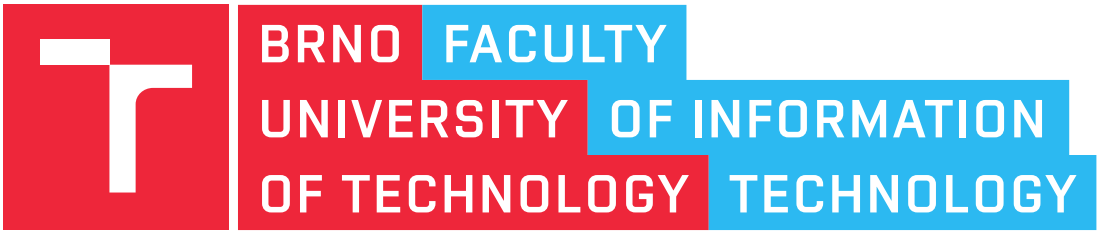


Databázové systémy
2017/2018

Dokumentace k projektu

Vladimír Dušek (xdusek27)
Tomáš Kukaň (xkukan00)

2. května 2018

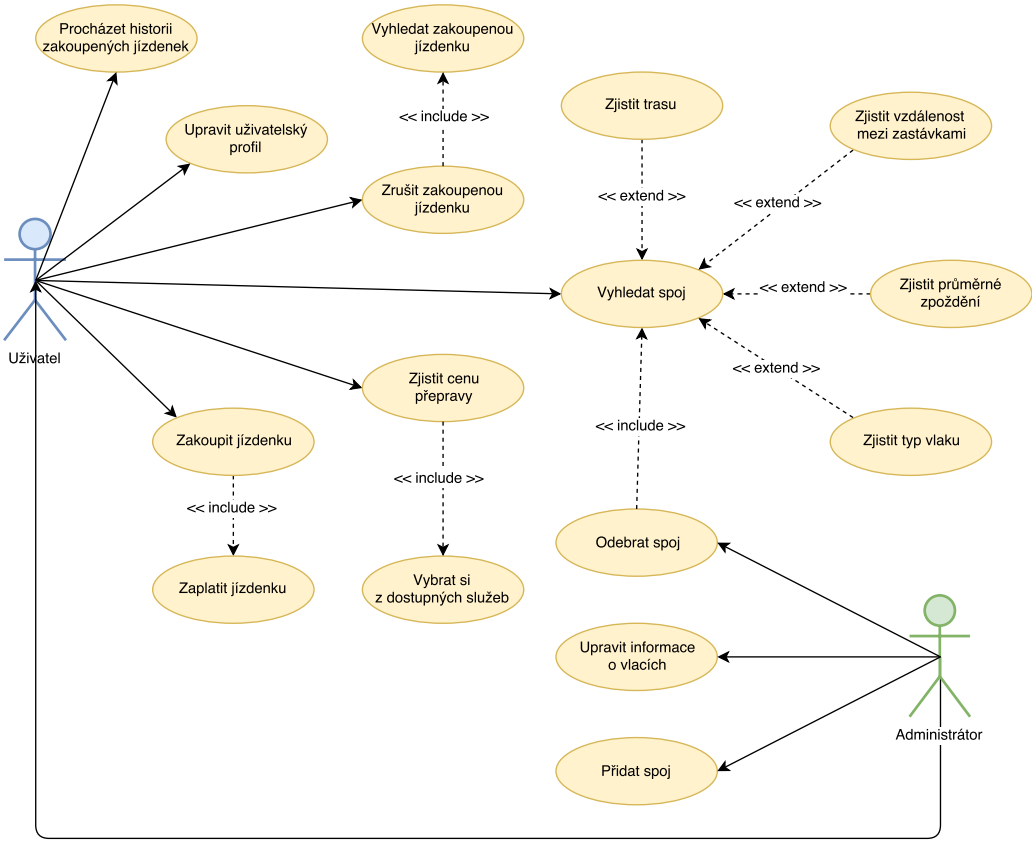
Obsah

1 Zadání

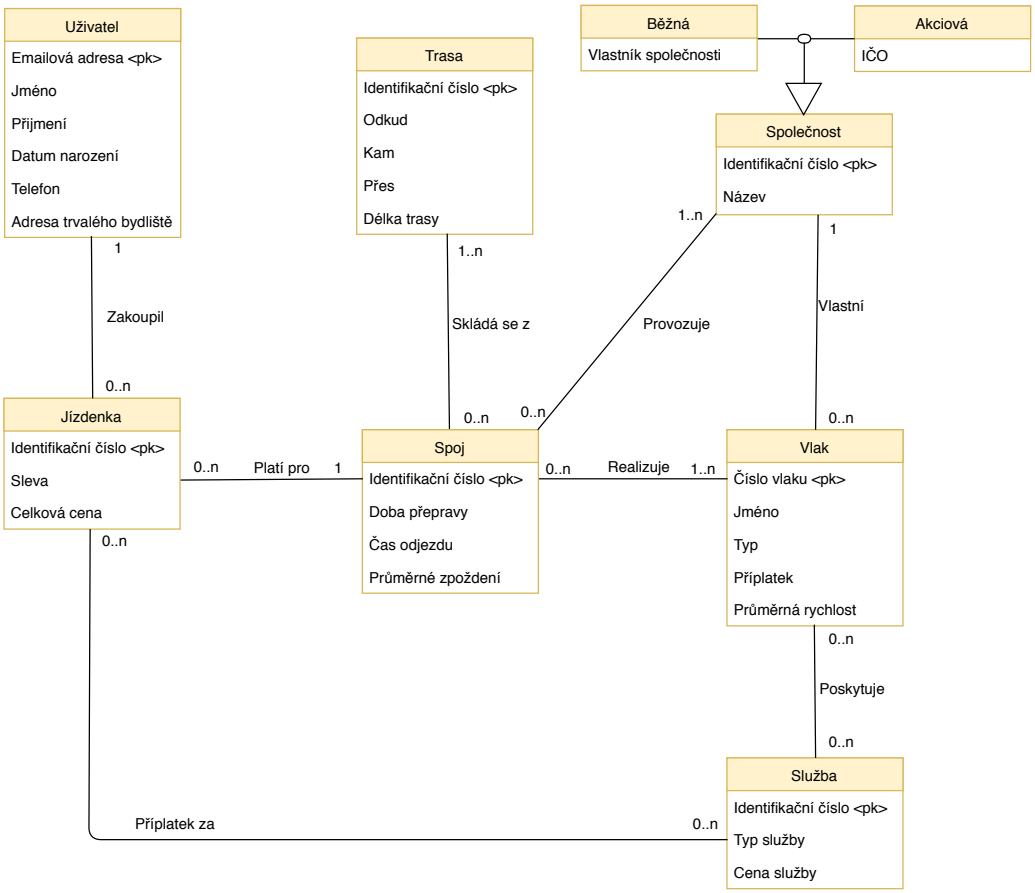
Navrhněte fragment informačního systému pro prodej lístků na vlakové spoje. Systém bude umět odpovědět na dotazy typu jakou trasou zvolený spoj jede, jaké jsou vzdálenosti (časové) mezi zastávkami, jaký typ vlaku jede, průměrné zpoždění vlaku, apod. Doba přepravy mezi dvěma zastávkami může být různá v závislosti na typu vlaku. Uvažujte více společností provozujících vlakové spoje. Různé vlaky nabízí různé služby (jídelní vůz, spací vůz, přeprava kol, apod.), za jejich využití se platí pevně daný příplatek. Dostupnost služby či velikost příplatku se liší v závislosti na vlaku. Na základě uložených dat by si měl uživatel být schopen vyhledat spoj mezi dvěma zastávkami, zjistit dostupné služby a zjistit výslednou cenu přepravy. Uživatel si může přes internet koupit (a zrušit již zakoupenou) jízdenku a procházet historii svých jízdenek.

Zadání je převzato z předmětu IUS – Lístky na vlak

2 Use case diagram



3 Entity-relationship diagram



4 SQL skript

Skript nejprve odstraní všechny databázové objekty kvůli zabránění potenciálním konfliktům. Postupně se vytvoří všechny tabulky a nastaví se jejich primární a cizí klíče. Tabulky jsou naplněny daty pro další práci s databází. Poté je vytvořeno několik triggerů a uložených procedur včetně otestování jejich funkcionality. Následuje několik dotazů select. Skript dále demonstruje využití indexu pro optimalizaci zpracování dotazu pomocí explain plan, dále využití materializovaného pohledu a přidělení přístupových práv k databázovým objektům druhému členovi týmu.

