CT60A7650 Database Systems Management

Project – Database management

Lappeenranta-Lahti University of Technology LUT
Software Engineering

Database Systems Management
Spring 2024

Natunen Aleksi, Nissinen Jesse

TABLE OF CONTENTS

1 Definition

The need in our project was to upkeep/manage a database, where we had almost zero knowledge. In the project we added different triggers, functions, procedures, views, etc.. By doing these we made the management of the database more easier or automated so for future use the database is more efficient. All these changes were made with precise briefing in our assignment.

2 Database implementation

During implementation, the following changes were created to the database:

- **Views**

We created 6 different views to help provide a better understanding of relations between the data.

The employee_job_titles view shows all employees and their job title.

```
CREATE VIEW employee_job_titles AS
SELECT employee.emp_name AS "Employee", job_title.title AS "Job Title"
FROM employee
JOIN job_title ON employee.j_id = job_title.j_id;
```

The department_personnel view shows which department each employee belongs to.

```
CREATE VIEW department_personnel AS
SELECT employee.emp_name AS "Employee", department.dep_name AS "Department"
FROM employee
JOIN department ON employee.d_id = department.d_id;
```

The project_customers view links all projects to their customers.

```
CREATE VIEW project_customers AS
SELECT customer.c_name AS "Customer", project.project_name AS "Project"
FROM customer
JOIN project ON customer.c_id = project.c_id;
```

The customer_locations view shows the country and city of each customer.

```
CREATE VIEW customer_locations AS
SELECT customer.c_name AS "Customer", geo_location.country AS "Country",
geo_location.city AS "City"
FROM customer
JOIN geo_location ON customer.l_id = geo_location.l_id;
```

The employee_teams view shows which group each of the employees belongs to.

```
CREATE VIEW employee_teams AS
SELECT employee.emp_name AS "Employee", user_group.group_title AS "Team"
FROM employee
JOIN employee_user_group ON employee.e_id = employee_user_group.e_id
JOIN user_group ON employee_user_group.u_id = user_group.u_id;
```

The project_employees shows which employees are working on which project.

```
CREATE VIEW project_employees AS
SELECT project.project_name AS "Project", employee.emp_name AS "Employee"
FROM employee
JOIN project_role on employee.e_id = project_role.e_id
JOIN project ON project_role.p_id = project.p_id;
```

- **Procedures**

We created 4 different procedures with unique functions.

The set_salary_base procedure sets all employees' salaries to the base level based on their job title.

```
CREATE OR REPLACE PROCEDURE set_salary_base()
LANGUAGE plpgsql AS
$$
BEGIN
	UPDATE employee SET salary = (SELECT job_title.base_salary FROM job_title WHERE
employee.j_id = job_title.j_id);
END;
$$;
```

The extend_temporary procedure adds 3 months to all temporary contracts.

```
CREATE OR REPLACE PROCEDURE extend_temporary()
LANGUAGE plpgsql AS
$$
BEGIN
	UPDATE employee SET contract_end = contract_end + (3 * INTERVAL '1 month')
WHERE contract_type = 'Temporary';
END;
$$;
```

The increase_salaries procedure increases salaries by a percentage based on the given percentage.

```
CREATE OR REPLACE PROCEDURE increase_salaries(increase NUMERIC)
LANGUAGE plpgsql AS
$$
BEGIN
	UPDATE employee SET salary = salary * (1 + increase);
END;
$$;
```

The calculate_salary_skill_bonus calculates the correct salary based on the acquired skills.

```
CREATE OR REPLACE PROCEDURE calculate_salary_skill_bonus()
LANGUAGE plpgsql AS
$$
DECLARE
	sum_of_salary_benefits NUMERIC = 0;
	emp_id INT;
BEGIN
	FOR emp_id IN SELECT e_id FROM employee LOOP
		SELECT SUM(salary_benefit_value) INTO sum_of_salary_benefits FROM
skills
			WHERE salary_benefit = true AND s_id IN (SELECT
employee_skills.s_id FROM employee_skills WHERE employee_skills.e_id = emp_id);
		IF sum_of_salary_benefits IS NOT NULL
		THEN
			UPDATE employee SET salary = salary + sum_of_salary_benefits
WHERE emp_id = e_id;
		END IF;
	END LOOP;
END;
$$;
```

The procedures were run once with the following queries

```
CALL set_salary_base();
CALL extend_temporary();
CALL increase_salaries(0.5);
CALL calculate_salary_skill_bonus();
```

- **Roles**

We created 4 different roles that have login access to the database.

