Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE

UNIVERZITA KARLOVA V PRAZE
matematicko-fyzikální fakulta

David Hoksza

Vícerozměrné indexování pro relační SŘBD

Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Tomáš Skopal, Ph.D.
Studijní program: Datové inženýrství

2006

Rád bych poděkoval RNDr. Tomáši Skopalovi za pečlivé vedení a čas strávený při odborných konzultacích.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 10. srpna 2006 David Hoksza

Obsah

Název práce: Vícerozměrné indexování pro relační SŘBD
Autor: David Hoksza
Katedra: Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Tomáš Skopal, Ph.D.
e-mail vedoucího: Tomas.Skopal@mff.cuni.cz

Abstrakt:

Cílem této práce bylo implementování vícerozměrné indexové metody do některého databázového systému a tuto indexovou metodu porovnat s již existujícími implementacemi vícerozměrných, případně jednorozměrných (nad více atributy), indexových metod nad stávajícími platformami (MSSQL, Oracle, ...).

Jako databázová platforma byla použita databáze PostgreSQL, která vyhovuje z hlediska možností integrování vlastních přístupových metod na úrovni modulů. Dále hrála také roli aktivní vývojová komunita kolem této platformy, která umožňovala případnou pomoc při problémech spojených s vývojem. A v neposlední roli pak přístup ke zdrojovému kódu PostgreSQL, který byl neocenitelným pomocníkem při snaze pochopit jádro PostgreSQL (především práci s pamětí), které má přímý vliv na fungování uživatelsky definovaných přístupových metod, konkrétně indexů.

Jako indexová metoda pak byla použita již existující implementace R-stromu nad objektovým frameworkem ATOM, který umožňuje implementovat persistentní stromové struktury.

Jako přímý důsledek práce vznikla nejenom implementace R-stromu, nýbrž obecný framework umožňující implementování externího indexování v PostgreSQL s minimální znalostí fungování této databázové platformy. Uživatel frameworku implementuje své persistentní vyhledávací metody a tyto metody pak připojí ke zmíněnému frameworku, který zajišťuje komunikaci mezi databází a danou naimplementovanou externí indexovou metodou.
Klíčová slova: indexování, relační databáze, B-strom, UB-strom, R-strom, PostgreSQL, ATOM

Title: Multidimensional indexing for relation databases
Author: David Hoksza
Department: Department of Software Engineering
Supervisor: RNDr. Tomáš Skopal, Ph.D.
Supervisor's e-mail address: Tomas.Skopal@mff.cuni.cz

Abstract:

The goal of this work was to implement a multidimensional indexing method into a database system and compare this method with existing implemtations of multidimensional indexing methods in current database platforms (MSSQL, Oracle, ...).

PostgreSQL was chosen for the implementation, because it suits the requirements of integrating own access methods in a modular way. The active development comunity which this platform unites has also played a part, since it assures help with problems related to the development. Last but not least, PostgreSQL was also chosen because its source code is accessible This was priceless in the effort of understanding the core of PostgreSQL (mainly work with memory), which has direct influence on user defined access methods, especially indexes.

An already existing implementation of R-tree over object framework ATOM was used as an indexing method, which allows implementing of persistent tree structures.

As a direct consequence of the work, not only has arised the implementation of R-tree in PostgreSQL, but also a generic framework which allows implementing external indexing in PostgreSQL with a minimum knowledge of how PostgreSQL works. The user of the framework implements his/her own persistent searchnig methods and connects these methods to the mentioned framework, which provides communication between database and the implemented extern indexing method.
Keywords: indexing, relational database system, B-tree, UB-tree, R-tree, PostgreSQL, ATOM

Kapitola 1

Úvod

Při přístupu k velkému množství dat vyvstává nutnost použití struktur pro rychlejší vyhledávání v databázi, zvláště pak v případě, kdy cílem dotazu je malé množství záznamů v porovnání s počtem všech záznamů. V takovém případě se může výrazně vyplatit neprocházet všechny záznamy a sledovat, zda splňují daná kritéria (sekvenční průchod), nýbrž mít data připravena v takové formě, nebo mít nad nimi vytvořeny takové struktury, které umožňují přistoupit pouze k malé podmnožině dat, které jsou vhodnými kandidáty na výsledek. Takové struktury nazýváme indexy.

Index je tedy struktura vytvořená nad existujícími daty tak, aby minimalizovala čas nutný k vyhledání relevantních dat. Snaží se minimalizovat počet nutných operací pro vyhledávání (porovnání prvků) a především pak počet diskových čtení, což je operace, která nejvíce přispívá k výslednému času vyhledávání.

Asi nejznámější indexovou metodou je B-strom. B-strom je metoda, která zajišťuje logaritmickou složitost vyhledání jednoho záznamu vzhledem k celkovému počtu záznamů. Je to metoda, která byla uvedena již v roce 1970 Rudolfem Bayerem. Má ovšem jeden nedostatek a tím je fakt, že ve své základní podobě dovoluje vyhledávat pouze podle jednoho atributu, což ovšem v dnešní době často není dostačující.

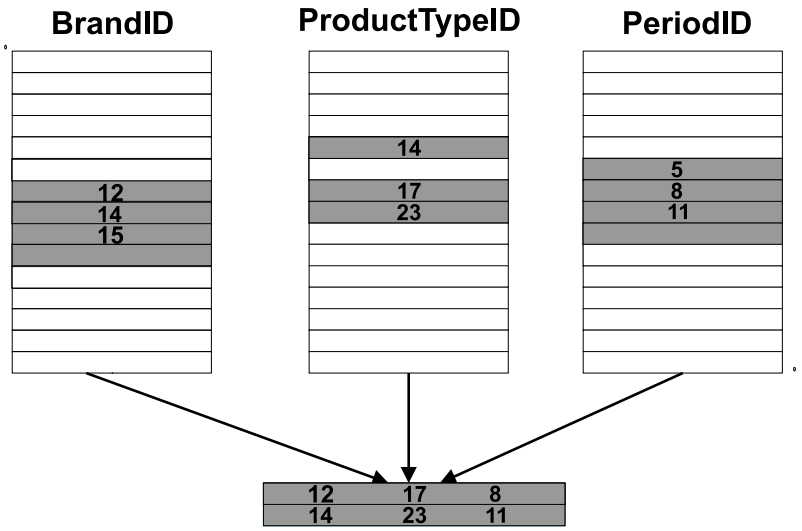
1.1 Formalizace problému

V současných databázových aplikacích v mnoha případech potřebujeme vyhledávat podle více než jednoho sloupce. Jmenujme například aplikace datových skladů, geografické informační systémy, dokumentografické informační systémy, archivační systémy atd. V těchto aplikacích je potřeba vyhledávat data podle více kritérií. Typický je dotaz na vyhledání všech prodaných produktů v prodejním systému podle pobočky, daného typu a v daném období, příkladně jednoduchý select tvaru:

```
SELECT * FROM Products WHERE
(BrandId>12 AND BrandID<15) AND
(ProductTypeID>12 AND ProductTypeID<25) AND
(PeriodID>=3 AND PeriodID<=11)
```

Je velice pravděpodobné, že výsledek takovéhoto dotazu bude velice malý vzhledem k počtu záznamů v tabulce *Products*. To je případ, kdy je výhodné použít nějaký typ indexu.

Nejjednodušší řešení spočívá v zaindexování každého sloupce, podle kterého chceme vyhledávat zvlášť a tyto výsledky následně spojit operací průniku. Tento princip je znázorněn na obrázku 1.1 (zvýrazněné řádky ve vrchních tabulkách odpovídají vyhledávacím podmínkám a spodní tabulka vyznačuje průnik všech tří atributů).



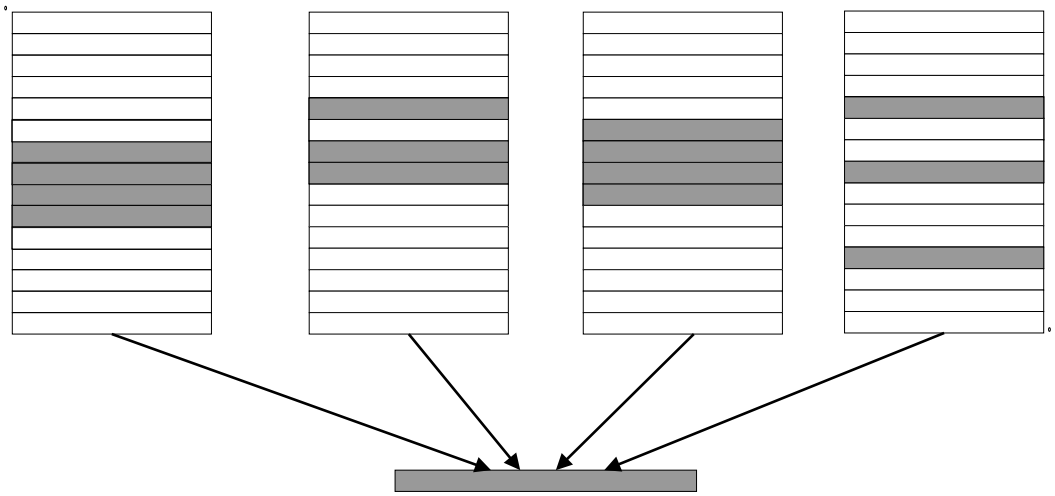
Obrázek 1.1: Průnik množiny výsledků indexů

Ovšem jak se bude situace měnit s přibývajícím počtem atributů, podle kterých bude chtít uživatel vyhledávat?

V reálných aplikacích chce uživatel v průměru velice málo dat. Tento průměr chce obvykle získat ať už vybírá záznamy podle jenoho atributu, nebo podle pěti (těžko lze předpokládat, že bude chtít načíst tisíce záznamů a pak v nich listovat). To znamená, že při rostoucím počtu atributů má být výsledný průnik přibližně stejně velký. Ovšem je-li průnik stejně velký pro více množin, pak roste velikost množin, z kterých průnik vzniká - pokud tyto množiny nerostou, pak nutně velikost průniku klesá. Tento princip je znázorněn na obrázku 1.2.

Z porovnání obrázků 1.1 a 1.2 můžeme vysledovat zásadní problém tohoto přístupu. Roste-li velikost množin, z kterých vytváříme průměr, pak také roste počet vybraných záznamů z jednotlivých množin (má-li se udržet stejná selektivita) a čím více roste počet množin, tím rychlejší je růst počtu záznamů, což je reálným projevem tzv. *prokletí dimenzionality*. Roste-li ovšem počet vybraných záznamů v poměru k počtu všech záznamů, pak se ztrácí význam indexu a začíná se nabízet úvaha, zda-li by nebylo lepší použít sekvenční průchod!

Další metodou (která je používána v současné době v komerčních databázových systémech), je rozšíření B-stromu tak, aby indexoval v uzlech složené klíče, tedy víceatributová data. Blíže o této metodě v kapitole 3.2.



Obrázek 1.2: Průnik množiny výsledků více indexů

B-strom se složenými klíči je sice efektivnější než složené indexy, ovšem přesto není pro indexování víceatributových dat příliš efektivní. Proto vyvstala potřeba metod, které budou pro indexování víceatributových dat vhodnější. Šlo především o indexování dvou a tří atributů, což bylo nutné pro rychlé vyhledávání v databázích geografických informačních systémů (GIS), nebo ve výkresových dokumentacích (CAD). V roce 1984 byla uvedena Guttmanem struktura zvaná R-strom (viz. [5]). Tato struktura byla daleko lépe uzpůsobena pro indexování prostorových dat (již v roce 1975 byly uvedeny KD-stromy založené na dělení prostoru ovšem tyto struktury dělí prostor daleko méně efektivně, než to dokáží R-stromy). Výhodou R-stromu při použití v této práci je skutečnost, že je automaticky rozšiřitelný do libovolné dimenze prakticky bez jakékoli změny (nepočítáme-li fakt, že místo se souřadnicemi dvojic či trojic, pracujeme obecně s n-ticemi).

Jak již bylo řečeno, stav v současných komerčních databázích je takový, že jsou pro víceatributové indexy použity pouze modifikace B-stromů. Například Oracle[1], nebo DB2[2] mají sice rozšíření, které umožňuje zavádět speciální grafické datové typy a nad nimi provádět efektivní indexování pomocí R-stromů, ale tyto rozšíření jsou určeny pro GIS/CAD aplikace. Tudíž neumožňují indexování nad více sloupci jednoduchých datových typů tak, aby bylo možné pro vyhledání dat výše zmíněným dotazem použít například právě R-strom, nebo jinou metodu specializovanou pro indexování víceatributových dat.

Jak lze vidět v tabulce 1.1, tak jediný současný databázový systém (alespoň z těch, které jsou alespoň minimálně v povědomí), který obsahuje efektivnější vícerozměrnou metodu indexování (konkrétně UB-strom), je Transbase (blíže o principu využití UB-stromu v Transbase v [2]). Oracle a PostgreSQL obsahují rozšíření, které umožňuje indexování R-stromem, ale pouze specializovaných datových typů (geometrií).

[1]Oracle Spatial Data Cartridge

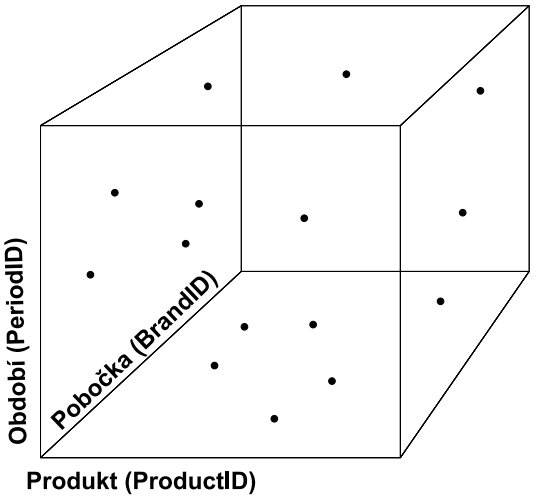
[2]Informix Spatial DataBlade Module

Databáze	Víceatributové indexy	Prostorové rozšíření
Oracle 9.i	B*-strom (složené klíče)	ANO (R-strom)
MS SQL Server	2000 B^{++}-strom [3] (složené klíče)	NE
PostgreSQL	B^{+}-strom (složené klíče)	ANO (R-strom)
Transbase	UB-strom	

Tabulka 1.1: Stav v současných DB

Cílem této práce bylo do některého ze stávajících systému (zvolen byl PostgreSQL) implementovat některou specializovanou metodu pro víceatributové indexování (zvolen byl R-strom) a umožnit tak velice jednoduše, se základní znalostí jazyka SQL a bez nutnosti znát specializované metody pro práci s prostorovými daty, indexovat záznamy přes více atributů. Bylo předpokládáno (což se také potvrdilo - viz. kapitola 7), že takovýto způsob přinese oproti klasickým B-stromům a jejich rozšířením nezanedbatelný přínos.

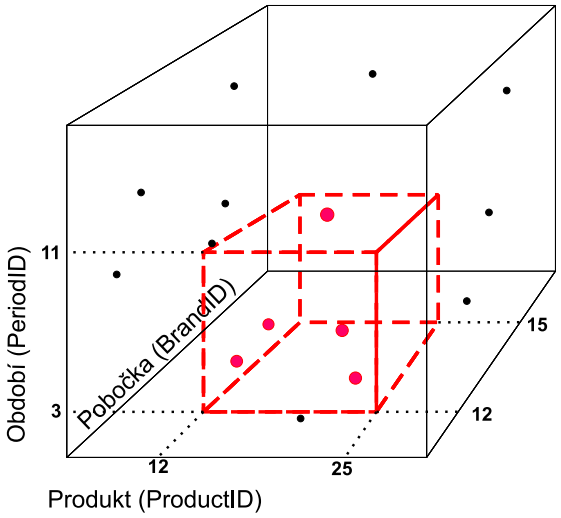
Prostorové rozšíření různých platforem má přirozeně svoje opodstatnění v tom, že umožňují indexovat rozličné polyobjekty, zatímco v této práci jsou indexovány pouze jednotlivé body. To vyplývá z typu indexovaných dat. Pro představu (blíže v kapitole 2.1) použití prostorových indexů pro účely této práce můžeme vzít příklad prodejního systému ze začátku této kapitoly a představu jeho dotazu. Zatímco R-stromy umožňují indexovat "libovolné" objekty tím, že je ohraničí "vícedimenzionálním kvádrem" a pak indexují tyto kvádry, tak pro naše účely stačí indexování bodů. Záznamy v prodejním systému se dají totiž graficky znázornit jako body v kvádru (mají-li 3 atributy) - viz. obrázek 1.3 (body reprezentují záznamy s danou pobočkou (*BrandID*), v daném období (*PeriodID*) a daného typu (*ProductID*)).



Obrázek 1.3: Záznamy v prostoru

[3] B^{+}-strom s provázanými uzly ve vnitřních úrovních - viz. kapitola 7.2.2

Dotaz pak můžeme znázornit jako menší kvádr obsažený v kvádru obsahujícím všechny záznamy (omezuje hodnoty, které jsou hranami reprezentovány). Takovou grafickou reprezentací dotazu je obrázek 1.4 (zvýrazněný kvádr reprezentuje dotaz a zvýrazněné body jsou záznamy, které dotazu odpovídají).



Obrázek 1.4: Dotaz v prostoru

Tímto způsobem tedy můžeme ve třech rozměrech graficky znázornit záznamy a dotazy nad nimi. Právě uvedený princip lze přirozeně zobecnit do dalších dimenzí.

Kapitola 2

Multidimenzionální indexování

2.1 Mapování multidimenzionálního prostoru

Pojem multidimenzionální (vícerozměrné) indexování je odvozen od myšlenky, že na data (záznamy) můžeme nahlížet jako na body vícerozměrného prostoru. Použijeme-li klasickou terminologii relačních databází, pak na každý řádek databáze nahlížíme jako na bod ve vícerozměrném prostoru, kde doménu dimenze prostoru reprezentuje atribut. Řádky tabulky jsou pak body v tomto prostoru. Tabulka reprezentuje podmnožinu zmíněného prostoru tvořeného kartézským součinem sloupců tabulky.

Možná nejlépe si to lze představit na příkladu třírozměrného prostoru, ve kterém můžeme reprezentovat tabulku o třech sloupcích, kde každý sloupec odpovídá jedné souřadnici a hodnoty na osách sloupců jsou mapovány na řádky tabulky (viz. obrázek 1.3).

2.2 Multidimenzionální data

Jednoduchým příkladem multidimenzionálních dat jsou data geografických informačních systémů, kdy dimenze indexovaného prostoru je 2 nebo 3. K indexování těchto dat je obvykle používána indexová struktura zvaná R-strom, nebo lépe některá z metod, které jsou z R-stromu odvozené (tj. R^*-strom, R^+-strom, ...). Multidimenzionálními daty ovšem také (nebo spíše především) rozumíme data, která existují ve vysokých dimenzích. Jsou to data, která můžeme "nějakým způsobem" interpretovat jako body vícerozměrného prostoru.

2.2.1 Multimediální aplikace

Klasický postoj k mapování multidimenzionálního prostoru zaujímají multimediální aplikace, které chápou svoje objekty jako body nalézající se ve vícerozměrném prostoru a tak k nim také přistupují.

Jako triviální příklad můžeme uvést obrázky mající rozměry 200x300 obrazových bodů (pixelů), kde každý bod může nabývat barev indexovaných od 0 do 255. Potom můžeme takový obrázek interpretovat jako bod v 60000 (200x300) dimenzionálním prostoru, kde

doména každé dimenze (počet možných hodnot v každé dimenzi) má velikost 256. Tento prostor tedy obsahuje všechny 8 bitové obrázky o velikosti 200x300 pixelů.

Výše zmíněná metoda je přirozeně pro praxi zcela nevhodná protože dimenze prostoru je příliš velká. Jako použitelné řešení zmíněného problému můžeme uvést histogram barev. Má-li obrázek 256 barev, pak je možné jej reprezentovat 256 čísly, kde každé značí počet výskytů pixelů s danou barvou v obrázku. Taková metoda není přirozeně zcela přesná, protože existuje více obrázku se shodným histogramem barev (například zaměníme-li 2 sousední pixely v obrázku), ale možné nepřesné výsledky vynahrazuje rychlost vyhledávání v takovém prostoru.

