Dokumentace projektu

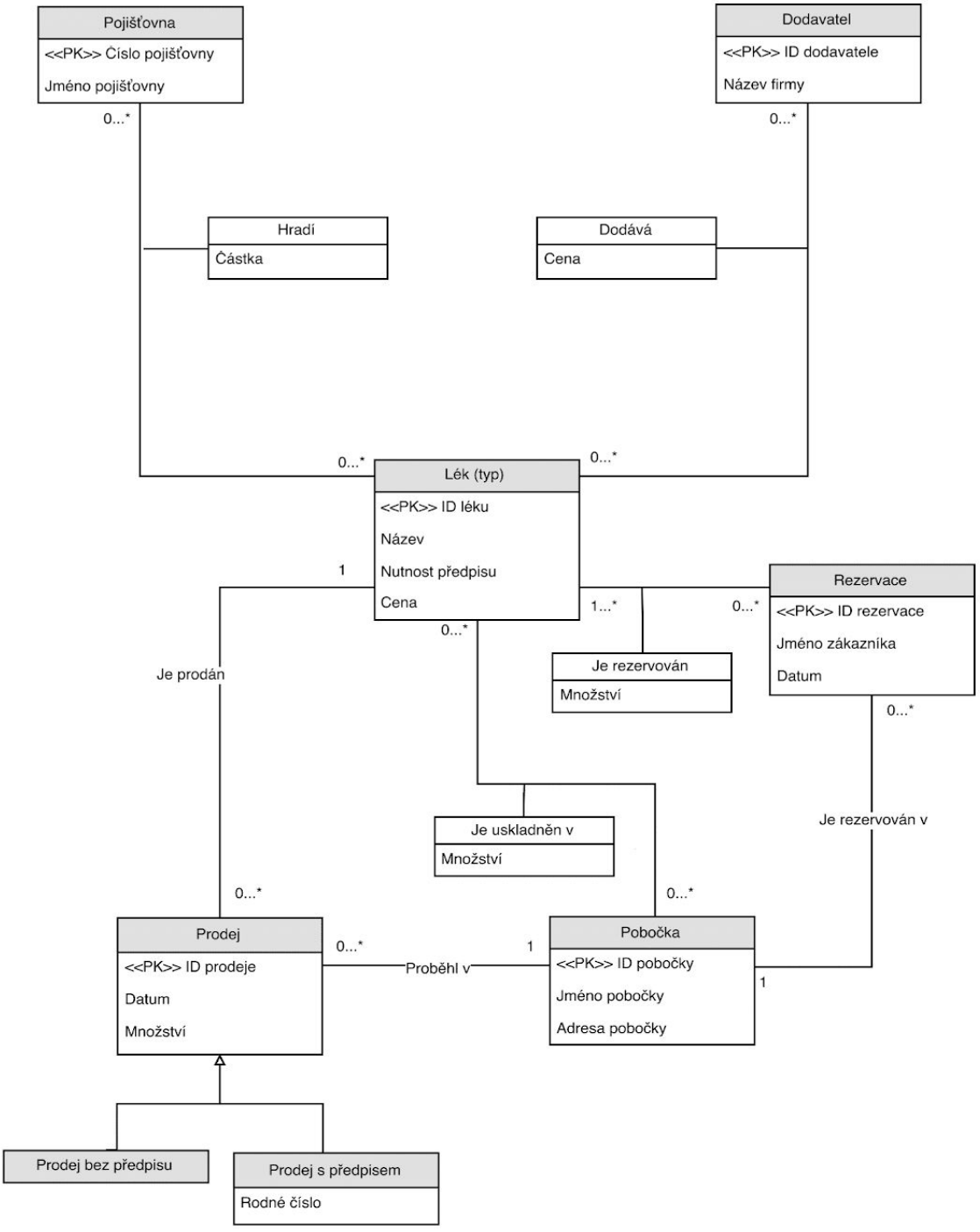
Databázové systémy

Jiří Furda (xfurda00)
Peter Havan (xhavan00)

Fakulta informačních technologií
Vysoké učení technické v Brně

1. května 2018

Návrhový diagram



Popis řešení

V tomto projektu jsme ve dvojici vytvářeli databázový trigger, proceduru, materializovaný pohled a zkoumali plán uskutečnění dotazu.

Popis TRIGGER

V našom riešení sa nachádzajú dva databázové triggre. Trigger *incrementid* je využitý na automatické generovanie a inkrementovanie identifikačného čísla (PK) v tabuľke *dodavatele.* Druhý trigger zaručuje fakt, že žiadna čiastka v tabuľke *hrazeni* nepresiane cenu lieku, ku ktorému sa vzťahuje.

