Министерство образования и науки РФ

федеральное государственное автономное образовательное учреждение высшего образования Санкт-Петербургский национальный исследовательский университет

информационных технологий, механики и оптики

Факультет Программной инженерии и компьютерных технологий

Курсовая работа.

Этап 3.

«Реализация моделей в базе данных»

По дисциплине: Информационные системы и базы данных

Группа:

P33112

Выполнил студент:

Рябикин И. Л.

Преподаватель:

Харитонова А. Е.

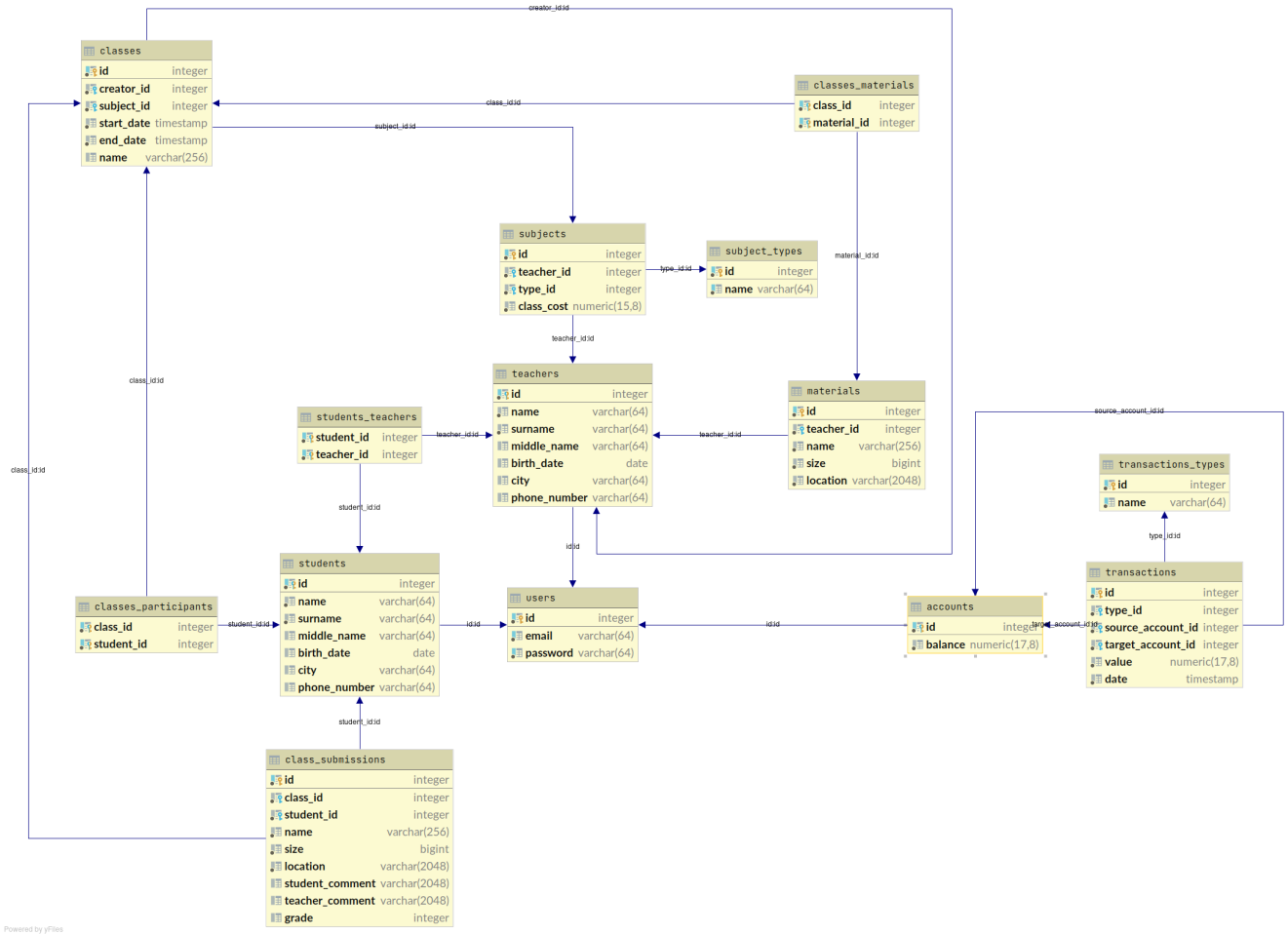
Санкт-Петербург

2020

Цель работы

Реализовать инфологическую и даталогическую модели по выбранной ранее предметной области. Также обеспечить целостность хранимых данных посредством статических проверок и триггеров, составить индексы и функции с бизнес-логикой. Выбранная раннее предметная область — онлайн-платформа для репетиторов.

Даталогическая модель



DDL скрипт

Для таблиц были заданы следующие ограничения целостности данных:

- электронная почта нового пользователя валидируется регулярным выражением;
- номер телефона нового ученика или учителя валидируется регулярным выражением (ожидается международный формат ввода номера телефона с кодом страны);
- баланс аккаунт пользователя не может быть ниже нуля, нельзя проводить транзакции с суммами меньше нуля исключительно;
- нельзя задать стоимость занятия меньше нуля включительно;
- размер физического файла материала или домашнего задания должен быть больше нуля;
- нельзя создать урок, дата окончания которого раньше даты начала;
- оценки по домашним заданиям ставятся по пятибалльной шкале.

```
CREATE TABLE IF NOT EXISTS users

(

    id       serial PRIMARY KEY,

    email    character varying(64) UNIQUE NOT NULL,

    password character varying(64)        NOT NULL,

     CHECK (email ~* '[A-Z0-9_!#$%&''*+/=?`{|}~^-]+(?:\.[A-Z0-9_!#$%&''*+/=?`{|}~^-]+)*@[A-Z0-9-]+(?:\.
[A-Z0-9-]+)*')

);


CREATE TABLE IF NOT EXISTS accounts

(

    id      integer PRIMARY KEY REFERENCES users ON DELETE CASCADE,

    balance numeric(17, 8) NOT NULL,

    CHECK (balance != 'NaN' AND

