Internátny ubytovací systém

Záverečná správa

Projekt z predmetu Databázy (2)

Školiteľ: Ing. Alexander Šimko PhD.

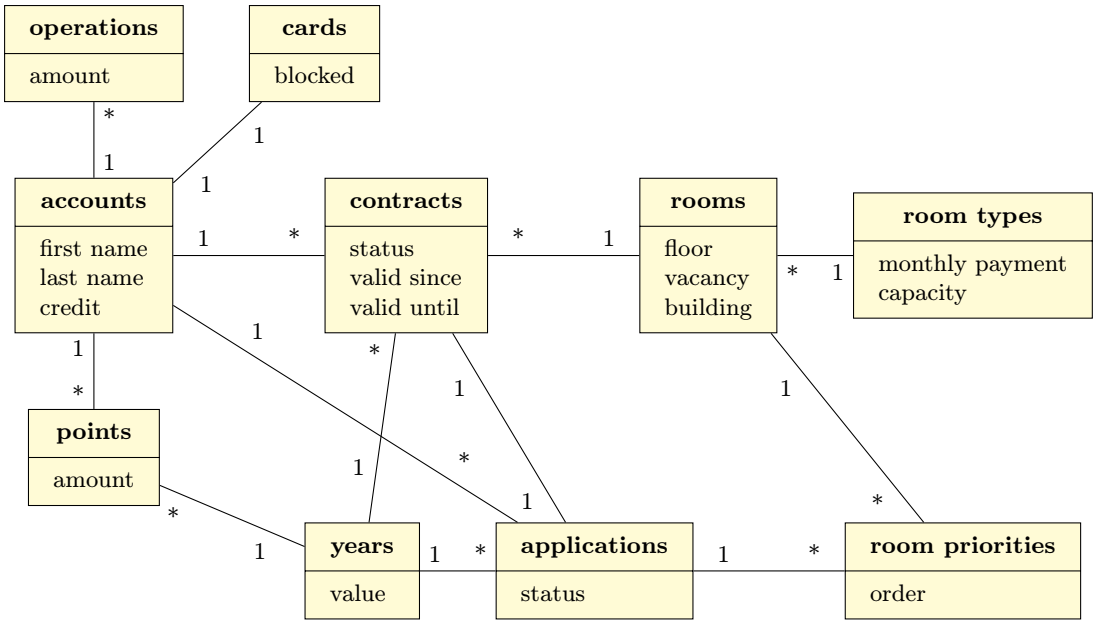
Filip Lajčin

8. mája 2020

1 O dokumente

Záverečná správa z projektu z predmetu Databázy (2) opisuje vytvorenie internátneho ubytovacieho systému, ktorý slúži na evidenciu študentov, ich žiadostí o ubytovanie, zmlúv s internátom, izieb, finančných operácií, preferencií pri výbere izieb, či ich bodové ohodnotenie na jednotlivé roky. Umožňuje spracovanie požiadaviek študentov o ubytovanie, prideľovanie izieb, navyšovanie kreditu, preubytovanie a ďalšie operácie.

2 Dátový model



Obr. 1: Entitno relačný model dát internátneho ubytovacieho systému

Účty študentov si systém eviduje v množine `accounts`. Ak pri stiahnutí mesačného poplatku za ubytovanie študent nemá na účte dostatok prostriedkov, zablokuje sa mu ISIC karta. ISIC karty sú evidované v množine `cards`.

