VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Hromadné operace vkládání ve vícerozměrných datových strukturách

Bulk-load Operations for Multidimensional Data Structures

2016 Ondřej Prda

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Ondřej Prda**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Hromadné operace vkládaní ve vícerozměrných datových strukturách
Bull-load Operations for Multidimensional Data Structures

Jazyk vypracování: čeština

Zásady pro vypracování:

Vícerozměrné datové struktury jsou využívány v současných databázových systémech nejen pro indexování prostorových dat, ale i pro indexování dat relačních. Vytváření vícerozměrných indexů spočívá nejčastěji v postupném dělení prostoru a vytváření hierarchie takto vytvořených prostorových objektů. Operace hromadného vkládání dat umožňuje dosáhnout řádově lepší výkon vytváření indexu než vkládání po jednotlivých záznamech.

1. Nastudujte vícerozměrné datové struktury a algoritmy pro hromadné vkládání.
2. Naimplementujte vybraný algoritmus hromadného vkládaní.
3. Porovnejte vytvořenou implementaci s klasickým vkládaním po jednotlivých záznamech.

Seznam doporučené odborné literatury:

Podle pokynů vedoucího bakalářské práce.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Peter Chovanec, Ph.D.**

Datum zadání: 01.09.2013

Datum odevzdání: 15.07.2016

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 4. července 2016 .....

Rád bych na tomto místě poděkoval všem, kteří mi s touto prací pomohli, protože bez nich by tato práce nevznikla. Zejména panu Ing. Petrovi Chovancovi za jeho podporu a vedení. Také mým rodičům, kteří mě velice podporovali přes všechno, co se do této doby stalo. Mimo jiné, bych chtěl také poděkovat různým lidem, kteří se nějakým způsobem zasadili o pomoc, při vzniku této práce a za jejich ochotu. Jsou to: Jan Vávra, Tomáš Komárek, Anna Urbánková, Vojtěch Kuchař a Luděk Prda.

Abstrakt

Tato bakalářská práce se věnuje problematice hromadného vkládání vícerozměrných dat do stromových datových struktur, jmenovitě B–stromu a R–stromu. Cílem této práce je navrhnout a naimplementovat algoritmus pro hromadné vkládání dat do výše zmíněných struktur a porovnat rychlost hromadného vkládání s vkládáním po jednotlivých záznamech.

Klíčová slova: B–strom, R–strom, Hromadné vkládaní, Hromadné načítání, Křivky vyplňující prostor

Abstract

This bachelor thesis deals with the issues of multidimensional data bulk loading in tree data structures, namely B-tree and R-tree. The goal of this thesis is to design and implement alghorithm for data bulk loading into data structures mentioned above and to compare bulk loading with the one-by-one insertion.

Key Words: B-tree, R-tree, Bulk insertion, Bulk loading, Space-filling curves

Obsah

Seznam použitých zkratek a symbolů

CPU	–	Central Processing Unit
I/O	–	Input/Output
MBR	–	Minimum Bounding Rectangle - minimální ohraničující obdelník

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

Data, neboli údaje, slouží k popisu nějakého jevu nebo vlastností určitého objektu. Tyto data mohou být jednorozměrná, ale i vícerozměrná, kdy vícerozměrná data jsou kolekce objektů ve vícerozměrném prostoru [2]. Data jako takové jsou reprezentovány pomocí datových typů a tyto datové typy lze dále skládat do datových struktur. Datová struktura není nic jiného, než pomocná struktura, která poskytuje operace nad sama sebou. Datové struktury umožnují lépe a hlavně efektivněji pracovat se samotnými daty a pamětmi, na kterých jsou tyto data uloženy. Tyto operace se mohou lišit v závislosti na typech struktury. Datové struktury se rozdělují na základní (např. pole či spojový seznam) a na pokročilé datové typy (např. graf, hash, strom). Tato bakalářská práce se zaobírá stromovými datovými strukturami a čtyřmi základními operacemi, které lze nad nimi provádět: vkládání, vyhledávání, mazání a aktualizace. K samotným operacím lze přistupovat postupně, kdy zpracovávám každou operaci samostatně, nebo hromadným způsobem. Hromadný způsob představuje provedení více těchto samostatných operací v rámci jedné operace. Mějme například za úkol vložit 200 záznamů do stromové struktury. Místo 200 samostatných operací vkládání je zavolána jenom jedna operace hromadného vkládání, která všech těchto 200 záznamů vloží do dané struktury.

Cílem této práce bylo navrhnout a implementovat algoritmus hromadného vkládání a následně porovnat tuto implementaci s klasickým vkládáním po jednotlivých záznamech. Celá implementace bude vytvořena nad frameworkem RadegastDB, který obsahuje širokou škálu persistentních datových struktur, například B–strom, R–strom, Hashovací tabulka či Sekvenční pole a jiné. Tento framework je vyvíjen výzkumnou skupinou Katedry informatiky, Fakulty elektrotechniky a informatiky, Vysoké školy Báňské - Technické univerzity Ostrava. Framework je implementován v jazyce C++ a momentálně je stále ve vývoji [2]. Způsob implementace hromadného vkládání jsem vybral takový, že testovací kolekce dat jsou nejprve setřízeny dle zvoleného klíče a poté je vytvořena nová stromová struktura a to v *bottom-up* stylu. Tento styl znamená, že stromová datová struktura je vytvořena od listových uzlů postupně, směrem ke kořenu. Jak už jsem zmínil, pro tuto práci jsou důležité hlavně stromové datové struktury a to konkrétně:

- B^+–strom s Lexikografickým třízením
- R–strom s využitím vícerozměrných křivek vyplňujících prostor

V této práci jsou zmíněny i jiné typy vícerozměrných datových struktur, ale jen okrajově. U R–stromu je zmíněn pojem *vícerozměrná křivka vyplňující prostor*. Těchto křivek je poměrné velké množství a některé jsou blíže zmíněny v této práci. Pro testování nad R–stromem byly využity *Hilbertova křivka* a *Mortonova Z křivka*.

2 Vícerozměrné data a datové struktury

2.1 Úvod

Abychom dosáhli efektivnější práce s daty, došlo k vytvoření různých datových struktur. Tyto datové struktury jsou abstraktní datové typy, které jsou nepřímo definované operacemi, které jsou nad datovými strukturami aplikovatelné. Jako příklad bych uvedl Pole, Objekt, Zásobník či Strom. Datové struktury se navzájem liší svými vlastnostmi a tudíž různé datové struktury jsou výhodné pro různé typy úloh, ve kterých jsou použity. Pro databázové využití se často využívají přávě stromové datové struktury pro indexaci dat. Data mohou být jednorozměrné či vícerozměrné. Tyto vícerozměrné data jsou kolekce objektů ve vícerozměrném prostoru. Dají se rozdělit na prostorové data a bodové data. Prostorová data představují souvislý fyzický prostor, jako například čáry, časové intervaly ohraničení různých tvarů. Bodová data mohou představovat bod v prostoru, stejně jako záznamy relačních tabulek v relačních databázových systémech s mnoha atributy, kdy jeden atribut odpovídá jednomu rozměru [2]. K této problematice se váže pojem *n-tice*. Jedná se o bod v n-rozměrném prostoru, kde n udává počet dimenzí daného prostoru. Jako n-tici si můžeme představit záznam v relační tabulce. V závislosti na tom, kolik má daný záznam atributů, tolik má daná n-tice dimenzí. Pokud tedy platí toto, pak platí, že vstupní data – záznamy jsou n-tice o dané dimenzi.

Příklad 1

n-tice o dimenzi 7 : $(2, 3, 6, 4, 10, 2, 4)$

n-tice o dimenzi 4 : $(331, 25, 0, 18092)$ ■

K ukládání dat a vyhledávání v nich, se v dnešních DBMS využívají různé stránkované datové struktury. Tyto datové struktury jsou perzistentní a ukládají data do stránek. Stránka je primárně uložena v sekundárním úložišti, proto je její velikost násobkem velikosti sektoru disku a jednotky souborového systému. Tato velikost je nejčastěji 8 nebo 16kB [2]. V následujících kapitolách jsou popsány dvě tyto strankované datové struktury (jmenovitě B–strom a R–strom), jejich vlastnosti a základní podporované operace.

2.2 B–strom

B–strom [31] je stromová persistentní datová struktura, která slouží k uchovávání dat. Abychom mohli říct o stromové datové struktuře, že se jedná o B–strom, tak musí platit následující vlastnosti:

- Kořen, pokud není listem, tak obsahuje alespoň dva potomky.
- Každý uzel s vyjímkou kořene ma minimálně k a maximálně $2k$ položek, kde k představuje hodnotu jakého řádu je daný strom.

- Jedná se o výškově vyvážený strom, což znamená, že všechny listové uzly mají vždy stejnou vzdálenost ke koření.
- Data v uzlu jsou organizována podle následujících pravidel:
  - p_0, $(k_1, p_1, d_1), ...(k_n, p_n, d_n)$
  - p ... ukazatel na potomky
  - k ... vzestupně (sestupně) uspořádané klíče
  - d ... asociovaná data
  - (k_i, p_i, d_i) ... datové položky
- Je-li $P(p_i)$ podstrom uzlu p_i, pak platí následující:
  - $\forall k \in P(p_i - 1) : k < k_i$
  - $\forall k \in P(p_i) : k > k_i$

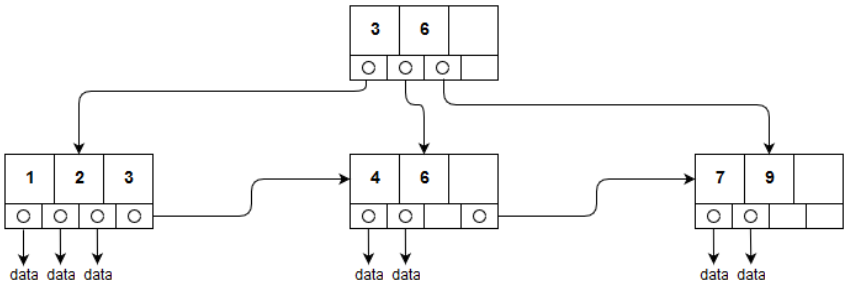
V dnešních databázových systémech se používá modifikace B–stromu, tzv. B^+–strom [25], který se od klasického B–stromu liší v následujících bodech:

- Data jsou obsaženy pouze v listových uzlech.
- Vnitřní uzly obsahují pouze klíče a odkazy na potomky, kde klíče jsou n-tice pomáhající vyhledávání uvnitř stromové struktury.
- Listové uzly obsahují odkaz na následující listový uzel.