5 Triggery

Implementovali jsme dva triggery. První přesně vyplývá ze zadání a slouží pro automatické generování hodnot primárního klíče, v našem případě je to id jízdenky – `jizdenka_id_auto_inc`. Trigger se realizoval pomocí sekvence `sq_last_jizdenka_id`, která uchovává poslední vložené ID. Začíná se od 1 a postupně se inkrementuje.

Druhý trigger má identifikátor `check_spolecnost_ico` a slouží pro kontrolu správného zadání IČO společnosti. Trigger kontroluje jestli má IČO správnou délku (8 znaků), jestli se skládá pouze z číslic a také jestli je platné. Neprojde-li IČO kontrolou je vyvoláno chybové hlášení s kódem -20010 pomocí `raise_application_error`.

6 Uložené procedury

Implementovali jsme také dvě uložené procedury. První `sp_kontrola_email` používá kurzor – `cr_uzivatel` a proměnnou odkazující se na sloupec tabulky – `ptr_column_email`. Slouží pro kontrolu správného formátu emailu uživatele. Neodpovídá-li formát, je vyhozena výjimka `ex_invalid_email`. Ta je později odchycena a je vyvoláno chybové hlášení s kódem -20020 pomocí `raise_application_error`.

Druhá procedura `sp_kontrola_vlak_id` funguje na podobném principu a slouží pro kontrolu id vlaku, tedy jestli se nejedná o záporné číslo. Nesplňuje-li id tuto podmínku je vyhozena výjimka `ex_invalid_id`. Ta je později odchycena a je vyvoláno chybové hlášení s kódem -20030 pomocí `raise_application_error`.

7 Explain plan a vytvoření indexu

`EXPLAIN PLAN` nám ukazuje jak je daný dotaz zpracováván databázovým strojem, dává nám tedy podněty k optimalizaci. Pro demonstraci možné optimalizace jsme vybrali dotaz, který spojí dvě tabulky, využije agregační funkci, klauzuli `GROUP BY` a vypíše jméno a přijmení uživatele a kolik celkem utratil v našem systému za jízdenky.

`EXPLAIN PLAN` pro náš dotaz vypadá následovně:

```
-----------------------------------------------------------------------------------------
| Id  | Operation                     | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |               |     3 |   363 |     7  (29)| 00:00:01 |
|   1 |  HASH GROUP BY                |               |     3 |   363 |     7  (29)| 00:00:01 |
|   2 |   NESTED LOOPS                |               |     3 |   363 |     6  (17)| 00:00:01 |
|   3 |    NESTED LOOPS               |               |     7 |   363 |     6  (17)| 00:00:01 |
|   4 |     VIEW                      | VW_GBC_5      |     7 |   280 |     4  (25)| 00:00:01 |
|   5 |      HASH GROUP BY            |               |     7 |   280 |     4  (25)| 00:00:01 |
|*  6 |       TABLE ACCESS FULL       | JIZDENKA      |     7 |   280 |     3   (0)| 00:00:01 |
|*  7 |     INDEX UNIQUE SCAN         | SYS_C00128562 |     1 |       |     0   (0)| 00:00:01 |
|   8 |    TABLE ACCESS BY INDEX ROWID| UZIVATEL      |     1 |    81 |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------
```

Ve sloupci *Operation* můžeme vidět jaké operace by databázový stroj vykonal pro provedení našeho dotazu. `SELECT STATEMENT` znamená, že se vykonal dotaz `SELECT`. `HASH GROUP BY` značí seskupení položek podle hashovacího klíče. `NESTED LOOPS` je vnořený cyklus. Následuje `TABLE ACCES FULL`, to nám značí přechod celou tabulkou bez použití indexů. `INDEX UNIQUE SCAN` značí jeden z přístupů k tabulkám přes B-strom. Ve sloupci *Cost* můžeme vidět cenu operace.

