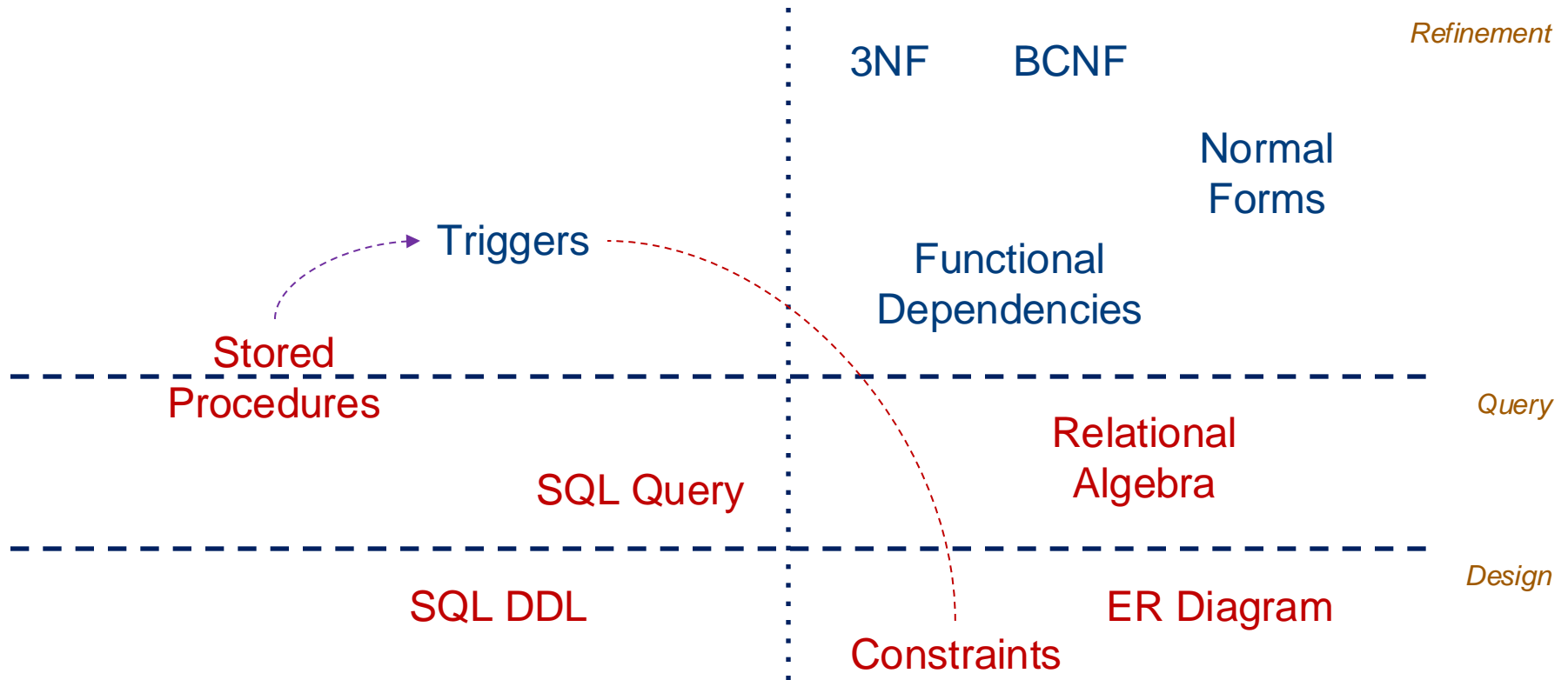


CS2102 Database Systems

Lecture 8 – Triggers
Instructor: Jiang Kan

Slides adapted from Prof Adi Yoga Sidi Prabawa

Roadmap



Previous Lectures

- **SQL Functions and Procedures**

- Named operations
- Expressive control structures

- **Constraints**

- Key constraints
- Referential constraints
- Uniqueness
- Check

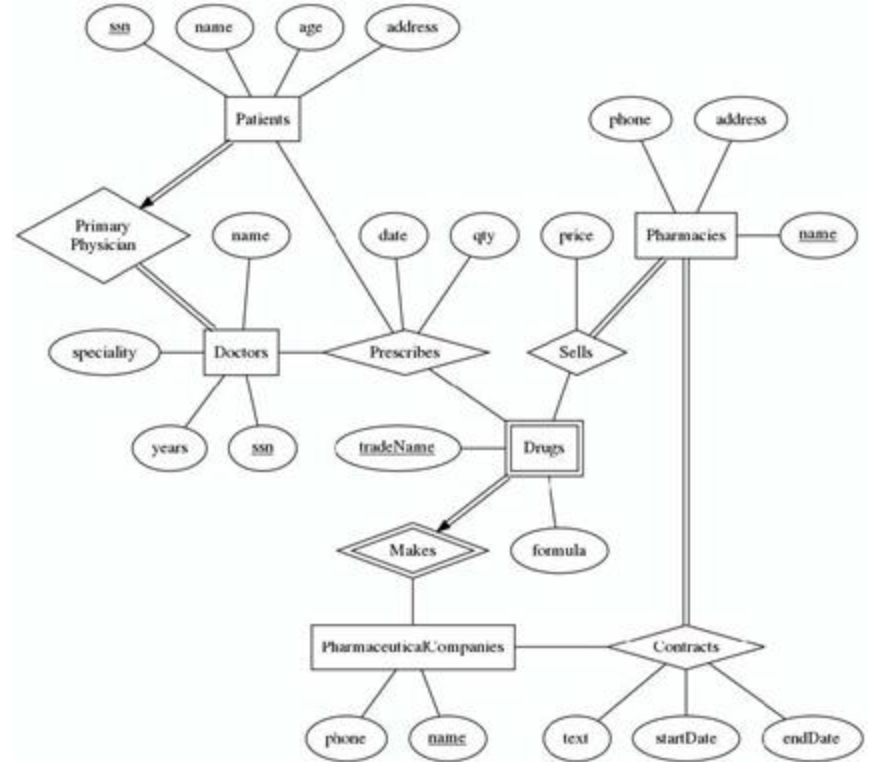
- What if there are constraints that cannot be handled by the constructs above?

Difficult Constraints

- **Tutorial ??**

- There is exactly one contract between a pharmacy and a pharmaceutical company if and only if that pharmacy sells some drug that is made by that pharmaceutical company.

This is quite complex, instead, we will use a simpler motivating example but Trigger is powerful enough to handle this



Triggers

Triggers

- **Simple Motivating Example**

Table "Scores"

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

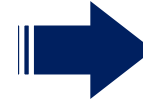


Table "Scores_Log"

<u>Name</u>	Date
Alice	2021-10-01
Bob	2021-10-09
Cathy	2021-10-12
David	2021-10-15

- Suppose we want to log when the marks are entered
 - This should be done automatically
 - The user of the database should not be bothered to write SQL statement to insert
 - This should be done *each time* an insertion occurs regardless of how it is done
