

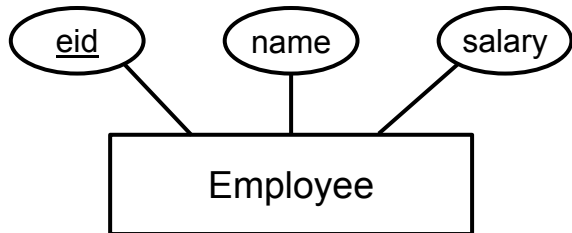
CS2102: Database Systems

Lecture 9 — Triggers

Overview

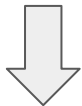
- **Triggers — Overview**
 - **Motivation**
 - Basic Example
 - Basic Concepts
- **Triggers — Options**
 - Events
 - Timing
 - Granularity
- **Triggers — Refinements**
 - Conditions
 - Deferrable Triggers
- **Summary**

Constraints Regarding Changes of Data



```
CREATE TABLE employees (  
  eid INT PRIMARY KEY,  
  name TEXT NOT NULL,  
  salary DECIMAL(12,2) NOT NULL DEFAULT 0,  
  CONSTRAINT check_pos_salary CHECK (salary >= 0)  
);
```

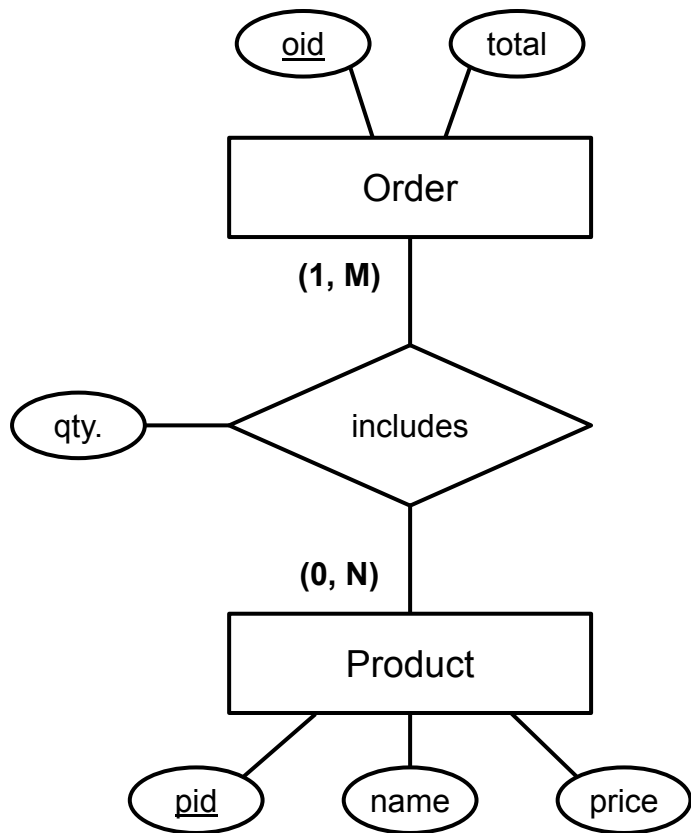
Additional constraint: The salary of an employee is only allowed to increase.



Question: How can we check and ensure integrity transparent to the user?

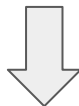
- The user/app should not be forced to call a stored procedure or function

Recall: ER Model — Stored Attributes but with Derived Values



```
CREATE TABLE orders (  
  oid INT PRIMARY KEY,  
  total DECIMAL(12,2) NOT NULL DEFAULT 0  
);
```

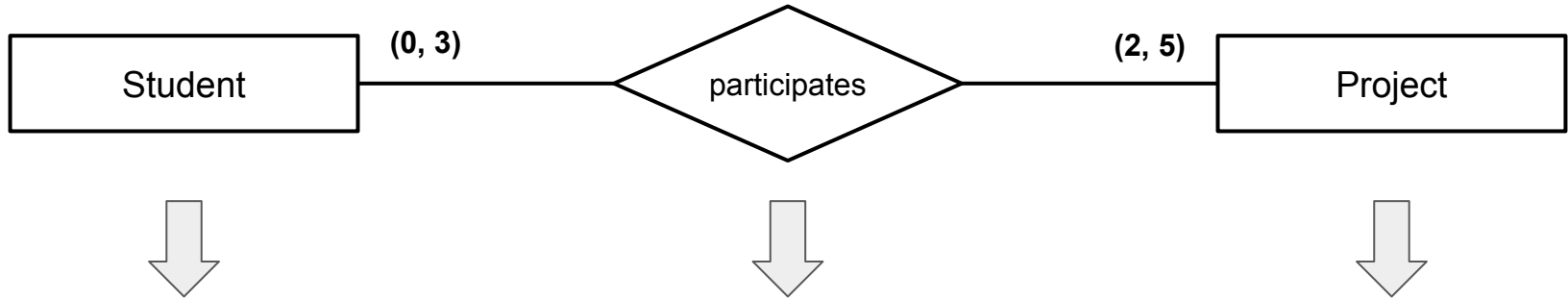
Integrity Constraint: `orders.total` must reflect the total price of the order!