           balance > 0)

);
```

```
CREATE TABLE IF NOT EXISTS transactions_types
(
    id   serial PRIMARY KEY,
    name character varying(64) UNIQUE NOT NULL
);

CREATE TABLE IF NOT EXISTS transactions
(
    id                serial PRIMARY KEY,
    type_id           integer        NOT NULL REFERENCES transactions_types ON DELETE RESTRICT,
    source_account_id integer        NOT NULL REFERENCES accounts ON DELETE CASCADE,
    target_account_id integer        NOT NULL REFERENCES accounts ON DELETE CASCADE,
    value             numeric(17, 8) NOT NULL,
    date              timestamp      NOT NULL,
    CHECK (value ≠ 'NaN' AND
           value ⩾ 0)
);

CREATE TABLE IF NOT EXISTS students
(
    id           integer PRIMARY KEY REFERENCES users ON DELETE CASCADE,
    name         character varying(64) NOT NULL,
    surname      character varying(64) NOT NULL,
    middle_name  character varying(64),
    birth_date   date,
    city         character varying(64),
    phone_number character varying(64),
    CHECK (phone_number IS NULL OR
           (phone_number IS NOT NULL AND
            phone_number ~ '\+(?:[0-9] ?){6,14}[0-9]'))
```

```
);

CREATE TABLE IF NOT EXISTS teachers
(
    id           integer PRIMARY KEY REFERENCES users ON DELETE CASCADE,
    name         character varying(64) NOT NULL,
    surname      character varying(64) NOT NULL,
    middle_name  character varying(64),
    birth_date   date,
    city         character varying(64),
    phone_number character varying(64),
    CHECK (phone_number IS NULL OR
           (phone_number IS NOT NULL AND
            phone_number ~ '\+(?:[0-9] ?){6,14}[0-9]'))
);

CREATE TABLE IF NOT EXISTS students_teachers
(
    student_id integer NOT NULL REFERENCES students ON DELETE CASCADE,
    teacher_id integer NOT NULL REFERENCES teachers ON DELETE CASCADE,
    PRIMARY KEY (student_id, teacher_id)
);

CREATE TABLE IF NOT EXISTS subject_types
(
    id   serial PRIMARY KEY,
    name character varying(64) UNIQUE NOT NULL
);

CREATE TABLE IF NOT EXISTS subjects
```

```
(
    id         serial PRIMARY KEY,