The admin role has all administrative rights.

```
CREATE ROLE admin LOGIN;
ALTER ROLE admin WITH SUPERUSER CREATEDB CREATEROLE BYPASSRLS REPLICATION;
GRANT ALL ON ALL TABLES IN SCHEMA "public" TO admin;
```

The employee role has rights to read all of the information but no write privileges.

```
CREATE ROLE employee LOGIN;
GRANT SELECT ON ALL TABLES IN SCHEMA "public" TO employee;
```

The trainee role has rights to read the project, customer, geo_location and project_role tables and reading privileges to the employee id, name and email columns of the employee table.

```
CREATE ROLE trainee LOGIN;
GRANT SELECT ON project, customer, geo_location, project_role TO trainee;
GRANT SELECT (e_id, emp_name, email) ON employee TO trainee;
```

The views_only role has rights to read all of the views of the database.

```
CREATE ROLE views_only LOGIN;
GRANT SELECT ON employee_job_titles, department_personnel, project_customers,
customer_locations, employee_teams, project_employees TO views_only;
```

- **Triggers**

We created triggers to trigger different events for the database, to automate the database when inserting or updating on different tables:

The 1st trigger we created is a trigger for before inserting a new skill, to make sure that the same skill does not already exist.

```
CREATE OR REPLACE FUNCTION checkifskillalready()
RETURNS TRIGGER
LANGUAGE plpgsql AS
$$
DECLARE
        employee_skill_ID INT;
        skill_ID INT;
        skill_amount INT;
BEGIN

        employee_skill_ID := (NEW.e_id);
        skill_ID := (NEW.s_id);

        skill_amount := (
        SELECT COUNT(*)
        FROM
                employee_skills ES
        JOIN
                employee E
                ON ES.e_id = E.e_id
        WHERE
                ES.s_id = skill_ID AND E.e_id = employee_skill_ID);

        IF (skill_amount > 0) THEN
                RAISE EXCEPTION 'Employee already has this skill.';
        END IF;
        RETURN NEW;

END;
$$;
CREATE OR REPLACE TRIGGER skill_already_on_employee BEFORE INSERT OR UPDATE ON employee_skills
FOR EACH ROW EXECUTE PROCEDURE checkifskillalready();
```

The 2nd trigger we created is a trigger for after inserting a new project, checking the customer country and selecting three employees from that country to start working with the project (i.e. create new project roles).

```
CREATE OR REPLACE FUNCTION new_project_means_new_team()
RETURNS TRIGGER
LANGUAGE plpgsql AS
$$
DECLARE
	project_ID INT;
	employee_ID INT;
	geo_location_ID INT;
	customer_department INT;
	customer_ID INT;
	customer_headquarter INT;
	customer_geo_location INT;
	employees INT[];
	var INT;

BEGIN
	project_ID := (NEW.p_id);
	customer_ID := (NEW.c_id);

	customer_geo_location := (
		SELECT CU.l_id
		FROM
			customer CU
		WHERE
			CU.c_id = customer_ID
	);
	customer_headquarter := (
		SELECT HQ.h_id
		FROM
			headquarters HQ
		WHERE
			HQ.l_id = customer_geo_location
	);
	customer_department := (
		SELECT DE.d_id
		FROM
			department DE
		WHERE
			DE.hid = customer_headquarter
		LIMIT (1)
	);
	SELECT ARRAY (
		SELECT E.e_id
		FROM
			employee E
		WHERE
			E.d_id = customer_department
		LIMIT (3)
	) INTO employees;

	FOREACH var IN ARRAY employees LOOP
		INSERT INTO project_role (e_id, p_id, prole_start_date) VALUES (var, project_ID,
current_date);
	END LOOP;

	RETURN NEW;
END;
$$;
CREATE OR REPLACE TRIGGER project_new_team_adder AFTER INSERT OR UPDATE ON project
FOR EACH ROW EXECUTE PROCEDURE new_project_means_new_team();
```

