VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Metody zamykání v B-stromu
Locking methods in B-Tree

2014 Tomáš Vacho

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Tomáš Vacho**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Metody zamykání v B-stromu
Locking Methods in B-Tree

Zásady pro vypracování:

Při práci s datovými strukturami v databázových systémech dochází ke konkurenčním přístupům k uzlům těchto datových struktur. Cílem práce je nastudovat a implementovat zvolenou metodu zamykání v datové struktuře B-strom.

1. Nastudujte metody zamykání v B-stromu.
2. Implementujte zvolenou metodu zamykání pro B-strom.
3. Proveďte testování a porovnejte zvolené metody zamykání.

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. Václav Bašniar**

Datum zadání: 01.09.2013
Datum odevzdání: 07.05.2014

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava.*

V Ostravě 24. dubna 2014

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

V Ostravě 24. dubna 2014

Děkuji vedoucímu této bakalářské práce p. Bašniarovi za pevné nervy a nekonečnou trpělivost, kterou musel při vedení mé práce vynaložit. Dále děkuji rodině a přítelkyni za psychickou podporu. Děkuji také spolužákům z kombinované formy studia za soudržnost a konzultaci vybraných témat.

Abstrakt

Při práci s datovými strukturami v databázových systémech dochází ke konkurenčním přístupům k uzlům těchto datových struktur. Cílem této práce je nastudovat a implementovat zvolenou metodu zamykání v datové struktuře B-strom.

Klíčová slova: B-strom, konkurenční přístup, zamykání

Abstract

When working with data structures in database systems is competing approaches to the nodes of these data structures. The goals of this work is study and implement selected technique of locking in the B-tree data structure.

Keywords: B-tree, concurrency access, locking

Seznam použitých zkratek a symbolů

ACID	–	Akronym pro vlastnosti databázové transakce
B-strom	–	Stromová datová struktura
C++	–	Programovací jazyk
COMMIT	–	Příkaz, který ukončí databázovou transakci s uložením výsledků modifikací
Deadlock	–	Uváznutí
Inner Node	–	Vnitřní uzel
Leaf Node	–	Listový uzel
Node	–	Uzel
OOP	–	Objektově orientovaný přístup
QuickDB	–	Implementace SŘBD vytvořená na katedře informatiky VŠB-TUO FEI
ReadLock	–	Sdílený zámek
ROLLBACK	–	Operace, která vrací databázi do nějakého předchozího stavu
Root	–	Kořen
Split	–	Rozdělení naplněného uzlu na dva nové
SŘBD	–	Systém řízení báze dat
Transakce	–	Proces změny stavu
WriteLock	–	Výlučný zámek

Obsah

1 Úvod **5**

2 Teorie problematiky **6**
2.1 B-strom 6
2.2 Konkurenční přístup obecně 9
2.3 Paralelismus v datové struktuře B-strom 13

3 Implementace vybrané metody zamykání funkce pro vkládání do B-stromu **18**
3.1 Konkrétní způsob zamykání v metodě Insert 22

4 Vyhodnocení implementace a testování **28**
4.1 Porovnání dosažené propustnosti s jednovláknovým vkládáním 28

5 Závěr **30**

6 Reference **31**

Přílohy **31**

A Vývojové diagramy **32**

B Obsah CD **33**

Seznam tabulek

1 Ztráta aktulizace: Aktualizace transakce T_1 je ztracena v čase t_4. 10
2 Dočasná aktualizace: Transakce T_1 se stala v čase t_2 závislou na nepotvrzené změně transakce T_2. 10
3 Neopakovatelné čtení: Transakce T_1 přečetla hodnotu t v čase t_1 a t_3, přičemž ve druhém případě byl vrácen jiný výsledek, než v tom prvním. . 11
4 Fantóm: Transakce T_1 obdržela množinu záznamů t v čase t_1 a t_3, přičemž ve druhém případě byl vrácen jiný počet záznamů, než v tom prvním. . . 11
5 Tabulka kompatibility základních způsobů uzamknutí objektu. 12
6 Uváznutí. 12
7 Matice kompatibilty záměrných zámků. 13
8 Výsledky testování vkládání do B-stromu jednovláknovým a dvouvláknovým průběhem 28

Seznam obrázků

1 B-strom 7
2 Vkládání hodnoty 7 do B-stromu 8
3 Nejhorší případ, který může nastat - splity v hierarchii B-stromu až do kořene 17
4 Výstup výsledné aplikace - přeházené pořadí čísel označujících počet aktuálně vložených hodnot dokazuje, že vkládání bylo provedeno vícevláknově 28
5 Vývojový diagram části vkládacího cyklu nazvané State0 (v červeném rámečku vyznačeny operace s listovým uzlem) 32

Seznam výpisů zdrojového kódu

1 Atributy a metody přidané do třídy cNode 18
2 Definice metod přidaných do třídy cNode 18
3 Deklarace lokálních proměnných přidaných do metody Insert třídy cCommonB+Tree 20
4 Naplnění polí hodnotami NULL 20
5 Příklad pokusu o získání ReadLocku na listový uzel 20
6 Odemknutí dotčených uzlů po skončení iterace vkládacího cyklu 21
7 Kód rozdělující vkládání prvků mezi jednotlivá vlákna 21
8 Funkce insertTuples 22
9 Metody pro zamykání celého B-stromu pro situaci při splitu přidané do třídy cCommonBpTree 23
10 Detekce rezervace nebo zamknutí celého B-stromu 25
11 Rezervace a následné zamknutí celého B-stromu 25
12 Příklad detekce uváznutí deseti neúspěšnými pokusy o získání zámku, případné odstranění dosud získaných zámků a následný ROLLBACK ve formě přesměrování zpět na začátek metody pomocí návěsti 26

1 Úvod

Fyzická implementace uložení dat je jedním z podstatných problémů databázových technologií, zajímáme se především o složitost operací vyhledání, vkládání, mazání a modifikace. V této práci budeme využívat stromovou datovou strukturu B-strom, která poskytuje dobrou složitost pro výše uvedené základní operace a je přirozeně perzistentní.

Při vývoji moderních aplikací musí být počítáno s víceuživatelským přístupem, tedy s konkurenčními přístupy k položkám v datových strukturách. Je nutné, aby datová struktura zůstala po dokončení operací konzistentní a aby se souběžné transakce navzájem neovlivňovaly. Také je důležité dbát na to, aby aplikace zůstala i po ošetření tzv. vláknovou bezpečností dostatečně efektivní, tedy aby měla dostatečnou propustnost.

Toho je dosaženo různými tzv. synchronizačními mechanismy. Jedním z nich, kterému se v této práci budeme věnovat, je zamykání. Zamykat budeme různě velké granule - od jediného uzlu, po celý strom. Zamykání bude prováděno prostřednictvím doimplementovaných metod v již hotovém školním kódu zajištujícím datovou strukturu B-strom. Konkrétní implementace bude vycházet v lastního návrhu techniky zamykání, ale popíšu i jiné.

Se zamykáním je spojeno několik problémů, které budu v této práci řešit - zejména jde o problém uváznutí, kdy dvě vlákna na sebe vzájemně čekají; a problém tzv. splitu, který při přetečení kapacity dotyčného uzlu tento uzel rozdělí na dva nové.

Cílem práce je tedy popsat a naimplementovat v C++ vybranou metodu zamykání v B-stromu, otestovat její funkčnost a získané výsledky porovnat. Implementace je řešena na školním frameworku QuickDB vyvíjeného na katedře informatiky VŠB-TUO FEI. Framework je schopen strukturu vytvořit a provádět nad ní operace. Jednou z těchto datových struktur je právě i B-strom.

Druhá část této práce obsahuje teoretickou stránku této problematiky. Je v ní popsáno, co jsou B-stromy, konkurenční přístup obecně a jaké jsou možnosti řešení problémů paralelismu v datové struktuře B-strom.

Třetí část obsahuje implementaci, v průběhu které byly do stávajícího řešení přidány funkční části a celky zajišťující vícevláknové vkládání do B-stromu.

Ve čtvrté kapitole prezentuji dosažené výsledky při testování testování aplikace a jejich porovnání s jednovláknovým vkládáním.

Poslední, pátá část této práce, obsahuje závěr a mé osobní poznatky.

2 Teorie problematiky

2.1 B-strom

B-strom[4][2][8] je stromová struktura, která uchovává tříděná data a umožňuje v nich vyhledávání, sekvenční přístup, vkládání a mazání v logaritmickém čase. Navrhl ho v roce 1970 R. Bayer. Je zobecněním binárního vyhledávacího stromu takovým, že uzel může mít více než dva potomky. B-strom je používán převážně v systémech, které pracují s velkými bloky dat, tj. v databázích a souborových systémech.

B-strom je tvořen kořenem, uzly a listy. B-strom řádu n je takový strom, který má všechny listy na stejné úrovni (hloubce), všechny jeho uzly mají maximálně n a minimálně $\frac{n}{2}$ potomků (kromě kořene) a jeho kořen obsahuje maximálně n potomků.

Vnitřní uzly B-stromu mohou mít variabilní počet podřízených uzlů v předem stanoveném rozmezí. Pokud jsou do uzlu vložena nebo z uzlu odstraněna data, počet podřízených uzlů se mění. Jelikož je nutné dodržovat stanovené rozmezí počtu podřízených uzlů, jsou vnitřní uzly buď spojovány, nebo rozdělovány.

