Vysoké učení technické v Brně

Fakulta informačních technologií

Databázové Systémy

2018

Projekt 5. část - Dokumentace

Zadání z IUS č. 53 - Velká Éra Pirátů

Miroslav Válka (xvalka05) Brno, 27. 4. 2018

53. Velká Éra Pirátů

Piráti dopili rum a chtěji vytvořit informační systém, který by zefektivnil jejich rabování. Pirátské posádky mají svá unikátní jména, své Jolly Rogery (tzn. vlajky) a sestávají (pochopitelně) z bandy pirátů. Jednotliví piráti, mimo svého jméno (nicméně existuje řada bezejmenných pirátů) a přezdívky (např. Černovous), jsou charakterizováni svou pozicí v posádce (navigator, kuchař, doktor, kormidelník, .), věkem, barvou vousů, časem stráveným v posádce a seznamem charakteristik (chybějící oko, papoušek na rameni, dřevěná noha,.).

Každá posádka má svého (hlavního) kapitána a vlastní buď jednu loď, nebo celé flotily sestávající z více lodí. Každá loď i flotila má pak svého divizního kapitána (může to být i kapitán celé posádky), a je charakterizována typem (karavela, brigantina,.) a přístavem ve kterém kotví, přičemž kapacita každé lodi pro posádku je omezená. Piráti se můžou plavit maximálně na jedné lodi.

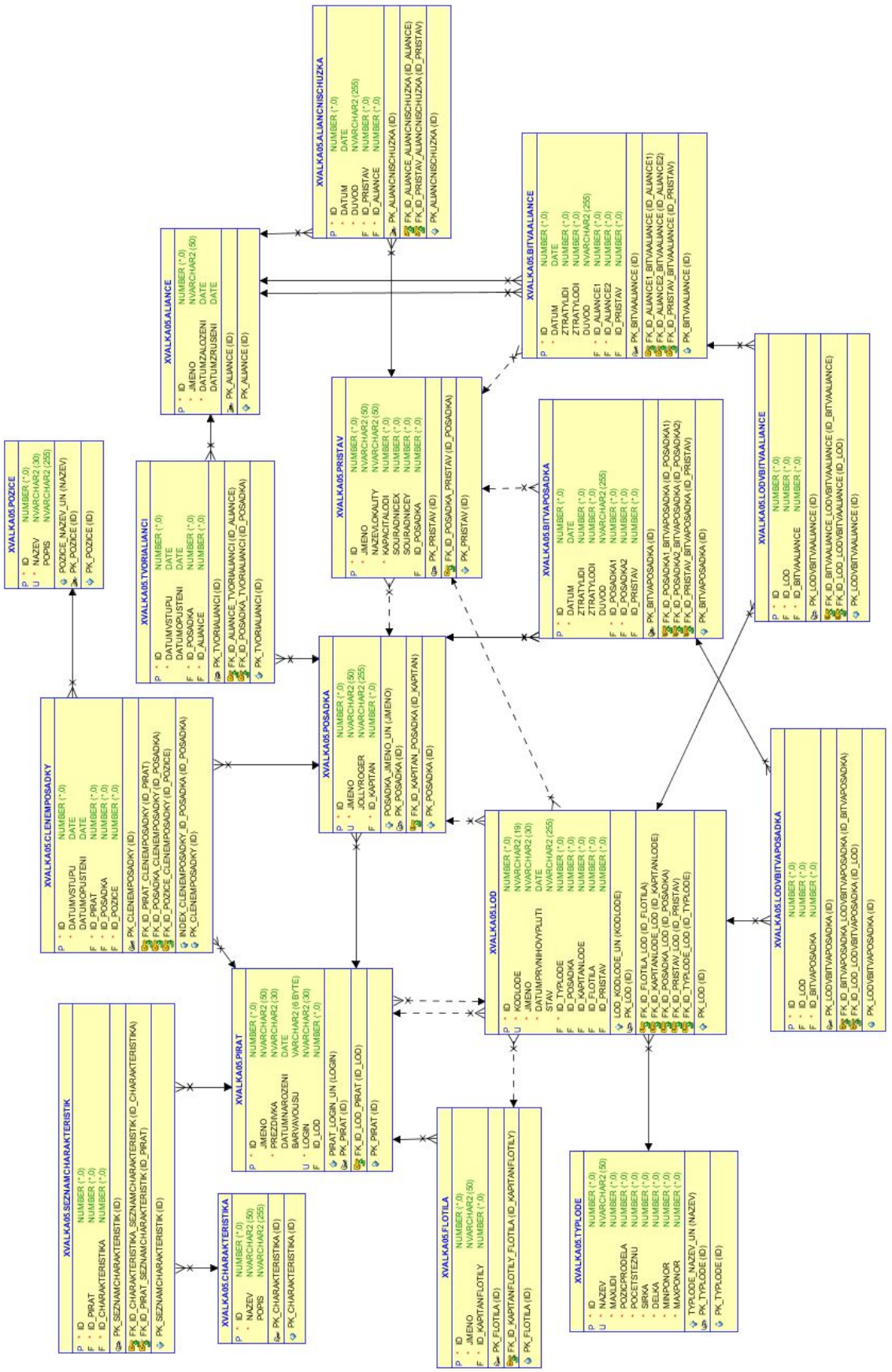
U přístavů uchováváme informace o tom, zda se jedná o teritorium specifické pirátské posádky, nebo o neutrální uzemí a název (polo)ostrovu, kde se nachází a kapacitu ukotvených lodí. Pirátské posádky dále vytváří vzájemné aliancie. Tyto aliance mají dohodnutý jeden přístav (může se jednat o teritorium jedné z posádek), ve kterém probíhají alianční schůzky. Jednotlivé posádky i celé aliance pak mezi sebou mohou bojovat.

