Министерство образования и науки Российской Федерации
Санкт-Петербургский Политехнический Университет Петра Великого
—
Институт компьютерных наук и кибербезопасности

**ЛАБОРАТОРНАЯ РАБОТА № 2**

«**Организация уровневого доступа к данным**»

по дисциплине «Системы управления базами данных»

Выполнила
студентка гр. 5131001/10302 Куковякина Д. А.

*<подпись>*

Преподаватель Зубков Е. А.

*<подпись>*

Санкт-Петербург
2025

## Цель работы

Получение навыков обработки событий и гранулирования доступа при работе с СУБД.

## Формулировка задания

Порядок выполнения работы:

1. Реализовать заданное ограничение целостности.

2. Реализовать представления в соответствии с требованиями по разграничению прав из работы 1. Выполнить все условия разграничения. При разработке представлений учитывать обе функции безопасности, которые они выполняют.

3. Дать соответствующим пользователям права только в отношении созданных представлений.

4. Для прав на модификацию данных обеспечить и протестировать обновляемость представлений.

5. Сделать выводы в отношении применения стандартных способов разграничения прав доступа и построения разграничения на основе представлений по итогам данной работы и работы 1.

6. Обеспечить аудит действий пользователя по манипулированию данными с использованием вспомогательной таблицы для логгирования (в отношении базовой таблицы по выбору студента).

7. Оценить ограничения созданного лог-контента и его операционные характеристики: размер, скорость поиска и т. д.

Описание данных согласно варианту:

Отношение 1 – ФИО сотрудника (PK), номер документа сотрудника (AK1), телефон сотрудника (AK2), дата рождения, число несовершеннолетних детей, семейное положение, уровень образования, общий стаж, сведения о квалификации (если есть).

Отношение 2 – ФИО сотрудника (PK, FK), дата приема на должность (PK), отдел, дата окончания работы в должности (если есть), название должности.

Описание прав доступа согласно варианту:

Сотрудник может видеть все данные о себе, а также ФИО и телефоны других сотрудников, но только для своего отдела, где он работает на данный момент.

Сотрудник может редактировать только сведения о повышении квалификации, число несовершеннолетних детей и семейное положение для себя самого.

Ограничение целостности: уровень образования может только возрастать, его снижение невозможно (как и общего стажа).

## Ход работы

### Контроль целостности

Ограничения целостности в базах данных — это правила, которые обеспечивают корректность и согласованность данных. Они предотвращают внесение в базу данных ошибочной или противоречивой информации. В их основе лежат бизнес–правила предметной области и ссылочные ограничения СУБД, основанные на связях между отношениями.

С точки зрения особенностей реализации, ограничения целостности могут быть декларативными и не декларативными. Декларативные ограничения целостности могут задаваться при создании или изменении таблиц базы данных или специальными операторами, определенными в языке SQL. К инструментам задания декларативных ограничений относятся:

- Ключи отношений базы данных (первичные ключи; внешние ключи; атрибуты с уникальными значениями, представляющие альтернативные ключи отношений).
- Проверочные ограничения уровня атрибутов и отношений задаются оператором CHECK.

Недекларативные ограничения целостности в базах данных обычно реализуются с помощью программного кода, который выполняется при попытке изменения данных.

Триггер – это процедура, хранимая в базе данных и вызываемая автоматически при возникновении каких-либо событий. В PostgreSQL код триггера не «вкладывается» в его тело как в других СУБД и стандарте SQL, а используется триггерная функция. При этом одна и та же триггерная функция может быть использована для нескольких триггеров.

Ограничение целостности согласно варианту: уровень образования может только возрастать, его снижение невозможно (как и общего стажа).

Сперва реализуем функцию, которая будет проверять допустимость значения уровня образования (рисунок 1). Добавим 4 уровня: среднее, среднее специальное, высшее, доктор наук. Возвращать данная функция

будет индекс переданного уровня или ошибку, в случае если такого уровня не существует.

```
CREATE OR REPLACE FUNCTION check_education_level(ed_level varchar)
RETURNS int AS $$
DECLARE
  levels varchar[] := ARRAY['Среднее', 'Среднее специальное', 'Высшее', 'Доктор наук'];
  ed_index int;
BEGIN
  ed_index := array_position(levels, ed_level);
  IF ed_index is NULL THEN
    RAISE EXCEPTION 'Недопустимый уровень образования';
  END IF;
  RETURN ed_index;
END;
$$ LANGUAGE plpgsql;
```

Рисунок 1 – Проверка уровня образования

Добавим триггер на добавление строки в Employee (рисунок 2). Это понадобится в дальнейшем, чтобы при обновлении данных точно быть уверенным, что в ячейке записано допустимое значение.

```
CREATE OR REPLACE FUNCTION check_employee_education()
RETURNS TRIGGER AS $$
BEGIN
    PERFORM check_education_level(NEW.education_level);
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE TRIGGER employee_education_trigger
BEFORE INSERT ON Employee
FOR EACH ROW EXECUTE PROCEDURE check_employee_education();
```

