

---

# **CS2102 Database Systems**

---

---

# Last Lecture

- We have talked about SQL functions in the last lecture

```
CREATE OR REPLACE FUNCTION MarkCnt ()  
RETURNS TABLE(Mark INT, Cnt INT) AS $$  
    SELECT Mark, COUNT(*)  
    FROM Scores  
    GROUP BY Mark ;  
$$ LANGUAGE sql;
```

# This Lecture

- We will talk about **trigger functions**
  - A special type of functions

```
CREATE OR REPLACE FUNCTION scores_log_func()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO AUDIT(Name, EntryDate)
    VALUES (new.Name, CURRENT_DATE);
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

# Motivating Example

**Scores**

<u>Name</u>	Mark
Alice	92
Bob	63
Cathy	58
David	47

**Scores\_Log**

<u>Name</u>	EntryDate
Alice	2021-03-01
Bob	2021-03-09
Cathy	2021-03-12
David	2021-03-15

- Suppose that we want to implement the following functionality:
  - Whenever there is new tuple inserted into Scores, we insert a tuple into Scores\_Log to record
    - The name of the student, and
    - The date of the insertion

# Motivating Example

**Scores**

<u>Name</u>	Mark
Alice	92
Bob	63
Cathy	58
David	47

**Scores\_Log**

<u>Name</u>	EntryDate
Alice	2021-03-01
Bob	2021-03-09
Cathy	2021-03-12
David	2021-03-15


- Idea: Create a function for this purpose

```
CREATE OR REPLACE FUNCTION scores_log_func() RETURNS VOID AS $$
BEGIN
    IF (there is an insertion into Scores) THEN
        INSERT INTO Scores_Log(Name, EntryDate)
            VALUES (New_Name, CURRENT_DATE);
    END IF;
END;
$$ LANGUAGE plpgsql;
```

# Motivating Example

- Two issues to address:
  - We need a way to express this condition about "an insertion occurring on Scores"
  - We need the database to check this condition whenever appropriate


```
CREATE OR REPLACE FUNCTION scores_log_func() RETURNS VOID AS $$  
BEGIN  
    IF (there is an insertion into Scores) THEN  
        INSERT INTO Scores_Log(Name, EntryDate)  
            VALUES (New_Name, CURRENT_DATE);  
    END IF;  
END;  
$$ LANGUAGE plpgsql;
```



# Motivating Example

- Two issues to address:
  - We need a way to express this condition about "an insertion occurring on Scores" - **trigger functions**
  - We need the database to check this condition whenever appropriate - **triggers**

```
CREATE OR REPLACE FUNCTION scores_log_func() RETURNS VOID AS $$  
BEGIN  
    IF (there is an insertion into Scores) THEN  
        INSERT INTO Scores_Log(Name, EntryDate)  
            VALUES (New_Name, CURRENT_DATE);  
    END IF;  
END;  
$$ LANGUAGE plpgsql;
```



# Motivating Example

Scores		Scores_Log	
<u>Name</u>	Mark	<u>Name</u>	EntryDate
...	...	...	...

## ■ The trigger:

```
CREATE TRIGGER scores_log_trigger
AFTER INSERT ON Scores
FOR EACH ROW EXECUTE FUNCTION score_log_func();
```

## ■ The trigger function:

```
CREATE OR REPLACE FUNCTION scores_log_func() RETURNS TRIGGER
AS $$
BEGIN
    INSERT INTO Scores_Log(Name, EntryDate)
        VALUES (NEW.Name, CURRENT_DATE);
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```



# Motivating Example

Scores		Scores_Log	
<u>Name</u>	Mark	<u>Name</u>	EntryDate
...	...	...	...

- The trigger:

```
CREATE TRIGGER scores_log_trigger  
AFTER INSERT ON Scores  
FOR EACH ROW EXECUTE FUNCTION score_log_func();
```

- This tells the database to
  - Watch out for insertions on Score, and
  - Call the score\_log\_func() function after each insertion of a tuple
- There are other options for the keywords in **blue**. We will discuss this shortly.

# Motivating Example

Scores		Scores_Log	
<u>Name</u>	Mark	<u>Name</u>	EntryDate
...	...	...	...