U těchto bitev evidujeme, zda probíhaly v přístavu (a případně v kterém) nebo na volném moři a dále počet ztrát (stačí kvantitativně) a konkrétní loďě co se v bitvě zapojily. Systém navíc umožňuje kapitánům posádek vyzvat alianční posádky o pomoc (při chystané bitvě; předpokládejte formu jednoduchého broadcastu).

Pro jednoduchost předpokládejte, že POUZE kapitán posádky může manipulovat v IS s údaji o svých posádkách a lodích (tedy divizní a flotilní kapitáni mají v systému stejná práva jako řádoví piráti a jejich speciální privilegia a povinnosti se projevují pouze ve reálných událostech, jako jsou bitvy a plavby).

Miroslav Válka (xvalka05) Brno, 27. 4. 2018

Schéma relační databáze



Generalizace / Specializace

Tento vztah se v projektu týká bitev. Bitvy mohou být mezi Posádkou a Posádkou nebo mezi Aliancí a Aliancí. Při transformaci do relační databáze došlo k vytvoření dvou samostatných tabulek pro bitvy a také k vytvoření dvou tabulek pro uchovávání účasti lodí v bitvě.

SQL skript

Procedury/Funkce

Ve skriptu se nachází procedury *DropTableIfExist, DropSequenceIfExist, NewForeignKey*. Tyto procedury seskládají SQL příkaz, který následně provedou. Slouží primárně k usnadnění práce při vytváření respektive mazání tabulek. Procedury *DropTableIfExist*, *DropSequenceIfExist* jsou určeny k elegantnějšímu mazání bez chybového hlášení, neboť PL SQL narozdíl od MySQL neobsahuje *DROP table_name IF EXISTS*, takže vyvolání mazaní nad neexistující tabulkou vyvolá chybu. Procedura *NewForeignKey* zase slouží k snadnějšímu vytváření pojmenovaných cizích klíčů.

Dale je ve skriptu i funkce *GeneratorKodLode*, která se pokusí vygenerovat unikátní klíč, který se nevyskytuje v tabulce u jiné lodě, pro loď, kterou dostane v parametru. Tento nově vygenerovaný kód lodě je následně funkcí vrácen.

Další procedura *JeNaLodiMisto* zase slouží k vypsání jestli je na dané lodi ještě místo nebo jestli je obsazená.

Triggery

Skript obsahuje tři databázové triggery. Jeden trigger je čistě pro generování primárního klíče ze sekvence. Tento trigger generuje primární klíče pro tabulku *Pozice*.