Proces získání nějaké vlastnosti, která popisuje jednotlivé objekty v databázi, nazýváme extrakce vlastností. V minulém příkladu to bylo získání histogramu, kde histogram je vlastnost obrázku. Taková extrakce je ztrátová, protože nepopisuje daný objekt jednoznačně. Sofistikovanější metodou extrakce může být získání kontur jednotlivých objektů z obrázku a ty pak nějakým způsobem reprezentovat. Získanou reprezentaci použijeme následně k indexování.

Obrazové databáze jsou často zaměřeny na nějaký konkrétní druh obrazových dat (otisky prstů, oční duhovky, rentgenové snímky, satelitní snímky, atd.), což umožňuje relativně dobře extrahovat typické vlastnosti zkoumaných dat, čímž můžeme vylepšit efektivitu indexování. Důvodem je fakt, že máme-li data, která mají podobnou strukturu, pak přesně víme, které informace extrahovat, což přináší menší ztrátu informace při extrakci. Víme totiž která data nejsou důležitá a můžeme se soustředit na znaky důležité, takže snižujeme dimenzi výsledného prostoru extrahovaných vlastností (zmenšujeme počet vlastností s kterými pracujeme) a zároveň snižujeme velikosti jednotlivých domén (pro každou vlastnost se zaměřujeme na její specifické projevy pro nás relevantní).

Multimediální aplikace se ovšem nezabývají pouze obrazem, nýbrž i například časovými řadami (které chápou jako multidimenzionální data), zvukovými záznamy, atd.

2.2.2 Klasické relační databáze

Multimediální databáze chápou svoje objekty jako speciální data, se kterými speciálně pracují. To je také případ použití R-stromů ve stávajících databázích, kde jsou používány na speciální datové typy (geometrie), což jsou objekty vícedimenzionálního prostoru.

V této práci ovšem přistupujeme k multidimenzionálním indexům jinak. Neindexuje jimi speciální datové typy prezentující vícedimenzionální objekty, nýbrž jimi indexujeme klasické záznamy relačních databází, ke kterým přistupujeme jako k multidimenzionálním objektům, kde roli dimenzí hrají atributy záznamu. Takový přístup pak umožňuje přirozeně použít existující metody vyvinuté pro multidimenzionální aplikace (konkrétně R-strom) a použít je v klasických relačních databázích.

Kapitola 3

Použité indexové metody

V této práci bylo pracováno s indexy, které jsou založené na stromových strukturách. Data jsou v takových strukturách obvykle uložena v listových uzlech (nebo jsou z listových uzlů odkazována) a jsou vyhledávána průchodem vnitřními uzly daného stromu (které u některých struktur můžou také obsahovat indexovaná data). Struktury se liší pravidly pro průchod vnitřními uzly a dále sémantikou datových položek.

3.1 B-strom

B-strom řádu m je výškově vyvážený vyhledávací m-ární (každý uzel má maximálně m následníků) strom, kde platí:

1. kořen má alespoň 2 potomky, pokud není listem
2. každý uzel kromě kořene má nejméně $\lceil m/2 \rceil$ a nejvíce m potomků
3. všechny větve jsou stejně dlouhé
4. každý uzel má nejméně $\lceil m/2 - 1 \rceil$ a nejvíce m datových položek
5. data v uzlu jsou organizována následovně:
   - $p_0, (k_1, p_1, d_1), \ldots (k_n, p_n, d_n)$
   - p ... ukazatele na potomky
   - k ... vzestupně (sestupně) uspořádané klíče
   - d ... asociovaná data
   - (k_i, p_i, d_i) ... datové položky
6. je-li $U(p_i)$ podstrom uzlu p_i, pak platí:
   - $\forall k \in U(p_{i-1}) : k < k_i$

- $\forall k \in U(p_i) : k > k_i$

Právě zmíněné body definují *neredundantní* B-strom. *Redundantní* B-strom má pozměněnou definici v bodech 5 a 6 následovně:

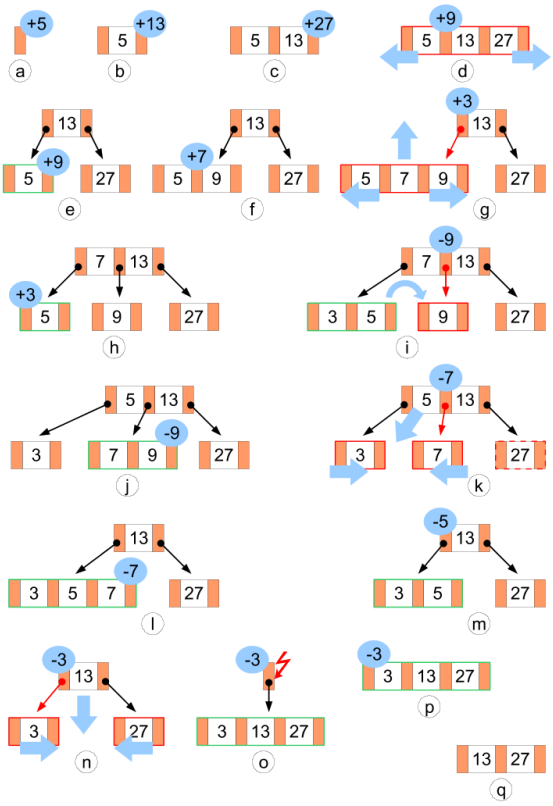
5. Data jsou umístěna pouze v listech, nebo jsou z listů odkazována.

6. je-li $U(p_i)$ podstrom uzlu p_i, pak platí:

   - $\forall k \in U(p_{i-1}) : k \leq k_i$
   - $\forall k \in U(p_i) : k > k_i$

Z definice (především z bodu 6) plyne způsob vyhledání položky ve stromu - začneme v kořenovém uzlu a porovnáme klíč, podle kterého vyhledáváme, s klíči v položkách a postoupíme do potomka daného uzlu podle pravidla definovaného v bodě 6. Takto projdeme celý strom až do listů, nebo se zastavíme v uzlu obsahujícím hledanou položku (podle toho, zda je B-strom redundantní nebo neredundantní).

V obrázku 3.1 (převzato z `http://commons.wikimedia.org/wiki/Category:B-Trees`) jsou pak názorně vidět operace vkládání a odebírání záznamu z B-stromu.



Obrázek 3.1: Fungování B-stromu

3.2 B-strom se složenými klíči

B-strom, jak byl zavedený v předcházející sekci, neumožňuje indexovat vícedimenzionální data. Nejjednodušší rozšíření B-stromu, aby mohl indexovat i vícerozměrná data, je chápat klíče jako složené (zřetězené) hodnoty klíčů jednotlivých dimenzí. Při porovnávání na klíči pak porovnáváme jednotlivé složky lineárně (jde vlastně o lexikografické uspořádání na řetězci čísel). Takovéto rozšíření používá většina databázových platforem pro indexování vícerozměrných dat pomocí B-stromu. Nevýhoda tohoto řešení spočívá v nutnosti zadávat celý klíč, nebo jeho prefix (není-li prováděna ještě následná konverze, při znalosti velikosti domény dimenze, která za nezadané hodnoty dosadí spodní a horní hranici domény - což obvykle není).

Největší problém přístupu se složenými klíči je asymetrie v pořadí atributů. První atribut je totiž hlavní atribut, podle kterého je prostor setříděn (shlukován). Z toho plyne, že v indexu jsou záznamy setříděny podle prvního atributu, ale ne už podle ostatních. Podle dalšího atributu jsou setříděny pouze v případě, kdy je první atribut duplicitní a právě toto je důvod, proč se doporučuje při indexování složenými klíči na první místa v klíči dávat atributy s malou doménou (zvyšuje se tím počet duplicit).

Asymetrie pak způsobí při rozsahovém dotazu nutnost prohlížet mnoho větví stromu a tím výrazně snižuje efektivitu dotazu zvýšením počtu procházených stránek.

Řešením problému s asymetrií prvního atributu je pouze udržování vícero B-stromů pro jednu množinu atributů - pro každé pořadí atributů jeden. Takový přístup je ovšem nereálný z hlediska růstu počtu nutných indexů vhledem k počtu indexovaných atributů (pro 4 atributy už je to 24 indexů) a vlastně odpovídá způsobu indexování s více indexy, který je neefektivní.

3.3 R-strom

R-strom je přímá multidimenzionální generalizace B-stromu, která si udržuje stejně jako B-strom výškovou vyváženost a minimálně poloviční využití uzlů stromu. Indexuje prostorové objekty, ať už body nebo složitější objekty. Pro jednoduchou reprezentaci nepravidelných útvarů zavádí pojem minimální ohraničující kostky - *MBR* (**m**inimal **b**ounding **r**ectangle). Tato kostka je ve dvourozměrném prostoru nejmenší obdélník nebo čtverec, který dokáže pojmout daný objekt. V třírozměrném prostoru je to krychle nebo kvádr. A podobně je MBR zobecněn do vyšších dimenzí. Příklad MBR lze vidět na obrázku 3.2

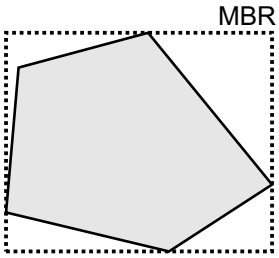
V R-stromu rozlišujeme 2 typy uzlů:

- *Listové* uzly
  - obsahují MBR indexovaných objektů.
- *Vnitřní* uzly
  - obsahují *regiony*, což jsou oblasti, obsahující MBR všech svých potomků. Regiony se mohou překrývat.

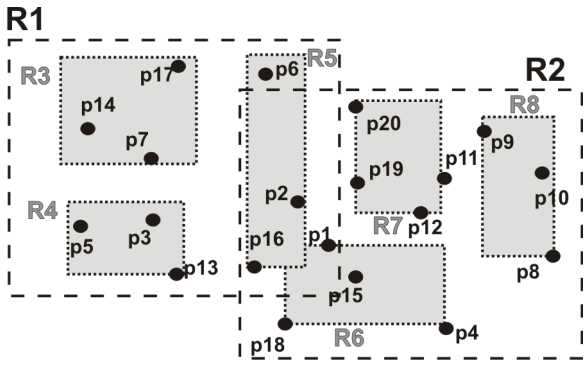
3.3.1 Vyhledávání

Vyhledávání objektu v R-stromu probíhá stejně jako v B-stromu až na skutečnost, že při vyhledávání jednoho objektu, může být prohledáno více podstromů. Tato skutečnost je způsobena faktem, že regiony se můžou navzájem překrývat. Leží-li tudíž objekt v překryvu, pak nelze rozhodnout, v kterém z podstromů se objekt skutečně nachází a musí být prohlédnuty všechny podstromy, které se překryvu účastní (příkladem toho může být vyhledání bodu p2 v obrázku 3.3 - musíme se podívat do R1 i R2).

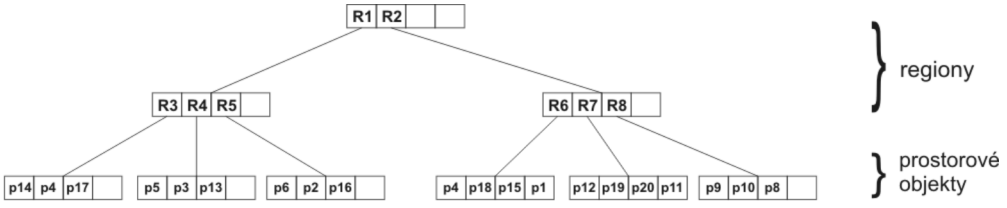
Vyhledáváme-li pomocí vyhledávacího okna, tj. nevyhledáváme jeden konkrétní bod, ale všechny body ležící ve vymezeném prostoru (okně), pak při procházení stromem musíme projít vždy všechny podstromy reprezentované pomocí MBR, které mají neprázdný průnik s daným vyhledávacím oknem.



Obrázek 3.2: MBR



Obrázek 3.3: Dělení prostoru pomocí regionů a MBR



Obrázek 3.4: R-strom příslušný k obrázku 3.3

3.3.2 Vložení objektu

Pro rozhodnutí, kam vložit objekt je důležitá minimalizace následujících veličin:

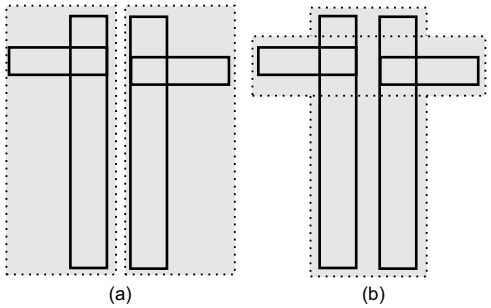
- *Pokrytí* úrovně stromu, tj. celková plocha regionů v dané úrovni stromu.
- *Přesah* na úrovni stromu, tj. celková velikost průniků ploch regionů.

Minimalizujeme-li pokrytí, pak tím také minimalizujeme tzv. *mrtvou plochu*, tj. plochu, která neobsahuje objekty. Tím se snižuje celkový prohledávaný prostor, který jistě neobsahuje objekty.

Minimalizujeme-li na druhé straně přesah, pak snižujeme pravděpodobnost prohledávání více podstromů, jelikož je menší pravděpodobnost, že hledaný objekt bude zasahovat do překrývajících se regionů.

Na základě znalosti faktů o pokrytí a přesahu můžeme zajistit vkládání takové, které bude tyto veličiny minimalizovat. Tedy listový uzel, kam vložit objekt, je nalezen tak, že je procházen strom od kořene a v každém uzlu se při rozhodnutí, do kterého potomka bude objekt vložen, řídíme pravidlem, že je vybrán takový uzel, který potřebuje nejmenší rozšíření, pokud do něj bude nový objekt vložen. V případě, že tomuto kritériu vyhovuje více uzlů, je vybrán ten, jehož výsledná plocha bude nejmenší. Takto rekurzivně dojdeme až do listu. Pokud není zaplněn, pak je do něj objekt vložen. V opačném případě dochází na dělení regionu.

Dělit region lze mnoha způsoby a jelikož dělení prostoru je pro efektivitu vyhledávání důležité, je třeba vybrat dělení co možná nejlepší, ovšem s ohledem na časovou složitost této operace. Příklad špatného a správného dělení lze vidět na obrázku 3.5 (a - špatné dělení, b - dobré dělení).



Obrázek 3.5: Dělení regionu

Uveďme 3 nejznámější způsoby dělení uvedené Guttmanem v [5] (všechny se snaží minimalizovat plochu regionů vzniklých po dělení):

- Úplný algoritmus procházející všechny možnosti
  - Nevýhodné z hlediska časové složitosti (při kapacitě uzlu M je to 2^{M-1})

- Kvadratický algoritmus
  - Kvadratický v kapacitě uzlu a lineární v počtu dimenzí
  - Nezaručuje optimální rozdělení
  - Rozděluje objekty postupně do dvou od sebe maximálně vzdálených skupin (nových regionů)
- Lineární algoritmus
  - Lineární v kapacitě uzlu a lineární v počtu dimenzí
  - Nezaručuje optimální rozdělení
  - Princip je stejný jako u kvadratického algoritmu, pouze se mění způsob vybrání objektu, který bude v daném kroku přiřazen k jedné z vytvářených skupin

3.3.3 Vyjmutí objektu

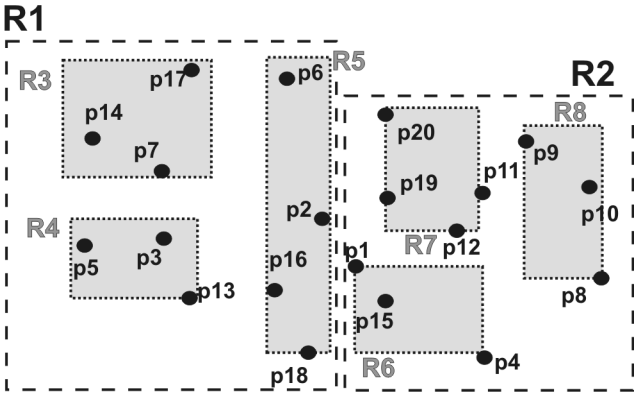
Při vyjmutí objektu a podtečení uzlu (počet položek je menší než polovina kapacity uzlu) nepoužíváme distribuci do sousedních uzlů jako v B-stromu, nýbrž všechny objekty regionu definovaného podtečeným uzlem vyjmeme ze stromu, znovu je vložíme a region vymažeme. Mohli bychom taky pro všechny přesouvané objekty zjistit s jakým sousedem by tvořily nejmenší rozšíření, jako při vkládání, a do toho regionu objekt vložit. Způsob znovuvložení je obvykle volen z následujících důvodů:

- Použít proceduru pro vkládání je jednodušší z implementačních důvodů
- Uzly, které budeme procházet při znovuvložení, již máme v paměti
- Znovuvložení lépe distribuuje objekty do správných regionů, než přesun k sousedům

3.4 R^+-strom

R^+-strom vznikl z R-stromu přidáním kritéria, že regiony mají nulový přesah. Tímto způsobem eliminuje prohledávání více podstromů v případě, že by objekt ležel právě v oblasti, kde se uvažované regiony pro prohledávání protínají. Na obrázku 3.6 můžeme vidět dělení prostoru s nulovým přesahem na stejné množině bodů, jako na obrázku 3.3.

Problém nastává u této metody hlavně ve vysokých dimenzích, kdy roste počet regionů. Když se totiž dva regiony protínají, tak je třeba je dělit. To se ve vysokých dimenzích stává relativně často a tudíž narůstá počet dělení, resp. počet regionů, resp. výška stromu. S rostoucí výškou stromu pak roste i čas nutný k vyhledání objektu.



Obrázek 3.6: Dělení prostoru v R^+-stromu

3.5 R*-strom

R*-strom je modifikace R-stromu, která vznikla v roce 1990 (viz. [6]) po analýze nevýhod R-stromu a pokusu o jejich řešení. Hlavní determinanty výkonu R-stromu jsou přesah a pokrytí. R*-strom se snaží minimalizovat obě tyto veličiny tím, že přidává další optimalizační kritérium a tím je *obvod MBR*. Preferuje čtvercové *MBR*. Intuitivně jde o to, že čtvercové *MBR* jsou kompaktnější a tudíž minimalizují přesah.

V listových uzlech R*-stromu je záznam vložen do uzlu, jehož výsledný *MBR* bude minimalizovat přesah s ostatními *MBR*. Shoda je řešena tak, že je vybrán uzel, který potřebuje nejmenší rozšíření plochy. Ve vnitřních uzlech je ovšem kritérium jiné. Vybrán je uzel, který potřebuje nejmenší rozšíření plochy a shoda je řešena tak, že je vybrán uzel, který má nejmenší výslednou plochu (což jsou právě čtverce).

Přetečení uzlu je řešeno znovuvložením části záznamů uzlu, což má příznivý vliv na distribuci záznamů ve stromě, ale v případě velikého stromu může také výrazně zvyšovat čas nutný pro vložení záznamu.

3.6 UB-strom

Roste-li výrazněji počet dimenzí, pak jsou R-strom a jeho alternativy technicky použitelné ovšem prakticky nevykazuje příliš optimální výsledky a to především proto, že roste počet překryvů a tudíž je třeba projít příliš velký počet podstromů. Počet procházených podstromů roste s dimenzí prostoru, který strom pokrývá.

Jedním z řešení je transformovat body z n-dimenzionální prostoru do prostoru jednodimenzionálního a indexovat tyto body klasickými jednorozměrnými indexovými strukturami.

Jde tedy o to, proložit n-dimenzionálním prostorem křivku takovým způsobem, aby zůstala co nejlépe zachována vzdálenost bodů. Jsou-li 2 body blízko u sebe v n-dimenzionálním prostoru, pak by měly být blízko sebe i na výsledné křivce. Přesná vzdálenost bodů je irelevantní - jde pouze o zachování poměrů vzdáleností (přirozeně je výhodné a obvykle

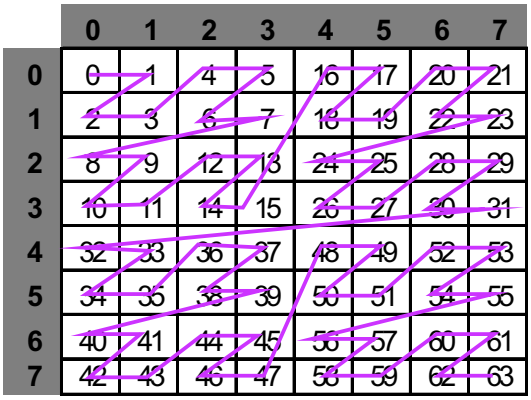
potřebné mít možnost provést zpětnou transformaci bodů do n-dimenzionálního prostoru). Tedy stručně řečeno musí být možné body v prostoru lineárně uspořádat (což ve vektorovém prostoru lze).