 - We want to record the following data
 - The name of the student
 - The date of entry

Triggers

- **Simple Motivating Example**

Table "Scores"

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

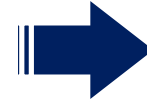


Table "Scores_Log"

<u>Name</u>	Date
Alice	2021-10-01
Bob	2021-10-09
Cathy	2021-10-12
David	2021-10-15

- Suppose we want to log when the marks are entered

- **Idea?**

- A procedure that enters both data
 - What if the user forgot to use the procedure?

```
CREATE OR REPLACE PROCEDURE enter_data
    (Name TEXT, Mark INT)
AS $$
    INSERT INTO Scores VALUES (Name, Mark);
    INSERT INTO Scores_Log
        VALUES (Name, CURRENT_DATE);
$$ LANGUAGE sql;
```

Triggers

- Other Motivating Example

Table "Scores"

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



Table "Scores_Log"

<u>Name</u>	Date
Alice	2021-10-01
Bob	2021-10-09
Cathy	2021-10-12
David	2021-10-15

- Suppose we want to log when the marks are entered

- What we want?

- A procedure to insert into Scores_Log
 - But called *automatically*

```
CREATE OR REPLACE PROCEDURE log_score()  
AS $$  
BEGIN  
    IF (there is an insertion into Scores) THEN  
        INSERT INTO Scores_Log  
            VALUES (Name, CURRENT_DATE);  
    END IF;  
END;  
$$ LANGUAGE plpgsql;
```