    teacher_id integer        NOT NULL REFERENCES teachers ON DELETE CASCADE,
    type_id    integer        NOT NULL REFERENCES subject_types ON DELETE RESTRICT,
    class_cost numeric(15, 8) NOT NULL,
    CHECK (class_cost ≠ 'NaN' AND
           class_cost ⩾ 0)
);

CREATE TABLE IF NOT EXISTS materials
(
    id         serial PRIMARY KEY,
    teacher_id integer                NOT NULL REFERENCES teachers ON DELETE CASCADE,
    name       character varying(256)  NOT NULL,
    size       bigint                 NOT NULL,
    location   character varying(2048) NOT NULL,
    CHECK (size > 0)
);

CREATE TABLE IF NOT EXISTS classes
(
    id         serial PRIMARY KEY,
    creator_id integer   NOT NULL REFERENCES teachers ON DELETE RESTRICT,
    subject_id integer   NOT NULL REFERENCES subjects ON DELETE RESTRICT,
    start_date timestamp NOT NULL,
    end_date   timestamp NOT NULL,
    name       character varying(256),
    CHECK (end_date > start_date)
);
```

```
CREATE TABLE IF NOT EXISTS classes_materials

(

    class_id    integer NOT NULL REFERENCES classes ON DELETE RESTRICT,

    material_id integer NOT NULL REFERENCES materials ON DELETE RESTRICT,

    PRIMARY KEY (class_id, material_id)

);


CREATE TABLE IF NOT EXISTS classes_participants

(

    class_id   integer NOT NULL REFERENCES classes ON DELETE RESTRICT,

    student_id integer NOT NULL REFERENCES students ON DELETE RESTRICT,

    PRIMARY KEY (class_id, student_id)

);


CREATE TABLE IF NOT EXISTS class_submissions

(

    id              serial PRIMARY KEY,

    class_id        integer                 NOT NULL REFERENCES classes ON DELETE RESTRICT,

    student_id      integer                 NOT NULL REFERENCES students ON DELETE RESTRICT,

    name            character varying(256)  NOT NULL,

    size            bigint                  NOT NULL,

    location        character varying(2048) NOT NULL,

    student_comment varchar(2048),

    teacher_comment varchar(2048),

    grade           integer,

    CHECK (grade IS NULL OR

           (grade ⩾ 0 AND

            grade < 6))

);
```

