен исключительно для чтения данных, изменения запрещены.
* **Updatable (обновляемый)**: предоставляет возможность обновлять строки непосредственно через курсор.

Также существуют дополнительные режимы курсора:

* **INSENSITIVE**: результаты фиксируются при открытии курсора и остаются неизменными независимо от изменений в таблице.
* **SENSITIVE**: отображаются любые изменения, внесённые в таблицу, пока курсор открыт.

### **Как работают курсоры?**

Работа с курсором состоит из нескольких этапов:

1. **Объявление курсора**. Определяем имя курсора и связанный с ним запрос (SELECT).

DECLARE emp\_cursor CURSOR FOR SELECT employee\_id, first\_name, last\_name FROM employees WHERE department = 'IT';

1. **Открытие курсора**. Запускаем выполнение запроса, связанного с курсором.

OPEN emp\_cursor;

1. **Перемещение и извлечение данных**. Извлекаем строки из курсора вручную или автоматически, используя цикл.Например, простой цикл в PL/pgSQL выглядит так:
2. FETCH NEXT IN emp\_cursor INTO variable\_list;
3. WHILE FOUND LOOP
4. -- обработка каждой строки
5. RAISE NOTICE '%', variable\_list;
6. FETCH NEXT IN emp\_cursor INTO variable\_list;
7. END LOOP;

CLOSE emp\_cursor;

1. **Закрытие курсора**. Завершаем работу с курсором и освобождаем ресурсы.

CLOSE emp\_cursor;

### **Пример использования курсора с циклом на практике**

Допустим, у вас есть таблица employees, содержащая поля:

CREATE TABLE employees (

employee\_id SERIAL PRIMARY KEY,

first\_name VARCHAR(50),

last\_name VARCHAR(50),

salary NUMERIC(10, 2)

);

Предположим, нам нужно увеличить зарплату каждого сотрудника на 10%, сохранив записи сотрудников с увеличением зарплаты. Для этого мы можем создать курсор и обработать каждую запись отдельно.

DO $$

DECLARE

emp\_rec RECORD;

BEGIN

-- Объявляем курсор

DECLARE emp\_cursor CURSOR FOR SELECT employee\_id, first\_name, last\_name, salary FROM employees;

-- Открываем курсор

OPEN emp\_cursor;

-- Циклический проход по результатам курсора

LOOP

FETCH emp\_cursor INTO emp\_rec;

EXIT WHEN NOT FOUND;

-- Увеличение зарплаты на 10%

UPDATE employees SET salary = salary \* 1.1 WHERE employee\_id = emp\_rec.employee\_id;

RAISE NOTICE 'Зарплата увеличена сотруднику % ', emp\_rec.first\_name || ' ' || emp\_rec.last\_name;

END LOOP;

-- Закрываем курсор

CLOSE emp\_cursor;

END $$;

### **Каверзные вопросы по курсорам с ответами**

1. **Что произойдет, если попытаться обновить строку через курсор, который объявлен как read-only?**
   * Ответ: Постгре выдаст ошибку, так как такие курсоры предназначены исключительно для чтения данных.
2. **Можно ли изменить порядок сортировки строк, используемых курсором, после открытия последнего?**
   * Ответ: Нет, изменение порядка невозможно после открытия курсора. Порядок фиксируется при создании курсора.
3. **Какой курсор лучше использовать, если нужно вернуть список отделов, отсортированных по алфавиту, и также иметь возможность вернуться обратно к предыдущему отделу?**
   * Ответ: Нужно использовать скроллируемый курсор.
4. **Какие ограничения накладывает работа с курсорами в транзакциях?**
   * Ответ: После завершения транзакции открытый курсор закрывается автоматически. Поэтому важно помнить, что курсоры имеют область видимости внутри одной транзакции.
5. **Как правильно закрыть курсор, если процедура была прервана досрочно?**
   * Ответ: Можно воспользоваться командой COMMIT или ABORT, которая закроет все открытые курсоры.
6. **Может ли один курсор одновременно использоваться несколькими пользователями?**
   * Ответ: Нет, каждый курсор привязан к одному соединению и сессии.
7. **Нужно ли закрывать курсор явно, если он больше не нужен?**
   * Ответ: Да, рекомендуется закрывать курсор явно, чтобы освободить ресурсы и предотвратить возможные блокировки таблиц.
8. **Почему иногда использование курсоров может привести к снижению производительности?**
   * Ответ: Если курсор используется неоптимально (например, выполняется массовое обновление большого числа строк), производительность системы может снизиться.
9. **Существуют ли альтернативы курсорам для небольших объёмов данных?**
   * Ответ: Да, можно использовать стандартные операторы SQL для загрузки всего набора данных разом в переменную или временную таблицу.
10. **Где хранятся результаты, полученные курсором?**
    * Ответ: Результаты сохраняются временно на стороне сервера PostgreSQL, однако реально читаемые строки поступают клиенту постепенно по мере выполнения операций FETCH.

### **Отличия курсоров от обычных запросов**

|  |  |  |
| --- | --- | --- |
| **Особенности** | **Обычные запросы** | **Курсоры** |
| **Способ возвращения данных** | Возвращают весь набор данных сразу | Возврат осуществляется поэтапно |
| **Использование памяти** | Требуют больше памяти для хранения полного набора данных | Используют меньше памяти, обрабатывая данные по очереди |
| **Типичные сценарии** | Простые операции с небольшим объемом данных | Работа с крупными массивами данных |
| **Поддержка итераций** | Нужны специальные конструкции для последовательного перебора данных | Интеграция с циклами для эффективного управления потоком данных |
| **Производительность** | Обычно быстрее для малых наборов данных | Более эффективная для крупных объемов данных благодаря возможности обработки порциями |

### **Другие аналогичные команды в PostgreSQL**

Помимо курсоров, в PostgreSQL существует ряд аналогичных инструментов для обработки больших наборов данных:

* **FOR LOOP**: конструкция цикла, встроенная в PL/pgSQL, позволяющая обойти результат любого запроса.
* **WITH RECURSIVE**: специальный оператор, используемый для рекурсивных запросов, помогает построить иерархические структуры.
* **RETURN QUERY**: возвращает результирующий набор, генерируя новый курсор динамически.

Однако курсоры обеспечивают большую гибкость и контроль над процессом извлечения данных, делая их незаменимыми инструментами при обработке сложных наборов данных в рамках процедурных расширений PL/pgSQL.

