Integrity, Assertion Procedure, Function, and Trigger

CE384: Database Design
Maryam Ramezani
Sharif University of Technology
maryam.ramezani@sharif.edu

Integrity Constraints

- An integrity constraint (IC) describes conditions that every legal instance of a relation must satisfy.
  - Inserts/deletes/updates that violate IC's are disallowed.
  - Can be used to ensure application semantics (e.g., sid is a key), or prevent inconsistencies (e.g., sname has to be a string, age must be < 200).
- Types of IC's:
  - domain constraints and NOT NULL constraints,
  - primary key constraints and foreign key constraints,
  - general constraints.

Not-Null Constraints

- The IC NOT NULL disallows NULL values for a specified attribute.

```
CREATE TABLE Students
   (sid VARCHAR(20) PRIMARY KEY,
    name VARCHAR(20) NOT NULL,
    login VARCHAR(10) NOT NULL,
    age INTEGER,
    gpa REAL);
```

- Primary key attributes are implicitly NOT NULL.

General Constraints

- Attribute-based CHECK
  - defined in the declaration of an attribute,
  - activated on insertion to the corresponding table or update of attribute.
- Tuple-based CHECK
  - defined in the declaration of a table,
  - activated on insertion to the corresponding table or update of tuple.
- Assertion
  - defined independently from any table,
  - activated on any modification of any table mentioned in the assertion.

General Constraints

- Can use general SQL queries to express constraints.
- Much more powerful than domain and key constraints.
- Constraints can be named.

Attribute-based CHECK

- Attribute-based CHECK constraint is part of an attribute definition.
- Is checked whenever a tuple gets a new value for that attribute (INSERT or UPDATE). Violating modifications are rejected.
- CHECK constraint can contain an SQL query referencing other attributes (of the same or other tables), if their relations are mentioned in the FROM clause.
- CHECK constraint is not activated if other attributes mentioned get new values.

Attribute-based CHECK

- Attribute-based CHECK constraints are most often used to restrict allowable attribute values.

```
CREATE TABLE   Sailors
   ( sid  INTEGER PRIMARY KEY,
   sname  VARCHAR(10),
   rating  INTEGER
      CHECK  ( rating >= 1
      AND rating <= 10),
   age  REAL);
```

Tuple-based CHECK

- Tuple-based CHECK constraints can be used to constrain multiple attribute values within a table.
- Condition can be anything that can appear in a WHERE clause.

```
CREATE TABLE   Sailors
  ( sid  INTEGER PRIMARY KEY,
  sname  VARCHAR(10),
  previousRating  INTEGER,
  currentRating  INTEGER,
  age  REAL,
    CHECK  (currentRating >= previousRating)
   );
```

Assertions

- Condition can be anything allowed in a WHERE clause.
- Violating modifications are rejected.
- Components include:
  - a constraint name,
  - followed by `CHECK`,
  - followed by a condition

Row-Based Checks

- You can also check a combination of attribute values at INSERT/UPDATE time
  - Only Joe's restaurant can sell food for more than $5:

```
CREATE TABLE Sells (
    restaurant          CHAR(20),s
    food          CHAR(20),
    price  REAL,
    CHECK (restaurant = 'Joe''s restaurant'
                        OR
                        price <= 5.00)
);
```

For more Complex Constraints

```
CREATE ASSERTION assertionName
      CHECK ( condition );
```

No restaurant can charge more than $5 on average for food.

```
CREATE ASSERTION NoExpensiverestaurants
CHECK (
       NOT EXISTS (
                   SELECT restaurant
                   FROM Sells
                   GROUP BY restaurant
                   HAVING 5.00 < AVG(price)
       ));
```