Výše zmíněný způsob používá struktura UB-strom (Univerzální B-strom), která pro transformaci používá Z-křivku. UB-strom byl definován R.Bayerem v roce 1996 v [7].

3.6.1 Z-křivka

Jednou z nejznámějších křivek, kterou lze proložit prostor a která zachovává vzdálenost bodů je Z-křivka, nebo také Peánova křivka. Názorně ji můžeme pro dvourozměrný prostor vidět na obrázku 3.7 (a - číslování prostoru, b - Z-křivka):

	0	1	2	3	4	5	6	7
0	0	1	4	5	16	17	20	21
1	2	3	6	7	18	19	22	23
2	8	9	12	13	24	25	28	29
3	10	11	14	15	26	27	30	31
4	32	33	36	37	48	49	52	53
5	34	35	38	39	50	51	54	55
6	40	41	44	45	56	57	60	61
7	42	43	46	47	58	59	62	63



Obrázek 3.7: Z-křivka

3.6.2 Z-adresa

Vrcholy křivky reprezentují body v prostoru. Cílem transformace je převést tyto body do jednorozměrného prostoru, tedy každý z nich reprezentovat právě jedním číslem. Toto číslo se nazývá *Z-adresa*. Definici uvedeme pro obecně n-dimenzionální prostor:

Definice 1 *Mějme bod x v prostoru dimenze d a definujeme jeho atributy binárně jako:*

$$x_i = x_{i,0}, \ldots, x_{i,s}, i \in \langle 1; d \rangle$$

pak Z-adresu *definujeme jako bijektivní funkci:*

$$Z(x) = \sum_{j=0}^{s-1} \sum_{i=1}^{d} x_{i,j} * 2^{j*d+i-1}$$

3.6.3 Z-region

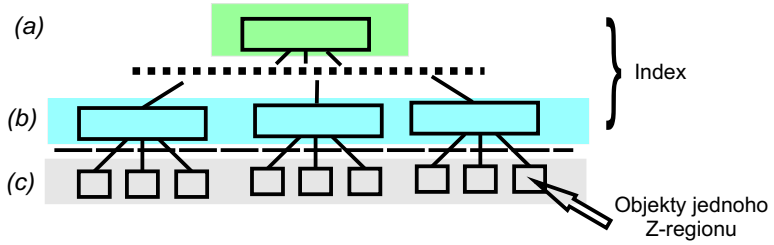
Z-region je množina bodů v prostoru definovaná intervalem $[\alpha; \beta]$. Na obrázku 3.8 můžeme vidět Z-region definovaný intervalem [3;22]. Body v jednom regionu jsou umístěny v jedné stránce, což je vzhledem k shlukovací podstatě Z-křivky výhodné pro intervalové dotazy, kde tato vlastnost minimalizuje počet stránek načítaných z disku.

t	0	1	2	3	4	5	6	7
0	0	1	4	5	16	17	20	21
1	2	3	6	7	18	19	22	23
2	8	9	12	13	24	25	28	29
3	10	11	14	15	26	27	30	31
4	32	33	36	37	48	49	52	53
5	34	35	38	39	50	51	54	55
6	40	41	44	45	56	57	60	61
7	42	43	46	47	58	59	62	63

Obrázek 3.8: Z-region

3.6.4 Využití Z-křivky v UB-stromu

UB-strom je použití výše popsaných principů týkajících se Z-křivky v klasickém B-stromu. Za klíče volíme adresy Z-regionů. Za adresu Z-regionu $[\alpha;\beta]$ bereme číslo β. Dělení prostoru na regiony vzniká tak, že jako spodní hranici regionu s adresou β bereme adresu regionu předcházejícího. Uložení dat a Z-regionů v UB-stromu můžeme vidět na obrázku 3.9 (a - vrcholové dělení prostoru obsahující v následnících dělení jemnější, b - stránky obsahující všechny Z-regiony (poslední vrstva), c - stránky se zaindexovanými objekty).



Obrázek 3.9: Schéma UB-stromu

Kapitola 4

Databázová platforma PostgreSQL

PostgreSQL je relační databáze, jejíž základy byly položeny na konci 70. let na univerzitě v Berkeley jako databáze s názvem Ingres. Tato databáze byla komercializována společností Computer Associates. Roku 1986 Michael Stonebraker z Berkeley vedl tým, který přidal do jádra Ingresu objektově-orientované vlastnosti a vznikl Postgres. I tato verze byla zkomercializována - tentokrát společností Illustra, která byla následně odkoupena společností Informix.

V polovině 90. let byla do Postgresu přidána podpora SQL (předchozí verze využívaly vlastní jazyk nazvaný Postquel). V roce 1996 bylo přidáno do Postgresu mnoho nových vlastností jako MVCC (Multi-Version Concurrency Control), korektní podpora SQL92, vylepšený výkon a Postgres byl přejmenován na PostgreSQL.

Pro PostgreSQL jsou typické následující vlastnosti:

- Je objektově-relační. V PostgreSQL každá tabulka definuje třídu a mezi třídami, resp. tabulkami existuje dědičnost. Funkce a operátory jsou polymorfní.
- Podporuje standard SQL92 a některé prvky SQL99.
- Je Open source. PostgreSQL udržuje a vyvíjí mezinárodní tým programátorů, jehož jádro tvoří lidé vyvíjející PostgreSQL od roku 1996.
- Podporuje transakce.
- *Klade důraz na rozšiřitelnost.* Do Postgresu lze implementovat vlastní datové typy, operátory, procedurální jazyky, *přístupové metody*, a další.

Původně byla tato databázová platforma vyvíjena pouze pro UNIXové prostředí (a pro Linuxové klony). Od verze 8.0 (začátek roku 2005) je PostgreSQL již plnohodnotně použitelný i pod prostředím Windows (dříve bylo toto možné pouze prostřednictvím emulace Linuxu pod Windows). Nicméně z průzkumu na webových stránkách PostgreSQL plyne, že 35% uživatelů pracuje s PostgreSQL pod Windows (větší podíl má pouze Linux - 40%).

PostgreSQL je vyvíjen v jazyce C a většina vývojářů používá pro vývoj Linux. Zdrojový kód je pak kompilován do většiny UNIXových platforem a do Windows (2000, XP). Z důvodu orientace většiny vývojářů na Linux jsou k vývoji potřeba GCC, GDB, autoconf, GNU make (přirozeně lze kompilovat i jiným kompilátorem než GCC, ale většinou je používán právě tento). Z toho důvodu je třeba při kompilaci kódu pod Windows pracovat s minimální emulací systému Linux, tedy s programem MingW (http://www.mingw.org/).

Pod tímto systémem lze vyvíjet stejným způsobem, jako v Linuxu (ovšem je omezen pouze na příkazovou řádku). Důležitá je především možnost ladění systému pomocí GDB, což byla při této práci neocenitelná pomůcka s ohledem na fakt, že jádrem indexových metod je manipulace s pamětí a adresami do ní odkazujícími. Silným nedostatkem je ovšem existence pouze textové verze GDB, která nenabízí zdaleka takový komfort, na jaký je dnešní programátor zvyklý a výrazně prodlužuje nutný čas vývoje.

Kapitola 5

Obecné požadavky na systém používající uživatelsky definované indexové metody

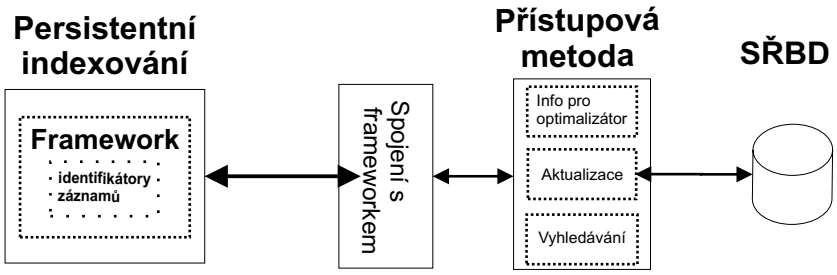
Databázová platforma, která má ambice umožňovat implementování uživatelsky definovaných indexových metod, musí poskytovat vhodné rozhraní, pomocí kterého je metoda schopna:

1. předávat optimalizátoru informace o očekávané časové náročnosti implementované indexové metody, aby se ten mohl rozhodnout o jejím případném použití. Informace předávané optimalizátoru by měly obsahovat následující:
   - počet očekávaných navštívených záznamů při vyhledávání
   - počet očekávaných načtených indexových stránek
   - cena nalezení prvního záznamu
   - cena nalezení následujícího záznamu
2. být v aktualizovaném stavu vzhledem k obsahu relace, nad kterou je index vybudován (definice *být aktualizován* závisí na databázové platformě), tj. poskytovat možnost notifikace na změnu dané relace. Sledované údaje by měly obsahovat:
   - vložení záznamu do relace
   - odebrání záznamu z relace
   - změna stávajícího prvku (především klíče užitého k indexování, nebo jeho části)
   - smazání relace (implikuje odebrání indexu a jeho případných podpůrných struktur)
3. vyhledávat nad danou relací. Funkce rozhraní by měly zahrnovat:

- inicializace vyhledávání (pro inicializaci struktur v závislosti na vyhledávacím klíči)
- nalezení následujícího prvku v probíhajícím vyhledávání
- ukončení vyhledávání
- finalizace (úklid)

Výše naznačené rozhraní je základem, který by platformy usilující o možnost uživatelské implementace indexových metod měli podporovat, nicméně není možné je brát doslova. Některé platformy mohou jisté body slučovat do jednoho, nebo naopak jednotlivé body dělit do více specializovaných úloh.

Řešením je tedy sada funkcí, která odstiňuje implementátora indexové metody do maximální možné míry od fyzických specifik dané databázové platformy. Tomu odpovídá i fakt, že ve výše zmíněném rozhraní zcela chybí výsledné prohledávání fyzických relací na disku, apod. Indexové metody obvykle pracují pouze s vnitřními identifikátory záznamů jednotlivých relací a tyto relace si předávají s fyzickou vrstvou, která pracuje pod nimi a která zajišťuje manipulaci s fyzickými záznamy. Schématické znázornění výše naznačeného přístupu lze vidět na obrázku 5.1.



Obrázek 5.1: Schéma propojení SŘBD s uživatelsky definovaným persistentním indexováním

Kapitola 6

Indexování v PostgreSQL

V následující části bude popsáno jakým způsobem PostgreSQL ukládá záznamy a indexy nad nimi a jaký mechanismus používá pro jejich vyhledání za pomoci indexu.

6.1 Haldové a indexové relace (vnitřní struktura indexu)

Relace v PostgreSQL jsou dvou typů - *haldové* relace (*HR* - Heap Relation) a *indexové* relace (*IR* - Index Relation). Hlavním úložným typem jsou HR. V HR jsou uložené jak všechny uživatelské relace, tak systémové katalogy. V haldě jsou HR uloženy logicky neuspořádaně. Samotná halda se skládá z bloků o konstantní velikosti, kde každý blok obsahuje žádný nebo více záznamů. Indexové relace obsahují dvojice *klíč-hodnota*, kde *klíč* umožňuje rychlé vyhledání jednoho nebo více záznamů a *hodnota* odkazuje na příslušný záznam na haldě.

6.1.1 Identifikátory záznamů

Záznam je adresován identifikátorem záznamu (*TID* -Tuple Identifier). TID se skládá z dvojice $< blok; offset >$. Mluvíme-li obecně při indexování o stránkách, pak v PostgreSQL mluvíme o blocích.

Blok s indexem i značí i-tý blok dané relace a pro každou relaci jsou bloky číslovány od nuly. Číslování je sekvenční a tudíž je-li třeba rozšířit relaci, pak je alokován nový blok s číslem o jedna větším než poslední alokovaný blok relace.

Offset v TID je offset záznamu v rámci bloku. Offsety pro každý blok jsou číslovány sekvenčně, podobně jako bloky pro relaci, ale na rozdíl od bloků začíná číslování od 1. Počet offsetů v bloku se liší přirozeně podle velikosti záznamu relace. Ovšem i počet offsetů v jednotlivých blocích stejné relace se může lišit (obsahuje-li relace datové typy o proměnné délce).

V minulosti záznamy v průběhu své existence nikdy neměnili své TID. V současnosti, kvůli podpoře současného běhu více uživatelů nad stejnou relací, jsou záznamy při updatu

kopírovány a tudíž mění svoje TID. Ovšem ve struktuře záznamu je uložen odkaz na novou lokaci a tudíž lze přes takto vzniklý řetěz "vystopovat" relaci až k současnému stavu. Při vymazání záznamu nejsou ostatní záznamy posunuty (nemění TID), nýbrž je místo smazaného záznamu rezervováno pro použití některým novým záznamem.

6.1.2 Struktura dat IR

Každý index v PostgreSQL je složen ze záznamů následujících typů:

```
typedef struct IndexTupleData
{
    ItemPointerData t_tid;
    unsigned short t_info;
}
```

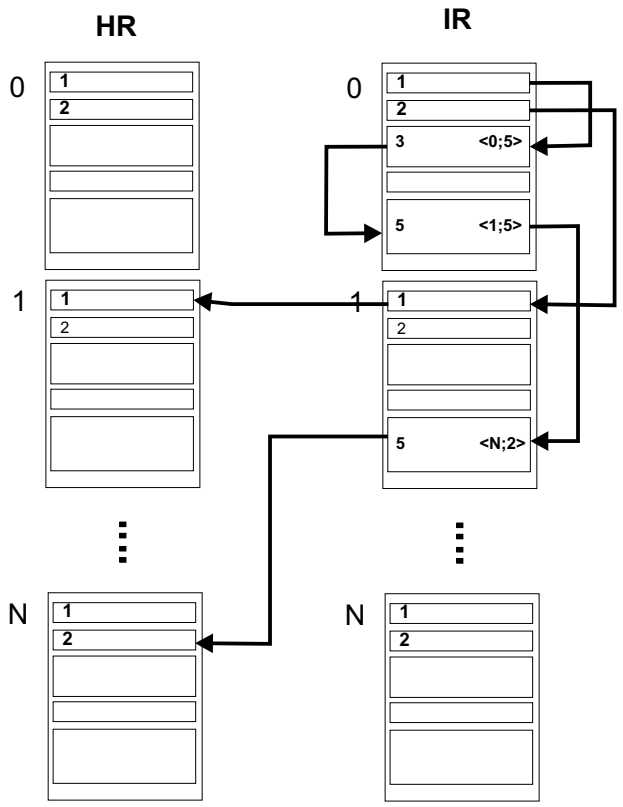
Význam položek je následující:

- *t_tid* - reference na záznam HR
- *t_info* - bitově orientovaná proměnná, jejíž bity mají následující hodnoty
  - 15 (horní): může obsahovat *null* hodnotu
  - 14: má atributy s proměnnou strukturou
  - 13: nepoužito
  - 12-0: velikost záznamu v IR

Dolních 12 bitů *t_info* určuje velikost záznamu a to z toho důvodu, že za strukturou *IndexTupleData* se nachází oblast obsahující klíčovou hodnotu a uživatelsky definovaná data. Tudíž velikost různých IR není konstantní. Oblast obsahující uživatelsky definovaná data je pro indexy životně důležitá, protože sem mohou ukládat informace o své struktuře a používat je pro svoje účely při vkládání a vyhledávání.

Záznamy IR dělíme na záznamy *interní* a *externí*. Externí záznamy jsou ty záznamy, jejichž hodnota *t_tid* struktury *IndexTupleData* se odkazuje do HR. Hodnota *t_tid* interního záznamu naproti tomu odkazuje znovu do IR. Názorně to lze vidět na obrázku 6.1, kde indexové záznamy v pravé části se odkazují jak do struktury indexu (interní), tak do struktury relací, které indexují (externí). Tímto způsobem můžou například stromové indexové metody jako je B-strom, nebo R-strom v HR vytvořit svoji strukturu. Paměť uživatelské části následující za *IndexTupleData* obsahuje právě data určující pomocné informace k dané struktuře. Může jít například o informace typu:

- Identifikace uzlu do kterého patří daný záznam
- Číslo hladiny stromu, kde se nalézá daný uzel
- Zda je uzel uzlem listovým a tudíž je jeho hodnota *t_tid* hledanou hodnotou, kterou je třeba vrátit.



Obrázek 6.1: Haldové a indexové relace

6.2 Vlastnosti indexu

Indexy v PostgreSQL se můžou lišit ne pouze, co se struktury týče, ale i vlastnostmi, které podporují. Každý index může podporovat následující vlastnosti:

- Podpora směru vyhledávání
- Podpora unikátního indexování (*unique index*)
- Podpora víceatributového indexování
- Podpora indexování bez podmínky na prvním indexovacím sloupci (pro indexy nepodporující víceatributové indexování to značí podporu sekvenčního průchodu celou tabulkou - *full table scanu*)
- Podpora indexování *null* hodnot
- Podpora současného updatu více uživatelů

Záznam o indexu je zanesen v katalogové tabulce *pg_am* (viz. dodatek B). Každá indexová metoda je povinna při svém zaregistrování založit záznam v této tabulce a tím i definovat svoje možnosti vzhledem k výše zmíněným vlastnostem.

6.3 Vytvoření indexu

Pro vytvoření vlastního indexu a jeho použití v PostgreSQL je třeba následujících kroků:

1. Implementovat množinu funkcí definující rozhraní, které jádro PostgreSQL používá ke komunikaci s indexem a které podporuje vlastní funkcionalitu.
2. Zaregistrovat funkce vytvořené v bodu 1 v PostgreSQL.
3. Využít funkce z bodu 2 při zaregistrování indexové metody.
4. Vytvořit třídu operátorů definující, které operátory a které typy registrovaná metoda podporuje.
5. Použít index nad sloupci tabulky obsahující typy pro něž existuje třída operátorů z bodu 4.

6.3.1 Implementace rozhraní

Rozhraní vyžadované jádrem PostgreSQL, aby bylo schopno s danou indexovou metodou komunikovat, obsahuje následující funkce (lze je pojmenovat libovolně, ale konvence je taková, že názvy jsou stejné, jako v následujícím seznamu a liší se v různých metodách pouze prefixem - přesné rozhraní a popis viz. dodatek A):

- *index_build*
  - vytvoření indexu (obvykle vytvoří strukturu indexu, tj. například strom)
- *index_insert*
  - vložení záznamu do indexu (záznam je atributem funkce)
- *index_beginscan*
  - započetí vyhledávání
- *index_gettuple*
  - získání jednoho záznamu vyhovujícího vyhledávacím podmínkám
- *index_getmulti*
  - získání zadaného množství záznamů vyhovujících vyhledávacím podmínkám
- *index_endscan*
  - ukončení vyhledávání

- *index_markpos*
  - označení aktuální pozice ve vyhledávání (*scanu*)
- *index_restrpos*
  - navrácení se k vyznačené pozici
- *index_rescan*
  - opakování vyhledávání se stejnou strukturou klíčů (ale případně jinými hodnotami - používá se například ev spojeních)
- *index_bulkdelete*
  - odebrání množiny prvků z indexu
- *index_costestimate*
  - odhadnutí ceny vyhledávání

Zmíněné funkce musí podporovat každá indexová metoda. Všechny potřebné údaje jsou předávány funkcím přes jejich parametry jádrem PostgreSQL.

Funkce mohou být implementovány v libovolném jazyce, který je v PostgreSQL podporován. Prakticky jsou ovšem všechny současné indexové metody napsány v C, stejně jako samotné jádro.

Po naprogramování zmíněných funkcí jsou tyto zkompilovány do jedné nebo více dynamických knihoven (nejsou-li součástí jádra a tudíž kompilovány spolu s jádrem) a tím je indexová metoda připravena pro zaregistrování.