Question: How can we check and ensure integrity transparent to the user?

- The user/app should not worry about that
- The user/app should not be forced to call a stored procedure or function

Recall: ER Model — Many-to-Many with Cardinality Constraints



```
CREATE TABLE student (  
  id INT PRIMARY KEY,  
  name TEXT NOT NULL,  
  email TEXT NOT NULL,  
  dob DATE,  
  ...  
);
```

```
CREATE TABLE participates (  
  sid INT FOREIGN KEY REFERENCES students (id),  
  pid INT FOREIGN KEY REFERENCES projects (id),  
  PRIMARY KEY (sid, pid)  
);
```

```
CREATE TABLE projects (  
  id INT PRIMARY KEY,  
  title TEXT NOT NULL,  
  budget INT NOT NULL,  
  dob DATE,  
  ...  
);
```

Problem: Relational Schema can **not** capture cardinality constraints!

Overview

- **Triggers — Overview**
 - Motivation
 - **Basic Example**
 - Basic Concepts
- **Triggers — Options**
 - Events
 - Timing
 - Granularity
- **Triggers — Refinements**
 - Conditions
 - Deferrable Triggers
- **Summary**

Running Example Database

- Toy database: Just 1 table
 - Over-simplified table `students`

```
CREATE TABLE students (  
  id SERIAL PRIMARY KEY,  
  name TEXT NOT NULL,  
  points INTEGER DEFAULT 0,  
  graduated BOOLEAN DEFAULT FALSE  
);
```

id	name	points	graduated
1	Bob	94	TRUE
2	Eve	82	FALSE
3	Sam	65	FALSE
4	Liz	86	TRUE
5	Tom	90	TRUE
6	Sue	94	FALSE
7	Zac	75	FALSE
8	Ida	84	TRUE
9	Leo	91	FALSE
10	Pam	70	FALSE

Motivation — Simple Use Case



- Application requirement

- Every time a new students gets entered, we need log this event
- Logging is done using a separate table: `basic_logs(student:INT, created_at:TIMESTAMP)`

Table `students`

id	name	points	graduated
1	Bob	94	TRUE
2	Eve	82	FALSE
3	Sam	65	FALSE
4	Liz	86	TRUE
5	Tom	90	TRUE
...

Desired design goals

- Insert into `basic_logs` automatically
- Do not force user to write too much SQL
- Log not matter how students are added



Table `basic_logs`

id	created_at
1	2023-08-24 09:02:50
2	2023-08-24 13:25:41
3	2023-08-24 15:40:23
4	2023-08-24 20:11:08
5	2023-08-25 08:55:10
...	...



Overview

- **Triggers — Overview**
 - Motivation
 - Basic Example
 - **Basic Concepts**
- **Triggers — Options**
 - Events
 - Timing
 - Granularity
- **Triggers — Refinements**
 - Conditions
 - Deferrable Triggers
- **Summary**

Triggers — Basic Concepts

- Trigger = event-condition-action (**ECA**) rule

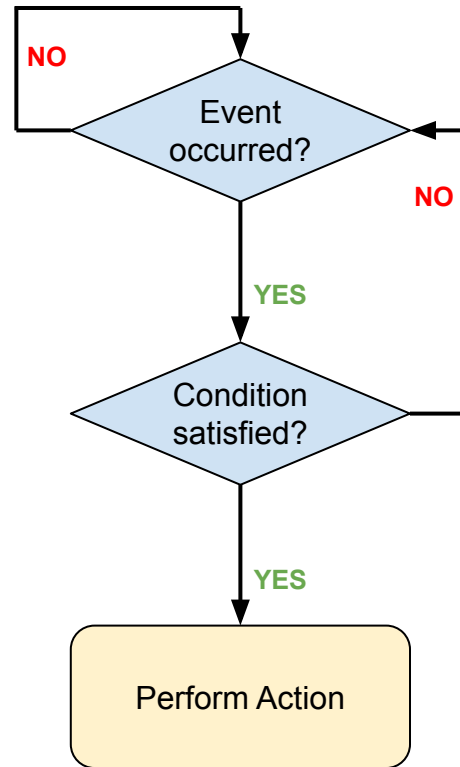
- When an **event** occurs...
- ...test **condition** and...
- ...if satisfied, perform **action**.