Созданные индексы

Целесообразность использования тех или иных индексов всплывает в процессе эксплуатации информационной системы, а также в связи с необходимостью увеличения производительности по конкретным часто используемым запросам.

На данный момент для меня очевидны пока два типа индексов: электронная почта пользователя и хэш пароля. По этим аттрибутам будут регулярно аутентифицироваться и авторизовываться пользователи в системе, поэтому можно утверждать, что индекс, ускоряющий сравнения заданной строки с одним из указанных аттрибутов положительно скажется на производительности.

```
CREATE INDEX user_email_index ON users USING hash (email);

CREATE INDEX user_password_index ON users USING hash (password);
```

Созданные функции

Нижеперечисленные функции позволяют одним вызовом создать ученика или учителя, указав лишь электронную почту, пароль, имя и фамилию, что является распространенным сценарием использования системы при регистрации.

```
CREATE OR REPLACE FUNCTION create_student(email users.email%TYPE,
                                          password users.password%TYPE,
                                          name students.name%TYPE,
                                          surname students.surname%TYPE)
    RETURNS VOID AS
$$
DECLARE
    user_id integer DEFAULT 0;
BEGIN
    RAISE DEBUG 'Function create_student() fired';
    INSERT INTO users (email, password) VALUES (email, password) RETURNING id INTO user_id;
    INSERT INTO accounts (id, balance) VALUES (user_id, 0);
    INSERT INTO students (id, name, surname) VALUES (user_id, name, surname);
END;
```

```
$$ LANGUAGE plpgsql;


CREATE OR REPLACE FUNCTION create_teacher(email users.email%TYPE,
                                          password users.password%TYPE,
                                          name students.name%TYPE,
                                          surname students.surname%TYPE)
    RETURNS VOID AS
$$
DECLARE
    user_id integer DEFAULT 0;
BEGIN
    RAISE DEBUG 'Function create_student() fired';
    INSERT INTO users (email, password) VALUES (email, password) RETURNING id INTO user_id;
    INSERT INTO accounts (id, balance) VALUES (user_id, 0);
    INSERT INTO students (id, name, surname) VALUES (user_id, name, surname);
END;
$$ LANGUAGE plpgsql;
```

Созданные триггеры

Для таблиц были созданы следующие триггеры, обеспечивающие логическую целостность данных в таблице:

- нельзя создать ученика, если учитель с таким же идентификатором уже существует в базе. Аналогичные ограничения действуют и на учителя;
- нчастниками урока могут быть только те ученики, которые являются учениками учителя, создавшего этот урок;
- новый урок можно создать только если он начинается после текущего времени, а также не накладывается поверх уже созданных других уроков учителя (подразумевается, что время указано в формате UTC без часового пояса);
- домашнюю работу к уроку могут добавить только ученики, являющиеся его участником;