Triggers

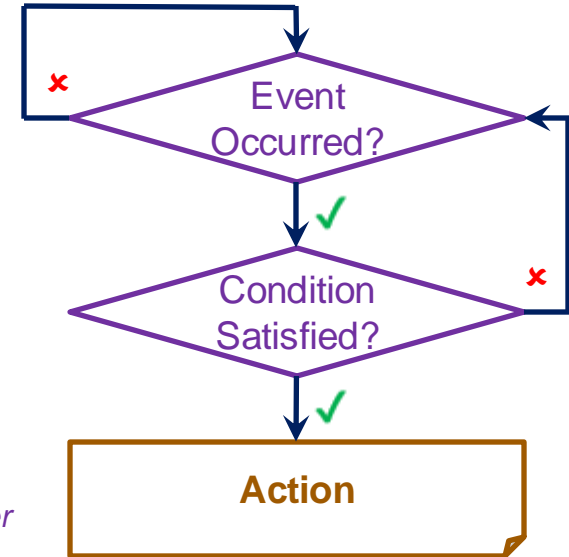
- **Basic of Triggers**

- Trigger is an **event-condition-action** (ECA) rule
 - When **event** occurs
 - Test **condition**
 - If satisfied, execute **action**

- **Example**

- **Event** New tuple inserted into Scores
- **Condition** Nothing *(always execute action)*
- **Action** Insert into Scores_Log

} Trigger
} Trigger Function



Triggers

- Other Motivating Example

Trigger

```
CREATE TRIGGER score_log  
AFTER INSERT ON Scores  
FOR EACH ROW EXECUTE FUNCTION  
log_score();
```

Table "Scores"

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

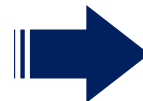


Table "Scores_Log"

<u>Name</u>	Date
Alice	2021-10-01
Bob	2021-10-09
Cathy	2021-10-12
David	2021-10-15

- Suppose we want to log when the marks are entered

- Trigger Function

- Actions to run when event occurred and conditions satisfied

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

Trigger Function

Triggers

- The Trigger Function

- RETURNS **TRIGGER** indicates that this is a trigger function
 - Can only RETURNS **TRIGGER**
- **NEW** refers to the new row inserted into Scores
 - Only accessible by trigger functions
- Other accessible data?
 - **TG_OP**
 - **TG_TABLE_NAME**
 - **OLD**
 - ...

Table "Scores"

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



Table "Scores_Log"

<u>Name</u>	Date
Alice	2021-10-01
Bob	2021-10-09
Cathy	2021-10-12
David	2021-10-15

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

Trigger Function

Triggers Return Value

- **Transition Variables**

- **NEW** The modified row *after* the triggering event
- **OLD** The modified row *before* the triggering event
- ❖ Not all make sense all the time

	NEW	OLD
INSERT	✓	✗
UPDATE	✓	✓
DELETE	✗	✓

- **Effect of Return Value**

	NULL tuple	non-NULL tuple t
BEFORE INSERT	No tuple inserted	Tuple t will be inserted
BEFORE UPDATE	No tuple updated	Tuple t will be the updated tuple
BEFORE DELETE	No deletion performed	Deletion proceeds as normal
AFTER INSERT	<i>Does not matter</i>	
AFTER UPDATE		
AFTER DELETE		

Triggers

- The Trigger

Trigger

```
CREATE TRIGGER score_log  
AFTER INSERT ON Scores  
FOR EACH ROW EXECUTE FUNCTION  
log_score();
```

- This tells the database to

- Watch out for insertion on Score
- ~~Check for certain condition~~
- Call the function log_score() after each insertion of a tuple
- There are other options, but that will be discussed later on

Table "Scores"

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

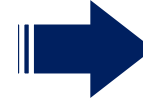


Table "Scores_Log"

<u>Name</u>	Date
Alice	2021-10-01
Bob	2021-10-09
Cathy	2021-10-12
David	2021-10-15

Triggers Event

- **Another Motivating Example**

Table "Scores"

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

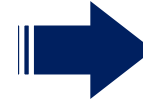


Table "Scores_Log2"

<u>Name</u>	Op	Date
Alice	Insert	2021-10-01
Bob	Delete	2021-10-09
Cathy	Update	2021-10-12
David	Insert	2021-10-15

- Suppose we want to log when the marks are entered
 - This should be done automatically
 - The user of the database should not be bothered to write SQL statement to insert
 - This should be done *each time* a modification occurs regardless of how it is done
 - We want to record the following data
 - The name of the student
 - The operation performed
 - The date of entry

Triggers Event

- The Trigger Function

Trigger Function

```
CREATE OR REPLACE FUNCTION log_score2() RETURNS TRIGGER AS $$
BEGIN
    IF (TG_OP = 'INSERT') THEN
        INSERT INTO Scores_Log2 VALUES (NEW.Name, 'Insert', CURRENT_DATE);
        RETURN NEW;
    ELSIF (TG_OP = 'DELETE') THEN
        INSERT INTO Scores_Log2 VALUES (OLD.Name, 'Delete', CURRENT_DATE);
        RETURN OLD;
    ELSIF (TG_OP = 'UPDATE') THEN
        INSERT INTO Scores_Log2 VALUES (NEW.Name, 'Update', CURRENT_DATE);
        RETURN NEW;
    END IF;
END;
$$ LANGUAGE plpgsql;
```