- Example (for our simple use case)

Event: New tuple inserted into `students`

Condition: (nothing)

Action: Insert into `basic_logs`



Triggers — Basic Concepts

- ECA rule split into 2 parts

Event: New tuple inserted into **students**

Condition: (nothing)

Action: Insert into **basic_logs**

} **Trigger**

} **Trigger Function**

Trigger

```
CREATE TRIGGER on_student_entered
AFTER INSERT ON students
FOR EACH ROW
EXECUTE FUNCTION log_student_basic();
```

Trigger Function

```
CREATE FUNCTION log_student_basic()
RETURNS TRIGGER AS
$$
BEGIN
    INSERT INTO basic_logs (student) VALUES (NEW.id);
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```



Trigger Function vs "Normal" Function

- Trigger function — requirements
 - Must take no (ordinary) arguments
 - Must have a return type **TRIGGER**
 - Must be defined before trigger itself
- Trigger function — "input"
 - Special internal data structure received from the trigger
 - Useful data within the trigger function
(which data is available depends on trigger definition!)



TG_NAME	Name of the trigger that fired
TG_OP	Operation that fired the trigger (INSERT, UPDATE, DELETE)
TG_WHEN	Time when the trigger was fired (BEFORE, AFTER, or INSTEAD OF)
NEW	Record holding the new row for INSERT/UPDATE operations
OLD	Record holding the old row for UPDATE/DELETE operations
...	...
TG_ARGV[]	Array of arguments from the CREATE TRIGGER statement.

Extended Use Case



- Additional requirements

- Log all events that might modify students' points (INSERT, UPDATE, DELETE)

→ `advanced_logs(student: INT, operation: TEXT, points_old: INT, points_new: INT, created_at: TIMESTAMP)`

```
CREATE OR REPLACE FUNCTION log_student_advanced()
RETURNS TRIGGER AS
$$
BEGIN
    IF TG_OP = 'INSERT' THEN
        INSERT INTO points_log_advanced VALUES (NEW.id, TG_OP, NULL, NEW.points, DEFAULT);
        RETURN NEW;
    ELSIF (TG_OP = 'DELETE') THEN
        INSERT INTO points_log_advanced VALUES (OLD.id, TG_OP, OLD.points, NULL, DEFAULT);
        RETURN OLD;
    ELSIF (TG_OP = 'UPDATE') THEN
        INSERT INTO points_log_advanced VALUES (OLD.id, TG_OP, OLD.points, NEW.points, DEFAULT);
        RETURN NEW;
    END IF;
END;
$$ LANGUAGE plpgsql;
```

New.id
works too

Extended Use Case

- Define trigger

- Can listen to multiple event types

- Example execution

- 4 SQL statements of different types affecting table **students**
- All statements reflected in log table

```
CREATE TRIGGER on_student_modified_advanced  
AFTER INSERT OR DELETE OR UPDATE ON students  
FOR EACH ROW  
EXECUTE FUNCTION log_student_advanced();
```



```
INSERT INTO students (name, points) VALUES ('Adi', 80);  
  
UPDATE students SET points = 92 WHERE id = 1;  
  
UPDATE students SET points = 75 WHERE id = 7;  
  
DELETE FROM students WHERE id = 4;
```



id	operation	points_old	points_new	created_at
11	INSERT	<i>null</i>	80	2023-09-25 09:02:50
1	UPDATE	94	92	2023-09-27 13:25:41
7	UPDATE	75	75	2023-09-28 13:25:41
4	DELETE	86	<i>null</i>	2023-09-28 15:40:23

Triggers — Trigger Options

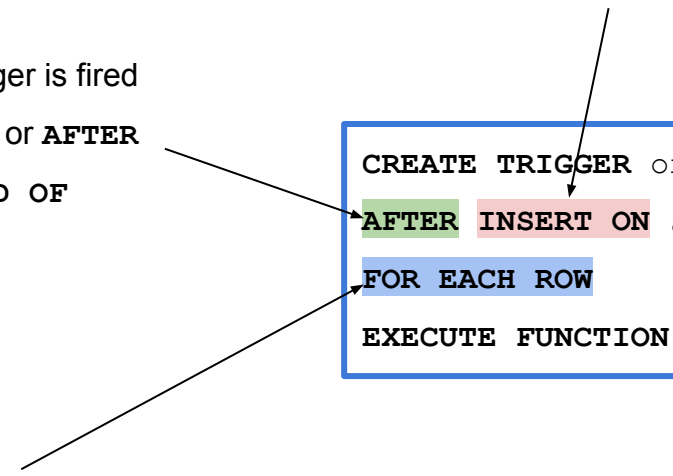
Timing

- Specifies *when* trigger is fired
- For tables: **BEFORE** or **AFTER**
- For views: **INSTEAD OF**

Event

- Occurrence or situation that fires the trigger
- Alternatives: **INSERT**, **UPDATE**, **DELETE**

```
CREATE TRIGGER on_student_entered  
AFTER INSERT ON students  
FOR EACH ROW  
EXECUTE FUNCTION log_student_basic();
```



Granularity

- Specifies if triggered for each affect row or only once
- Alternatives: **FOR EACH ROW** or **FOR EACH STATEMENT**

Overview

- Triggers — Overview
 - Motivation
 - Basic Example
 - Basic Concepts
- Triggers — Options
 - **Events**
 - Timing
 - Granularity
- Triggers — Refinements
 - Conditions
 - Deferrable Triggers
- Summary

Triggers — Events

- Trigger event — the **Why?**

- Operation that causes the trigger to fire
- 3 event types reflect the 3 basic DB operations

INSERT ON *table*

DELETE ON *table*

UPDATE [OF column] ON *table*

→ TG_OP =

'INSERT'

'DELETE'

'UPDATE'

- Multiple events can fire the same trigger