Рисунок 2 – Триггер на добавление записи

Затем реализуем триггер, который будет проверять обновление стажа и уровня образования, тем самым обеспечивая ограничение целостности (рисунок 3).

```
CREATE OR REPLACE FUNCTION check_employee_update()
RETURNS TRIGGER AS $$
BEGIN
    IF check_education_level(NEW.education_level) < check_education_level(OLD.education_level) THEN
        RAISE EXCEPTION 'Уровень образования не может быть понижен';
    END IF;
    IF NEW.work_experience < OLD.work_experience THEN
        RAISE EXCEPTION 'Общий стаж не может быть уменьшен';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE TRIGGER employee_update_trigger
BEFORE UPDATE ON Employee
FOR EACH ROW EXECUTE PROCEDURE check_employee_update();
```

Рисунок 3 – Триггер для ограничения целостности

Проверим работу. Изначальная таблица представлена на рисунке 4, на рисунке 5 – повышение уровня, на рисунке 6 – понижение.

| | full_name [PK] character varying (100) | document_number integer | phone_number character varying (12) | birth_date date | children_count integer | family_status character varying (50) | education_level character varying (50) | work_experience integer | qualification text |
|---|---|---|---|---|---|---|---|---|---|
| 1 | Иван Иванов | 123456789 | 89123456789 | 1990-06-10 | 2 | Женат | Высшее | 10 | Инженер-программист |
| 2 | Мария Иванова | 987654321 | 89234567890 | 2000-11-20 | 0 | Не замужем | Высшее | 3 | Бухгалтер |
| 3 | Александр Иванов | 555555555 | 89345678901 | 1980-03-01 | 1 | Разведен | Среднее | 15 | [null] |

Рисунок 4 – Изначальные данные

```
1  UPDATE Employee SET education_level = 'Доктор наук' WHERE full_name = 'Иван Иванов';
2  SELECT * FROM EMPLOYEE
```

Data Output Messages Notifications

| | full_name [PK] character varying (100) | document_number integer | phone_number character varying (12) | birth_date date | children_count integer | family_status character varying (50) | education_level character varying (50) |
|---|---|---|---|---|---|---|---|
| 1 | Мария Иванова | 987654321 | 89234567890 | 2000-11-20 | 0 | Не замужем | Высшее |
| 2 | Александр Иванов | 555555555 | 89345678901 | 1980-03-01 | 1 | Разведен | Среднее |
| 3 | Иван Иванов | 123456789 | 89123456789 | 1990-06-10 | 2 | Женат | Доктор наук |

Рисунок 5 – Повышение уровня образования

```
1  UPDATE Employee SET education_level = 'Высшее' WHERE full_name = 'Иван Иванов';
2  SELECT * FROM EMPLOYEE
```

Data Output Messages Notifications

```
ERROR:  Уровень образования не может быть понижен
CONTEXT:  функция PL/pgSQL check_employee_update(), строка 4, оператор RAISE

ОШИБКА:  Уровень образования не может быть понижен
SQL state: P0001
```

Рисунок 6 – Понижение уровня образования

**Разграничение прав доступа**

В таблице 1 представлена исходная матрица доступа.

Таблица 1 – Матрица доступа

| Атрибут | ФИО = full_name | Отдел = department<br>ФИО ≠ full_name |
|---|---|---|
| full_name | R | R |
| document_number | R | - |
| phone_number | R | R |
| birth_date | R | - |
| children_count | RU | - |
| family_status | RU | - |
| education_level | R | - |
| work_experience | R | - |
| qualification | RU | - |
| appointment_date | R | - |
| department | R | - |

| | | |
|---|---|---|
| termination_date | R | - |
| position_name | R | - |

Реализуем два представления. Первое будет отвечать за предоставление доступа пользователю к своим данным (рисунок 7).

```
CREATE OR REPLACE VIEW EmployeeSelf AS
SELECT
    e.*,
    ep.department,
    ep.appointment_date,
    ep.termination_date,
    ep.position_name
FROM
    Employee e
JOIN
    EmployeePosition ep ON e.full_name = ep.full_name
WHERE
    e.full_name = current_user;
```

Рисунок 7 – Первое представление

Представление объединяет данные из обеих таблиц. Рассмотрим запрос подробнее:

- SELECT указывает, какие столбцы будут выбраны. В данном случае выбираются все атрибуты из таблицы Employee (e) и все, кроме имени, из таблицы EmployeePosition (ep).
- FROM Employee e указывает, что данные берутся из таблицы Employee, которой присваивается псевдоним e для удобства.
- JOIN EmployeePosition ep ON e.full_name = ep.full_name – этот оператор объединяет записи из таблиц Employee и EmployeePosition на основе совпадения значений столбца full_name в обеих таблицах.
- WHERE e.full_name = current_user – условие фильтрации. Оно оставляет в результате только те записи, для которых значение столбца full_name в таблице Employee совпадает с именем текущего пользователя, выполняющего запрос (current_user). current_user — это системная функция, возвращающая имя текущего пользователя базы данных.