- строка пароля для нового пользователя или новый пароль для уже существующего пользователя хэшируется с помощью функции Bcrypt со случайно сгенерированной «солью» автоматически после вставки (необходим модуль pgcrpyto, недоступный на сервере helios.cs.ifmo.ru. Поэтому при запуске скрипта лучше закомментировать эти триггеры и аналогичную функциональность реализовать на уровне приложения).

```
/* Insert teacher only if student with the same id doesn't exist */

CREATE OR REPLACE FUNCTION check_teacher_student_duplicate_function()

    RETURNS TRIGGER AS

$$

BEGIN

    RAISE DEBUG 'Trigger function check_teacher_student_duplicate_function() fired';

    IF EXISTS(

            SELECT 1

            FROM users

                     INNER JOIN students ON users.id = students.id

            WHERE students.id = new.id

        )

    THEN

        RAISE WARNING 'Teacher with id % already exists as student',

            new.id;

        RETURN NULL;

    ELSE

        RETURN new;

    END IF;

END;

$$ LANGUAGE plpgsql;


CREATE TRIGGER check_teacher_student_duplicate_trigger

    BEFORE INSERT

    ON teachers
```

```
    FOR EACH ROW
EXECUTE PROCEDURE check_teacher_student_duplicate_function();


/* Insert student only if teacher with the same id doesn't exist */
CREATE OR REPLACE FUNCTION check_student_teacher_duplicate_function()
    RETURNS TRIGGER AS
$$
BEGIN
    RAISE DEBUG 'Trigger function check_student_teacher_duplicate_function() fired';
    IF EXISTS(
            SELECT 1
            FROM teachers
            WHERE teachers.id = new.id
        )
    THEN
        RAISE WARNING 'Student with id % already exists as teacher',
            new.id;
        RETURN NULL;
    ELSE
        RETURN new;
    END IF;
END;
$$ LANGUAGE plpgsql;


CREATE TRIGGER check_student_teacher_duplicate_trigger
    BEFORE INSERT
    ON students
    FOR EACH ROW
EXECUTE PROCEDURE check_student_teacher_duplicate_function();
```

```
/* Insert only if creator of the class with specified id is the teacher of

   the student with specified id */

CREATE OR REPLACE FUNCTION class_participants_function()

    RETURNS TRIGGER AS

$$

BEGIN

    RAISE DEBUG 'Trigger function class_participants_function() fired';

    IF EXISTS(

            SELECT 1

            FROM classes

                     INNER JOIN teachers ON classes.creator_id = teachers.id

                     INNER JOIN students_teachers ON teachers.id = students_teachers.teacher_id

                     INNER JOIN students ON students_teachers.student_id = students.id

            WHERE classes.id = new.class_id

              AND students.id = new.student_id

        )

    THEN

        RETURN new;

    ELSE

        RAISE WARNING 'Creator of the class with id % is not teacher of student with id %',

            new.class_id, new.student_id;

        RETURN NULL;

    END IF;

END;

$$ LANGUAGE plpgsql;


CREATE TRIGGER class_participants_trigger

    BEFORE INSERT

    ON classes_participants

    FOR EACH ROW
```

```
EXECUTE PROCEDURE class_participants_function();


/* Insert only if new class doesn't overlap other teacher's classes and starts
   after now */
CREATE OR REPLACE FUNCTION class_check_time_function()
    RETURNS TRIGGER AS
$$
DECLARE
    row classes%rowtype;
BEGIN
    RAISE DEBUG 'Trigger function class_check_time_function() fired';
    IF row.start_date < localtimestamp
    THEN
        RAISE WARNING 'Start time of new class is after now';
        RETURN NULL;
    END IF;
    FOR row IN
        SELECT *
        FROM classes
        WHERE classes.creator_id = new.creator_id
          AND classes.start_date ⩾ localtimestamp
        LOOP
            IF NOT (new.start_date > row.start_date AND new.start_date ⩾ row.end_date) OR
               (new.end_date ⩽ row.start_date AND new.end_date < row.end_date)
            THEN
                RAISE WARNING 'Class start time is not unique and overlaps class with id %', row.id;
                RETURN NULL;
            END IF;
        END LOOP;
    RETURN new;
```

```
END;
$$ LANGUAGE plpgsql;


CREATE TRIGGER class_check_time_trigger
    BEFORE INSERT
    ON classes
    FOR EACH ROW
EXECUTE PROCEDURE class_check_time_function();


/* Insert only if student with specified id is participant of
   class with specified id */
CREATE OR REPLACE FUNCTION class_submission_function()
    RETURNS TRIGGER AS
$$
BEGIN
    RAISE DEBUG 'Trigger function class_submission_function() fired';
    IF EXISTS(
            SELECT 1
            FROM classes_participants
            WHERE student_id = new.student_id
              AND class_id = new.class_id
        )
    THEN
        RETURN new;
    ELSE
        RAISE WARNING 'Student with id % is not participant of class with id %',
            new.student_id, new.class_id;
        RETURN NULL;
    END IF;
END;
```

```
$$ LANGUAGE plpgsql;


CREATE TRIGGER class_submission_trigger
    BEFORE INSERT
    ON class_submissions
    FOR EACH ROW
EXECUTE PROCEDURE class_submission_function();


/* Hash passwords using BCrypt function */
CREATE OR REPLACE FUNCTION hash_password_function()
    RETURNS TRIGGER AS
$$
BEGIN
    RAISE DEBUG 'Trigger function hash_password() fired';
    UPDATE users
    SET password = crypt(new.password, gen_salt('bf'))
    WHERE id = new.id;
END;
$$ LANGUAGE plpgsql;


CREATE TRIGGER hash_new_password_trigger
    AFTER INSERT
    ON users
    FOR EACH ROW
EXECUTE PROCEDURE hash_password_function();


CREATE TRIGGER hash_updated_password_trigger
    AFTER UPDATE OF password
    ON users
    FOR EACH ROW
```

```
EXECUTE PROCEDURE hash_password_function();
```

Вывод

В процессе выполнения данного этапа курсовой работы я реализовал модели предметной области в конкретной физической базе данных. В целом, такая реализация уже практически готова к применению в реальных системах — они, по сути, будут являться лишь интерфейсом для созданной базы данных с добавлением некоторой новой бизнес-логики и дополнительной валидации и верификации вводимых данных