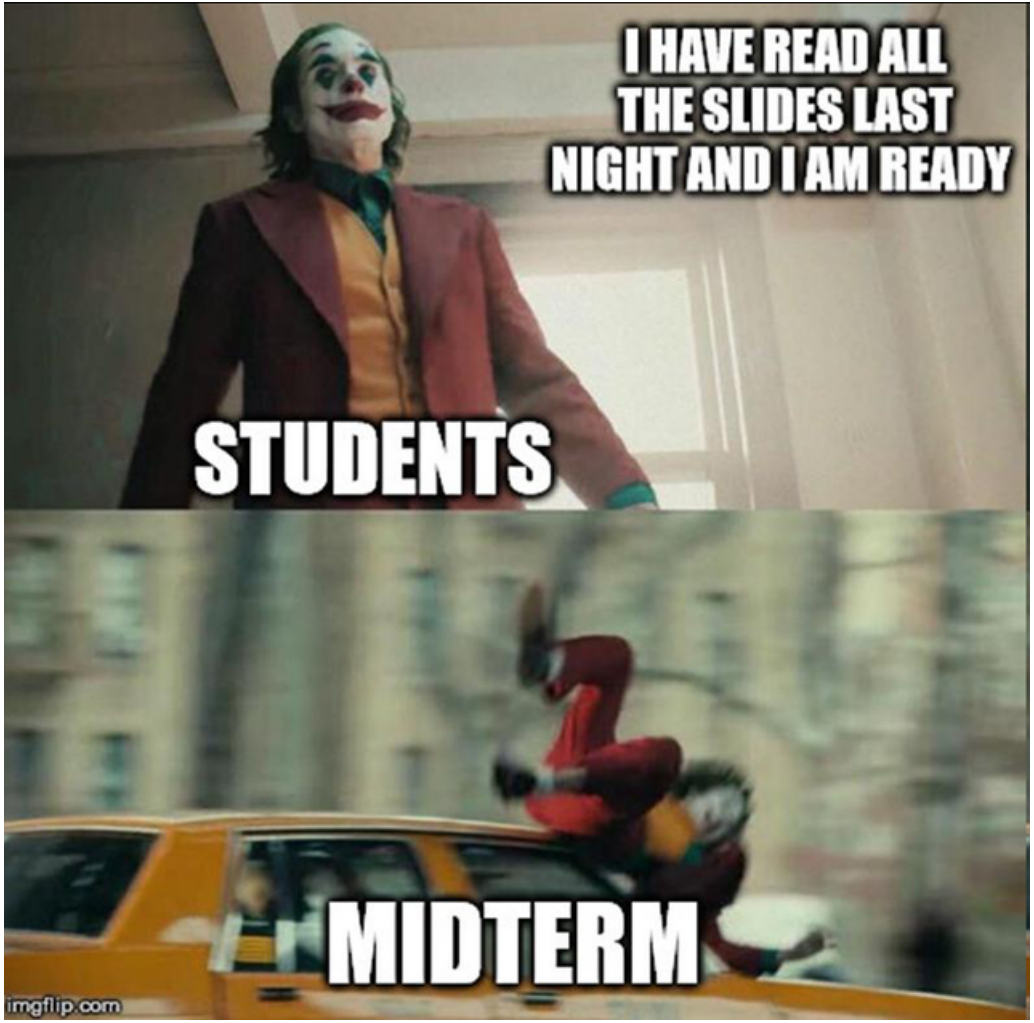

Hardest choice ever
i love
you
memes


I HAVE READ ALL THE SLIDES LAST NIGHT AND I AM READY
STUDENTS
MIDTERM
imgflip.com

Databases - Tutorial 09
Stored procedures, functions, triggers and PL/ pgSql

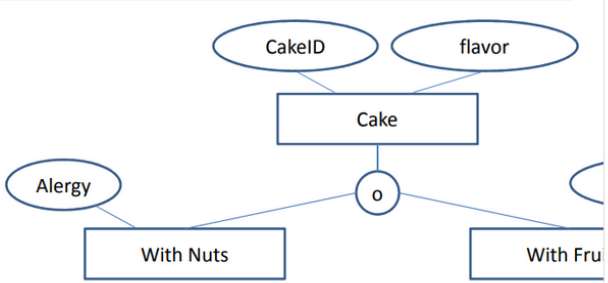
Hamza Salem - Innopolis University

Question **23**

Incorrect

Mark 0.00 out of 1.00

Flag question



Given the digram above, which of the following statements are correct?

