IDS

xkamen21 & **xfiala60**

Finální dokumentace k projektu

OBSAH:

Úvod

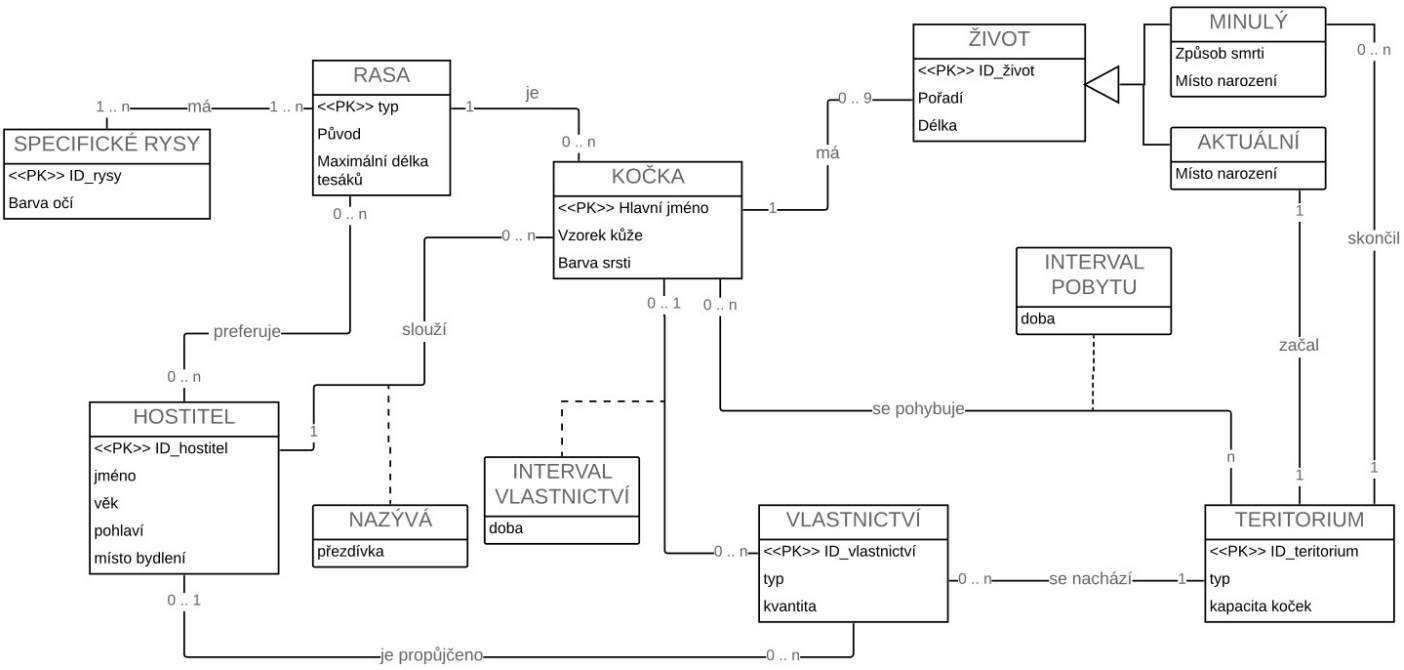
Naším úkolem bylo vypracovat návrh a implementaci relační databáze na určité téma. Rozhodli jsme se navázat na zadání projektu z předmětu IUS jednoho člena týmu.