Таким образом, курсоры представляют собой мощный инструмент для разработчиков баз данных PostgreSQL, предоставляя возможность эффективной обработки огромных наборов данных с минимизацией нагрузки на систему и оптимизацией производительности приложений.

## **Обработка исключений (Exceptions) PL/pgSQL**

### **Основные термины:**

**Ошибка** — это событие, возникающее в ходе выполнения программы, которое нарушает нормальный ход её работы.

**Исключение** — процедура, реагирующая на возникновение ошибки и принимающая меры для восстановления нормального хода программы.

### **Теория обработки исключений:**

PL/pgSQL позволяет перехватывать ошибки и исключения с помощью специальной конструкции EXCEPTION. Если программа сталкивается с проблемой, обработчик ловит её и предпринимает дальнейшие шаги (например, выводит сообщение об ошибке или восстанавливает состояние программы).

## Обработка исключений (EXCEPTIONS) в PL/pgSQL

### **Введение**

PL/pgSQL поддерживает обработку ошибок и исключительных ситуаций с использованием блока EXCEPTION. Этот механизм позволяет разработчику контролировать возникновение непредвиденных проблем, возникающих при выполнении SQL-команд или операторов, таких как нарушение ограничений целостности, синтаксические ошибки, деление на ноль и другие ситуации, приводящие к ошибкам.

### **Почему важна обработка исключений?**

Когда возникает ошибка в ходе выполнения SQL-кода, стандартный механизм PostgreSQL прерывает выполнение текущего блока, отменяя все незавершенные изменения и выводя сообщение об ошибке. Однако в некоторых случаях такая реакция нежелательна. Часто бывает полезно перехватывать возникшую ошибку, обрабатывать её соответствующим образом (например, записывая ошибку в журнал, уведомляя администратора, исправляя проблему и продолжая выполнение программы).

Именно тут вступает в игру механизм обработки исключений, позволяющий реализовать следующий сценарий:

* Ошибка возникает → перехватывается вашим кодом → предпринимаете необходимые действия → возобновляете нормальное выполнение скрипта.

Это значительно повышает надежность и устойчивость ваших процедур, функций и триггеров.

### **Структура блока EXCEPTION**

Общая структура блока обработки исключения в PL/pgSQL выглядит следующим образом:

BEGIN

-- основной блок вашего кода

EXCEPTION

WHEN exception\_condition THEN

-- ваш обработчик данной конкретной ошибки

WHEN another\_exception THEN

-- обработчик другой ошибки

...

END;

Здесь:

* exception\_condition: условие возникновения ошибки (определённое стандартом SQL или вашей собственной классификацией).
* Код между ключевыми словами WHEN и THEN — это обработчик ошибки, содержащий инструкции, выполняемые при возникновении указанного условия.

### **Примеры условий ошибок в PostgreSQL**

PostgreSQL поддерживает различные классы стандартных ошибок SQL, определённых стандартом ISO/IEC 9075 (SQL Standard):

* UNIQUE\_VIOLATION: Нарушение уникального ключа.
* FOREIGN\_KEY\_VIOLATION: Нарушение внешнего ключа.
* NOT\_NULL\_VIOLATION: Попытка вставить NULL-значение в столбец, который обязан содержать значение.
* CHECK\_VIOLATION: Нарушено ограничение CHECK.
* DATA\_EXCEPTION: Общая категория исключений, связанных с некорректностью вводимых данных.
* NUMERIC\_VALUE\_OUT\_OF\_RANGE: Число выходит за допустимые пределы диапазона.
* DIVISION\_BY\_ZERO: Деление на ноль.
* INTERNAL\_ERROR: Внутренняя ошибка сервера.
* INVALID\_TRANSACTION\_STATE: Транзакционное состояние нарушено.
* STATEMENT\_COMPLETION\_UNKNOWN: Результат выполнения оператора неизвестен.

Вы можете добавить собственные идентификаторы исключений с помощью специальных инструкций (см. ниже).

### **Блок обработки исключений в действии**

Рассмотрим пример процедуры, которая пытается вставить нового сотрудника в таблицу employees, учитывая потенциальные нарушения ограничений:

-- Таблица employees с ограничением unique на email

CREATE TABLE employees (

id SERIAL PRIMARY KEY,

first\_name TEXT NOT NULL,

last\_name TEXT NOT NULL,

email TEXT UNIQUE NOT NULL

);

-- Процедура добавления сотрудника с обработкой исключений

CREATE OR REPLACE PROCEDURE add\_employee(first\_name\_in text, last\_name\_in text, email\_in text)

LANGUAGE plpgsql AS $$

BEGIN

INSERT INTO employees (first\_name, last\_name, email) VALUES (first\_name\_in, last\_name\_in, email\_in);

EXCEPTION

WHEN UNIQUE\_VIOLATION THEN

RAISE NOTICE 'Ошибка! Сотрудник с таким e-mail (%s) уже существует.', email\_in;

WHEN OTHERS THEN

RAISE NOTICE 'Необработанная ошибка: %', SQLERRM;

END;

$$;

CALL add\_employee('Иван', 'Иванов', 'ivan@example.com');

CALL add\_employee('Анна', 'Петрова', 'anna@example.com');

CALL add\_employee('Сергей', 'Сергеев', 'ivan@example.com'); -- Повторяем существующий email

В данном примере:

* Если сотрудник с указанным адресом электронной почты уже существует (условие UNIQUE\_VIOLATION), появится уведомление об ошибке.
* Любая другая ошибка (все остальные случаи) приведет к выводу общего уведомления о произошедшей необработанной ошибке.

### **Перехват произвольных ошибок с помощью RAISE**

В дополнение к встроенным условиям исключений, вы можете самостоятельно инициировать исключение, используя команду RAISE.

Пример:

BEGIN

IF salary < 0 THEN

RAISE EXCEPTION 'Заработная плата должна быть положительной!';

ELSE

-- Продолжаем нормальную обработку

END IF;

END;

Здесь программа сама вызывает исключение, если зарплата отрицательная, предотвращая внесение некорректных данных.

### **Дополнительные возможности обработки исключений**

PostgreSQL предлагает дополнительную функциональность для подробного анализа ошибок:

* **SQLSTATE**: Стандартизированный пятизначный код состояния SQL, обозначающий конкретный класс ошибки.
* **SQLERRM**: Сообщение об ошибке, сформированное системой PostgreSQL.