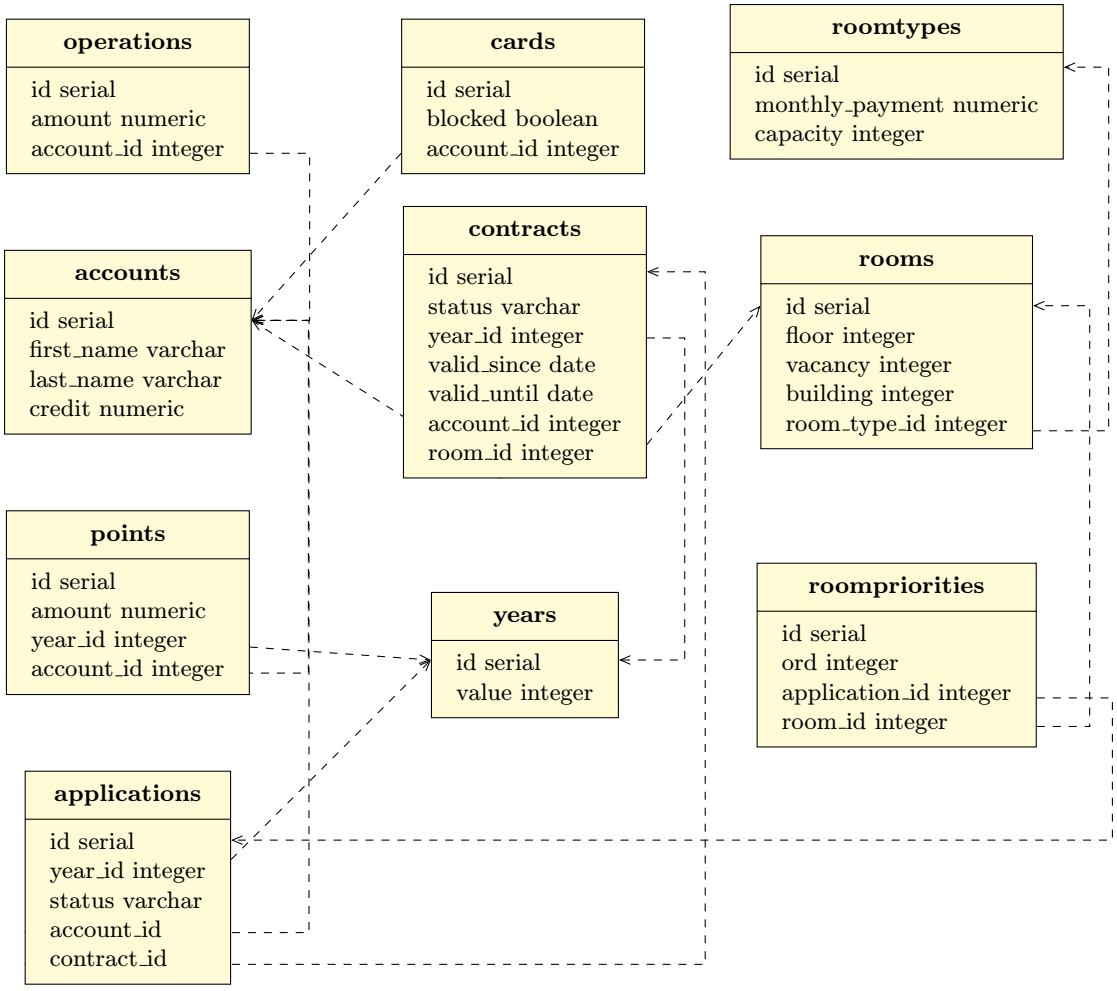
Študenti môžu podávať žiadosti o ubytovanie, ktoré sú uchovávané v množine `applications`. Ak má študent záujem o konkrétnu izbu/izby, môže k žiadosti pridať zoznam izieb očíslovaný podľa preferencie (*1 - najvyššia preferencia izby*). Tieto zoznamy sú evidované v množine `room priorities`. Izby, ktorými internát disponuje sú v množine `rooms`. Izby môžu byť iba určitého typu - všetky typy izieb sú evidované v množine `room types`, ktorá popisuje cenu a kapacitu izby.

Žiadosti pri prideľovaní izieb študentom sú zoradené podľa počtu bodov, ktorý majú študenti v tomto roku. Záznamy bodového ohodnotenia sú evidované v množine `points`. Roky, počas ktorých internátny ubytovací systém funguje sú zaznamenané v množine `years` a slúžia na jednoduché vyhľadávanie dát v databáze podľa roku (*bodové ohodnotenia študentov daného roku, žiadostí, zmlúv*).

Po úspešnom pridelení izieb sa študentom vygenerujú zmluvy, ktoré sú evidované v množine `contracts`. Všetky finančné operácie ako vklad na účet, či stiahnutie mesačného poplatku za ubytovanie sú zaznamenané v množine `operations`.

3 Relačná databáza

Obr. 2 reprezentuje relačný model dát internátneho ubytovacieho systému.



Obr. 2: Relačný model dát internátneho ubytovacieho systému

4 Organizácia kódu

Systém je naprogramovaný v jazyku Java, niektoré databázové funkcie boli vytvorené pomocou PL/pgSQL. Spojenie s databázou prebieha prostredníctom JDBC drivera. Kód je rozdelený do viacerých packagov.