6.3.2 Zaregistrování funkcí rozhraní

Jsou-li funkce z předchozí části naimplementované ve vybraném jazyce, pak je třeba tyto funkce zaregistrovat. Registrování funkce indexu se nijak neliší od registrace libovolné jiné funkce, například:

```
CREATE OR REPLACE FUNCTION atomrtgettuple (INTERNAL ,INT4)
RETURNS BOOL
AS 'libatomrtree.dll', 'atomrtgettuple'
LANGUAGE 'C';
```

Tedy je třeba pro každou funkci říci, jakou má hlavičku, kde se nachází dynamická knihovna s funkcí, jak se funkce jmenuje uvnitř knihovny a v jakém jazyce je knihovna napsána.

6.3.3 Zaregistrování indexové metody

Zaregistrování metody spočívá pouze v zápisu informací o metodě do systémové tabulky *pg_am*. V této tabulce jsou uloženy informace o vlastnostech indexu a jména funkcí z předchozího kroku.

6.3.4 Vytvoření třídy operátorů

Aby se mohly indexové metody v rámci své implementace odkazovat pomocí čísel na jednotlivé operátory při posuzování, zda je pro daný záznam splněna podmínka pro vyhledávání (a nemusely používat nějaké řetězcové konstanty), jsou pro každou indexovou metodu definovány třídy operátorů. Třídu operátorů využívá i jádro při posuzování, zda WHERE podmínka položeného dotazu může být vyhodnocena danou indexovou metodou. Příklad vytvoření třídy operátorů je:

```
CREATE OPERATOR CLASS atomrtree_int_ops
DEFAULT FOR TYPE int4 USING ATOMRTREE AS
OPERATOR 1 =,
OPERATOR 2 >,
OPERATOR 3 >=,
OPERATOR 4 <,
OPERATOR 5 <=;
```

6.3.5 Použití indexu

Index pak musí být použit pouze nad sloupci typu, pro který existuje třída operátorů. Následuje příklad použití indexu:

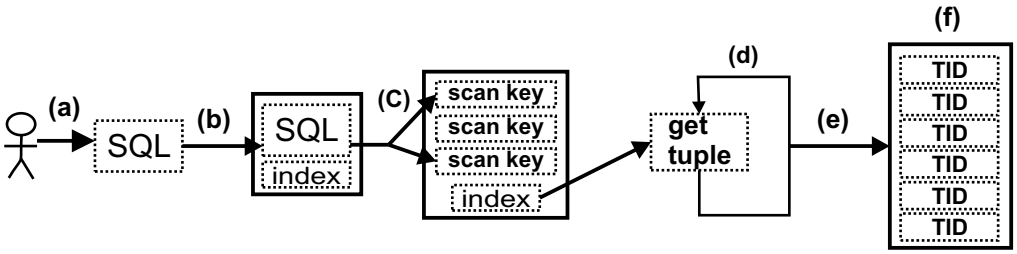
```
CREATE INDEX i ON table_name USING index_name(col1, col2, col3)
```

6.4 Vyhledávání v existujícím indexu

Vyhledávání v indexu PostgreSQL probíhá následovně (schematicky na obrázku 6.2):

1. Uživatel zadá SQL dotaz.
2. PostgreSQL vyhledá všechny indexy, které jsou nad danou tabulkou vytvořeny.
3. Pro každý index je odhadnut čas potřebný k vyhledání odpovídající množiny záznamů (použitím *index_costestimate*).
4. Je parsováná WHERE klauzule dotazu a je získána konjunkce podmínek pro vyhledávání.
5. Započne vyhledávání v indexu, který je podle bodu 3 optimální.

6. Opakovaně je volána metoda zvolené indexové metody, která vrací záznam nebo množinu záznamů odpovídající vyhledávacím podmínkám.



(a) Zadání SQL dotazu
(b) Vyhledání předpokládaného optimálního indexu optimalizátorem
(c) Parsování WHERE klauzule SQL dotazu pro získání množiny vyhledávacích klíčů
(d) Opakované volání indexové metody pro získání záznamu
(e) Výsledek - množina ukazatelů na záznamy odpovídající dotazu

Obrázek 6.2: Vyhledávání v indexu PostgreSQL

6.4.1 Vyhledávací klíče

V bodu 4 kapitoly 6.4 bylo zmíněno, že "je získána konjunkce podmínek pro vyhledávání". Tato vágní formulace vyjadřuje fakt, že implementátor indexové metody nemusí řešit parsování WHERE klauzule dotazu, ale tuto práci za něj provede jádro PostgreSQL. Indexové metodě je pak předána pouze struktura, která obsahuje množinu trojic *atribut-operátor-hodnota*. Tyto trojice se nazývají *vyhledávací klíče (scan keys)* a tvoří konjunkci. Pokud *WHERE klauzule* obsahuje operátor OR, pak je situace řešena vícenásobným vyhledáváním a sjednocením výsledků jednotlivých vyhledvání operací průniku.

Vyhledávací klíč je definován strukturou *ScanKeyData*:

```
typedef struct ScanKeyData
{
   int sk_flags;
   AttrNumber       sk_attno;
   StrategyNumber   sk_strategy;
   Oid              sk_subtype;
   FmgrInfo         sk_func;
   Datum            sk_argument;
} ScanKeyData;
```

Význam položek je následující:

- *sk_flags* - bitová disjunkce vyjadřující, zda
  - sk_argument je *null*
  - jde o unární operátor
  - jde o negaci
- *sk_attno* - číslo atributu (v rámci relace), kterého se klíč týká
- *sk_strategy* - číslo strategie operátoru, tedy o jaký operátor se jedná $(<, =, <=, \ldots)$
- *sk_subtype* - subtyp strategie
- *sk_func* - funkce pro vyhodnocení kvalifikátoru
- *sk_argument* - hodnota k porovnání

Indexová metoda při vyhledávání projde záznamy, které obsahuje a které připadají do úvahy (v stromu projde pouze relevantní větve), porovná uložené hodnoty v indexových uzlech s vyhledávacími klíči a případně označí daný indexový záznam jako záznam obsahující ukazatel na záznam na haldě splňující podmínky vyhledávání (jádro jej pak již vyzvedne samo).

6.4.2 Procházení indexu

Samotný průchod indexem je v PostgreSQL nazýván *scan*. Jde pouze o strukturu, která udržuje informace důležité pro vyhledávání v daném indexu vzhledem k zadaným vyhledávacím klíčům. Na začátku vyhledávání je otevřen *scan* reprezentovaný strukturou *scandesc* a tato struktura existuje po celou dobu průběhu vyhledávání. Pro popis vyhledávání je třeba nejdříve ukázat strukturu scanu:

```
typedef struct IndexScanDescData
{
      Relation          heapRelation;
      Relation          indexRelation;
      Snapshot          xs_snapshot;
      int               numberOfKeys;
      ScanKey           keyData;
      bool              is_multiscan;

      bool              kill_prior_tuple;
      bool              ignore_killed_tuples;

      bool              keys_are_unique;
```

```
    bool            got_tuple;
    void            *opaque;
    ItemPointerData currentItemData;
    ItemPointerData currentMarkData;

    HeapTupleData   xs_ctup;
    Buffer          xs_cbuf;

    int             unique_tuple_pos;
    int             unique_tuple_mark;

    PgStat_Info     xs_pgstat_info;
} IndexScanDescData;
```

Význam relevantních položek je následující:

- *heapRelation* - deskriptor HR, pro kterou vyhledávání probíhá, kde se nachází vyhledávané záznamy
- *indexRelation* - deskriptor IR, nad kterou vyhledávání probíhá
- *numberOfKeys* - počet vyhledávacích klíčů
- *keyData* - pole vyhledávacích klíčů
- *is_multiscan* - zda index může vracet najednou více hodnot vyhovující vyhledávacím klíčům
- *got_tuple* - je nastaveno na true, pokud byl nalezen alespoň 1 záznam vyhovující vyhledávacím klíčům
- *opaque* - specifické informace indexové metody
- *currentItemData* - odkaz na naposledy nalezený indexový záznam
- *xs_ctup* - odkaz do HR na nalezený prvek

Průchod indexem je redukován na opakované volání funkce *index_gettuple*, která je součástí jádra PostgreSQL. Tato volá zaregistrovanou funkci indexové metody, která při každém zavolání vrací jeden záznam, případně její obdobu, která vrací množinu záznamů. Tento cyklus končí tehdy, když již indexová metoda nevrátí žádný záznam, případně když vrátí menší počet záznamů, než je maximum v případě vracení více prvku v jednom průchodu.

Jediná struktura, která existuje během celého *scanu* je, *IndexScanDescData*, resp. ukazatel na ni, jak již bylo zmíněno. Tím pádem informace potřebné v průběhu celého scanu

uchovávají indexové metody právě tam, konkrétně v její *opaque* struktuře. Například stromové metody si tam ukládají informaci o tom, kde se právě ve stromu nacházejí a tím pádem i informaci o tom, kde při dalším volání mají pokračovat.

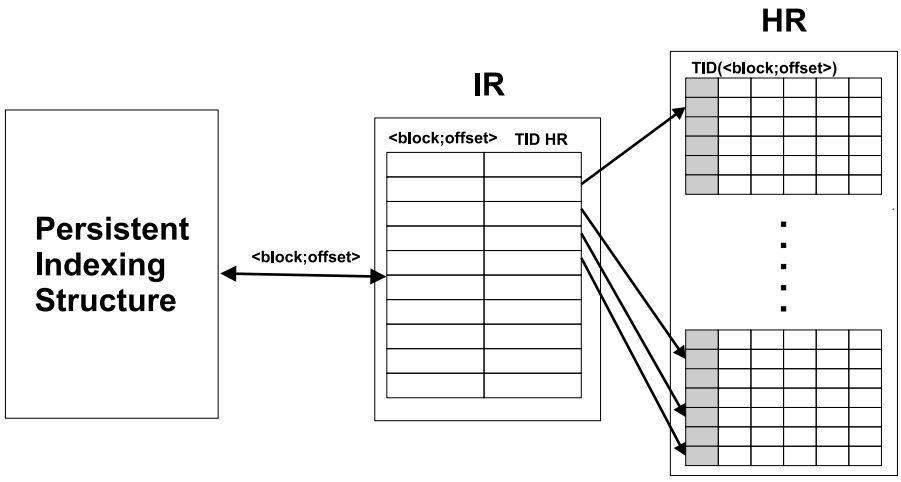
Při každém volání funkce pro získání záznamu je kontrolováno, zda nejde o první volání této funkce. Kontrola spočívá v porovnání hodnoty *currentItemData* na *null*. Pokud jde o první volání, pak metoda inicializuje potřebné údaje v *opaque* struktuře. Následuje samotné vyhledávání. Pokud je úspěšné, pak je do *currentItemData* vložena hodnota posledního indexového záznamu, který odpovídá vyhledávacím klíčům a do *xs_ctup* přímo odkaz na nalezený záznam. O samotné vyzvednutí hodnot nalezeného záznamu se už stará jádro PostgreSQL.

6.5 Indexování v PostgreSQL externím frameworkem

V předchozích částech byl popsán mechanismus klasického indexování v PostgreSQL. Jádro spočívalo v používání relací, jako úložného prostoru pro indexovou metodu a nad těmito relacemi se prováděly operace příslušné dané metodě. Chceme-li použít nějakou již implementovanou metodu, pak se veškerá práce přenese z tohoto místo do frameworku. Framework sám pak řeší uložení záznamů a práci nad nimi. Je pak "pouze" třeba zprostředkovat můstek mezi externím frameworkem a databázovou platformou. V kapitole 4 byly zmíněny metody, které musí každá indexová metoda podporovat. Tyto metody musí být přirozeně podporovány stále, ale nyní budou na straně databáze pouze můstky, které budou delegovat práci do externího frameworku, který je bude implementovat.

V optimálním případě by systém fungoval tak, že by byly indexovány TID stránek indexované relace (HR) a IR by byly zcela vynechány. Tím pádem by nevznikala prakticky žádná režie navíc oproti standardnímu indexování v PostgreSQL (kromě volání funkcí frameworku, což je zanedbatelné). Bohužel se ovšem zdá, že s takovým řešením architekti PostgreSQL nepočítali a nelze ho implementovat. Překážkou je totiž způsob, jakým PostgreSQL pracuje s množinou nalezených záznamů. Jádru se, jak již bylo zmíněno, nevrací identifikátor (nebo množina) HR, nýbrž identifikátor IR. Z dané adresy v indexové relaci si poté jádro samo vyzvedne adresu záznamu HR a následně vyzvedne i odkazovaný záznam. Tento systém tudíž vyžaduje i u externího indexování nutnost udržování IR. Externí framework pak neindexuje TID HR, nýbrž TID IR, tj. indexuje adresy indexových záznamů, kde jsou uloženy adresy reálných záznamů. Při dotazu na vyhledání záznamu podle zadaných vyhledávacích klíčů je dotaz postoupen externímu frameworku, který vrátí množinu TID indexových záznamů, kde se nacházejí odkazy do HR a tyto TID jsou předány jádru PostgreSQL (viz. obrázek 6.3).

Při použití právě zmíněného principu pak není třeba za jistých okolností vůbec implementovat metody, které jsou požadovány databází (index_insert, index_beginscan, atd.) a to tehdy, když již existuje nějaká šablona, která zajišťuje podrobnosti komunikace s databází (označení záznamu IR, ukládání a mazání z IR, atd.). Vlastně jde o framework na druhé straně, tj. na straně databáze, který implementátora indexové metody odstiňuje nejen od jádra databáze, nýbrž minimalizuje ostatní komunikaci s databází.



Obrázek 6.3: Převod TID IR do externí indexové struktury

Takový framework vznikl jako důsledek této diplomové práce. Má-li někdo implementováno vlastní persistentní indexování, pak jej může relativně lehce vyzkoušet na reálné DB platformě, v tomto případě PostgreSQL.

6.6 Framework pro externí indexování v PostgreSQL

Framework je sada funkcí, které je třeba realizovat pro implementaci propojovacího můstku mezi externí indexovou metodou a PostgreSQL. Jak již bylo uvedeno, jde vlastně o zobecnění metody pro implementaci uživatelského indexu PostgreSQL na úroveň, která odstiňuje implementátora od databáze ještě více, než to zajišťuje PostgreSQL.

Pro implementování vezme programátor již existující naprogramovanou přístupovou metodu využívající externí framework (například tu, která vznikla v rámci této diplomové práce) a přepíše funkce, které komunikují s externím frameworkem. Tyto metody zkompiluje a výslednou knihovnu použije stejně jako je popsáno v sekci 6.3.2. Následuje seznam potřebných funkcí a jejich sémantika:

- void *FW_CreateStructure*(Relation index_relation);
  - V této funkci lze říci externímu frameworku na základě *index_relation*, jakou strukturu mají indexované záznamy (počet atributů a jejich typ).
- void **FW_PrepareInsert*(Relation index_relation);
  - Tato funkce je volána před započetím vkládání množiny prvků a jejím smyslem je možné připravení na vkládání (otevření souboru s indexovou strukturou, apod.). Potřebuje-li daná indexová metoda udržovat globální proměnné po celou dobu vkládání množiny prvku (odkaz na soubor atp.), pak to může provést vrácením

těchto proměnných (ve formě odkazu na strukturu) jako návratové hodnoty funkce. Struktura je pak přístupná jako parametr následujících funkcí.

- void *FW_InsertTuple*(void *fw_data, Relation index_relation, IndexTuple index_tuple, BlockNumber block_number, OffsetNumber offset);
  - Zajišťuje vložení prvku do externího indexu. Odkaz na strukturu s informacemi vytvořenými v *FW_PrepareInsert* je ve vstupním parametru *fw_data*.
- void *FW_FinishInsert*(void *fw_data);
  - Volána po ukončení vkládání (v rámci ní je možné provést potřebný úklid, či uvolnění paměti). *fw_data* znovu reprezentuje odkaz na globální data (právě ta by měla být předmětem úklidu).
- void *FW_InitSearch*(IndexScanDesc scan, ScanDirection dir);
  - Volána před započetím vyhledávání. Je-li externí metoda stavěna tak, že na počátku provede vyhledání všech prvků a ty pak po jednom vrací, pak právě v této funkci bude provedeno vyhledávání. Navrácené prvky jsou pak uloženy do opaque struktury proměnné *scan* (viz. kapitola 6.4.2).
- bool *FW_GetNextTID*(IndexScanDesc scan, ScanDirection dir, BlockNumber *block_number, OffsetNumber *offset);
  - V této funkci je buď v externím úložišti vyhledán další prvek, nebo je z *opaque* struktury proměnné *scan* vyzvednut další záznam z množiny již dříve vyzvednutých záznamů. V druhém případě je třeba si také nějakým způsobem pamatovat již navrácené záznamy (obvykle indexem do pole navrácených záznamů).
- void *FW_DeleteTuple*(BlockNumber block_number, OffsetNumber offset);
  - Provede vymazání záznamu z externího úložiště.

Implementace pak bude pravděpodobně obsahovat ještě další metody, které již ovšem budou použity interně (například parsování vyhledávacích klíčů do podoby vhodné pro danou externí metodu, ...) Výše zmíněný seznam je nutným minimem.

Je-li vlastní persistentní indexová metoda psána v C, pak je možné ji kompilovat spolu s kompilací funkcí pro PostgreSQL. V opačném případě je třeba ji zkompilovat do knihovny a pravděpodobně navíc "obalit" wrapperem, jak je ukázáno v kapitole 6.7.2 na příkladu zabalení *ATOMu*.

6.6.1 Výhody použití

Výhody použití externího persistentního indexování s právě zmíněným frameworkem na straně PostgreSQL jsou zejména následující:

- *Minimalizace nutných znalostí PostgreSQL* - stačí vědět, že je třeba indexovat dvojici $< blok; offset >$ a struktury, která vstupují jako parametry do funkcí.
- *Rychlost implementace* - existuje-li již persistentní indexová metoda, pak napsání propojovacího můstku pravděpodobně zabere (ve srovnání s vývojem metody) zanedbatelný čas (nejvíce stráveného času při implementaci připadne na pochopení fungování a propojení funkcí typu *index_xxx* a způsob práce s pamětí v PostgreSQL, což zde zcela odpadá).
- *Implementace indexové metody je zcela nezávislá na databázové platformě* a lze ji případně beze změny použít pro srovnání v dalších databázových platformách, které umožňují implementovat vlastní uživatelské indexy jako PostgreSQL (například MySQL od své pětkové verze - viz. [18], Oracle, DB2).

6.6.2 Problémy (a jejich řešení)

Použití externího indexování v PostgreSQL s sebou přináší ovšem i nevýhody. Jsou to především:

- *Zdvojené načítání stránek* (v nejhorším případě) - index je uložen ne pouze na úrovni indexové metody, nýbrž i v indexových relacích PostgreSQL (tam jsou odkazy do HR) a tudíž čtení diskových stránek může být víc, než když prohledávání samotné je realizováno nad IR. Je třeba ovšem podotknout, že tato nevýhoda je zmírněna faktem, že záznamy v IR externích indexů jsou výrazně menší, než záznamy klasických interních indexů, protože externí indexy udržují v IR pouze externí záznamy. To způsobuje, že se jich vejde do jednoho bloku více,respektive se jich více vejde do databázové cache a nejsou tak často načítány z disku.
- *Globální data* - může nastat potřeba udržovat globální data po delší dobu, než je doba *scanu* (například pro udržení cache). Takovou službu ovšem jádro PostgreSQL indexovým metodám neposkytuje a tudíž je třeba hledat náhradní metody (například nepříliš čistý způsob, jakým je využívání souborů k ukládání adres odkazů, nebo využívání registrů ve Windows - což ale není multiplatformní).
- *Současný přístup více uživatelů* - persistentní indexování ze své podstaty vyžaduje uložení dat do souborů. Zde pak nastává problém, při updatu indexu více uživateli najednou. V současné verzi je problém řešen tak, že při vkládání záznamů je soubor s indexovými záznamy zamknut a při každém vkládání je testován a případný konflikt předáván PostgreSQL jako chyba.