Triggers Event

- The Trigger Function

```
CREATE OR REPLACE FUNCTION log_score2() RETURNS TRIGGER AS $$  
BEGIN  
    IF      (TG_OP = 'INSERT') THEN ...  
    ELSIF (TG_OP = 'DELETE') THEN ...  
    ELSIF (TG_OP = 'UPDATE') THEN ...  
    END IF;  
END;  
$$ LANGUAGE plpgsql;
```

Trigger Function

- The Trigger

```
CREATE TRIGGER score_log2  
AFTER INSERT OR DELETE OR UPDATE ON Scores  
FOR EACH ROW EXECUTE FUNCTION log_score2();
```

Trigger

Triggers Options

- Trigger Options

- Events:

- **INSERT ON** *table*

- **DELETE ON** *table*

- **UPDATE [OF column] ON** *table*

TG_OP {
'INSERT'
'DELETE'
'UPDATE'

- Timing:

- **AFTER** or **BEFORE** *(the triggering event)*

- **INSTEAD OF** *(the triggering event on views)*

- Granularity:

- **FOR EACH ROW** *(modified)*

- **FOR EACH STATEMENT** *(that performs the modification)*

Triggers Timing

AFTER
BEFORE
INSTEAD OF

Triggers Timing

Table "Scores"

<u>Name</u>	Mark
Adi	100

- The Trigger Function

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

Trigger Function

Effect?

Adi will always get 100

- The Trigger

```
CREATE TRIGGER adi_should_get_full_mark
BEFORE INSERT ON Scores
FOR EACH ROW EXECUTE FUNCTION give_adi_full_mark();
```

Trigger

Triggers Return Value

- Transition Variables

- **NEW** The modified row *after* the triggering event
- **OLD** The modified row *before* the triggering event
- ❖ Not all make sense all the time

	NEW	OLD
INSERT	✓	✗
UPDATE	✓	✓
DELETE	✗	✓

- Effect of Return Value

	NULL tuple	non-NULL tuple <i>t</i>
BEFORE INSERT	No tuple inserted	Tuple <i>t</i> will be inserted
BEFORE UPDATE	No tuple updated	Tuple <i>t</i> will be the updated tuple
BEFORE DELETE	No deletion performed	Deletion proceeds as normal
AFTER INSERT	<i>Does not matter</i>	
AFTER UPDATE		
AFTER DELETE		

Triggers Return Value

Table "Scores"

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

- The Trigger Function

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

Trigger Function

Effect?

No tuple is inserted

- The Trigger

```
CREATE TRIGGER adi_should_get_full_mark
BEFORE INSERT ON Scores
FOR EACH ROW EXECUTE FUNCTION give_adi_full_mark();
```

Trigger



Triggers Return Value

Table "Scores"

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

- The Trigger Function

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

Trigger Function

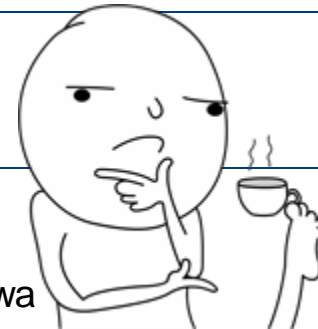
Effect?

Tuple still not inserted

- The Trigger

```
CREATE TRIGGER adi_should_get_full_mark
BEFORE INSERT ON Scores
FOR EACH ROW EXECUTE FUNCTION give_adi_full_mark();
```

Trigger



Triggers Return Value

Table "Scores"

<u>Name</u>	Mark
Adi	100

- The Trigger Function

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

Trigger Function

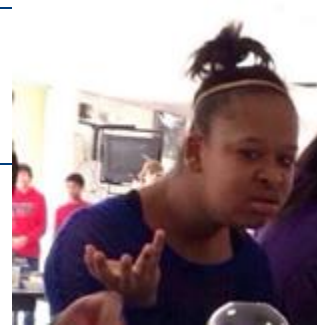
Effect?