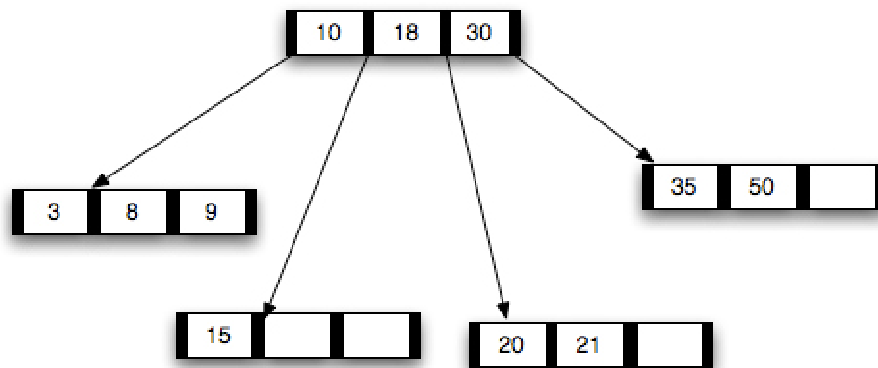
Každý vnitřní uzel B-stromu obsahuje k klíčů a tyto klíče rozdělují potomky do $k + 1$ podstromů.

Příklad 2.1
Pokud například vnitřní uzel má čtyři podstromy, pak musí mít tři klíče K_1, K_2 a K_3. Všechny hodnoty v levém krajním podstromu budou nižší než K_1, všechny hodnoty v podstromu druhém zleva budou mezi K_1 a K_2, všechny hodnoty v podstromu třetím zleva budou mezi K_2 a K_3 a všechny hodnoty v nejpravějším podstromu budou vyšší než K_3. ■

B-stromy mají značné výhody oproti jiným datovým strukturám, u kterých čas přístupu k datům z uzlu výrazně převyšuje čas potřebný ke zpracování těchto údajů. U B-stromů může být takto získaný čas využit k jiným operacím. K tomu obvykle dochází, když jsou data uzlu v sekundárním úložišti typu diskové jednotky. Zde se snažíme maximalizovat počet klíčů v každém vnitřním uzlu, tím klesá výška stromu, čímž se také se snižuje počet přístupů do uzlů.

B-strom drží klíče v setříděném pořadí pro sekvenční přístup, používá hierarchický index, aby se minimalizovalo množství diskových operací, využívá částečně plných bloků a minimalizuje nevyužitý prostor tím, že vnitřní uzly jsou vždy alespoň z poloviny plné a proto jsou dnes hojně využívány.

Jsou známy různé varianty B-stromů. B+-strom má kopie klíčů uloženy ve vnitřních uzlech, klíče a data jsou uloženy v listech, navíc může uzel obsahovat ukazatel na další uzel, což umožňuje rychlý sekvenční přístup. B*-strom drží vnitřní uzly více hustě zabalené, tzn., zvyšuje spodní hranici počtu potomků na $\frac{2}{3} * n$ a strom se pak nerozpadá tak rychle, jako B-strom.



Obrázek 1: B-strom

Definice 2.1 *B-Strom výšky n je strom, který splňuje tyto podmínky:*

- *kořen má nejméně dva potomky, pokud není listem,*
- *každý uzel kromě kořene a listu má nejméně $\frac{n}{2}$ a nejvýše n potomků,*
- *každý uzel kromě kořene má nejméně $\frac{n}{2} - 1$ a nejvýše $n - 1$ položek,*
- *všechny cesty od kořene k listům jsou stejně dlouhé*

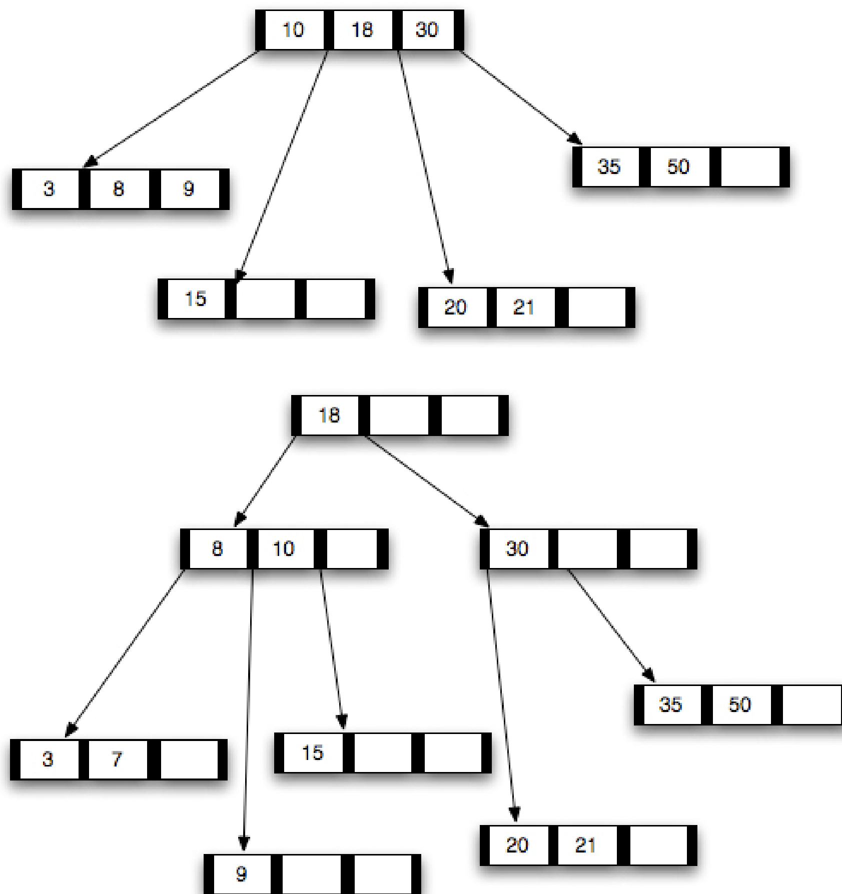
2.1.1 Operace vyhledávání

Vyhledávání v B-stromu probíhá tak, že začneme v kořeni a postupně postupujeme hierarchií stromu směrem dolů. Hledáme-li hodnotu x, pak ji v každém uzlu porovnáváme s klíči $K_1, K_2, \ldots, K_m$. V případě, že hodnota x není rovna žádnému z klíčů obsažených v uzlu, musíme postoupit o úroveň níže, tedy vyhledávat v potomkovi. Potomka, ve kterém budeme vyhledávat, určíme tak, že:

- Porovnáme, zda platí $K_1 > x$, a pokud ano, pak pokračujeme do potomka P_0
- Porovnáme, zda platí $K_i < x < K_{i+1}$, přičemž $1 \leq i < m$, a pokud ano, pak pokračujeme do potomka P_i
- Porovnáme, zda platí $K_m < x$, a pokud ano, pak pokračujeme do potomka P_m

Pokud po provedení výše uvedených kroků nenalezneme potomka, pak hodnota x ve stromu neexistuje.

Asymptotická složitost[14][5] operace vyhledávání v B-stromu je v nejhorším případě řádu $\log_n(N)$, kde N je počet položek, které strom obsahuje.



Obrázek 2: Vkládání hodnoty 7 do B-stromu

2.1.2 Operace vkládání

V případě, že se pokoušíme o vložení prvku do uzlu, který má méně než $n - 1$ položek, tzn. do uzlu, který není zcela naplněn, vložíme tento prvek přímo do tohoto uzlu tak, aby položky v uzlu zůstaly setříděny zleva doprava od nejmenšího po největší.

Pokud má ale uzel, do kterého chceme vložit prvek, $n-1$ prvků, tedy je zcela naplněn, pak tento uzel musíme rozdělit na dva nové uzly. Všech n prvků (kromě prostředního) pak rozdělíme rovnoměrně do těchto dvou uzlů, přičemž prostřední prvek vložíme do rodičovského uzlu. Pokud v tomto rodičovském uzlu dojde ke stejné situaci, tedy k přetečení prvků, provede se stejný proces rozdělení. V nejhorším případě je možné takto postupovat až ke kořeni. Pokud dojde k rozdělení kořene, vytvoří se kořen nový a výška stromu se tak zvýší. Toto je jediný způsob růstu stromu.

2.1.3 Operace mazání

Při mazání prvku postupujeme takto:

- V případě, že se mazaný prvek nachází v listu a tento list má více než $\frac{n}{2} - 1$ položek, stačí prvek jednoduše odstranit - list má dostatečnou rezervu a vlastnosti B-stromu zůstávají v platnosti.
- V případě, že se mazaný prvek nachází v listu a tento list má $\frac{n}{2} - 1$ položek, tedy minimální počet, pak postupujeme takto:
  - V případě, že levý sourozenec listu (pokud existuje), ze kterého mažeme prvek, má alespoň o jeden prvek více, než je minimum ($\frac{n}{2} - 1$ položek), pak mazaný prvek nahradíme i-tým prvkem jeho rodiče a ten nahradíme nejpravějším prvkem ze sourozence.
  - V opačném případě provedeme postup z předchozího bodu pro pravého sourozence (pokud existuje). Pokud jeho levý sourozenec má minimální počet prvků ($\frac{n}{2} - 1$ položek), pak oba uzly sloučíme, a to tak, že přidáme do levého sourozence $(i - 1)$-tý prvek z jejich otce a zbylé prvky listu, ze kterého je mazáno. Místo, které vzniklo v rodič,i je zaplněno posunem prvků v rodiči směrem doleva.
  - V případě, že levý sourozenec neexistuje, sloučíme uzel s pravým sourozencem.
- V případě, že se mazaný prvek nachází ve vnitřním uzlu, musíme mazaný prvek i buďto nahradit nejlevějším prvkem v $(i + 1)$-tém potomku, nebo nejpravějším prvkem v i-tém potomku. Nahrazující prvek následně z listu vymažeme postupem popsaným v předchozích bodech.

2.2 Konkurenční přístup obecně

V dnešní době, kdy komponenty informačních systémů současně komunikují prostřednictvím zpráv nebo sdíleného přístupu dat (v paměti nebo v úložišti), může být určitá konzistence komponent porušena další komponentou.