Zadání

Kočky chtějí zefektivnit jejich dominanci lidského světa a proto Vám zadaly vytvořit KIS (Kitty Information System). Tento systém uchovává informace o jednotlivých rasách koček, jejich specifické rysy, jako možné barvy očí, původ, maximální délku tesáků, apod. a u konkrétních koček pak jejich hlavní jméno, vzorek kůže, barvu srsti a pod. Každá kočka má právě devět životů, nicméně v systému vedeme pouze ty, které již proběhly a aktuálně probíhají, a vedeme u nich informaci o délce života, místo narození a případně (u minulých životů) o místě (v rámci, kterého teritoria) a způsobu smrti. Kočky jsou samozřejmě majetnické a chtějí si vést všechny teritoria (máme teritoria různých typů, jako např. jídelna, klub, .), ve kterých se kdy pohybovaly a které věci si přivlastnily a v kterém intervalu je vlastnily (kočky se lehce znudí a své věci prostě zahodí). Systém rovněž vede informace o jejich hostitelích, kteří jim slouží. Veďte u nich jejich základní informace (jméno, věk, pohlaví, místo bydlení, .), které rasy koček preferují a rovněž jméno, kterým kočku nazývali (např. Pan Tlapoň, Bublina, Gaston, .). Některé vlastnictví koček však mohou být propůjčována svým hostitelům. Současně veďte informaci (pokud je přítomna) o teritoriu v rámci kterého se vlastnictví nachází, typ vlastnictví (hračka, cokoliv,.) a jeho kvantitu. Jednotlivá teritoria však mají omezenou kapacitu na kočky.

Týmová práce

Práce v našem týmu probíhala bez jakýchkoli komplikací. O potřebných věcech jsme diskutovali společně, a následně jsme si práci spravedlivě rozdělili. Komunikace probíhala osobně v případě tvorby ER Diagramu. Následná komunikace ohledně implementace programu probíhala pomocí aplikací typu *Discord, Skype, atd.* Jako prostředí pro práci jsme využili IDE DataGrip od JetBrains, které nám škola poskytla. Celý projekt jsme měli také přidaný na GitHub, ke kterému jsme měli přístup z obou stran. Práce poté byla jednodušší a efektivnější.



Implementace

Nejprve bylo potřeba vytvořit tabulky entit:

```
CREATE TABLE Kocka
CREATE TABLE Zivot
CREATE TABLE Vlastnictvi
CREATE TABLE Hostitel
CREATE TABLE Rasa
CREATE TABLE Specificke_rysy
CREATE TABLE Aktualni
CREATE TABLE Minuly
```

A následně tabulky zobrazující vztahy mezi tabulkami, případně atributy vztahu:

```
CREATE TABLE Pohyb_kocky
CREATE TABLE Interval_vlastnictvi
CREATE TABLE Slouzi
CREATE TABLE Rysy_rasy
CREATE TABLE Preference
```

U určitých atributů jsme museli ověřit jejich správnost, například ověření, že pohlaví hostitele je muž nebo žena. Tuto správnost jsme ověřovali pomocí regulárních výrazů. Např.:

```
CHECK (REGEXP_LIKE(pohlavi, '^([m|M][u|U][z|Z]|[z|Z][e|E][n|N][a|A])$'));
```

Poté už následovalo naplnění databáze ukázkovými daty a postupné splnění jednotlivých bodů zadání rozšiřující naši databázi – rozepsáno níže.

Při tvorbě primárních klíčů, jsme se snažili o unikátnost nejen v rámci dané tabulký, ale i v celé databázi. Rozhodli jsme se tedy přistoupit k řešení spojení úvodních písmen z názvu dané tabulky, a následně čísla unikátního pro každý záznam v tabulce. Příklady primárních klíčů pro:

tabulku **Kocka** tedy vypadá:
K1, K2, K3 …

pro tabulku **Pohyb_kocky**:
PK1, PK2, PK3 ...

SELECT

Slouží pro vrácení monžiny záznamů z jedné nebo více tabulek. V našem vypracování jsme implementovali 7 různých příkazů SELECT.

První dva dotazy jsou nejjednodušší, jedná se pouze o dotaz obsahující spojení dvou tabulek.