- *Nemožnost rozpoznání ukončení existence indexu v PostgreSQL* - tento problém vznikl nedůslednou implementací uživatelského indexování v PostgreSQL. Pro vznik indexu nad relací a jejím updatu je vždy volána nějaká funkce rozhraní uživatelsky definované indexové metody. Toto ovšem již neplatí pro odebrání indexu! Důvodem je zřejmě skutečnost, že vývojáře PostgreSQL nenapadla možnost nutnosti úklidu po odebrání indexu. Počítali s tím, že každá uživatelsky definovaná metoda bude používat jako svoje úložiště indexové relace PostgreSQL. Tyto jsou po zrušení indexu nad relací odstraněny automaticky samotným jádrem PostgreSQL. Ovšem pravděpodobně v průběhu času se objevila nutnost úklidu i po standardních metodách a proto v jádru existují při ukončení existence indexu volání uklízecích metod pro standardní indexové metody (B-strom, R-strom, GiST). Nicméně tyto metody jsou napevno zabudované v jádru. Pro externí persistentní metodu je bohužel tento bod důležitý z důvodu možnosti smazání persistentního úložiště při konci existence indexu. Zde neexistuje zcela korektní řešení. Pravděpodobně jedinou možností je recyklace souborů, při vzniku indexu (neaktivní indexový soubor se pozná například tím, že má stejné jméno jako indexový soubor právě vznikajícího indexu).

6.7 Indexování v PostgreSQL ATOMem

Konkrétní implementací externího persistentního indexování je implementace frameworku *ATOM* do PostgreSQL. Tato implementace může být vzorem, z nějž by mohli vycházet další implementace principem naznačeným v předchozí kapitole. Všechny výhody a nevýhody spojené s používáním externího persistentního indexování byly zjištěny právě při implementování tohoto frameworku, stejně jako vznikli jejich proprietární řešení.

Zde je třeba upozornit, že tato práce byla zaměřena především na porovnání výkonu R-stromu s ostatními metodami a výsledná implementace umí indexovat pouze celá čísla, což je pro tento účel zcela dostačující.

6.7.1 ATOM

ATOM (Amphora Tree Object Model) je objektový framework pro implementaci persistentních stromových datových struktur (viz. [9]). Je součástí hypertexotého systému *Amphora* vyvíjeného skupinou *Amphora Research Group* na Katedře informatiky FEI, VŠB. *ATOM* může v současnosti sloužit jako základ pro implementaci "libovolné" stromové persistentní datové struktury. Framework je implementován v jazyku C++. Výrazným rysem použitým při implementace *ATOMu* je použití šablon. Například třída stromu je parametrizována typem svých uzlů.

Základní myšlenkou *ATOMu* je skutečnost, že při velkém množství operací nad velkými daty je nejpomalejší operací alokování paměti (nepočítáme-li přístup na disk, kterému se ovšem vyvarovat nedá). Proto *ATOM* na začátku alokuje paměť o určité velikosti (cache) a tuto používá v průběhu celé práce se strukturou a neprovádí žádnou další alokaci místa pro uzly stromu.

Základní třídy s kterými je v *ATOMu* pracováno jsou:

- cPersistentTree
  - Hlavní třída reprezentující strom.
  - Zapouzdřuje všechny ostatní třídy - konkrétně:
    - cTreeHeader *mHeader
      - hlavička stromu
    - cTreeCache<TNode,TLeafNode> mCache
      - rozhraní k sekundární paměti
    - cTreePool<TItem,TLeafItem,TNode,TLeafNode> *mTreePool
      - ukazatel na objekt obsahující předalokované objekty (uzly, položky uzlů)
    - cPool *mPool
      - uživatelský mTreePool
    - cQueryResult<TLeafItem> mQueryResul
      - objekt obsahující výsledky dotazu
    - cQueryStatistic *mQueryStatistics
      - objekt obsahující statistiky operací
- cObjTreeItem
  - Rozhraní, které musí implementovat položka stromu.
  - Obsahuje metody jako *smazání obsahu*, *změna velikosti*, *zápis položky do streamu*,...
- cObjTreeNode<TItem>
  - Třída parametrizovaná položkou, která může sloužit jako kořen třídy listových i nelistových uzlů (ATOM rozlišuje instance listových a nelistových uzlů, protože některé stromové struktury to vyžadují - např. BUB-strom).
  - Obsahuje metody specifické pro strukturu (štěpení, slučování, ...).
- cTreeHeader
  - Obsahuje parametry stromu
    - Kapacita vnitřních a listových uzlů
    - Maximální počet potomků
  - Obsahuje informace o stromu
    - Výška
    - Počet uzlů
    - Identifikace

  - Obsahuje odkazy na objekty
    * cQueryStatistic
    * cTreePool
    * cPool

- cTreeCache
  - Uzly jsou v persistentním stromu mapovány na diskové stránky a toto mapování má na starosti cache.
  - Uložení uzlů lze provést pouze skrz tento objekt pomocí metod
    * void Write(const TNode &node);
    * void WriteLeaf(const TNode &node);
    * void Read(const tNodeIndex index, TNode &node);
    * void ReadLeaf(const tNodeIndex index, TLeafNode &node);
  - ATOM používá pro cache zřetězené hashování, tzn. uzly jsou uloženy v dvourozměrném poli a v případě kolize, tj. dva uzly se hashují na stejný index, se záznamy vkládají do "kapsy" na daném indexu

- cCacheStatistics
  - Třída, pro měření počtu přístupů k disku a počtu přístupů ke cachi pro listové a nelistové uzly.

- cPool, cTreePool
  - Zde jsou uloženy potřebné informace k jednotlivým uzlům. Jde o předalokovaná pole, aby nemuselo docházet k dynamické alokaci.

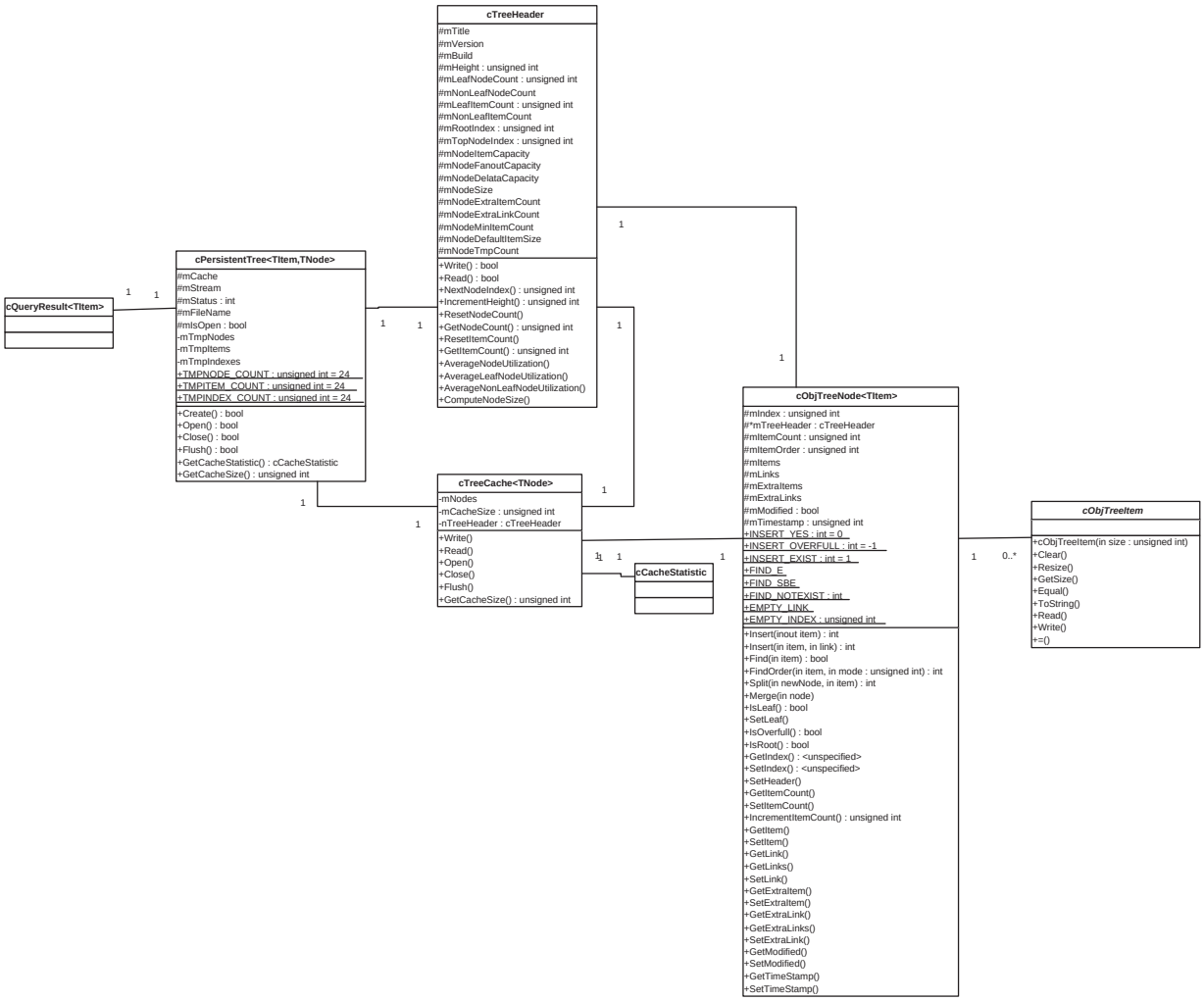
Každá konkrétní indexová metoda implementovaná za pomoci *ATOMu* pak využívá těchto tříd ať už tak, že je parametrizuje, nebo od nich dědí a tím implementuje vlastní funkcionalitu (R-strom, UB-strom, Pyramid-strom v [14], ...).

Hlavním nedostatkem současné verze *ATOMu* je neexistence operace *delete* na stromu! Tím pádem žádná ze struktur, které v současné době nad *ATOMem* existují neumožňuje odebrání prvků ze stromu (a tedy to neumožňuje ani R-strom, který byl v této práci implementován do PostgreSQL).

6.7.2 Použití ATOMu v PostgreSQL

Konkrétní postup zabudování *ATOMu* do PostgreSQL je následující:

1. Vzít stávající implementaci nějaké persistentní stromové struktury vybudované s podporou *ATOMu* (v této práci byl použit R-strom).



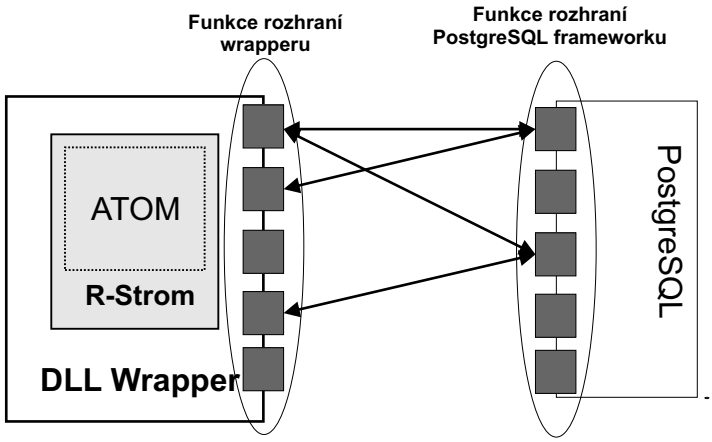
Obrázek 6.4: CASE diagram *ATOMu*

2. Vzít v úvahu všechny metody, které budou pro komunikaci s databází zapotřebí a zabalit je do volání funkcí s C rozhraním (*wrapper* funkce).

3. Zkompilovat zdrojové kódy *ATOMu* spolu s *wrapper* funkcemi (případně lze přirozeně nejdříve do knihovny zkompilovat samotnou indexovou metodu a při kompilaci *wrapper* funkcí tuto knihovnu přilinkovat).

4. Použít knihovnu z bodu 3 při kompilaci PostgreSQL indexové metody.

Zabalení persistentní struktury do DLL *wrapperu* (bod 2) a její komunikace s PostgreSQL je naznačeno ve schématu 6.5.



Obrázek 6.5: Propojení ATOMu a PostgreSQL skrz rozhraní

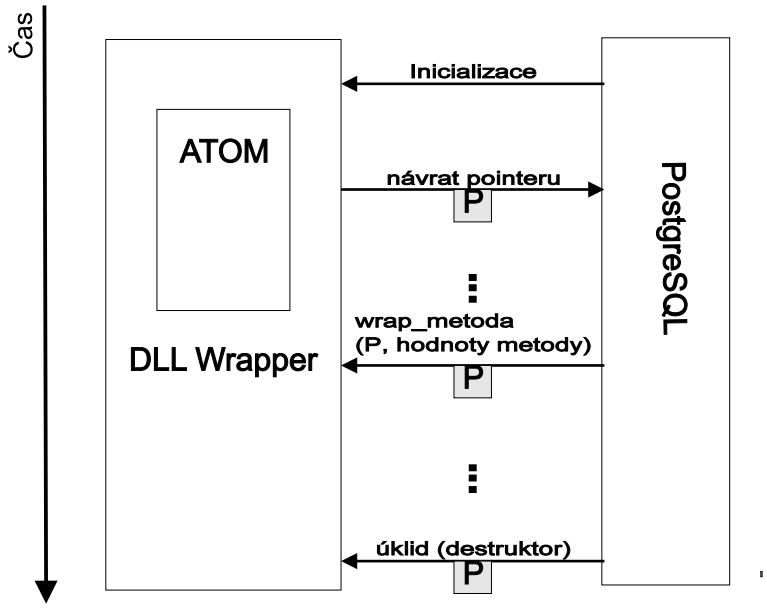
Wrapper ATOMu

Již bylo zmíněno, že PostgreSQL je vyvíjeno v jazyce C a že je-li externí indexová metoda napsána v jiném jazyce, je třeba ji zabalit do funkcí jazyka C, tj. do *wrapperu*. *DLL Wrapper* je tedy dynamická knihovna, která slouží pro komunikaci C kódu s C++ kódem (ale potenciálně s kódem v libovolném jazyku) persistentní indexové metody. V *ATOMu* je celé indexování soustředěno do jedné třídy reprezentující persistentní strom. Při inicializaci je volán konstruktor této třídy, který vytvoří ukazatel na třídu, s kterým je dále pracováno. Každá metoda třídy, kterou je potřeba využívat v PostgreSQL, je zabalena do C funkce, která má stejné parametry, jako zapouzdřovaná metoda a navíc obsahuje ukazatel na objekt třídy, která danou metodu obsahuje (ve většině případů je to strom). Uvnitř této funkce je ukazatel přetypován na objekt dané třídy a je zavolána příslušná metoda s parametry předanými ve vstupních parametrech funkce. Tedy definice takové zapouzdřovaní funkce je schematicky:

```
ret_val wrapper_function (void *tree, param1_type param1, ...,
                          paramn_type paramn)
{
  return ((TreeType)tree)->function(param1, ..., paramn);
}
```

Ukládání adresy IR

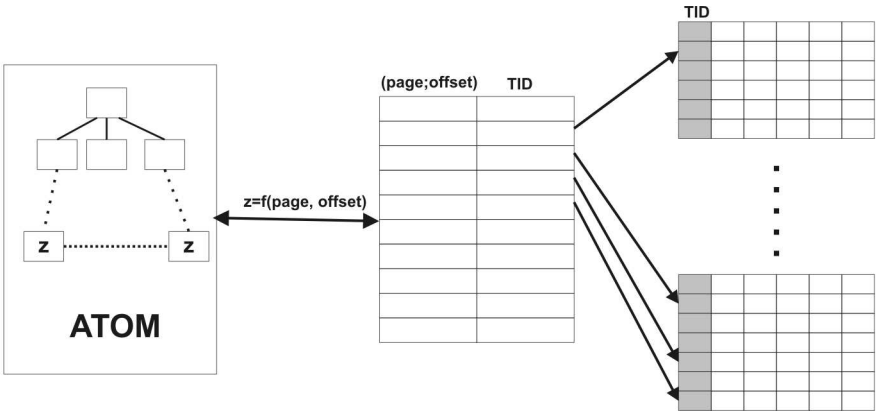
ATOM umí jako hodnoty indexovat pouze kladná celá čísla (*unisgned integer*). Zde nastává problém, protože PostgreSQL musí ukládat dvojici $< blok; offset >$. Proto při ukládání adresy IR záznamu je třeba provádět konverzi adresy na kladné celé číslo a při načítání záznamu z *ATOMu* provést inverzní operaci, tj. z kladného celého čísla získat *blok* a *offset*. Toto je zajištěno jednoduchou konverzí typu:



Obrázek 6.6: Volání funkcí wrapperu v čase

$$value = block * multiplicator + offset$$

Při načtení z ATOMu se provede inverzní operace (viz. obrázek 6.7). Vzhledem k faktu, že počet bloků v relaci není ani pro desítky milionů záznamů veliký (zvláště pak vezmeme-li v úvahu, že záznam v IR je u externího indexování malý), lze multiplikátor volit relativně malý a tedy nenastávají problémy s tím, že by se hodnota ukládaná do *ATOMu* nevešla do velikosti *unsigned integeru* a nastalo přetečení hodnoty. Obrázek 6.7 je modifikací obrázku 6.3 pro *ATOM* (z je hodnota spočítaná z dvojice page a offset, protože *ATOM* umožňuje ukládat pouze unsigned integer).



Obrázek 6.7: Převod TID IR do ATOMu

Kapitola 7

Experimenty

V této kapitole se podíváme na výsledky experimentů pro srovnání efektivity implementovaného R-stromu v PostgreSQL s několika dalšími komerčními databázovými systémy a jejich indexovými metodami podporujícími víceatributové indexování. Testy porovnávají výkon jednotlivých metod především v závislosti na selektivitě, dimenzi dat a velikosti databáze.

7.1 Testovací data

Data použita v testování byla rozdělena do tří typů:

1. Syntetická
   - Uniformní rozložení
   - Gaussovské (normální) rozložení
2. Reálná

Data syntetická byla generována generátorem náhodných čísel a to buď zcela náhodně v případě uniformního rozložení, nebo tak, aby rozložení pravděpodobnosti výskytu bodů v daném prostoru bylo gaussovské (viz. kapitola 7.1.2).

Pro uniformní i gaussovské rozložení byla generována data o velikostech 10^4, $5 * 10^4$, 10^5, $2 * 10^5$, $5 * 10^5$, 10^6, $2 * 10^6$, $5 * 10^6$ bodů. Pro každou tuto sadu byly vygenerovány body v dimenzích, 1, 2, 3, 4, 5, 10, 15, aby bylo možné komplexně prostudovat vliv, který má na výsledek velikost testovaných dat a jejich dimenze.

Za účelem snahy o maximalizaci rovnosti podmínek pro vyhledávání ve všech databázových platformách byly všechny dotazy vytvořeny před samotným testováním. Tedy dotazy do každé platformy byly naprosto stejné a nad stejnými sadami dat.

Navíc byl každý dotaz upraven do tvaru:

```
SELECT COUNT(*) FROM tabulka ...
```

K agregaci funkcí *COUNT* bylo přistoupeno z toho důvodu, aby do času vyhledání prvků nebyl započítán i čas pro jejich vyzvednutí (například PostgreSQL vyzvedává záznamy i tehdy, když klíč záznamu obsahuje všechny selektované atributy, čímž několikanásobně zvyšuje celkový čas dotazu) a také proto, že některé platformy nemusí při tomto způsobu vyhledat všechny prvky, ale pouze tolik, o kolik si aplikace řekne a výsledky testů by pak nebyly vypovídající. Přirozeně v reálných aplikacích je důležité, zda jsou najednou vyhledány všechny prvky, nebo zda je vyhledána pouze část a další jsou vyhledávány postupně, nebo zda jádro databáze vyzvedává záznamy i když má všechny potřebné atributy v klíči. Nicméně takové chování není předmětem této práce a proto bylo z testů eliminováno. Lze navíc předpokládat, že agregace pomocí *COUNT* není drahá operace a proto nijak výrazně neovlivní výsledky testů (i kdyby ano, tak u všech testovaných platforem přibližně stejně).

7.1.1 Uniformní rozložení

Uniformní data byla generována tak, aby tvořila shluky a podobala se tedy alespoň částečně reálným datům, kde údaje také nejsou obvykle rovnoměrně rozloženy v celém prostoru. Velikost shluku v dané sadě byla závislá na doméně dimenze, počtu dimenzí a počtu shluků a řídila se pravidlem 7.1.