`main`

Tento balík obsahuje triedu `Main`, ktorá vytvorí spojenie s databázou a spustí celú aplikáciu. Spojenie s databázou sa uloží do triedy `DbContext`, ktoré všetkým operáciam aplikácie toto spojenie dodáva.

Posledná trieda v tomto balíčku je `Initializer`, ktorá zabezpečuje spustenie `createScriptu` a `generateScript` priamo z prostredia aplikácie.

`ui`

Tento balík obsahuje triedy, ktoré zabezpečujú vypisovanie užívateľského prostredia a interakciu s užívateľom. Trieda `UserInterface` vypisuje menu všetkých sekcií a ďalej na základe vyžiadanej operácie volá metódy z tried `UIHandler` a `UIPrinter`.

Trieda `UIHandler` slúži na interakciu s užívateľom a následné spúšťanie zvolených operácií. Takisto odchytáva výnimky a umožňuje používateľovi niektoré operácie zopakovať, ak sa nepodarili uskutočniť.

Trieda `UIPrinter` sa stará o operácie, ktoré majú za úlohu vypísať informácie z databázy - výpis používateľov, žiadostí o ubytovanie, výpis kreditu a podobne. Pri väčších výpisoch podporuje aj stránkovanie ovládané užívateľom.

`rdg`

Balík `rdg` je organizovaný podľa vzoru *Row Data Gateway*. To znamená, že pre každú tabuľku sú vytvorené dve triedy - jedna trieda reprezentuje jeden riadok tabuľky a obsahuje členské premenné, ktoré označujú stĺpce tabuľky. Táto trieda zdieľa s tabuľkou rovnaký názov (*okrem jednej triedy - `PointInfo`, ktorej názov som zmenil z dôvodu, že trieda Points mi v Jave znela krkolomnejšie*).

Druhá trieda pre tabuľku bola *Finder*, ktorá dedí z abstraktnej triedy `Finder`. Tá obsahuje metódy na vrátenie jednej inštancie/listu inštancií danej *Gateway* triedy.

`domainOperations`

Balík `domainOperations` obsahuje triedy slúžiace na vykonávanie doménových operácií. Kód je vytvorený podľa vzoru *Transaction Script*.

Navyšovanie kreditu na účte sa realizuje v triede `CreditDeposit`, pridelenie izieb študentom (vykonanie ubytovacieho kola) zabezpečuje trieda `RoomAssignment`, podpísanie zmluvy s pridelenou izbou robí trieda `Accommodate`, preubytovanie vykonáva `Reaccommodate` a nakoniec sťahovanie mesačného poplatku za ubytovanie robí `MonthlyPayment`.

5 Optimalizácie pre veľké dáta

Optimalizácia triedy Room Assignment

Operácia prechádza veľkým počtom dát a pôvodný kód v Jave, ktorý prechádzal cyklom všetky žiadosti o ubytovanie a v každej iterácií vykonával updaty viacerých riadkov v databáze, som nahradil funkciou v databázovom systéme vytvorenou jazykom PL/pgSQL. Pôvodný kód:

```
int result = 0;
       List<RoomAssignmentJoinedInfo> contracts =
           contractsOrderedByPoints(pointThreshold);

       for (RoomAssignmentJoinedInfo c : contracts) {

           RoomPriority highestRoomPriorityWithFreeRoom =
                  RoomPriorityFinder.getInstance()
                         .findHighestRoomPriorityWithFreeRoom(c.getAppId());

           Room assignedRoom = null;
           if (highestRoomPriorityWithFreeRoom == null)
               assignedRoom = RoomFinder.getInstance().findRandomFree();
           else
               assignedRoom =
                   RoomFinder.getInstance().findById(highestRoomPriorityWithFreeRoom.getRoomId());

           if (assignedRoom == null) {
               NotEnoughRoomsException e = new NotEnoughRoomsException();
               e.setAssignedRooms(result);
               throw e;
           }

           assignedRoom.setVacancy(assignedRoom.getVacancy() - 1);
           c.getC().setRoomId(assignedRoom.getId());
           assignedRoom.update();
           c.getC().update();
           result++;
       }
       return result;
```