Adi will always get 100
(Others will not be inserted)

- The Trigger

```
CREATE TRIGGER adi_should_get_full_mark
BEFORE INSERT ON Scores
FOR EACH ROW EXECUTE FUNCTION give_adi_full_mark();
```

Trigger



Triggers Return Value

Table "Scores"

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

- Return **NULL** skips the **triggering** operation only

```
CREATE OR REPLACE FUNCTION give_adi_full_mark() RETURNS TRIGGER AS $$
BEGIN
  IF (NEW.Name = 'Adi') THEN
    NEW.Mark := 100;
    insert into scores_log2 values ('Adi', 'insert', CURRENT_DATE);
    RETURN NULL;
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

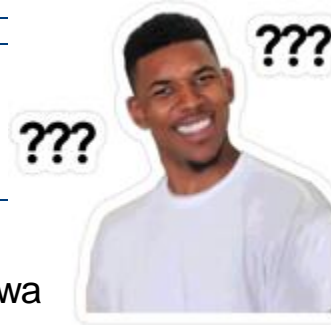
Trigger Function

Effect?

Adi: not inserted (scores),
but inserted in scores_log2.
Others: inserted (scores), not
inserted in scores_log2.

```
CREATE TRIGGER adi_should_get_full_mark
BEFORE INSERT ON Scores
FOR EACH ROW EXECUTE FUNCTION give_adi_full_mark();
```

Trigger



Triggers Timing

- **INSTEAD OF Trigger**

- This kind of trigger can only be defined on **VIEWS**

- **First Off...**

- What is a **VIEW**?
 - A “*virtual*” table defined by a query to compute the view
 - Can be used in queries just like a regular table
- Why use a **VIEW**?
 - Hide data and/or complexity from users
 - Logical data independence
 - Real database applications use **tons** of VIEW

```
CREATE VIEW overdue AS
SELECT b.title, u.name, ...
FROM books b, users u, ...
WHERE ...

SELECT * FROM overdue;
```

Triggers Timing

- **INSTEAD OF Trigger**
 - This kind of trigger can only be defined on **VIEWS**
- **Secondly ...**
 - Why do we want to modify a **VIEW**?
 - User sees it as a table and they can modify table
 - Does it even make sense?
 - In most cases, we cannot modify **VIEW** directly
 - But we can modify the underlying table

Triggers Timing

- **INSTEAD OF Trigger**

- This kind of trigger can only be defined on **VIEWS**

- **Modifying a VIEW**

- Simple case

- `CREATE VIEW Students AS
SELECT Name FROM Scores;`

- `DELETE FROM Students WHERE Name = 'Alice';`
automatically translates to
`DELETE FROM Scores WHERE Name = 'Alice';`

Triggers Timing

- **INSTEAD OF Trigger**

- This kind of trigger can only be defined on **VIEWS**

- **Modifying a VIEW**

- Too many possible case

- ```
CREATE VIEW Top_Marks AS
SELECT MAX(Mark) AS Mark FROM Scores;
```

- ```
UPDATE Top_Marks SET Mark = 80;
```

Views that return aggregate functions are not automatically updatable

Triggers Timing

- **INSTEAD OF Trigger**
 - This kind of trigger can only be defined on **VIEWS**
- **OK, so what to do here?**
 - Let's say we want to modify all people with the maximum mark
 - We modify the underlying table instead
 - Use **INSTEAD OF** trigger

Triggers Timing

- The Trigger Function

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

Trigger Function

Return Value

- **NULL**
 - Ignore all operations on current row
- **Non-NULL**
 - Signals the database to proceed as normal

- The Trigger

```
CREATE TRIGGER modify_top_mark  
INSTEAD OF UPDATE ON Top_Marks  
FOR EACH ROW EXECUTE FUNCTION update_top_mark();
```

Trigger

Triggers Timing

- The Trigger Function

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

Trigger Function

Return Value

- **NULL**
 - Ignore all operations on current row
 - The update scores set ... is still completed!

- The Trigger

```
CREATE TRIGGER modify_top_mark  
INSTEAD OF UPDATE ON Top_Marks  
FOR EACH ROW EXECUTE FUNCTION update_top_mark();
```

Trigger

Triggers Granularity

FOR EACH ROW
FOR EACH STATEMENT

Triggers Granularity

- **Row-Level Trigger** *(FOR EACH ROW)*
 - Executes the trigger function for every tuple encountered
- **Statement-Level Trigger** *(FOR EACH STATEMENT)*
 - Executes the trigger function only once
 - Why would we want to do this?
 - If we already prevent an operation, then just once is enough
 - There are other contextual data available, but we will not discuss this

