# EPIC REPORT

Made By Konsta Jalkanen 000489689, and Lauri Heiskanen 001135864.

Contents

[EPIC REPORT 1](#_Toc165804367)

[Roles: 2](#_Toc165804368)

[Triggers 4](#_Toc165804369)

[Procedures 9](#_Toc165804370)

[Functions 12](#_Toc165804371)

[Constraints 14](#_Toc165804372)

[Partitions 16](#_Toc165804373)

[Views 19](#_Toc165804374)

[Zip\_code 22](#_Toc165804375)

## Roles:

Made roles:

* Admin
* Employee
* Trainee
* Views\_only

We gave admin superuser status.

We gave employee predefined role ‘pg\_read\_all\_data’ which gives: “Read all data (tables, views, sequences), as if having SELECT rights on those objects, and USAGE rights on all schemas, even without having it explicitly. This role does not have the role attribute BYPASSRLS set. If RLS is being used, an administrator may wish to set BYPASSRLS on roles which this role is GRANTed to.” – postgresql documentation, e-documentation, referenced [21/04/2024], available at <https://www.postgresql.org/docs/current/predefined-roles.html>

Trainee has been granted ability to read tables project, customer, geo\_location, and project\_role. It also has permission to see traineeView which consists information from employee table rows id, name and email.

Views\_only has the permission to select on the created views.

Done by running:

CREATE ROLE admin SUPERUSER;

CREATE ROLE employee;

CREATE ROLE trainee;

CREATE ROLE views\_only;

GRANT pg\_read\_all\_data TO employee;

GRANT SELECT ON project, customer, geo\_location, project\_role TO trainee;

create or replace view traineeView

as select e\_id, emp\_name, email from employee;

GRANT SELECT ON traineeView TO trainee;

GRANT SELECT ON employees\_on\_project, employees\_in\_department\_in\_hq, employees\_by\_skill, customers\_by\_location, employees\_by\_group, employees\_by\_title TO views\_only;

## Triggers

Made triggers:

* skillCheck
* assignTrigger
* contractTrigger
* groupTrigger

With each having their own procedure

* skillChecking
* assignEmployees
* contractCheck
* groupCheck

Each trigger only calls the corresponding procedure. Here’s table to show when they trigger and what is the corresponding procedure

|  |  |  |
| --- | --- | --- |
| Trigger | Procedure | when |
| skillCheck | skillCheck | before insert on skills |
| assignTrigger | assignTrigger | after insert on project |
| contractTrigger | contractTrigger | before update of contract\_type on employee |
| groupTrigger | groupCheck | After insert on employee |

Explanation what each trigger does:

skillCheck:

Checks that is there skill with same name than the new input.

assignTrigger:

finds all employees that are in same country than the customer and chooses 3 on the top found to be in the project.

contractTrigger

checks that contract start date is today. Checks that there is an end date for temporary contract and that it is exactly two years after current day. Checks that there is no end date for non temporary contracts.

groupTrigger

checks if given job title ‘HR secretary’ and insert that employee to HR group. Checks if job title contains world admin and inserts that into admin group. Else they go employee group.

Done by running:

Create or replace function skillChecking() returns trigger

LANGUAGE plpgsql

AS $$

BEGIN

IF (SELECT count(\*) FROM skills where skills.skill = new.skill) > 0 THEN RAISE EXCEPTION 'This skill already exists';

END IF;

RETURN NEW;

END;

$$;

Create or replace trigger skillCheck

before insert on skills

FOR EACH ROW

EXECUTE function skillChecking();