Funkcia room_assignment(point_threshold integer):

```
CREATE OR REPLACE FUNCTION room_assignment(point_threshold integer) RETURNS INTEGER AS
$$
DECLARE
    c contracts;
    a applications;
    rp room_priorities;
    r rooms;
    y years;
    numberOfRows integer;
BEGIN
    numberOfRows := 0;
    SELECT * INTO y FROM years WHERE years.id = (SELECT max(id) FROM years);
    FOR c IN SELECT contracts.id, contracts.status, contracts.year_id,
        contracts.valid_since, contracts.valid_until, contracts.account_id,
        contracts.room_id
            FROM contracts
                    JOIN points ON contracts.account_id = points.account_id AND
                        contracts.year_id = points.year_id
            WHERE contracts.status IS NULL AND contracts.year_id = y.id AND
                points.amount >= point_threshold
            ORDER BY points.amount DESC
        LOOP
           SELECT * INTO STRICT a FROM applications WHERE applications.contract_id =
               c.id;
           for rp in SELECT * FROM room_priorities WHERE
```

```
                room_priorities.application_id = a.id ORDER BY ord ASC
                LOOP
                    SELECT * INTO STRICT r FROM rooms WHERE rooms.id = rp.room_id;
                    IF r.vacancy != 0 THEN
                        r.vacancy := r.vacancy - 1;
                        c.room_id := r.id;

                        UPDATE rooms SET vacancy = vacancy - 1 WHERE id = r.id;
                        UPDATE contracts SET room_id = r.id WHERE id = c.id;
                        numberOfRows := numberOfRows +1;
                        EXIT;
                    end if;
                end loop;

            IF c.room_id IS NOT NULL THEN
                CONTINUE;
            end if;
            IF c.room_id IS NULL THEN
                SELECT * INTO r FROM rooms WHERE vacancy != 0 LIMIT 1;
                IF NOT FOUND THEN
                    RAISE EXCEPTION 'No free room to assign';
                end if;
                IF FOUND THEN
                    UPDATE rooms SET vacancy = vacancy - 1 WHERE id = r.id;
                    UPDATE contracts SET room_id = r.id WHERE id = c.id;
                    numberOfRows := numberOfRows +1;
                end if;

            end if;
        end loop;
    RETURN numberOfRows;
end;
$$ LANGUAGE plpgsql;
```

Rozdiel v rýchlosti vykonania operácie zobrazuje tabuľka 1. Pri Java kóde bol čas odmeraný pomocou `System.currentTimeMillis` a v prípade funkcie `room_assignment` bol použitý príkaz `EXPLAIN ANALYZE`.

veľkosť tabuľky `accounts` (v riadkoch)	čas pred optimalizáciou (v ms)	čas po optimalizácii (v ms)
1000	2515	94
10 000	25 714	2661
50 000	133 953	61 694
100 000	353 841	240 029

Tabuľka 1: Porovnanie rýchlosti vykonania operácie pridelenia izieb pred a po optimalizácií podľa veľkosti tabuľky `accounts`

Optimalizácia štatistiky priemernej dennej obsadenosti rôznych kapacít izieb

Cieľom operácie je vypočítať, koľko daná kapacita bola v minulosti priemerne obsadená.

Pôvodne som postupoval tak, že som každý dátum, ktorý je zaznamenávaný v systéme, zaevidoval do pomocnej tabuľky dates, ktorú som si vytvoril. Následne som prechádzal všetkými dátumami a konštruoval som si priemernú dennú obsadenosť pre daný dátum. Nakoniec, som zo všetkých týchto dát urobil priemer.

Vykonávanie operácie takýmto postupom sa ukázalo ako veľmi náročné, keďže pre 15 rokov musí štatistika vykonštruovať tabuľku obsadenosti izieb pre približne 5 500 dní.