```
CREATE TRIGGER modify_top_mark  
INSTEAD OF UPDATE ON Top_Marks  
FOR EACH ROW EXECUTE FUNCTION update_top_mark();
```

Triggers Granularity

Table "Scores_Log"

<u>Name</u>	Date
...	...

- The Trigger Function

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

Trigger Function

- The Trigger

```
CREATE TRIGGER warn_delete  
BEFORE DELETE ON Scores_Log  
FOR EACH STATEMENT EXECUTE FUNCTION show_warning();
```

Trigger

Effect?

- **RAISE NOTICE**
 - Database will give you prompt whenever a deletion is attempted
- **RETURN NULL**
 - Unfortunately, still perform deletion

Triggers Granularity

Table "Scores_Log"

<u>Name</u>	Date
...	...

- The Trigger Function

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

Trigger Function

- Statement-Level Trigger *(FOR EACH STATEMENT)*

- Ignores the values returned by the trigger functions
- RETURN NULL would not make the database omit the subsequent operation
- What to do?
 - RAISE EXCEPTION, aborts current transaction.

Triggers Granularity

- **Granularity and Timing**

- **INSTEAD OF** is only allowed on row-level granularity
- **AFTER** or **BEFORE** are allowed on both row-level as well as statement-level granularity

Granularity

Timing	Row-Level	Statement-Level
AFTER	Tables	Tables and View
BEFORE	Tables	Tables and View
INSTEAD OF	Views	-

Triggers Condition

Event	Condition	Action
<i>Trigger</i>	<i>... we are here ...</i>	<i>Trigger Function</i>

Triggers Condition

Table "Scores"

<u>Name</u>	Mark
Adi	100

- The Trigger Function

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

Trigger Function

Observation

Trigger function only cares about the case when name is *Adi*

We can move this condition to the trigger definition

- The Trigger

```
CREATE TRIGGER adi_should_get_full_mark
BEFORE INSERT ON Scores
FOR EACH ROW WHEN (NEW.Name = 'Adi')
EXECUTE FUNCTION give_adi_full_mark();
```

Trigger

Triggers Condition

- **What Can We Use?**

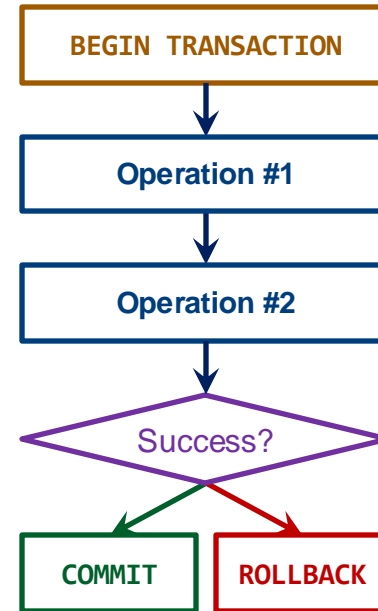
- In general, the condition in **WHEN()** could be more complicated
- Subject to the following requirements:
 - **NO SELECT** in **WHEN()**
 - **NO OLD** in **WHEN()** for **INSERT**
 - **NO NEW** in **WHEN()** for **DELETE**
 - **NO WHEN()** for **INSTEAD OF**