Эти элементы помогают определить точную причину сбоя и выбрать наиболее подходящий способ обработки ошибки.

Например:

EXCEPTION

WHEN others THEN

RAISE NOTICE 'Код ошибки: %', SQLSTATE;

RAISE NOTICE 'Сообщение об ошибке: %', SQLERRM;

END;

Эта информация полезна для диагностики и устранения неполадок.

### **Альтернативы обработки исключений в PostgreSQL**

Хотя блоки EXCEPTION являются основным инструментом обработки ошибок в PL/pgSQL, существуют и другие подходы:

* **IF-THEN конструкции**: Вместо блока обработки исключений можно проверять условия перед выполнением потенциально опасных действий.
* **Флаги**: Используйте флаги, чтобы отслеживать успешность выполнения различных шагов в вашем коде.
* **Проверочные запросы**: Выполняйте предварительные проверки данных, например, проверка наличия дубликатов.

Однако полноценный механизм обработки исключений предоставляет гораздо больший уровень контроля и безопасности.

### **10 каверзных вопросов по обработке исключений с ответами**

1. **Какова разница между обработчиками исключений и проверочными конструкциями (например, IF)?**
   * Ответ: Обработчики исключений срабатывают только тогда, когда возникла реальная ошибка, в то время как конструкции IF позволяют проверить условия заранее, не дожидаясь появления ошибки.
2. **Зачем вообще нужны обработчики исключений, если можно просто избегать ошибок путём проверок?**
   * Ответ: Иногда ошибки неизбежны (например, аппаратные сбои, внешние зависимости); кроме того, даже тщательные проверки не всегда гарантируют отсутствие ошибок.
3. **Что произойдёт, если вызвать процедуру, которая завершается исключением, внутри другого блока обработки исключений?**
   * Ответ: Исключение поднимется вверх по стеку вызовов, и если оно перехвачено внешним блоком, будет выполнен соответствующий обработчик.
4. **Существует ли возможность перехватить абсолютно любую ошибку?**
   * Ответ: Да, условие OTHERS обеспечивает захват любых неопределённых ошибок.
5. **Будет ли ошибка повторно вызвана, если блок обработки исключений завершён успешно?**
   * Ответ: Нет, раз уж ошибка обработана, она считается разрешённой и не повторяется.
6. **Возможно ли восстановить исходное состояние транзакции после обработки ошибки?**
   * Ответ: Это зависит от контекста. Вы можете откатить транзакцию целиком (ROLLBACK), либо продолжить выполнение текущего блока, оставив произведённые изменения в силе.
7. **Какие типы исключений считаются стандартными в PostgreSQL?**
   * Ответ: Упомянутые выше (UNIQUENESS\_VIOLATION, FOREIGN\_KEY\_VIOLATION и др.) являются частью стандарта SQL и реализованы в PostgreSQL.
8. **Могу ли я создавать собственные классы исключений?**
   * Ответ: Да, можно вызывать свои исключения с помощью конструкции RAISE EXCEPTION.
9. **Возникает ли проблема конкуренции, если два процесса пытаются записать одно и то же значение одновременно?**
   * Ответ: Такая ситуация возможна, и её нужно учитывать при проектировании. Используйте соответствующие механизмы защиты, такие как блокировка строк или сериализация транзакций.
10. **Использует ли PostgreSQL механизм транзакций для восстановления согласованности после ошибки?**
    * Ответ: Да, если ошибка привела к недопустимому состоянию, вся текущая транзакция откатится автоматически, восстанавливая целостность данных.

## **Процедуры (Procedure). Создание, удаление и вызов. Просмотр информации о процедурах PL/pgSQL**

### **Что такое процедуры?**

Процедура — это отдельная единица программы, которая выполняет определенные действия и не возвращает никакого значения. Процедуры часто используются для автоматизации часто повторяющихся операций, интеграции с внешними приложениями или ведения журналов аудита.

## **Процедуры PostgreSQL**

Процедуры — это блоки SQL-кода, объединённые в одну именованную структуру, которую можно вызывать многократно из разных мест базы данных. Их основное отличие от функций заключается в том, что процедуры предназначены исключительно для выполнения определённых действий над базой данных и **не возвращают значений**.

### **Основные понятия:**

#### Отличия процедур от функций:

* **Процедура**: используется для выполнения операций и изменения состояния базы данных (например, изменение записей, выполнение нескольких запросов последовательно), но не возвращает никаких результатов наружу.
* **Функция**: предназначена для вычисления результата и возвращения значения обратно вызывающей стороне. Может изменять состояние базы данных, но обязательно должна вернуть какое-то значение.

### **Как создать процедуру?**

Для создания процедуры применяется специальный синтаксис PL/pgSQL (CREATE PROCEDURE):

CREATE OR REPLACE PROCEDURE имя\_процедуры(

список\_параметров

)

LANGUAGE plpgsql AS $$

BEGIN

-- Тело процедуры

END;

$$;

Пример процедуры, создающей новую запись в таблице employees:

-- Создаем таблицу сотрудников

CREATE TABLE IF NOT EXISTS employees (

id SERIAL PRIMARY KEY,

name VARCHAR(100),

salary NUMERIC(10, 2)

);

-- Процедура добавления нового сотрудника

CREATE OR REPLACE PROCEDURE add\_employee(

emp\_name VARCHAR(100),

emp\_salary NUMERIC(10, 2)

)

LANGUAGE plpgsql AS $$

BEGIN

INSERT INTO employees(name, salary) VALUES(emp\_name, emp\_salary);

END;

$$

Здесь:

- Параметры передаются процедуре внутри скобок (`emp\_name`, `emp\_salary`) и используются в теле процедуры.

- Используется блок BEGIN...END для группировки операторов.

- После ключевого слова LANGUAGE указывается название используемого языка программирования (PL/pgSQL).