Další trigger, který automaticky vytváří / upravuje *KodLode* v tabulce *Lod* při manipulaci se záznamy. Formát kódu lodě je následující: První 4 písmena z názvu typu lodě, rok 1. vyplutí, prvních 4 až 8 písmen z nazvu lodě (bez bílých znaků). Příklad: Loď typu Karavela, datum vyplutí 1987-12-30, jméno lodě Bleda smrt => Kód lodě: Kara-1987-Bleda .

Třetí trigger slouží právě ke kontrole správnosti kódu lodě zda odpovídá danému formátu..

Přidělování práv

I když pracuji jako jednotlivec, tak se ve skriptu nachází část s přidělováním práv. Práva která jsou nastavována odpovídají právům k akcím, která může vykonávat řadový pirát.

Taktéž se zde nachází ukázka, jak práva odebrat nebo jak případně používat cizí objekty.

Materializovaný pohled

Nejprve skript vytvoří materializované logy, kde se uchovávají změny hlavní tabulky. Díky tomu v materializovaném pohledu je možné použít *FAST REFRESH ON COMMIT* a není tak potřeba dělat *COMPLETE REFRESH*, který provádí znovu celý dotaz pro tvorbu materialitovaného pohledu.

Dále je vytvořen samotný materializovaný pohled. Jelikož pracuji jako jednotlivec, tak nemohu manipulovat s tabulky jiných osob, ale postup se příliš neliší. Pro práci s tabulky jiné osoby by bylo potřeba ještě přidat před názvy tabulek název učtu vlastníka tabulek, nebo si vytvořit synonyma a ty poté použít v dotazu a vlastník tabulek by musel mi nastavit přístupová práva k tabulkám.

Jinak u pohledu byly dále nastaveny následující vlastnosti: *CACHE* (optimalizace načítání pohledu), *BUILD IMMEDIATE* (Naplnění pohledu po vytvoření), *ENABLE QUERY REWRITE*.

Explain plan a index

EXPLAIN PLAN byl použit pro dva různé dotazy. Pro jeden jednoduší dotaz, který počítá kolik lidí se v daný den přidalo do nějaké posádky. A druhý dotaz, který je komplikovanější, který vypisuje kolik pirátů měla která posádka v době konání bitvy.

První plán činnosti:

```
-------------------------------------------------------------------------------------
| Id  | Operation          | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |               |    77 |   693 |     4  (25)| 00:00:01 |
|   1 |  SORT GROUP BY     |               |    77 |   693 |     4  (25)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| CLENEMPOSADKY |    77 |   693 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------------
```

Rozebereme si obsah plánu. Sloupeček *Operation* obsahuje akci, která bude probíhat, Sloupeček *Name* jest název objektu se kterým se pracuje, *Rows* je počet zpracovávaných řádků, *Cost (%CPU)* přestavuje hodnotu udělenou optimalizátorem *Cost-based optimiser (CBO)* a procentuální využití procesoru.

SELECT STATEMENT oznamuje, že se uskuteční *SELECT* dotaz. *SORT GROUP BY* jest sloučení akce *GROUP BY* a *ORDER BY* neboť seskupení i seřazení probíhá dle stejného sloupečku hodnot. A na závěr zpracovávání řádků *TABLE ACCESS FULL* při kterém dochází k postupnému procházení všech řádků a porovnávání hodnot. Tato akce je ohodnocena CBO 3 a *SORT GROUP BY* s ohodnocením o jedna vetší.

První plán činnosti s indexem:

Neboť dotaz pracuje převážně s jednou hodnotou při seskupování, řazení a výpisu, tak pro ni založíme index a podíváme se na změnu.

```
-----------------------------------------------------------------------------------------------------
| Id  | Operation         | Name                              | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |                                   |    77 |   693 |     2  (50)| 00:00:01 |
|   1 |  SORT GROUP BY    |                                   |    77 |   693 |     2  (50)| 00:00:01 |
|   2 |   INDEX FULL SCAN| INDEX_CLENEMPOSADKY_DATUMVSTUPU |    77 |   693 |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------
```

Vidíme výraznou změnu v řádku s id 2. Namísto akce *TABLE ACCESS FULL* se provede *INDEX FULL SCAN*. Takže řádky budou prohledávány, seskupovány a řazeny na základě vytvořeného indexu. CBO ohodnocení hledání v indexu je 1 a pro *SORT GROUP BY* jest opět o jedna větší. Nicméně i když bude dotaz proveden rychleji, tak využití procesoru bude také větší. Podle informacích jsem se dočetl, že je to způsobeno rychlostí čtení dat při vykonávání dotazu. Dvakrát rychleji přečtená data mohou být dvakrát rychleji zpracována a je tak lépe využit procesorový výkon.