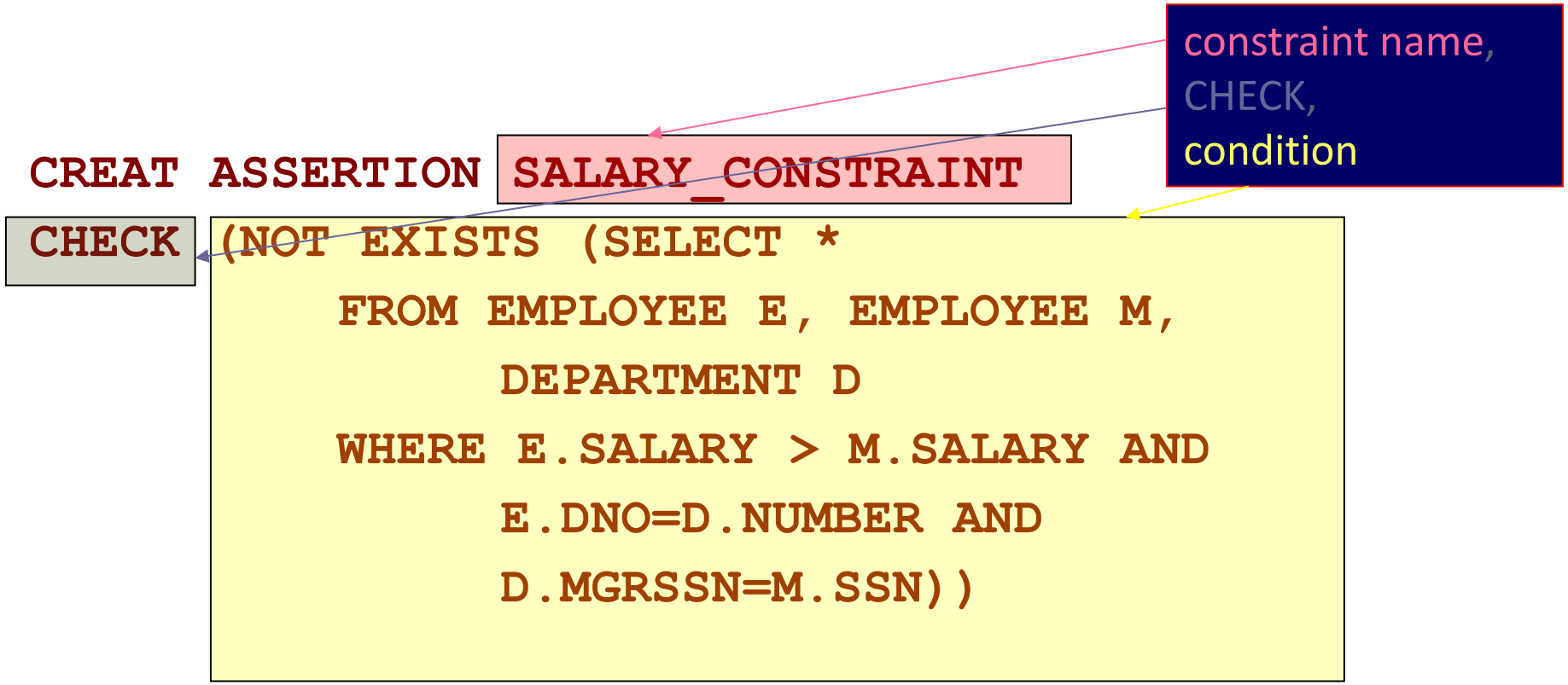
Foods with an average price above $5

Constraints as Assertions

- General constraints: constraints that do not fit in the basic SQL categories
  Mechanism: **`CREAT ASSERTION`**
  - Components include:
    - a constraint name,
    - followed by `CHECK`,
    - followed by a condition

Assertions: An Example

- "The salary of an employee must not be greater than the salary of the manager of the department that the employee works for''



Using General Assertions

- Specify a query that violates the condition; include inside a `NOT EXISTS` clause.
- Query result must be empty.
  - if the query result is not empty, the assertion has been violated

Assertion

```
customer(name, addr, phone)
restaurant(name, addr, license)
food(name, nationality)
```

There cannot be more restaurants than customers.

```
CREATE ASSERTION FewRestaurant CHECK (
      (SELECT COUNT (*) FROM restaurant)
      <=
      (SELECT COUNT (*) FROM customer)
);
```

Note

In theory, every ASSERTION is checked after every INSERT/ DELETE/UPDATE.

In practice, the DBMS only has to check sometimes:

- Adding a customer can't violate Fewrestaurants.
- Removing a restaurant can't violate NoExpensiverestaurants.
- Lowering a food price can't violate NoExpensiverestaurants.

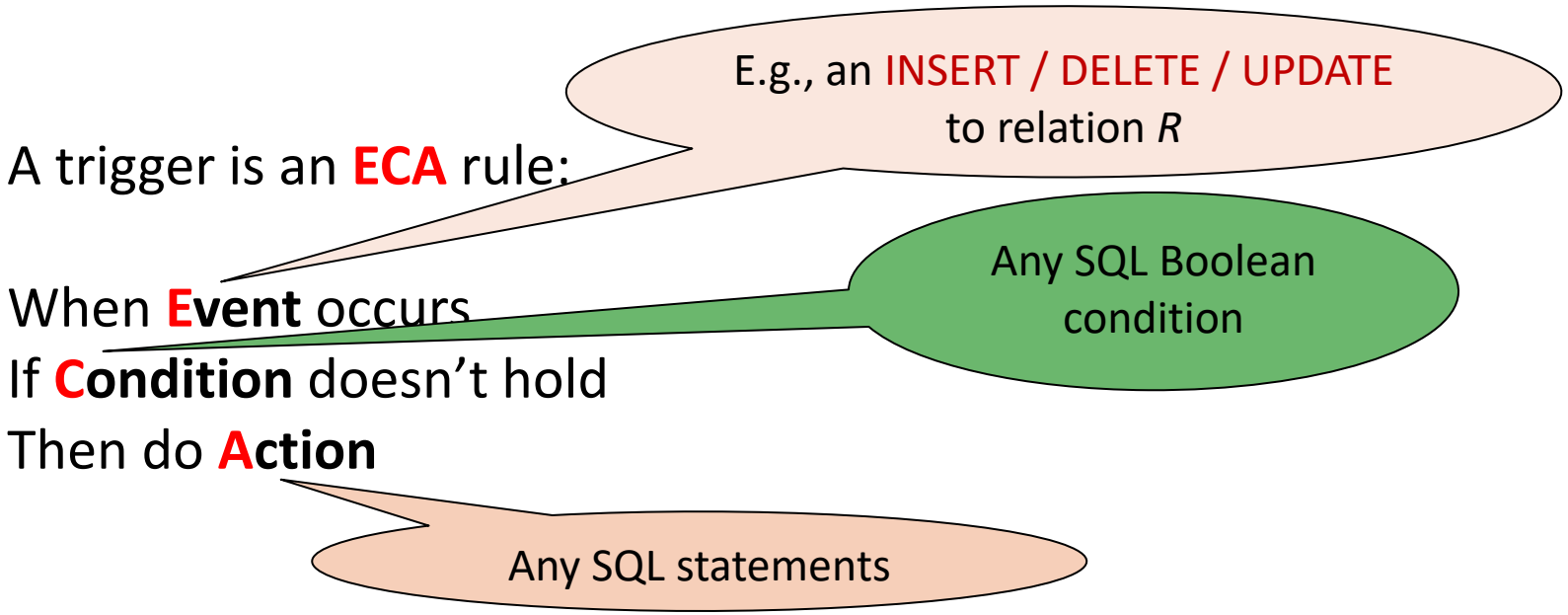
But is the DBMS smart enough to figure this out?