Typickými příklady jsou víceuživatelská, nebo vícevláknová aplikace, ve kterých dochází k souběhu. Souběh sebou přináší celou řadu problémů, které musíme řešit. Základním stavebním kamenem řešení konkurenčního přístupu je zavedení transakcí[9].

2.2.1 Transakce

Transakce je elementární jednotkou práce s databází, která v sobě zahrnuje sérii zásahů, kterými jsou vkládání, změna a odstraňování položek. Transakce musí vždy dodržovat standard ACID:

- A = Atomičnost - buďto jsou provedeny všechny operace transakce, nebo žádná

- C = Korektnost - mezi začátkem a koncem transakce musí být databáze v korektním stavu, pak tato transakce převádí databázi z jednoho korektního stavu do druhého
- I = Izolovanost - změny provedené jednou transakcí jsou pro ostatní transakce viditelné až po provedení COMMIT, tzn., že transakce jsou navzájem izolovány
- D = Trvalost - po potvrzení změn se tyto změny stávají trvalými a to i po případném pádu systému

Ve standardu ACID jsou operace COMMIT, kterou potvrzujeme úspěšně dokončenou transakci, a ROLLBACK, kterou vykonáme při chybě, čímž dovedeme databázi do konzistentního stavu před začátkem transakce.

2.2.2 Anomálie při zpracování transakcí

2.2.2.1 Ztráta aktualizace Ztráta aktualizace nastane v případě, že jedna transakce přepíše výsledek druhé.

T_1	Čas	T_2
READ t	t_1	-
-	t_2	READ t
WRITE t	t_3	-
-	t_4	WRITE t

Tabulka 1: Ztráta aktulizace: Aktualizace transakce T_1 je ztracena v čase t_4.

2.2.2.2 Dočasná aktualizace Příčinou této anomálie je čtení nepotvrzených dat - mějme dvě transakce T_1 a T_2. T_1 změní hodnotu t, následně transakce T_2 přečte obsah t dříve, než dojde k potvrzení transakce T_1. V čase t_3 je transakce T_2 zrušena a transakce T_1 pak pracuje s nesprávnými daty.

T_1	Čas	T_2
-	t_1	WRITE t
READ t	t_2	-
-	t_3	ROLLBACK

Tabulka 2: Dočasná aktualizace: Transakce T_1 se stala v čase t_2 závislou na nepotvrzené změně transakce T_2.

2.2.2.3 Neopakovatelné čtení Podstatou této anomálie je čtení dat před změnou a po potvrzení této změny - transakce T_1 přečte záznam, transakce T_2 tento záznam změní a změnu potvrdí. Pokud pak transakce T_1 přečte opět ten samý záznam, dostane odlišnou hodnotu, než při přechozím načtení.

T_1	Čas	T_2
READ t	t_1	-
-	t_2	WRITE t
READ t	t_3	-

Tabulka 3: Neopakovatelné čtení: Transakce T_1 přečetla hodnotu t v čase t_1 a t_3, přičemž ve druhém případě byl vrácen jiný výsledek, než v tom prvním.

2.2.2.4 Fantóm Jedná se o podobný problém, jako je neopakovatelné čtení. V tomto případě se změní počet vrácených záznamů – transakce T_1 přečte množinu záznamů na základě určených podmínek. Poté transakce T_2 odstraní nebo vloží nový záznam, který ale taktéž odpovídá určeným podmínkám z T_1 a T_2 je následně ukončena COMMIT. T_1 nyní zopakuje přečtení množiny záznamů na základě určených podmínek, ale obdrží odlišný počet záznamů, než jak tomu bylo v prvním případě.

T_1	Čas	T_2
READ t WHERE id>5	t_1	-
-	t_2	WRITE t, id=8
READ t WHERE id>5	t_1	-

Tabulka 4: Fantóm: Transakce T_1 obdržela množinu záznamů t v čase t_1 a t_3, přičemž ve druhém případě byl vrácen jiný počet záznamů, než v tom prvním.

2.2.3 Zamykání

Jedním ze způsobů řešení anomálií hrozících při paralelním souběhu transakcí je zajištění, aby přístup k datovým objektům byl výhradní. To znamená, že pokud jedna transakce přistupuje k jednomu objektu, žádná jiná jej nemůže modifikovat, dokud jej první transakce neuvolní.

Nejobvyklejším způsobem, jak tuto exkluzivitu zaručit, je dovolit transakci přístup k danému objektu pouze v tom případě, že na něj získá zámek.

Známy jsou dva základní způsoby uzamknutí objektu:

- **Sdílený zámek** - pokud transakce obdrží na objekt tento zámek, může objekt číst, ale nemůže jej modifikovat. Dokud transakce drží nad tímto objektem tento zámek, ostatní transakce na něj mohou také získat pouze sdílený zámek, nikoliv exkluzivní.

- **Exkluzivní zámek** - pokud transakce obdrží na objekt tento zámek, může objekt číst a může jej i modifikovat. Dokud transakce drží nad tímto objektem tento zámek, ostatní transakce na něj nemohou získat sdílený ani exkluzivní zámek.

Jedním z protokolů založených právě na zámcích je dvoufázové zamykání. Protokol obsahuje tato tři pravidla:

	X	S	-
X	N	N	A
S	N	A	A
-	A	A	A

Tabulka 5: Tabulka kompatibility základních způsobů uzamknutí objektu.

- Před transakční operací nad objektem je nejprvé nutné otestovat, zda zámek potřebný pro tuto operaci nekoliduje se zámkem jiné transakce. Pokud koliduje, musí se na přidělení zámku počkat.
- Jakmile transakce obdrží zámek, pak tento nemůže být uvolněn dříve, než bude dokončena operace, která zámek vyžádala.
- Pokud transakce zámek uvolní, nemůže již transakce stejný zámek získat znovu.

Tento protokol tedy dělí transakci na fázi zamykání, ve které transakce zámky získává, ale nemůže je uvolňovat, a na fázi odemykání, kde může transakce zámky již pouze uvolňovat. Tato základní varianta dvoufázového zamykání není odolná proti uváznutí.

2.2.3.1 Uváznutí Při zamykání objektů může dojít k uváznutí. Transakce T_1 a T_2 na sebe navzájem čekají, až dojde k uvolnění zámků, které si tyto transakce navzájem blokují. Uváznutí se řeší použitím ROLLBACK na transakci, kterou určíme na základě různých kritérií.

T_1	Čas	T_2
S_LOCK(A)	t_1	
S_LOCK(B)	t_2	
	t_3	X_LOCK(C)
	t_4	X_LOCK(A) - **wait!**
S_LOCK(C) - **wait!**	t_5	

Tabulka 6: Uváznutí.

2.2.3.2 Strukturované zamykání Zamykání se ale neprovádí pouze na úrovni určitého objektu, v některých případech je výhodnější objekty shluknout do objemnějších celků a zamknutí provést právě na tyto celky. Tímto způsobem dojde ke zmenšení počtu potřebných zámků. Tento způsob zamykání se nazývá strukturované zamykání.

Pro dosažení strukturovaného zamykání se používají záměrné zámky. Ty zamykají veškeré předchůdce dané granule, a to podle požadované operace, která má být provedena. Jedná se o tyto zámky:

- pro záměrné čtení (intention shared lock) - IS_LOCK,
- pro záměrný zápis (intention exclusive lock) - IX_LOCK,

- pro záměrné čtení s následným zápisem (shared with exclusive lock) - SIX_LOCK

Při zamknutí granule musí být jako první uzamčen kořen hierarchie, tyto zámky navíc musejí být drženy ve shodě s maticí kompatibility7. Uzel hierarchie může být transakcí uzamčen prostřednictvím S_LOCK nebo IS_LOCK, pouze v případě, že byl uzamčen prostřednictvím IS_LOCK nebo IX_LOCK jeho přímý předchůdce. Stejně tak může být uzel transakcí uzamčen prostřednictvím X_LOCK, IX_LOCK nebo SIX_LOCK pouze tehdy, když je uzamčen jeho přímý předek zámkem IX_LOCK nebo SIX_LOCK.

Transakce má právo odemykat uzel pouze v případě, že žádný z jeho dětí není uzamčen žádným zámkem, a naopak může transakce zamknout uzel pouze v případě, že předtím žádný uzel neodemkla.

	S	X	IS	IX	SIX
S	A	-	A	-	-
X	-	-	-	-	-
IS	A	-	A	A	A
IX	-	-	A	A	-
SIX	-	-	A	-	-

Tabulka 7: Matice kompatibilty záměrných zámků.

2.3 Paralelismus v datové struktuře B-strom

S datovou strukturou typu B-strom je v praxi nutné provádět základní operace uvedené v kapitole 2.1, tedy čtení, vkládání, aktualizaci a mazání. Datová struktura B-strom je v této práci reprezentována školním frameworkem QuickDB, který obsahuje již naimplementované funkce a metody umožňující s B-stromem tyto operace provádět.

Vzhledem k tomu, že obecnou operaci čtení v této školní aplikaci ošetřil můj kolega v roce 2013 ve své bakalářské práci [7] tak, aby byla vláknově bezpečná, budu se v této práci zabývat ošetřením stávající metody pro vkládání do B-stromu, aby byla taktéž vláknově bezpečná.

Ošetřit metodu pro vkládání tak, aby byla vláknově bezpečná [11], již, oproti metodě umožňující pouze čtení, vyžaduje zamykání nejen sdílené, ale také výlučné, chcete-li exkluzivní. Zapsání hodnoty do B-stromu je totiž formou transakce, jejíž obecná forma je popsána v kapitole 2.2.1. Z toho vyplývá, že transakce, která provede takovéto vložení hodnoty do B-stromu, musí samozřejmě dodržovat standard ACID. Při zapisování musíme navíc uvažovat zamykání nejen samostatných objektů, ale celých granulí, které mohou být potenciálně ovlivněny tak, že se změní jejich vlastnosti.