```
CREATE TRIGGER on_student_modified_advanced
AFTER INSERT OR DELETE OR UPDATE ON students
...
```

```
CREATE TRIGGER on_student_entered
AFTER INSERT ON students
FOR EACH ROW
EXECUTE FUNCTION log_student_basic();
```

Triggers — Events

- Trigger event specifies access to **transition variables**

- **NEW**: modified row *after* the triggering event
(can only exist for **INSERT** and **UPDATE** operations)
- **OLD**: modified row *before* the triggering event
(can only exist for **DELETE** and **UPDATE** operations)

	NEW	OLD
INSERT	✓	✗
UPDATE	✓	✓
DELETE	✗	✓

cannot do an insert as OLD.id will be used as pri key, cannot be null

```
CREATE TRIGGER on_student_entered
AFTER INSERT ON students
FOR EACH ROW
EXECUTE FUNCTION log_student_basic();
```

```
CREATE FUNCTION log_student_basic()
RETURNS TRIGGER AS
$$
BEGIN
    INSERT INTO basic_logs (student) VALUES (OLD.id);
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

This will cause problems
since OLD.id is NULL!

Overview

- Triggers — Overview
 - Motivation
 - Basic Example
 - Basic Concepts
- Triggers — Options
 - Events
 - **Timing**
 - Granularity
- Triggers — Refinements
 - Conditions
 - Deferrable Triggers
- Summary

Triggers — Timing

- Trigger timing — the **When?**

- The moment the trigger is fired
- 3 possible timings

AFTER

Triggers fires after the operation has completed
(this includes that any relevant constraints have been checked)

more commonly used

BEFORE

Trigger fires before the operation is attempted

INSTEAD OF

Trigger fires if an operation on a view is attempted

- Importance of timing

- **BEFORE** and **INSTEAD OF** triggers can skip or modify the operation
- For **BEFORE** and **INSTEAD OF** triggers the return value of the trigger function matters

```
CREATE TRIGGER on_student_entered  
AFTER INSERT ON students  
FOR EACH ROW  
EXECUTE FUNCTION log_student_basic();
```

Triggers — BEFORE VS AFTER

- Effects of return values

- **BEFORE** triggers can intercept and modify / change the operation
- **AFTER** triggers cannot affect operations that fired them

return NULL = the operation
won't be carried out

	RETURN value	
	NULL tuple	non-NULL tuple t
BEFORE INSERT	No tuple inserted	Tuple t inserted
BEFORE UPDATE	No tuple updated	Tuple t updated
BEFORE DELETE	No tuple deleted	Tuple t deleted
AFTER INSERT	No effects! "After " Trigger doesn't care about the return values	
AFTER UPDATE		
AFTER DELETE		

```
CREATE TRIGGER on_student_entered
AFTER INSERT ON students
FOR EACH ROW
EXECUTE FUNCTION log_student_basic();
```

```
CREATE FUNCTION log_student_basic()
RETURNS TRIGGER AS
$$
BEGIN
    INSERT INTO basic_logs (student) VALUES (NEW.id);
    RETURN NULL; -- or RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Quick Quiz




Triggers — Intercepting Operations with **BEFORE** Triggers

- Example use case

- If we insert or update a student named "Adi", we give him full points
- Use **BEFORE** trigger to intercept initial **INSERT** or **UPDATE** operation

```
CREATE TRIGGER on_student_entered
BEFORE INSERT OR UPDATE ON students
FOR EACH ROW
EXECUTE FUNCTION help_adi_cheat();
```



```
CREATE FUNCTION help_adi_cheat()
RETURNS TRIGGER AS
$$
BEGIN
    IF NEW.name = 'Adi' THEN
        NEW.points := 100;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```


Triggers — Intercepting Operations with BEFORE Triggers

- Alternative implementations of trigger function
 - Any valid non-NULL tuple will be inserted into students
 - We can even set and return OLD transition variable despite INSERT operation

```
CREATE FUNCTION help_adi_cheat()  
RETURNS TRIGGER AS  
$$  
BEGIN  
    IF NEW.name = 'Adi' THEN  
        OLD := NEW;    OLD used to be NULL, reassigning it to NEW  
        OLD.points := 100  
    RETURN OLD;  
END;  
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION help_adi_cheat()  
RETURNS TRIGGER AS  
$$  
DECLARE  
    s RECORD;  
BEGIN  
    IF NEW.name = 'Adi' THEN  
        s := NEW;  
        s.points := 100  
    RETURN s;  
END;  
$$ LANGUAGE plpgsql;
```

Triggers — Timing

- **INSTEAD OF trigger**

- Can only be defined on **views**
- Why do we want / need those?