Create or replace function assignEmployees() returns trigger

LANGUAGE plpgsql

AS $$

DECLARE

employee1 integer := (select e\_id from employee where employee.e\_id in

(select d\_id from department where department.hid in

(select h\_id from headquarters where headquarters.l\_id in

(select l\_id from geo\_location where geo\_location.l\_id > 999 and geo\_location.country

in (select country from geo\_location where geo\_location.l\_id = (select l\_id from customer where customer.c\_id = 5))))) limit 1 offset 0);

employee2 integer := (select e\_id from employee where employee.e\_id in

(select d\_id from department where department.hid in

(select h\_id from headquarters where headquarters.l\_id in

(select l\_id from geo\_location where geo\_location.l\_id > 999 and geo\_location.country

in (select country from geo\_location where geo\_location.l\_id = (select l\_id from customer where customer.c\_id = 5))))) limit 1 offset 1);

employee3 integer := (select e\_id from employee where employee.e\_id in

(select d\_id from department where department.hid in

(select h\_id from headquarters where headquarters.l\_id in

(select l\_id from geo\_location where geo\_location.l\_id > 999 and geo\_location.country

in (select country from geo\_location where geo\_location.l\_id = (select l\_id from customer where customer.c\_id = 5))))) limit 1 offset 2);

BEGIN

INSERT INTO project\_role (e\_id, p\_id, prole\_start\_date) VALUES(employee1, new.p\_id, (SELECT CURRENT\_DATE));

INSERT INTO project\_role (e\_id, p\_id, prole\_start\_date) VALUES(employee2, new.p\_id, (SELECT CURRENT\_DATE));

INSERT INTO project\_role (e\_id, p\_id, prole\_start\_date) VALUES(employee3, new.p\_id, (SELECT CURRENT\_DATE));

--RAISE NOTICE ':D';

RETURN NEW;

END;

$$;

Create or replace trigger assignTrigger

after insert on project

FOR EACH ROW

EXECUTE function assignEmployees();

Create or replace function contractCheck() returns trigger

LANGUAGE plpgsql

AS $$

BEGIN

IF ((SELECT CURRENT\_DATE) != new.contract\_start) THEN RAISE EXCEPTION 'Contract is invalid, Start date is not today';

ELSIF new.contract\_type = 'Temporary' and ((SELECT CURRENT\_DATE) + interval '2' year != new.contract\_end or new.contract\_end IS NULL) THEN RAISE EXCEPTION 'Contract is invalid, Contract should end 2 years from now';

ELSIF new.contract\_type != 'Temporary' and new.contract\_end IS NOT NULL THEN RAISE EXCEPTION 'Contract is invalid, Contract should not have end';

END IF;

RETURN NEW;

END;

$$;

Create or replace trigger contractTrigger

before update of contract\_type on employee

FOR EACH ROW

EXECUTE function contractCheck();

Create or replace function groupCheck() returns trigger

LANGUAGE plpgsql

AS $$

BEGIN

IF 'HR secretary' = (SELECT title from job\_title where job\_title.j\_id = new.j\_id) THEN INSERT INTO employee\_user\_group(e\_id, u\_id, eug\_join\_date) values(new.e\_id, 6, (SELECT CURRENT\_DATE)); -- u\_id 6 = HR group

ELSIF (SELECT title from job\_title where job\_title.j\_id = new.j\_id) like '%admin%' THEN INSERT INTO employee\_user\_group(e\_id, u\_id, eug\_join\_date) values(new.e\_id, 3, (SELECT CURRENT\_DATE)); -- u\_id 3 = admin group

ELSE INSERT INTO employee\_user\_group(e\_id, u\_id, eug\_join\_date) values(new.e\_id, 9, (SELECT CURRENT\_DATE)); -- u\_id 9 = employee group

END IF;

RETURN NEW;

END;

$$;

Create or replace trigger groupTrigger

after insert on employee

FOR EACH ROW

EXECUTE function groupCheck();

## Procedures

Made procedures:

* salaryBase()
* temporaryIncrease()
* percentSalaryIncrease(percentValue numeric, maximumValue numeric)
* correctSalary()

SalaryBase sets all employees salary to the salary given by their job title.

TemporaryIncrease gives all employees with temporary contract 3 months more contract time

percentSalaryIncrease takes in any numeric values. percentValue proceeds with integers being per cents like 20 = 20% and so on. Can use negative value. It also takes maximum value and if the original value was higher than the given value then the value wasn’t increased. percentSalaryIncrease increased current salary value by given per cent value. If given null as salary increase nothing happens.

correctSalary first call salaryBase procedure to give them their salary a base value and then gives them additional salary for each benefit salary marked in the skills they have.