Příkaz SELECT vypíše všechny kočky a jejich typ rasy.

```
SELECT  Kocka.hlavni_jmeno as JMENO_KOCKY, Kocka.ID_rasy as ID_RASY, Rasa.typ as ZEME_PUVODU
FROM Kocka JOIN Rasa ON Kocka.ID_rasy = Rasa.ID_rasy
ORDER BY Kocka.ID_rasy;
```

Druhý SELECT vypíše všechny hostitele a jejich preferující rasu.

```
SELECT Hostitel.jmeno as JMENO_HOSTITELE, Hostitel.ID_hostitel as ID_HOSTITELE, Rasa.typ as TYP_RASY
FROM Preference JOIN Hostitel ON Preference.ID_hostitele = Hostitel.ID_hostitel
                JOIN Rasa ON Preference.ID_rasy = Rasa.ID_rasy
ORDER BY Hostitel.ID_hostitel;
```

Další dotaz je stále poměrně jednoduchý, jedná se o SELECT spojující tři tabulky. Dotaz vypíše všechny kočky a jejich typ rasy, k dané rase jsou vypsány specifické rysy.

```
SELECT Kocka.hlavni_jmeno as JMENO_KOCKY, Rasa.typ as TYP_RASY, Specificke_rysy.barva_oci as
SPECIFICKE_RYSY_BARVA_OCI, Specificke_rysy.max_delka_tesaku as SPECIFICKE_RYSY_DELKA_TESAKU,
Specificke_rysy.puvod as SPECIFICKE_RYSY_PUVOD
FROM Rysy_rasy JOIN Specificke_rysy ON Specificke_rysy.ID_rysy = Rysy_rasy.ID_rysy
               JOIN Rasa ON Rasa.ID_rasy = Rysy_rasy.ID_rasy
               JOIN Kocka on Rasa.ID_rasy = Kocka.ID_rasy
ORDER BY Kocka.hlavni_jmeno;
```

Při dalších dvou dotazech byla využita klauzule GROUP BY a agregáční funkce.

První dotaz SELECT vypíše jméno kočky a v kolikátém životě se nachází.

```
SELECT Zivot.jmeno_kocky as JMENO_KOCKY, COUNT(*) as V_KOLIKATEM_ZIVOTE_SE_KOCKA_NACHAZI
FROM Zivot
GROUP BY Zivot.jmeno_kocky;
```

Druhý vypíše typ teritoria, kolik koček v daném teritoriu žilo a nejdelší délku pobytu kočky.

```
SELECT Teritorium.typ_teritoria as TYP_TERITORIA, COUNT(Pohyb_kocky.ID_teritoria) as
KOLIK_KOCEK_ZDE_ZILO, MAX(Pohyb_kocky.interval_pobytu) as NEJDELSI_DELKA_POBYTU_KOCKY
FROM Pohyb_kocky JOIN Teritorium ON Pohyb_kocky.ID_teritoria = Teritorium.ID_teritorium
GROUP BY Teritorium.typ_teritoria;
```

V neposlední řadě zde máme dotaz obsahující predikát *EXISTS*

Dotaz vypíše v jakých teritoriich se vvyskytovala více jak jedna kočka

```
SELECT Teritorium.ID_teritorium as ID_TERITORIA, Teritorium.typ_teritoria as TYP_TERITORIA
FROM Teritorium
WHERE EXISTS(
        SELECT COUNT(Pohyb_kocky.ID_teritoria)
        FROM Pohyb_kocky
        WHERE Pohyb_kocky.ID_teritoria = Teritorium.ID_teritorium
        GROUP BY Teritorium.typ_teritoria
        HAVING COUNT(Pohyb_kocky.ID_teritoria) > 1
        )
ORDER BY Teritorium.typ_teritoria;
```

V poslední řadě zde máme SELECT s predikátem IN a vnořeným selectem

Dotaz vypíše všechny kočky, které žijí nebo žili pouze rok nebo méně.

```
SELECT Kocka.hlavni_jmeno as JMENO_KOCKY, Kocka.barva_srsti as BARVA_SRSTI, Kocka.vzorek_kuze as
VZOREK_KUZE
FROM Kocka
WHERE Kocka.hlavni_jmeno IN (
        SELECT Zivot.jmeno_kocky
        FROM Zivot
        WHERE Zivot.delka < 365
        );
```

