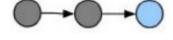
# Datové struktury a patterny

# Persistentní struktury

# Ephemeral DS = struktury, které nejsou persistentní

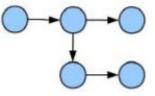
#### Partial persistence DS (= částečná persistence)

- můžeme se dotazovat na jakoukoliv minulou verzi dat
- updatovat lze jen poslední verzi
- podporovány operace:
  - read(var, version)
  - o newversion = write(var, val)
- dělám-li updaty jen do poslední verze, lze uložit jako lineární seznam



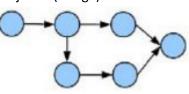
#### Full persistence DS (= plná persistence)

- můžeme se dotazovat na a updatovat jakoukoliv minulou verzi dat
- podporované operace:
  - read(var, version)
  - o newversion = write(var, version, val)
- provedeme-li update do starší verze, musíme udělat novou branch
  - → stromová reprezentace



## Confluent persistence DS (= slévající se persistence)

- jako full PDS, ale navíc umožňuje složení více minulých verzí do jedné (merge)
- podporavané operace:
  - read(var, version)
  - o newversion = write(var, version, val)
  - newversion = combine (var, val, ver1, ver2)
- implikuje uspořádání verzí do acyklického přímého grafu



#### Functional persistence DS (= funkční persistence)

- nody zde jsou immutable revize (updaty) nemění existující nody v DS
- v každém kroku se vytváří klon celé DS, kterou updatuji standardně O(lg n)

## Retroactive DS (= retroaktivní persistence)

- předchozí modely persistence fungují tak, že změna do starší verze vytvoří novou branch
  - o původní větev zůstavá beze změny
- zde je změna do starší verze provedena přímo do ní
- ve všech navazujících verzích se znovu přehrávají provedené operace

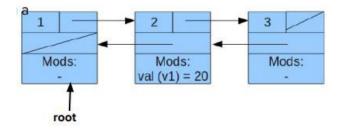
# Partial persistence

- ephemeral DS má následující strukturu:
  - o jsou v ní tedy objekty, které mají property (val1,...,valN) naplněné hodnotami a ukazatele na další objekty (ptr1,...,ptrN)
- do této hodnoty chceme začít dělat změny změna hodnoty property, přelinkování objektu,..
- pro realizaci partial persistence rozšíříme eph. DS o:
  - o **log s modifikacemi (mods)** zapisuje updaty obsahu a ukazatelů ve formě obsahnové-verze a její číslo (field, version, new value)

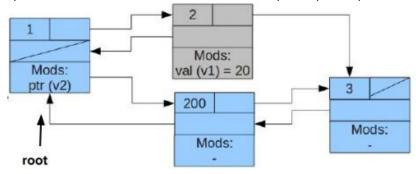
val

ptr

o zpětné reference – pro každý ukazatel v DS zavádí i druhý opačným směrem



- pokud se zaplní mods, vytvoříme nový node a převážeme linky ze starého node do nového
- zpětné ukazatele na starší node nedržíme výhoda partial persistence modelu



- read(var, version)
  - o začneme root nodem a sledujeme ukazatele
  - o vybíráme ukazatel ve verzi nejvyšší: w <= v
  - v každém node vybíráme data ve verzi nejvyšší: w <= v</li>
- write(var, version, val)
  - o provedeme updatem node, se kterým aktuálně pracujeme způsobem výše

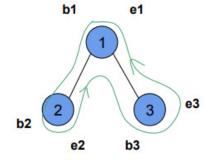
# Full persistence

- realizována na stejných principech jako partial PDS, ale chci-li držet efektivitu algoritmu, musíme vyřešit:
  - verzování zpětných referencí
    - dělám-li update do node, od kterého mám již vytvořený node', potřebuji zpětnou referenci pro všechny verze
    - ... abych věděl, na jakém node převázat přímé reference
    - ... jinak bych musel procházet celou DS a hledat vazby v opačném směru

- o roztržení starých nodes při jejich update
  - dělám-li update do node, od kterého mám již vytvořený node', tak ten má již plný mods log
  - řeším tak, že z daného node udělám dva kusy každý s polovinou záznamů v mods logs a přelinkuji vazby vedoucí na tento node
- o udržování stromu verzí a jeho linearizace
  - už si nevystačíme jene s logováním změn přímo do DS a ukazatelem na root
  - musíme mít reprezentaci, kde:
    - umožníme update do starší v., kdy nám de facto vzniká nová branch
    - budeme zpětně zjišťovat, kdy a v jaké verzi došlo k větvení
    - z každé verze lze provést Backtracking zpětně projít historií updatů, které vedly k mé verzi
- vede v podstatě na strom, kde:
  - každý node = nějaká verze
  - o link mezi dvěma nody = update mezi dvěma verzemi
  - o větvení = udělali jsme update do některé starší verze
- provedeme linearizaci následujícím způsobem:
  - o jelikož každý node reprezentuje okamžik v čase, kdy jsem dělal update, zavedu si index definující okamžik před tímto updatem (b) a okamžik po updatu (e)
  - o strom zlinearizuji jeho in-order průchodem do zápisu:

b1 b2 e2 b3 e3 e1

- což de facto znamená, že jsem provedl update 1 a z něj update 2 a 3
- o chci-li vložit update do node 2:



z toho jasně poznám, že 4. verze bylo provedena jako zpětný update do verze mezi
 b2 a e2 – tedy nová branch v bodě 2

# Lazy loading

- přístup, při kterém odkládáme nahrání objektu až do doby, kdy ho potřebujeme
- používáme, je-li nahrání objektů z repository náročné, zabírají-li objekty hodně paměti apod.
- příklad: Máme list uživatelů, ke každému hodně informací/entit (adresa, objednávky,...).
   Zákazníků mohou být tisíce. 