Второе представление отвечает за доступ к ФИО и номеру телефона сотрудников своего отдела (рисунок 8).

```sql
CREATE VIEW EmployeeContacts AS
SELECT
    e.full_name,
    e.phone_number
FROM
    Employee e
JOIN
    EmployeePosition ep ON e.full_name = ep.full_name
WHERE
    ep.department = (SELECT department FROM EmployeePosition WHERE full_name = current_user
    AND termination_date is NULL) AND termination_date is NULL;

GRANT SELECT ON EmployeeContacts TO employee;
```

Рисунок 8 – Второе представление

Ключевой частью является условие фильтрации. Оно состоит из двух частей:

- Первая находит отдел текущего пользователя, в котором нет даты увольнения, а затем ищет тех сотрудников, чей отдел совпадает с отделом текущего пользователя.
- Вторая гарантирует, что в представление попадут только действующие сотрудники (у которых termination_date равно NULL).

Обновление требуется только первому представлению, так как пользователь может изменять только данные о себе. Для этого реализуем триггер, срабатывающий при вызове Update над представлением (рисунок 9).

```sql
CREATE OR REPLACE FUNCTION update_info()
RETURNS TRIGGER
SECURITY DEFINER
AS $$
BEGIN
    IF (OLD.qualification != NEW.qualification) THEN
        UPDATE Employee SET qualification = NEW.qualification WHERE full_name = OLD.full_name;
    ELSIF (OLD.family_status != NEW.family_status)  THEN
        UPDATE Employee SET family_status = NEW.family_status WHERE full_name = OLD.full_name;
    ELSIF (OLD.children_count != NEW.children_count)  THEN
        UPDATE Employee SET children_count = NEW.children_count WHERE full_name = OLD.full_name;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE TRIGGER trigger_update_info
INSTEAD OF UPDATE ON EmployeeSelf
FOR EACH ROW EXECUTE PROCEDURE update_info();

GRANT UPDATE (children_count, family_status, qualification) ON EmployeeSelf TO employee;
```

Рисунок 9 – Обновление представления

Важным элементом является SECURITY DEFINER. Он указывает, что функция выполняется с правами владельца функции, а не пользователя, который ее вызывает. Это необходимо, так как триггер срабатывает от имени

пользователя, который пытается обновить представление EmployeeSelf, а у этого пользователя нет прав на непосредственное обновление таблицы Employee.

На рисунке 10 представлен пример обновления числа детей.

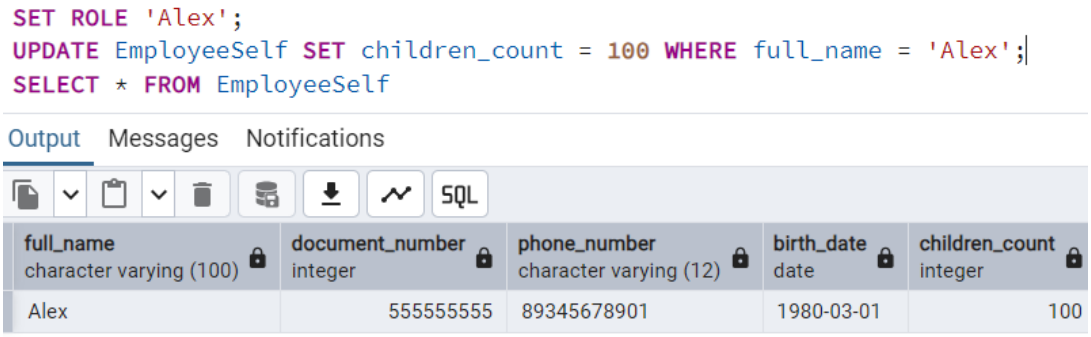


Рисунок 10 – Обновление представления

В современных СУБД разграничение доступа к базе данных, или грануляция объектов, возможно на нескольких уровнях: уровне таблиц (TLS –table level security), уровне записей (RLS – row level security) или уровне ячеек (CLS – cell level security).

GRANT используется для предоставления или отзыва привилегий (прав доступа) на уровне объектов базы данных (таблицы, представления, последовательности и т. д.). Можно разрешить или запретить выполнение определенных операций (SELECT, INSERT, UPDATE, DELETE) для конкретных пользователей или ролей.

POLICY позволяет создавать политики безопасности, которые определяют, какие строки таблицы доступны для конкретных пользователей или ролей. Политики основаны на логических выражениях, которые могут ссылаться на столбцы таблицы, текущего пользователя и другие параметры.

Представления — это виртуальные таблицы, которые создаются на основе SQL-запросов. Можно создавать представления, которые содержат только те данные, которые необходимо предоставить пользователям.

Триггеры — это специальные функции, которые автоматически выполняются при возникновении определенных событий в базе данных (например, вставка, обновление или удаление данных). Их можно

использовать для проверки данных перед их изменением или для выполнения дополнительных действий, связанных с сокрытием данных.

Представления и триггеры позволяют реализовать более точечное ограничение на уровне ячеек.