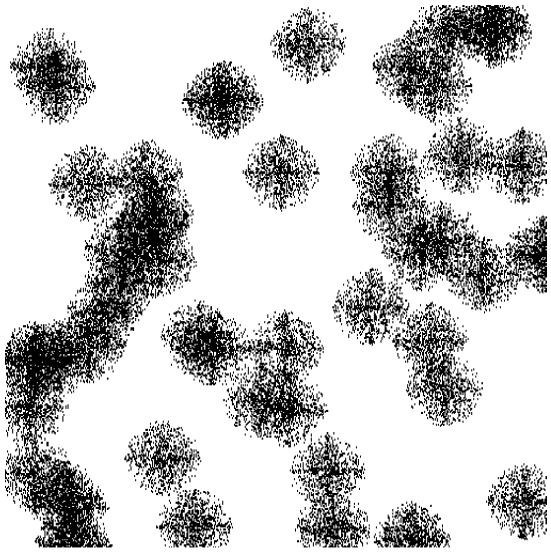
$$a = \sqrt[b]{\frac{\frac{c^b}{e}}{d^2}} \tag{7.1}$$

a ... velikost shluku, b ... počet dimenzí, c ... velikost dimenze, d ... dimenze, e ... počet shluků

Na obrázku 7.1 lze vidět uniformní rozložení 50000 bodů ve 2 dimenzích v 50 shlucích. Počet shluků závisel na velikosti datové sady - viz. tabulka 7.1.

Počet bodů	Počet shluků
10000	10
50000	50
100000	100
200000	200
500000	500
1000000	600
2000000	700
5000000	800

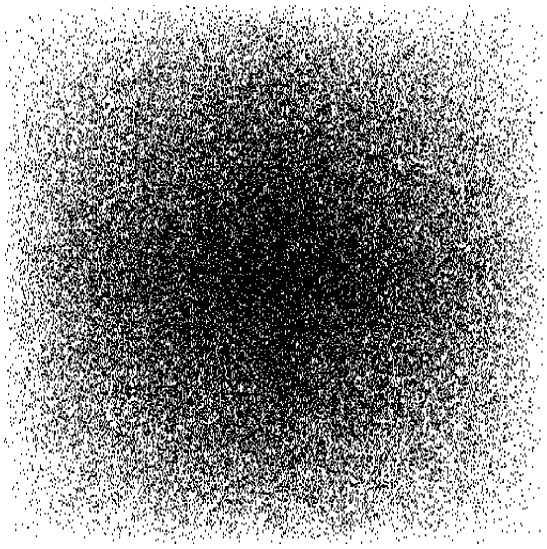
Tabulka 7.1: Vztah počtu shluků na počtu bodů datové sady



Obrázek 7.1: Uniformní rozložení bodů ve 2D

7.1.2 Gaussovské rozložení

Gaussovské rozložení bodů v prostoru je takové, pro něž platí, že pravděpodobnost výskytu bodu je nejvyšší ve středu prostoru. Gaussovské rozložení nebylo generováno do shluků, jako rozložení uniformní, nýbrž využívalo celý prostor. Na obrázku 7.2 lze vidět gaussovské rozložení 100000 bodů ve 2 dimenzích o doméně dimenze 500000.



Obrázek 7.2: Gaussovské rozložení bodů ve 2D

7.1.3 Reálná data

Pro testování reálných dat byla zvolena databáze článku z oblasti informatiky - *DBLP* (Digital Bibliography & Library Project - viz. [20]). Tato databáze je přístupná ve formě XML souboru a pro potřeby testu obsahovala 435373 záznamů. Atributy, které obsahovala a proti kterým byla testována jsou:

- Jméno autora
- Typ publikace
- Rok vydání
- Počet stran

Pro účely testování bylo vygenerováno 28 sad dotazů, kde každá sada obsahovala 100 dotazů, jejichž výsledky byly zprůměrovány. Sady se lišily selektivitou a počtem atributů, podle kterých bylo indexováno a vyhledáváno. Počty atributů (dimenzí) byly 1, 2, 3 a 4 a selektivity nabývaly hodnot 1, 2, 3, 5, 10, 20 a 50 záznamů. Vzhledem k tomu, že u složených B-stromů závisí výsledek na pořadí atributů, uveďme spolu s atributy vyskytujících se v jednotlivých dimenzích i jejich pořadí:

Počet	Pořadí atributů	Příklad dotazu
1	Jméno autora	*SELECT COUNT(*) FROM dblp WHERE AUTHOR LIKE 'eva%'*
2	Typ publikace, Jméno autora	*SELECT COUNT(*) FROM dblp WHERE type=1* *AND AUTHOR LIKE 'eva%'*
3	Rok vydání, Typ publikace, Jméno autora	*SELECT COUNT(*) FROM dblp WHERE year>1998 AND year <= 2001* *AND type BETWEEN(1,2)* *AND AUTHOR LIKE 'eva%'*
4	Počet stran, Rok vydání, Typ publikace, Jméno autora	*SELECT COUNT(*) FROM dblp WHERE pages >= 10 AND pages <= 20* *AND year > 1998 AND year <= 2001* *AND type BETWEEN(1,2)* *AND AUTHOR LIKE 'eva%'*

Tabulka 7.2: Atributy

Vhledem k faktu, že implementace R-stromu v PostgreSQL vzniklá v této práci dovoluje implementovat pouze celá čísla, byla data předpřipravena do formy, kdy řetězce byly setříděny a ke každému bylo přiřazeno číslo podle jeho pořadí. Predikát *LIKE* v dotazu byl pak převeden do formy rozsahového dotazu na číselný sloupec reprezentující jméno (viz. obrázek 7.3).

TITLE	PUBLICATION	AUTHOR	YEAR	TYPE	PAGES	AUTHOR_NUM
Collaborative IV&V by SPEED: a tool-kit for the performance IV&V of critical software.	WETICE	Andrea D"Ambrogio	1995	2	9	79520
Experimental Results on Subgoal Reordering.	IEEE Trans. Computers	David A. Plaisted	1990	1	3	272750
Toward Self-organizing, Self-repairing and Resilient Distributed Systems.	Future Directions in Distributed Computing	Alberto Montresor	2003	2	7	40124
PAS-II: An Interactive Task-Free Version of an Automatic Protocol Analysis System.	IEEE Trans. Computers	Allen Newell	1976	1	11	62564
Triviality, NDOP and Stable Varieties.	Ann. Pure Appl. Logic	Bradd Hart	1993	1	27	186027
Modeling and Simulation of High-Frequency Integrated Circuits Based on Scattering Parameters.	DAC	C. H. Chan	1991	2	5	203051
A 5 dof Haptic Interface for Pre-operative Planning of Surgical Access in Hip Arthroplasty.	WHC	Darwin G. Caldwell	2005	2	1	270321
Flexible Handling in Disassembly with Screwnail Indentation.	ICRA	Bing-Ran Zuo	2000	2	5	175693
Complex document image segmentation using localized histogram analysis with multi-layer matching and clustering.	SMC (4)	Bing-Fei Wu	2004	2	7	175665
Report from the First Workshop on Geo Sensor Networks.	SIGMOD Record	Agnes Voisard	2004	1	3	22207
Hierarchical Cellular Automata structures.	Parallel Computing	Adonios Thanailakis	1992	1	7	19497
Image-Based Re-Rendering of Faces for Continuous Pose and Illumination Directions.	CVPR	Arne Stoschek	2000	2	5	126360
Spracherkennung und Prosodie.	KI	Anton Batliner	1997	1	5	113013
Heuristics for model checking Java programs.	STTT	Alex Groce	2004	1	16	45946

Obrázek 7.3: Záznamy z DBLP

7.2 Srovnávané platformy

Pro srovnání implementovaného R-stromu byly testovány víceatributové indexy v PostgreSQL, Oraclu, Microsoft SQL Serveru a Transbasu. Každá z testovaných platforem měla velikost stránky 8KB, stejně jako byla nastavena velikost stránky v *ATOMu* na 8KB, aby srovnávání počtu přístupů bylo relevantní.

7.2.1 PostgreSQL 8.1.3

Na platformě PostgreSQL byly testovány dva typy indexů - B^+-strom (úprava B-stromu, kde listy jsou obousměrně provázány - popsáno Bayerem v [4]), který je standardní součástí této databázové platformy a dále R-strom, který byl vyvinut v rámci této práce a který běžel nad frameworkem *ATOM*. Vzhledem k popsaným nedůslednostem v architektuře PostgreSQL (nemožnost vynechání indexových relací) byly měřeny zvlášť časy trvání průběhu celého dotazu a časy, které vykazovalo pouze vyhledání záznamu, tj. vyhledávání a navrácení záznamu *ATOMem*. Počet načtených stránek byl měřen pouze pro *ATOM*.

7.2.2 Microsoft SQL Server 2000

MSSQL Server poskytuje pouze jeden typ indexu a to vylepšený B^+-strom. Vvylepšení spočívá v tom, že nejsou provázány pouze uzly na listové úrovni, nýbrž jsou obousměrným seznamem provázány listy na každé úrovni (podrobněji o indexech v MS SQL Serveru 2000 v [15]). Pro účely této práce byl nazván B^{++}- *strom*. Index může být dvojího typu - klastrovaný a neklastrovaný. Klastrovaný index přeorganizovává data tak, aby fyzické uspořádání indexovaných záznamů bylo stejné, jako uspořádání logické. Vzhledem k faktu, že by bylo obtížné srovnávat platformy bez jejich detailních znalostí (možnost pozapínání různých optimalizací, přeorganizování do různých tabulkových prostorů na různých discích, ...), nebyly na žádných z databází použity speciální techniky pro vylepšení výkonu. To znamená, že ani u SQL Serveru 2000 nebyly použity klastrované indexy (jako u Oraclu nebyly použity *index organized tables*, což je podobná věc). Navíc výhoda klastrovaných indexů spočívá v rychlém načtení výsledného záznamu (odpadává načtení další stránky s obsahem záznamu) a jelikož bylo načítání eliminováno operací *COUNT*, tak klastrovaný

index ztrácí mnoho ze své síly.

7.2.3 Oracle 9i Release 2

U Oraclu byla k testování použita varianta B-stromu zvaná B*-strom, která se liší od B-stromu ve skutečnosti, že uzly jsou naplněny minimálně ze 2/3.

7.2.4 Transbase 6.4.1

Transbase je jediná databáze zastoupená v testu, která používá při indexování multidimenzionální metodu pro indexování stejným způsobem, jako to dělá implementace R-stromu v PostgreSQL. Narozdíl od této práce ovšem k indexování používá UB-strom.

7.3 Měřené veličiny

- Čas
  - Měřen byl čas procesu (*process time*), který prováděl výpočet a tedy na dobu běhu neměly vliv ostatní procesy, které v systému v danou dobu běželi. Tím byly zajištěny stejné podmínky pro běh všech platforem.
  - Čas procesu se dělí na čas strávený v jádru systému (*kernel time*) a čas strávený přímo výpočtem (*user time*). Pro účely testu by bylo optimální brát do úvahy pouze *user time*, ovšem vzhledem k faktu, že tyto informace bylo možné získat pouze v PostgreSQL (ATOM tuto informaci zprostředkovává a pro získání času dotazu v Postgresu bylo upraveno jádro PostgreSQL, aby tuto informaci poskytovalo do logu), je v testech uváděn pouze celkový čas procesu.
- Počet načtených stránek
  - Informace o počtu načtených stránek je k výkonu indexové metody ještě relevantnější než čas procesu, protože umožňuje měřit výkon metody bez ohledu na konkrétní konfiguraci stroje, na kterém je měření prováděno.
  - Obvykle jsou uváděny údaje o počtu načtených fyzických a paměťových stránek. Toto dělení se provádí z toho důvodu, že načtení fyzické stránky je pro čas vyhledávání důležitější než čas načtení paměťové stránky, protože přístup na disk je časově náročnější, než přístup do paměti. Ovšem z hlediska výkonu metody je vhodnější prezentovat součet těchto přístupů, jelikož tato hodnota není závislá na konkrétní velikosti cache, potažmo konfiguraci stroje (velikost RAM).
- Velikost indexu
  - Informace obsahující velikost souboru obsahujícího index (případně část souboru reprezentující index) na databázových platformách, z kterých jsem byl schopen tuto informaci získat.

7.4 Způsob testování

7.4.1 Hardware

Všechny databáze byly testovány na stroji osazeném procesorem Pentium4, 3GHz s 1GB RAM a 80GB HD s frekvencí 5400 ot/s.

7.4.2 Metodika

Sady dat s uniformním rozložením byly testovány tak, aby se daly porovnat vztahy selektivity[1], dimenze a velikosti databáze. Tedy byly předem vygenerovány dotazy, s určitými selektivitami. Tyto selektivity měli u každé sady velikost 0.5%, 1%, 1.5%,2%,2.5,3%,6%,9%, ..., 30%. Pro každou selektivitu bylo vygenerováno 100 záznamů a jejich výsledky pak byly zprůměrovány.

Pro data s gaussovským rozložením nebyly vygenerovány dotazy se stejnou selektivitou, nýbrž byla pro každou sadu rozdělena úhlopříčka prostoru na 11 částí a každá z těho částí pak tvořila úhlopříčku n-dimenzionální kostky (n je dimenze prostoru). V každé této kostce pak bylo náhodně vybráno 100 bodů a kolem nich se vytvořily okna[2] o velikosti jedenáctiny úhlopříčky prostoru. Výsledky dotazů odpovídající bodům v každé kostce byly následně zprůměrovány.

Pro testování byla použita pomocná aplikace, která se na testované platformy připojovala buď přes ODBC rozhraní, nebo spouštěla dávkové soubory s dotazy. Následně také získávala statistiky - buď online(např. v PostgreSQL lze získat počet stránek ze systémového katalogu), nebo z výstupů konzole (MSSQL server).

Výsledné údaje pak byly převedeny do vstupu programu *R-project*, který byl použit pro generování grafů (http://www.r-project.org).

7.5 Výsledky

V následujících kapitolách budou prováděny testy na klastrovaných datech s uniformním rozložením, kde budou testovány vlivy selektivity, dimenze a velikosti databáze na počtu načtených stránek. následně bude prezentováno několik testů na gaussovském rozložení. Poté budou porovnány počty přístupů s dosaženými časy a nakonec provedeny testy na reálných datech.

[1] Selektivita uváděna v procentech je podíl vybraných záznamů na počtu všech záznamů v databázi

[2] Oknem rozumíme rozsahový dotaz, který omezí shora i zdola hodnoty v každé z dimenzí

7.5.1 Velikost indexu

Na platformách, kde jsem dokázal získat informace o velikosti indexu, jsem danou informaci extrahoval a výsledek lze vidět v tabulce 7.3. Vzhledem k tomu, že velikost indexu roste přibližně lineárně s velikostí dimenze i s velikostí databáze, tak jsou v tabulce zobrazeny pouze velikosti indexů pro dva atributy a pro všechny testované velikosti databází.

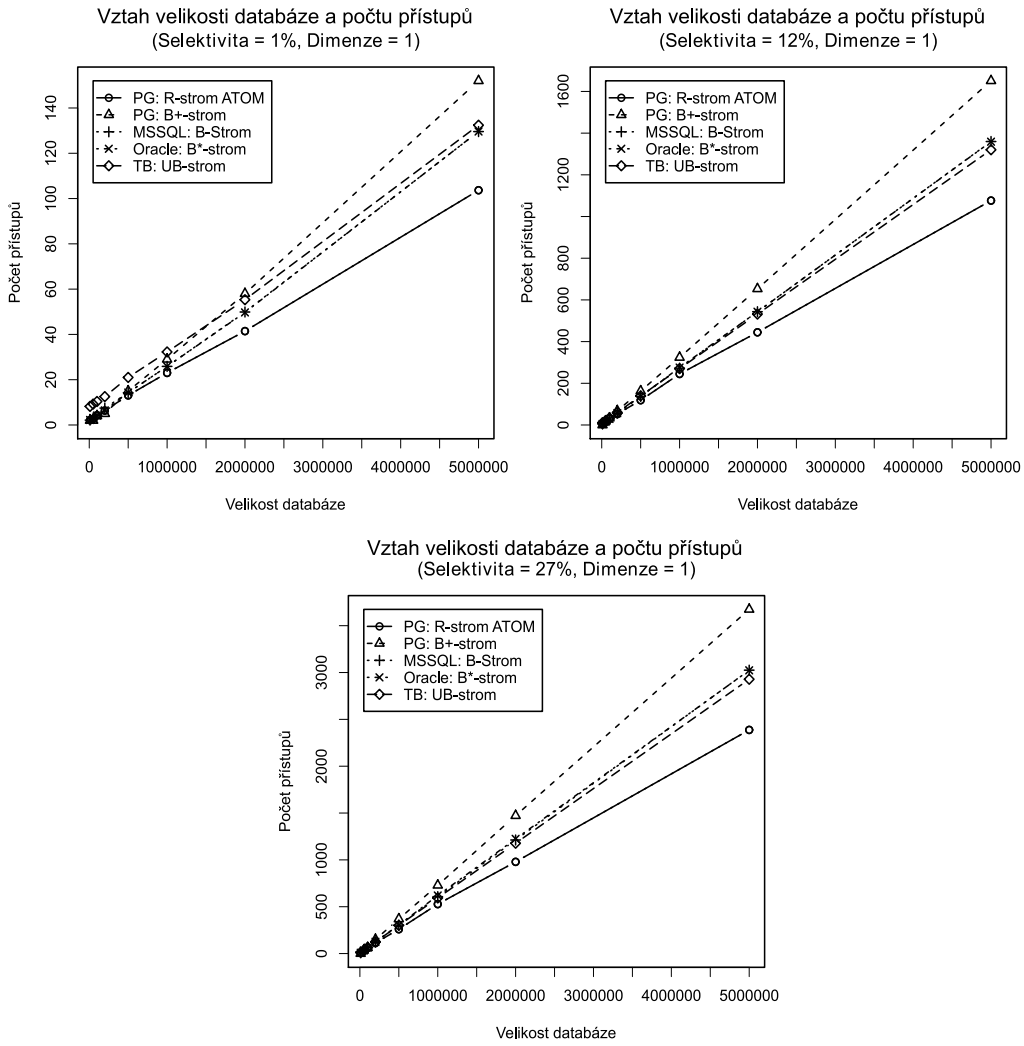
Velikost databáze	ATOM(R)	PG+ATOM(R)	PG(B^+)	MSSQL(B^{++})
10^4	207	412	246	248
$5 * 10^4$	1 053	2 061	1 139	1 112
10^5	2 113	4 129	2 261	2 280
$2 * 10^5$	4 085	8 116	4 514	4 440
$5 * 10^5$	10 261	20 329	11 256	11 024
10^6	20 695	40 831	22 487	21 936
$2 * 10^6$	41 595	81 859	44 950	43 752
$5 * 10^6$	111 970	212 617	112 334	109 200

Tabulka 7.3: Velikost indexů pro dimenzi 2 v KB

Z tabulky lze vyčíst, že režie R-stromu je prakticky stejná, jako režie B-stromu. Dvounásobná velikost indexu při jeho zabudování do PostgreSQL jednoznačně plyne ze zmiňovaného problému nutnosti uložení indexu navíc do indexových relací PostgreSQL (ačkoli je to zcela zřejmě redundantní krok). Z faktu, že ve všech měřených platformách je velikost indexu prakticky stejná, plyne, že velikost indexu je v zásadě dána množstvím dat, které udržuje a režie navíc je všude stejná.