Postgres: SQL Error [0A000]: ERROR: CREATE ASSERTION is not yet implemented

Assertion is not implemented in vast majority of DBMS.

Trigger

You can help your not-so-smart DBMS by using TRIGGERs instead of ASSERTIONs.



You can use triggers to code very complex stuff

- You can allow your users to update their views --- but you catch their updates and rewrite them to behave the way you want, avoiding view anomalies.
- You can encode new strategies for handling violations of constraints, different from what the DBMS offers.
- When the event happens, the system will check the constraint, and if satisfied, will perform the action.
- NOTE: triggers may cause cascading effects.
- Triggers not part of SQL2 but included in SQL3… however, database vendors did not wait for standards with triggers!

Procedures

- A procedure is a module performing one or more actions; it does not need to return any values.
- The syntax for creating a procedure is as follows:

```
CREATE OR REPLACE PROCEDURE name
  [(parameter[, parameter, ...])]
AS
  [local declarations]
BEGIN
  executable statements
[EXCEPTION
  exception handlers]
END [name];
```

Main Procedure Constructs

- Local variables (DECLARE)
- RETURN values for FUNCTION
- Assign variables with SET
- Branches and loops:
  - IF (condition) THEN statements;
    ELSEIF (condition) statements;
    … ELSE statements; END IF;
  - LOOP statements; END LOOP
- Queries can be parts of expressions
- Can use cursors naturally without “EXEC SQL”

Procedures

- A procedure may have 0 to many parameters.
- Every procedure has two parts:
  - The header portion, which comes before AS (sometimes you will see IS—they are interchangeable), keyword (this contains the procedure name and the parameter list),
  - The body, which is everything after the AS keyword.
- The word REPLACE is optional. When the word REPLACE is not used in the header of the procedure, in order to change the code in the procedure, it must be dropped first and then re-created.
- Parameters are the means to pass values to and from the calling environment to the server. These are the values that will be processed or returned via the execution of the procedure.
- There are three types of parameters:
  - IN, OUT, and IN OUT. Modes specify whether the parameter passed is read in or a receptacle for what comes out. [In Postgres we don't have "out" parameter]

Procedures

- Type of parameters

IN	OUT	INOUT
The default	Explicitly specified	Explicitly specified
Pass a value to function	Return a value from a function	Pass a value to a function and return an updated value.
`in` parameters act like constants	`out` parameters act like uninitialized variables	`inout` parameters act like initialized variables
Cannot be assigned a value	Must assign a value	Should be assigned a value

Procedures

- A stored procedure does not return a value. You cannot use the return statement with a value inside a store procedure like this:
  - return expression;

- However, you can use the return statement without the expression to stop the stored procedure immediately:
  - return;

- If you want to return a value from a stored procedure, you can use parameters with the inout mode.

Procedures Example

```
drop table if exists accounts;

create table accounts (
    id int generated by default as identity,
    name varchar(100) not null,
    balance dec(15,2) not null,
    primary key(id)
);

insert into accounts(name,balance)
values('Bob',10000);

insert into accounts(name,balance)
values('Alice',10000);
```

```
create or replace procedure transfer
(sender int, receiver int, amount dec )
language plpgsql
as
$$
begin
-- subtracting the amount from the sender's
account
        update accounts
        set balance = balance - amount
        where id = sender;
-- adding the amount to the receiver's
account         update accounts
        set balance = balance + amount
        where id = receiver;
commit;
 end;
$$;
```

Note

- Stored procedure do not have to be written in SQL:

```
CREATE PROCEDURE TopSailors(IN num INTEGER)
LANGUAGE JAVA
EXTERNAL NAME "file:///c:/storedProcs/rank.jar"
```

Call a Procedure

- `Call stored_procedure_name(argument_list);`

```
call transfer(1,2,1000);
```

```
select * from accounts;
```

Output:

```
 id | name  | balance
----+-------+----------
  1 | Bob   | 10000.00
  2 | Alice | 10000.00
(2 rows)
```