- "RETURNS **TRIGGER**" indicates that this is a trigger function
- **NEW** refers to the new row inserted into Scores
- **CURRENT\_DATE** returns the current date
- The trigger function:

```
CREATE OR REPLACE FUNCTION score_log_func() RETURNS TRIGGER
AS $$
BEGIN
    INSERT INTO Scores_Log(Name, EntryDate)
        VALUES (NEW.Name, CURRENT_DATE);
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

# Motivating Example

Scores	
Name	Mark
...	...

Scores_Log	
Name	EntryDate
...	...

- Can it be "RETURNS RECORD"?
- No. Only "RETRUNS **TRIGGER**" is allowed.
- Rationale: The function needs to access **NEW**. Only **TRIGGER** function has such accesses.
- The trigger function:

```
CREATE OR REPLACE FUNCTION score_log_func() RETURNS TRIGGER
AS $$
BEGIN
    INSERT INTO Scores_Log(Name, EntryDate)
        VALUES (NEW.Name, CURRENT_DATE);
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

# Motivating Example

- What else can a trigger function access?
  - **TG\_OP**: the operation that activates the trigger, i.e., 'INSERT', or 'UPDATE', or 'DELETE', ...
  - **TG\_TABLE\_NAME**: the name of table that caused the trigger invocation
  - **OLD**: the old tuple being updated/deleted
  - And a lot of other contextual information...

```
CREATE OR REPLACE FUNCTION score_log_func() RETURNS TRIGGER
AS $$
BEGIN
    INSERT INTO Scores_Log(Name, EntryDate)
        VALUES (NEW.Name, CURRENT_DATE);
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

## Example 2

**Scores**

<u>Name</u>	Mark
Alice	92
Bob	63
Cathy	58
David	47

**Scores\_Log2**

<u>Name</u>	Op	OpDate
Alice	Insert	2021-03-01
Bob	Delete	2021-03-09
Cathy	Update	2021-03-12
David	Insert	2021-03-15

- What we want:
  - Whenever there is an insert/delete/update on Scores, we insert a tuple into Scores\_Log2 to record
    - The name of the student, and
    - The operation performed, and
    - The date of the operation

# Example 2

## ■ The trigger function:

Scores		Scores_Log2		
<u>Name</u>	Mark	<u>Name</u>	Op	OpDate
...	...	...	...	

```
CREATE OR REPLACE FUNCTION scores_log2_func() RETURNS TRIGGER AS $$
BEGIN
    IF (TG_OP = 'INSERT') THEN
        INSERT INTO Scores_Log2 SELECT NEW.Name, 'Insert', CURRENT_DATE;
        RETURN NEW;
    ELSEIF (TG_OP = 'DELETE') THEN
        INSERT INTO Scores_Log2 SELECT OLD.Name, 'Delete', CURRENT_DATE;
        RETURN OLD;
    ELSIF (TG_OP = 'UPDATE') THEN
        INSERT INTO Scores_Log2 SELECT NEW.Name, 'Update', CURRENT_DATE;
        RETURN NEW;
    END IF;
END;
$$ LANGUAGE plpgsql;
```

# Example 2

Scores		Scores_Log2		
Name	Mark	Name	Op	OpDate
...	...	...	...	

## ■ The trigger function:

```
CREATE OR REPLACE FUNCTION scores_log2_func() RETURNS TRIGGER AS $$
BEGIN
    IF (TG_OP = 'INSERT') THEN
        INSERT INTO Scores_Log2 SELECT NEW.Name, 'Insert', CURRENT_DATE;
        RETURN NEW;
    ELSEIF (TG_OP = 'DELETE') THEN
        INSERT INTO Scores_Log2 SELECT OLD.Name, 'Delete', CURRENT_DATE;
        RETURN OLD;
    ELSIF (TG_OP = 'UPDATE') THEN
        INSERT INTO Scores_Log2 SELECT NEW.Name, 'Update', CURRENT_DATE;
        RETURN NEW;
    END IF;
END;
$$ LANGUAGE plpgsql;
```

- Can it be "RETURN NULL" here?
- It depends
- We will discuss this shortly

## Example 2

Scores		Scores_Log2		
<u>Name</u>	Mark	<u>Name</u>	Op	OpDate
...	...	...	...	

### ■ The trigger function:

```
CREATE OR REPLACE FUNCTION scores_log2_func() RETURNS TRIGGER AS $$
BEGIN
    IF (TG_OP = 'INSERT') THEN ...
    ELSEIF (TG_OP = 'DELETE') THEN ...
    ELSIF (TG_OP = 'UPDATE') THEN ...
    END IF;
    ...

```

### ■ The trigger:

```
CREATE TRIGGER scores_log2_trigger
AFTER INSERT OR DELETE OR UPDATE ON Scores
FOR EACH ROW EXECUTE FUNCTION scores_log2_func();

```



---

# Trigger Timing

```
CREATE TRIGGER scores_log_trigger  
AFTER INSERT ON Scores  
FOR EACH ROW EXECUTE FUNCTION scores_log_func();
```

- AFTER indicates that score\_log\_func() is executed after the insertion on Scores is done
- Two other options:
  - BEFORE: scores\_log\_func() would be executed before the insertion
  - INSTEAD OF: scores\_log\_func() would be executed instead of the insertion

# BEFORE Trigger Example

Scores
--------

Name	Mark
------	------

...
-----

...
-----

## ■ The trigger:

```
CREATE TRIGGER for_Elise_trigger
BEFORE INSERT ON Scores
FOR EACH ROW EXECUTE FUNCTION for_Elise_func();
```

## ■ The trigger function:

```
CREATE OR REPLACE FUNCTION for_Elise_func() RETURNS TRIGGER AS $$
BEGIN
    IF (NEW.Name = 'Elise') THEN
        NEW.Mark := 100;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

# BEFORE Trigger Example

Scores	
Name	Mark
Elise	100

## ■ The trigger:

```
CREATE TRIGGER for_Elise_trigger
BEFORE INSERT ON Scores
FOR EACH ROW EXECUTE FUNCTION for_Elise_func();
```

## ■ The trigger function:

```
CREATE OR REPLACE FUNCTION for_Elise_func() RETURNS TRIGGER AS $$
BEGIN
    IF (NEW.Name = 'Elise') THEN
        NEW.Mark := 100;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

- Effect: Elise's mark would be 100 regardless of what we insert

# BEFORE Trigger Example

Scores
--------

Name	Mark
------	------

...
-----

...
-----

## ■ The trigger:

```
CREATE TRIGGER for_Elise_trigger
BEFORE INSERT ON Scores
FOR EACH ROW EXECUTE FUNCTION for_Elise_func();
```

## ■ The trigger function:

```
CREATE OR REPLACE FUNCTION for_Elise_func() RETURNS TRIGGER AS $$
BEGIN
    IF (NEW.Name = 'Elise') THEN
        NEW.Mark := 100;
    END IF;
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

# BEFORE Trigger Example

Scores

Name	Mark
------	------

...

...

## ■ The trigger:

```
CREATE TRIGGER for_Elise_trigger
BEFORE INSERT ON Scores
FOR EACH ROW EXECUTE FUNCTION for_Elise_func();
```

## ■ The trigger function:

```
CREATE OR REPLACE FUNCTION for_Elise_func() RETURNS TRIGGER AS $$
BEGIN
    IF (NEW.Name = 'Elise') THEN
        NEW.Mark := 100;
    END IF;
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

- Effect: No tuple can be inserted
- Reason: RETURN NULL in a BEFORE trigger tells the database to ignore the operation

# BEFORE Trigger Example

Scores	
Name	Mark
...	...

## ■ The trigger:

```
CREATE TRIGGER for_Elise_trigger
BEFORE INSERT ON Scores
FOR EACH ROW EXECUTE FUNCTION for_Elise_func();
```

## ■ The trigger function:

```
CREATE OR REPLACE FUNCTION for_Elise_func() RETURNS TRIGGER AS $$
BEGIN
    IF (NEW.Name = 'Elise') THEN
        NEW.Mark := 100;
    END IF;
    RETURN OLD;
END;
$$ LANGUAGE plpgsql;
```

# BEFORE Trigger Example

Scores

Name	Mark
------	------

...

...

## ■ The trigger:

```
CREATE TRIGGER for_Elise_trigger
BEFORE INSERT ON Scores
FOR EACH ROW EXECUTE FUNCTION for_Elise_func();
```

## ■ The trigger function:

```
CREATE OR REPLACE FUNCTION for_Elise_func() RETURNS TRIGGER AS $$
BEGIN
    IF (NEW.Name = 'Elise') THEN
        NEW.Mark := 100;
    END IF;
    RETURN OLD;
END;
$$ LANGUAGE plpgsql;
```

- Effect: No tuple can be inserted
- Reason:
  - For INSERT, OLD is initially set to NULL
  - RETURN OLD is the same as RETURN NULL

# BEFORE Trigger Example

Scores	
Name	Mark
...	...

## ■ The trigger:

```
CREATE TRIGGER for_Elise_trigger
BEFORE INSERT ON Scores
FOR EACH ROW EXECUTE FUNCTION for_Elise_func();
```

## ■ The trigger function:

```
CREATE OR REPLACE FUNCTION for_Elise_func() RETURNS TRIGGER AS $$
BEGIN
    OLD.Name := 'Haha';
    OLD.Mark := 0;
    RETURN OLD;
END;
$$ LANGUAGE plpgsql;
```



# BEFORE Trigger Example

Scores	
Name	Mark
...	...

## ■ The trigger:

```
CREATE TRIGGER for_Elise_trigger
BEFORE INSERT ON Scores
FOR EACH ROW EXECUTE FUNCTION for_Elise_func();
```

## ■ The trigger function:

```
CREATE OR REPLACE FUNCTION for_Elise_func() RETURNS TRIGGER AS $$
BEGIN
    OLD.Name := 'Haha';
    OLD.Mark := 0;
    RETURN OLD;
END;
$$ LANGUAGE plpgsql;
```

- Effect: ('Haha', 0) will be inserted
- Reason:
  - Whenever the function returns a non-null tuple, the trigger would use it as the tuple to be inserted

# Return Values of Trigger Functions

- For a BEFORE INSERT trigger:
  - Returning a non-null tuple  $t$ :  $t$  will be inserted
  - Returning a null tuple: no tuple will be inserted
- For a BEFORE UPDATE trigger:
  - Returning a non-null tuple  $t$ :  $t$  will be the updated tuple
  - Returning a null tuple: no tuple will be updated
- For a BEFORE DELETE trigger:
  - Returning a non-null tuple  $t$ : deletion proceeds as normal
  - Returning a null tuple: no deletion will be performed

---

# Return Values of Trigger Functions

- For an **AFTER** INSERT trigger:
  - The return value does not matter
- For an **AFTER** UPDATE trigger:
  - The return value does not matter
- For an **AFTER** DELETE trigger:
  - The return value does not matter
  
- Reason: The trigger function is invoked **after** the main operation is done

# INSTEAD OF Trigger

- This kind of trigger can be defined on views only
- Typically usage:
  - Instead of doing something on a view, do it on a table
- Example below:
  - CREATE OR REPLACE VIEW Max\_Score AS  
SELECT \* FROM Scores ORDER BY Mark DESC LIMIT 1;

Scores	<u>Name</u>	Mark
	Alice	92
	Bob	63
	Cathy	58
	David	47

Max_Score	<u>Name</u>	Mark
	Alice	92

# INSTEAD OF Trigger

- What we want:
  - Whenever someone wants to update the tuple in Max\_Score, we update the corresponding tuple in Scores
- We will use an INSTEAD OF trigger

Scores	<u>Name</u>	Mark
	Alice	92
	Bob	63
	Cathy	58
	David	47

Max_Score	<u>Name</u>	Mark
	Alice	92

# INSTEAD OF Trigger Example

Max_Score	
Name	Mark
...	...

## ■ The trigger:

```
CREATE TRIGGER update_max_trigger
INSTEAD OF UPDATE ON Max_Score
FOR EACH ROW EXECUTE FUNCTION update_max_func();
```

## ■ The trigger function:

```
CREATE OR REPLACE FUNCTION update_max_func() RETURNS TRIGGER AS $$
BEGIN
    UPDATE Scores
    SET Mark = NEW.Mark WHERE Name = OLD.Name;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

---

# Return Values of Trigger Functions

- For an INSTEAD OF trigger:
  - Returning NULL signals the database to ignore all operations on the current row
  - Returning a non-null tuple signals the database to proceed as normal

# Trigger Levels

```
CREATE TRIGGER scores_log_trigger  
AFTER INSERT ON Scores  
FOR EACH ROW EXECUTE FUNCTION score_log_func();
```

- This is a **row-level** trigger that executes the trigger function for every tuple encountered
- The alternative: a **statement-level** trigger that executes the trigger function only once
- Example:

```
CREATE TRIGGER ....  
AFTER INSERT ON ....  
FOR EACH STATEMENT EXECUTE FUNCTION ....
```



# Statement-Level Trigger Example

## ■ The trigger:

```
CREATE TRIGGER del_warn_trigger
BEFORE DELETE ON Scores_Log
FOR EACH STATEMENT EXECUTE FUNCTION del_warn_func();
```

Scores\_Log

Name	EntryDate
...	...

