

## **Что такое безопасная система?**

**Безопасная система** – это система, которая защищена от несанкционированного доступа, использования, раскрытия, нарушения, модификации или уничтожения, тем самым обеспечивая конфиденциальность, целостность и доступность данных и функций.

## **CIA Triad**

- Конфиденциальность (Confidentiality)
- Целостность (Integrity)
- Доступность (Availability)

# Определение пользователя

- Идентификация – процесс, когда информационная система определяет, существует конкретный пользователь или нет. Делает она это с помощью идентификатора. Идентификатором может быть логин, электронная почта, номер телефона или другой признак, который есть только у одного пользователя.
- Аутентификация – процесс, когда пользователь вводит ключ, например пароль или пин-код, подтверждая своё право на доступ к той или иной учётной записи и хранящейся в ней информации
- Авторизация – процесс определения того, какие действия позволено совершать аутентифицированному пользователю.

# Идентификация

- То, что субъект знает
- То, что субъекту принадлежит
- То, что является неотъемлемой характеристикой субъекта

# Многофакторная идентификация

- Если для входа требуются пароль и, например, ответ на секретный вопрос (который тоже является "знанием"), то это не многофакторная аутентификация, а всего лишь двухэтапная проверка в рамках одного фактора.

# Авторизация

	User 1	User 2	User 3
Object 1	READ		
Object 2		READ, ADD	
Object 3		UPDATE, READ	
Object 4	READ		

## Дискретный уровень доступа

- Каждый объект имеет владельца
- Владелец полностью контролирует доступ к своему объекту
- Права доступа определяются на основе списков контроля доступа (ACL)

- SELECT \*
- FROM pg\_class

```
[null]  
[null]  
robin155=arwdDxt/robin155  
robin155=arwdDxt/robin155,r/robin155  
[null]  
[null]
```

- SYSTEM PRIVILEGES
- DATABASE PRIVILEGES
- SCHEMA PRIVILEGES
- TABLE PRIVILEGES
- COLUMN PRIVILEGES
- ROW-LEVEL PRIVILEGES

# Модель доступа

## Владелец объекта

- В PostgreSQL каждый объект базы данных имеет владельца, который создаёт этот объект. Этот владелец обладает полным контролем над объектом.
- По умолчанию владелец объекта может передавать свои права другим ролям, позволяя им выполнять определённые действия с объектом.

## **Модель доступа**

### **Пользователи и группы**

- Реализована через концепцию ролей, которая служит для управления доступом и правами пользователей к объектам базы данных. В этой системе каждая роль может выступать как в роли пользователя, так и в роли группы пользователей.

# Модель доступа

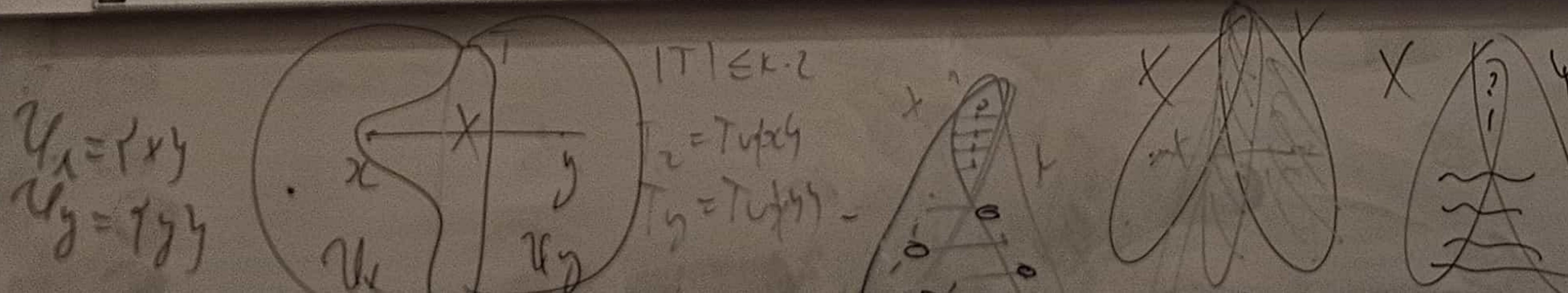
## Роли

- В PostgreSQL понятие “пользователь” является частным случаем роли. Каждая роль может иметь или не иметь возможность входа в систему (login privilege). Если у роли установлено свойство LOGIN, она может использоваться как учетная запись пользователя.
- Роли могут быть членами других ролей. Это позволяет организовать группы пользователей – например, создать роль developers, которой будут принадлежать все разработчики, и назначить этой роли общие привилегии.