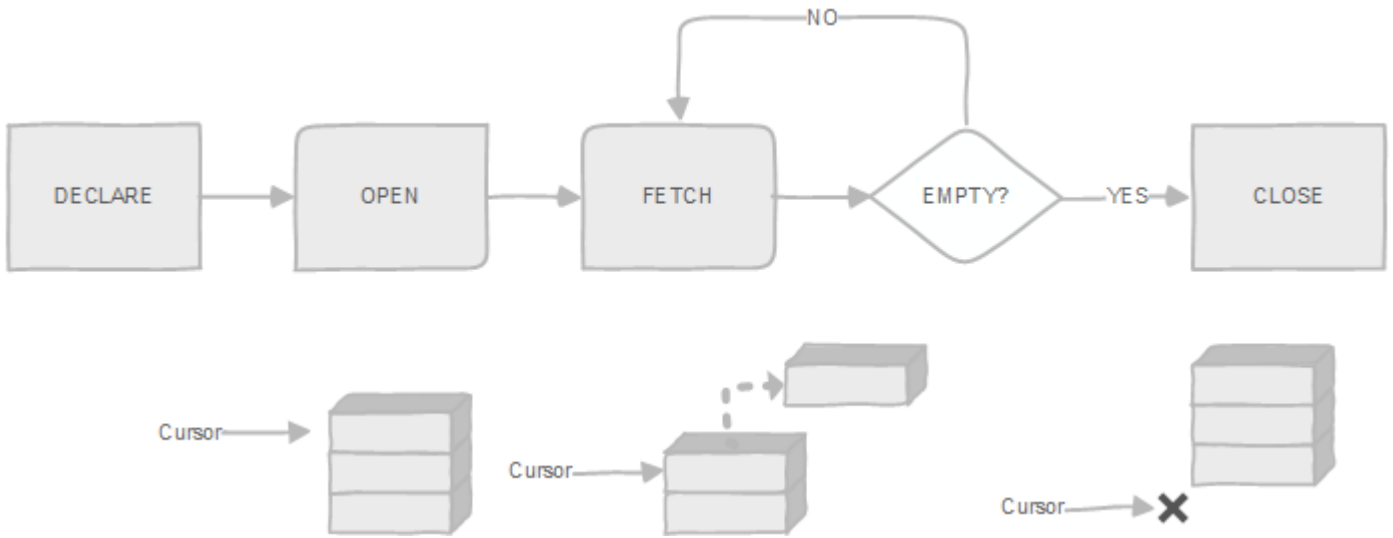
```
SELECT * FROM accounts;
```

Output:

```
 id | name  | balance
----+-------+----------
  1 | Bob   |  9000.00
  2 | Alice | 11000.00
(2 rows)
```

PL/pgSQL Cursor

- A cursor is a database object that allows you to traverse the result set of a query one row at a time.
- Cursors can be useful when you deal with large result sets or when you need to process rows sequentially.



1. First, declare a cursor.
2. Next, open the cursor.
3. Then, fetch rows from the result set into a record or a variable list.
4. After that, process the fetched row and exit the loop if there is no more row to fetch.
5. Finally, close the cursor.

Function

- Functions are a type of stored code and are very similar to procedures.
- The significant difference is that a function is a PL/SQL block that returns a single value.
- Functions can accept one, many, or no parameters, but a function must have a return clause in the executable section of the function.
- The datatype of the return value must be declared in the header of the function.
- A function is not a stand-alone executable in the way that a procedure is: It must be used in some context. You can think of it as a sentence fragment.
- A function has output that needs to be assigned to a variable, or it can be used in a SELECT statement.

Function

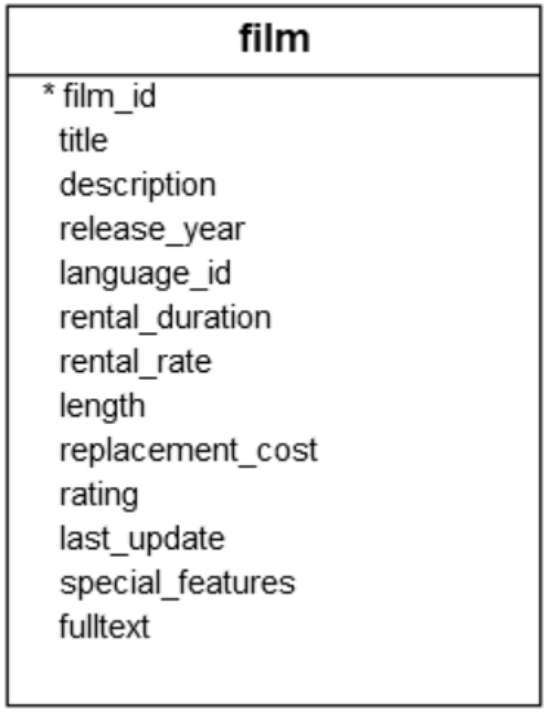
- The function does not necessarily have to have any parameters, but it must have a RETURN value declared in the header, and it must return values for all the varying possible execution streams.
- The RETURN statement does not have to appear as the last line of the main execution section, and there may be more than one RETURN statement (there should be a RETURN statement for each exception).
- A function may have IN, OUT, or IN OUT parameters. but you rarely see anything except IN parameters.

Function

```
create [or replace] function
function_name(param_list)
   returns return_type
   language plpgsql
  as
$$
declare
   -- variable declaration
begin
   -- logic
end;
$$;
```

- First, specify the name of the function after the create function keywords. To replace the existing function, use the or replace option.
- Then, list out parameters surrounded by parentheses after the function name. A function can have zero or more parameters.
- Next, define the datatype of the returned value after the returns keyword.
- After that, use the language plpgsql to define the procedural language of the function. Note that PostgreSQL supports many languages including plpgsql.
- Finally, place a block in the dollar-quoted string constant to define the function body.

Example



creates a function that returns the number films whose length between the len_from and len_to parameters

```
create function get_film_count(len_from int, len_to int)
returns int
language plpgsql
as
$$
declare
   film_count integer;
begin
   select count(*)
   into film_count
   from film
   where length between len_from and len_to;

   return film_count;
end;
$$;
```

Output:

```
CREATE FUNCTION
```

Call a Function

- Using positional notation

```
select get_film_count(
    len_from => 40,
     len_to => 90
);
```

- Using named notation

```
select get_film_count(
    len_from := 40,
    len_to := 90
);
```

- Using mixed notation

```
select get_film_count(40, len_to => 90);
```

```
select get_film_count(len_from => 40, 90);
```

Functions with different type of parameters

- In mode:

```
select * from find_film_by_id(1);
```

```
create or replace function find_film_by_id(p_film_id int)
returns varchar
language plpgsql
as $$
declare
   film_title film.title%type;
begin
  -- find film title by id
  select title
  into film_title
  from film
  where film_id = p_film_id;

  if not found then
     raise 'Film with id % not found', p_film_id;
  end if;

  return film_title;

end;$$
```

Functions with different type of parameters

- Out mode:

```
select get_film_stat();
```

```
create or replace function get_film_stat(
    out min_len int,
    out max_len int,
    out avg_len numeric)
language plpgsql
as $$
begin

  select min(length),
         max(length),
                 avg(length)::numeric(5,1)
  into min_len, max_len, avg_len
  from film;

end;$$
```