**Логгирование**

Логгирование также реализуем при помощи триггера (рисунок 11). В логи будет записываться имя пользователя, таблица, действие и данные.

```
CREATE OR REPLACE FUNCTION audit()
RETURNS TRIGGER
AS $$
BEGIN
    IF (TG_OP = 'DELETE') THEN
        INSERT INTO logs(username, act, ch_table, changed_on, change)
        VALUES(current_user, 'delete', TG_TABLE_NAME, now(),
            format('full_name: ''%s''';
                    ..',
                    OLD.full_name,
                    ..);

        RETURN OLD;
    ELSIF (TG_OP = 'UPDATE') THEN
        INSERT INTO logs(username, act, ch_table, changed_on, change)
        VALUES(current_user, 'update', TG_TABLE_NAME, now(),
        format('full_name: ''%s'' -> ''%s''';
                ..',
                OLD.full_name, NEW.full_name,
                ..));
        RETURN NEW;
    ELSIF (TG_OP = 'INSERT') THEN
        INSERT INTO logs(username, act, ch_table, changed_on, change)
        VALUES(current_user, 'insert', TG_TABLE_NAME, now(),
        format('full_name: ''%s''';
                ..',
                NEW.full_name,
                ..);
        RETURN NEW;
    END IF;
```

Рисунок 11 – Логгирование

Установим триггер на изменение, добавление и удаление строк таблицы Employee, а также на обновление представления EmployeeSelf (рисунок 12). Это позволит отслеживать изменения через представление, так как функция обновления выполняется от имени создателя (postgres).

```
CREATE OR REPLACE TRIGGER audit_logs
AFTER INSERT OR UPDATE OR DELETE ON Employee
FOR EACH ROW EXECUTE PROCEDURE audit();

CREATE OR REPLACE TRIGGER audit_logs
INSTEAD OF UPDATE ON EmployeeSelf
FOR EACH ROW EXECUTE PROCEDURE audit();

GRANT INSERT ON logs TO employee;
```

Рисунок 12 –Установка триггера

Проверим работоспособность. Обновим запись, после чего удалим ее. Соответствующие записи появятся в таблице (рисунок 13). Также обновим данные от имени другого пользователя – появятся две записи (рисунок 14).

```
1  UPDATE Employee SET children_count = 8 WHERE full_name = 'Мария Иванова';
2  DELETE FROM Employee WHERE full_name = 'Иван Иванов';
3  SELECT * FROM logs
```

Data Output | Messages | Notifications

| | username character varying (100) | act text | ch_table text | changed_on timestamp without time zone (6) | change text |
|---|---|---|---|---|---|
| 1 | postgres | insert | employee | 2025-02-13 21:38:12.776359 | full_name: 'Иван Иванов'; |
| 2 | postgres | insert | employee | 2025-02-13 21:38:12.776359 | full_name: 'Мария Иванова'; |
| 3 | postgres | insert | employee | 2025-02-13 21:38:12.776359 | full_name: 'Александр Иванов'; |
| 4 | postgres | update | employee | 2025-02-13 21:39:49.76492 | full_name: 'Мария Иванова' -> 'Мария Иванова'; |
| 5 | postgres | delete | employee | 2025-02-13 21:39:49.76492 | full_name: 'Иван Иванов'; |

Рисунок 13 – Обновление от postgres

```
2  SET ROLE 'Александр Иванов';
3  UPDATE EmployeeSelf SET children_count = 100 WHERE full_name = 'Александр Иванов';
4  RESET ROLE;
5  SELECT * FROM logs
```

Data Output | Messages | Notifications

| | username character varying (100) | act text | ch_table text | changed_on timestamp without time zone (6) | change text |
|---|---|---|---|---|---|
| 1 | postgres | insert | employee | 2025-02-13 21:38:12.776359 | full_name: 'Иван Иванов'; |
| 2 | postgres | insert | employee | 2025-02-13 21:38:12.776359 | full_name: 'Мария Иванова'; |
| 3 | postgres | insert | employee | 2025-02-13 21:38:12.776359 | full_name: 'Александр Иванов'; |
| 4 | postgres | update | employee | 2025-02-13 21:39:49.76492 | full_name: 'Мария Иванова' -> 'Мария Иванова'; |
| 5 | postgres | delete | employee | 2025-02-13 21:39:49.76492 | full_name: 'Иван Иванов'; |
| 6 | Александр Иванов | update | employeeself | 2025-02-13 21:41:22.42519 | full_name: 'Александр Иванов' -> 'Александр Иванов'; |
| 7 | postgres | update | employee | 2025-02-13 21:41:22.42519 | full_name: 'Александр Иванов' -> 'Александр Иванов'; |

Рисунок 14 – Обновление от другого пользователя

Воспользуемся запросом EXPLAIN ANALYZE. Проанализируем обновление и удаление записи (рисунок 15).

```
postgres=# EXPLAIN ANALYZE UPDATE Employee SET children_count = 100 WHERE full_name = 'Alex';
                                                                 QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------------------------
 Update on employee  (cost=0.14..8.16 rows=0 width=0) (actual time=0.692..0.693 rows=0 loops=1)
   ->  Index Scan using employee_pkey on employee  (cost=0.14..8.16 rows=1 width=10) (actual time=0.031..0.032 rows=1 loops=1)
         Index Cond: ((full_name)::text = 'Alex'::text)
 Planning Time: 2.255 ms
 Trigger audit_logs: time=0.957 calls=1
 Trigger employee_update_trigger: time=0.559 calls=1
 Execution Time: 1.761 ms
(7 rows)


postgres=# EXPLAIN ANALYZE DELETE FROM Employee WHERE full_name= 'Alex';
                                                                 QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------------------------
 Delete on employee  (cost=0.14..8.16 rows=0 width=0) (actual time=0.048..0.048 rows=0 loops=1)
   ->  Index Scan using employee_pkey on employee  (cost=0.14..8.16 rows=1 width=6) (actual time=0.030..0.031 rows=1 loops=1)
         Index Cond: ((full_name)::text = 'Alex'::text)
 Planning Time: 0.102 ms
 Trigger for constraint employeeposition_full_name_fkey: time=1.448 calls=1
 Trigger audit_logs: time=0.187 calls=1
 Execution Time: 1.703 ms
(7 rows)
```