- [ ] a. cake must be with nuts or fruit
- [x] b. cake can be with nuts and fruit together ✔
- [x] c. cake can be with nuts only ✖
- [x] d. cake can be without nuts or fruit ✖

Your answer is incorrect.

The correct answers are:
cake must be with nuts or fruit,

cake can be with nuts and fruit together

Specialization **Disjointness** Constraint

Specifies that the subclasses of the specialization must be **disjoint**: **an entity can be a member of at most one of the subclasses of the specialization**

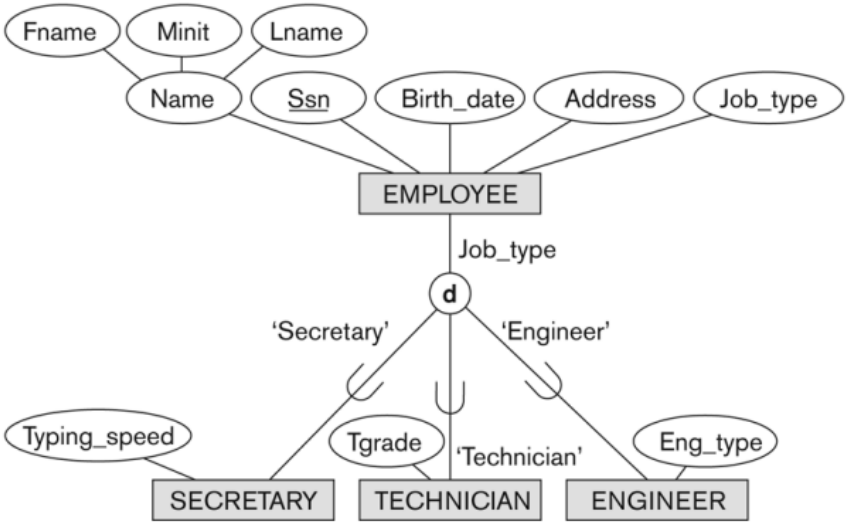
1 or 0

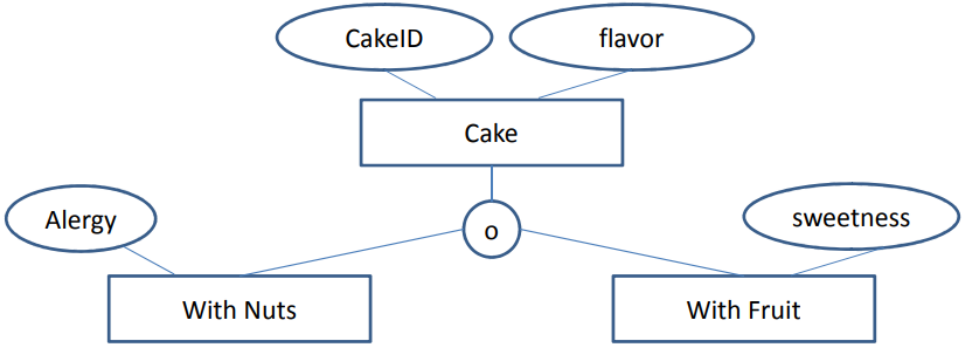
Specified by d in EER diagram

If not disjoint, specialization is **overlapping**: **that is the same entity may be a member of more than one subclass of the specialization**

>1 or 0

Specified by o in EER diagram





Specialization **Completeness** Constraint

Total specifies that **every entity in the superclass must be a member of some subclass in the specialization**