Done by running:

Create or replace procedure salaryBase()

LANGUAGE plpgsql

AS $$

BEGIN

UPDATE employee SET salary = (SELECT base\_salary from job\_title where job\_title.j\_id = employee.j\_id);

END;

$$;

Create or replace procedure temporaryIncrease()

LANGUAGE plpgsql

AS $$

BEGIN

UPDATE employee SET contract\_end = employee.contract\_end + interval '3' month where employee.contract\_type = 'Temporary';

END;

$$;

Create or replace procedure percentSalaryIncrease(percentValue numeric, maxiumValue numeric)

LANGUAGE plpgsql

AS $$

BEGIN

IF (maxiumValue IS NULL or maxiumValue = 0) and percentValue IS NOT NULL THEN UPDATE employee SET salary = salary\*(1+(percentValue/100));

ELSIF percentValue IS NOT NULL THEN UPDATE employee SET salary = salary\*(1+(percentValue/100)) where employee.salary < maxiumValue;

END IF;

END;

$$;

Create or replace procedure correctSalary()

LANGUAGE plpgsql

AS $$

BEGIN

CALL salaryBase();

UPDATE employee SET salary = salary + (SELECT SUM(salary\_benefit\_value) FROM skills where skills.s\_id in (SELECT s\_id from employee\_skills where employee\_skills.e\_id = employee.e\_id ));

END;

$$;

## Functions

Made function:

* getProjects(givenDate date)

getProjects takes a date and returns in a table all projects which end date is later than given date and start date is earlier than given date. In the table are information about the project and the customer information.

Done by running:

Create or replace function getProjects(givenDate date) returns table(project\_id int, name varchar, budget numeric, commission\_percentage numeric, start\_date date, end\_date date, c\_id int, customer\_id int, c\_name varchar, c\_type varchar, phone varchar, email varchar, l\_id int)

LANGUAGE plpgsql

AS $$

BEGIN

return query (SELECT \* from project inner join customer on project.c\_id = customer.c\_id where p\_end\_date > givenDate and p\_start\_date <= givenDate);

END;

$$;

## Constraints

Added following constraints:

* *customer*: *email* not null
* *project*: *p\_start\_date* not null
* *employee*: *salary* > 1000

Done by running:

ALTER TABLE customer

ALTER COLUMN email SET NOT NULL;

ALTER TABLE project

ALTER COLUMN p\_start\_date SET NOT NULL;

-- change all salaries from 0 to 1001 to not get errors

UPDATE employee

SET salary = 1001

WHERE salary = 0;

ALTER TABLE employee

ADD CONSTRAINT salary\_check CHECK (salary > 1000);

## Partitions

PostgreSQL does not support partitioning an existing table so instead the tables were copied except the copies were partitioned by the corresponding column and then renamed.

The following partitions were made this way:

* *customer* by *c\_id*
* *project* by *p\_id*

Each table has 3 partitions and a default partition.

After the tables have been renamed, all foreign keys referring to the original tables are updated to refer to the new ones. (project should have a fkey c\_id but in the backup given, the constraint does not exist so it was created)

Done by running:

CREATE TABLE new\_customer (LIKE customer INCLUDING ALL)

PARTiTION BY RANGE (c\_id);

CREATE TABLE customer\_1 PARTITION OF new\_customer

FOR VALUES FROM (1) TO (300);

CREATE TABLE customer\_2 PARTITION OF new\_customer

FOR VALUES FROM (300) TO (600);

CREATE TABLE customer\_3 PARTITION OF new\_customer

FOR VALUES FROM (600) TO (900);

CREATE TABLE customer\_default PARTITION OF new\_customer

DEFAULT;

INSERT INTO new\_customer SELECT \* FROM customer;

BEGIN;

ALTER TABLE customer RENAME TO old\_customer;

ALTER TABLE new\_customer RENAME TO customer;

COMMIT;

--ALTER TABLE project add constraint project\_c\_id\_fkey FOREIGN KEY (c\_id) REFERENCES customer(c\_id);