- **Views — quick recap**

- Virtual table (permanently named query)
- Result of a query is not permanently stored! (query is executed each time the view is used)
- Looks and can be used like any other table (well, kind of...)
- Hide data or complexity from users (recall: logical data independence)
- Heavily used in real-world database applications

```
CREATE VIEW <name> AS
    SELECT ...
    FROM ...
    ...
;
```

Given a table with name, job, salary, age
want everyone to be only able to see name, job
but only HR can see salary age

Create a view of name + job,
everyone can access it. but only HR
can access whole table

Triggers — Working with Views

- Views vs "normal" tables

- No restriction when used in SQL queries (**SELECT** statements)
- But what about **INSERT**, **UPDATE**, **DELETE** statements?

→ Updatable View — requirements

- Only **one entry in FROM** clause (table or updatable view)
- **No WITH, DISTINCT, GROUP BY, HAVING, LIMIT, or OFFSET**
- **No UNION, INTERSECT or EXCEPT**
- **No aggregate functions**
- etc. (incl. no constraint violations)

Direct modification of view
not possible in most cases.

Triggers — Updatable Views

```
CREATE VIEW students_view AS
  SELECT name, points
  FROM students
;
```

Example of an Updatable View

able to easily convert to
a query that changes
the main table

DELETE FROM students_view **WHERE** name = 'Bob';



DELETE FROM students **WHERE** name = 'Bob';



UPDATE students_view **SET** points = points + 3;



UPDATE students **SET** points = points + 3;



INSERT INTO students_view **VALUES** ('Adi', 80);



INSERT INTO students **VALUES** (DEFAULT, 'Adi', 80, DEFAULT);



works since we have default values → no constraint violations

Triggers — Non-Updatable Views

```
CREATE VIEW point_groups AS
  SELECT points, COUNT(*) AS num_students
  FROM students
  GROUP BY points
;
```

Example of a **Non-Updatable View**



```
SELECT * FROM point_groups;
```

points	num_students
65	1
70	1
82	3
86	1
90	1
91	1
94	2

"Give the best students a bonus!"

```
UPDATE point_groups
SET points = points + 3
WHERE points = 94;
```

→ **This will throw an error!** (GROUP BY in view)

Triggers — INSTEAD OF

```
CREATE TRIGGER on_point_groups_updated
INSTEAD OF UPDATE ON point_groups
FOR EACH ROW
EXECUTE FUNCTION update_student_points();
```

```
CREATE FUNCTION update_student_points()
RETURNS TRIGGER AS
$$
BEGIN
    UPDATE students SET points = NEW.points
    WHERE points = OLD.points;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

table students

id	name	points	graduated
1	Bob	97	TRUE
2	Eve	82	FALSE
3	Sam	65	FALSE
4	Liz	86	TRUE
5	Tom	90	TRUE
6	Sue	97	FALSE
7	Zac	75	FALSE
8	Ida	84	TRUE
9	Leo	91	FALSE
10	Pam	70	FALSE

"Give the best students a bonus!"

```
UPDATE point_groups
SET points = points + 3
WHERE points = 94;
```

→ This works now

Quick Quiz



Quick Quiz — Solution



Overview

- Triggers — Overview
 - Motivation
 - Basic Example
 - Basic Concepts
- Triggers — Options
 - Events
 - Timing
 - **Granularity**
- Triggers — Refinements
 - Conditions
 - Deferrable Triggers
- Summary

Triggers — Granularity

- Row-level triggers

- Trigger function is executed **for each affected row**
- Keyword: **FOR EACH ROW**

- Statement-level triggers

- Trigger function is executed **once for each transaction** (no matter how many rows are affected)
- Keyword: **FOR EACH STATEMENT**
- **Ignored return value of trigger function**
(Enforcing a rollback requires **RAISE EXCEPTION!**)

Example for a statement-level trigger

- Prohibit the deletion of rows from the logs
- Show warning to user only once no matter how many rows the user attempted to delete

```
CREATE TRIGGER on_delete_from_log
BEFORE DELETE ON advanced_log
FOR EACH STATEMENT
EXECUTE FUNCTION show_warning();
```

```
CREATE FUNCTION show_warning()
RETURNS TRIGGER AS
$$
BEGIN
    RAISE EXCEPTION 'Do not DELETE the logs!!!';
    RETURN NULL; return NULL for a BEFORE trigger means nothing will happen
END;
$$ LANGUAGE plpgsql;
```

Triggers — Granularity

- Granularity and timing
 - **AFTER** and **BEFORE** allowed for *both* row-level and statement-level triggers
 - **INSTEAD OF** *only* allowed for row-level triggers
- Possible combinations

Timing	Row-Level	Statement-Level
AFTER	Tables	Tables & Views
BEFORE	Tables	Tables & Views
INSTEAD OF	Views	—

Overview

- Triggers — Overview
 - Motivation
 - Basic Example
 - Basic Concepts
- Triggers — Options
 - Events
 - Timing
 - Granularity
- Triggers — Refinements
 - **Conditions**
 - Deferrable Triggers
- Summary

Triggers — Conditions

- Throwback to previous example
 - Log any update of points in the students table
 - Example on the right shows initial solution
- Now: additional refinement
 - Let's not log updates that do not really change points