V našem případě budou tedy těmito granulemi větve, po kterých musíme v hierarchii B-stromu projít, než se dostaneme do uzlu, do kterého chceme zapisovat. Laik by mohl namítnout, že bude stačit jednoduše exkluzivně zamknout všechny objekty po cestě do uzlu, který hledáme, jenže zkušenější čtenář již tuší, že proměnit tento proces v dokonale vláknově bezpečný sebou nese mnoho komplikací a problémů, které je třeba vyřešit [3][12].

2.3.1 První problém: Propustnost

Jak již bylo naznačeno výše, vícevláknové spuštění funkce, která vkládá hodnoty do B-stromu, sebou nese vysokou režii na ošetření vláknové bezpečnosti. Z tohoto důvodu je nutné, aby zavedení vícevláknového vkládání bylo opodstatněné, respektive by mělo být efektivnější [10], než vkládání jednovláknové. Z toho plyne, že nebude-li zajištěna dostatečná datová propustnost vkládací metody ošetřené synchronizačními mechanismy (zámky), nemá smysl vícevláknové vkládání vůbec implementovat. Dostatečná propustnost by měla být zajištěna správným návrhem velikosti zamykaných granulí a správným způsobem volby konkrétních typů zámků v konkrétních případech průchodů či operací s B-stromem.

2.3.2 Druhý problém: Dodržení ACID

Propustnost úzce souvisí s dodržením standardu ACID. Při zajištění co největší propustnosti chceme logicky klást důraz na zamykání co nejmenších granulí, respektive na co nejvyšší poměr sdílených zámků. Nejmenší granule je současně dána tím, jakou režii zamykání je nutné uplatnit pro dodržení ACID.

Dodržení ACID by nikdy nemělo být porušeno na úkor vyšší propustnosti.

2.3.3 Třetí problém: Uváznutí

Uváznutí je potřeba ošetřit v každé synchronizační metodě. Chceme-li zvýšit propustnost co nejvyšším počtem sdílených zámků, je jednoduché si představit, co se stane, když například pustíme dvě vlákna do jednoho uzlu na základě sdíleného zámku za účelem čtení a obě tato vlákna v dalším kroku provedou split potomka. Samozřejmě dojde k uváznutí.

Jak ale ošetříme potřebnou aktualizaci oběma vlákny sdíleně zamknutého rodiče, je veliký problém. Takovýchto případů uváznutí může mezi zamykáním a odemykáním konkrétních zámků nastat hned několik, vždy závisí na konkrétním algoritmu provádějícím operaci nad B-stromem.

Řešením může být nastavení priorit jednotlivých transakcí, určení, která ze zablokovaných transakcí již provedla více práce, nebo časové razítko, podle kterého zjistíme, která transakce začala svou práci vykonávat dříve [13]. Dle těchto kritérií určíme, která ze zablokovaných transakcí bude pokračovat a která bude odvolána ROLLBACKem, respektive spuštěna znovu.

2.3.4 Metody zamykání

Za účelem stanovení standardizace vláknově bezpečných operací v B-stromu byly popsány některé metody zamykání. Z nich jsem vybral a popsal dvě, které mne nejvíce zaujaly a k tomu jsem nabídnul své vlastní řešení.

2.3.4.1 Hypotetická metoda č. 1: Zamykání rozsahu záznamů

Účelem zamykání rozsahu je chránit jednu transakci před zápisem jiné transakce. Například, pokud transakce provádí dotaz typu *select count* $(*)$ *where ... , between ... a ...*, pak druhý výkon stejného dotazu by měl vracet stejný výsledek. Jinými slovy - kromě ochrany stávající položky B-stromu v rozsahu dotazu, musejí zámky získané a držené touto transakcí chránit také mezery mezi klíčovými záznamy proti vkládání nových záznamů. Ještě jinými slovy - zamykání rozsahu zajišťuje uzamčení mezer mezi existujícími hodnotami.

Zamykání rozsahu je speciální forma predikátového zamykání [6]. Predikáty jsou zde definovány intervaly v setříděném B-stromu. Hranice intervalu jsou hodnoty existující v B-stromu. Obvyklou formou jsou napůl otevřené intervaly zahrnující mezery mezi dvěma sousedními hodnotami a jeden z koncových bodů s tzv. "next-key zamknutím", nebo tzv. "prior-key zamyknutím". Next-key zamknutí vyžaduje schopnost uzamknout umělé hodnoty $+\infty$. Prior-key zamknutí může být získáno zamknutím hodnoty NULL, za předpokladu, že to je nejnižší možná hodnota v setříděném B-stromu.

V nejjednodušší formě zamykání rozsahu jsou hodnota a mezery k sousedovi uzamčeny jako jeden celek. Exkluzivní zámek je vyžadován pro jakoukoliv formu aktualizace záznamu B-stromu, nebo mezery k jeho sousedovi. Aktualizace hodnoty vyžaduje zámek na starou hodnotu a jejího souseda, posledně uvedený je povinen zajistit schopnost nové aktualizace hodnoty v případě zavolání ROLLBACK.

2.3.4.2 Hypotetická metoda č. 2: Bayerova a Schkolnickova sada protokolů zamykání

Zajišťuje integritu přístupů do B-stromu a zároveň umožňuje souběžnou činnost [1].

Při čtení jde v podstatě o výlučné zámky držené nad aktuálním uzlem (a v určitém momentu i jeho předkem), to znamená, že dokud jsou nad uzlem drženy, jiné vlákno do něj nemůže zasahovat, a to ani pro čtení. Vzhledem k tomu, že při čtení, respektive hledání požadovaného uzlu, se postupuje do větší hloubky hierarchie B-stromu, zámek na předka v předchozí hloubce se uvolní, což umožní čtení jiným vláknem. Z toho vyplývá, že při čtení zamykáme většinou maximálně dva uzly. To je oproti běžnému sdílenému zámku velký rozdíl.

Aktualizace či zápis však, stejně jako u všech metod, představuje složitější problém, který vyžaduje složitější řešení. Vzhledem k tomu, že při aktualizaci či zápisu můžeme ovlivnit vyšší úrovně hloubky, než ve které se právě nacházíme, ponecháváme si výhradu (rezervaci) zamknout každý uzel, ke kterému bylo při průchodu hierarchií B-stromu přistoupeno. Později může být rezervace převedena na zámek, a to v případě, že aktualizace se bude šířit hierarchií stromu směrem nahoru (typicky při splitu). Případně může být rezervace zrušena, pokud budeme mít jistotu, že vyhrazené uzly nebudeme aktualizovat.

Rezervace převedená na zámek zaručuje, že žádná jiná transakce nebude mít přístup do tohoto uzlu, je to tedy obdoba exkluzivního zámku. Aktualizace uzlů probíhá pouze v případě, že jsme místo rezervací obdrželi zámky na všechny uzly dotčené touto aktualizací v hierarchii B-stromu směrem nahoru. Poté, co se provedou všechny změny, jsou zámky zrušeny a všechny dotčené uzly jsou k dispozici pro ostatní transakce. V drtivé většině

případů aktualizací uzlů je nutné takto ovlivnit pouze několik úrovní v blízkosti zdroje aktualizace vyšší hloubky, což v sobě skrývá výhodu toho, že nemusíme zamykat celou cestu směrem nahoru až do kořene. Přesto je zde riziko, že pokud zamkneme příliš málo uzlů, mohlo by se stát, že bude nutné začít znovu u kořene.

2.3.4.3 Hypotetická metoda č. 3: Vlastní návrh Vlastní řešení jsem se rozhodnul navrhnout ze dvou důvodů. Jednak jsem začal polemizovat nad vylepšením laikova nápadu zamknutí celé cesty v hierarchii B-stromu exkluzivním zámkem z kapitoly 2.3, tak, že bych zmenšil granule exkluzivně zamknutých uzlů pouze na ty nejnutnější a zbytek by zůstal zamknut sdíleným zámkem; a jednak je dle mého názoru výše uvedené dvě metody obtížné aplikovat na stávající implementaci metody pro vkládání do B-stromu ve frameworku QuickDB, protože metoda nevyužívá rekurze, nýbrž je implementována formou cyklu for, takže veškeré údaje o zámcích a cestách bude nutné ukládat explicitně. A konečně - žádný z těchto mechanismů nepopisuje jak synchronizovat transakce při případném několikanásobném hierarchickém splitu, takže bych řešení tohoto problému stejně musel hledat svou cestou.

Metoda je tedy založena na zamykání konkrétních uzlů a ukládání cesty v B-stromu.