# Модель доступа

## Роли

- Создание CREATE ROLE {Name} {attr1, attr2, ...}
- Изменение ALTER ROLE developer WITH PASSWORD 'newsecret';
- LOGIN
- SUPERUSER
- CREATEDB
- CREATEROLE
- PASSWORD 'secret'



# Модель доступа

## Наследование привилегий

- роль, являющаяся членом другой роли, по умолчанию автоматически наследует все привилегии (права доступа) этой родительской роли.
- GRANT role1 to role2
- REVOKE role1 FROM role2

# Модель доступа

## Смена владельца

- ALTER {obj} OWNER TO {role1}
- Владеет роли и роли включающиеся в нее

# Модель доступа

## Привилегии таблиц

- SELECT
- INSERT
- UPDATE
- REFERENCES
- DELETE
- TRUNCATE
- TRIGGER

## Оффтоп Truncate

- TRUNCATE не проверяет каждую строку для удаления, а просто освобождает данные, удаляя содержимое таблицы целиком.
- Вместо того, чтобы логировать каждое удаление строки, TRUNCATE записывает в журнал меньший объем информации.
- При использовании TRUNCATE не вызываются триггеры, определенные для операций удаления.
- Если TRUNCATE выполняется в транзакционном блоке, то операция может быть отменена командой ROLLBACK до фиксации транзакции.

# Модель доступа

## Привилегии БД

- CREATE
- CONNECT
- TEMPORARY

# Модель доступа

## Привилегии схем

- CREATE
- USAGE
- Права доступа к схеме влияют на возможность создания новых объектов, но сами по себе не контролируют доступ к уже существующим объектам. Для этого необходимо назначать соответствующие привилегии непосредственно на уровне объектов.

## Оффтоп

### Схема public

- Создаётся по умолчанию при создании новой базы данных.
- Объекты, в ней, доступны всем пользователям, если не настроены отдельные ограничения.
- При создании объектов не указана какая-либо другая схема, они автоматически создаются в public.

## Оффтоп

### Стандартные БД

- Когда создаётся новая база данных без явного указания шаблона, используется база данных template1.
- Template0 – это «чистая» база данных, которая не изменилась после установки PostgreSQL. Она содержит минимальное количество объектов и служит эталоном.

# Мандатный доступ



- Мандатный уровень доступа (Mandatory Access Control - MAC) используется в системах, где требуется высший уровень безопасности и централизованный контроль. В отличие от дискреционной модели, где владелец решает, кому предоставить доступ, в MAC доступ определяется системными политиками.

- "Нет права давать права"
- Протоколирование ВСЕХ действий
- Защита от копирования

# Шифрование БД

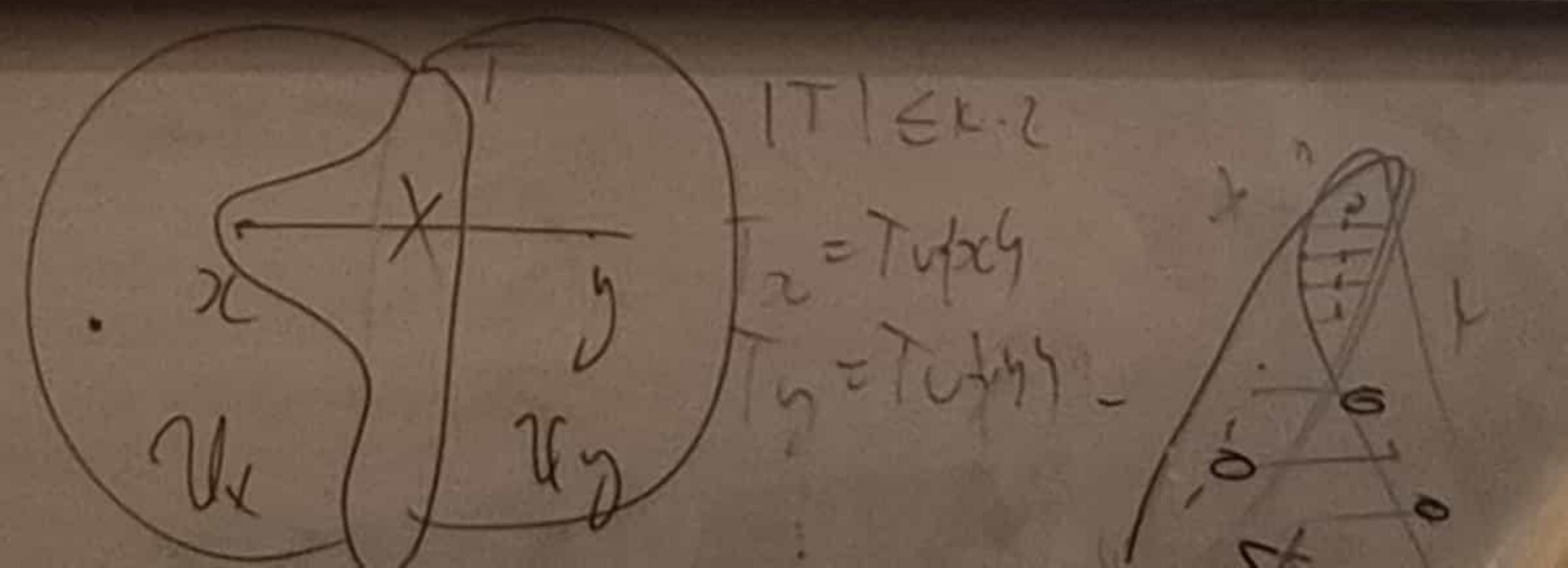
- Прозрачное шифрование
- На уровне столбцов
- На уровне приложения