Functions with different type of parameters

- InOut mode:

```
select * from swap(10,20);
```

```
create or replace function swap(
        inout x int,
        inout y int
)
language plpgsql
as $$
begin
   select x,y into y,x;
end; $$;
```

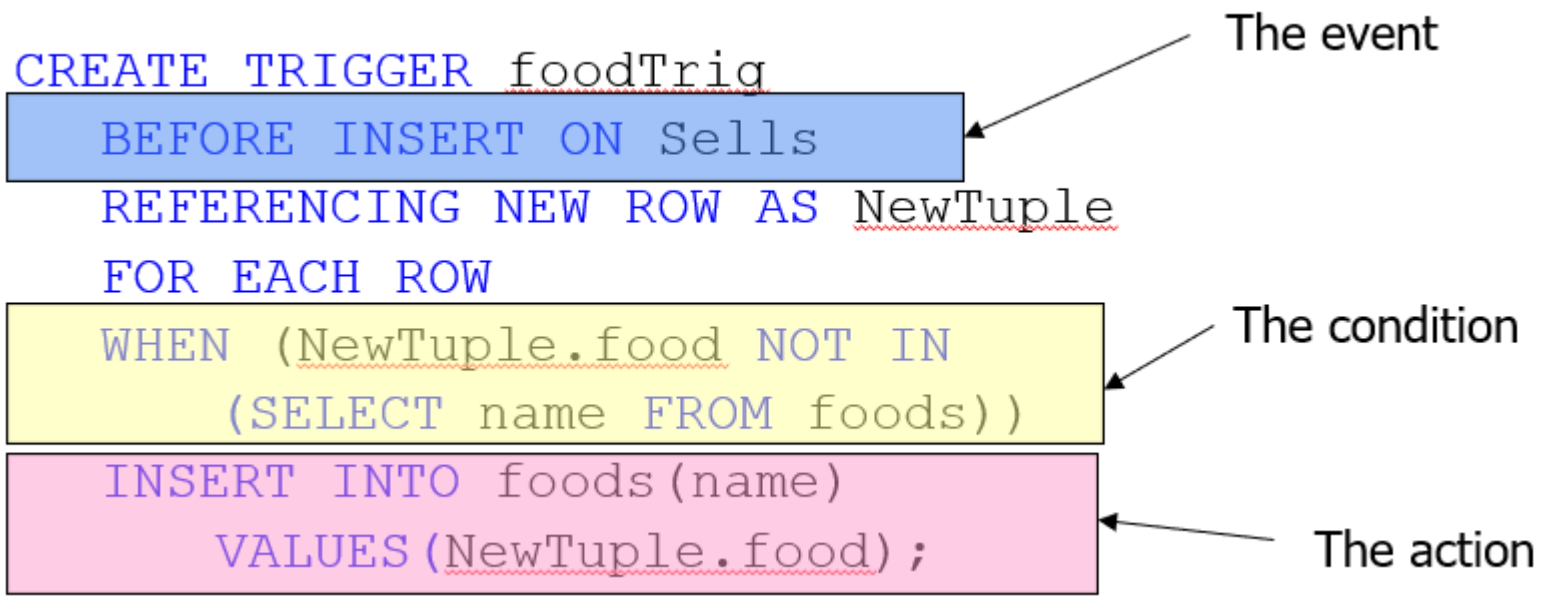
Function

- List all defined functions:

```
select * from pg_proc p
left join pg_namespace n on p.pronamespace = n.oid
left join pg_language l on p.prolang = l.oid
left join pg_type t on t.oid = p.prorettype
where n.nspname not in ('pg_catalog', 'information_schema')
order by n.nspname , p.proname
```

Trigger Example

- If someone inserts an unknown food into `Sells(restaurant,food,price)` add it to `food(name,nationality)` with a NULL nationality.