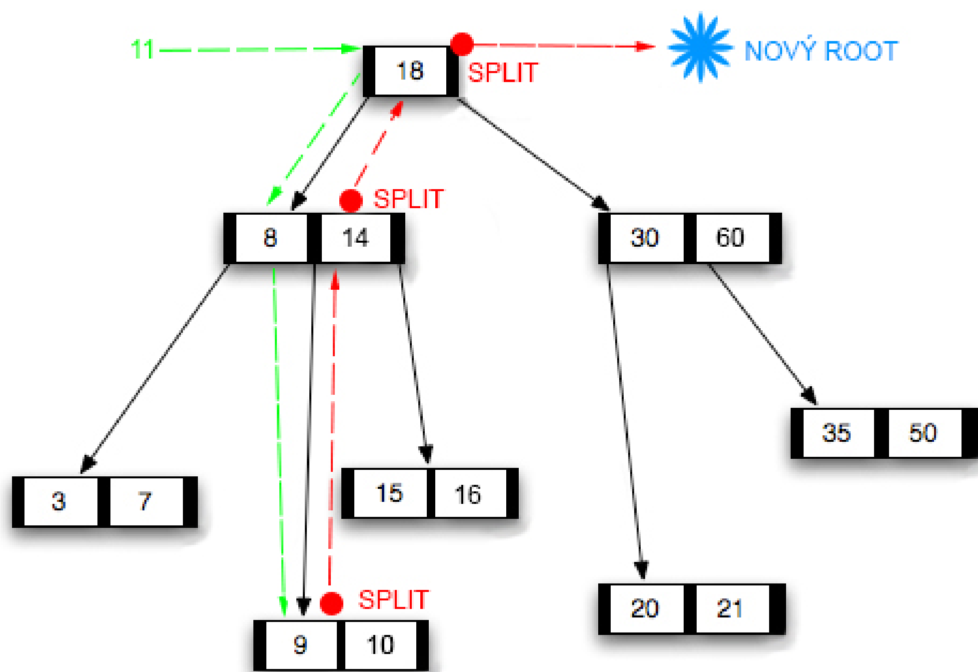
Při čtení, respektive procházení herarchií B-stromu směrem dolů, jsou do lokálního pole ukládány odkazy na uzly, kterými jsme prošli, respektive které jsme zamkli sdíleným zámkem. Tím máme zajištěno, že víme, které uzly budeme muset odemknout po doběhnutí cyklu, respektive po dosažení listu. Do takto zamknutých uzlů mají přístup ostatní transakce, ovšem také pouze pro čtení.

Při aktualizaci nebo zápisu do uzlu je potřeba sdílený zámek aktuálního uzlu, ve kterém se právě nacházíme, přetransformovat na zámek exkluzivní. To musí být zajištěno tak, aby nevzniknul prostor pro získání jakéhokoliv zámku na tento uzel jinou transakcí. V praxi to znamená, že exkluzivní zámek obdrží transakce pouze tehdy, nedrží-li nad konkrétním uzlem žádná jiná transakce, a to jak exkluzivní, tak ani sdílený. Pro exkluzivně zamknuté uzly máme také lokální pole. Po doběhnutí cyklu, respektive dokončení vložení hodnoty do B-stromu, opět projdeme pole a poodemykáme všechny zamčené uzly.

Výše uvedený popis platí pro nejjednodušší formu vkládání do B-stromu, tedy v případech, kdy se hierarchií pohybujeme výhradně směrem dolů (odemknutí pak provedeme jednoduše od nejspodnějšího uzlu po nejvyšší).

Komplikace však přicházejí v případě, že v listovém uzlu, do kterého chceme vkládat, dojde místo. V takovém případě dochází ke splitu, tedy rozdělení listu na dva nové, respektive jeden stávající a jeden nový. Následovat musí aktualizace rodiče. Musíme na něj získat exkluzivní zámek, tedy v tomto případě se již pohybujeme hierarchií B-stromu směrem nahoru. Aktualizace rodičovského uzlu spočívá mimo jiné v tom, že se zde musí přidat prvek odkazující na nově vlivem splitu vzniklého potomka.

Daleko větší komplikace však nastává, když není dostatek místa na nový prvek ani v tomto rodiči. V úplně nejhorším případě se může stát, že není místo na nový prvek v celé hierarchii rodičů směrem nahoru až do samotného kořene B-stromu. V takovém případě je rozdělen kořen B-stromu na dvě poloviny a nad nimi vznikne kořen nový, čímž se o



Obrázek 3: Nejhorší případ, který může nastat - splity v hierarchii B-stromu až do kořene

jeden stupeň zvýší výška stromu. Ošetřit tyto operace jako vláknově bezpečné je velice složité a komplikované, nicméně v implementaci níže se pokusím najít řešení.

3 Implementace vybrané metody zamykání funkce pro vkládání do B-stromu

Implementoval jsem tedy svou vlastní metodu. Nejprve bylo potřeba upravit třídu cNode, respektive přidat do ní atributy a metody.

```
std::atomic <unsigned int> readLockCounter;
std::atomic <unsigned int> writeLockCounter;
bool tiskniHlasky = false;

inline bool rLockGet();
inline bool rLockRelease();
inline bool wLockGet();
inline bool wLockRelease();
inline bool upgradeToWriteLock();
inline bool downgradeToReadLock();
inline int countReadLocks();
inline int countWriteLocks();
```

Výpis 1: Atributy a metody přidané do třídy cNode

Atributy, které jsem do třídy přidal, jsou dvě atomické proměnné typu $unsigned\ int$. Jedná se o countery počtu aktuálně držených locků nad instancí potomka této třídy. Další atribut $tiskniHlasky$ slouží k aktivaci výpisu kontrolních hlášek na standardní výstup. Jak vyplývá z názvů metod, jsou zde dvě metody obsluhující přidělení/odebrání $ReadLocku$, stejně fungující dvě metody jsou zde i pro $WriteLock$. Další dvě metody slouží k přeměně $ReadLocku$ na $WriteLock$ (respektive obráceně), zajišťující tuto úlohu bez rizika, že by se mezi změnou locku dostala k tomuto uzlu jiná transakce. Všech těchto šest metod vrací návratovou hodnotu typu bool, která nabývá hodnoty **true** v případě přidělení zámku konkrétní transakci nad aktuálním uzlem a hodnoty **false**, pokud se zámek získat nepodaří. Poslední dvě metody plní kontrolní funkci, vracejí hodnotu typu int s celkovým počtem aktuálně držených zámků nad tímto uzlem.

```
inline bool cNode::rLockGet()
{
    if (writeLockCounter == 0){
        readLockCounter++;
        return true;
    }
    else{
        return false;
    }
}
inline bool cNode::rLockRelease()
{
    if (readLockCounter != 0){
        readLockCounter--;
        return true;
    }
    else{
        return false;
```

```
    }
}
inline bool cNode::wLockGet()
{
    if (writeLockCounter == 0 && readLockCounter == 0){
        writeLockCounter++;
        return true;
    }
    else{
        return false;
    }
}
inline bool cNode::wLockRelease()
{
    if (writeLockCounter != 0){
        writeLockCounter−−;
        return true;
    }
    else{
        return false;
    }
}
inline bool cNode::upgradeToWriteLock()
{
    if (writeLockCounter == 0 && readLockCounter <=1 ){
        writeLockCounter++;
        readLockCounter−−;
        return true;
    }
    else{
        return false;
    }
}
inline bool cNode::downgradeToReadLock(){
    if (writeLockCounter != 0){
        readLockCounter++;
        writeLockCounter−−;
        return true;
    }
    else{
        return false;
    }
}
inline int cNode::countReadLocks(){
    return readLockCounter;
}
inline int cNode::countWriteLocks(){
    return writeLockCounter;
}
```

Výpis 2: Definice metod přidaných do třídy cNode

Třída, která je v této aplikaci používána pro vytvoření datové struktury typu B-strom, se jmenuje $cCommonB{+}Tree$. Její metoda, která je v této práci podstatná, se jmenuje $Insert$.

V této metodě probíhá veškerá agenda a práce týkající se vkládání a hlavně zamykání - jednotlivé uzly jsou zamykány a odemykány prostřednictvím metod, které jsem umístil do třídy $cNode$ 2. Již výše bylo uvedeno, že tato vkládací metoda nepracuje rekurzivně, ale pracuje na základě cyklu for. To je důvod, proč bylo nutné i v této metodě vytvořit několik lokálních proměnných.

```
bool printLocks = true;
bool printDeadLocks = true;
int waitMSec = 100;
int hloubka = 0;
TNode* SlockedInnerNodes[10];
TNode* WlockedLeafNodes[10];
TNode* WlockedInnerNodes[10];
TNode* SlockedLeafNodes[10];
```

Výpis 3: Deklarace lokálních proměnných přidaných do metody Insert třídy cCommonB+Tree

První dvě proměnné slouží opět k aktivaci/deaktivaci informačních výpisů na standardní výstup. Proměnná $waitMSec$ stanovuje v milisekundách dobu, jakou bude algoritmus dělat přestávky mezi neúspěšnými žádostmi o přidělení zámku. $Hloubka$ je proměnná sloužící k zaznamenávání toho, jak hluboko jsme se ve stromu již zanořili. Následuje deklarace čtyř polí typu $TNode*$, která slouží k uchovávání zamknuté cesty, respektive pointerů na uzly, které byly v této konkrétní instanci metody zamčeny. Pole je třeba nejprve naplnit hodnotami $NULL$, abychom na konci iterace vkládacího cyklu byli schopni všechna pole projít a poodemykat zamknuté uzly.

```
for (int k = 9; k > -1; k--) {
    WlockedInnerNodes[k] = NULL;
    SlockedInnerNodes[k] = NULL;
    WlockedLeafNodes[k] = NULL;
    SlockedLeafNodes[k] = NULL;
}
```

Výpis 4: Naplnění polí hodnotami NULL

Pak již stačí v kódu metody $Insert$ volat jednotlivá zamykání na správných místech. K dispozici je tedy $ReadLock$, $WriteLock$, transformace z $ReadLocku$ na $WriteLock$ a obráceně.

```
tryLockCount = 0;
while (!currentLeafNode->rLockGet()){
    if (printLocks) printf ("cekam na prideleni readlocku 0! index uzlu: %i, vlakno: %i\n",
        nodeIndex, threadId);
    tryLockCount++;
    if (printDeadLocks && tryLockCount >= 10) printf("deadlock? 0!\n");
    Sleep(waitMSec);
}
tryLockCount = 0;
SlockedLeafNodes[++hloubka] = currentLeafNode;
if (printLocks) printf ("zamknul jsem LeafNode %i Readlockem0 v hloubce %i, vlakno: %i\n",
    nodeIndex, hloubka, threadId);
```