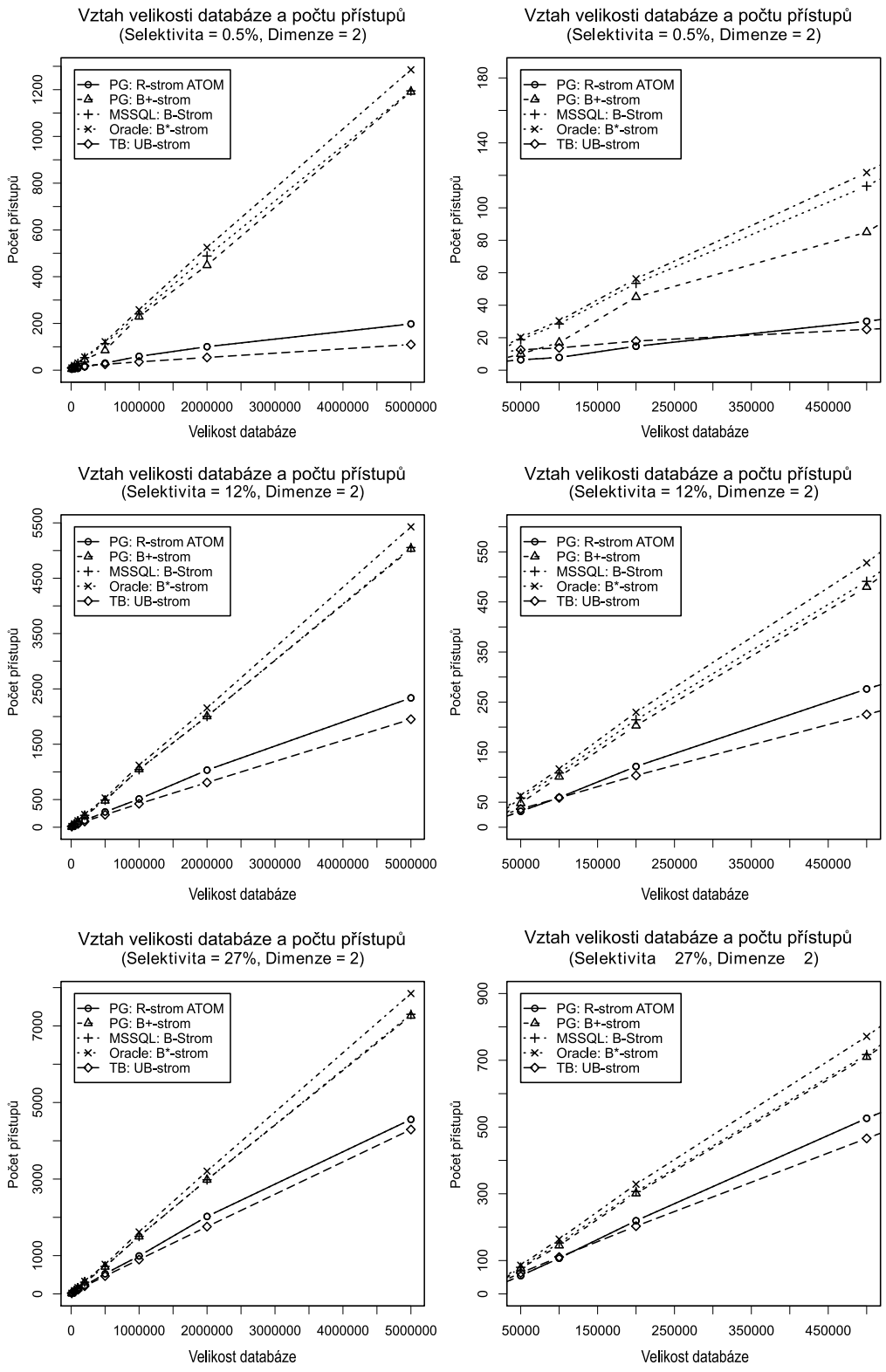
7.5.2 Vliv velikosti databáze při dané dimenzi a selektivitě

Jak ovlivňuje velikost databáze výsledky indexových metod? Jak můžeme vidět na obrázcích 7.4 až 7.6, tak vliv velikosti je lineární. Navíc při dimenzi 1 (obrázek 7.4) vykazuje R-strom z porovnávaných metod nejlepší efektivitu. Rozdíl mezi R-stromem a ostatními metodami zůstává při různé selektivitě prakticky stejný (mluvíme zde přirozeně o relativní podobnosti, protože počet přístupů, jak si můžeme všimnout rapidně roste). To je zřejmý rozdíl mezi dimenzí jedna a dvě, kdy R-strom a UB-strom (tedy multidimenzionální metody) jsou při nízké selektivitě zřejmě efektivnější, než B-stromy.



Obrázek 7.4: Vliv velikosti databáze při dimenzi 1

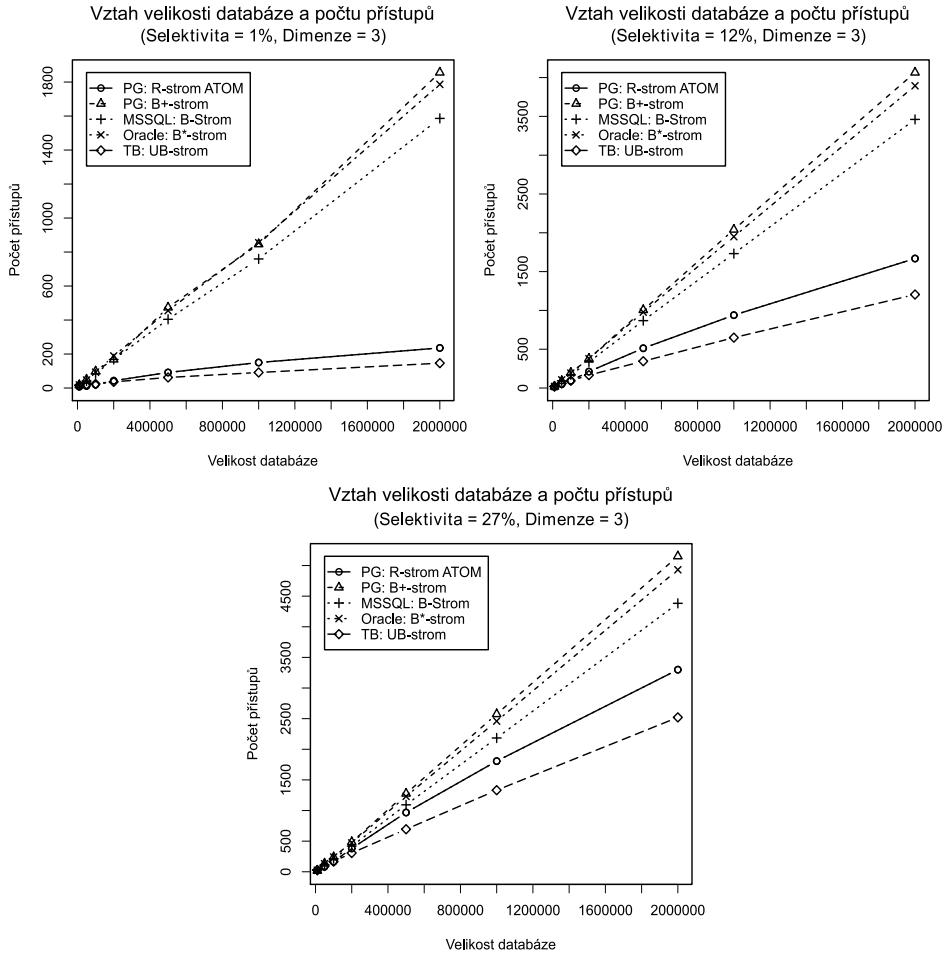
I ve dvou dimenzích počet přístupů roste lineárně, nicméně je zde rozdíl v rychlosti růstu B-stromových metod a metod multidimenzionálních. Zatímco při velikosti databáze $2 * 10^6$ jsou metody multidimenzionální zhruba 2x efektivnější, tak při velikost $5 * 10^6$ jsou efektivnější zhruba 7x.



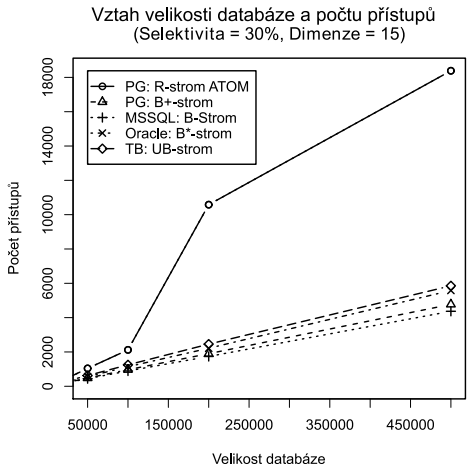
Obrázek 7.5: Vliv velikosti databáze při dimenzi 2

Ve 2 dimenzích se viditelně projevuje trend, který ukazuje, že při vyšší selektivitě se rozdíl mezi B-stromovými metodami a metodami multidimenzionálními smazává.

Tento trend můžeme dále pozorovat ve 3 dimenzích, kde je při selektivitě 27 ještě výraznější. Je tedy zřejmé, že se zvyšující se selektivitou a dimenzí se ztrácí výhoda multidimenzionálních metod, zatímco při nízké selektivitě je tomu naopak (nicméně u velmi vysokých dimenzí to již neplatí). Toto je podpořeno extrémním případem, kdy na obrázku 7.7 vidíme srovnání metod při dimenzi 15 a selektivitě 30, kde B-stromové metody jsou efektivnější. Dále vidíme, že ačkoli UB-strom je horší, než B-stromy, tak růst počtu přístupů se vzrůstající velikostí databáze je velice pozvolný (stejně jako u B-stromů) v porovnání s R-stromem. Toto je způsobeno tím, že při dimenzi 15 už silně v R-stromu roste počet překryvů uzlů a tedy roste počet větví, které je třeba při průchodu stromem projít. Jak napovídá obrázek 7.7, s rostoucí velikostí databáze roste počet přístupů výrazněji. UB-strom tímto netrpí. Pouze se přirozeně zhoršuje schopnost optimálně porovnat vzájemnou vzdálenost bodů po transformaci z vícerozměrného do 1-rozměrného prostoru při růstu počtu bodů.



Obrázek 7.6: Vliv velikosti databáze při dimenzi 3

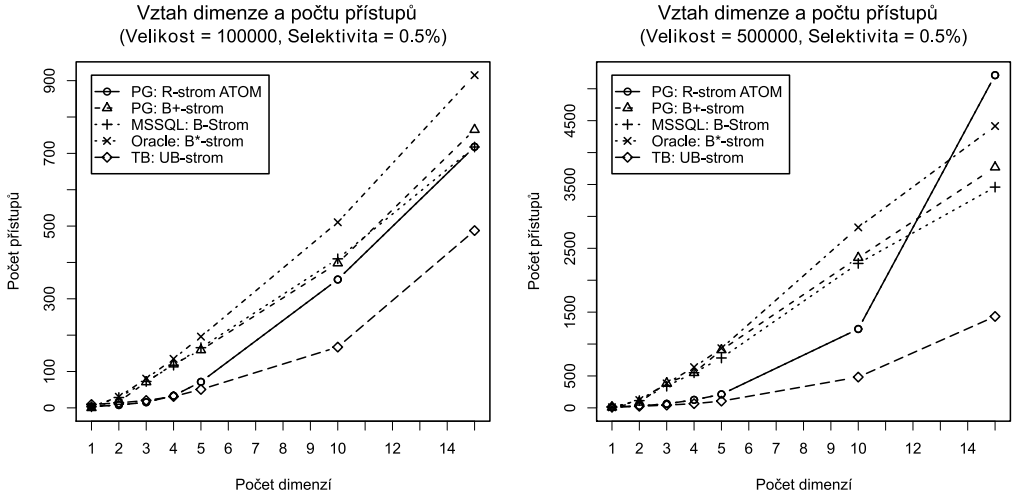


Obrázek 7.7: Vliv velikosti databáze při dimenzi 3

7.5.3 Vliv dimenze při dané selektivitě a velikosti databáze

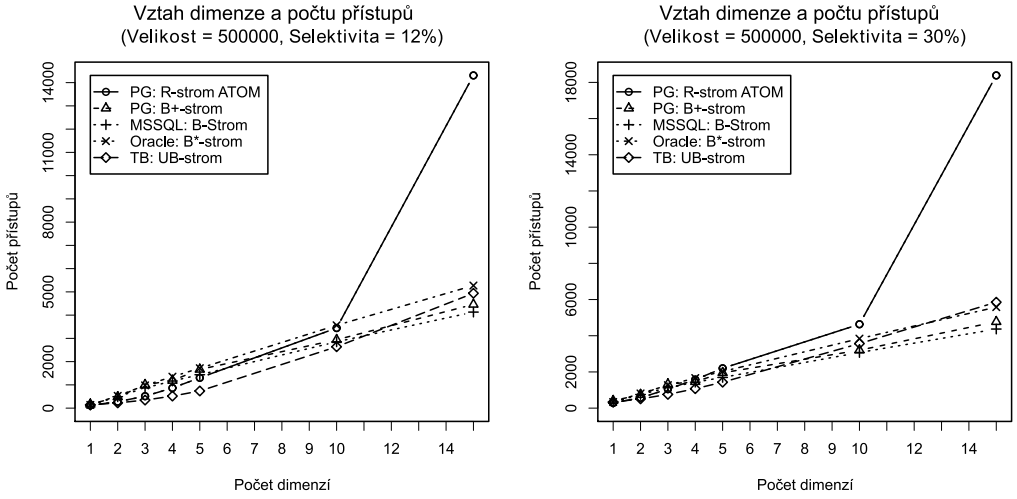
Předchozí kapitola napověděla, že se R-strom pro vysoké dimenze a veliké databáze stává neefektivní. Nyní budeme zkoumat právě vliv dimenze na efektivitu přístupových metod. Ve vysokých dimenzích roste výrazně počet překryvů a tudíž je přirozené snížení efektivity R-stromu. Zvýšení počtu přístupů by mělo být dokonce exponenciální (důsledek *prokletí dimenzionality*). Otázkou tedy je, od jaké dimenze už není R-strom použitelný?

Nejdříve zkoumejme vliv dimenzionality při nízké selektivitě, kde, jak se ukázalo, je síla multidimenzionálních metod oproti B-stromu. Na obrázku 7.8 vidíme rostoucí dimenzi pro velikosti databáze 100000 a 500000. Již v testech na velikost databáze se projevilo, že tato má vliv na výkon především R-stromu ve vysokých dimenzích a zde se to potvrzuje. Od páté dimenze se začíná efektivita R-stromu zhoršovat a nastává exponenciální nárůst počtu přístupů. Tento nárůst je ještě výraznější při zvyšující se velikosti databáze. Naproti tomu nárůst UB-stromu i pro vysoké dimenze je poměrně stejný, jako u B-stromových metod.



Obrázek 7.8: Vliv dimenze při nízké selektivitě

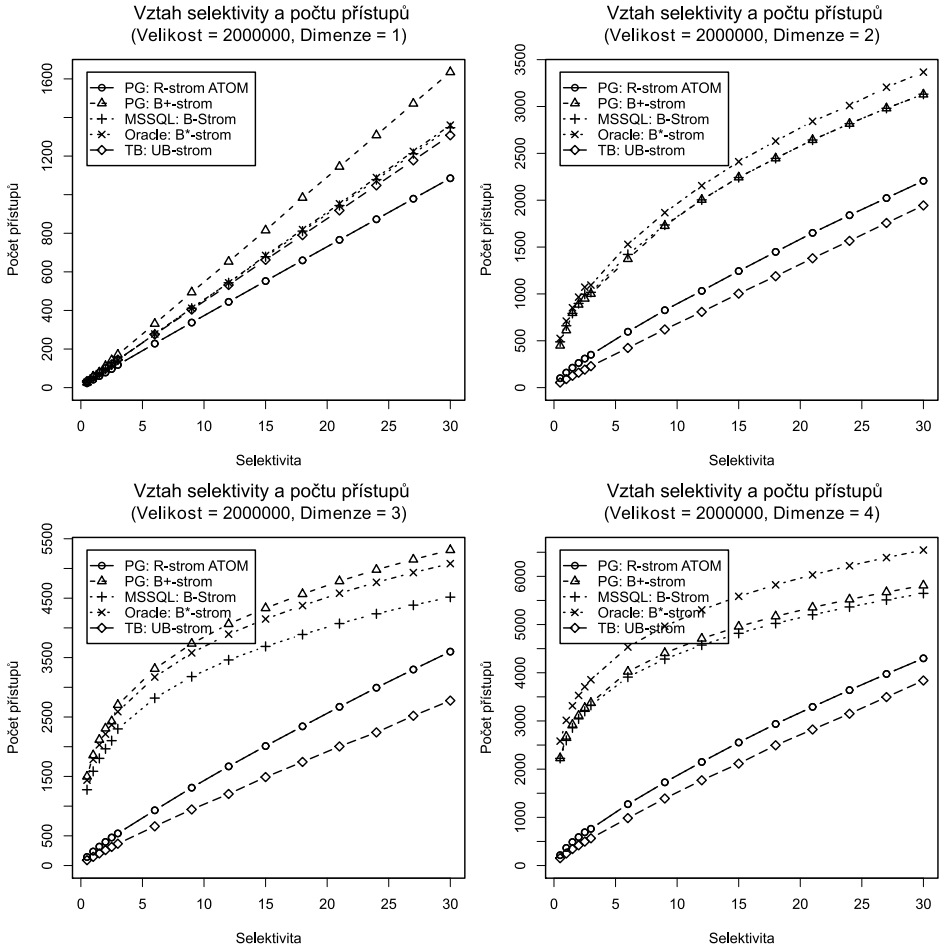
Fakt, že nárůst počtu přístupů u UB-stromu a B-stromů je relativně stejný je názorně vidět na obrázku 7.9, kde efektivita všech metod je podobná (což je způsobeno vyšší selektivitou), ale zatímco R-strom zcela odpodává, tak UB-strom vykazuje velice podobný nárůst jako B-stromy. Nicméně i zde je vidět tendence UB-stromu ke zhoršování efektivity.



Obrázek 7.9: Vliv dimenze při zvyšující se selektivitě

7.5.4 Vliv selektivity při dané dimenzi a velikosti databáze

Z předchozích testů vyplynulo, že dimenze je jedním z hlavních faktorů, který určuje míru rozdílu v efektivitě metod B-stromových a metod multidimenzionálních. Ukázalo se, že při nízké selektivitě jsou multidimenzionální metody výrazně lepší, než metody B-stromové a s rostoucí selektivitou se rozdíl stírá. Vyvstává otázka, zda se při vysoké selektivitě zhoršuje výkon multidimenzionálních indexů, nebo se naopak zlepšuje výkon B-stromových. Z pohledu na obrázek 7.10 jednoznačně plyne, že správná je druhá ze zmíněných variant. Multidimenzionální metody s rostoucí selektivitou rostou přísně lineárně, zatímco všechny B-stromové metody zlepšují logaritmicky svůj výkon. Toto je způsobeno tím, že nevýhoda shlukování podle prvního atributu se nutně stírá s počtem selektovaných záznamů. Navštíví-li totiž algoritmus B-stromu při průchodu stromem uzel, který je "špatně" umístěn, pak při zvyšujícím se počtu selektovaných záznamů se také zvyšuje pravděpodobnost, že ve stejné části stromu bude více záznamů, které je třeba vybrat a tudíž se snižuje "cena" za vyhledání daného záznamu (průchod větví stromu).



Obrázek 7.10: Vliv selektivity při zvyšující se dimenzi

7.5.5 Gaussovské rozložení

V gaussovském rozložení je zajímavé testovat, jak se metody chovají, pokud se okno dotazu přibližuje ke středu prostoru, protože tím roste selektivita dotazu. Podívejme se tedy na obrázky 7.11 a 7.12.