The 3rd trigger we created is a trigger for before updating the employee contract type, to make sure that the contract start date is also set to the current date and end date is either 2 years after the start date contract is of Temporary type, NULL otherwise. (Temporary contract in Finnish is "määräaikainen". It's a contract that has an end date specified).

```
CREATE OR REPLACE FUNCTION checkforthecontract()
RETURNS TRIGGER
LANGUAGE plpgsql AS
$$
BEGIN
        IF NEW.contract_type <> OLD.contract_type THEN
                IF NEW.contract_type ILIKE 'määräaikainen' OR NEW.contract_type ILIKE 'temporary'
THEN
                        NEW.contract_start = CURRENT_DATE;
                        NEW.contract_end = CURRENT_DATE + INTERVAL '2 years';
                ELSE
                        NEW.contract_start = CURRENT_DATE;
                        NEW.contract_end = NULL;
                END IF;
        END IF;
                RETURN NEW;
END;
$$;
CREATE OR REPLACE TRIGGER contract_type_trigger BEFORE INSERT OR UPDATE ON employee
FOR EACH ROW EXECUTE PROCEDURE checkforthecontract();
```

The 4th trigger we created is a trigger, that after inserting an employee, if the employee's job title is HR secretary, add them to the HR user group, if the employee's job title is any of the admin related, add them to the Administration group and everyone else is added to the employee group

```
CREATE OR REPLACE FUNCTION employee_user_group()
RETURNS TRIGGER
LANGUAGE plpgsql AS
$$
BEGIN
    IF NEW.j_id = 12 THEN
        IF NOT EXISTS (SELECT 1 FROM employee_user_group WHERE e_id = NEW.e_id AND u_id = 6) THEN
            INSERT INTO employee_user_group(e_id, u_id) VALUES (NEW.e_id, 6);
        END IF;
    ELSIF NEW.j_id = 5 OR NEW.j_id = 6 OR NEW.j_id = 7 THEN
        IF NOT EXISTS (SELECT 1 FROM employee_user_group WHERE e_id = NEW.e_id AND u_id = 3) THEN
            INSERT INTO employee_user_group(e_id, u_id) VALUES (NEW.e_id, 3);
        END IF;
    ELSE
        IF NOT EXISTS (SELECT 1 FROM employee_user_group WHERE e_id = NEW.e_id AND u_id = 9) THEN
            INSERT INTO employee_user_group(e_id, u_id) VALUES (NEW.e_id, 9);
        END IF;
    END IF;

    RETURN NEW;
END;
$$;
CREATE OR REPLACE TRIGGER employee_user_group_trigger AFTER INSERT OR UPDATE ON employee
FOR EACH ROW EXECUTE PROCEDURE employee_user_group();
```

- **Partitions**

We created partitions to give different partitions of two (2) different tables. These partitions divides the tables for more accessible “child” tables:

The 1st partition divides the employee table into three (3) different child tables (partitions). One partition table for part-time, one for temporary and one for full-time employees.

```
CREATE TABLE employee_partition(
        e_id INT NOT NULL DEFAULT(nextval('employee_e_id_seq'::regclass)),
        emp_name varchar DEFAULT ('No Name.'),
        email VARCHAR,
        contract_type VARCHAR NOT NULL,
        contract_start DATE NOT NULL,
        contract_end DATE,
        salary INT DEFAULT(0),
        supervisor INT,
        d_id INT,
        j_id INT,
        FOREIGN KEY (d_id) REFERENCES department(d_id),
        FOREIGN KEY (j_id) REFERENCES job_title(j_id)
) PARTITION BY LIST(contract_type);

CREATE TABLE part_time_employees PARTITION OF employee_partition
        FOR VALUES IN ('Part-time');

CREATE TABLE temporary_employees PARTITION OF employee_partition
        FOR VALUES IN ('Temporary', 'Määräaikainen');

CREATE TABLE full_time_employees PARTITION OF employee_partition
        FOR VALUES IN ('Full-time');

INSERT INTO employee_partition SELECT * FROM employee;
```

The 2nd partition divides the project table into three (3) different child tables (partitions). One partition table for commission percentage between 0-10, one for commission percentage between 10-25 and one for commission percentage between 25-100.

```
CREATE TABLE project_partition(
        p_id INT NOT NULL DEFAULT(nextval('project_p_id_seq'::regclass)),
        project_name varchar,
        budget NUMERIC,
        commission_percentage NUMERIC,
        p_start_date DATE,
        p_end_date DATE,
        c_id INT,
        FOREIGN KEY (c_id) REFERENCES customer(c_id)
) PARTITION BY range(commission_percentage);

CREATE TABLE low_percentage PARTITION OF project_partition
    FOR VALUES FROM (0) TO (11);

CREATE TABLE medium_percentage PARTITION OF project_partition
    FOR VALUES FROM (11) TO (25);

CREATE TABLE high_percentage PARTITION OF project_partition
    FOR VALUES FROM (25) TO (100);

ALTER TABLE low_percentage ADD CONSTRAINT low_percentage_check
    CHECK (commission_percentage >= 0 AND commission_percentage <= 10);

ALTER TABLE medium_percentage ADD CONSTRAINT medium_percentage_check
    CHECK (commission_percentage >= 11 AND commission_percentage <= 25);

ALTER TABLE high_percentage ADD CONSTRAINT high_percentage_check
    CHECK (commission_percentage >= 25 AND commission_percentage <= 100);

INSERT INTO project_partition SELECT * FROM project;
```

- Function

We created a function to easen the search for a project that is going on during the time that is searched for. Function can be run with example:

```
SELECT * FROM get_running_projects('2012-09-08');
```

```
CREATE OR REPLACE FUNCTION get_running_projects(
        shipment_date DATE
)
RETURNS TABLE(
        "Project ID" INT,
        "Project Name" VARCHAR,
        "Budget" NUMERIC,
        "Commission Percentage" NUMERIC,
        "Start Date" DATE,
        "End Date" DATE,
        "Customer ID" INT,
        "Customer Name" VARCHAR,
        "Customer Type" VARCHAR,
        "Customer Phone" VARCHAR,
        "Customer Email" VARCHAR
)
LANGUAGE plpgsql AS
$$
BEGIN
        RETURN QUERY
                SELECT
                        p.p_id,
                        p.project_name ,
                        p.budget,
                        p.commission_percentage ,
                        p.p_start_date ,
                        p.p_end_date ,
                        c.c_id ,
                        c.c_name ,
                        c.c_type ,
                        c.phone ,
                        c.email
                FROM
                        project p
                JOIN
                        customer c
                        ON p.c_id = c.c_id
                WHERE
                        shipment_date BETWEEN p.p_start_date AND p.p_end_date;
END;
$$;
```

- Miscellaneous changes

We made miscellaneous changes to the database to add new constraints such as email from customer cannot be null, project start date cannot be null, and salary must be more than 1000 for an employee. In addition to that we expanded the geo_location table with a zip_code column.

```
ALTER TABLE geo_location ADD zip_code VARCHAR;

ALTER TABLE customer
       ALTER COLUMN email SET NOT NULL;

ALTER TABLE project
       ALTER COLUMN p_start_date SET NOT NULL;

ALTER TABLE employee
       ADD CONSTRAINT check_salary_more_than_1000 CHECK (salary >= 1000);
```