Výpis 5: Příklad pokusu o získání ReadLocku na listový uzel

Jelikož se jedná o složitější problematiku, je podrobnější popis toho co, jak a kdy je zamykáno či odemykáno, popsán níže. Zamknuté uzly se odemknou najednou po skončení iterace vkládacího cyklu (není pravidlem, v některých případech je nutné požadovaný uzel odemknout dříve, než na konci iterace vkládacího cyklu). Pomocí polí je zaznamenána cesta, kterou bylo třeba projít pro vložení daného záznamu. Pole tedy projdeme a dotčené uzly odemkneme.

```
for (int k = 9; k > -1; k--) {
   if (WlockedInnerNodes[k] != NULL){
      WlockedInnerNodes[k]->wLockRelease();
   }
   if (SlockedInnerNodes[k] != NULL){
      SlockedInnerNodes[k]->rLockRelease();
   }
   if (WlockedLeafNodes[k] != NULL){
      WlockedLeafNodes[k]->wLockRelease();
   }
   if (SlockedLeafNodes[k] != NULL){
      SlockedLeafNodes[k]->rLockRelease();
   }
}
```

Výpis 6: Odemknutí dotčených uzlů po skončení iterace vkládacího cyklu

Takto ošetřená metoda $Insert$ je vláknově bezpečná a můžeme ji tedy volat paralelně více vlákny. Toho docílíme podobně jako je tomu u metody $RunPointQuery$, která provádí vyhledávání v B-stromu právě prostřednictvím vícevláknového přístupu. Metodu $Create_{F}ixedLen$, volanou v hlavní spouštěcí funkci $main$ tedy upravíme tak, aby se vkládané prvky rozdělily mezi jednotlivá vlákna.

```
for (unsigned int i = 0 ; i < NOF_THREADS_M1 ; i++)
{
   inThreads[i] = new std::thread(insertTuples, mIndex_Fixedlen, data, loQueryOrder,
        hiQueryOrder);
   loQueryOrder += count;
   hiQueryOrder += count;
}
for (unsigned int i = 0 ; i < NOF_THREADS-1 ; i++)
{
   inThreads[i]->join();   // wait for all threads created
}
for (unsigned int i = 0 ; i < NOF_THREADS-1 ; i++)
{
   delete inThreads[i];
}
```

Výpis 7: Kód rozdělující vkládání prvků mezi jednotlivá vlákna

Funkce $insertTuples$ obsahuje mírně modifikovaný původní kód jednovláknového vkládání do B-stromu. Cyklus zde probíhá pouze na intervalu prvků, který dostane přidělen v parametrech $loQueryOrder$ a $hiQueryOrder$.

```
void insertTuples(tBpTreeType_Fixedlen* mIndex_Fixedlen, char* data, unsigned int loQueryOrder,
        unsigned int hiQueryOrder){
    for (unsigned int i = loQueryOrder; i < hiQueryOrder; i++)
    {
        if (i % 25000 == 0)
        {
            cout << ".";
        }
        mIndex_Fixedlen->Insert(*mKey_Fixedlen[i], data);
    }
}
```

Výpis 8: Funkce insertTuples

3.1 Konkrétní způsob zamykání v metodě Insert

Metoda Insert je tvořena nekonečným cyklem for, ukončen je tedy pouze tehdy, pokud dosáhneme cíle, za kterým byl spuštěn (nalezení či vložení požadované hodnoty). Tento cyklus slouží k procházení B-stromem, a to v hierarchii jak směrem dolů (zanořování), tak i směrem nahoru (vynořování). Cyklus je rozdělen na tři základní části (tzv. staty), pomocí kterých je určeno jakou operaci budeme v aktuální iteraci cyklu provádět.

State0 je nejzákladnější část kódu. Jak ukazuje vývojový diagram A, na samém začátku proběhne větvení, kdy určíme, zda se jedná o listový, nebo vnitřní uzel. V obou případech provedeme zamknutí uzlu $ReadLockem$. V případě, že se jedná o vnitřní uzel, si algoritmus zjistí, do jakého potomka má v příští iteraci vstoupit a práce v aktuálním uzlu končí, další zamykání tedy již neprobíhá. V případě, že se jedná o listový uzel, je situace složitější - v závislosti na námi zvolených vlastnostech listů (duplikátní hodnoty klíče, respektive povolená/zakázaná inkrementace hodnot) mohou nastat různé případy. V našem případě jde o tyto:

- vkládaná položka v tomto listovém uzlu ještě není a je zde dostatek místa, transformujeme tedy $ReadLock$ na $WriteLock$ a vložíme položku (pokud je položka vložena na konec uzlu, musíme při další iteraci modifikovat rodiče, tedy budeme pracovat ve State2)

- vkládaná položka v tomto listovém uzlu ještě není a není zde dostatek místa - transformujeme tedy $ReadLock$ na $WriteLock$ a provedeme split, položku vložíme do jednoho z těchto dvou rozdělených uzlů a v příští iteraci budeme pracovat v části State1 (vložení odkazu na nového potomka do vnitřního uzlu rodiče)

Jak již bylo naznačeno výše, State1 a State2 jsou části, ve kterých se pracuje výhradně s vnitřními uzly. Tyto části jsou používány při vynořování v hierarchii B-stromu, tedy při postupu směrem nahoru, který je potřeba pouze v případě nutnosti modifikace rodičů.

State1 část je zavolána na vnitřní uzel v případě, že v potomkovi došlo ke splitu a je tedy nutné přidat odkaz na nového potomka. V ideálním případě je v tomto uzlu dostatek volného místa pro další položku. Na tento vnitřní uzel bychom tedy mohli získat $WriteLock$ a položku přidat, pak práce celého algoritmu končí. Pokud ale v tomto vnitřním uzlu místo není, pak dojde ke splitu i zde a celá výše uvedená operace se musí provést nad dalším, v hierarchii o jeden stupeň výše nacházejícím se, rodičovském vnitřním uzlu. Zde nastává problém, protože je zde vysoká pravděpodobnost častého uváznutí při vícevláknovém přístupu u kterého nelze jednoduše provést ROLLBACK, protože jsme již provedli jistou aktualizaci dat. Z toho plyne, že zamykání jednotlivých uzlů při práci v části State1 nebude fungovat.

Musíme tedy zvolit jiné řešení. Vzhledem k této konkrétní implementaci B-stromu, která zajišťuje velké množství položek v listových uzlech, můžeme předem počítat s tím, že výška B-stromu nebude příliš velká. Z toho plyne, že ani cesta z kořene do listového uzlu nebude vzhledem k počtu položek nijak dlouhá a tím máme zajištěno, že případný split, který by v nejhorším případě způsobil výše uvedený problém zvětšení výšky B-stromu, bude minimálně využívaným procesem (vzhledem k celkovému počtu operací s B-stromem se jedná o mizivé procento). Konkrétně jsem naměřil při standardním počtu 2000000 vkládaných položek a velikosti bloku 8192 bajtů přesně 5656 splitů, což je cca 0,29% z celkového počtu operací. Na základě tohoto údaje si tedy při splitu ve State1 můžeme z hlediska zachování dostatečné propustnosti **dovolit zamknout celý B-strom**. Provedeme to tak, že na B-strom si před detekovaným splitem ve State0 nejprve zavoláme $ReserveBpTreeLock$ a počkáme na doběhnutí již započatých transakcí ostatních vláken, která o rezervaci budou vědět a nebudou tak vytvářet nové transakce. Jakmile nad B-stromem nebude prováděna žádná jiná operace, přeměníme rezervaci na $AllBpTreeLock$ a zahájíme operaci splitování listu, která může přerůst v operaci splitování vnitřního uzlu ve State1. Po skončení celé této operace lock a rezervaci na celý B-strom uvolníme. Pokud by při čekání na rezervaci došlo k uváznutí, budeme samozřejmě přednostně provádět ROLLBACK na všechny ostatní transakce. V případě, že dvě vlákna vzájemně vyvolají rezervaci zamknutí celého B-stromu, budeme postupovat standardně jako při jakémkoliv jiném uváznutí, tedy ROLLBACK provedeme na později spuštěnou transakci. A to si můžeme bez obav dovolit, protože rezervaci provedeme ještě před jakoukoliv změnou dat.

```
bool cCommonBpTree<TNode, TLeafNode, TKey>::reserve_Btree_lock()
{
    if (bTreeReserveCounter == 0){
        bTreeReserveCounter++;
        return true;
    }
    else{
        return false;
    }
}
bool cCommonBpTree<TNode, TLeafNode, TKey>::unreserve_Btree_lock()
{
    bTreeReserveCounter--;
    return true;
```

```
}
bool cCommonBpTree<TNode, TLeafNode, TKey>::get_Btree_lock()
{
    if (bTreeLockCounter == 0){
        bTreeLockCounter++;
        return true;
    }
    else{
        return false;
    }
}
bool cCommonBpTree<TNode, TLeafNode, TKey>::release_Btree_lock()
{
    if (bTreeLockCounter != 0){
        bTreeLockCounter−−;
        return true;
    }
    else{
        return false;
    }
}
int cCommonBpTree<TNode, TLeafNode, TKey>::count_Btree_reservations()
{
    return bTreeReserveCounter;
}
int cCommonBpTree<TNode, TLeafNode, TKey>::count_Btree_locks()
{
        return bTreeLockCounter;
}
void cCommonBpTree<TNode, TLeafNode, TKey>::set_Btree_waiting()
{
    bTreeWaitingCounter++;
}
void cCommonBpTree<TNode, TLeafNode, TKey>::release_Btree_waiting()
{
    if (bTreeWaitingCounter != 0){
        bTreeWaitingCounter−−;
    }
}
```

Výpis 9: Metody pro zamykání celého B-stromu pro situaci při splitu přidané do třídy cCommonBpTree

Ve výpisu kódu výše 9 jsou uvedeny metody obsluhující zamykání celého B-stromu, které jsem přidal do třídy $cCommonBpTree$. Metody pracují se dvěma atomickými proměnnými typu unsigned int, které fungují jako countery pro zamykání celého B-stromu. Tyto metody budeme volat v metodě $Insert$. První čtyři metody pracují stejně jako metody pro zamykání uzlů, tedy vracejí návratovou hodnotu typu bool. Pátá a šestá slouží ke zjištění počtu zámků a vracejí návratovou hodnotu typu int. Poslední dvě umožňují uložit informaci o tom, že vlákno tzv. ”stojí”a dává prostor jinému vláknu na přeměnu z rezervace na zámek celého B-stromu.