Popis PROCEDURE

Vytvorili sme dve procedúry. Procedúra *percent_predpis* bez parametrov vypíše percentuálne zástupnie liekov, ktoré vyžadajú predpis. Táto procedúra pracuje s výnimkou *zero_divide,* čiže s prípadom, že počet liekov, ktoré vyžadujú predpis je 0, v ktorom prípade sa vypíše 0%. Druhá procedúra s parametrom *jmeno* (lieku) vyhľadá pomocou tabuľky *uskladneni* všetky pobočky, v ktorých sa daný liek nachádza a vypíše ich.

Popis MATERIALIZED VIEW

Pro demonstraci jsme v SQL vytvořili dva pohledy - klasický (*example_view*) a materializovaný (*example_view_materialized*). Následně jsme do tabulky, které se pohledy týkají vložili novou položku. Pokud nyní provedeme dotaz *SELECT* na tyto pohledy, zjistíme, že klasický nově přidanou položku obsahuje, ale materializovaný pohled zůstal nezměněn. Tato nekonzistence dat vznikla tím, že materializovaný pohled při svém stvoření vytvoří novou tabulku, kde uloží data, jež obsahuje. Tyto data se v praxi obnovují periodicky, ale v našem případě jsme obnovení nenastavovali.
Naopak klasický pohled žádnou tabulku fyzicky nevytváří, jedná se jen o jakousi zkratku *SELECT* dotazu, který proběhne vždy, když zavoláme dotaz na tento pohled. Proto také obsahuje aktuální data.
Z této ukázky vyplývá, že materializovaný pohled lze využít v případě, kdy nutně nepotřebujeme aktuální data, ale chceme snížit zatížení databáze.

Popis EXPLAIN PLAN

Z důvodu úspory místa následující tabulky neobsahují sloupec *„Time“* obsahující vždy hodnotu *00:00:01.* Dále ve sloupci *„Operation“* byly mezery značící hiearchii operací pro přehlednost nahrazeny pomlčkami.

Plán dotazu bez použití indexu

Id	Operation	Name	Rows	Bytes	Cost (%CPU)
0	SELECT STATEMENT		2	362	7 (15)
1	-FILTER				7 (15)
2	--HASH GROUP BY		2	362	6 (0)
3	---HASH JOIN		2	362	3 (0)
4	----TABLE ACCESS FULL	PRODEJE	2	52	3 (0)
5	----TABLE ACCESS FULL	LEKY	3	465	3 (0)

Plán dotazu s použitím indexu

Id	Operation	Name	Rows	Bytes	Cost (%CPU)
0	SELECT STATEMENT		2	362	6 (17)
1	-FILTER				
2	--HASH GROUP BY		2	362	6 (17)
3	---NESTED LOOPS		2	362	5 (0)
4	----NESTED LOOPS		2	362	5 (0)
5	-----TABLE ACCESS FULL	PRODEJE	2	52	3 (0)
6	-----INDEX UNIQUE SCAN	EXAMPLE_INDEX	1		0 (0)
7	----TABLE ACCESS BY INDEX ROWID	LEKY	1	155	1 (0)

Popis plánu

Oba způsoby začínají operací *SELECT STATEMENT*, která zaštiťuje celý dotaz typu *SELECT*. Ta obsahuje operaci *FILTER*, která odstraní všechny řádky nesplňující podmínku (v tomto případě *SUM(prodej_mnozstvi) > 1*).

Operace *HASH GROUP BY* zase seskupí položky pomocí tabulky s rozptýlenými položkami. Dále se dotazy liší na základě (ne)využití indexů.

Popis plánu bez využití indexu

Operace *HASH JOIN* opět využívá tabulku s rozptýlenými položkami, nejprve do ni vloží všechny záznamy z první tabulky a poté je propojuje s každým záznamem uvnitř druhé tabulky. K tomu je třeba použít dvakrát funkci *TABLE ACCESS FULL*, která kompletně přečte všechny řádky i sloupce dané tabulky.

Popis plánu s využitím indexu

Oproti tomu dotaz s využítím indexu ke spojení použije dvakrát operaci *NESTED LOOP*, která pro každý řádek první tabulky prohledá všechny řádky druhé tabulky.
První *NESTED LOOP* opět přečte celou tabulku pomocí *TABLE ACCESS FULL*, ale u druhé tabulky využije již vytvořené indexy a proto proběhne operace *INDEX UNIQUE SCAN*, která k hledání indexu využije průchod B-stromem.
Druhý *NESTED LOOP* využije záznamy z předešlého *NESTED LOOP* a s využítím operace *TABLE ACCESS BY INDEX ROWID* již bez dvojího prohledávání získá odpovídající řádek.
Z předchozích tabulek lze vyčíst, že tato varianta je levnější co se týče procesorového času.

Zdroje

https://use-the-index-luke.com/sql/explain-plan/oracle/operations
https://docs.oracle.com/en/
https://www.visual-paradigm.com/VPGallery/datamodeling/EntityRelationshipDiagram.html