## УГРОЗА ДЕЙСТВИЙ ПРИВИЛЕГИРОВАННЫХ ПОЛЬЗОВАТЕЛЕЙ

- Принцип разделения обязанностей
- Принцип наименьших привилегий

$$\begin{aligned} \chi_1 &= \Gamma_{xy} \\ \chi_y &= \Gamma_{yy} \end{aligned}$$

IT EK2

$$\begin{aligned} \Gamma_x &= T_{0x} \\ \Gamma_y &= T_{0y} \end{aligned}$$


# Плюсы

- Инкапсулирование функциональности
- Изоляция пользователей от таблиц.
- В некоторых случаях ускорение выполнения (Предкомпиляция)

$$\begin{aligned} \psi_1 &= f(x) \\ \psi_2 &= f(y) \end{aligned}$$

IT16K.2

$$\begin{aligned} x &= T_1 x_1 \\ y &= T_2 y_1 \end{aligned}$$



## В чем отличие?

- Функция имеет возвращаемый тип и возвращает значение
- Использование DML запросов внутри функции невозможно. В функциях разрешены только SELECT-запросы
- Вызов хранимой процедуры из функции невозможно
- Вызов функции внутри SELECT запросов возможен
- Может не возвращать значение или возвращать через OUTPUT параметры
- Использование DML-запросов возможно в хранимой процедуре.
- Использование или же управление транзакциями возможно в хранимой процедуре
- Вызов хранимой процедуры из SELECT запросов невозможно

# Создание функции

```
1 CREATE OR REPLACE FUNCTION get_table_count()
2 RETURNS BIGINT
3 AS $$
4   SELECT COUNT(*) FROM your_table_name;
5 $$ LANGUAGE SQL;
```

# Создание функции

```
1 CREATE OR REPLACE FUNCTION get_student_id(_group_id integer)
2 RETURNS TABLE(student_id integer)
3 AS $$ 
4 BEGIN
5   RETURN QUERY
6   SELECT id FROM student
7   WHERE group_id = _group_id;
8 END;
9 $$ LANGUAGE plpgsql;
10
11 -- Вызов функции
12 SELECT * FROM get_student_id(2);
```

# ФУНКЦИИ

- Функции могут состоять из нескольких операторов SQL.
- Возвращается значение из RETURN
- Нельзя использовать операторы управления транзакции (BEGIN, COMMIT, ROLLBACK)
- Нельзя использовать служебные команды (CREATE INDEX)

# Создание функции

```
● ● ●  
1 CREATE FUNCTION add(int, int, int) RETURNS int AS $$  
2 BEGIN  
3     RETURN $1 + $2 + $3;  
4 END;  
5 $$ LANGUAGE plpgsql;
```

## Типы переменных

- IN
- OUT
- INOUT
- RETURN

$$\begin{aligned} u_1 &= pxy \\ u_2 &= pyy \end{aligned}$$

IT16K.2

$$\begin{aligned} x &= Tu_1x \\ y &= Tu_1y \end{aligned}$$

## Привелегии доступа

- Definer – Создателя
- Invoker – Вызывающего

# Перегрузка операторов

```
1 CREATE FUNCTION add(IN a int, IN b int, IN c int DEFAULT 0) RETURNS int AS $$  
2     SELECT a + b + c;  
3 $$ LANGUAGE sql;  
4  
5 CREATE FUNCTION add(IN a float, IN b int) RETURNS float AS $$  
6     SELECT a + b;  
7 $$ LANGUAGE sql;  
8  
9 CREATE FUNCTION add(IN a float, IN b float) RETURNS float AS $$  
10    SELECT a + b;  
11 $$ LANGUAGE sql;
```