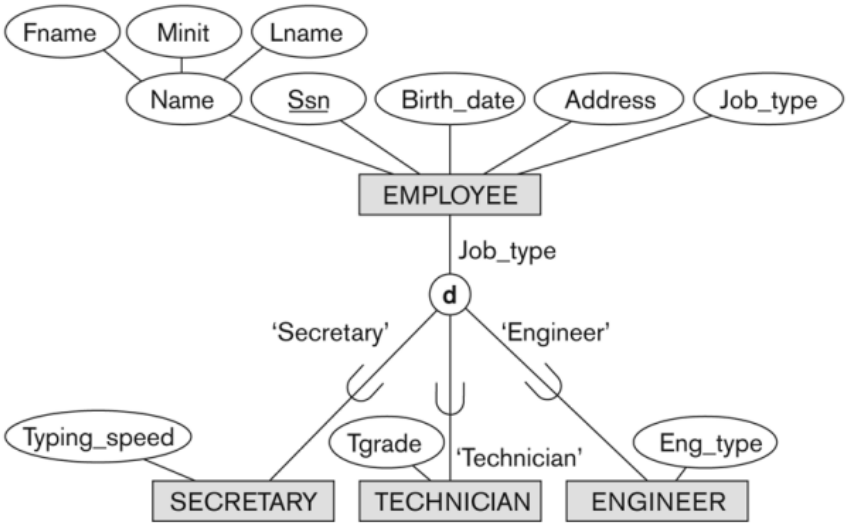
1 and only 1

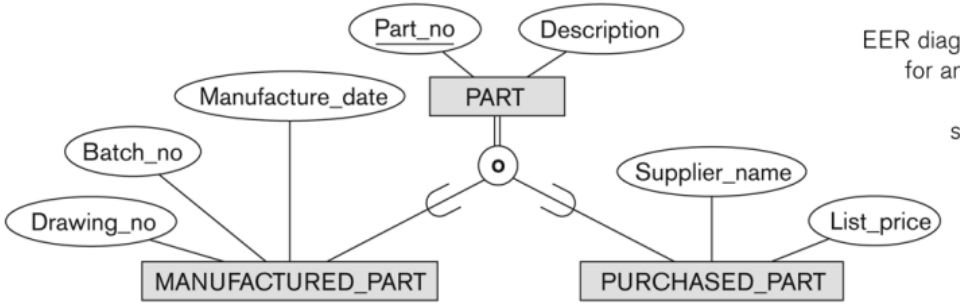
Shown in EER diagrams by a double line.

Partial allows **an entity not to belong to any of the subclasses**

Can be 0

Shown in EER diagrams by a single line.





Contents

- Stored procedures
- Functions
- Triggers
- PL/ pgSql

Stored procedures

The store procedures define functions for creating triggers or custom aggregate functions.

PostgreSQL categorizes the procedural languages into two main groups:

1. Safe languages can be used by any users. **SQL and PL/pgSQL are safe languages**.
2. Sand-boxed languages are only used by **super users because sand-boxed languages provide the capability to bypass security and allow access to external sources.** C is an example of a sandboxed language.

```
CREATE PROCEDURE insert_data(a integer, b integer)
LANGUAGE SQL
AS $$
INSERT INTO tbl VALUES (a);
INSERT INTO tbl VALUES (b);
$$;
```

Stored procedures advantage

1. **Reduce the number of round trips between applications and database servers.** All SQL statements are wrapped inside a function stored in the PostgreSQL database server so the application only has to issue a function call to get the result back instead of **sending multiple SQL statements** and wait for the result between each call.
2. **Increase application performance because the user-defined functions and stored procedures are pre-compiled and stored in the PostgreSQL database server**.
3. Reusable in many applications. Once you develop a function, you can reuse it in any applications.

```
CREATE PROCEDURE insert_data(a
integer, b integer)
LANGUAGE SQL
BEGIN ATOMIC
  INSERT INTO tbl VALUES (a);
  INSERT INTO tbl VALUES (b);
END;
```

```
CALL insert_data(1, 2);
```

Functions

In a function, it is mandatory to use the RETURNS and RETURN arguments, whereas in a stored procedure is not necessary

```
CREATE [OR REPLACE] FUNCTION function_name
(arguments)
RETURNS return_datatype
LANGUAGE plpgsql
AS $variable_name$
DECLARE
declaration;
[...] -- variable declaration
 BEGIN
< function_body >
[...]  -- logic
RETURN { variable_name | value }
END;
$$
```

Functions

In a function, it is mandatory to use the RETURNS and RETURN arguments, whereas in a stored procedure is not necessary

```
Create function get_car_Price(Price_from int, Price_to
int)
returns int
language plpgsql
as
$$
Declare
 Car_count integer;
Begin
  select count(*)
  into Car_count
  from Car
  where Car_price between Price_from and Price_to;
  return Car_count;
End;
$$;
```

How to Call a user-defined function

In PostgreSQL, we can call the user-defined function in three ways, which are as follows:

- **Positional notation**
- **Named notation**
- **The mixed notation**

```
Select get_car_Price(26000,70000);
```

```
select get_car_Price(
 Price_from => 26000,
   Price_to => 70000
);
```

```
select get_car_Price(
 Price_from := 26000,
    Price_to := 70000
);
```

```
select get_car_Price(Price_from=>26000,70000);
```

Triggers

why we need to use the triggers and when to use them and also see the **merits and demerits** of **PostgreSQL triggers, features of PostgreSQL Triggers** and various command, which are performed under the PostgreSQL Trigger section.