Рисунок 15 – Анализ обновления и удаления

Видно, что большую часть времени выполнения запроса обновления приходится на триггер аудита (audit_logs). При удалении аудит занимает меньше времени, но также достаточно много. Также рассмотрим добавление записи. Опять же основное время занимает логгирование (рисунок 16).

| | QUERY PLAN<br>text |
|---|---|
| 1 | Insert on employee  (cost=0.00..0.01 rows=0 width=0) (actual time=1.389..1.389 rows=0 loops=… |
| 2 | -> Result  (cost=0.00..0.01 rows=1 width=544) (actual time=0.001..0.002 rows=1 loops=1) |
| 3 | Planning Time: 0.037 ms |
| 4 | Trigger audit_logs: time=0.756 calls=1 |
| 5 | Trigger employee_education_trigger: time=0.302 calls=1 |
| 6 | Execution Time: 2.163 ms |

Рисунок 16 – Анализ добавления

Также проанализируем саму таблицу (рисунок 17). Воспользуемся командой \d+, которая выводит подробную информацию о таблице, представлении или другом объекте базы данных.

```
postgres=# \d+ logs
                                                    Table "public.logs"
   Column   |              Type              | Collation | Nullable | Default | Storage  | Compression | Stats target | Description
------------+--------------------------------+-----------+----------+---------+----------+-------------+--------------+-------------
 username   | character varying(100)         |           | not null |         | extended |             |              |
 act        | text                           |           | not null |         | extended |             |              |
 ch_table   | text                           |           | not null |         | extended |             |              |
 changed_on | timestamp(6) without time zone |           | not null |         | plain    |             |              |
 change     | text                           |           | not null |         | extended |             |              |
Access method: heap
```

Рисунок 17 – Анализ таблицы

«Access method: heap» означает, что для хранения данных используется куча. Куча (heap) — это наиболее простой способ организации хранения данных в PostgreSQL. В куче записи хранятся в произвольном порядке, и для

поиска нужной записи требуется последовательный просмотр всего файла таблицы. Куча подходит для небольших таблиц, таблиц, к которым редко обращаются, или таблиц, где порядок хранения данных не важен.

Преимущества использования кучи для таблицы логгирования:

- Куча обеспечивает высокую скорость записи данных, так как новые записи просто добавляются в конец файла таблицы без необходимости какой-либо сортировки или индексации. Это идеально подходит для хранения логов, где важна скорость добавления новых записей.
- Если необходимо читать логи в порядке их поступления (например, для анализа или отладки), куча может быть достаточно эффективной, так как записи хранятся в порядке добавления.

Недостатки:

- Если необходимо искать конкретные записи в логах по определенным критериям (например, по времени, типу события или другим параметрам), куча может быть неэффективной, так как потребуется последовательный просмотр всего файла таблицы.

Также важным является просмотр Storage. Он показывает, как именно хранятся данные этого столбца на диске. Эта информация важна для понимания производительности и эффективности хранения данных.

В PostgreSQL существует несколько типов хранения для столбцов:

- PLAIN – тип хранения по умолчанию. Данные хранятся непосредственно в строке таблицы. Подходит для небольших полей фиксированной длины, таких как целые числа или даты.
- EXTERNAL – данные хранятся вне основной строки таблицы в отдельной TOAST-таблице. В основной строке остается только указатель на TOAST-таблицу. Используется для больших полей переменной длины, таких как текст или массивы.
- EXTENDED – аналогично EXTERNAL, но данные сжимаются перед сохранением в TOAST-таблице. PostgreSQL использует этот формат для хранения данных, которые не помещаются в стандартный размер страницы

(обычно 8 КБ). Это может быть вызвано большими объемами текста или другими типами данных переменной длины. Extended может немного снизить производительность операций чтения и записи для этих столбцов, так как PostgreSQL требуется дополнительное время для обработки данных, хранящихся вне страниц таблицы. Однако, для текстовых данных это обычно оправдано.

- MAIN – аналогично PLAIN, но применяется оптимизация для предотвращения "раздутия" таблицы.

В данном случае для столбцов username, act, ch_table и change используется extended, так как они содержат текст, длина которого может варьироваться.

Для столбца changed_on используется plain, так как тип timestamp имеет фиксированный размер.

## Контрольные вопросы

**1. В чем отличие практического применения триггеров FOR EACH ROW и FOR EACH STATEMENT? В каких практических случаях лучше применять первый, а в каких второй тип?**

FOR EACH ROW – триггер срабатывает для каждой строки, которая была изменена в результате выполнения SQL-оператора. Триггеры FOR EACH ROW могут использоваться для проверки сложных ограничений целостности, которые не могут быть выражены с помощью стандартных средств SQL. Также они подходят для создания журналов изменений данных, реализации обновления представлений.