Data v této stromové datové struktůře jsou seřazena a díky tomu, B^+–strom dosahuje rychlého vyhledávání záznamů. Základem pro setřízení celé struktury nejsou data, ale klíče, které obsahují jak listy, tak i vnitřní uzly stromu. Operace podporované B^+–stromem jsou:

- Vyhledávaní (angl. *Query Processing*)
- Vkládání (angl. *Insert*)
- Smazání (angl. *Delete*)
- Aktualizace (angl. *Update*)

Operace aktualizace se nad B^+–stromem provádí pomocí použití operace smazání a následné použití operace vkládání. Z tohoto důvodu se k operaci aktualizace u B^+–stromu nebudu více zmiňovat. Časové složitosti těchto operací se pohybují v logaritmickém řádu [27]. Ukázka struktury B^+–strom je zobrazena na obrázku č.1.



Obrázek 1: Ukázka struktury B^+–stromu

2.2.1 Vyhledávání

U vyhledávání nad B^+–stromem dochází k tomu, že se vyhledávaný záznam porovnává s klíči uvnitř kořene. Na základě tohoto porovnávaní je rozhodnuto, do jakého potomka je vyhledávání posunuto. Standardně, pokud hledaný záznam je menší než daný klíč, jedná se o potomka, na kterého je odkazováno na nejbližší levé pozici od daného klíče a naopak, pokud je větší, tak o potomka na nejbližší pravé pozici. Tímto způsobem se prochází celým stromem, až se algoritmus vyhledávání dostane do listového uzlu, kde jsou porovnávány obsažené klíče s vyhledávaným záznamem. V případě, že není nalezen odpovídající klíč k danému záznamu, tento záznam není obsažen v daném stromu. U tohoto typu vyhledávání je časová složitost $O(log_m n)$, kde n je počet záznamů ve stromu, m je řád stromu.

U rozsahového dotazování má B^+–strom výhodu a to takovou, že jednotlivé listové uzly obsahují index – odkaz na následující listový uzel a díky tomu dochází ke snažšímu rozsahovému dotazování. Časová složitost v takovém případě je $O(log_m n + k)$, kde n je počet záznamů ve stromu, m je řád stromu a k je rozsah rozsahového dotazování.

2.2.2 Vkládání

Operace vkládání je postavena na dvou krocích:

- pomocí podobného algoritmu, jako u operace vyhledávání, se nalezne listový uzel, do kterého se měl vkládaný záznam uložit
- pokus o vložení záznamu do listového uzlu

Ve výsledku mohou nastat dvě rozdílné situace, jak se celá struktura zachová při vkládání záznamu do uzlu [4]:

1. Cílený listový uzel má volný prostor pro vložení dalšího záznamu. V takovém případě je vkládaný záznam vložený na určenou pozici v listovém uzlu a nedojde k žádné změně stromové datové struktury.
2. Cílený listový uzel je plný a musí dojít k jeho rozdělení. V takovém případě přibude operace rozdělení listového uzlu, jelikož volné místo na uložení vkládaného záznamu není,

ale rodičovský uzel, daného uzlu, má stále místo, na přidání dalšího potomka. V takovém případě, tedy dochází k vytvoření nového listového uzlu, redistribuci dat mezi novým a starým listovým uzlem tak, aby nedošlo k podtečení uzlu, a následně přídání indexu s klíčovou hodnotou do rodičovského uzlu. Samozřejmě může nastat situace, kdy rodičovský uzel je naplněn a nedovoluje přidání dalšího listového uzlu. V takovém případě dochází k prohledávání stromové struktury, zda-li nějaký rodičovský uzel, vyššího řádu, umožňuje vytvoření splitu cíleného uzlu. Pokud takový uzel není nalezen, je potřeba redistribuovat kořenové prvky (klíč a index) mezi starý a nově vytvořený uzel stejné úrovně, jako byl původní uzel. Poté se vytvoří rodičovský uzel pro tyto dva uzly a jejich nový rodičovský uzel je označen, jako kořenový uzel celého stromu.

2.2.3 Smazání

Mimojiné B^+–strom také nabízí operaci smazání (angl. *Delete*). Nejprve se pomocí vyhledávání lokalizuje cílený uzel a po smazání cílených informací dochází ke kontrole celého stromu, aby nedocházelo k podtečení uzlů. Pokud k tomuto podtečení dojde, přichází na řadu sjednocení uzlů, kde je potřeba kromě samotného sjednocení daných větví stromu, také rekurzivně vymazat rodičovské klíče a ukazatele na daný smazaný uzel.

2.3 R–strom

R–strom [5] je stromová datová struktura, která je podobná stromové datové struktůře B^+–strom. Stejně jako je tomu u B^+–stromu, se jedná o výškově vyvážený strom. Vnitřní uzly obsahují *MBR* a indexový odkaz na své potomky. Listové uzly uchovávají indexový záznam ve formě (MBR, klíč). MBR představuje n-rozměrný obdelník, který slouží jako ohraničující schránka pro indexované vícerozměrné data. Klíč je unikátní identifikátor, který odkazuje na data.

Nechť M je nejvetší možný počet záznamů, které se vejdou do jednoho uzlu a nechť $m \leq \frac{M}{2}$ je parametr specifikující minimální počet záznamů v uzlu. Pro R–strom pak platí tyto vlastnosti:

- Každý listový uzel obsahuje mezi m a M indexovanými záznamy, pokud se nejedná o kořenový uzel
- Pro každý indexový záznam (MBR, klíč) v listovém uzlu, MBR představuje nejmenší obdelník, který prostorově obsahuje n-rozměrné data objektu, který je zastoupen uvedeným klíčem.
- Každý uzel, který není listem, má mezi m a M potomků, pokud se nejedná o kořen.
- Pro každý indexovaný záznam (MBR, klíč) v listovém uzlu, MBR představuje nejmenší obdelník, který prostorově obsahuje obdelníky v uzlu potomka.
- Kořen má nejméně dva potomky, pokud se nejedná o list.

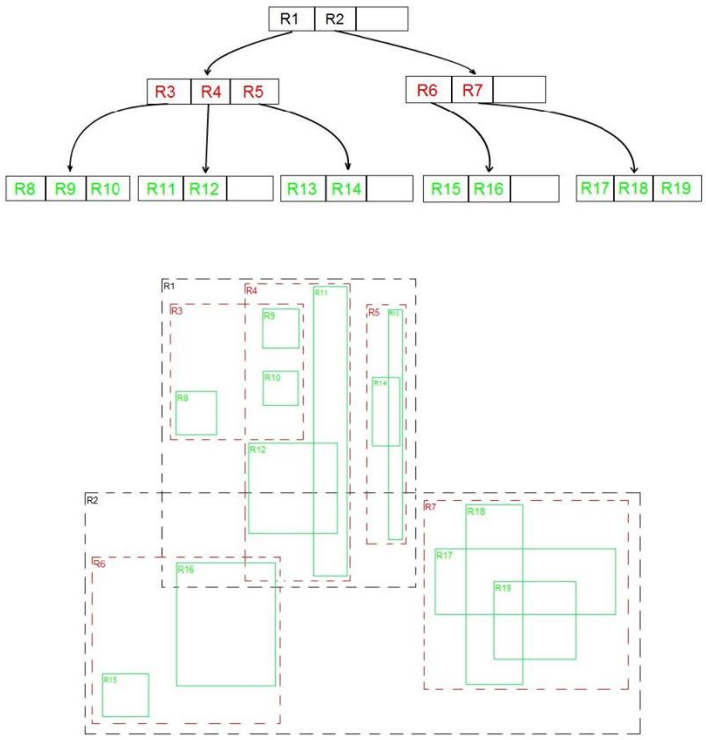
- Všechny listy jsou na stejné úrovni.

Výška R–stromu obsahujícího N indexovaných záznamů je nejvýše $[\log_m N] - 1$.

Maximální počet uzlů je $\left[\frac{N}{m}\right] + \left[\frac{N}{m^2}\right] + ... + 1$. V nejhorším případě se jedná o $\frac{m}{M}$ [5]. Operace podporované R–stromem jsou:

- Vyhledávaní (angl. *Query Processing*)
- Vkládání (angl. *Insert*)
- Smazání (angl. *Delete*)
- Aktualizace (angl. *Update*)

. Ukázka struktury R–strom tvořené pomocí MBR je zobrazena na obrázku č.2



Obrázek 2: Planární reprezentace dat a jejich uložení v datové struktuře R–strom

2.3.1 Vyhledávání v R–stromu

Vyhledávání v R–stromu probíhá stejně jako u ostatních stromových datových struktur a to sestupně – od kořene k listům, dokud se hledaná oblast nenajde nebo neni prohledaná celá

stromová struktura. Stěžejní věc, pomocí které dochází k vyhledávání v R–stromu, je právě MBR. Když dochází k vyhledávání, tak se hledají všechny obdélníky, které překrývají prostor vyhledávacího okna (obdelníku). Vyhledávácí algoritmus zjišťuje, jestli se v daném MBR nachází část, která se překrývá s daným vyhledávajícím obdelníkem. Pokud ano, tak se zavolá opětovné vyhledávání na daný MBR. Pokud je zavolán vyhledávací algoritmus na listový uzel a zároveň je nalezena část, u které dochází k překrytí vyhledavajícího obdelníku a daného MBR, pak se jedná o hledaný záznam. Jak je vidět na obrázku 2 prostory jednotlivých obdélníků se mohou překrývat.

2.3.2 Vkládání do R–stromu

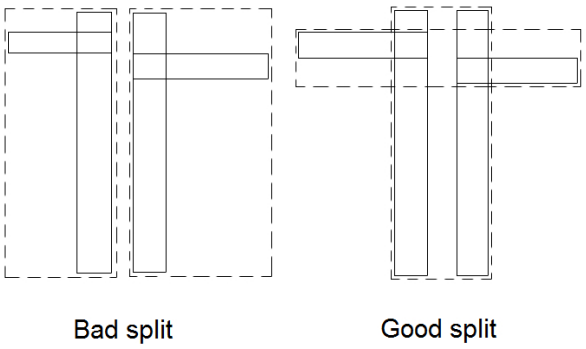
Operace vkládání u R–stromu je taktéž podobná operaci vkládání u B^+–stromu. Vždy se začíná od kořene a postupně se procházejí potomci s tím, že se vybere právě ten, který potřebuje nejmenší rozšíření pro vložení záznamu. Pokud dojde k situaci, kdy je adekvátních více potomků, rozhoduje velikost výsledné plochy uzlu s tím, že se vybere potomek s menší plochou MBR. Pokud vložením do listového uzlu dojde k přetečení daného listového uzlu, pak je nutné rozdělit daný uzel (angl. *split*). U tohoto rozdělení rozhoduje hlavně velikost rozdělené oblasti. Může nastat situace tzv. *"bad split"*, což je rozdělení daného obdelníku na větší 2 obdélníky, než lze provézt. K tomu se váže tzv. *"good split"*, což není nic jiného než rozdělení toho samého obdelníku na 2 části takové, že neexistuje žádné další rozdělení, které by splňovalo, že rozdělená plocha by byla menší než rozdělená plocha u good splitu. Tato situace je znázorněna na obrázku 3.