```
CREATE TRIGGER adi_should_get_full_mark
BEFORE INSERT ON Scores
FOR EACH ROW WHEN (NEW.Name = 'Adi')
EXECUTE FUNCTION give_adi_full_mark();
```

Triggers Condition

- **Deferred Trigger**

- Triggers happen at the end of either statement or at the end of transaction
- **Operation** consisting of multiple statements may leave the database in an intermediate *inconsistent* state
- In such cases, we want the trigger to check consistency constraint only at the end of a transaction
- This is called a *deferred* trigger



Triggers Condition

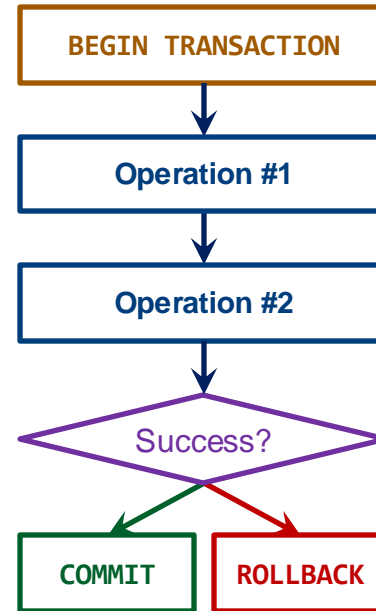
- **Deferred Trigger**

- Example:
 - Customer may have multiple account
 - Total balance for all accounts must be at least 150

Table "Account"

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

- Task: Transfer money from one account to another



Triggers Condition

- **Deferred Trigger**

- Naïve approach:

1. Deduct amount from account 1
2. Add amount to account 2

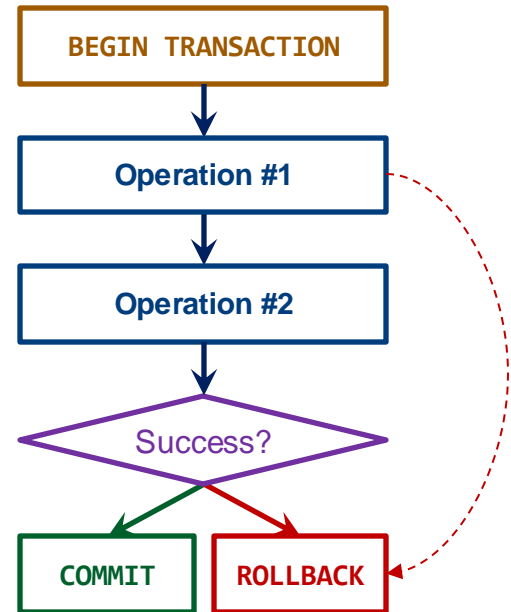
Table "Account"

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

- Solution:

1. Put the two update into one transaction
2. Defer trigger to check only at the end of transaction

May violate constraint!



Triggers Condition

Table "Account"

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

- **Deferred Trigger**

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

- **CONSTRAINT** and **DEFERRABLE** *together* indicate that the trigger can be deferred
- **INITIALLY DEFERRED** indicates that by default, the trigger is deferred
 - In other words, only check at the end of transaction
 - Other option is **INITIALLY IMMEDIATE**
 - *i.e.*, the trigger is not deferred by default
- Deferred triggers only work with **AFTER** and **FOR EACH ROW**

Triggers Condition

- Deferred Trigger only work with AFTER

Table "Account"

<u>AID</u>	Name	Balance
1	Alice	100
2	Alice	100
3	Bob	200

```
CREATE OR REPLACE FUNCTION check_balance()
RETURNS TRIGGER AS $$
DECLARE
    total_balance INT;
BEGIN
    IF TG_OP = 'DELETE' THEN
        SELECT SUM(balance) INTO total_balance FROM accounts WHERE name = OLD.name;
    ELSE
        SELECT SUM(balance) INTO total_balance FROM accounts WHERE name = NEW.name;
    END IF;
    IF total_balance < 150 THEN
        RAISE EXCEPTION 'Total balance must be >= 150';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Triggers Condition

Table "Account"

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

- **Deferred Trigger**

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

- Now we can do the following, `check_balance()` triggered for both rows are deferred to the end of the transaction

```
BEGIN TRANSACTION;  
  
UPDATE Account SET Balance = Balance - Amount WHERE AID = Account1;  
UPDATE Account SET Balance = Balance + Amount WHERE AID = Account2;  
  
COMMIT;
```

Triggers Condition

Table "Account"

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

- **Deferred Trigger**

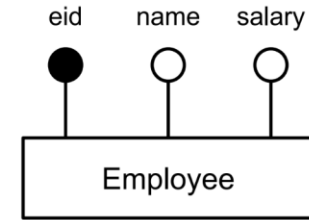
```
CREATE CONSTRAINT TRIGGER balance_check  
AFTER INSERT OR UPDATE OR DELETE ON Account  
DEFERRABLE INITIALLY IMMEDIATE  
FOR EACH ROW EXECUTE FUNCTION check_balance();
```

- What if the trigger is INITIALLY IMMEDIATE?

- Change it on the fly

```
BEGIN TRANSACTION;  
SET CONSTRAINTS balance_check DEFERRED;  
UPDATE Account SET Balance = Balance - Amount WHERE AID = Account1;  
UPDATE Account SET Balance = Balance + Amount WHERE AID = Account2;  
  
COMMIT;
```