---

### Удаление процедуры

Удалять процедуру можно командой DROP PROCEDURE:

```sql

DROP PROCEDURE IF EXISTS имя\_процедуры;

Это удаляет процедуру, если она существует, иначе ничего не произойдет.

Например:

DROP PROCEDURE IF EXISTS add\_employee;

### **Вызов процедуры**

Чтобы вызвать процедуру, используется оператор CALL:

CALL имя\_процедуры(список аргументов);

Примеры вызова ранее созданной процедуры:

CALL add\_employee('Иван Иванов', 50000);

CALL add\_employee('Анна Петрова', 70000);

### **Информация о процедурах**

Посмотреть существующие процедуры можно несколькими способами:

1. Использовать команду \df+ в psql консоли:

\df+

Эта команда выводит список всех хранимых процедур и функций с дополнительной информацией.

1. Посмотреть метаданные процедур в специальной схеме information\_schema. Например, посмотреть созданные нами процедуры:

SELECT routine\_name FROM information\_schema.routines WHERE specific\_schema = 'public';

### **Примеры усложненных процедур**

Рассмотрим создание более сложной процедуры, которая проверяет наличие записи и обновляет её, если сотрудник уже есть в базе данных:

CREATE OR REPLACE PROCEDURE update\_or\_create\_employee(

IN emp\_id INT,

IN emp\_name VARCHAR(100),

IN emp\_salary NUMERIC(10, 2)

)

LANGUAGE plpgsql AS $$

DECLARE

existing\_record RECORD;

BEGIN

-- Проверяем существование сотрудника

SELECT \* INTO existing\_record FROM employees WHERE id = emp\_id;

IF FOUND THEN

UPDATE employees SET name=emp\_name, salary=emp\_salary WHERE id = emp\_id;

ELSE

INSERT INTO employees(id, name, salary) VALUES(emp\_id, emp\_name, emp\_salary);

END IF;

END;

$$

Теперь вызовем эту процедуру:

```sql

CALL update\_or\_create\_employee(1, 'Сергей Сергеев', 80000); -- Обновляем существующую запись

CALL update\_or\_create\_employee(2, 'Дмитрий Дмитриев', 60000); -- Добавляем новую запись

### **Каверзные вопросы с ответами**

1. **Вопрос:** Чем отличается процедура от триггерной функции?**Ответ:** Процедура выполняется вручную посредством команды CALL, тогда как триггерная функция запускается автоматически при наступлении события (INSERT, UPDATE, DELETE и др.).
2. **Вопрос:** Можно ли передать в процедуру данные типа JSON?**Ответ:** Да, можно передавать любые типы данных, включая JSON. Однако потребуется правильно обрабатывать такие данные внутри тела процедуры.
3. **Вопрос:** Что произойдёт, если попытаться вернуть значение из процедуры?**Ответ:** Попытка возврата значения приведет к ошибке компиляции, поскольку процедуры не предназначены для возврата данных. Для возврата значений используют функции.
4. **Вопрос:** Почему важно избегать транзакций внутри процедур?**Ответ:** Внутри процедуры нельзя управлять внешними транзакциями, потому что транзакционная обработка определяется уровнем внешнего вызова. Если необходимо управлять транзакциями, лучше сделать это снаружи процедуры.
5. **Вопрос:** Какие ещё языки программирования поддерживаются PostgreSQL кроме PL/pgSQL?**Ответ:** Помимо PL/pgSQL, PostgreSQL поддерживает также Python (PL/Python), JavaScript (PL/V8), Perl (PL/Perl), Lua (PL/Lua) и другие языки.
6. **Вопрос:** Зачем нужен параметр INOUT в параметрах процедуры?**Ответ:** Этот параметр позволяет одновременно получать входные данные и отправлять обработанный результат назад вызывающему коду.
7. **Вопрос:** Как предотвратить рекурсивный вызов одной и той же процедуры?**Ответ:** Обычно достаточно следить за структурой вызовов и использовать переменную-флаг внутри процедуры, которая бы отслеживала глубину вложенности вызовов.
8. **Вопрос:** Нужно ли создавать отдельную схему для хранения процедур?**Ответ:** Нет, необязательно. Процедуры можно размещать непосредственно в основной схеме приложения, однако выделение отдельной схемы повышает удобство управления и разделения ответственности между компонентами.
9. **Вопрос:** Возможно ли написать один и тот же код как функцию и как процедуру?**Ответ:** Теоретически да, однако смысл процедуры именно в отсутствии возврата данных, тогда как функция обязана возвращать хотя бы одно значение.
10. **Вопрос:** Обязательно ли указывать аргумент LANGUAGE plpgsql при создании процедуры?**Ответ:** Да, обязательно. Это определяет, какой интерпретатор будет использоваться для обработки вашей процедуры.

## **Функции (Function). Создание, удаление и вызов. Просмотр информации о процедурах PL/pgSQL**

### **Что такое функции?**

Функция — это объект, который принимает входные параметры, выполняет определенные действия и возвращает результат. Функции обычно используются для вычислений, манипуляций с данными и возвращения готового результата.

Функции — важнейший инструмент базы данных PostgreSQL, позволяющий многократно использовать код и автоматизировать выполнение сложных операций над данными. Рассмотрим подробнее особенности функций в PL/pgSQL и разберем практическое применение на примере таблицы employees.

### **Что такое функция?**

Функция представляет собой блок SQL-кода, который выполняется одним оператором при вызове с определёнными аргументами. После её создания функция доступна из любого места базы данных и позволяет повысить производительность разработки приложений благодаря сокращению повторений кода.

### **Преимущества использования функций:**

* Возможность многократного повторного использования одной процедуры обработки данных.
* Улучшение читаемости и структуры программы.
* Ускорение отладки и исправления ошибок.
* Повышение производительности запросов путем предварительного компилирования кода.

### **Типы функций в PostgreSQL**

PostgreSQL поддерживает следующие типы функций:

#### 1. Императивные функции (IMMEDIATE)

Такие функции выполняются немедленно при вызове и возвращают результат сразу же. Обычно используются для простых вычислений или выборок данных.

Пример простой функции:

CREATE OR REPLACE FUNCTION employee\_salary(emp\_id integer)

RETURNS numeric AS $$

BEGIN

RETURN (

SELECT salary FROM employees WHERE id = emp\_id

);

END;

$$ LANGUAGE plpgsql;

Эта функция принимает идентификатор сотрудника и возвращает его зарплату.

#### 2. Функция-процедуры (PROCEDURE)

Это разновидность функций, выполняющих набор действий (обычно вставка, обновление или удаление), но не обязательно возвращающих значения.

Пример процедуры:

CREATE PROCEDURE update\_employee\_name(

IN p\_emp\_id INTEGER,

IN new\_firstname VARCHAR,

IN new\_lastname VARCHAR

)

LANGUAGE plpgsql

AS $$

BEGIN

UPDATE employees SET first\_name = new\_firstname, last\_name = new\_lastname WHERE id = p\_emp\_id;

END;

$$

Процедура обновляет имя и фамилию сотрудника с заданным ID.