## ■ The trigger function:

```
CREATE OR REPLACE FUNCTION del_warn_func() RETURNS TRIGGER
AS $$
BEGIN
    RAISE NOTICE 'You are not supposed to delete from the log.';
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

- Effect: The database will prompt 'You are ...' whenever a deletion is attempted on Scores\_Log

# Statement-Level Trigger Example

## ■ The trigger:

```
CREATE TRIGGER del_warn_trigger  
BEFORE DELETE ON Scores_Log  
FOR EACH STATEMENT EXECUTE FUNCTION del_warn_func();
```

Scores\_Log

Name	EntryDate
...	...

## ■ The trigger function:

```
CREATE OR REPLACE FUNCTION del_warn_func() RETURNS TRIGGER  
AS $$  
BEGIN  
    RAISE NOTICE 'You are not supposed to delete from the log.';  
    RETURN NULL;  
END;  
$$ LANGUAGE plpgsql;
```

- Does this prevent the deletion?
- No

---

# Return Values of Trigger Functions

- Statement-level triggers ignore the values returned by the trigger functions
- So RETURN NULL would not make the database omit the subsequent operations
- What if we want the subsequent operations to be omitted?
- Answer: raise an **exception**

# Statement-Level Trigger Example

- The trigger:

```
CREATE TRIGGER del_warn_trigger  
BEFORE DELETE ON Scores_Log  
FOR EACH STATEMENT EXECUTE FUNCTION del_warn_func();
```

Scores_Log	
Name	EntryDate
...	...

- The trigger function:

```
CREATE OR REPLACE FUNCTION del_warn_func() RETURNS TRIGGER  
AS $$  
BEGIN  
    RAISE EXCEPTION 'No deletion from the log is allowed.';  
    RETURN NULL;  
END;  
$$ LANGUAGE plpgsql;
```

---

# Trigger Timing vs. Levels

- **INSTEAD OF** is only allowed on row-level
- **BEFORE/AFTER** are allowed on both row-level and statement-level

# Trigger Condition

Scores	
Name	Mark
...	...

## ■ The trigger:

```
CREATE TRIGGER for_Elise_trigger
BEFORE INSERT ON Scores
FOR EACH ROW EXECUTE FUNCTION for_Elise_func();
```

## ■ The trigger function:

```
CREATE OR REPLACE FUNCTION for_Elise_func() RETURNS TRIGGER AS $$
BEGIN
    IF (NEW.Name = 'Elise') THEN
        NEW.Mark := 100;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

- Observation: The trigger function only cares about the case when NEW.Name = 'Elise'
- We can move this condition to the trigger definition

# Trigger Condition

Scores	
Name	Mark
...	...

## ■ The trigger:

```
CREATE TRIGGER for_Elise_trigger
BEFORE INSERT ON Scores
FOR EACH ROW
WHEN (NEW.Name = 'Elise')
EXECUTE FUNCTION for_Elise_func();
```

## ■ The trigger function:

```
CREATE OR REPLACE FUNCTION for_Elise_func() RETURNS TRIGGER AS $$
BEGIN
    NEW.Mark := 100;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

# Trigger Condition

Scores	
<u>Name</u>	Mark
...	...

- The trigger:

```
CREATE TRIGGER for_Elise_trigger
BEFORE INSERT ON Scores
FOR EACH ROW
WHEN (NEW.Name = 'Elise')
EXECUTE FUNCTION for_Elise_func();
```

- In general, the condition in WHEN() could be more complicated, subject to the following requirements:
  - ❑ No SELECT in WHEN ()
  - ❑ No OLD in WHEN () for INSERT
  - ❑ No NEW in WHEN () for DELETE
  - ❑ No WHEN for INSTEAD OF



# Deferred Trigger

- There are scenarios where we need to **defer** the checking of triggers
- Example:
  - We have a trigger on insert/update/delete that checks the total balance of each customer's account
  - Requirement: total balance should be at least 150


## Account

<u>AID</u>	Name	Bal
1	Alice	100
2	Alice	100

# Deferred Trigger

- Trigger requirement: total balance should be at least 150
- Suppose that Alice wants to transfer 100 from Account 1 to Account 2
- We use two update statements:
  - One to deduct 100 from Account 1
  - One to add 100 to Account 2
- Problem: The trigger requirement is violated after the first update statement

Account						Account		
AID	Name	Bal				AID	Name	Bal
1	Alice	100				1	Alice	0
2	Alice	100				2	Alice	100



Account						Account		
AID	Name	Bal				AID	Name	Bal
1	Alice	0				1	Alice	0
2	Alice	200				2	Alice	200

# Deferred Trigger

- Trigger requirement: total balance should be at least 150
- Problem: The trigger requirement is violated after the first update statement
- Solution:
  - Put the two update statements into one transaction
  - Defer the trigger check to the end of the transaction

Account		
<u>AID</u>	Name	Bal
1	Alice	100
2	Alice	100

⇒

Account		
<u>AID</u>	Name	Bal
1	Alice	0
2	Alice	100

⇒

Account		
<u>AID</u>	Name	Bal
1	Alice	0
2	Alice	200


# Deferred Trigger