Na tomto splitování a vytváření co nejméně se překrývajících obdélníků stojí nejen R–strom, ale i jeho variace. Například R^+–strom [**?**], u kterého nedochází k vzájemnému překrývaní ohraničujících obdelníků. Tato vlastnost je nicméně vykoupena tím, že jednotlivé záznamy mohou být obsaženy ve více než jen jednom uzlu a to z důvodu zachování nepřekrývajících se ohraničujících obdelníků. Za zmínku stojí také R*–strom [23], který při porovnávání nebere v potaz pouze velikost datových obdelníku tak, aby byly co nejmenší, jako tomu bylo u R–stromu, ale také se zaměřuje na velikost vzájemného překrývání a velikosti okrajů. Díky těmto dalším parametrům R*–strom překonává ostatní varianty R–stromu. Jediným nedostatkem je vyšší časová složitost algoritmu, který řeší *split*.

Jestliže rozdělením daného uzlu, dojde k přetečení rodičovského uzlu, dochází ke splitu rodičovského uzlu. Tato situace se může opakovat skrze celou stromovou strukturu až ke kořeni. Pokud dojde ke splitu kořene, pak stejně jako u B^+–stromu, dochází k rozdělení kořene na dva vnitřní uzly a vytvoření nového kořene s odkazem na tyto dva vnitřní uzly. S tím souvisí i zvýšení výšky stromu o jeden stupeň.

2.3.3 Mazání z R–stromu

Pokud dojde ke smazání záznamu z uzlu, může dojít ke sloučení dvou uzlů. Tato situace nastává, pokud uzel, ze kterého byl odebrán záznam, obsahuje příliš mnoho volného místa a zároveň



Obrázek 3: Ukázka rozdělení uzlu při jeho naplnění

uzel, se kterým se má spojit, obsahuje dostatek místa k pojmutí všech zaznamů obsažených v rušeném uzlu. Nicméně toto spojování uzlů je u R–stromu nepraktické a místo toho dochází k opětovnému vložení záznamů z rušeného uzlu do stromové struktury. Jelikož algoritmus pro vkládání do stromové struktury je již naimplementován, tak nedochází k navýšení náročnosti implementace a zároveň je zde jistota, že jsou data vložena tak, aby nevznikly větší MBR, než je potřeba. V případě, že je pouze smazán záznam z daného uzlu, je potřeba upravit hodnoty MBR tak, aby odpovídaly nejmenší možné velikosti a to nejen u daného uzlu, ale i patřičných rodičovských uzlů.

2.3.4 Aktualizace R–stromu

Standartní postup při této operaci, je prohledat stromovou strukturu pro daný záznam, poté smazat tento záznam a následně jej znovu vložit. S tímto vložením je spjaté nové vyhledávání optimálního místa pro daný záznam. Jelikož došlo ke změně, je pravděpodobné, že mohlo dojít i ke změně MBR a právě kvůli této změně je nutné smazání a opětovné vložení aktualizované části. Tato operace probíhá od zhora—dolů a musí vždy projít celý strom, než dojde k listovým uzlům. Tyto operace jsou poměrně náročné a tudíž existují i vědecké práce zabývající se aktualizací R–stromu v *bottom-up* stylu [24]. Nejjednodušší způsob je přímá aktualizace v případě, že nedojde ke změně hranic MBR. Tento způsob se při testování neosvědčil jako efektivní (využití pouze u 18% případů). Další možností, je posunutí hranic MBR listových uzlů. Toto navýšení je limitováno hranicemi MBR rodičovského uzlu. V tomto případě však může nastat snížení výkonnosti rychlosti dotazování na stromovou strukturu.

2.4 Další vícerozměrné datové struktury

Mezi další vícerozměrné datové struktury se řadí například UB–strom [28] či BUB–strom [29]. Jedná se o vyvážené stromové datové struktury, které používají křivky vyplňující prostor k rozdělení vícerozměrného prostoru, a také používají B^+–strom jako uložiště [30].

Vícerozměrné datové struktury však nemusí být jen stromové datové struktury. Jako příklad bych uvedl strukturu Grid [32], která používá dekompozici prostoru do mřížky (grid), ve které funguje pravidlo, že body, které se nacházejí velmi blízku u sebe v daném prostoru, tak jsou také uložené blízko sebe v sekundárním uložišti. K vytvoření těchto závislosti slouží hashovací funkce.

Nicméně pro tuto práci jsou stěžejní struktury B^+–strom a R–strom a tudíž se v této práci nebudu více zaobírat dalšími vícerozměrnými datovými strukturami.

3 Bulk loading

Bulk loading (česky Hromadné načítání) je proces, kdy místo postupného vkládání jednotlivých záznamů dojde k před–třízení vstupních dat a následné konstrukci datové struktury. Tento proces se používá u persistentních datových struktur a to nejčastěji u stromové datové struktury. Při tvorbě hromadného načítání lze vycházet z více ověřených algoritmů, které jsou popsány níže. Rozhodující faktor při výběru daných algoritmů je cílená oblast urychlení procesů, popřípadě druh datové stromové struktury, na kterou chceme hromadné načítání aplikovat. Cílenou oblastí pro výběr algoritmů je myšleno, jestli je upřednostňována rychlost procesů dotazování, či je dán důraz na složitost *I/O* operací. Ideální řešení by bylo, kdyby existoval jeden algoritmus, který by poskytoval vysokou rychlost vyhledávání v dané struktůře, zároveň by využíval malý počet *I/O* operací během hromadného načítání a v neposlední řadě by byl implementovatelný na všechny druhy datové stromové struktury. Jelikož však dochází k situacím, kdy daný algoritmus je aplikovatelný pouze na určitou část těchto struktur, vybírá se algoritmus, který je nejpřijatelnější se všemi svými plusy i mínusy. Algoritmy pro zpracovávání hromadného načítání se dají rozdělit na:

- Hromadné načítání pomocí třízení (angl. *sort-based bulk loading*) [11]
- Hromadné načítání pomocí vyrovnávací paměti (angl. *buffer-based bulk loading*) [11]
- Hromadné načítání pomocí vzorků (angl. *sample-based bulk loading*) [11]
- Generické hromadné načítání (angl. *generic bulk loading*) [11]

Mějme proměnné B, N, M, kde B je nejvyšší možný počet záznamů, které mohou být uloženy do jednoho uzlu, N představuje počet operací vložení a M je maximální možný počet záznamů, které je schopna pojmout volná hlavní paměť. Poté bude platit, že N/B je představováno proměnnou n a M/B proměnnou m.

3.1 Hromadné načítání pomocí třízení

Jedná se o algoritmus, který se nejčastěji využívá pro tvorbu kompletně nových B^+–stromů v komerčních databázových systémech. Tento algoritmus je založen na dvou bodech. V první řadě vstupní data setřídíme dle daného klíče a následně vytvoříme kompletně novou datovou stromovou strukturu v tzv. bottom-up stylu. Tento styl popisuje tvorbu datové stromové struktury, kdy jsou nejprve vytvořeny všechny listové uzly a poté se vytváří po úrovních stromu, jednotlivé rodičovské uzly v rekurzivním stylu, až je dosažen stav, kdy je vytvořen kořenový uzel. U této metody je předpoklad, že se do dané struktury bude zasahovat v omezené či téměř žádné formě. Z tohoto důvodu se uzly při vytváření naplňují do téměř maxima své kapacity. Nicméně je doporučeno, aby se nechalo volné místo, pokud víme, že dojde k operacím vkládání po samotném bulk loadingu.

Hromadné načítání pomoci třízení není limitováno pouze na jedno-rozměrné datové struktury, ale s využitím například Hilbertova seřazení tento algoritmus můžeme aplikovat i na prostorové datové struktury jako například R–strom.

Hromadné načítání pomoci třízení se však nemusí nutně držet jen tohoto postupu. Další metoda založena na hromadném načítání pomocí třízení je popsána v [12]. Tato metoda třídí zdrojové data s ohledem na první dimenzi. Poté se vytvoří sousedící oddíly, kde každý oddíl obsahuje téměř stejný počet objektů. V dalším kroku, pak je každý oddíl individuálně setřízen s ohledem na další dimenzi. Stejným způsobem, jako předtím, se vytvoří oddíly o přibližně stejné velikosti. Tento proces se opakuje, dokud nejsou zpracovány všechny dimenze. Výsledné oddíly obsahují nejvíce B objektů. V [12] je ukázáno, že tato metoda hromadného načítání pomoci třízení vytváří takový R–strom, který má výsledné operace vyhledávání rychlejší než metoda popsána výše s využitím Hilbertova seřazení. Každopádně nevýhodou této metody je počet setřízení, který se musí opakovat d-krát, kde d představuje počet dimenzí [11].

3.2 Hromadné načítání pomocí vyrovnávací paměti

Algoritmy založené na tomto principu, využívají externí fronty tzv. *"buffers"*, které jsou propojené s vnitřními uzly dané stromové struktury, kromě kořenového uzlu. Během vkládání záznamu do stromové struktury, dochází k procesu, který by mohl být nazván "dočasné zablokování". Tento proces se řeší ve chvíli, kdy daný záznam dorazí do uzlu. Místo toho, aby záznam procházel stromovou strukturou od kořene dolů k listovému uzlu, záznam je vložen do bufferu. Pokaždé když počet záznamu v bufferu přesáhne předem definovanou hranici (velikostní omezení), dojde k přesunu velkého množství záznamů do další úrovně.Toto rozdělení je řešeno pomocí algoritmu *chooseSubtree*, který vypočte odkaz na podstrom, kde by měla směřovat operace vkládání. Algoritmus *chooseSubtree* pracuje tak, že prochází strom od kořene k listu a na každé jeho úrovni hledá nejpřijatelnější podstrom daného stromu. Během tohoto vyhledávání je kladen důraz na dodržení co nejmenších ohraničujících obdelníků (MBR). Metoda hromadného načítání [13] sestaví celý strom úroveň po úrovni.

Celkový počet I/O operací pro danou metodu je tedy roven $O(n\ log_m n)$, což je asymptoticky rovno nižší hranici externího třízení. Nevýhodou této metody je, že z možné hlavní paměti stránek m se použije pouze $B^{\lfloor log_B m}$]. Jinými slovy algoritmus v nejhorším případě využije pouze m/B hlavní paměti stránek [11].

3.3 Hromadné načítání pomocí vzorků

Postup pro hromadné načítání pomocí vzorků je poměrně odlišný od předcházejících. Tato metoda se například používá pro M-strom [14]. V tomto případě metoda náhodně vybere vzorky objektů ze vstupních údajů. Tyto objekty se nazývají zástupci (angl. representatives) a sestaví z nich strukturu, která je nazývána seeded tree. Do češtiny by se toto dalo přeložit jako nasazený strom. Poté, co je sestavena struktura, se vezmou zbylé záznamy a přiřadí se k jednomu ze zá-

stupců. Pro každého zástupce se opakuje stejný postup jako na začátku. Výsledkem je M-strom, složený z M-stromů. Tato struktura nabízí jisté vlastnosti, jako například plně nenaplněné uzly či nevyváženou strukturu, které porušují vlastnosti původní struktury M-stromu.