```
CREATE TRIGGER on_student_modified_advanced
AFTER INSERT OR DELETE OR UPDATE ON students
FOR EACH ROW
EXECUTE FUNCTION log_student_advanced();
```



```
INSERT INTO students (name, points) VALUES ('Adi', 80);

UPDATE students SET points = 92 WHERE id = 1;

UPDATE students SET points = 75 WHERE id = 7;

DELETE FROM students WHERE id = 4;
```



id	operation	points_old	points_new	created_at
11	INSERT	<i>null</i>	80	2023-09-25 09:02:50
1	UPDATE	94	92	2023-09-27 13:25:41
7	UPDATE	75	75	2023-09-28 13:25:41
4	DELETE	86	<i>null</i>	2023-09-28 15:40:23

No need to log this row,
just wastes disk space?



Triggers — Conditions



- Approach 1: Modify trigger function

performance wise , not that great as we are already executing the function.

```
CREATE OR REPLACE FUNCTION log_student_advanced()  
RETURNS TRIGGER AS  
$$  
BEGIN  
    IF TG_OP = 'INSERT' THEN  
        INSERT INTO points_log_advanced VALUES (NEW.id, TG_OP, NULL, NEW.points, DEFAULT);  
        RETURN NEW;  
    ELSIF (TG_OP = 'DELETE') THEN  
        INSERT INTO points_log_advanced VALUES (OLD.id, TG_OP, OLD.points, NULL, DEFAULT);  
        RETURN OLD;  
    ELSIF (TG_OP = 'UPDATE') THEN  
        IF NEW.points <> OLD.points THEN  
            INSERT INTO points_log_advanced VALUES (OLD.id, TG_OP, OLD.points, NEW.points, DEFAULT);  
        END IF;  
        RETURN NEW;  
    END IF;  
END;  
$$ LANGUAGE plpgsql;
```

Triggers — Conditions

- Approach 2: Modify trigger
 - Move condition from trigger function to trigger
 - Only execute trigger function if condition is true

```
CREATE TRIGGER on_student_updated_advanced
AFTER UPDATE ON students
FOR EACH ROW WHEN (NEW.points <> OLD.points)
EXECUTE FUNCTION log_student_advanced();
```

we can't use include insert and delete in this trigger function anymore as insert and delete do not have access to OLD and NEW respectively

Fires for every **UPDATE** but executes trigger function only if the points are indeed different.



separate out into 2 different trigger functions

```
CREATE TRIGGER on_student_modified_advanced
AFTER INSERT OR DELETE ON students
FOR EACH ROW
EXECUTE FUNCTION log_student_advanced();
```

Triggers — Conditions

- What conditions can we formulate
 - In general, any Boolean expression
 - In principle, can be arbitrarily complex
- Restrictions
 - **NO** SELECT in WHEN ()
 - **NO** OLD in WHEN () for INSERT
 - **NO** NEW in WHEN () for DELETE
 - **NO** WHEN () for INSTEAD OF

Overview

- Triggers — Overview
 - Motivation
 - Basic Example
 - Basic Concepts
- Triggers — Options
 - Events
 - Timing
 - Granularity
- **Triggers — Refinements**
 - Conditions
 - **Deferrable Triggers**
- Summary

Triggers — Deferrable Triggers

- Triggers — default behavior

- Triggers run immediately for **every** statement that fire them
- Problem: operations of **multiple statements** yielding intermediate inconsistent states

- Example

- Table with customer accounts
(Customers can have multiple accounts)
- Constraint: all balances of the same customer's account must be at least 150
(we already know how to write a trigger for that)

id	name	balance
...
10	Bob	100
11	Bob	80
...

Triggers — Deferrable Triggers

```
CREATE TRIGGER on_customers_modified
BEFORE INSERT OR DELETE OR UPDATE ON customers
FOR EACH ROW
EXECUTE FUNCTION check_balance();
```

```
CREATE OR REPLACE FUNCTION check_balance()
RETURNS TRIGGER AS
$$
...
    SELECT SUM(balance) INTO total_balance
    FROM customers WHERE id = NEW.id;
    IF total_balance < 150 THEN
        RETURN NULL;
...
$$ LANGUAGE plpgsql;
```

Question: How to transfer 50 dollar between Bob's accounts? — naive approach:

```
BEGIN TRANSACTION;
UPDATE customers SET balance = balance - 50 WHERE id = 10;
UPDATE customers SET balance = balance + 50 WHERE id = 11;
END TRANSACTION;
```

← **Trigger will fire and complain here!**

Triggers — Deferrable Triggers

- **Deferred triggers — behavior**

- Run trigger only at the end of transactions
- Ignore potential inconsistent states within transaction

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

```
BEGIN TRANSACTION;  
UPDATE customers SET balance = balance - 50 WHERE id = 10;  
UPDATE customers SET balance = balance + 50 WHERE id = 11;  
END TRANSACTION;
```

This works now!

Triggers — Deferrable Triggers

- **Deferred triggers** — requirements

- Only work for **AFTER** and **FOR EACH ROW** triggers^{check again!}
- Both **CONSTRAINT** and **DEFERRABLE** must be specified
- Two different default behaviors

INITIALLY DEFERRED: triggered is deferred by default

INITIALLY IMMEDIATE: triggered is not deferred by default (but can be deferred on demand)

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

Triggers — Deferrable Triggers

- Deferred triggers — deferred on demand

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

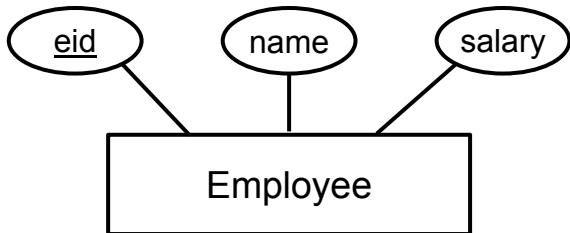
```
BEGIN TRANSACTION;  
SET CONSTRAINT on_customer_modified DEFERRED;  
UPDATE customers SET balance = balance - 50 WHERE id = 10;  
UPDATE customers SET balance = balance + 50 WHERE id = 11;  
END TRANSACTION;
```

Without that line, the
transaction would fail!

Overview

- Triggers — Overview
 - Motivation
 - Basic Example
 - Basic Concepts
- Triggers — Options
 - Events
 - Timing
 - Granularity
- Triggers — Refinements
 - Conditions
 - Deferrable Triggers
- Summary

Constraints Regarding Changes of Data



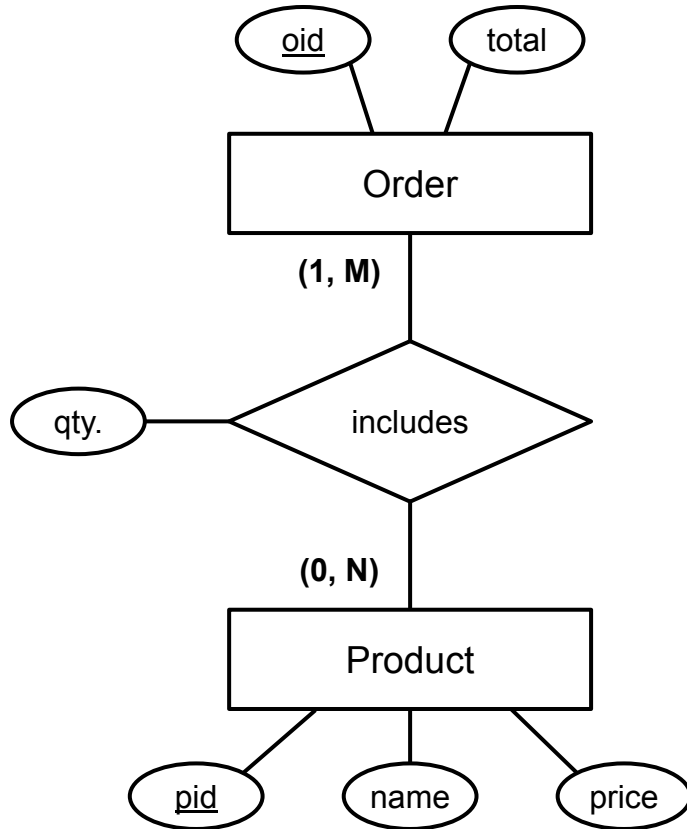
```
CREATE TABLE employees (  
  eid INT PRIMARY KEY,  
  name TEXT NOT NULL,  
  salary DECIMAL(12,2) NOT NULL DEFAULT 0,  
  CONSTRAINT check_pos_salary CHECK (salary >= 0)  
);
```

```
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;
```


Recall: ER Model — Stored Attributes but with Derived Values

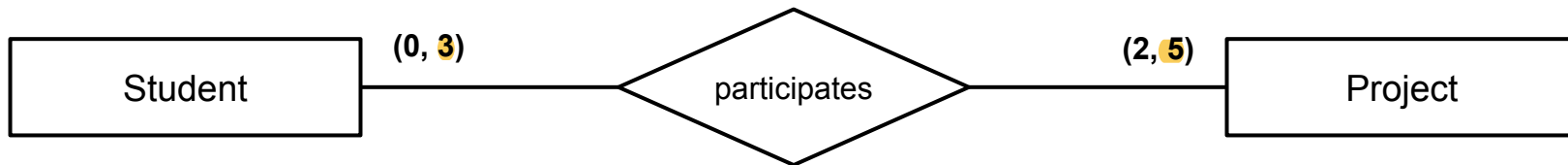
recalculate derived value everytime something changed