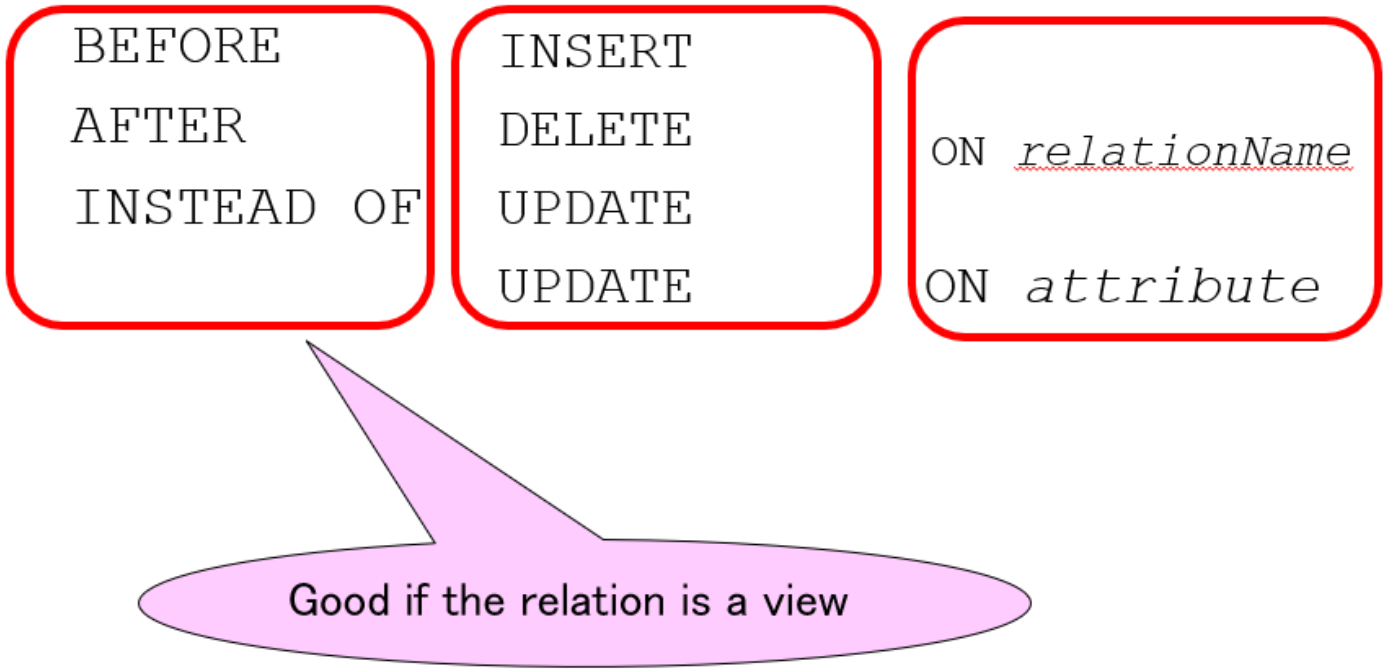


Syntax for Naming the Trigger

- `CREATE TRIGGER` *name*
- `CREATE OR REPLACE TRIGGER` *name*
  - Useful when there is a trigger with that name and you want to modify the trigger.
- Two different triggers on a table?

Syntax for Describing the Condition

- Take one element from each of the three columns:



Execute Trigger

- You can execute a trigger once per modified row, or once per triggering statement.

The default

- Statement-level triggers execute once for each SQL statement that triggers them, regardless of how many rows are modified.
- Row level triggers are executed once for each modified row.

Request explicitly by including
FOR EACH ROW

Execute Trigger

- A statement trigger fires once per triggering event and regardless of whether any rows are modified by the insert, update, or delete event.
- A row trigger fires once for each row affected by the triggering event. If no rows are affected, the trigger does not fire.
- `[FOR EACH {ROW | STATEMENT}]`

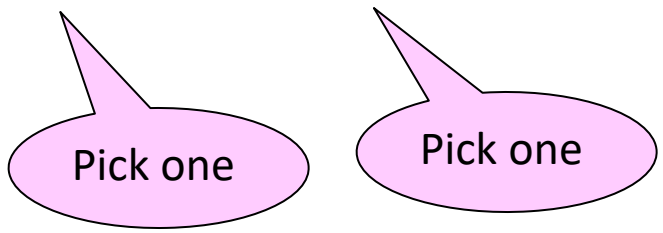
DML

Your condition & action can refer to the rows being inserted / deleted/updated.

- INSERT statements imply a new row (for row-level) or new set of rows (for statement-level).
- DELETE implies an old row (row-level) or table (statement-level).
- UPDATE implies both.

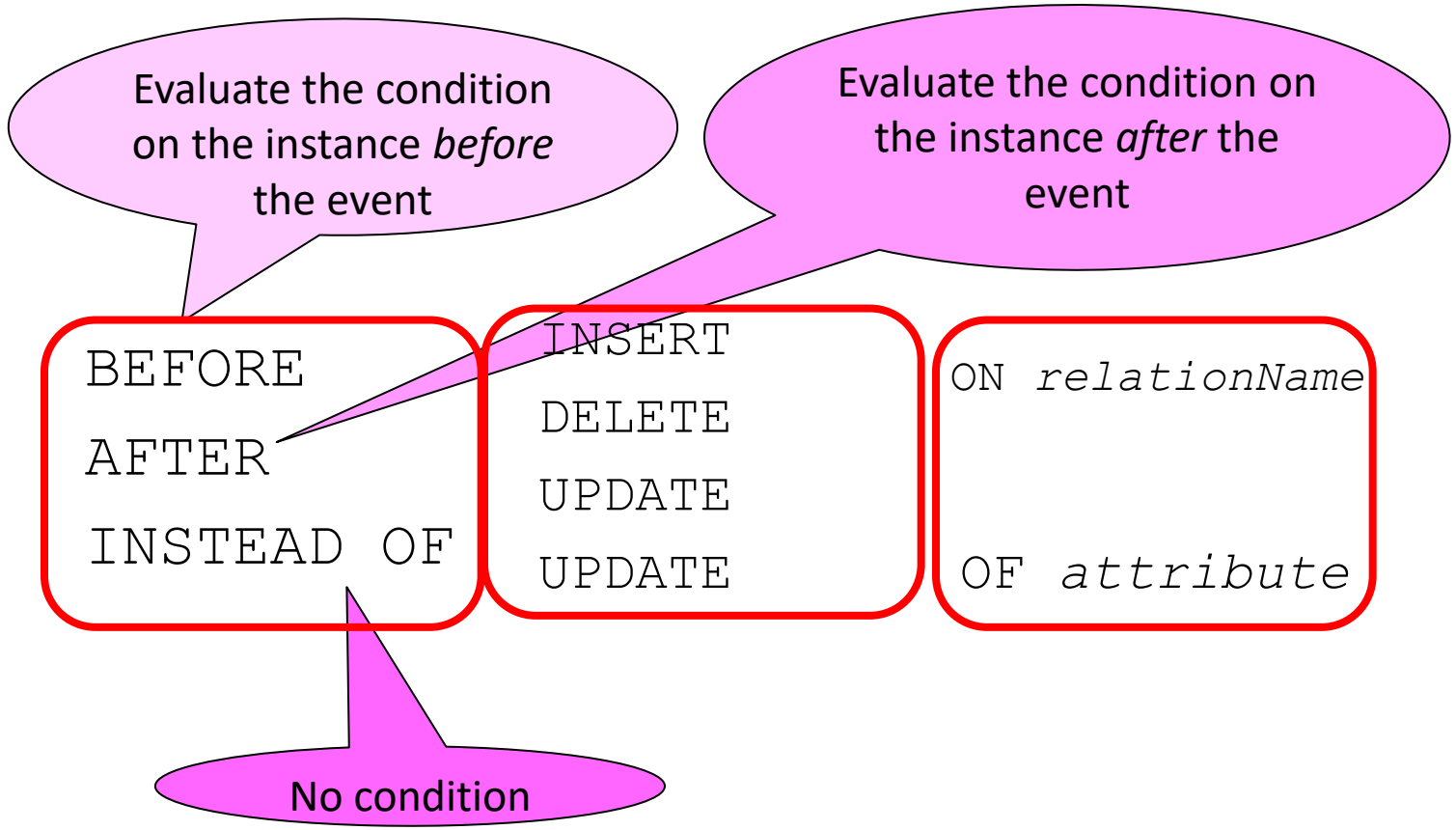
Syntax:

```
REFERENCING [NEW OLD][ROW TABLE] AS name
```