Jiný přístup, ale také založený na metodě hromadného načítání pomoci vzorků je popsán v [15], kde kd-stromová struktura je postavena pomocí rychlého externího algoritmu pro spočtení medianu. V tomto případě je vzorek použit pro tvorbu kostry kd-stromu, která je držena jako index ve vnitřním uzlu indexové struktury stejně jako je tomu použito u X-stromu [20]. Tato metoda nicméně závisí na rekurzivním oddělování daných dat na dvě části, stejně jako je tomu u Quicksortu. Toto využití vede k velké naročnosti na I/O, jelikož při práci s daty se často využívá procesů čtení a zápisu.

3.4 Generické hromadné načítání

Tento algoritmus je využíván pro hromadné načítání tzv. *"GrowAndPost−tree"* ve zkratce GP-strom [21]. Jedná se o algoritmus, který pracuje od zhora dolů, kde data jsou rozdělována rekurzivním způsobem do oddílů, dokud oddíl není natolik velký, aby se vlezl do paměti. Tato metoda je aplikovatelná na jakýkoliv druh GP-stromu – například hB–stromy [16].

Pokud se všechny vstupní data vlezou do paměti jedná se o poměrně jednoduchou tvorbu indexu. Požadovaný index se v takovém případě nejprve vytvoří v paměti ,a pak se převede na disk. Jestliže pamět je moc malá, začne se vytvářet paměťový index vkládáním záznamů ze vzorků, dokud není dostupná pamět zaplněna. Poté každému listovému uzlu je přidělen tzv. bucket na disku. Zbylé záznamy jsou pak přiřazeny k bucketům opakovaným voláním metody *chooseSubtree* (metoda popsána v části 3.2), dokud není dosaženo listového uzlu. V této fázi nedochází ke splitu uzlů. Jakmile jsou všechny záznamy zpracovány, tak uzly v hlavní paměti jsou zapsány na disk. Navíc, dvojice neprázdných bucketů a jejich reference na odpovídající listové uzly, jsou zapsány do listu ke zpracování na sekundárním uložišti. V další fázi je vzata dvojice z listu ke zpracování. Odpovídající bucket je považován za zdroj dat a záznamy z daného bucketu jsou zpracovány stejným rekurzivním stylem, jaký již byl popsán výše.

3.5 Shrnutí

V rámci této práce jsem se rozhodl naimplementovat metodu hromadného načítání pomocí třízení. Jelikož je u této metody důležitý prvek setřízení vstupních dat, v následující kapitole budou popsány různé způsoby setřízení dat nad frameworkem RadegastDB. Jelikož u poloviny těchto způsobů dochází k setřízení dat na základě mapování dat pomocí křivek vyplňujících prostor, budou tyto křivky taktéž popsány v následné kapitole.

4 Setřízení dat

Vstupní data, která následně slouží pro tvorbu stromové datové struktury, je nutné setřídit z důvodu ideálního využití stromových datových struktur a jejich operací. Tento proces je součástí bulk loadingu, jak je zmíněno v předchozí kapitole. U B^+–stromu je použito lexikografické setřízení, aby byly dodrženy vlastnosti této struktury. U R–stromu jsou vstupní data seřazeny tak, aby záznamy, které se nacházejí blízko sebe při průchodu vícerozměrných křivek (záleží na volbě algoritmu setřízení), byly i blízko sebe v struktuře R–stromu. Možná setřízení implementována vrámci této práce jsou:

- Lexikografické setřízení
- Setřízení dle Z-seřazení
- Setřízení dle H-sežazení
- Setřízení dle Taxi-seřazení

Tato kapitola se dělí na třízení použité v rámci B^+–stromu (lexikografické setřízení) a na třízení které se mohou aplikovat nad datovou strukturou R–strom. Pro lepší pochopení dané problematiky, jsou mezi těmito setřízeními, vysvětleny křivky vyplňující prostor.

4.1 Setřízení pomocí lexikografické uspořádání

Jedná se o setřízení na základě lexikografické velikosti n-tic. Daný algoritmus porovnává velikosti n-tic pro první dimenzi, pokud dojde k výsledku, kdy výsledkem jsou rozdílné hodnoty, pak n-tice s menší hodnotou je posunutá na přednější pozici než n-tice s vyšší hodnotou. Pokud dojde k výsledku, kdy hodnota obou dimenzí daných n-tic je rovna sama sobě, pak dochází na porovnávání další dimenze a to v rekurzivním stylu. Pokud se narazí na výsledek, kdy obě n-tice jsou stejné, pak ve výsledku nehraje žádnou roli která n-tice bude umístěna na prvním a který na druhém místě, tudíž záleží jen na algoritmu, kterou umístí výše. Toto setřízení se v této práci využívá pro práci s B^+–stromem.

4.2 Setřízení pomocí křivek vyplňujících prostor

4.2.1 Křivky vyplňující prostor

Křivka může být definována, jako obraz jednotkového intervalu $[0, 1]$ pod spojitou funkcí. Křivka vyplňující prostor je pak obraz konkrétního typu funkce, který je surjektivní a jeho rozsah je v rozmezí jednotkového čtverce $[0, 1] \times [0, 1]$, krycdrostor je udán konečným počtem dimenzí. Jednotlivé křivky často mívají při vyšších stupních opakující se vzory sebe sama. Díky čemuž je řadíme mezi fraktály.

4.2.1.1 Historie Počátek křivek vyplňujících prostor se váže ke konci 19. století a připisuje se ke jménu Guiseppe Peano [6]. Tento muž, matematicky vyjádřil a představil pozice bodů v prostoru na trojné bázi. Na jeho práci navázal David Hilbert, který představil první grafické či geometrické zpracování křivek vyplňujících prostor[7]. Tento muž ilustroval koncept dvourozměrného prostoru a použil binární bázi k vyjádření pozic bodů v prostoru [8]. Nyní lze naleznout využití různých křivek například v počitačové grafice, kde pomocí těchto křivek se budují různé tvary, které mají v sobě opakování základního tvaru, stejně jako na daných křivkách lze vypozorovat opakování základního tvaru křivky na vyšších dimenzích.

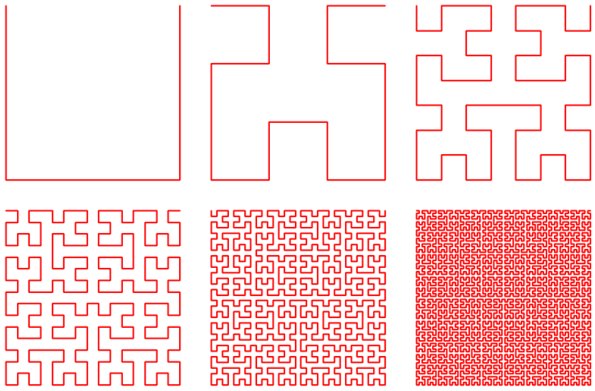
4.2.1.2 Peanova křivka Peano křivka [22] je první známá křivka vyplňující prostor. Jedná se o křivku, vytvořenou Giuseppe Peanem v roce 1890. Peanova křivka je surjektivní, spojitá funkce, která promítá jednotky intervalu na plochu čtverce. V žádném místě se tato křivka neprotíná. Tato křivka používá trojité dělení místo dvojitého. Pod tímto pojmem se skrývá změna výplně prostoru zadaného první iterací křivky. Každá další iterace, kromě té první, rozděluje své předchozí bloky na stejnorozměrná pole o velikosti 3×3. Tudíž při druhé iteraci Peanovy křivky dostáváme 9 polí, do níchž je vložen základní tvar a to buď v nepozměněné formě či transformované formě. Podobně jako *dračí křivka* vyplňuje pouze prostor zadaný první iterací.

4.2.1.3 Mortonova Z-křivka Také známo jako *Z-order křivka* [10]. Jedná se o křivku vyplňující prostor, která udává lineární pořadí průchodu vícerozměrným prostorem. Jinými slovy mapuje vícerozměrný prostor do jednorozměrného. Využití této křivky je v indexování vícerozměrných dat [10]. Jelikož křivku řadíme k fraktálům, jde nalézt na jejích jednotlivých stupních sebeopakování. Tato křivka při každé své další iteraci rozděluje prostor (každý čtverec), nad kterým je zobrazena, na rovnoměrné 2×2 čtverce.

4.2.1.4 Hilbertova křivka Hilbertova křivka [19] je variantou Peanovy křivky. Jedná se o křivku, která prochází každým bodem čtvercové sítě o velikosti 2×2, 4×4 či jiných velikostí daných mocninou 2 a slouží k zápisu prostorových dat. Základním elementem této křivky je zakřivení ve tvaru písmene U. Ať už je toto zakřivení v ostatních iteracích transformováno do jakékoliv pozice – rozumějme natočení do různých směrů – původní tvar zůstává pořád stejný. Toto zakřivení lze vypozorovat i na vyšších stupních této křivky, kde dochází k dělení daného prostoru, který popisuje křivka, pomocí druhé mocniny. U vyšších dimenzí může dojít ke generalizováni Hilbertovy křivky.

V praktickém využití je Hilbertova křivka výkonnější, než Mortonova Z-křivka a tudíž se častěji využívá v práci s prostorovými daty [19]. Časté využití této křivky se například objevuje při budování indexů pro prostorové databáze či v počitačové grafice.

Ibrahim Kamel a Christos Faloutsos popsali ve své práci "Hilbert R-tree: An improved R-tree using fractals"[19] několik způsobů třízení datových obdelníků (*MBR*) v rámci R–stromu za použití Hilbertovy křivky. Jednotlivé způsoby:



Obrázek 4: Různé stupně Hilbertovy křivky [9]

- $2D - c$
- $4D - xy$
- $4D - cd$
- $2Dz - c$

se převážně liší pouze v hodnotách, které slouží k vypočtení hilbertovy hodnoty pro daný datový obdelník. Všechny tyto metody se používají pro statické případy práce s daty – data se skoro nemění a nedochází k častým vkládáním či mazáním dat ze stromu.

4.2.1.4.1 4-d Hilbert through corners ("4D-xy"): Každý datový obdelník je mapován na bod uvnitř čtyř rozměrném prostoru. Tento prostor je tvořen nižším levým a vyšším pravým rohem - konkrétně sestavených ze čtyř hranic (lowx, lowy, highx, highy). Hilbertova hodnota tohoto čtyř-rozměrného bodu je zároveň hilbertovou hodnotou obdelníku.

4.2.1.4.2 4-d Hilbert through center and diameter ("4D-cd"): Každý datový obdelník je namapován na čtyř-rozměrný bod, daný body c_x,c_y,d_x,d_y, kde c_x,c_y jsou souřadnice prostředku datového obdelníku a d_x,d_y jsou souřadnice stran datového obdelníku. Jako v přechozím případě u 4D-xy, hilbertova hodnota tohoto čtyř-rozměrného bodu je hilbertovou hodnotou obdelníku.