# Полиморфизм

```
CREATE FUNCTION add(IN a anyelement, IN b anyelement, IN c anyelement) RETURNS anyelement
as $$  
SELECT a + b + c;  
$$ LANGUAGE sql;
```

## Категории изменчивости

- **Volatile** — возвращаемое значение может быть произвольно менять на одних и тех же входных.
- **Stable** — Значение не меняется в рамках одного оператора, функция не может менять таблицы (Оптимизатор может **кэшировать** результаты в пределах команды)
- **Immutable** — значение не меняется, функция детерминирована, функция не может менять таблицы (Можно вызывать на этапе **планирования запроса**)

# Создание процедуры с возвращаемым значением

```
CREATE PROCEDURE pay_in() RETURN int  
AS $$  
UPDATE student SET grant = grant + 200;  
SELECT max(grant) FROM student  
$$ LANGUAGE sql;  
  
CALL pay_in();
```

## **PL/pgSQL**

**это загружаемый процедурный язык для системы управления базами данных.**

- может быть использован для создания функций, процедур и триггеров
- добавляет структуры управления к языку SQL
- наследует все пользовательские типы, функции, процедуры и операторы
- прост в использовании

# Форма блока

```
-- Нетка  
DECLARE  
-- Описание переменных  
BEGIN  
-- Тело  
EXCEPTION  
-- Обработка ошибок  
END
```

# STRICT

```
BEGIN
    SELECT * INTO STRICT var FROM student WHERE student_name = my_name;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE EXCEPTION 'Студент % не найден', my_name;
    WHEN TOO_MANY_ROWS THEN
        RAISE EXCEPTION 'Студентов больше одного';
END;
```

# DIAGNOSTICS

```
DECLARE
  r record;
  count int;
BEGIN
  SELECT id, code INTO r FROM table;

  GET DIAGNOSTICS count = row_count;
  RAISE NOTICE 'rowcount = %', count;
END
```

```
<<outer_block>>
DECLARE
    a := 10;
BEGIN
    <<inner_block>>
    DECLARE
        a := 2
        RAISE NOTICE '%', outer_block.a + inner_blkok.a;
    END inner_block;
END outer_block
```

# Подпрограммы PL/pgSQL

```
CREATE FUNCTION add(IN a int, IN b int) RETURNS int
AS $$ 
BEGIN
    RETURN a + b;
END;
$$ LANGUAGE plsql IMMUTABLE;
```

# FOR

```
FOR i IN 1..10 LOOP  
    -- операции  
END LOOP;  
  
FOR i REVERSE 10..1 LOOP  
    -- операции  
END LOOP;
```

# Триггеры

Используется, когда мы хотим действия в момент изменения данных.

```
CREATE [ OR REPLACE ] [ CONSTRAINT ] TRIGGER имя { BEFORE | AFTER | INSTEAD OF } { событие [ OR ... ] }
}
ON имя_таблицы
[ FROM ссылающаяся_таблица ]
[ NOT DEFERRABLE | [ DEFERRABLE ] [ INITIALLY IMMEDIATE | INITIALLY DEFERRED ] ]
[ REFERENCING { { OLD | NEW } TABLE [ AS ] имя_переходного_отношения } [ ... ] ]
[ FOR [ EACH ] { ROW | STATEMENT } ]
[ WHEN ( условие ) ]
EXECUTE { FUNCTION | PROCEDURE } имя_функции ( аргументы )
```

# Триггер

- BEFORE/AFTER - Выполняется перед действием или после. Если используем перед, то мы можем менять данные.
- FROM - на какую то таблицу
- FOR ROW/STATEMENT - На весь запрос целиком или на каждую строчку. У стейтмента триггера нет информации об измененных данных
- WHEN - содержит условие, определяющее будет ли вызываться функция или нет.

# Переменные триггера

- NEW - Содержит новую строку
- OLD - Содержит старую строку
- TG\_NAME - Содержит название триггера
- TG\_OP - Содержит название операции
- TG\_TABLE\_NAME - Содержит название таблицы

# Пример триггера

```
CREATE TRIGGER check_update  
BEFORE UPDATE ON accounts  
FOR EACH ROW  
WHEN (OLD.balance IS DISTINCT FROM NEW.balance)  
EXECUTE FUNCTION check_account_update();
```