  Nahrajeme konkrétní entity, až budeme pracovat se daným zákazníkem.

- existují čtyři druhy implementací LL: Virtual proxy; Lazy initialization; Ghost; Value holder

příklady:https://cw.fel.cvut.cz/wiki/ media/courses/b6b36omo/7 - data design patterns 2018.pdf

## Virtual Proxy

- objekt obsahuje místo referencí na konkrétní objekty tzv. virtuální proxy
- ten je dotáhne až na explicitní zavolání

## Lazy initialization

- při prvním přístupu k property objektu se testuje na null
- v kladném případě se nahrává obsah

#### Value holder

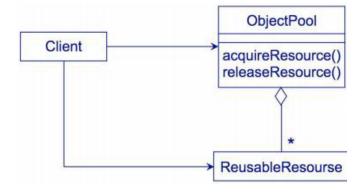
- podobný virtual proxy, ale používá se, když jsou potřeba generika
- VH sám se stará o lazy loading
- k získání hodnoty obaleného objektu je volána metoda getValue()
  - o přistupuje k DB pouze poprvé, pak už jen vrací inicializovaný objekt

#### Ghost

- reálný objekt existuje jen v částečném stavu
- příklad:
  - o nahrávání grafu objektů z DB, kdy místo instanci některých objektů z DB pracujeme jen s jejich identifikátory
  - o plné objekty jsou pomocí ID nahrány ve chvíli, kdy je opravdu potřebujeme
- hojně využíván v ORM (Hibernate,...)

# Object pool

- "zásobárna" přepoužitelných resources, které jsou náročné na vytvoření
- po vytvoření si je ponecháme pro další použití (místo likvidace a opětovného vytváření)
- příklad pooly DB spojení, file handles, proxy externích služeb, náročné java objekty,...
- výrazné zlepšení performance u objektů náročných na vytvoření a masivně používaných
- výhodou také predikovatelný čas a memory footprint při práci se zdroji

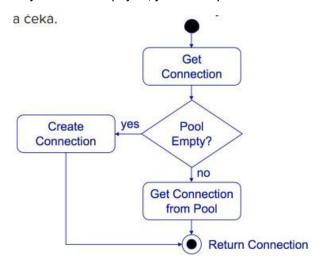


- parametry object pool:
  - o maximální počet resources množství res. je omezeno shora → maximalizace memory footprint

- o minimální počet resources omezeno zdola → resources se vytvoří dopředu, není tedy třeba čekat ani při prvních voláních, kdy ještě nejsou resources vytvořeny
- o doba nepoužívání po které se resources likviduje → zamezuje tomu, aby byly v poolu drženy nepoužívané res.

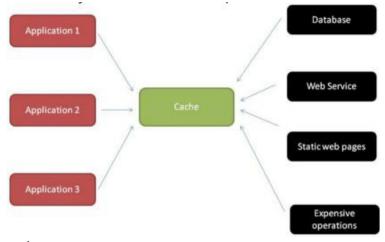
#### - příklad – database connection

- o mechanismus poolingu spojení k databázi v JDBC API
- z pohledu programátora neexistuje rozdíl mezi standartdními a poolovanými spojeními
- o když je spojení uzavřeno, je vráceno do poolu → k dispozici pro opětovné použití
- množství připojení je omezeno shora pro každého klienta (sdílení DB zdrojů)
- nejsou-li volná spojení, je klientský thread blokován a čeká



# Cache

- použijeme, chceme-li načíst resource nebo data najednou, ale používat vícekrát
- princip:
  - o aplikace požaduje data z cache pomocí klíče
  - není-li klíč nalezen, aplikace načte data z externího zdroje dat (pomalé) a vloží do cache
  - o další požadavek na klíč je obsluhován z cache
- cache jsou na různých vrstvách systému, většinou se jich používá několik najednou



- hit/miss ratio
  - = hlavní charakteristika výkonnosti cache

- vysoký poměr = efektivní cache, nízký poměr = malá cache, anebo se cachuji data, která by se cacheovat neměla
- o optimalizace cache, aby se nec. nevhodná data a aby měla optimální velikost

#### cache eviction policy

- algoritmus, jak jsou prvky odebírány / přidávany do cache tak, aby nebyla překročena její kapacita
- o vhodný algoritmus LRU (Last Recently Used)
  - typickou implementací je Map a LinkedList
  - Map drží cachované prvky, které se vybírají podle klíče
  - LL udržuje požadí, v jakém byly prvky naposledy použity
    - znovu čteny / nově přidaný prvek je na začátku seznamu

#### faktory ovlivňující řešení cache:

- o TTL (Time to Live)
- o frekvence čtení/zápisu
- škálovatelnost

# application cache v Java

- cache, ke které aplikace přistupuje napřímo
- o nejčastěji používaná data jsou uložena v paměti
- o základní cache pattern v Java:
  - serializované Java objekty
  - uložení v paměti nebo souboru
  - objekty jsou uloženy jako key-value páry
- o použití pro:
  - statiská data
  - číselníky
  - pseudo-statická data (obsah katalogu,...)
  - kontextově spec. data (uživatelská data,...)

# - cache v ORM toolech (Hibernate apod.)

- L1 cache
  - její scope je Session (uživatelská)
  - unikátní pouze pro daného uživatele (jeho session)
  - v 1. kroku se dotazuje právě L1 cache, až pak další úrovně

#### Query cache

funguje jako L1, jen si místo objektů pamatuje DB dotaz a odpověď

## L2 cache

- její scope je SessionFactory
- sdílená mezi transakcemi a uživateli
- dostaneme se i k datům/dotazům, které provedli ostatní uživatelé
- musí být dost velká a řešit neaktuální data