4.2.1.4.3 2-d Hilbert through Centers Only ("2D-c"): Každý datový obdelník je reprezentován pouze pomocí svého středu . Hilbertova hodnota daného středu je zároveň hilbertovou hodnotou datového obdelníku.

4.2.1.4.4 Z-order through Centers only ("2Dz-c"): Navzdory tomu, že Z-seřazení poskytuje nižší výkon než Hilbertova křivka, toto seřazení se opírá o vyhodnocování Z-hodnoty. Hodnota datového obdelníku je Z-hodnota středu daného datového obdelníku.

4.2.2 Z-seřazení

Jedná se o seřazení, které slouží jako základ pro *Z-order* křivku. Tato křivka je někdy také nazývána Mortonova Z křivka. Samotné třízení v základě vypočítává normalizované hodnoty jednotlivých dimenzí dvou porovnávajících se záznamů. Tudíž, pro oba záznamy, o stejné dimenzi, platí to, že pro každou dimenzi se vypočítají dané normalizované hodnoty obou záznamů, které jsou poté uloženy do pole hodnot, které má každý záznam zvlášť. Ve výsledku tedy existují dvě pole, která pak slouží k porovnávání uložených hodnot. Pomocí výsledků tohoto porovnávání, pak dochází ke setřízení obou porovnávajících se záznamů. Tabulka č.1 znázorňuje ukázku tohoto seřazení vstupních dat.

pořadí	první polovina	druhá polovina	po sloučení
1.	$1, 1, 1, 13, 2, 4, 2, 3, 1, 12, 0$	$1, 3, 4, 5, 3, 4, 1, 12, 4, 6, 0$	$1, 3, 4, 5, 3, 4, 1, 12, 4, 6, 0$
2.	$1, 4, 3, 13, 2, 13, 2, 1, 3, 6, 1$	$3, 2, 4, 9, 3, 7, 4, 3, 4, 5, 0$	$3, 2, 4, 9, 3, 7, 4, 3, 4, 5, 0$
3.	$1, 9, 4, 6, 1, 4, 3, 2, 3, 9, 1$	$2, 6, 4, 11, 2, 3, 4, 9, 1, 7, 0$	$1, 1, 1, 13, 2, 4, 2, 3, 1, 12, 0$
4.	$3, 12, 3, 2, 3, 11, 4, 5, 2, 5, 1$	$3, 10, 2, 7, 1, 2, 2, 11, 4, 9, 0$	$2, 6, 4, 11, 2, 3, 4, 9, 1, 7, 0$
5.			$1, 4, 3, 13, 2, 13, 2, 1, 3, 6, 1$
6.			$1, 9, 4, 6, 1, 4, 3, 2, 3, 9, 1$
7.			$3, 10, 2, 7, 1, 2, 2, 11, 4, 9, 0$
8.			$3, 12, 3, 2, 3, 11, 4, 5, 2, 5, 1$

Tabulka 1: Z Order seřazení

4.2.3 H-seřazení

Jedná se o Hilbertovo setřízení, které se používá pro Hilbertovu křivku na základě které, se budují prostorové databáze. Využití tohoto seřazení se hlavně aplikuje u budování R–stromu. Podobně jako u Z-seřazení dochází k vypočítání normalizovaných hodnot jednotlivých dimenzí, které jsou pak uloženy do pomocných polí. Rozdíl oproti Z-seřazení je takový, že nedochází přímo k porovnávání normalizovaných hodnot mezi sebou, ale je volána funkce, která obsahuje algoritmus pro porovnávání hodnot. Pokud neexistuje rozdíl mezi dvěmi dimenzemi, vrací se rovnost. V opačném případě se vyhodnocuje rovnice daných parametrů a v závislosti na výsledku je rozhodnuto o vzájemném vztahu těchto dvou dimenzí daných n-tic. Tabulka č.2 znázorňuje ukázku tohoto seřazení vstupních dat.

4.2.4 Taxi-seřazení

Jedná se o setřízení pomocí vypočtení vzdálenosti jednotlivých n-tic od začátku prostoru. V závislosti na výsledné vzdálenosti porovnávaných n-tic se rozhodne, zda se jedná o bližší či vzdálenější n-tici a následně je porovnávána vypočtená hodnota vzdálenosti těchto n-tic, podle toho, jestli se jedná o vzdálenější či bližší n-tici. Tabulka č.3 znázorňuje ukázku tohoto seřazení vstupních dat.

pořadí	první polovina	druhá polovina	po sloučení
1.	$1,9,4,6,1,4,3,2,3,9,1$	$1,3,4,5,3,4,1,12,4,6,0$	$1,3,4,5,3,4,1,12,4,6,0$
2.	$1,1,1,13,2,4,2,3,1,12,0$	$3,10,2,7,1,2,2,11,4,9,0$	$1,9,4,6,1,4,3,2,3,9,1$
3.	$1,4,3,13,2,13,2,1,3,6,1$	$2,6,4,11,2,3,4,9,1,7,0$	$3,10,2,7,1,2,2,11,4,9,0$
4.	$3,12,3,2,3,11,4,5,2,5,1$	$3,2,4,9,3,7,4,3,4,5,0$	$1,1,1,13,2,4,2,3,1,12,0$
5.			$2,6,4,11,2,3,4,9,1,7,0$
6.			$3,2,4,9,3,7,4,3,4,5,0$
7.			$1,4,3,13,2,13,2,1,3,6,1$
8.			$3,12,3,2,3,11,4,5,2,5,1$

Tabulka 2: Hilbertovo třízení

pořadí	první polovina	druhá polovina	po sloučení
1.	$1,1,1,13,2,4,2,3,1,12,0$	$1,3,4,5,3,4,1,12,4,6,0$	$1,1,1,13,2,4,2,3,1,12,0$
2.	$1,9,4,6,1,4,3,2,3,9,1$	$3,2,4,9,3,7,4,3,4,5,0$	$1,3,4,5,3,4,1,12,4,6,0$
3.	$1,4,3,13,2,13,2,1,3,6,1$	$2,6,4,11,2,3,4,9,1,7,0$	$1,9,4,6,1,4,3,2,3,9,1$
4.	$3,12,3,2,3,11,4,5,2,5,1$	$3,10,2,7,1,2,2,11,4,9,0$	$3,2,4,9,3,7,4,3,4,5,0$
5.			$2,6,4,11,2,3,4,9,1,7,0$
6.			$1,4,3,13,2,13,2,1,3,6,1$
7.			$3,10,2,7,1,2,2,11,4,9,0$
8.			$3,12,3,2,3,11,4,5,2,5,1$

Tabulka 3: Taxi seřazení

5 Implementace

Hlavním úkolem této práce je implementace hromadného načítání (*Bulk loading*) pro framework *RadegastDB*. V rámci tohoto frameworku, který je vyvíjený katedrou Informatiky, jsou implementované datové struktury B^+–strom, R–strom, Hashovací mapa či persistentní pole. Bližší informace jsou dostupné na webových stránkách [26]. Postup práce se rozdělil na tři části a to:

- Práce se vstupními daty a jejich příprava, pro samotnou tvorbu stromové datové struktury.
- Vytvoření stromové datové struktury a vložení dat
- Experimentování

Implementace je rozdílná pro B^+–strom a pro R–strom. Z toho plyne odlišný přístup k jednotlivým bodům implementace a různé výsledky experimentů. Při rozhodování, ke které metodě hromadného načítání se přikloním, byla vzata v úvahu jak datová struktura, na kterou se budou algoritmy hromadného načítání aplikovat, tak jejich následná kompatibilita s jinými procesy a v neposlední řadě jejich nároky a rychlost zpracování procesů. Ve výsledku bylo rozhodnuto, že se aplikuje algoritmus, který spadá mezi hromadné načítání pomocí třízení. U R–stromu dochází ke změně ve způsobu třízení oproti B^+–stromu, kdy daný algoritmus, který setřízuje vstupní data, využívá jedno ze tří možných setřízení pro prostorová data.

5.1 Setřízení dat

Implementace setřízení se nachází ve třídě *cBulkLoadArray.h*. Tato třída a její funkce slouží hlavně ke zpracování vstupních dat před tvorbou samotné datové struktury. Tato funkce zaštítuje celkové setřízení a práci se třídou *cBulkLoading.h*. Pro účely třízení je implementována metoda *Sort()*. Uvnitř této metody dochází ke setřízení dat, pomocí třídícího algoritmu *Introsort*. Použitý třídící algoritmus je postaven v základu na třídícím algoritmu *Quicksort*. *Quicksort* je však v této implementaci omezen počtem zanoření do sebe sama. Jinými slovy, existuje daná hranice zanoření, která určuje, kdy se přepne, pro danou třízenou část, třídící algoritmus *Quicksort* na třídící algoritmus *Heapsort*. Toto řešení umožňuje vysoké zrychlení třízení v určitých případech, kdy samotný *Quicksort* je pomalejší. Tyto případy nastávají například, když vstupní data jsou téměř setřízena. I přes zvolení pivota pomocí tzv. metody *median of three partitioning* [17], dochází k situaci, kdy časová složitost *Quicksortu* se pohybuje až na $O(n^2)$, zatímco *Introsort* se v nejhorším případě pohybuje v časové složitosti $O(n \cdot log\ n)$, kde n představuje počet záznamů ke setřízení [18]. Touto volbou se také předchází nestabilitě *Quicksortu*, kdy právě u velkého počtu vnoření dochází k jeho pádu. Níže je uvedena ukázka pro tento třídící algoritmus.

```
input  : uint left, uint right, int nestLimit
int nesting = nestLimit;
nesting++;
if nesting == dana_hodnota then
    if left<right then
        HeapSort(left, right);
    end
end
if left <= right && nesting < dana_hodnota then
    int pivot;
    pivot = Partition(left, right);
    ;                                                    // volba pivota
    if pivot > left then
        Introsort(left, pivot−1, nesting);
    end
    if pivot < right then
        Introsort(pivot+1, right, nesting);
    end
end
nesting−−;
```

Výpis 1: Třídící algoritmus Introsort

5.2 B$^+$–strom

B$^+$–strom jako struktura se vytváří pomocí instancí dvou tříd a to *cQuickDB*, a také pomocí *cBpTreeHeader<cTuple>*.