Pôvodný kód:

```
create or replace function rooms_at_certain_date(d date)
returns table(room_id integer, vacancy bigint, capacity integer)
 language sql as
    $$
    SELECT rooms.id,
           capacity - (SELECT count(*)
                       FROM contracts
                       WHERE room_id = rooms.id
                                 AND (valid_since <= d
                                 AND valid_until >= d)
                       GROUP BY room_id),
           capacity
    FROM rooms
    JOIN room_types rt on rooms.room_type_id = rt.id;
    $$;

DROP TABLE IF EXISTS dates CASCADE;
CREATE TABLE dates (datum date);

INSERT INTO dates
SELECT make_date(
          (SELECT min(value) FROM years), 9, 1)
           + make_interval(0, 0, 0, seq.i) AS datum
FROM generate_series(0,
      (SELECT max(valid_until) FROM contracts)
       - (SELECT min(valid_since) FROM contracts)) AS seq(i);

SELECT capacity,
      avg(result)
FROM dates
CROSS JOIN LATERAL (SELECT capacity,
                                    total_occupied::float / (number_of_rooms *
                                        capacity) * capacity AS result
                                    FROM (SELECT count(*) as number_of_rooms,
                                                                sum(capacity - vacancy)
                                                                    as total_occupied,
                                                                capacity as capacity
                                                FROM rooms_at_certain_date(dates.datum)
                                                GROUP BY capacity) AS calculation)
                                     AS result_at_certain_day
GROUP BY capacity;
```

Kód som úplne prekopal, zmenil som aj spôsob výpočtu.

Postupoval som tak, že v tabuľke `contracts` som prechádzal všetkými zmluvami a pre každú izbu som spočítaval, koľko dní bola zazmluvnená. Nakoniec som pre každú izbu vypočítal priemernú dennú obsadenosť vzorcom $n/c * n_0$, kde n je počet všetkých zazmluvnených dní izby, c je kapacita izby a n_0 je počet všetkých dní v evidencii systému a z týchto dát som urobil priemer podľa kapacity izby.

Výsledný kód:

```
SELECT capacity,
          sum(average_daily_occupancy_of_room)::float /
              (SELECT count(*)
               FROM rooms
               WHERE room_type_id IN
                      (SELECT id
                      FROM room_types rt2
                      WHERE rt2.capacity = rt3.capacity))) * rt3.capacity::float as
                          result
FROM
   (SELECT room_id,
               sum(valid_until - valid_since + 1) /
                    ((SELECT capacity
                      FROM room_types
                      WHERE room_types.id =
                             (SELECT room_type_id
                              FROM rooms
                              WHERE rooms.id = room_id))
                     * ( (SELECT max(valid_until) FROM contracts)
                          - (SELECT min(valid_since) from contracts) + 1)::float)
           AS average_daily_occupancy_of_room --pocet vsetkych dni
      FROM contracts
      GROUP BY room_id) as calculation
JOIN rooms on room_id = rooms.id
JOIN room_types rt3 on rooms.room_type_id = rt3.id
GROUP BY capacity;
```

Rozdiel v rýchlosti vykonania operácie zobrazuje tabuľka 2. Čas bol odmeraný pomocou `EXPLAIN ANALYZE` a počet rokov zaznamenaných v databáze bol 15.

V tabuľke sa nachádza aj hodnota −, ktorá označuje, že trvanie operácie trvalo oproti optimalizovanej verzii oveľa dlhšie, teda tento čas už nemalo zmysel merať.

veľkosť tabuľky `rooms` a `accounts` (v riadkoch)	čas pred optimalizáciou (v ms)	čas po optimalizácii (v ms)
100	35592	6
1000	3 360 540	55
10 000	-	547
100 000	-	4940

Tabuľka 2: Porovnanie rýchlosti vypočítania štatistiky pred a po optimalizácií podľa veľkosti tabuľky `accounts` a `rooms`

Optimalizácia stiahnutia mesačného poplatku

Stiahnutie mesačného poplatku za ubytovanie, je opäť operácia, ktorá beží cez viacero riadkov. Pôvodne som celú operáciu naprogramoval v Jave, kde som z databázy dostal zoznam platných zmlúv a každému nájomcovi sa následne stiahli peniaze, alebo v prípade nedostatku prostriedkov, sa im zablokovala karta. Tento spôsob mal nevýhodu v tom, že v každej iterácií niekoľkokrát posielal databázovému systému po sieti príkaz, čo spôsobovalo zbytočné zdržanie.