FOR EACH STATEMENT – триггер срабатывает один раз для всего SQL-оператора, независимо от количества измененных строк. Триггеры FOR EACH STATEMENT подходят для создания журналов операций над таблицами, таких как создание, изменение или удаление таблиц. Также они могут проверять права доступа пользователя перед выполнением SQL-запроса.

**2. В чем отличие практического применения триггеров BEFORE и AFTER? В каких практических случаях лучше применять первый, а в каких второй тип?**

BEFORE – триггер срабатывает перед выполнением SQL-запроса, который вызвал изменение данных. Триггеры BEFORE идеально подходят для проверки данных, которые пытаются изменить. Например, можно убедиться, что значение поля находится в допустимом диапазоне, или что пользователь имеет право на изменение данных. Если проверка данных в триггере BEFORE не проходит, можно отменить выполнение SQL-запроса, вызвав исключение.

AFTER – триггер срабатывает после выполнения SQL-запроса, который вызвал изменение данных. Триггеры AFTER идеально подходят для создания журналов изменений данных, могут также использоваться для отправки

уведомлений о произошедших изменениях и отвечать за синхронизацию данных между разными таблицами или базами данных.

**3. В чем специфика триггеров типа INSTEAD OF и в каких практических задачах они применяются? Чем они лучше или хуже в конкретных случаях триггеров BEFORE и AFTER?**

В отличие от триггеров BEFORE и AFTER, которые выполняются до или после события (вставка, обновление, удаление), триггеры INSTEAD OF заменяют собой исходную операцию. То есть, вместо выполнения исходного SQL-запроса, выполняется код, определенный в триггере.

Триггеры INSTEAD OF в основном применяются для работы с представлениями (VIEW). Они позволяют определить, как именно должны быть изменены базовые таблицы, лежащие в основе представления, при попытке изменения данных через представление. Например, они позволяют корректно обрабатывать изменения данных через представления, которые объединяют несколько таблиц. Триггер должен определить, в какие именно таблицы и каким образом должны быть внесены изменения.

**4. С какими привилегиями выполняется код внутри созданного вами триггера?**

По умолчанию триггер выполняется от имени пользователя, который выполнил операцию, вызвавшую срабатывание триггера. Это означает, что триггер имеет доступ к тем же объектам базы данных и привилегиям, что и пользователь, инициировавший операцию.

При этом функция-триггер, связанная с триггером, может быть определена с ключевым словом SECURITY DEFINER, что изменяет контекст выполнения на контекст владельца функции.

**5. Какие две функции безопасности в общем случае выполняют представления? Разверните ответ на примере созданных вами.**

С целью защиты информации представления могут выполнять две функции:

1) Сокрытие реальной структуры данных. Например, благодаря представлению пользователь не знает, что изначальные данные хранятся в двух таблицах и какие атрибуты они имеют.

2) Изоляция объектов и данных при разграничении прав доступа. Например, с помощью второго представления сотрудник получает доступ только к ФИО и телефонам других сотрудников своего отдела.

**6. Какова максимальная длина лога, который вы записываете?**

Лог представлен в виде строки типа TEXT. Максимальная длина этого типа данных 65535 символов или 64 Кб.

**7. Какова максимальная длина записи в организованном вами логе? Имеет ли лог в связи с этим операционные ограничения?**

Запись лога представлена в виде строки типа TEXT. Максимальная длина этого типа данных составляет 65535 символов или 64 Кб.

**8. В каких случаях встроенные механизмы разграничения доступа наиболее эффективны?**

С помощью GRANT можно предоставить пользователю или группе право на выполнение определенных SQL-запросов (например, SELECT, INSERT, UPDATE, DELETE) над конкретными таблицами или представлениями – грануляция на уровне таблиц.

С помощью POLICY можно установить ограничения на доступ к данным на основе различных критериев, например значения определенных столбцов – грануляция на уровне строк.

Данные механизмы эффективны:

- Когда структура базы данных относительно проста и не требует сложных правил доступа, основанных на содержимом данных или контексте запроса.
- В небольших базах данных, где количество пользователей и объектов относительно невелико, механизмы GRANT и POLICY обеспечивают достаточную гибкость и простоту управления доступом.

- Встроенные механизмы обычно оптимизированы для работы с базой данных и не оказывают существенного влияния на производительность системы.

**9. В каких случаях более рационально организовать доступ через представления, чем использовать встроенные средства? Перечислите и обоснуйте все варианты.**

Представления позволяют реализовать грануляцию на уровне ячеек. Они используются:

- Когда нужно предоставить пользователям доступ к данным, которые являются результатом сложных запросов, объединяющих данные из нескольких таблиц или выполняющих агрегацию и фильтрацию.
- Когда нужно скрыть от пользователей определенные столбцы или строки таблицы, не предоставляя им полных прав доступа к таблице.
- Когда нужно предоставить пользователям простой и понятный интерфейс для доступа к данным, скрывая сложность структуры базы данных.
- Когда нужно обеспечить дополнительный уровень защиты конфиденциальных данных, предоставляя доступ к ним только через представления, которые выполняют предварительную обработку или фильтрацию данных.

Например:

- Можно создать представление, которое объединяет данные из нескольких таблиц и показывает только те столбцы, которые нужны конкретному пользователю.
- Можно создать представление, которое фильтрует данные по определенному критерию (например, по дате или по значению определенного столбца) и показывает только те строки, которые соответствуют этому критерию.