Additional Example

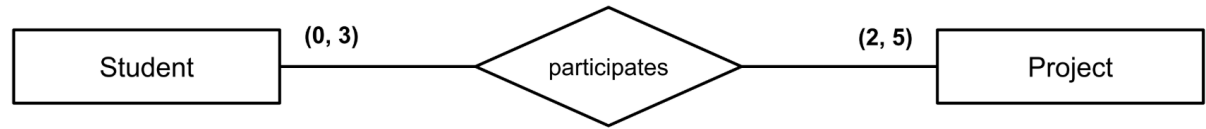


- **Employee salary can only be increased.**

```
CREATE TRIGGER on_employee_updated
BEFORE UPDATE ON employees
FOR EACH ROW EXECUTE FUNCTION
    check_valid_salary();
```

```
CREATE OR REPLACE FUNCTION check_valid_salary()
RETURNS TRIGGER AS $$
BEGIN
    IF OLD.salary > NEW.salary THEN
        RAISE EXCEPTION 'Salary may not decrease!';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Additional Example



- Enforce the cardinality constraints.

```
CREATE CONSTRAINT TRIGGER on_new_allocation  
AFTER INSERT ON participates DEFERRABLE INITIALLY DEFERRED  
FOR EACH ROW EXECUTE FUNCTION on_insert_participates();
```

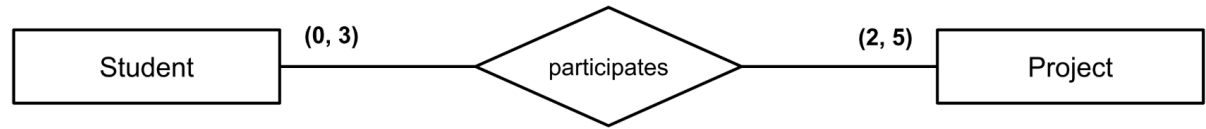
Need to put 2 inserts into one transaction, deferred trigger must be after trigger!

```
CREATE OR REPLACE FUNCTION on_insert_participates()  
RETURNS TRIGGER AS $$  
DECLARE team_size INT; num_projects INT  
BEGIN  
    SELECT COUNT(*) INTO team_size FROM participates  
    WHERE pid = NEW.pid;  
    SELECT COUNT(*) INTO num_projects FROM participates  
    WHERE sid = NEW.sid;  
    IF team_size < 2 OR team_size > 5 OR num_projects > 3 THEN  
        RAISE EXCEPTION 'Invalid cardinality';  
    END IF;  
    RETURN NEW;  
END; $$ LANGUAGE plpgsql;
```

Need BEFORE DELETE, BEFORE UPDATE triggers!

Since this is after trigger, check cardinality after, raise exception if invalid!

Additional Example



- **Enforce the cardinality constraints.**

```
CREATE TRIGGER on_remove_allocation
BEFORE DELETE ON participates
FOR EACH ROW EXECUTE FUNCTION on_delete_participates();
```

```
CREATE OR REPLACE FUNCTION on_delete_participates()
RETURNS TRIGGER AS $$
    DECLARE team_size INT;
BEGIN
    SELECT COUNT(*) INTO team_size FROM participates
    WHERE pid = OLD.pid___;
    IF team_size <= 2 THEN
        RETURN NULL;
    END IF;
    RETURN OLD;
END; $$ LANGUAGE plpgsql;
```

Additional Example



```
CREATE TRIGGER on_change_allocation BEFORE UPDATE ON participates
FOR EACH ROW EXECUTE FUNCTION on_update_participates();

CREATE OR REPLACE FUNCTION on_update_participates() RETURNS TRIGGER AS $$
DECLARE old_team_size INT; new_team_size INT; num_projects INT;
BEGIN
    IF OLD.pid <> NEW.pid THEN
        SELECT COUNT(*) INTO old_team_size FROM participates WHERE pid = OLD.pid;
        IF old_team_size <= 2 THEN RETURN NULL; ENDIF;
        SELECT COUNT(*) INTO new_team_size FROM participates WHERE pid = NEW.pid;
        IF new_team_size >= 5 THEN RETURN NULL; ENDIF;
    END IF;
    IF OLD.sid <> NEW.sid THEN
        SELECT COUNT(*) INTO num_projects FROM participates WHERE sid = NEW.sid;
        IF num_projects >= 3 THEN RETURN NULL; ENDIF;
    END IF;
    RETURN NEW;
END; $$ LANGUAGE plpgsql;
```

Final Note on Triggers

- **Multiple Triggers**

- There can be multiple triggers defined for the same event on the same table
 - There need to be an *order of 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 BEFORE row-level trigger returns NULL, then subsequent triggers on the same row are omitted

- **Universality of Triggers?**

- Our discussions are based on PostgreSQL syntax and implementation only

QUESTION?

Instructor: Jiang Kan