#### 3. Асинхронные функции (ASYNC)

Эти функции запускаются асинхронно и продолжают выполняться даже после завершения основной транзакции. Применяются редко и требуют особых условий среды.

#### 4. Специальные функции (TRIGGER, RULE):

Эти функции работают автоматически при наступлении определенного события (например, вставке новой строки).

### **Основные команды управления функциями**

Рассмотрим самые важные команды, относящиеся к созданию, просмотру и удалению функций.

#### Команда CREATE FUNCTION

Используется для объявления новой функции.

**Общий синтаксис:**

CREATE [OR REPLACE] FUNCTION name([argtype[, …]]) RETURNS returntype AS $$

...

$$ LANGUAGE langname;

Например:

CREATE FUNCTION total\_sales\_by\_dept(dept\_id INT)

RETURNS NUMERIC AS $$

SELECT SUM(sales\_amount) FROM sales WHERE department\_id = dept\_id;

$$ LANGUAGE sql;

Функция получает отдел и возвращает сумму продаж по этому отделу.

#### Команда DROP FUNCTION

Удаляет существующую функцию из базы данных.

**Общий синтаксис:**

DROP FUNCTION IF EXISTS function\_name(argument\_type);

Пример удаления функции:

DROP FUNCTION IF EXISTS total\_sales\_by\_dept(INT);

#### Команда ALTER FUNCTION

Изменяет свойства существующей функции, такие как права доступа, безопасность или оптимизацию.

**Общий синтаксис:**

ALTER FUNCTION function\_name(argument\_type) RENAME TO new\_function\_name;

Или изменение безопасности:

ALTER FUNCTION total\_sales\_by\_dept(INT) SECURITY DEFINER;

### **Особенности реализации функций на PL/pgSQL**

PL/pgSQL — процедурный язык расширения SQL, специально разработанный для PostgreSQL. Этот язык позволяет создавать мощные функции с циклами, условиями и обработкой исключений.

Основные конструкции PL/pgSQL:

* Блок BEGIN…END
* Операторы ветвления (IF, CASE)
* Циклы (FOR, WHILE, LOOP)
* Обработка исключений (EXCEPTION WHEN ...)

Простой пример функции на PL/pgSQL:

CREATE FUNCTION count\_employees\_in\_dept(dept\_id int)

RETURNS integer AS $$

DECLARE

cnt integer := 0;

BEGIN

SELECT COUNT(\*) INTO cnt FROM employees WHERE department\_id = dept\_id;

RETURN cnt;

END;

$$ LANGUAGE plpgsql;

### **Отличия между функцией и процедурой**

|  |  |  |
| --- | --- | --- |
| **Критерий** | **Функция** | **Процедура** |
| Возвращаемое значение | Да | Нет |
| Вычисление результата | Обязательно | Опционально |
| Использование | Может использоваться в выражениях и запросах | Только исполняемый оператор |
| Порядок исполнения | Выполняется мгновенно | Запускается отдельной командой |

Проще говоря, функция предназначена для возврата значений, тогда как процедура больше ориентирована на выполнение набора действий без обязательного возвращения какого-то конкретного результата.

### **Примеры сложных функций на PL/pgSQL**

#### Пример 1: Сложная обработка данных с возвратом множества строк

Допустим, нам нужно вернуть всех сотрудников, зарплата которых превышает среднюю зарплату отдела.

CREATE FUNCTION top\_paid\_employees()

RETURNS TABLE(first\_name text, last\_name text, salary numeric) AS $$

BEGIN

RETURN QUERY

WITH avg\_sal AS (

SELECT AVG(salary) as average\_salary FROM employees GROUP BY department\_id

)

SELECT e.first\_name, e.last\_name, e.salary

FROM employees e JOIN avg\_sal ON e.department\_id = avg\_sal.department\_id

WHERE e.salary > avg\_sal.average\_salary;

END;

$$ LANGUAGE plpgsql;

Запрос вернёт таблицу с сотрудниками, чья заработная плата выше средней зарплаты своего отдела.

#### Пример 2: Логирование изменений с использованием триггерной функции

Создадим простую функцию-триггер, которая записывает изменения в журнал аудита.

CREATE TRIGGER log\_changes\_trigger

AFTER INSERT OR UPDATE OR DELETE ON employees

FOR EACH ROW EXECUTE FUNCTION audit\_log();

CREATE FUNCTION audit\_log() RETURNS trigger AS $$

BEGIN

IF TG\_OP = 'INSERT' THEN

INSERT INTO audit\_log VALUES ('INSERT', NEW.id, NOW());

ELSIF TG\_OP = 'UPDATE' THEN

INSERT INTO audit\_log VALUES ('UPDATE', OLD.id, NOW());

ELSE

INSERT INTO audit\_log VALUES ('DELETE', OLD.id, NOW());

END IF;

RETURN NULL;

END;

$$ LANGUAGE plpgsql;

Здесь используется триггерная функция, которая регистрирует каждое событие обновления данных в таблице журнала аудита.

### **Каверзные вопросы по функциям PostgreSQL**

1. **Вопрос:** Какова разница между функцией и процедурой в PostgreSQL?**Ответ:** Функция должна возвращать значение, тогда как процедура может лишь выполнять операции без обязательности возврата результата.
2. **Вопрос:** Можно ли внутри функции вызвать другую функцию?**Ответ:** Да, функции в PostgreSQL поддерживают рекурсивность и вложенность.
3. **Вопрос:** Чем отличается создание функции через команду CREATE FUNCTION от оператора CREATE PROCEDURE?**Ответ:** Разница заключается в семантике: функции возвращают результат, процедуры выполняют действия.
4. **Вопрос:** Какие ограничения существуют на количество аргументов функции?**Ответ:** Ограничения зависят от версии PostgreSQL, но стандартно поддерживаются десятки аргументов.
5. **Вопрос:** Какой самый быстрый способ удалить все созданные функции в базе данных?**Ответ:** Удалять каждую функцию вручную или создать скрипт динамического удаления с применением метаданных pg\_proc.
6. **Вопрос:** Почему иногда лучше использовать процедуру вместо функции?**Ответ:** Если операция требует сложной последовательности шагов и не предполагает возвращение результата, удобнее использовать процедуру.
7. **Вопрос:** Возможно ли изменить порядок выполнения операторов внутри функции?**Ответ:** Да, операторы можно переупорядочить путём добавления блоков условия или цикла.
8. **Вопрос:** Поддерживает ли PostgreSQL перегрузку функций?**Ответ:** Да, поддержка возможна через различные сигнатуры и аргументы.
9. **Вопрос:** Возможна ли передача переменных внутрь функции по ссылке?**Ответ:** Нет, все передачи осуществляются строго по значению.
10. **Вопрос:** Нужно ли устанавливать режим SECURITY DEFINER для функции?**Ответ:** Устанавливайте этот режим, если хотите обеспечить доступ к данным вне зависимости от прав текущего пользователя.