```
CREATE CONSTRAINT TRIGGER bal_check_trigger  
AFTER INSERT OR UPDATE OR DELETE ON Account  
DEFERRABLE INITIALLY DEFERRED  
FOR EACH ROW  
EXECUTE FUNCTION bal_check_func();
```

- **CONSTRAINT** and **DEFERRABLE** together indicate that the trigger can be deferred
- **INITIALLY DEFERRED** indicates that by default, the trigger is deferred
  - Other option: **INITIALLY IMMEDIATE**, i.e., the trigger is not deferred by default

# Deferred Trigger

```
CREATE CONSTRAINT TRIGGER bal_check_trigger  
AFTER INSERT OR UPDATE OR DELETE ON Account  
DEFERRABLE INITIALLY DEFERRED  
FOR EACH ROW  
EXECUTE FUNCTION bal_check_func();
```

- Deferred triggers only work with **AFTER** and **FOR EACH ROW**
- Why?
  - **AFTER**: To defer the trigger, it has to be allowed to execute after the main operation
  - **FOR EACH ROW**: I don't know... 

# Deferred Trigger Example

Account		
AID	Name	Bal
1	Alice	100
2	Alice	100

```
CREATE CONSTRAINT TRIGGER bal_check_trigger
AFTER INSERT OR UPDATE OR DELETE ON Account
DEFERRABLE INITIALLY DEFERRED
FOR EACH ROW
EXECUTE FUNCTION bal_check_func();
```

- With the above deferred trigger, we can do the following:

```
BEGIN TRANSACTION;
UPDATE Account SET Bal = Bal - 100 WHERE AID = 1;
UPDATE Account SET Bal = Bal + 100 WHERE AID = 2;
COMMIT;
```

- The trigger will be activated at "COMMIT"

# Deferred Trigger Example

## Account

AID	Name	Bal
1	Alice	100
2	Alice	100

```
CREATE CONSTRAINT TRIGGER bal_check_trigger  
AFTER INSERT OR UPDATE OR DELETE ON Account  
DEFERRABLE INITIALLY IMMEDIATE  
FOR EACH ROW  
EXECUTE FUNCTION bal_check_func();
```

- What if the trigger is "initially immediate"?
- Answer: we just need to change it on the fly

```
BEGIN TRANSACTION;  
UPDATE Account SET Bal = Bal - 100 WHERE AID = 1;  
UPDATE Account SET Bal = Bal + 100 WHERE AID = 2;  
COMMIT;
```

- The trigger will be activated at "COMMIT"

# Deferred Trigger Example

Account		
AID	Name	Bal
1	Alice	100
2	Alice	100

```
CREATE CONSTRAINT TRIGGER bal_check_trigger  
AFTER INSERT OR UPDATE OR DELETE ON Account  
DEFERRABLE INITIALLY IMMEDIATE  
FOR EACH ROW  
EXECUTE FUNCTION bal_check_func();
```

- What if the trigger is "initially immediate"?
- Answer: we just need to change it on the fly

```
BEGIN TRANSACTION;  
SET CONSTRAINTS bal_check_trigger DEFERRED;  
UPDATE Account SET Bal = Bal - 100 WHERE AID = 1;  
UPDATE Account SET Bal = Bal + 100 WHERE AID = 2;  
COMMIT;
```

- The trigger will be activated at "COMMIT"



---

# Multiple Triggers

- There can be multiple triggers defined for the same event on the same table
  - Order of trigger activation
    - BEFORE statement-level triggers
    - BEFORE row-level triggers
    - AFTER row-level triggers
    - AFTER statement-level triggers
  - Within each category, triggers are activated in alphabetic order
  - If a BEFORE row-level trigger returns NULL, then subsequent triggers on the same row are omitted
-

---

# Note

- Our discussions about triggers are based on PostgreSQL's syntax and implementation
- Different databases have different syntaxes and implementations