- Можно создать представление, которое агрегирует данные (например, вычисляет сумму или среднее значение) и показывает только итоговые результаты.

## Вывод

В ходе данной лабораторной работы были изучены механизмы обработки событий и гранулирования доступа в базах данных.

Обработка событий позволяет отслеживать различные события, происходящие в базе данных, и реагировать на них соответствующим образом. Например, можно настроить систему так, чтобы она записывала информацию о каждом изменении данных в специальный журнал аудита.

Гранулирование доступа дает возможность точно контролировать, кто и к каким данным имеет доступ. Это значительно повышает безопасность и надежность базы данных, защищая ее от несанкционированного доступа и утечек информации.

Оба этих механизма играют важную роль в управлении базой данных и обеспечении ее безопасности и надежности. Они помогают контролировать происходящие события, защищать конфиденциальные данные и поддерживать целостность информации.

**Приложение**

```
DROP TABLE IF EXISTS Employee CASCADE;
CREATE TABLE Employee(
full_name varchar(100) primary key not null,
document_number int unique not null,
phone_number varchar(12) unique not null,
birth_date date not null,
children_count int not null,
family_status varchar(50) not null,
education_level varchar(50) not null,
work_experience int not null,
qualification text
);

DROP TABLE IF EXISTS EmployeePosition  CASCADE;
CREATE TABLE EmployeePosition (
full_name varchar(100) not null,
appointment_date date not null,
department varchar(50) not null,
termination_date date,
position_name varchar(50) not null,
primary key (full_name, appointment_date),
foreign key (full_name) references Employee(full_name)
on delete cascade on update cascade
);



-- контроль целостности
CREATE OR REPLACE FUNCTION check_education_level(ed_level varchar)
```

```
RETURNS int AS $$
DECLARE
  levels varchar[] := ARRAY['Среднее', 'Среднее специальное', 'Высшее', 'Доктор наук'];
  ed_index int;
BEGIN
  ed_index := array_position(levels, ed_level);
  IF ed_index is NULL THEN
    RAISE EXCEPTION 'Недопустимый уровень образования';
  END IF;
  RETURN ed_index;
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION check_employee_update()
RETURNS TRIGGER AS $$
BEGIN
IF check_education_level(NEW.education_level) < check_education_level(OLD.education_level) THEN
        RAISE EXCEPTION 'Уровень образования не может быть понижен';
        END IF;
IF NEW.work_experience < OLD.work_experience THEN
        RAISE EXCEPTION 'Общий стаж не может быть уменьшен';
END IF;
RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE TRIGGER employee_update_trigger
```

```
BEFORE UPDATE ON Employee
FOR EACH ROW EXECUTE PROCEDURE check_employee_update();

CREATE OR REPLACE FUNCTION check_employee_education()
RETURNS TRIGGER AS $$
BEGIN
PERFORM check_education_level(NEW.education_level);
RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE TRIGGER employee_education_trigger
BEFORE INSERT ON Employee
FOR EACH ROW EXECUTE PROCEDURE check_employee_education();

-- создание пользователей
DO $$
BEGIN
   IF NOT EXISTS (SELECT FROM pg_catalog.pg_roles WHERE rolname
= 'employee') THEN
      CREATE ROLE employee;
   END IF;
END$$;

DROP USER IF EXISTS "Ivan";
CREATE USER "Ivan" WITH PASSWORD '123';
GRANT employee TO "Ivan";

DROP USER IF EXISTS "Masha";
CREATE USER "Masha" WITH PASSWORD '123';
```

```
GRANT employee TO "Masha";

DROP USER IF EXISTS "Alex";
CREATE USER "Alex" WITH PASSWORD '123';
GRANT employee TO "Alex";

-- контроль доступа (свои данные)
CREATE OR REPLACE VIEW EmployeeSelf AS
SELECT
    e.*,
ep.department,
    ep.appointment_date,
    ep.termination_date,
    ep.position_name
FROM
    Employee e
JOIN
    EmployeePosition ep ON e.full_name = ep.full_name
WHERE
    e.full_name = current_user;

GRANT SELECT ON EmployeeSelf TO employee;

-- контроль доступа (чужие данные)
CREATE VIEW EmployeeContacts AS
SELECT
    e.full_name,
    e.phone_number
FROM
    Employee e
```

```
    JOIN
       EmployeePosition ep ON e.full_name = ep.full_name
    WHERE
            ep.department = (SELECT department FROM EmployeePosition
WHERE full_name = current_user
    AND termination_date is NULL) AND termination_date is NULL;

    GRANT SELECT ON EmployeeContacts TO employee;

    -- обновление представления
    CREATE OR REPLACE FUNCTION update_info()
    RETURNS TRIGGER
    SECURITY DEFINER
    AS $$
    BEGIN
    IF (OLD.qualification != NEW.qualification) THEN
            UPDATE Employee SET qualification = NEW.qualification WHERE
full_name = OLD.full_name;
    ELSIF (OLD.family_status != NEW.family_status)  THEN
            UPDATE Employee SET family_status = NEW.family_status
WHERE full_name = OLD.full_name;
    ELSIF (OLD.children_count != NEW.children_count)  THEN
            UPDATE Employee SET children_count = NEW.children_count
WHERE full_name = OLD.full_name;
    END IF;
    RETURN NEW;
    END;