### **Заключение**

Функции и процедуры являются мощнейшими инструментами разработчика баз данных PostgreSQL. Они позволяют значительно упростить работу с большими объемами данных, избежать дублирования кода и реализовать бизнес-логику прямо в самой СУБД. Правильное использование функций повышает стабильность системы и облегчает сопровождение проектов.

Понимание разницы между простыми и сложными конструкциями, знание особенностей выполнения функций и процедур помогает эффективно проектировать приложения и управлять производительностью серверов баз данных.

## **Пакеты (Package). Создание, удаление и вызов процедур и функций из пакета. Просмотр информации о процедурах PL/pgSQL**

### **Что такое пакеты?**

Пакет — это особая структура, объединяющая процедуры, функции и вспомогательные переменные в единое целое. Пакеты упрощают организацию кода, улучшают его управляемость и уменьшают дублирование.

### **Компоненты пакета:**

* **Спецификация пакета**: описывает публичные элементы (процедуры, функции, глобальные переменные).
* **Тело пакета**: реализует логику публичных компонентов и содержит приватные элементы.

### **Пример спецификации и тела пакета:**

CREATE PACKAGE my\_package AS

PROCEDURE proc1();

FUNCTION func1() RETURNS INTEGER;

END my\_package;

CREATE PACKAGE BODY my\_package AS

PROCEDURE proc1() IS BEGIN ... END;

FUNCTION func1() RETURNS INTEGER IS BEGIN ... END;

END my\_package;

### **Преимущества пакетов:**

* Упрощение поддержки и развития больших приложений.
* Сокращение дублирования кода за счет повторного использования частных компонентов.
* Улучшенная безопасность и контроль доступа (за счет публичной и частной областей).

### **Что такое пакеты?**

**Пакеты** — это механизм организации кода SQL и PL/pgSQL таким образом, чтобы процедуры, функции и другие объекты были объединены в один объект базы данных. Подобная группировка позволяет лучше структурировать код, облегчает управление объектами и повышает производительность за счёт повторного использования кода.

PostgreSQL не поддерживает стандартные **пакеты**, как это реализовано в Oracle, однако существует альтернативный способ достижения похожего результата через использование схем (schemas), расширений и внешних библиотек. Например, пакет в PostgreSQL можно рассматривать как объединение нескольких связанных объектов внутри одной схемы.

#### Преимущества пакетов:

* **Структурирование**: Все связанные элементы находятся вместе.
* **Безопасность**: Ограничение прав доступа упрощается благодаря ограничению области видимости объектов пакета.
* **Производительность**: Упрощённая структура уменьшает количество зависимых компонентов и снижает нагрузку на систему.

#### Недостатки пакетов:

* **Отсутствие встроенной поддержки**. PostgreSQL не имеет полноценной реализации понятия "пакет" в стиле Oracle.
* **Необходимость ручного управления доступом**: Доступ к объектам внутри пакета регулируется вручную, путем назначения привилегий на уровне отдельных объектов.

### **Создание, удаление и вызов процедур и функций из пакета**

Так как PostgreSQL не предлагает прямой эквивалент пакету, мы можем реализовать подобную функциональность путём объединения элементов в схему. Рассмотрим создание и работу с такой структурой на примере таблиц employees.

#### Таблица Employees

CREATE TABLE employees (

id SERIAL PRIMARY KEY,

first\_name VARCHAR(50),

last\_name VARCHAR(50),

department\_id INT REFERENCES departments(id)

);

Допустим, мы хотим создать набор процедур и функций для управления сотрудниками, объединённых в одну схему (hr\_package).

#### Шаг 1: Создаем схему

CREATE SCHEMA hr\_package;

SET search\_path TO hr\_package, public;

#### Шаг 2: Добавляем необходимые права

Чтобы ограничить доступ, назначаем соответствующие роли и права на схему:

GRANT USAGE ON SCHEMA hr\_package TO some\_user\_role;

REVOKE ALL PRIVILEGES ON ALL TABLES IN SCHEMA hr\_package FROM PUBLIC;

#### Шаг 3: Процедура добавления сотрудника

Создадим процедуру для вставки нового сотрудника:

CREATE OR REPLACE FUNCTION add\_employee(firstname text, lastname text, deptid int)

RETURNS void AS $$

BEGIN

INSERT INTO employees(first\_name, last\_name, department\_id)

VALUES(firstname, lastname, deptid);

END;

$$ LANGUAGE plpgsql;

#### Шаг 4: Функция для изменения отдела сотрудника

Теперь создадим функцию для обновления отдела конкретного сотрудника:

CREATE OR REPLACE FUNCTION change\_department(emp\_id int, new\_dept\_id int)

RETURNS void AS $$

BEGIN

UPDATE employees SET department\_id = new\_dept\_id WHERE id = emp\_id;

END;

$$ LANGUAGE plpgsql;

#### Шаг 5: Удаляем сотрудника

Создаём процедуру удаления сотрудников:

CREATE OR REPLACE PROCEDURE delete\_employee(emp\_id integer)

LANGUAGE plpgsql

AS $$

BEGIN

DELETE FROM employees WHERE id = emp\_id;

END;

$$

#### Шаг 6: Выполнение операций

Выполняем операции через созданные функции и процедуры:

SELECT add\_employee('Иван', 'Иванов', 1); -- добавить сотрудника

SELECT change\_department(1, 2); -- изменить отдел сотруднику

CALL delete\_employee(1); -- удалить сотрудника

### **Получение информации о процедурах и функциях**

Просмотреть список всех процедур и функций можно с использованием системных представлений PostgreSQL:

-- Информация обо всех процедурах и функциях

SELECT proname, prosrc

FROM pg\_proc

WHERE pronamespace::regnamespace::text = 'hr\_package';