Pôvodný kód:

```
List<ContractRoomInfo> contracts = getValidInfo();
           int successfulPayments = 0;
           int unsuccessfulPayments = 0;
```

```
            BigDecimal sumOfAllPayments = BigDecimal.valueOf(0);

            for (ContractRoomInfo cri : contracts) {

                if
                    (cri.getAcc().getCredit().compareTo(cri.getRt().getMonthlyPayment())
                    < 0) {
                    unsuccessfulPayments++;
                    Card c =
                        CardFinder.getInstance().findByAccountId(cri.getAcc().getId());
                    c.setBlocked(true);
                    c.update();
                    continue;
                }

                Operation op = new Operation();
                op.setAmount(cri.getRt().getMonthlyPayment());
                op.setAccId(cri.getAcc().getId());

                cri.getAcc().setCredit(cri.getAcc().getCredit().
                             subtract(cri.getRt().getMonthlyPayment()));
                op.insert();
                cri.getAcc().update();
                successfulPayments++;
                sumOfAllPayments =
                    sumOfAllPayments.add(cri.getRt().getMonthlyPayment());
            }
            var res = List.of(successfulPayments, unsuccessfulPayments,
                sumOfAllPayments.intValue());
            DbContext.getConnection().commit();
            return res;
```

Celú operáciu som nahradil databázovou funkciou v PL/pgSQL, ktorá pošle databázovému systému príkaz iba raz a ten už všetko vykoná.

Optimalizovaný kód:

```
REATE OR REPLACE FUNCTION monthly_payment()
RETURNS TABLE (sum numeric, num_of_blocked int, num_of_successful int)
AS
$$
DECLARE
    c contracts;
    rt room_types;
    r rooms;
    card cards;
    a accounts;
    amount numeric;
    sum numeric;
    num_of_blocked int;
    num_of_successful int;
BEGIN
    sum := 0;
    num_of_blocked := 0;
    FOR c IN SELECT * FROM contracts WHERE contracts.status = 'Valid'
        LOOP
            SELECT * INTO STRICT a FROM accounts WHERE accounts.id = c.account_id;
            SELECT * INTO STRICT r FROM rooms WHERE rooms.id = c.room_id;
            SELECT * INTO STRICT rt FROM room_types WHERE room_types.id =
                r.room_type_id;
```

```
            SELECT * INTO STRICT card FROM cards WHERE cards.account_id = a.id;
            amount := rt.monthly_payment;

            IF amount <= a.credit THEN
                sum := sum + amount;
                INSERT INTO operations (amount, account_id) VALUES (amount, a.id);
                UPDATE accounts SET credit = credit - amount WHERE accounts.id = a.id;
                num_of_successful := num_of_successful + 1;
                continue;
            end if;

            IF amount > a.credit THEN
                UPDATE cards SET blocked = true WHERE cards.id = card.id;
                num_of_blocked := num_of_blocked + 1;
            end if;

        end loop;
    RETURN QUERY SELECT sum, num_of_blocked, num_of_successful;
end;
$$ LANGUAGE plpgsql;
```

Rozdiel v rýchlosti vykonania operácie zobrazuje tabuľka 3. V prípade pôvodného kódu bol čas odmeraný pomocou `System.currentTimeMillis()` a nový spôsob bol odmeraný pomocou `EXPLAIN ANALYZE`.

počet ubytovaných / veľkosť tabuľky `accounts` (v riadkoch)	čas	
	pred optimalizáciou (v ms)	po optimalizácii (v ms)
229 / 1000	4469	37
2212 / 10 000	40 910	293
11 025 / 50 000	203 119	1149
21 966 / 100 000	391 669	3327

Tabuľka 3: Porovnanie rýchlosti vykonania operácie stiahnutia mesačného poplatku pred a po optimalizácií podľa veľkosti tabuľky `accounts` a počtu ubytovaných študentov

6 Vybraný riešený problém

Medzi najzložitejšie problémy, ktoré som riešil určite patrí spomínaná optimalizácia operácií, ktoré potrebovali prechádzať viacero riadkov v tabuľkách, čo spomaľovalo vykonanie operácie kvôli volaniu databázy v každej iterácii cyklu.

Pôvodne som nevedel ako problém vyriešiť, avšak školiteľ mi neskôr poradil možnosť tieto operácie nahradiť vlastnými databázovými funkciami, ktoré by problém častého volania databázy vyriešili.

Naučil som sa, že pri optimalizácií operácií podobného tipu je najlepšie zmenšiť počet volaní databázy na minimum práve použitím `PL/pgSQL`.

Zdroje

- Na tvorbu tohto dokumentu bola použitá šablóna a štylizačný súbor zo záverečnej správy príkladového projektu na Databázy (2) od Ing. Alexandra Šimka PhD.

- Podobne v kóde boli použité časti príkladového projektu. Všetky takéto časti sú vyznačené komentárom s uvedeným zdrojom.