Pomocí těchto dvou instancí, je pak vytvořena instance stromu. První třída slouží k uskladnění výsledků operací nad stromem. Druhá třída obsahuje informace o stromu, jako například index kořene stromu či výšku stromu. Ve funkci, která vytváří instanci stromu, dochází k vytvoření kořenového uzlu, nastavení odkazů na vedlejší listy pro listový uzel a nastavení hodnot headeru stromu. V headeru stromu jsou uloženy informace o samotném stromu, jako například jeho název, velikost a informace o kořeni. Samotné vytvoření by mohlo vypadat takovýmhle způsobem:

```
BpTree *tree = new BpTree();
if (!tree->Create(mHeader, quickDB))
{
   printf("Critical Error:Creation of Index file failed! \n");
   exit(1);
}
```

Výpis 2: Ukázka vytváření stromové instance

5.2.1 Tvorba stromu

K vytvoření stromové struktury slouží funkce *CreateBpTree()* uvnitř třídy *test.cpp*. Uvnitř této funkce dochází k vytvoření listových uzlů a následně je vnitřně zavolána funkce *CreateBpTreeInnerNodes()* s parametrem leafNodesCount, který předává počet listových uzlů. Tato funkce zaštituje tvorbu vnitřních uzlů a následné vytvoření kořene. Pro práci na listových uzlech, je potřeba si nejprve vytvořit instanci listového uzlu a následně k ní přiřadit funkci ReadNewLeafNode(), která vrací veškeré základní nastavení pro uzel i s alokací paměti cache. Důležitým prvkem je logické nastavení uzlu — jestli se jedná o list nebo o vnitřní uzel. V závislosti na tomto údaji se pak vyhodnocují vyhledávací procesy ve stromu. Zjednodušený systém vytváření uzlů na úrovni listů, by mohl vypadat nějak takhle:

Nejdůležitější atribut, který je využíván v rámci této metody je atribut *mAvgLeafItemsCount*. Tento atribut udává průměrnou hodnotu zaplnění uzlů a v mé implementaci je nastaven na hodnotu 90% tzn. jednotlivé uzly jsou naplněné z 90% jejich velikosti. Nicméně jako první bod této metody se vytvoří instance listového uzlu, vytvoří se odkazy na vedlejší listy (v případě, že se jedná o prvotní listový uzel, jsou tyto hodnoty nastaveny na hodnotu *EMPTY_LINK*). Poté dochází ke kopírování setřízených vstupních záznamů z paměti do daného listového uzlu. Po dosažení hodnoty naplnění uzlu (v našem případě 90%) dochází k uložení n-tice posledního záznamu uvnitř tohoto uzlu a indexu (odkazu) na tento uzel do pomocného pole. Poté je tento uzel uzavřen a dochází k opakování celého procesu do doby, dokud nejsou vloženy všechny vstupní data. V tento moment je dokončeno vytváření listové úrovně stromu a je zavolána metoda *CreateBpTreeInnerNodes*.

```
input : mAvgLeafItemsCount
leafNodesCount = (kapacita uzlu / mAvgLeafItemsCount) + 1;
; // Jednička je přičtena pro zbylé záznamy, které zcela nezaplní daný uzel
na plných 90%
for i ← 0 to leafNodesCount do
    if listový uzel == NULL then
        inicializace nového uzlu pomocí ReadNewLeafNode();
        nastavení linků na oba krajní uzly;
        nastavení linku na tento uzel pro přechozí uzel;
    end
    kopíruj záznamy z pomocného pole do uzlu;
    ;                          // Počet kopírovaných záznamů je omezen hodnotou
    "kapacita_uzlu/průměrný počet záznamů v uzlu"
    ulož klíč posledního záznamu a index uzlu do pomocného pole;
    odzamknutí tohoto uzlu;
    listový uzel = NULL;
end
```

Výpis 3: Tvorba listových uzlů B^+–stromu

Vnitřní uzly se vytvářejí na podobném principu, jako se vytvářejí uzly listové. Vnitřní uzly však neobsahují jednotlivé záznamy (n-tice a data), ale obsahují poslední klíče svých potomků a odkazy na tyto uzly. Pro vytvoření vnitřních listů je volána metoda *CreateBpTreeInnerNodes(uint pItemCount)*, která vytváří jednu úroveň stromu, ve které jsou vnitřní uzly. Při každém průchodu touto metodou dojde k navýšení výšky stromu, vytvoření požadovaného počtu vnitřních uzlů a k jejich naplnění. Práce s vnitřními uzly je velmi podobná práci s uzly listovými.

Na začátku je vypočteno, kolik vnitřních uzlů se celkově vytvoří. Tento výpočet je založen na podílu počtu vložených n-tic v pomocném poli (parametr této funkce) a na hodnotě, která udává průměrný počet itemů (klíč + odkaz) ve vnitřním uzlu. Nyní následují dvě situace, které se mohou naskytnout:

- Počet předaných itemů je menší než průměrná hodnota itemů v jednom vnitřním uzlu
- Počet předaných itemů je větší než průměrná hodnota itemů v jednom vnitřním uzlu

První případ znamená, že se všechny listové uzly vejdou do jednoho vnitřního uzlu, který bude zároveň kořenem tohoto stromu. V takovém případě se vytvoří nový vnitřní uzel pomocí funkce *ReadNewNode()* a následně se do něj nakopírují všechny itemy předány této funkci. Poté se nastaví počet itemů v uzlu, a také se tento uzel nastaví jako kořenový uzel. Pak už jen dojde k odemčení tohoto uzlu a je vše hotovo.

Druhý případ se od toho prvního odlišuje v tom, že se vytváří více než jen jeden vnitřní uzel. V takovém případě jsou rozdíly oproti prvnímu případu v tom, že před odemknutí vnitřního uzlu dojde k uložení posledního vloženého klíče z vnitřního uzlu a odkazu na tento uzel do pomocné struktury, která uchovává dvojice klíč a odkaz na daný uzel pro vytváření další úrovně vnitřních uzlů stromu. Ve chvíli, kdy jsou vytvořeny a naplněny všechny vnitřní uzly dané úrovně je opětovně zavolána tato metoda a jako parametr této metody slouží počet vložených dvojic do pomocné struktury. Konec této rekurze nastává ve chvíli, kdy nastane první situace. Zde je ukázka pseudokódu algoritmu pro vytváření vnitřních uzlů:

```
input  : počet itemů, které se mají vložit
počet vnitřních uzlů = počet itemů / průměrný počet itemů v uzlu;
if počet itemů < průmerný počet itemů v uzlu then
    inicializace nového uzlu;
    kopírování všech předaných itemů do uzlu;
    nastavení počtu itemů v uzlu;
    nastavení uzlu jako root;
    uvolnění uzlu;
end
else
    for i ← 0 to počet vnitřních uzlů do
        inicializace nového uzlu pomocí ReadNewNode();
        if poslední uzel then
            kopírování zbylých itemů do uzlu;
        end
        else
            kopírování průměrného počtu itemů do uzlu;
        end
        nastavení počtu itemů v uzlu;
        přidání posledního klíče a odkazu na tento uzel do pomocné struktury;
        uvolnění tohoto uzlu;
    end
    volání metody CreateBpTreeInnerNodes(počet vložených itemů do pomocné
    struktury);
end
```

Výpis 4: Tvorba vnitřních uzlů metodou CreateBpTreeInnerNodes(počet itemů)

5.3 R–strom

Algoritmus, který zpracovává tvorbu struktury R–strom, vychází z implementace tvorby struktury B^+–strom. Rozdíl mezi těmito dvěmi přístupy je ten, že vnitřní uzly neuchovávají dvojice klíč—odkaz na uzel, ale dvojice klíč—MBR. Klíč je poslední vložený klíč uvnitř daného potomka a MBR je minimální ohraničující obdelník, který má své hodnoty nastavené tak, aby pokrýval vícerozměrný rozsah tohoto potomka v minimální míře. Implementace je rozdělená na dvě části a to tvorbu listových uzlů a tvorbu vnitřních uzlů. Obě tyto části jsou téměř stejné jako u předchozí struktury. Rozdíl je v přidání metody *CreateLeafMbr(uzel)* pro listové uzly a metody *CreateInnerMbr(uzel)* pro vnitřní uzly. Obě tyto metody můžete vidět na následujících ukázkách:

```
// nastaveni low a high klice do daneho MBR
TKey::Copy(TMbr::GetLoTuple(mTmpMbr), pLeafNode->GetCKey(0), mSD);
TKey::Copy(TMbr::GetHiTuple(mTmpMbr, mSD), pLeafNode->GetCKey(0), mSD);

//pro dany pocet itemu v uzlu se modifikuje MBR pomoci klicu ulozenych uvnitr
    uzlu
for (uint i = 1; i < pLeafNode->GetItemCount(); i++)
{
   TMbr::ModifyMbr(TMbr::GetLoTuple(mTmpMbr), TMbr::GetHiTuple(mTmpMbr, mSD),
       pLeafNode->GetCKey(i), mSD);
}
```

Výpis 5: Metoda CreateLeafMbr(pLeafNode)

```
// nastaveni low klice uvnitr MBR
TKey::Copy(TMbr::GetLoTuple(mTmpMbr), pNode->GetCKey(0), mSD);
//nastaveni high klice uvnitr MBR
TKey::Copy(TMbr::GetHiTuple(mTmpMbr, mSD), pNode->GetCKey(0) + mSD->GetSize(),
    mSD);
//modifikovani MBR pomoci vsech ulozenych klicu uvnitr uzlu (krome prvniho)
for (uint i = 1; i < pNode->GetItemCount(); i++)
{
   TMbr::ModifyMbrByMbr(mTmpMbr, pNode->GetCKey(i), mSD);
}
```

Výpis 6: Metoda CreateInnerMbr(pNode)

Jelikož je vytváření listových i vnitřních uzlů velice podobné implementaci nad strukturou B^+–strom, tak na následujícím pseudokódu je ukázáno vytváření pouze vnitřních uzlů nad strukturou R–strom.

```
input  : počet itemů, které se mají vložit
navýšení výšky stromu;
počet vnitřních uzlů = počet itemů / průměrný počet itemů v uzlu;
if počet itemů < průmerný počet itemů v uzlu then
    inicializace nového uzlu;
    kopírování všech předaných itemů do uzlu;
    nastavení počtu itemů v uzlu;
    nastavení uzlu jako root;
    uvolnění uzlu;
end
else
    for i ← 0 to počet vnitřních uzlů do
        nastavení pole pro MBR;
        inicializace nového uzlu pomocí ReadNewNode();
        if poslední uzel then
            kopírování zbylých itemů do uzlu;
        end
        else
            kopírování průměrného počtu itemů do uzlu;
        end
        nastavení počtu itemů v uzlu;
        vytvoření vnitřního MBR pro daný uzel;
        přidání MBR tohoto uzlu a odkazu na tento uzel do pomocné struktury;
        uvolnění tohoto uzlu;
    end
    volání metody CreateRTreeInnerNodes(počet vložených itemů do pomocné struktury);
end
```

Výpis 7: Tvorba vnitřních uzlů struktury R–strom

6 Experimenty

6.1 Úvod

Cílem experimentů je porovnání navrženého přístupu pro *bulk loading*, v sekci implementace, s implementací klasického vkládání po jednotlivých záznamech nad strukturami B^+–strom a R–strom. Vybrané atributy pro porovnávání se dělí na:

- Výsledný čas setřízení a tvorby stromu
- Výsledný čas dotazování
- Počet vnitřních a listových uzlů
- Velikost indexu
- Propustost dotazů
- Logický přistup u dotazování

Mimo jiné byly provedeny dva druhy experimentů. První experiment obsahoval všechny vytvořené uzly stromu v paměti, zatímco u druhého experimentu došlo k naplnění paměti pouze 10 procenty ze všech vytvořených uzlů. K testování došlo na serveru dbsys.cs.vsb.cz přes vzdálený přistup k ploše. Parametry serveru dbsys:

Název počítače	dbsys
Operační systém	Windows Server 2008 R2 Datacenter
Typ systému	64-bitový operační systém
Procesor	2x Intel Xeon X5670 2,93 GHz
Počet jader procesoru	2x6 jader / 2x12 vláken
Paměť RAM	96,0 GB

Tabulka 4: Parametry počítačového serveru dbsys

6.2 Kolekce

V rámci tohoto testování bylo použito pět kolekcí, které se vzájemně lišily a to jak počtem dimenzí, tak počtem záznamů a tím i danou velikostí. Pro první experiment jsou výsledky pro všech pět kolekcí. Pro druhý experiment jsou výsledky nad třemi kolekcemi a to z důvodu časových hodnot tvorby stromu u postupného vkládání. Tyto časové hodnoty se pohybovaly v řádu desítek hodin a to u kolekcí *METEO* a *USA ROADS*. Seznam těchto kolekcí a jejich parametry jsou uvedené zde:

Název kolekce	počet záznamů	počet dimenzí	velikost kolekce[MB]
METEO	57 852 305	5	$1120,4$
POKER	1 000 000	11	$23,9$
TIGER	5 889 786	2	$115,2$
USA ROADS	58 333 337	3	$1319,6$
XMARK	20 532 805	3	$411,7$

Tabulka 5: Kolekce a jejich parametry

6.3 B$^+$–strom

V následujících tabulkách jsou obsažena data z experimentů nad B$^+$–stromem. Nejprve jsou zobrazena data z postupného vkládání po jednom záznamu do předem vytvořené stromové struktury. Velikost *cache* pro toto testování byla nastavena na hodnotu 500 000 možných uzlů v paměti. Logický přístup u dotazovaní je rozdělen na tři části a reprezentován třemi zkratkami a to $\#IN$, $\#LN$ a $\#Rel.LN$. První představuje průměrný počet projitých vnitřních uzlů při dotazování. Druhá počet průměrně projitých listových uzlů při dotazování a poslední představuje průměrný počet relevantních listových uzlů. To znamená uzlů, které skutečně obsahují hledaný záznam.

Název kolekce	METEO	POKER	TIGER	USA ROADS	XMARK
Čas vkládání	$202,711s$	$2,264s$	$7,294s$	$66,746s$	$22,757s$
Čas dotazování	$208,7s$	$2,218s$	$8,29s$	$70,357s$	$23,659s$
Počet vnitřních uzlů	1028	65	27	760	167
Počet listových uzlů	249 417	8 470	12 496	197 128	57 902
Velikost indexu[MB]	$1\,956,6$	$66,68$	$97,84$	1 546	$453,66$
Výkon dotaz.[dotaz/sekundu]	$277\,203,2$	$450\,856,6$	$710\,468,8$	$82\,910,5$	$867\,864,4$
#IN	3.00	2.00	2.00	3.00	2.00
#LN	1.00	1.00	1.00	1.00	1.00
#Rel. LN	0.00	0.00	0.00	0.00	0.00

Tabulka 6: B$^+$–strom postupné vkládání

V druhé tabulce 7 jsou zobrazeny výsledky testu bulk loadingu pro B$^+$–strom za využití lexikografického setřízení.

V poslední tabulce 8 v rámci B$^+$–stromu jsou zobrazeny výsledky pro případ, kdy do paměti lze umístit pouze 10% vytvořených uzlů. Za tímto účelem byla upravena hodnota *cashe* na těchto 10% uzlů z celkového množství vytvořených uzlů. Tato tabulka obsahuje testování nad třemi kolekcemi. Za každým jménem této kolekce je uvedeno písmeno P či B, kdy P značí postupné vkládání a B značí bulk loading. Jelikož se větší polovina atributů rovná předchozím tabulkám, u jednotlivých kolekcí budou zobrazeny pouze atributy — Čas vkládání, Čas dotazování, Výkon dotazování.

Název kolekce	METEO	POKER	TIGER	USA ROADS	XMARK
Čas vkládání	$70,627s$	$1,397s$	$4,951s$	$87,258s$	$19,271s$
Čas dotazování	$206,188s$	$2,355s$	$8,465s$	$77,593s$	$27,83s$
Počet vnitřních uzlů	622	44	17	278	99
Počet listových uzlů	189 061	6 537	9 625	127 089	44 735
Velikost indexu[MB]	1 481,90	51,41	75,33	909,57	350,27
Výkon dotazování[dotaz/sekundu]	280 580,4	424 628,5	695 780,9	751 786,1	737 793,9
#IN	2.00	2.00	2.00	7.17	2.00
#LN	1.00	1.00	1.00	3.08	1.00
#Rel. LN	0.00	0.00	0.00	0.00	0.00

Tabulka 7: B^+–strom hromadné vkládání

Název kolekce	POKER P	POKER B	TIGER P	TIGER B	XMARK P	XMARK B
Čas vkládání	$71,839s$	$2,714s$	$36,928s$	$15,803s$	$106,253s$	$45,568s$
Čas dotazování	$47,316s$	$49,109s$	$39,606s$	$66,785s$	$172,32s$	$109,514s$
Výkon dotaz.[dotaz/s]	21 312	20 362,8	149 025	88 190,3	119 492	187 490,2

Tabulka 8: B^+–strom hromadné vkládání při práci 10% uzlů v paměti

6.4 R-tree testování

U postupného vkládání záznamu do R–stromu je cache pamět pro R–strom v rámci testování nastavena taktéž na hodnotu 500 000 uzlů v paměti. Pravidla pro zkratky u logického přístupu dotazování platí stejně jako u B^+–stromu. Nejprve je zobrazena tabulka 9 která popisuje postupné vkládání, následně tabulka 10, která se věnuje hromadnému vkládání při využití *H-seřazení* dat a jako třetí tabulka 11 s *Z-seřazením* vstupních dat. Úplně nakonec je zobrazena tabulka 12, která má nastavenou hodnotu cashe paměti na velikost 10% vytvořených uzlů. U této tabulky se za názvy kolekcí nachází vždy jeden z tří možných znaků. Tyto znaky P, H, Z reprezentují postupné vkládání, H-seřazení a Z-seřazení.

Název kolekce	METEO	POKER	TIGER	USA ROADS	XMARK
Čas vkládání	$1\ 879,78s$	$29,241s$	$78,435s$	$856,318s$	$247,015s$
Čas dotazování	$3\ 050,17s$	$50,645s$	$64,926s$	$1358,12s$	$144,555$
Počet vnitřních uzlů	2 001	188	48	1 316	296
Počet listových uzlů	246 384	8 465	13 229	206 897	57 714
Velikost indexu[MB]	1 940,51	67,60	103,73	1 626,66	453,08
Výkon dotazování[dotaz/sekundu]	18 966,9	19 745,3	90 715,4	42 951,6	142 041,5
#IN	6.73	8.90	2.30	5.99	3.00
#LN	4.19	6.66	1.18	2.33	1.00
#Rel. LN	1.00	1.00	1.00	1.00	1.00

Tabulka 9: R–strom postupné vkládání

Název kolekce	METEO	POKER	TIGER	USA ROADS	XMARK
Čas vkládání	1 522, 99s	38, 461s	68, 844s	933, 478s	424, 438s
Čas dotazování	1 417, 82s	14, 846s	96, 582s	2 039, 06s	543, 691s
Počet vnitřních uzlů	1 148	87	29	490	173
Počet listových uzlů	189 060	6 536	9 624	127 088	44 734
Velikost indexu[MB]	1 486	51, 74	75, 41	996, 70	350, 84
Výkon dotazování[dotaz/sekundu]	40 803, 7	67 358, 2	60 982, 2	28 607, 9	37 765, 6
#IN	4.34	3.99	3.97	4, 42	2.89
#LN	2.05	1.83	1.30	3, 26	2.66
#Rel. LN	1.00	1.00	1.00	1.00	1.00

Tabulka 10: R–strom hromadné vkládání H-order

Název kolekce	METEO	POKER	TIGER	USA ROADS	XMARK
Čas vkládání	427, 241s	11, 991s	18, 455s	281, 713s	81, 67s
Čas dotazování	2288, 1s	29, 26s	176, 643s	2 043, 61s	168, 256s
Počet vnitřních uzlů	1 148	87	29	490	173
Počet listových uzlů	189 060	6 536	9 624	127 088	44 734
Velikost indexu[MB]	1 486	51, 74	75, 41	996, 70	350, 14
Výkon dotazování[dotaz/sekundu]	25 284	34 176, 3	33 342, 9	28 544, 2	122 033, 1
#IN	4.85	4.99	3.34	4, 67	2.00
#LN	3.81	5.00	2.13	3, 23	1.00
#Rel. LN	1.00	1.00	1.00	1.00	1.00

Tabulka 11: R–strom hromadné vkládání Z-order

Název kolekce	Čas vkládání	Čas dotazování	Výkon dotaz.[dotaz/s]
POKER P	93, 96s	315, 685s	3 167, 7
POKER H	41, 091s	109, 342s	9 145, 6
POKER Z	12, 948s	183, 567s	5 447, 6
TIGER P	82, 832s	70, 235s	83 858, 3
TIGER H	73, 516s	101, 498s	58 028, 6
TIGER Z	19, 702s	183, 008s	32 183, 2
XMARK P	338, 509s	293, 019s	70 073
XMARK H	489, 051s	695, 3s	29 530, 9
XMARK Z	126, 003s	255, 22s	80 451

Tabulka 12: R–strom hromadné vkládání při práci 10% uzlů v paměti

7 Závěr

7.1 Výsledky experimentů

Výsledky experimentů u B^+–stromu potvrdily můj předpoklad a to ten, že díky většímu využití jednotlivých uzlů, došlo k výraznému poklesu velikosti indexu daného stromu. Tato velikost je navíc menší, než velikosti B^+–stromů, které prošly kompresí. Tyto kladné vlastnosti jsou však vyvažovány tím, že je zapotřebí setřídit vstupní data. Toto setřízení se v některých případech ukázalo náročnější na čas, než samotné vytvoření nového stromu pomocí postupného vkládání. Tento jev je nicméně zapříčiněn tím, že hodnota *cashe* byla nastavena tak, aby se všechmi uzly bylo pracováno v rámci paměti, a tudíž aby nedocházelo k přístupu na disk. Jako důkaz může sloužit to, že když jsem se pokoušel o vytvoření stromové struktury pomocí postupného vkládání a měl jsem hodnotu *cashe* nastavenou tak, aby se do paměti vešlo jen 10% z vytvořených uzlů, tak odhadovaný čas potřebný pro tvorbu datové struktury se pohyboval v rozmezí desítek hodin, zatímco u bulk loadingu v rozmezí desítek až stovek minut. Tím, že v mé implementaci dochází vždy u bulk loadingu ke setřízení vstupních dat, daná kolekce svým setřízením a velikostí má dopad jak na rychlost tvorby datové stromové struktury, tak i na dotazování. Tento jev je viditelný u kolekce *XMARK* u *H-seřazení*, kdy se časy pro tuto kolekci vymykají ostatním výsledkům jiných kolekcí.