-- Информацию о параметрах конкретной функции

SELECT \*

FROM information\_schema.parameters

WHERE specific\_schema = 'hr\_package' AND specific\_name = 'add\_employee';

### **Каверзные вопросы с ответами:**

1. **Вопрос:** Как организовать доступ к процедурам и функциям из разных пользователей в схеме hr\_package, чтобы обеспечить безопасность?**Ответ:** Назначьте права отдельно каждому пользователю или группе пользователей с помощью команды GRANT, например: GRANT EXECUTE ON FUNCTION hr\_package.add\_employee(text,text,int) TO user\_role. Это позволит контролировать кто именно сможет вызывать ваши процедуры и функции.
2. **Вопрос:** Почему выборка данных в процедуре выполняется медленнее, чем вне её?**Ответ:** Внутри процедуры используется динамический SQL (например, через конструкцию EXECUTE), который каждый раз заново компилируется сервером. Для оптимизации рекомендуется использовать подготовленные запросы или хранимые процедуры с параметрами.
3. **Вопрос:** Можно ли вернуть данные сразу несколькими способами из процедуры в PostgreSQL?**Ответ:** Да, можно возвращать значения как OUT-параметрами, так и используя RETURN QUERY. Но стоит помнить, что одновременно нельзя использовать оба метода возврата значений.
4. **Вопрос:** Какие типы аргументов поддерживаются в процедурах и функциях PostgreSQL?**Ответ:** Аргументы могут быть любого типа данных PostgreSQL, включая скалярные типы (INTEGER, TEXT), составные типы (RECORDS), массивы и даже пользовательские типы данных.
5. **Вопрос:** Зачем нужны переменные уровня блока в процедурах PL/pgSQL?**Ответ:** Переменные уровня блока позволяют хранить промежуточные значения внутри тела процедуры, облегчают написание сложных запросов и повышают читаемость кода.
6. **Вопрос:** Чем отличается RETURNING от OUTPUT-параметров в PostgreSQL?**Ответ:** RETURNING возвращает строки непосредственно из оператора DML (INSERT, UPDATE, DELETE), тогда как OUTPUT-параметры используются для передачи результатов из функции обратно вызывающей стороне.
7. **Вопрос:** Может ли одна функция обновлять две разные таблицы одновременно?**Ответ:** Да, одна функция может изменять любые доступные ей таблицы, включая выполнение нескольких операторов UPDATE или INSERT подряд.
8. **Вопрос:** Что произойдёт, если вызвать процедуру без необходимых входных параметров?**Ответ:** Возникнет ошибка, поскольку PostgreSQL требует точное совпадение типов и количества аргументов.
9. **Вопрос:** Есть ли разница между процедурой и функцией в плане производительности?**Ответ:** Функции выполняются быстрее, так как поддерживают оптимизацию запросов, выполняемых внутри тела функции. Процедуры же менее оптимизированы и требуют больше ресурсов сервера.
10. **Вопрос:** Нужно ли явно указывать schema в вызове процедуры?**Ответ:** Если схема указана в пути поиска (search\_path), то явное указание схемы не обязательно. Однако, чтобы избежать путаницы и конфликтов именования, рекомендуется явно указывать имя схемы перед именем процедуры.

## **Триггеры. Типы триггеров. Создание, удаление, отключение (Disabling), включение (Enabling). Варианты срабатывания триггеров. Просмотр информации о триггерах PL/pgSQL**

### **Что такое триггеры?**

Триггер — это специальная процедура, автоматически запускаемая при наступлении события (например, при изменении данных в таблице). Триггеры обычно используются для поддержания целостности данных, автоматической генерации событий и выполнения специфических действий при изменениях.

### **Типы триггеров:**

* **BEFORE**: срабатывает перед событием (INSERT, UPDATE, DELETE).
* **AFTER**: срабатывает после события.
* **INSTEAD OF**: заменяет обычное действие на собственное (чаще используется с представлениями).

### **Пример создания триггера:**

CREATE TRIGGER trigger\_name

BEFORE INSERT ON table\_name

FOR EACH ROW EXECUTE FUNCTION trigger\_function();

### **Активация и деактивация триггеров:**

Активированные триггеры автоматически срабатывают при событии. Если триггер отключен, он игнорируется.

ALTER TABLE table\_name ENABLE TRIGGER trigger\_name; -- включение

ALTER TABLE table\_name DISABLE TRIGGER trigger\_name; -- выключение

### **Преимущества триггеров:**

* Обеспечение целостных связей между таблицами.
* Автоматизация реакции на изменения данных.
* Минимизация ручной работы администраторов и повышение безопасности данных.

### **Что такое триггеры?**

**Триггеры** представляют собой специальный вид механизма в реляционных базах данных, позволяющий автоматически запускать заранее определённые действия (процедуры, функции) при наступлении определенных событий (операций INSERT, UPDATE, DELETE или TRUNCATE). Они помогают поддерживать целостность данных, автоматизировать процессы обработки данных и обеспечивать согласованность транзакций.

### **Основные типы триггеров**

Существует два основных типа триггеров в зависимости от момента запуска:

* **BEFORE-триггеры**: запускаются перед выполнением основной операции (INSERT, UPDATE, DELETE).
* **AFTER-триггеры**: запускаются после завершения основной операции.

Также есть третий тип — **INSTEAD OF-триггеры**, применяемые исключительно для представления VIEW, когда необходимо перехватывать операцию над представлением и заменить её другой операцией.

Кроме того, различают триггеры по уровню воздействия:

* **ROW LEVEL (строчные)**: триггер срабатывает для каждой отдельной строки, затронутой оператором.
* **STATEMENT LEVEL (уровень оператора)**: триггер срабатывает всего один раз независимо от числа строк, затронутых оператором.