- **Row-level trigger**
- **Statement-level trigger**

For example, if we issue an **UPDATE** command, which affects 10 rows, the **row-level trigger** will be invoked **10 times**, on the other hand, the **statement level trigger** will be invoked **1 time**.

- **CREATE Trigger**
- **ALTER Trigger**
- **DROP Trigger**
- **ENABLE Trigger**
- **DISABLE Trigger**

Triggers

- **Create trigger:** In PostgreSQL, the CREATE TRIGGER command generates our first trigger step by step.
- **Alter trigger:** The ALTER TRIGGER command is used to rename a trigger.
- **Drop trigger:** The DROP TRIGGER command is used to define the steps to remove a trigger from a table
- **Enable triggers:** In the PostgreSQL trigger, the ENABLE TRIGGER statement allows a trigger or all triggers related to a table.
- **Disable trigger:** The DISABLE TRIGGER is used to display how we can disable a trigger or all triggers linked with a table.

Triggers

The triggers can be used in the following aspects:

- The triggers can be used to **authenticate the input data**.
- The triggers can also **implement business rules**.
- It can easily **retrieve the system functions**.
- The triggers can be used to create a **unique value for a newly-inserted row in a diverse file**.
- The trigger can be used to **get the data from other files for cross-referencing objectives**.

```
CREATE TABLE emp (
    empname text,
    salary integer,
    last_date timestamp,
    last_user text
);

CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
    BEGIN
        -- Check that empname and salary are given
        IF NEW.empname IS NULL THEN
            RAISE EXCEPTION 'empname cannot be null';
        END IF;
        IF NEW.salary IS NULL THEN
            RAISE EXCEPTION '% cannot have null salary', NEW.empname;
        END IF;

        -- Who works for us when they must pay for it?
        IF NEW.salary < 0 THEN
            RAISE EXCEPTION '% cannot have a negative salary',
NEW.empname;
        END IF;

        -- Remember who changed the payroll when
        NEW.last_date := current_timestamp;
        NEW.last_user := current_user;
        RETURN NEW;
    END;
$emp_stamp$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE FUNCTION emp_stamp();
```

PL/ pgSql

PL/ pgSQL

- PL/ pgSQL is **easy to learn and simple to use**.
- PL/ pgSQL **comes with PostgreSQL by default**.
- The user defined functions and stored procedures developed in PL/ pgSQL can be used like any **built in functions** and stored procedures.
- PL/ pgSQL inherits all user defined **types, functions, and operators**.
- PL/ pgSQL has **many features that allow you to develop complex functions and stored procedures**.
- PL/ pgSQL can be defined to be trusted by the PostgreSQL database server

For loop

For loop contains a counting variable which is not necessary to declare outside the for a loop. It can be declared in the for loop statement itself. This counting variable has START VALUE and an END VALUE as its range for which it will iterate.

```
FOR [counting variable name] IN [REVERSE] [START VALUE] .. [END VALUE] [BY step value] LOOP
[code/statements to repeat];
END LOOP;
```

For loop

```
CREATE TABLE foo (fooid INT, foosubid INT, fooname TEXT);
INSERT INTO foo VALUES (1, 2, 'three');
INSERT INTO foo VALUES (4, 5, 'six');

CREATE OR REPLACE FUNCTION get_all_foo() RETURNS SETOF foo AS
$BODY$
DECLARE
    r foo%rowtype;
BEGIN
    FOR r IN
        SELECT * FROM foo WHERE fooid > 0
    LOOP
        -- can do some processing here
        RETURN NEXT r; -- return current row of SELECT
    END LOOP;
    RETURN;
END;
$BODY$
LANGUAGE plpgsql;

SELECT * FROM get_all_foo();
```

If statement vs Case statement

Simple IF statements:

IF condition THEN
statement; END IF;

IF THEN ELSE statements:

IF condition THEN
statements;
ELSE
additional statements;
END IF;

```
SELECT
last_name , job_id , salary,
CASE job_id
WHEN 'ACCOUNT' THEN 1.10*salary
WHEN 'IT_PROG' THEN 1.15*salary
WHEN 'SALES' THEN 1.20*salary
ELSE salary END "REVISED_SALARY" FROM
employees;
```

Useful Links

- https://www.pgadmin.org/docs/pgadmin4/development/procedure_dialog.html