Do metody $Insert$ jsem dále přidal proměnné $isBPTreeReserved$ a $isBPTreeLocked$ pro uložení informace o tom, zda držíme rezervaci nebo zámek nad celým B-stromem pro dané vlákno. Dále je nutné přidat na začátek metody $Insert$ ověření, zda nad B-stromem není držena rezervace nebo samotné zamknutí jiným vláknem. Zde setrváme tak dlouho, dokud je nad B-stromem držena jakákoliv rezervace či zámek.

```
set_Btree_waiting () ;  //davam vedet ostatnim vlaknum, ze neprovadim zadne vkladaci operace
while (count_Btree_reservations() != 0 || count_Btree_locks() != 0){
    if (printLocks) printf ("cekam na uvolneni rezervace ci locku celeho B−stromu, vlakno: %i\
        n", threadId);
    Sleep(waitMSec);
}
release_Btree_waiting () ;  //davam vedet ostatnim vlaknum, ze jdu provadet vkladaci operace a
    nemohou tak ziskat lock na cely B−strom
```

Výpis 10: Detekce rezervace nebo zamknutí celého B-stromu

Samotné zamykání (kterému předchází rezervace) celého B-stromu je pak vyvoláno při přepnutí ze State0 do State1, čímž si zajistíme "čistý stůl"pro všechny případy, včetně toho nejhoršího v podobě splitnutí kořene.

```
tryLockCount = 0;
while (!reserve_Btree_lock()){
    if (printLocks) printf ("cekam na prideleni rezervace4 na cely B−strom, vlakno: %i\n",
        threadId);
    tryLockCount++;

    //detekujeme deadlock a provedeme rollback
    if (tryLockCount >= 10){
        if (printDeadLocks) printf("DEADLOCK 4! ROLLBACK, vlakno: %i\n", threadId);

        //pred rollbackem je potreba odemknout vsechny cestou zamknute uzly
        for (int k = hloubka; k > −1; k−−) {
            if (WlockedInnerNodes[k] != NULL){
                WlockedInnerNodes[k]−>wLockRelease();
                WlockedInnerNodes[k] = NULL;
            }
            if (SlockedInnerNodes[k] != NULL){
                SlockedInnerNodes[k]−>rLockRelease();
                SlockedInnerNodes[k] = NULL;
            }
            if (WlockedLeafNodes[k] != NULL){
                WlockedLeafNodes[k]−>wLockRelease();
                WlockedLeafNodes[k] = NULL;
            }
            if (SlockedLeafNodes[k] != NULL){
                SlockedLeafNodes[k]−>rLockRelease();
                SlockedLeafNodes[k] = NULL;
            }
        }
        Sleep(100);
        goto NAVEST_START0; //presmerovani zpet na zacatek cyklu, KDE JSOU ZNOVU
            NASTAVENY DEFAULTNI HODNOTY PROMENNYCH = ROLLBACK
```

```
    }
Sleep(waitMSec);
}

tryLockCount = 0;
isBPTreeReserved = true;

if (printLocks)
printf ("rezervoval␣jsem␣B−strom␣AllBTreeLockem4,␣vlakno:␣%i\n", threadId);
//pokud mame rezervaci, musime pockat az ostatni vlakna ukonci stavajici cinnost a pak dostaneme
    zamek na cely B−strom, tady nemuze nastat deadlock, protoze mame prednost pred vsim
while (!get_Btree_lock()){
    if (printLocks) printf ("cekam␣na␣prideleni␣locku5␣na␣cely␣B−strom,␣vlakno:␣%i\n", threadId);
    Sleep(waitMSec);
}

isBPTreeLocked = true;
if (printLocks) printf ("zamknul␣jsem␣B−strom␣AllBTreeLockem5,␣vlakno:␣%i\n", threadId);
```

Výpis 11: Rezervace a následné zamknutí celého B-stromu

State2 je pak nejjednodušší částí vkládacího cyklu - pouze zajišťuje modifikaci rodičovského, respektive vnitřního uzlu, je-li to potřeba. Navzdory tomu ale i v této části budeme zamykat celý B-strom. A to proto, že do State2 se můžeme dostat jak ze State0, tak ze State1. Způsobí to výrazné snížení propustnosti, ale jiné řešení by vyžadovalo modifikaci samotné implementace metody $Insert$, což není v možnostech rozsahu ani cílem této práce.

3.1.1 Detekce uváznutí a ROLLBACK

Na některých místech zamykání v metodě $Insert$ je nutné ošetřit postup, který se stane v případě uváznutí. Uváznutí jsem naimplementoval jako deset neúspěšných pokusů o získání zámku. Tato detekce musí být provedena před jakoukoliv změnou dat, aby případný ROLLBACK mohl celou transakci zrušit, respektive spustit znovu při zachování atomičnosti transakce. Před zavoláním ROLLBACK je nutné zrušit všechny dosud získané zámky, změnit všechny lokální proměnné metody $Insert$ na defaultní a pomocí návěsti spustit celý cyklus od začátku.

```
//UPGRADE READLOCK to writelock
tryLockCount = 0;
while (!currentLeafNode−>upgradeToWriteLock()){
    if (printLocks) printf ("cekam␣na␣prideleni␣writelocku␣3!,␣vlakno:␣%i\n", threadId);
    tryLockCount++;
    if (tryLockCount >= 10){
        if (printDeadLocks) printf("DEADLOCK␣3!␣ROLLBACK,␣vlakno:␣%i\n", threadId);
        //pred rollbackem musime odemkout vsechny dosud zamknute nody
        for (int k = hloubka; k > −1; k−−) {
            if (WlockedInnerNodes[k] != NULL){
                WlockedInnerNodes[k]−>wLockRelease();
                WlockedInnerNodes[k] = NULL;
            }
```

```
            if (SlockedInnerNodes[k] != NULL){
               SlockedInnerNodes[k]->rLockRelease();
               SlockedInnerNodes[k] = NULL;
            }
            if (WlockedLeafNodes[k] != NULL){
               WlockedLeafNodes[k]->wLockRelease();
               WlockedLeafNodes[k] = NULL;
            }
            if (SlockedLeafNodes[k] != NULL){
               SlockedLeafNodes[k]->rLockRelease();
               SlockedLeafNodes[k] = NULL;
            }
         }
         Sleep(100);
         goto NAVEST_START0; //presmerovani zpet na zacatek
      }
      Sleep(waitMSec);
   }
   tryLockCount = 0;
   SlockedLeafNodes[hloubka] = NULL;
   WlockedLeafNodes[hloubka] = currentLeafNode;
   if (printLocks) printf ("zamknul jsem LeafNode %i Writelockem3 v hloubce %i, vlakno: %i\n",
      nodeIndex, hloubka, threadId);
```

Výpis 12: Příklad detekce uváznutí deseti neúspěšnými pokusy o získání zámku, případné odstranění dosud získaných zámků a následný ROLLBACK ve formě přesměrování zpět na začátek metody pomocí návěsti

4 Vyhodnocení implementace a testování

Výše uvedená implementace paralelního vkládání do datové struktury B-strom **je funkční**. Vložení 2000000 hodnot proběhne paralelně dvěma vlákny, přičemž všechny vkládané hodnoty jsou mezi tato vlákna rovnoměrně napůl rozděleny. Všechny možné případy uváznutí jsou ošetřeny, během testování tedy program vždy skončil korektně bez nekonečného deadlocku.

```
C:\Users\Vac\Desktop\bpsvn\src\test\paged\btree_test\Release\btreeWithIntIn...

---------------------------------------------
Fixed length B-tree test
DSMODE: DEFAULT
jsem ve fci insertTuples
loqueryorder: 666666
hiqueryorder: 1333332
jsem ve fci insertTuples
loqueryorder: 0
hiqueryorder: 666666
vytvoreno vlakno 538181216
vytvoreno vlakno 536477280
 0 675000 700000 725000 25000 750000 775000 800000 825000 850000 875000 900000 9
25000 950000 975000 1000000 1025000 1050000 1075000 50000 75000 1100000 1125000
100000 125000 150000 175000 1150000 200000 1175000 1200000 225000 250000 1225000
 275000 300000 325000 350000 375000 400000 1250000 1275000 425000 450000 475000
1300000 1325000 500000 525000 550000 575000 600000 625000 650000
5.91 (3.01563 (3+0.015625))s - Insert time
Performance: 226551.3 Inserts/s
```

Obrázek 4: Výstup výsledné aplikace - přeházené pořadí čísel označujících počet aktuálně vložených hodnot dokazuje, že vkládání bylo provedeno vícevláknově

4.1 Porovnání dosažené propustnosti s jednovláknovým vkládáním

Při deseti po sobě následujících testovacích spuštění výsledné aplikace v módu release na PC Lenovo IdeaPad Y580 (2-jádrové x64 CPU i5, 8GB RAM) byly při vkládání stejného počtu (2000000) položek stejného datového typu naměřeny hodnoty uvedené v tabulce 8.