DML

- Any boolean-valued *Condition* is ok in WHEN *Condition*.

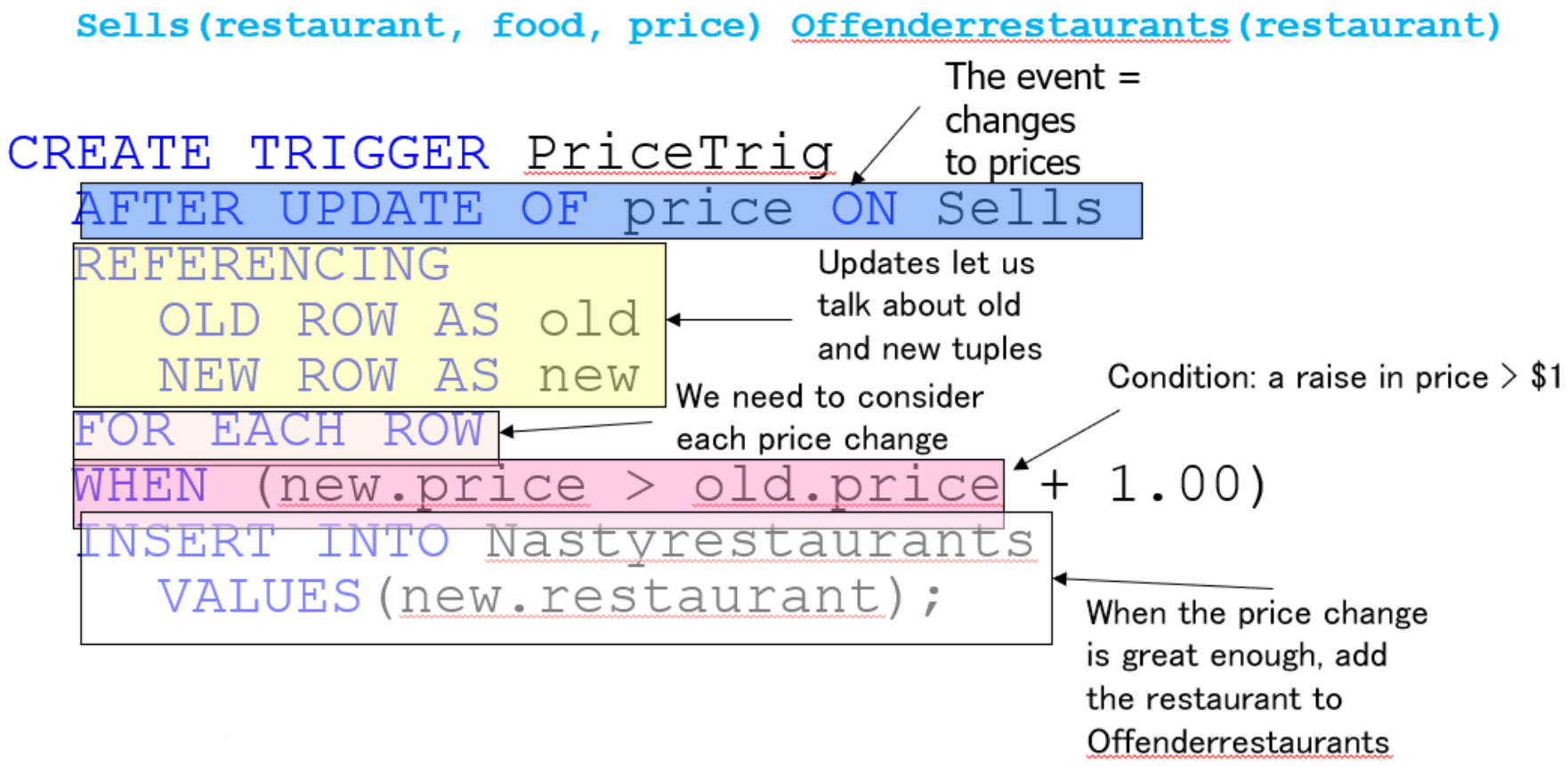


Action as a Sequence

- The Action is a sequence of SQL statements (modifications).
- Surround them by BEGIN . . . END if there is more than one.

Example

Remember restaurants that raise the price of a food by > $1.



Note

Triggers are great for implementing view updates.

Example: Updating Views

- How can I insert a tuple into a table that doesn't exist?