### **Пример таблицы employees**

Для иллюстрации рассмотрим таблицу сотрудников (employees), используемую далее в примерах:

CREATE TABLE employees (

employee\_id SERIAL PRIMARY KEY,

first\_name VARCHAR(50),

last\_name VARCHAR(50),

salary DECIMAL(10, 2),

hire\_date DATE DEFAULT CURRENT\_DATE

);

### **Создание триггера**

Рассмотрим процесс создания триггера на простом примере. Предположим, нам необходимо следить за изменениями зарплаты сотрудников и записывать каждую такую операцию в отдельную таблицу аудита (salary\_changes\_log):

-- Создаем таблицу для ведения журнала изменений зарплат

CREATE TABLE salary\_changes\_log (

log\_id SERIAL PRIMARY KEY,

employee\_id INTEGER NOT NULL,

old\_salary DECIMAL(10, 2),

new\_salary DECIMAL(10, 2),

change\_date TIMESTAMP DEFAULT NOW()

);

-- Определяем функцию, которую будет исполнять триггер

CREATE OR REPLACE FUNCTION track\_salary\_change() RETURNS TRIGGER AS $$

BEGIN

IF NEW.salary <> OLD.salary THEN

INSERT INTO salary\_changes\_log(employee\_id, old\_salary, new\_salary)

VALUES(NEW.employee\_id, OLD.salary, NEW.salary);

END IF;

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

-- Создаем AFTER-триггер, который сработает после каждого изменения зарплаты

CREATE TRIGGER after\_salary\_update\_trigger

AFTER UPDATE ON employees

FOR EACH ROW WHEN (OLD.salary IS DISTINCT FROM NEW.salary)

EXECUTE FUNCTION track\_salary\_change();

Разберемся подробнее, что делает каждая команда:

* **Функция**: Мы создали функцию track\_salary\_change() на языке PL/pgSQL, которая проверяет изменение поля salary и сохраняет запись о старом и новом значении в таблице аудита.
* **Триггер**: Триггер привязывается к событию UPDATE таблицы employees, и срабатывает после каждой модификации записи (оператор FOR EACH ROW). Условием является отличие новой и старой зарплаты (WHEN (OLD.salary IS DISTINCT FROM NEW.salary)).

### **Изменение и удаление триггеров**

Удалять триггеры можно следующим образом:

DROP TRIGGER if exists after\_salary\_update\_trigger ON employees;

Команда DROP удалит указанный триггер, если он существует.

Изменить существующий триггер невозможно напрямую, но можно переписать новый триггер поверх старого.

### **Отключение и включение триггеров**

Иногда возникает необходимость временно отключить триггеры для повышения производительности операций массового импорта данных или при выполнении специальных операций:

ALTER TABLE employees DISABLE TRIGGER after\_salary\_update\_trigger;

Это временно отключает указанный триггер. Чтобы снова включить его, выполните команду:

ALTER TABLE employees ENABLE TRIGGER after\_salary\_update\_trigger;

Вы также можете отключить ВСЕ триггеры для указанной таблицы:

ALTER TABLE employees DISABLE TRIGGER ALL;

### **Просмотр информации о триггерах**

Получить информацию о существующих триггерах можно с помощью системных каталогов:

SELECT tgname, tgenabled, tgfoid::regprocedure AS trigger\_function

FROM pg\_trigger

WHERE tgrelid = 'employees'::regclass;

Эта команда покажет имена триггеров, статус их активности и названия соответствующих функций.

### **Варианты срабатывания триггеров**

Основные варианты срабатывания триггеров включают следующие моменты:

* **BEFORE vs AFTER**: BEFORE-триггер позволяет внести изменения до того, как произойдет операция (например, проверить ограничения целостности), а AFTER-триггер применяется для действий после успешной операции (например, ведение журнала изменений).
* **INSTEAD OF**: Применяется только для представлений (VIEW) и позволяет переопределить поведение стандартных операций CRUD (Create, Read, Update, Delete).
* **FOR EACH STATEMENT / FOR EACH ROW**: Уровень оператора запускает триггер единожды для всей операции, уровень строки — для каждой затронутой строки.

### **Каверзные вопросы с ответами:**

1. **Вопрос:** Могут ли триггеры обращаться к другим таблицам кроме той, на которой они установлены?**Ответ:** Да, триггеры могут взаимодействовать с любыми таблицами, имеющими отношение к запросу, будь то чтение или модификация данных.
2. **Вопрос:** Всегда ли нужно создавать отдельную функцию для триггера?**Ответ:** Нет, можно определить триггер без отдельного определения функции, если логика достаточно проста. Но обычно удобно иметь отдельные функции для улучшения повторяемости и удобства тестирования.
3. **Вопрос:** Когда предпочтительнее использовать BEFORE-триггеры, а когда AFTER-триггеры?**Ответ:** BEFORE-триггеры применяются для проверки ограничений и подготовки данных перед операциями, а AFTER-триггеры полезны для записей изменений в журнал, отправки уведомлений и выполнения последующих обработок.
4. **Вопрос:** Возможно ли установить несколько триггеров одного типа на одно событие?**Ответ:** Да, можно настроить несколько триггеров одного типа (BEFORE или AFTER) на одно событие, порядок их исполнения задаётся командой CREATE TRIGGER.
5. **Вопрос:** Могут ли триггеры сами себя рекурсивно активировать?**Ответ:** Да, триггеры могут запускаться повторно ("рекурсия"), что иногда полезно, но часто приводит к нежелательным последствиям, вызывая бесконечные циклы. Поэтому важно внимательно проектировать такие сценарии.
6. **Вопрос:** Что означает конструкция NEW и OLD в теле триггера?**Ответ:** Конструкции NEW и OLD содержат новые и старые значения полей строки соответственно. Они доступны только в триггерах уровня строки и предоставляют доступ к полям обновляемой строки.
7. **Вопрос:** Возможна ли отмена операции INSERT или UPDATE через триггер?**Ответ:** Да, возможно. В BEFORE-триггере можно поднять исключение или возвратить значение NULL, чтобы отменить операцию (возврат NULL запрещает внесение изменений).
8. **Вопрос:** Будет ли срабатывать триггер при ошибке во вложенном запросе?**Ответ:** Если вложенный запрос вызывает ошибку, то сам триггер тоже завершится неудачей, и вся транзакция откатится назад, включая вызвавшую триггер операцию.
9. **Вопрос:** Как влияет установка INSTEAD OF-триггера на представление (VIEW)?**Ответ:** INSTEAD OF-триггер позволяет задать свою собственную обработку операций над виртуальной таблицей (представление), что даёт контроль над действиями над данными представления.
10. **Вопрос:** Что произойдёт, если попытаться обновить несуществующее поле в строке с триггером?**Ответ:** Произойдет ошибка времени выполнения, так как попытка обратиться к отсутствующему полю приведёт к сбою.

Триггеры являются мощным инструментом для автоматизации процессов в базах данных, помогающим сохранять целостность данных и улучшать удобство администрирования системы. Использование триггеров требует внимательного проектирования, особенно в больших системах, где важна производительность и масштабируемость.