CREATE TABLE new\_project (LIKE project INCLUDING ALL)

PARTiTION BY RANGE (p\_id);

CREATE TABLE project\_1 PARTITION OF new\_project

FOR VALUES FROM (1) TO (300);

CREATE TABLE project\_2 PARTITION OF new\_project

FOR VALUES FROM (300) TO (600);

CREATE TABLE project\_3 PARTITION OF new\_project

FOR VALUES FROM (600) TO (900);

CREATE TABLE project\_default PARTITION OF new\_project

DEFAULT;

INSERT INTO new\_project SELECT \* FROM project;

BEGIN;

ALTER TABLE project RENAME TO old\_project;

ALTER TABLE new\_project RENAME TO project;

COMMIT;

ALTER TABLE project\_role drop constraint project\_role\_p\_id\_fkey;

ALTER TABLE project\_role add constraint project\_role\_p\_id\_fkey FOREIGN KEY (p\_id) REFERENCES project(p\_id);

## Views

Created the following views:

* employees\_on\_project

lists all employees on a project

* employees\_in\_department\_in\_hq

lists all employees by department and headquarter

* employees\_by\_skill

lists all employees by their skills

* customers\_by\_location

lists all customers by their location

* employees\_by\_group

lists all employees by the group(s) they belong to

* employees\_by\_title

lists all employees by the job title(s) they have

Done by running:

CREATE VIEW employees\_on\_project AS

SELECT p.project\_name AS project\_name, e.emp\_name AS employee\_name,

p.p\_id AS p\_id, e.e\_id AS e\_id

FROM project p, project\_role r, employee e

WHERE r.e\_id = e.e\_id AND r.p\_id = p.p\_id

ORDER BY p.p\_id, employee\_name;

CREATE VIEW employees\_in\_department\_in\_hq AS

SELECT h.hq\_name, d.dep\_name, e.emp\_name, j.title, e.salary, h.h\_id AS h\_id, d.d\_id AS d\_id, e.e\_id AS e\_id

FROM headquarters h, department d, employee e, job\_title j

WHERE e.d\_id = d.d\_id AND d.hid = h.h\_id AND e.j\_id = j.j\_id

ORDER BY h.hq\_name, d.dep\_name, e.emp\_name, j.title;

CREATE VIEW employees\_by\_skill AS

SELECT e.emp\_name, s.skill, j.title, e.e\_id AS e\_id, s.s\_id AS s\_id

FROM employee AS e, employee\_skills AS es, skills AS s, job\_title AS j

WHERE e.e\_id = es.e\_id AND s.s\_id = es.s\_id AND e.j\_id = j.j\_id

ORDER BY e.emp\_name, s.skill;

CREATE VIEW customers\_by\_location AS

SELECT c.c\_name AS customer, g.country AS country, g.city AS city, g.street AS street, c.c\_id AS c\_id, g.l\_id AS l\_id

FROM geo\_location AS g, customer AS c

WHERE c.l\_id = g.l\_id

ORDER BY c.c\_name, g.country, g.city, g.street;

CREATE VIEW employees\_by\_group AS

SELECT e.emp\_name AS employee, g.group\_title AS group, g.group\_rights AS rights, e.e\_id AS e\_id, g.u\_id AS u\_id

FROM employee AS e, employee\_user\_group AS eg, user\_group AS g

WHERE e.e\_id = eg.e\_id AND g.u\_id = eg.u\_id

ORDER BY e.emp\_name, g.group\_title, g.group\_rights;

CREATE VIEW employees\_by\_title AS

SELECT e.emp\_name AS employee, j.title AS title, e.contract\_type AS contract\_type, e.salary AS salary,

e.e\_id AS e\_id, j.j\_id AS j\_id

FROM employee e, job\_title j

WHERE e.j\_id = j.j\_id

ORDER BY j.j\_id, e.j\_id;

## Zip\_code

Added a *zip\_code* column to *geo\_location*. It holds an integer and has no constraints.

Done by running:

ALTER TABLE geo\_location ADD COLUMN zip\_code INTEGER;