INDEX & EXPLAIN PLAN FOR

```
EXPLAIN PLAN FOR
  SELECT Kocka.hlavni_jmeno, count(*) AS POCET_UMRTI
  FROM Kocka, Zivot, Minuly
  WHERE Kocka.hlavni_jmeno = Zivot.jmeno_kocky AND Zivot.ID_zivot = Minuly.ID_zivot
  GROUP BY Kocka.hlavni_jmeno;
SELECT PLAN_TABLE_OUTPUT FROM table (DBMS_XPLAN.DISPLAY());

CREATE INDEX index_minuly ON Minuly(ID_zivot);
```

EXPLAIN PLAN FOR slouží pro zobrazení plánu příkazů provadění vybrané optimalizátorem pro příkazy SELECT, UPDATE, INSERT a DELETE. V našem kódu jsme si proto vytvořili nový příkaz SELECT, který má za úkol zobrazit data o umrtí koček (přesněji počet úmrtí). Při vypsání EXPLAIN PLAN pro daný SELECT jsme dostali zde uvedený výsledek.

```
----------------------------------------------------------------------------
| Id  | Operation           | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |        |     7 |   651 |     7  (15)| 00:00:01 |
|   1 |  HASH GROUP BY      |        |     7 |   651 |     7  (15)| 00:00:01 |
|*  2 |   HASH JOIN         |        |     7 |   651 |     6   (0)| 00:00:01 |
|   3 |    TABLE ACCESS FULL| MINULY |     7 |    28 |     3   (0)| 00:00:01 |
|   4 |    TABLE ACCESS FULL| ZIVOT  |    17 |  1513 |     3   (0)| 00:00:01 |
----------------------------------------------------------------------------
```

Celková “cena” (cost) se nám vyčíslila na 26. V operacích také vidíme, že SELECT přistupoval do ’celé tabulky’, takzvaně měl přístup k celému obsahu tabulky i když to nebylo potřebné. Zde máme místo na použití optiamlizace pomocí vytvoření INDEXU.

INDEX by se dal přirovnat k ukazateli, který se odkazuje na určitá data, která mu přiřadíme.

Náš INDEX jsme si tedy zvolili na tabulku Minuly a přesněji na sloupec ID_zivot. Po spuštěni jsme hned dostali optimálnější výsledky.

```
-----------------------------------------------------------------------------
| Id  | Operation           | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |              |     7 |   651 |     4  (25)| 00:00:01 |
|   1 |  HASH GROUP BY      |              |     7 |   651 |     4  (25)| 00:00:01 |
|   2 |   NESTED LOOPS      |              |     7 |   651 |     3   (0)| 00:00:01 |
|   3 |    TABLE ACCESS FULL| ZIVOT        |    17 |  1513 |     3   (0)| 00:00:01 |
|*  4 |    INDEX RANGE SCAN | INDEX_MINULY |     1 |     4 |     0   (0)| 00:00:01 |
-----------------------------------------------------------------------------
```

Jak můžeme vidět, celková cena rapidně klesla a to o 12 z celkové “ceny” (cost). Ve sloupci operace mužeme vidět, že SELECT už nepřistupoval k celé tabulce, ale použil námi vytvořený index, který odkazuje pouze na část tabulky.

Dále se nám zde nabízí optimalizace přístupu do tabulky Zivot.

```
CREATE INDEX index_zivot ON Zivot(ID_zivot, jmeno_kocky);
```

Triggery

Triggerů v naší databázi máme hned několik, avšak 9/10 jich plní tu samou funkci, a to generování primárních klíčů všem tabulkám s primárními klíči v databázi. Plnění veškerých dat, je tedy realizováno bez primárních klíčů, právě kvůli těmto triggerům.