```
CREATE TRIGGER on_order_change
AFTER INSERT OR DELETE OR UPDATE ON includes
FOR EACH STATEMENT
EXECUTE FUNCTION calculate_total();
```

```
CREATE FUNCTION calculate_total()
RETURNS TRIGGER AS
$$
BEGIN
    UPDATE orders
    SET total = (SELECT SUM(p.price*i.qty)
                  FROM products p, includes i
                  WHERE p.pid = i.pid
                  AND i.oid = NEW.oid)
    WHERE o.oid = NEW.oid;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Recall: ER Model — Many-to-Many with Cardinality Constraints

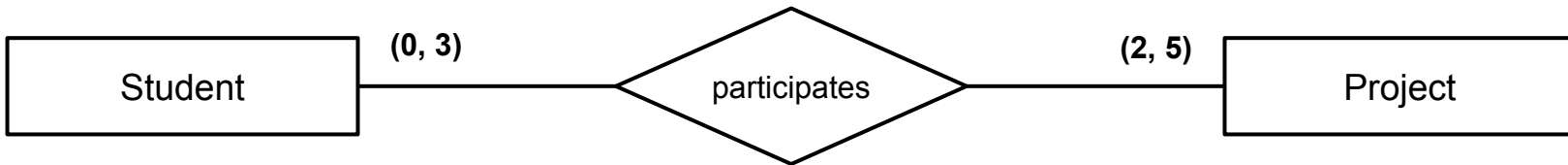


```
CREATE TRIGGER on_new_allocation
BEFORE INSERT ON participates
FOR EACH ROW
EXECUTE FUNCTION on_insert_participates();
```

Comment: BEFORE UPDATE and
BEFORE DELETE triggers needed

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

Recall: ER Model — Many-to-Many with Cardinality Constraints



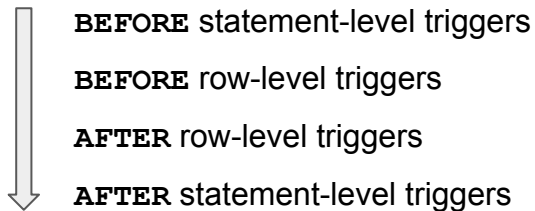
Things to consider

- **BEFORE UPDATE** trigger requires to set of checks
(e.g., moving a student to different team affects both teams' sizes!)
- What if we want to remove a student?
(this might yield an under-staffed project)
- How can we add a new project?
(just an INSERT into "Project" would violate participation constraint)

Triggers — Final Notes

- Multiple triggers for the same event on the same table → What to do?

- Basic order of activation:



- With each category, triggers are fired in **alphabetic order**

- If **BEFORE row-level trigger returns NULL**, subsequent triggers on the same row are omitted


- Universality of triggers

- Focus on PostgreSQL; syntax and exact behavior might vary between DBMS

Triggers — Final Notes

- Triggers give you the freedom to do **odd** things
 - Example: delete any student we just added
 - Question: Is there requirement where this would be meaningful?
 - Even if, a **BEFORE** trigger with **RETURN NULL** more suitable

```
CREATE TRIGGER on_student_added
AFTER INSERT ON students
FOR EACH ROW
EXECUTE FUNCTION remove_student();
```



```
CREATE OR REPLACE FUNCTION remove_student()
RETURNS TRIGGER AS
$$
BEGIN
    DELETE FROM students WHERE id = NEW.id;
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

Triggers — Final Notes

- Triggers give you the freedom to do **dangerous** things
 - Recursive or circular triggers are perfectly valid
 - Difficult to spot when many tables are involved → chain reaction
 - Example: An **INSERT** into Table **A**, triggers an **INSERT** into **A**, which triggers in **INSERT** into **A**, ...

infinite loop

```
CREATE TRIGGER on_insert_A
AFTER INSERT ON table_A
FOR EACH ROW
EXECUTE FUNCTION insert_into_A();
```

```
CREATE OR REPLACE FUNCTION insert_into_A()
RETURNS TRIGGER AS
$$
BEGIN
    INSERT INTO table_A (0);
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

Summary

- Triggers — event-condition-action (ECA) rule for databases

- Powerful tool to automate actions for certain events (i.e., database operations)
- Built on top of stored function (trigger function = "special" stored function)
- Common use case: check constraint not captured by relational schema
- Well-defined arguments: events, timing, granularity

- Powerful → (potentially) dangerous

- Typically require take care
- Behavior not always intuitive
- Possibility of "bad cases"
(e.g., chain reactions, circular firings)

*"With great power comes
great responsibility!"*

Spiderman's Uncle Ben

Solutions to Quick Quizzes



Solutions to Quick Quizzes