R–strom a jeho experimenty proběhly v podobném duchu, kdy platí výše zmíněné vlastnosti experimentů. Opět dochází ke zmenšení velikosti indexu stromu a s tím i menšímu počtu listových i vnitřních uzlů. Testy pomocí různých setřízení potvrdily vlastnosti třízení při použití Hilbertovy křivky a Mortonovy křivky, kdy H-seřazení sice delší dobu třídí data, ale dosahuje lepších výsledků v dotazování. Oproti tomu u Z-seřazení dochází k tomu, že rychlost setřízení je vyšší, ale rychlost dotazování nad hotovou strukturou je pomalejší.

Vzhledem k tomu, že server *dbsys* byl používán více lidmi najednou během testování je právděpodobné, že časové výsledky se mohou lehce pohybovat nahoru či dolů v závislosti na aktuálním využití. Nicméně server se nedostal k takovému vytížení, aby to mělo vážný dopad na samotné testování. Dalším faktorem ovlivňujícím výsledky testů je správné určení velikosti cache paměti u postupného vkládání. V závislosti na tom, jak moc uzlů bude obsaženo v paměti, záleží na kolik se projeví výhoda bulk loadingu oproti postupnému vkládání. Těmito testy bylo ukázáno, že u menších kolekcí, nedochází k takovým výsledkům, které by plně zvýhodňovaly bulk loadingu, co se týče tvorby struktury a dotazování na strukturu. Nicméně u velkých kolekcí docházelo k výrazně lepším výsledkům použité metody bulk loadingu. V neposlední řadě ve všech případech bylo dosaženo menších hodnot indexů a počtu vytvořených uzlů.

Možnosti k dalšímu vývoji či zdokonalení bych viděl především ve výběru a otestování nejlepšího třídícího algoritmu pro danou kolekci. Takový algoritmus by dosahoval nejlepší možné časové složitosti a výsledků pro danou kolekci. V této práci jsem aplikoval jeden algoritmus, který je schopen stabilně setřídit všechny zpracované kolekce, nicméně nelze aplikovat jeden algoritmus,

který bude mít nejlepší výsledky pro všechny typy kolekcí — aspoň o takovém nevím.

Další bod leží ve využití vícerozměrných křivek vyplňujících prostor. Jak jsem zmínil v kapitole, která se jim věnuje, existuje více možností, jak díky nim vypočítávat vzdálenost n-tic ve vícerozměrném prostoru. Tyto změny by měly mít dopad jak na čas třízení, tak i na čas dotazování nad stromovou strukturou.

Toto bylo mé první setkání s větším frameworkem a s tím i související práce v něm. Myslím, že práce na této bakalářské práci mi byla velmi přínosná a to nejen co se týče stránky programátorské, kde došlo k výraznému posunu mých schopností, ale i stránky osobní, kde jsem zdokonalil svou trpělivost při hledání řešení pro opakující se problémy či u zdlouhavého testování. Díky této práci jsem si rozšířil znalosti, o dle mého velmi přínosné věci, jako například práce s pamětí, návrhy a testování algoritmu pro třízení či samotné fraktály v podobě křivek vyplňujících prostor, které jak doufám v budoucnu využiji.

Literatura

[1] L.Arge, *Efficient External-Memory Data Structures and Applications*, BRICS Dissertation Series, DS-96-3, University of Aarhus 1996

[2] M. Smolka, *Webové rozhraní pro testování frameworku RadegastDB*, Bakalářská práce, Vysoká škola Báňská - Technická univerzita Ostrava, 2016

[3] J. Bercken, B. Seeger, P. Widmayer *A Generic Approach to Bulk Loading Multidimensional Index Structures*, In Proc. International Conf. on Very Large Databases, 1997

[4] Miller Lab @ University of Toronto *Database System Technology CSC443*, [online], [cit. 2015-04-26], Fall 2014. Dostupné z:

http://dblab.cs.toronto.edu/courses/443/2014/05.btree-index.html

[5] A. Guttman, *R-Trees: A Dynamic Index Structures for Spatial Searching*, In Proceedings of Annual Meeting. Boston(MA), ACM Press, 1984, Dostupné z:

http://www-db.deis.unibo.it/courses/SI-LS/papers/Gut84.pdf

[6] Giuseppe Peano, Sur une courbe, qui remplit toute une aire plane (on a curve which completely fills a planar region). *Mathematishe Annalen* , 36:157-160, 1890

[7] David Hilbert, *Matematishe Annalen*, Ueber stetige abbildung einer linie auf ein flachenstuck, 38:459-460, 1891

[8] Janathan Lawder, *The Application of Space-filling Curves to the Storage and Retrieval of Multi-dimensional Data*, Thesis for degree of Doctor of Philosophy, University of London, 1999, Dostupné z: http://www.dcs.bbk.ac.uk/ jkl/thesis.pdf

[9] Hilbert Curves, *degrees of Hilbert Curve* [cit. 2015-04-27], Obrázek ve formátu PNG. Dostupné z: http://www.datagenetics.com/blog/march22013/

[10] Mortonův rozklad. In: *Wikipedia: the free encyclopedia*[online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2015-04-27]. Dostupné z: http://cs.wikipedia.org/wiki/

[11] J. Van den Bercken, B. Seeger, *An Evaluation of Generic Bulk Loading Techniques*, In Proceedings of the Twenty-seventh International Conference on Very Large Data Bases, Roma, 11-14th September 2001, Dostupné z: http://www.vldb.org/conf/2001/P461.pdf

[12] S. T. Leutenegger, M. A. Lopez: STR: *A Simple and Efficient Algorithm for R-Tree Packing.*, ICDE 1997: 497-506

[13] J. van den Bercken, B. Seeger, P. Widmayer: *A Generic Approach to Bulk Loading Multidimensional Index Structures.*, VLDB: 1997: 406-415

[14] P. Ciaccia, M.Patella: *Bulk loading the M-tree*, Proc. of the 9th Australian Database Conference, pp. 15-26, 1998

[15] S. Berchtold, C. Böhm, H.-P. Kriegel: *Improving the Query Performance of High-Dimensional Index Structures by Bulk-Load Operations*, EDBT 1998: 216-230

[16] D. B. Lomet, B. Salzberg: *The hB-Tree: A Multiattribute Indexing Method with Good Guaranteed Performance.* TODS 15(4): 625-658 (1990)

[17] Robert Sedgewick, Kevin Wayne: *Quicksort* Dostupné z: http://algs4.cs.princeton.edu/23quicksort/

[18] Juerg Moser, Kyle Ross, WuiChia Su: *Introsort Algorithm concepts* 2003 Dostupné z: http://www.cs.rpi.edu/ musser/gp/algorithm-concepts/introsort-screen.pdf

[19] Ibrahim Kamel, Christos Faloutsos: *Hilbert R-tree: An improved R-tree using fractals* Technical Research Report, University of Maryland, February 18, 1993

[20] Alvin M. Despain, David A. Patterson> *X-Tree: A tree structured multi-processor computer architecture*, Proceeding ISCA '78 Proceedings of the 5th annual symposium on Computer architecture, Pages 144-151

[21] David B. Lomet: *Grow and post index trees: Role, techniques and future potential*, Lecture Notes in Computer Science Volume 525, 1991, pp 181-206, Jun 2005

[22] G. Peano: *Sur une courbe, qui remplit une aire plane"*, Math. Ann. 36, 157-160, 1890

[23] N. Beckmann, Hans-Peter Kriegel, R. Schneider, B. Seeger: *The R*-tree: an efficient and robust access method for points and rectangles*Proceeding SIGMOD '90 Proceedings of the 1990 ACM SIGMOD international conference on Management of data, Pages 322-331, New York, 1990

[24] M. L. Lee, W. Hsu, C. S. Jensen, B. Cui, K. L. Teo: *Supporting Frequent Updates in R-Trees: A Bottom-Up Approach* Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003

[25] Chu Jiang-Hsing, Gary D Knott: *An analysis of B-trees and their variants*, Department of Computer Science, University of Maryland, College Park, 31 May 1988

[26] Web pages of Database Research Group at Department of Computer Science, Faculty of Electrical Engineering and Computer Science, Technical University of Ostrava, Czech Republic. Dostupné na: http://db.cs.vsb.cz

[27] COMER, Douglas: *Ubiquitous B-tree*, ACM Computing Surveys (CSUR), 1979, 11.2: 121-137

[28] R. Bayer: *The Universal B-Tree for Multidimensional Indexing: General Concepts* , In Proceedings of the International Conference on Worldwide Computing and Its Applications (WWCA). Springer-Verlag, 1997.

[29] R. Fenk: *The BUB-Tree*, In Proceedings of the 28th International Conference on Very Large Data Bases (VLDB). Morgan Kaufmann, 2002.

[30] Ing. Chovanec Peter: *Reduction of Disk Accesses in multidimensional data structures*, Dizertační práce, Vysoká škola Báňská - Technická Univerzita Ostrava, 2015.

[31] R. Bayer, E. M. McCreight: *Organization and Maintenance of Large Ordered Indices.* Acta Informatica, pages 173-189, 1972.

[32] J. Nievergelt, H. Hinterberger, and K. C. Sevcik: *The Grid File: An Adaptable, Symmetric Multikey File Structure.* ACM Trans. Database Syst., 9(1):38-71, Mar. 1984.

[33] SELLIS, Timos; ROUSSOPOULOS, Nick; FALOUTSOS, Christos: *The R+-Tree: A Dynamic Index for Multi-Dimensional Objects.* 1987.

A Obsah CD

Přiložené CD obsahuje adresář *bakalářská práce*, který obsahuje:

- Adresář s názvem Implementace
- Adresář s názvem Kolekce
- Adresář s názvem Text

V adresáři *Implementace*, se nachází soubory s třídami frameworku a podadresáře *test/paged/b-tree_test* a *test/paged/rtree_test*, které obsahují spouštěcí soubory k jednotlivým implementacím. V obou těchto adresářích existují soubory *Config.txt*, pomocí kterých lze nastavovat základní parametry testování před spuštením. Prosím o dodržení stylu, ve kterém jsou napsány. Uvnitř těchto souborů je několik atributů:

- Vstup - cesta ke vstupní kolekci
- Item count - počet záznamů v dané kolekci
- Cashe size - počet uzlů, které se vejdou do paměti
- Dimension - dimenze záznamů v kolekci
- Bulk loading - 1 pro metodu bulk loading, cokoliv jiného pro postupné vkládání
- SortType - atribut u R–stromu, 0 - Z-seřazení, 1 - H-seřazení

Uvnitř adresáře *Kolekce* se nachází jedna testovací kolekce, ke které je nutno nastavit cestu v konfiguračním souboru. V adresáři *Text* se nachází text této bakalářské práce ve formátu PDF.