Každý tento jednotlivý trigger má vlastní sekvenci, která se o jedna zvyšuje s každým vygenerovaným klíčem. Vygenerované číslo z této sekvence tedy vždy spojíme s písmeny, kterými začíná jméno dané tabulky. Např. primární klíč pro tabulku Teritorium:

```
:NEW.ID_teritorium := 'T'|| teritorium_pk_seq.nextval;
```

Poslední trigger slouží ke kontrole životů kočky. Každá kočka totiž může mít pouze 9 životů. Trigger je implementován tak, že prochází všechny záznamy v tabulce *Život* a kontroluje zda-li pořadí životů koček nepřesahuje rozmezí 1 až 9.

V případě chyby je aplikace ukončena s chybovou hláškou.

Procedury

Procedury se používají v SQL k opakování určité části kódu, bez zbytečné duplikace. Dalo by se je přirovnat k funkcím ostatních programovacích jazyků jako jsou: C, C++, Java, Python atd..

V naší databázi máme vytvořeny 2 procedury.

- prumerna_kapacita_teritoria()
  Tato procedura spočítá průměrnou kapacitu všech teritorií nacházejících se v naší databázi, a zároveň vypíše název a kapacitu teritoria s nejmenší a největší kapacitou.
  Je implementována jako smyčka procházející všechny řádky v tabulce *Teritorium* a kapacita procházeného teritoria je porovnávána s dosud známým největším a nejmenším teritoriem. Kapacita všech teritorií je sčítána a ve výsledku podělena počtem teritorií. Výsledek zaokrouhlujeme na 0 desetinných míst, jelikož kočka v teritoriu musí být vždy celá. Po ukončení smyčky jsou výsledky vypsány (demonstrace).

- procentualni_rozdeleni_hostitelu_podle_veku_a_pohlavi()
  Tato procedura vrátí celkové rozložení zastoupení hostitelů v daných věkových skupinách a také jejich pohlaví. Celá procedura funguje na stejném principu jak předešlá, nekonečný cyklus, který se ukončí při vyčerpání všech dat v databázi.
  Navyšuje počítadla specifických skupin, které pak pro demonstraci vypíše.

Obě procedury v případě nenalezení žádných dat vypíší chybovou hlášku a aplikace je ukončena. Obdobně je řešena i chyba v proceduře. Vše je vráceno přes *RAISE_APPLICATION_ERROR().*

Přístupová práva

Přístupová práva k jednotlivým prvkům dané databáze jsou převážně využívána z důvodu bezpečnosti.

V naší databázi jsou přístupová práva udělena druhému členu (xkamen21). Jsou to práva INSERT, UPDATE, SELECT, DELETE pro všechny entity v dané databázi, a veškerá přístupová práva pro ostatní tabulky a procedury.

Demonstrace přístupových práv je znázorněna v následující sekci Materializovaný pohled, kde veškerý přístup a úprava databáze je realizována právě uživatelem xkamen21.
V materializovaném pohledu se avšak využívají pouze práva pro tabulku Kočka a Rasa.

Materializovaný pohled

Materializovaný pohled se používá pro výsledek dotazu, který je uložen v databázi, ale je zjevný až po aktualizaci dané tabulky.

V implementaci naší databáze se sledují tabulky zobrazující všechny existující kočky určité rasy. Aktualizace pohledu je zřejmá až po provedení příkazu COMMIT.

1. Ukázka funkčnosti materializovaného pohledu je znázorněna právě výpisem samotného SELECTu všech koček dané rasy, následné vložení příkazem INSERT nové kočky se jménem „zabka“ a opětným výpisem všech koček. Při druhém výpisu je vidět, že tabulka je stále nezměněna, jelikož nebyl použit aktualizační příkaz COMMIT. Následně je tabulka aktualizována, a poté je znovu vypsána. Zde již vidíme aktuální tabulku i s novou kočkou. Obdobným způsobem je dále demonstrován materializovaný pohled s příkazem DELETE nad kočkou se jménem „zabka“.