    $$ LANGUAGE plpgsql;

    CREATE OR REPLACE TRIGGER trigger_update_info
```

```
INSTEAD OF UPDATE ON EmployeeSelf
FOR EACH ROW EXECUTE PROCEDURE update_info();

GRANT UPDATE (children_count, family_status, qualification) ON EmployeeSelf TO employee;

-- логирование
DROP TABLE IF EXISTS logs CASCADE;
CREATE TABLE logs
(
   username varchar(100) NOT NULL,
act TEXT NOT NULL,
ch_table TEXT NOT NULL,
changed_on TIMESTAMP(6) NOT NULL,
   change text NOT NULL
);

CREATE OR REPLACE FUNCTION audit()
RETURNS TRIGGER
AS $$
BEGIN
   IF (TG_OP = 'DELETE') THEN
        INSERT INTO logs(username, act, ch_table, changed_on, change)
        VALUES(current_user, 'delete', TG_TABLE_NAME, now(),
                format('full_name: "%s";
                                document_number: "%s";
                                phone_number: "%s";
                                birth_date: "%s";
                                children_count: "%s";
                                family_status: "%s";
```

```
                education_level: "%s";
                work_experience: "%s";
                qualification: "%s"',
                OLD.full_name,
                OLD.document_number,
                OLD.phone_number,
                OLD.birth_date,
                OLD.children_count,
                OLD.family_status,
                OLD.education_level,
                OLD.work_experience,
                OLD.qualification));

        RETURN OLD;
    ELSIF (TG_OP = 'UPDATE') THEN
        INSERT INTO logs(username, act, ch_table, changed_on, change)
        VALUES(current_user, 'update', TG_TABLE_NAME, now(),
        format('full_name: "%s" -> "%s";
                document_number: "%s" -> "%s";
                phone_number: "%s" -> "%s";
                birth_date: "%s" -> "%s";
                children_count: "%s" -> "%s"
                family_status: "%s" -> "%s"
                education_level: "%s" -> "%s"
                work_experience: "%s" -> "%s"
                qualification: "%s" -> "%s"',
                OLD.full_name, NEW.full_name,
                OLD.document_number, NEW.document_number,
                OLD.phone_number, NEW.phone_number,
                OLD.birth_date, NEW.birth_date,
```

```
                    OLD.children_count, NEW.children_count,
                    OLD.family_status, NEW.family_status,
                    OLD.education_level, NEW.education_level,
                    OLD.work_experience, NEW.work_experience,
                    OLD.qualification, NEW.qualification));
        RETURN NEW;
ELSIF (TG_OP = 'INSERT') THEN
        INSERT INTO logs(username, act, ch_table, changed_on, change)
        VALUES(current_user, 'insert', TG_TABLE_NAME, now(),
        format('full_name: "%s";
                    document_number: "%s";
                    phone_number: "%s";
                    birth_date: "%s";
                    children_count: "%s";
                    family_status: "%s";
                    education_level: "%s";
                    work_experience: "%s";
                    qualification: "%s"',
                    NEW.full_name,
                    NEW.document_number,
                    NEW.phone_number,
                    NEW.birth_date,
                    NEW.children_count,
                    NEW.family_status,
                    NEW.education_level,
                    NEW.work_experience,
                    NEW.qualification));
        RETURN NEW;
END IF;
   RETURN NULL;
```

```
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE TRIGGER audit_logs
AFTER INSERT OR UPDATE OR DELETE ON Employee
FOR EACH ROW EXECUTE PROCEDURE audit();

CREATE OR REPLACE TRIGGER audit_logs_view
INSTEAD OF UPDATE ON EmployeeSelf
FOR EACH ROW EXECUTE PROCEDURE audit();

GRANT INSERT ON logs TO employee;

-- заполнение
INSERT INTO Employee (full_name, document_number, phone_number, birth_date, children_count, family_status, education_level, work_experience, qualification)
VALUES ('Ivan', 123456789, '89123456789', '1990-06-10', 2, 'Женат', 'Высшее', 10, 'Инженер-программист'),
       ('Masha', 987654321, '89234567890', '2000-11-20', 0, 'Не замужем', 'Высшее', 3, 'Бухгалтер');
INSERT INTO Employee (full_name, document_number, phone_number, birth_date, children_count, family_status, education_level, work_experience)
VALUES ('Alex', 555555555, '89345678901', '1980-03-01', 1, 'Разведен', 'Среднее', 15);

INSERT INTO EmployeePosition (full_name, appointment_date, department, termination_date, position_name)
VALUES ('Ivan', '2020-06-15', 'IT', '2021-08-25', 'Разработчик'),
       ('Masha', '2021-09-01', 'Финансы', '2022-07-31', 'Стажер');
```

```
INSERT INTO EmployeePosition (full_name, appointment_date, department, position_name)
VALUES  ('Ivan', '2021-08-25', 'IT', 'Старший разработчик'),
        ('Masha', '2022-07-31', 'Финансы', 'Бухгалтер'),
        ('Alex', '2010-04-05', 'IT', 'Системный администратор');

SELECT * FROM logs
```