Druhý plán činnosti:

```
-----------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |               |     3 |   458 |    26  (27)| 00:00:01 |
|   1 |  SORT UNIQUE                  |               |     3 |   458 |    25  (24)| 00:00:01 |
|   2 |   UNION-ALL                   |               |       |       |            |          |
|   3 |    HASH GROUP BY              |               |     2 |   288 |    13  (31)| 00:00:01 |
|*  4 |     HASH JOIN                 |               |     2 |   288 |    11  (19)| 00:00:01 |
|   5 |      MERGE JOIN               |               |    10 |   790 |     8  (25)| 00:00:01 |
|   6 |       SORT JOIN               |               |    51 |  2448 |     4  (25)| 00:00:01 |
|   7 |        TABLE ACCESS FULL      | BITVAPOSADKA  |    51 |  2448 |     3   (0)| 00:00:01 |
|*  8 |       FILTER                  |               |       |       |            |          |
|*  9 |        SORT JOIN              |               |    77 |  2387 |     4  (25)| 00:00:01 |
|  10 |         TABLE ACCESS FULL     | CLENEMPOSADKY |    77 |  2387 |     3   (0)| 00:00:01 |
|  11 |      TABLE ACCESS FULL        | POSADKA       |     9 |   585 |     3   (0)| 00:00:01 |
|  12 |    HASH GROUP BY              |               |     1 |   170 |    12  (17)| 00:00:01 |
|  13 |     NESTED LOOPS              |               |     1 |   170 |    10   (0)| 00:00:01 |
|  14 |      NESTED LOOPS             |               |     1 |   170 |    10   (0)| 00:00:01 |
|  15 |       NESTED LOOPS            |               |     1 |   105 |     9   (0)| 00:00:01 |
|  16 |        NESTED LOOPS           |               |     1 |    79 |     6   (0)| 00:00:01 |
|  17 |         TABLE ACCESS FULL     | BITVAALIANCE  |     1 |    48 |     3   (0)| 00:00:01 |
|* 18 |         TABLE ACCESS FULL     | CLENEMPOSADKY |     1 |    31 |     3   (0)| 00:00:01 |
|* 19 |        TABLE ACCESS FULL      | TVORIALIANCI  |     7 |   182 |     3   (0)| 00:00:01 |
|* 20 |       INDEX UNIQUE SCAN       | PK_POSADKA    |     1 |       |     0   (0)| 00:00:01 |
|  21 |      TABLE ACCESS BY INDEX ROWID| POSADKA       |     1 |    65 |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------
```

Druhý dotaz je značně komplikovanější a dochází při něm ke spojení několika tabulek, selekci, seskupování a řazení. *SELECT STATEMENT* opět označuje daný *SELECT* dotaz. *SORT UNIQUE* seřadí výsledky výběru. *UNION-ALL* sjednotí dvě množiny neboť dotaz je tvořen ze dvou částí. Následně tu máme dvojici *HASH GROUP BY*, kdy dojde k seskupení na základě hash klíčů.

Teď projdeme část dotazu manipulující s bitvami posádka proti posádce. Posloupnost *HASH JOIN* a *MERGE JOIN* propojí tabulky navzájem. *MERGE JOIN* se hodí pro spojení seřazených výsledků a *HASH JOIN* spojuje dle hash klíčů, které vytváří. *SORT JOIN* při spojování seřazuje řádky. *TABLE ACCES FULL* nám říká, že dochází k postupnému procházení všech řádků tabulky.