Typ vkládání	⌀ čas vkládání	⌀ insertů za sekundu
jednovláknové	4,3	465 000
dvouvláknové	8,4	236 000

Tabulka 8: Výsledky testování vkládání do B-stromu jednovláknovým a dvouvláknovým průběhem

Tyto výsledky pro mne nejsou překvapivé. Slabý výkon a malá propustnost vícevláknového vkládání, v tomto konkrétním případě, jsou způsobeny:

- Především zamykáním celého B-stromu na místech, kde to původně nebylo plánováno, tedy hlavně v části vkládacího algoritmu nazvané State2
- Výkon práce s B-stromem nestojí na složitých výpočtech procesoru, ale na rychlosti operací s pamětí - ať by byla implementace sebeideálnější, k dvojnásobnému výkonu vkládání při zpracování dvěma vlákny by to stejně nevedlo
- Režie vláknové bezpečnosti nemalou měrou zpomaluje celý vkládací algoritmus, pro časová razítka jsou použity cykly while v různých variacích
- Taktéž je nutno připočíst režii na samotné vytváření jednotlivých vláken a rozdělování vkládaných hodnot mezi ně

5 Závěr

Má implementace zamykání vkládací metody ve frameworku QuickDB není dokonalá, naopak má vícero nedostatků.

Tím hlavním je, že se při řešení zamykání vnitřních uzlů objevil problém v části State2, která by musela být buď naimplementována jinak (nejspíše její duplikování a upravení do další State3, kterou bych volal pouze pro rodiče listových uzlů se zámkem jen na tohoto rodiče), nebo za stávající implementace musel být zamykán celý B-strom. V této bakalářské práci jsem použil druhou variantu.

Dalším pravděpodobným nedostatkem je, že rezervace B-stromu jednou transakcí může být provedena právě v okamžiku, kdy jiná transakce je uprostřed zápisu do listového uzlu, což by teoreticky mohlo způsobit ztrátu aktualizace.

Ideální také není krkolomné řešení časových razítek pro ROLLBACK způsobem, že se řídím tím, který cyklus dříve proběhne desetkrát.

Zamykání celého B-stromu by navíc mohlo být nahrazeno nějakou sofistikovanější metodou - například aby spolu transakce, potažmo vlákna, komunikovala prostřednictvím událostí dle pravidel nějakého předem definovaného uzamykacího protokolu.

Vyřešení těchto aspektů by znamenalo zefektivnění celého algoritmu a zvýšení propustnosti. V budoucnu bych na tomto chtěl pracovat v rámci diplomové práce.

Největší obtíže při vypracovávání této bakalářské práce pro mne bylo porozumět stávající implementaci školního frameworku QuickDB. Nakonec se ukázalo, že studování zdrojových kódů již hotové fungující aplikace pro mne bylo zároveň největším přínosem.

Tato práce byla mým prvním setkáním s paralelně pracující aplikací v takovémto rozsahu. Vypracování bylo náročné, ale přišel jsem na spoustu nových věcí, zejména na nesporné výhody dodržování pravidel OOP.

Dosud jsem byl zvyklý pracovat v jazycích vyšší úrovně. Domnívám se, že nebýt této bakalářské práce, do hlubin a zákoutí nižšího jazyka, jakým C++ je, bych se jinak nedostal.

Závěrem uvádím, že jsem v této bakalářské práci splnil všechny požadavky uvedené v zadání. Výjimkou je podrobné porovnání výkonnosti jednotlivých metod zamykání, které jsem v této práci uvedl, jelikož implementována byla pouze jedna z nich.

6 Reference

[1] Bayer, R., a Schkolnick, M., *Concurrency of operations on B-trees,*Acta Inf 9 (1977).

[2] Bayer, R. a McCreight, E., *Organization and maintenance of large ordered indexes,* Springer Berlin Heidelberg (2002).

[3] Bernstein, A. a Goodman, N., *Concurrency control in distributed database systems,*ACM Computing Surveys (CSUR) 13.2 (1981).

[4] Comer, D., *Ubiquitous B-tree,* ACM Computing Surveys (CSUR) 11.2 (1979).

[5] Dvorský, J. Ochodková, E. a Ďuráková, D., *Základy algoritmizace,* Technická univerzita Ostrava, http://www.cs.vsb.cz/kratky/courses/za/za.pdf (2004).

[6] Graefe, G. a Kuno, H., *Modern B-tree techniques,*Trends databases (2011).

[7] Kačírek, J., *R-stromy a jejich paralelizace,* Bakalářská práce. Ostrava, VŠB-TUO FEI (2013).

[8] Krátký M. a Dvorský J., *Úvod do programování (10. přednáška),* VŠB, Technická univerzita Ostrava (2004).

[9] Krátký, M., Bača, R., *Databázové systémy,*VŠB, Technická univerzita Ostrava (2010).

[10] Lehman, L., *Efficient locking for concurrent operations on B-trees,*ACM Transactions on Database Systems (TODS) 6.4 (1981).

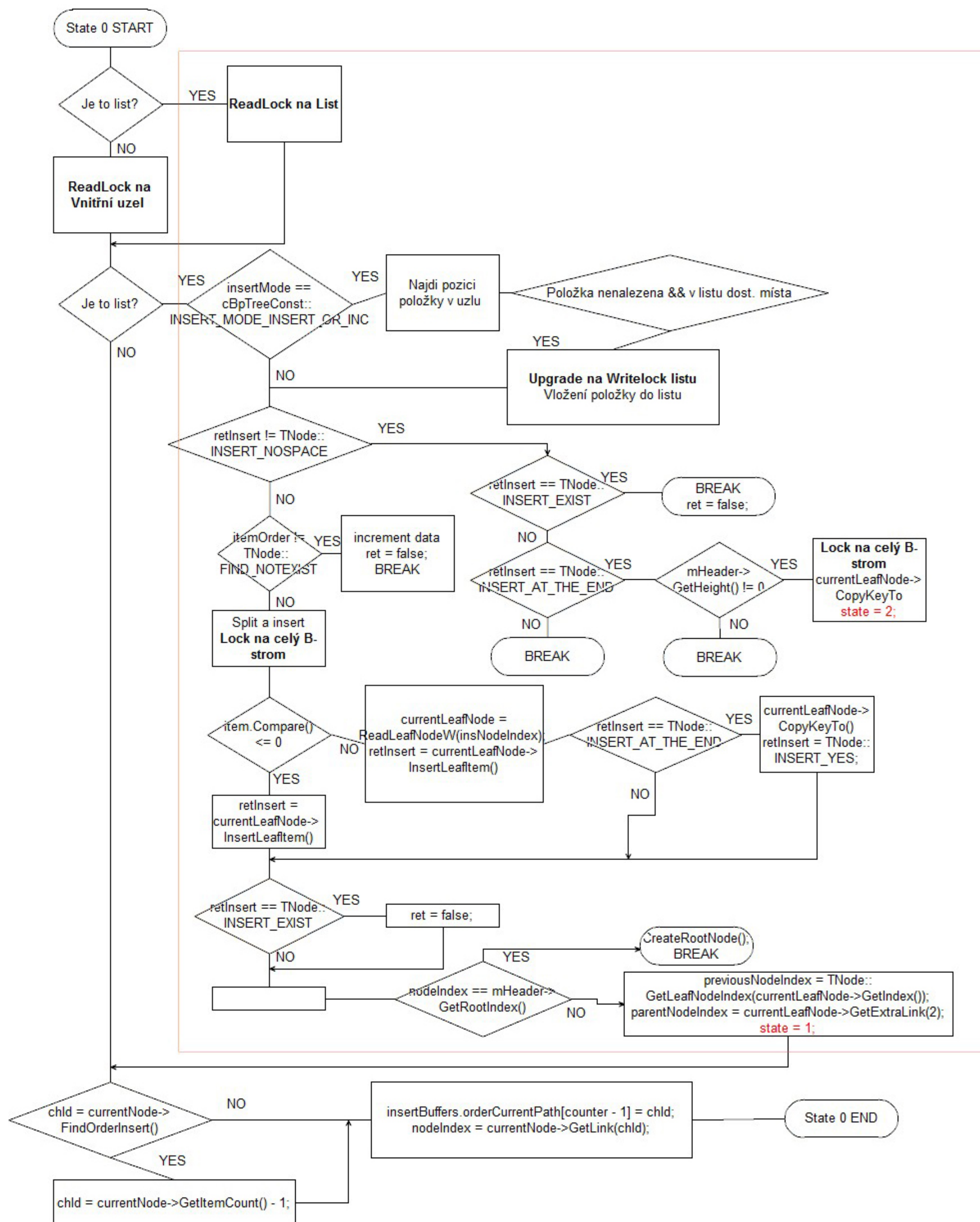
[11] Miller, R., a Snyder, L., *Multiple access to B-trees,*Johns Hopkins Univ., Baltimore (1978).

[12] Rosenkrantz, J., Richard E, a Philip M., *System level concurrency control for distributed database systems,*ACM Transactions on Database Systems (TODS) 3.2 (1978).

[13] Samadi, B., *B-trees in a system with multiple users,* Inf. Process. Lett. 54 (1976).

[14] Weiss, M., *Data Structures and Algorithms,* Benjamin/Cummings (1992).

A Vývojové diagramy



Obrázek 5: Vývojový diagram části vkládacího cyklu nazvané State0 (v červeném rámečku vyznačeny operace s listovým uzlem)

B Obsah CD

Přiložené CD obsahuje:

- adresář $codes$ obsahující všechny zdrojové soubory testovací aplikace
- adresář BP obsahující text této bakalářské práce ve formátu PDF