Pro optimalizaci našeho dotazu jsme vytvořili index pomocí příkazu `CREATE INDEX idx_uz_email ON Jizdenka(uzivatel_email)`, pro snažší přístup k emailu uživatele v tabulce `Jizdenka`. Znovu jsme spustili `EXPLAIN PLAN`.

```
-------------------------------------------------------------------------------------------------
| Id  | Operation                                | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                         |               |     3 |   363 |     5  (20)| 00:00:01 |
|   1 |  HASH GROUP BY                           |               |     3 |   363 |     5  (20)| 00:00:01 |
|   2 |   NESTED LOOPS                           |               |     3 |   363 |     4   (0)| 00:00:01 |
|   3 |    NESTED LOOPS                          |               |     7 |   363 |     4   (0)| 00:00:01 |
|   4 |     VIEW                                 | VW_GBC_5      |     7 |   280 |     2   (0)| 00:00:01 |
|   5 |      HASH GROUP BY                       |               |     7 |   280 |     2   (0)| 00:00:01 |
|   6 |       TABLE ACCESS BY INDEX ROWID BATCHED| JIZDENKA      |     7 |   280 |     2   (0)| 00:00:01 |
|*  7 |        INDEX FULL SCAN                   | IDX_UZ_EMAIL  |     1 |       |     1   (0)| 00:00:01 |
|*  8 |     INDEX UNIQUE SCAN                    | SYS_C00128562 |     1 |       |     0   (0)| 00:00:01 |
|   9 |    TABLE ACCESS BY INDEX ROWID           | UZIVATEL      |     1 |    81 |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------
```

Jak můžeme vidět cena operací se snížila. Operace `TABLE ACCES BY INDEX ROWID BATCHED` nám říká, že do tabulky se přistupuje přes konkrétní řádek, databázový stroj tedy použil náš index.

8 Materializovaný pohled

Nejprve jsme vytvořili materializované logy, kde se uchovávají změny původních tabulek. Ty jsou potřeba, abychom materializovanému pohledu mohli přiřadit vlastnost `FAST REFRESH ON COMMIT` místo výchozího `COMPLETE REFRESH`. Při tom se musí znovu spouštět celý dotaz materializovaného pohledu, což znamená vyšší cenu.

Poté jsme již vytvořili materializovaný pohled a demonstrovali jeho funkčnost na příkladu. Změnili jsme obsah původní tabulky, změny v materializovaném pohledu se projevili až po příkazu `COMMIT`. Materializovanému pohledu jsme dále nastavili vlastnosti `CACHE` – optimalizace čtení z pohledu a `BUILD IMMEDIATE` – naplnění pohledu ihned po jeho vytvoření.

Dále jsme pomocí příkazu `EXPLAIN PLAN` požádali o vysvětlení zpracování původního dotazu `SELECT`. Získali jsme přehled o jednotlivých operacích, které databázový stroj musí vykonat a jejich náročnosti.

```
-----------------------------------------------------------------------------
| Id  | Operation          | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |            |     6 |   624 |     6   (0)| 00:00:01 |
|*  1 |  HASH JOIN         |            |     6 |   624 |     6   (0)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| SPOLECNOST |     2 |   104 |     3   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| VLAK       |     6 |   312 |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------
```

Následně jsme požádali o vysvětlení zpracování dotazu `SELECT` našeho materializovaného pohledu.

```
-----------------------------------------------------------------------------------------
| Id  | Operation            | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |                     |     6 |   276 |     3   (0)| 00:00:01 |
|   1 |  MAT_VIEW ACCESS FULL| MV_VLAK_SPOLECNOST  |     6 |   276 |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------
```

Dosáhli jsme snížení ceny, tedy menší zátěže CPU díky materializovanému pohledu.

9 Přídělení přistupových práv

Druhý člen týmu má přidělená přístupová práva pomocí příkazu `GRANT`. Konkrétně
`GRANT ALL ON table_name TO login2` pro tabulky a materializovaný pohled a
`GRANT EXECUTE ON stored_procedure_name TO login2` pro uložené procedury. Druhý člen týmu se poté připojí k databázi prvního člena týmu pomocí příkazu
`ALTER SESSION SET CURRENT_SCHEMA = login1` a následně může s databázovými objekty pracovat.