Druhá část dotazu manimupluje s větším počtem tabulek neboť je nutné vzít bitvy aliance proti alianci u nich zjistit členské posádky v alianci a až následně počet členů v dané posádce. Je zde spousta *NESTED LOOP*. Tato operace představuje naivní spojování tabulek, kdy jsou postupně procházeny všechny řádky a k nim hledány schody k propojení. Jinak je zde opět vidět *TABLE ACCES FULL* (viz. výše). Nové operace jenž se zde vyskytují jsou *INDEX UNIQUE SCAN* při kterém dochází k prohledávání za pomocí indexu pro jedinečné výsledky a *TABLE ACCESS BY INDEX ROWID*, kde dochází k procházení na základě indexu pro id řádků.

Druhý plán činnosti s indexem:

Optimalizace tohoto dotazu se ukázala jako nad moje síly. Zavádění indexů pro různé hodnoty se ukázalo jako neefektivní. Náročnost dotazu zůstávala víceméně stejná.

```
-----------------------------------------------------------------------------------------------------------------
| Id  | Operation                      | Name                           | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |                                |     3 |   458 |    26  (31)| 00:00:01 |
|   1 |  SORT UNIQUE                   |                                |     3 |   458 |    25  (28)| 00:00:01 |
|   2 |   UNION-ALL                    |                                |       |       |            |          |
|   3 |    HASH GROUP BY               |                                |     2 |   288 |    13  (39)| 00:00:01 |
|   4 |     MERGE JOIN                 |                                |     2 |   288 |    11  (28)| 00:00:01 |
|   5 |      SORT JOIN                 |                                |    77 |  7392 |     7  (29)| 00:00:01 |
|   6 |       MERGE JOIN               |                                |    77 |  7392 |     6  (17)| 00:00:01 |
|   7 |        TABLE ACCESS BY INDEX ROWID| CLENEMPOSADKY                  |    77 |  2387 |     2   (0)| 00:00:01 |
|   8 |         INDEX FULL SCAN        | INDEX_CLENEMPOSADKY_ID_POSADKA |    77 |       |     1   (0)| 00:00:01 |
|*  9 |        SORT JOIN               |                                |     9 |   585 |     4  (25)| 00:00:01 |
|  10 |         TABLE ACCESS FULL      | POSADKA                        |     9 |   585 |     3   (0)| 00:00:01 |
|* 11 |      FILTER                    |                                |       |       |            |          |
|* 12 |       SORT JOIN                |                                |    51 |  2448 |     4  (25)| 00:00:01 |
|  13 |        TABLE ACCESS FULL       | BITVAPOSADKA                   |    51 |  2448 |     3   (0)| 00:00:01 |
|  14 |    HASH GROUP BY               |                                |     1 |   170 |    12  (17)| 00:00:01 |
|  15 |     NESTED LOOPS               |                                |     1 |   170 |    10   (0)| 00:00:01 |
|  16 |      NESTED LOOPS              |                                |     1 |   170 |    10   (0)| 00:00:01 |
|  17 |       NESTED LOOPS             |                                |     1 |   105 |     9   (0)| 00:00:01 |
|  18 |        NESTED LOOPS            |                                |     1 |    79 |     6   (0)| 00:00:01 |
|  19 |         TABLE ACCESS FULL      | BITVAALIANCE                   |     1 |    48 |     3   (0)| 00:00:01 |
|* 20 |         TABLE ACCESS FULL      | CLENEMPOSADKY                  |     1 |    31 |     3   (0)| 00:00:01 |
|* 21 |        TABLE ACCESS FULL       | TVORIALIANCI                   |     7 |   182 |     3   (0)| 00:00:01 |
|* 22 |       INDEX UNIQUE SCAN        | PK_POSADKA                     |     1 |       |     0   (0)| 00:00:01 |
|  23 |      TABLE ACCESS BY INDEX ROWID | POSADKA                        |     1 |    65 |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------------------
```

Závěr

Skript byl psán v editoru Sublime Text 3 a ve Vimu. Spuštěn a testován byl skript v Oracle SQL Developer nad databází na školním serveru s Oracle 12c. Informace jsem převážně čerpal z oficiální dokumentace Oraclu k PL SQL a k SQL Doveloper a taktéž z fóra StackOverflow. I když jsem již s databázemi pracoval v minulosti, tak musím říci, že PL SQL se liší od MySQL v mnoha ohledech.