Employee(ssn, name, department, project, salary)

```
CREATE VIEW  Developers AS
   SELECT ssn,name, project
   FROM  Employee
   WHERE department = "Development"
```

We cannot insert into Developers --- it is a view. But we can use an INSTEAD OF trigger to turn a (name, project) triple into an insertion of a tuple (name, `Development', project) to Employee.

If we make the following insertion:

```
INSERT INTO  Developers
VALUES(12,"Joe", "Optimizer")
```

This must be "Development"

It becomes:

```
INSERT INTO  Employee
VALUES(12, "Joe", NULL, "Optimizer", NULL)
```

Allow insertions into Developers

```
CREATE TRIGGER AllowInsert
   INSTEAD OF INSERT ON Developers
   REFERENCING NEW ROW AS new
   FOR EACH ROW
   BEGIN
       INSERT INTO Empolyees(name, department, project)
       VALUES(new.name, `Development', new.project);
   END;
```

SQL Triggers: An Example

- A trigger to compare an employee's salary to his/her supervisor during insert or update operations:

```
CREATE TRIGGER INFORM_SUPERVISOR
BEFORE INSERT OR UPDATE OF
   SALARY, SUPERVISOR_SSN ON EMPLOYEE
   FOR EACH ROW
      WHEN
      (NEW.SALARY> (SELECT SALARY FROM EMPLOYEE
                    WHERE SSN=NEW.SUPERVISOR_SSN))
      INFORM_SUPERVISOR (NEW.SUPERVISOR_SSN,NEW.SSN);
```

Example: Row Level Trigger

```
CREATE TRIGGER    NoLowerPrices
AFTER UPDATE OF  price  ON Product
REFERENCING
     OLD  AS OldTuple
     NEW  AS  NewTuple
FOR EACH ROW
WHEN (OldTuple.price > NewTuple.price)
     UPDATE  Product
     SET  price = OldTuple.price
     WHERE  name = NewTuple.name
```

Statement Level Trigger

emp(dno…), dept(dept#, …)

- Whenever we insert employees tuples, make sure that their dno's exist in Dept.

```
CREATE TRIGGER  deptExistTrig
AFTER INSERT ON emp
REFERENCING
     OLD_TABLE  AS OldStuff
     NEW_TABLE AS  NewStuff
WHEN (EXSITS (SELECT * FROM NewStuff
              WHERE dno NOT IN
                    (SELECT dept# FROM dept)))
DELETE FROM NewStuff
         WHERE dno NOT IN
             (SELECT dept# FROM dept));
```

Bad Things Can Happen

```
CREATE TRIGGER  Bad-trigger
AFTER UPDATE OF price IN Product
REFERENCING  OLD AS  OldTuple
             NEW AS NewTuple
FOR EACH ROW
WHEN    (NewTuple.price > 50)
         UPDATE  Product
         SET  price = NewTuple.price * 2
         WHERE  name = NewTuple.name
```

Insert on View Example

```
STT1 (STID, NAME, MAJOR, LEVEL)
STT2 (STID, DEPT, BDATE, NATID)

CREATE  VIEW  CE-STT
        AS  SELECT  STID,  NAME,  MAJOR
        FROM  STT1  JOIN  STT2
        WHERE  DEPT='CE'  AND  LEVEL ='BS'

CREATE TRIGGER  INS-VIEW-TRIG
        INSTEAD  OF  INSERT  ON  CE-STT
        REFERENCING  NEW  AS NST
        FOR EACH ROW
        BEGIN
              INSERT  INTO  STT1  VALUES  ( NST.STID, NST.NAME,
                                            NST.MAJOR,  'BS')
              INSERT  INTO  STT2  VALUES  ( NST.STID, 'CE', NULL, NULL)
        END
```

Check - Example

- Checking the TOTAL_SAL of department is sum of the salary of employees.

EMPL (EID, ENAME, SALARY, DNO)
DEPT (DNO, DNAME, TOTAL_SAL, MANAGER)

```
(R1) CREATE TRIGGER TOTALSAL1
        AFTER  INSERT ON EMPL
        FOR EACH ROW
        WHEN (NEW.DNO IS NOT NULL)
             UPDATE DEPT
                SET  TOTAL_SAL = TOTAL_SAL + NEW.SALARY
                WHERE DNO = NEW.DNO

(R2) CREATE TRIGGER TOTALSAL2
        AFTER  UPDATE OF SALARY ON EMPL
        FOR EACH ROW
        WHEN (NEW.DNO IS NOT NULL)
             UPDATE DEPT
                SET  TOTAL_SAL = TOTAL_SAL + NEW.SALARY - OLD.SALARY
                WHERE DNO = NEW.DNO
```

Check - Example

- Checking the TOTAL_SAL of department is sum of the salary of employees.

EMPL (EID, ENAME, SALARY, DNO)
DEPT (DNO, DNAME, TOTAL_SAL, MANAGER)

```
(R3) CREATE TRIGGER TOTALSAL3
          AFTER  UPDATE OF DNO ON EMPL
          FOR EACH ROW
          BEGIN
               UPDATE DEPT
                  SET  TOTAL_SAL = TOTAL_SAL + NEW.SALARY
                  WHERE DNO = NEW.DNO
               UPDATE DEPT
                  SET  TOTAL_SAL = TOTAL_SAL - OLD.SALARY
                  WHERE DNO = OLD.DNO
          END

(R4) CREATE TRIGGER TOTALSAL4
          AFTER  DELETE ON EMPL
          FOR EACH ROW
          WHEN (OLD.DNO IS NOT NULL)
               UPDATE DEPT
                  SET   TOTAL_SAL = TOTAL_SAL - OLD.SALARY
                  WHERE  DNO = OLD.DNO
```