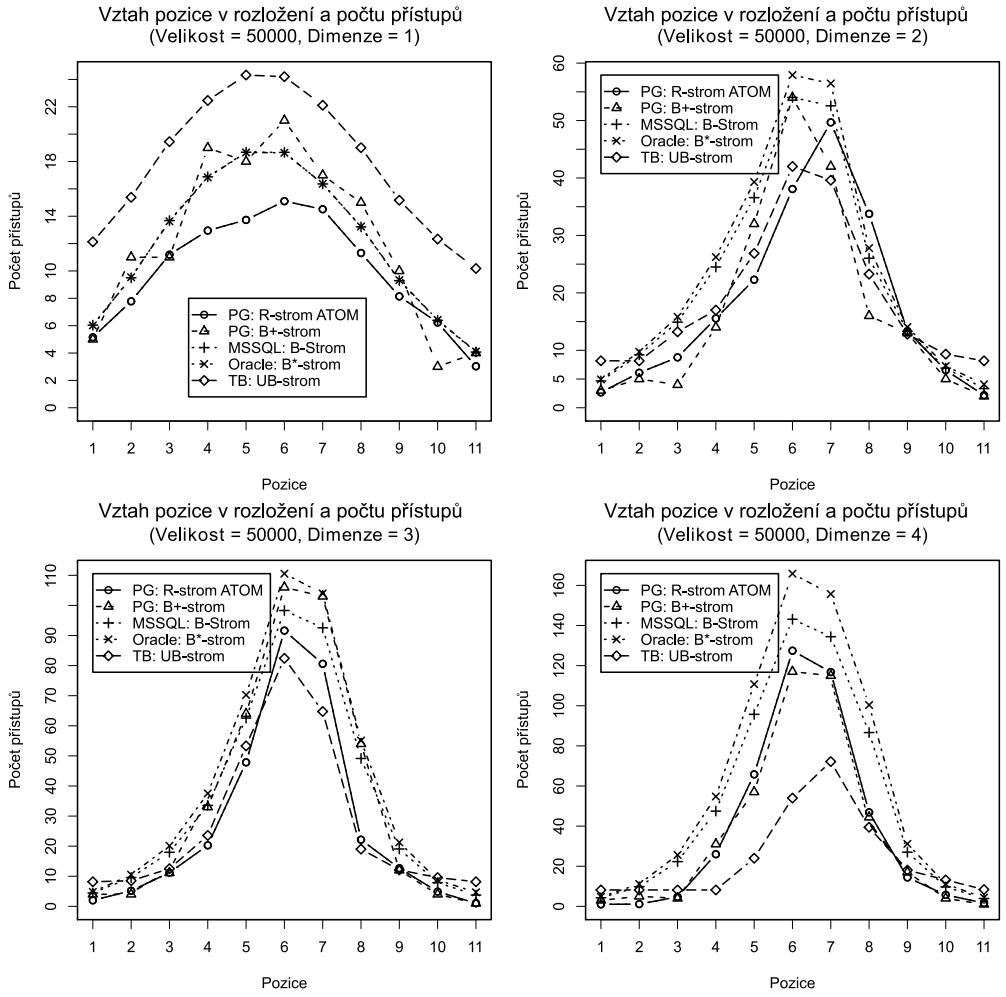
Každý z obrázků vyjadřuje změnu počtu přístupů při posouvání okna dotazu přes úhlopříčku prostoru a to pro zvyšující se dimenze na dvou velikostech databáze. S rostoucí dimenzí pozorujeme zvyšování rozdílu mezi R-stromem a UB-stromem na straně jedné a mezi B-stromovými metodami na straně druhé. Důvodem je především způsob dotazování. U testů na gaussovském rozložení nahrazuje konstantní selektivitu konstantní velikost dotazovacího okna. Se zvyšující se dimenzí se snižuje selektivita pro dané dotazovací okno (viz. tabulky 7.4 a 7.5), protože se zvyšuje pravděpodobnost, že alespoň v jedné dimenzi bod vybočí ze zvoleného dotazovacího okna. Proto můžeme vidět, že pro rostoucí dimenze se zlepšuje efektivita multidimenzionálních metod. Vezmeme-li do úvahy výsledky z kapitoly 7.5.4, kde byl studován vliv selektivity, a uvážíme-li selektivity dotazů, pak by mělo platit, že multidimenzionální metody by měli být viditelně lepší, než B-stromové a že UB-strom by pak měl dosahovat nejlepších výsledků. Toto vše se zde opravdu potvrzuje, ale zajímavý je rozdíl mezi R-stromem a UB-stromem. Pro velikost databáze 10^6 a dimenzi 4 je uprostřed prostoru UB-strom zhruba 8x efektivnější. Tento výrazný rozdíl, oproti uniformnímu rozložení lze pravděpodobně připsat faktu, že body v gaussovském rozložení jsou hustěji nakumulovány uprostřed prostoru, čímž nutně roste počet překryvů v R-stromu, což je hlavní determinant jeho výkonu.

Na okrajích prostoru se snižuje relativně rozdíl mezi selektivitami v různých dimenzí a to z toho důvodu, že se snižuje sama selektivita v důsledku gaussovského rozložení dat a to na takovou úroveň, kde už začíná být rozdíl mezi jednotlivými metodami minimální.

Dim	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11
1	1933	3391	5215	6658	7519	7462	6478	5047	3270	1961	879
2	5	33	245	1102	3053	6256	6182	2000	320	37	2
3	0	1	31	492	3816	8794	5072	695	24	1	0
4	0	0	0	10	332	2775	3547	1043	47	1	0

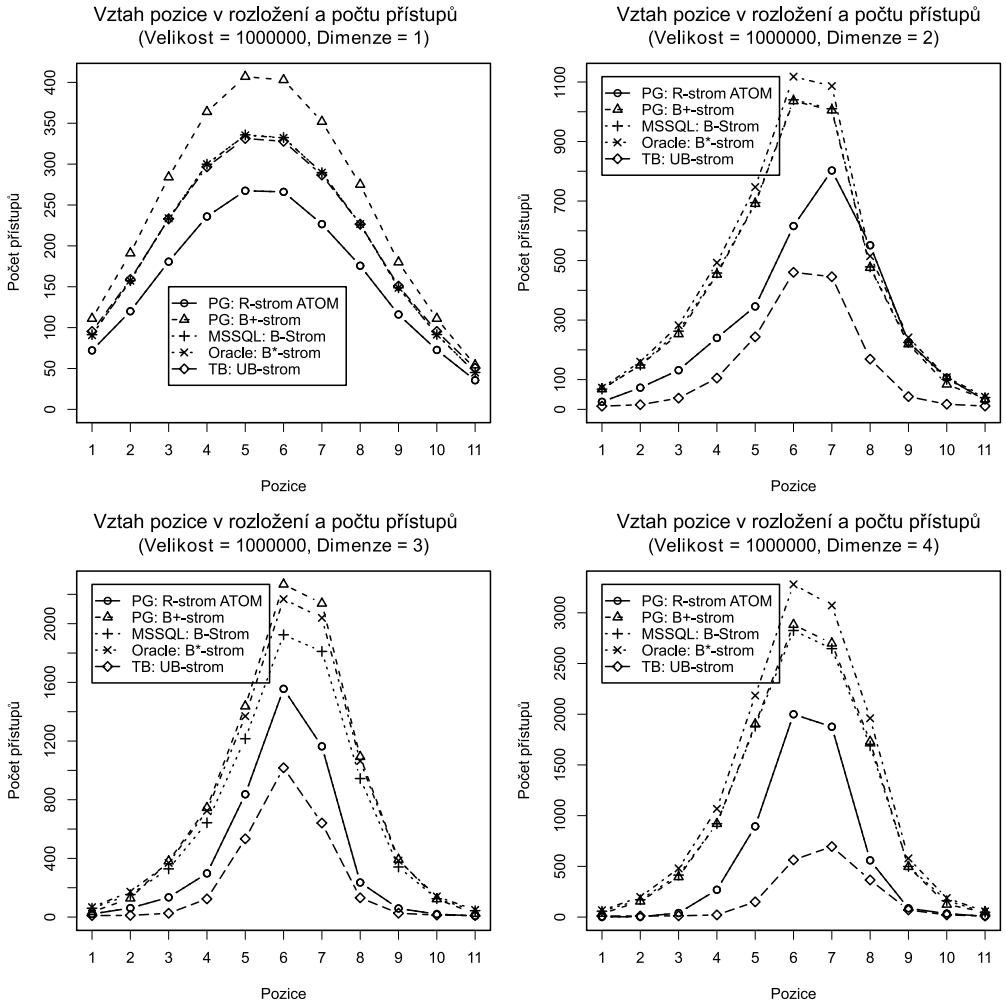
Tabulka 7.4: Počet navrácených záznamu při gaussovském rozložení při velikosti databáze 50000

Velice zajímavé je pak porovnání vzájemného pořadí efektivity jednotlivých metod v gaussovském rozložení a v rozložení uniformním. Posouvání dotazovacího okna po úhlopříčce odpovídá změně selektivity, jak bylo výše zmíněno. Tedy pořadí při dané velikosti databáze a dimenzi by mělo reflektovat srovnatelné grafy v obou typech rozložení. Porovnáme-li obrázek 7.12 (gaussovské rozložení s velikostí databáze 10^6) s obrázkem 7.10 (uniformní rozložení s velikostí databáze $2 * 10^6$), tak můžeme vidět, že tento předpoklad je splněn. Jediná dimenze, kde je R-strom efektivnější než UB-strom je v obou rozloženích 1. V dimenzích 2 a 4 je nejhorší B^*-strom a B^+ i B^{++}-strom jsou prakticky stejně efektivní, což



Obrázek 7.11: Gaussovské rozložení na malé databázi

platí znovu pro obě rozložení. A nakonec v 3. dimenzi si relativně pohoršuje B^+-strom, který vykazuje nejhorší efektivitu (znovu v obou rozloženích).



Obrázek 7.12: Gaussovské rozložení na velké databázi

Dim	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11
1	39777	69303	103601	133329	149180	147775	128678	100415	65461	39716	18806
2	73	799	4917	22075	60532	124466	121770	39205	6307	890	65
3	0	11	567	10118	75900	175444	99758	13745	419	4	0
4	0	0	1	135	6958	55613	70802	21196	920	14	0

Tabulka 7.5: Počet navrácených záznamu při gaussovském rozložení při velikosti databáze 1000000

7.5.6 Reálné naměřené časy

Ve všech předchozích testech jsme uváděli pouze počet načtených stránek, v čemž byly zahrnuty přístupy do paměti i na disk. Rozložení počtu těchto typů přístupů hraje významnou roli při měření času, jelikož přístupy do paměti jsou rychlejší. Z toho také plyne problém při porovnávání efektivity jednotlivých metod pouze z hlediska času na různých databázových platformách, protože různé platformy používají různé metody pro přístup k disku a především jinak nakládají s pamětí, kterou si rezervují v rámci svého procesu v operačním systému, čímž ovlivňují poměr stránek načtených z paměti a z disku, a následkem toho snižují, resp. zvyšují čas potřebný k vykonání dotazu.

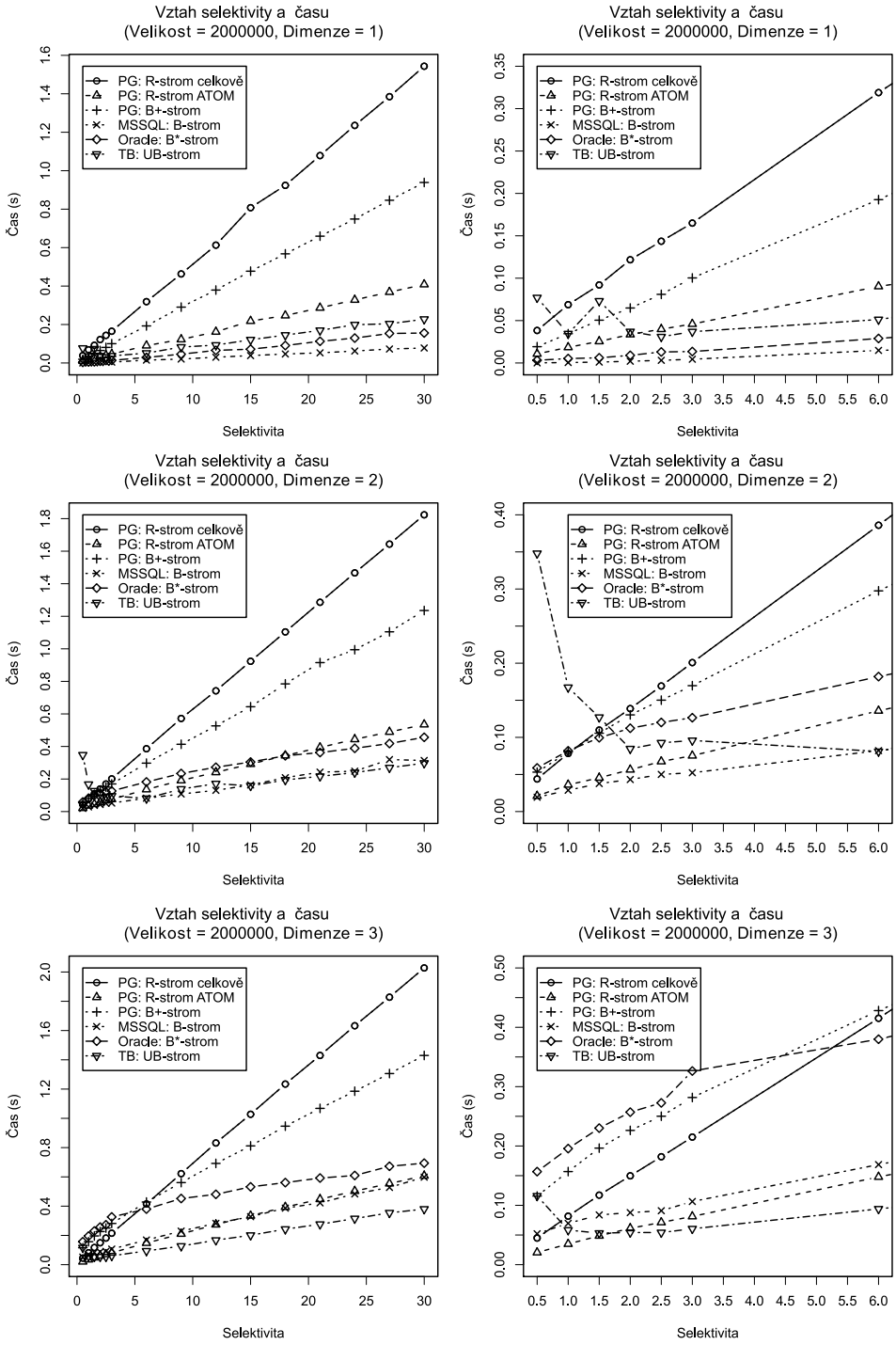
K následujícím výsledkům je třeba upozornit na způsob, jakým se jednotlivé platformy k paměti chovají a jak s ní nakládají. Každá z testovaných platforem, kromě MS SQL Serveru, se chovala k paměti korektně z toho hlediska, že za dobu svého běhu velikost zabrané paměti víceméně nezvyšovala[3]. Velikost zabrané paměti MS SQL Serverem se ovšem dostala přibližně k 1 GB. Je pravděpodobné, že tuto paměť MS SQL Server využívá pro cache, tedy pro urychlení vyhledávání.

Chování MS SQL Serveru prezentované v minulém odstavci je příkladem, proč měření pouze času není z hlediska měření efektivity jednotlivých metod příliš vhodné. Na druhou stranu toto měření může poukázat na neefektivní chování databázového serveru při vyhledávání prvků, nebo obecně při práci s pamětí.

Na obrázku 7.13 vidíme dosažené časy pro zvyšující se selektivitu na databázi o velikosti $2 * 10^6$ záznamů. Je to komplementární obrázek k obrázku 7.10, který zobrazuje počty přístupů. První, čeho si asi z pohledu na obrázek každý všimne (zvláště pak při pohledu na pravý sloupec, který zobrazuje detaily příslušných obrázků z levého sloupce do selektivity 6%), je zvláštní chování Transbasu při nízké selektivitě (zhruba do selektivity 6). Toto nedeterministické chování se projevuje na různých velikostech databáze i v různých dimenzích a zdá se to tedy být *bug* Transbasu[4]. Mimo tyto anomálie ovšem Transbase vykazuje dobré výsledky odpovídající nízkému počtu načtených stránek. Dále pak pozorujeme, jak systémy Oracle a MS SQL Server pracují natolik efektivně, že dokážou z hlediska času takřka vyrovnat výhody R-stromu. Porovnáme-li časy a počty přístupů, tak vidíme, že pouze v dimenzi 3 R-strom vykazoval tak vysoký rozdíl v počtu přístupů, že dokázal i časově předčít efektivní fungování Oraclu a MS SQL Serveru. To jsme ovšem mluvili o času vyhledávání v R-stromu, který spotřebuje pouze ATOM. Připočteme-li režii PostgreSQL, pak jsou časy přibližně dvakrát až třikrát vyšší. Toto chování vychází očividně z neefektivní práce PostgreSQL. Vezmeme-li totiž v úvahu velice slušné výsledky, co se diskových stránek týče B^+-stromu v PostgreSQL, pak můžeme vidět, že z hlediska dosažených časů zcela propadává.

[3]Různé platformy ovšem měli různou velikost cache

[4]Počet přístupů ovšem nevybočuje.



Obrázek 7.13: Dosažené časy

7.5.7 Reálná data

V této kapitole se budeme zabývat testy na reálných datech jejichž struktura je popsána v kapitole 7.1.3. Nejdříve se podívejme na tabulku 7.6 obsahující počty přístupů pro jednotlivé dimenze a selektivity. Na první pohled vidíme členění, co se výkonu týče, podle dimenzí. V rámci jedné dimenze pak při takto nízké selektivitě[5] nevykazují počty přístupů pro rostoucí selektivitu nějaký viditelný trend. Ovšem mezi různými dimenzemi už je trend viditelný - s rostoucí dimenzí roste počet přístupů. Nejzajímavější je ovšem přechod, mezi dimenzí jedna a dva. Tam můžeme zcela zřejmě vidět nárůst počtu přístupů u B-stromových metod, zatímco R-strom a UB-strom zvyšují počet přístupů pouze mírně. Důvodem je již dříve zmíněná asymetrie prvního atributu při složených klíčích. Navíc již na testech uniformního rozložení jsme se přesvědčili, že při nižší selektivitě roste rozdíl mezi B-stromovými metodami a metodami multidimenzionálními. Zde vidíme extrémní příklad nízké selektivity, z čehož plyne rozdíl, který sledujeme.

Při pohledu na UB-strom je zajímavé sledovat, že při nízké dimenzi je počet přístupů relativně vysoký, ale na druhou stranu pro rostoucí dimenze vykazuje pouze mírné zhoršení výkonu. To odpovídá chování, které jsme viděli při uniformním rozložení na obrázku 7.8. Mezi B-stromy lehce propadá B^+-strom v PostgreSQL, zatímco efektivity stromů Oraclu a Microsoft SQL Serveru jsou prakticky totožné.

Porovnáme-li pak počty přístupů s reálnými dosaženými časy (tabulka 7.7), pak pozorujeme podobnou změnu v pořadí jednotlivých metod, jako tomu bylo při sledování reálných časů při uniformním rozložení. Microsoft SQL Server a Oracle dokáží svojí prací s pamětí zcela eliminovat výhody R-stromu i UB-stromu. Naopak PostgreSQL se prezentuje oproti ostatním platformám relativně špatnou efektivitou (porovnáme-li rozdíl mezi počtem přístupů a dosaženým časem). Veliká režie PostgreSQL plyne i z faktu, že i v případě, kdy je vyzvednut z indexové relace pouze jeden záznam, vzroste čas (oproti času potřebného *ATOMem* k vyhledání záznamu) i více než dvakrát.

[5] 50 záznamů z 435373 dává selektivitu přibližně 0.0001

Dim	Selektivita	PG(R)	PG(B^+)	MSSQL(B^{++})	Oracle(B^*)	TB(UB)
1	1	2	4	3.02	3	7.07
	2	2	4.01	3	3	7.07
	3	2	4	3.02	3	7
	5	2.01	4.01	3.02	3	7
	10	2.02	4.05	3.01	3.01	7.04
	20	2.04	4.03	3.01	3.01	7.01
	50	2.08	4.22	3.1	3.17	7.05
2	1	3.52	60.06	40.85	41.76	7.87
	2	3.62	71.88	51.59	52.87	8
	3	3.62	63.13	43.08	43.99	7.97
	5	3.54	61.66	43.17	44.11	7.83
	10	3.59	62.06	43.51	44.45	7.98
	20	3.64	70.96	49.68	50.77	7.99
	50	4.05	58.92	37.49	38.33	7.92
3	1	5.03	97.1	73.87	73.17	10.48
	2	4.31	94.82	71.42	71.04	10.25
	3	3.95	109.45	78.71	78.88	10.05
	5	4.44	93.06	70.82	70.27	10.18
	10	5.14	98.75	76.29	75.65	10.69
	20	5.19	89.55	65.5	64.82	10.23
	50	5.8	89.37	66.14	65.49	10.45
4	1	10.81	87.87	75.7	76.68	10.75
	2	9.87	86.31	70.83	71.44	12.43
	3	10.73	92.08	76.02	76.1	13.15
	5	8.82	99.51	82.29	81.94	15.21
	10	12.45	102.18	86.24	85.53	14.45
	20	11.22	85.73	68.1	67.9	20.22
	50	11.72	84.74	67.63	67.62	18.6

Tabulka 7.6: Porovnání počtu přístupů na reálných datech

Dim	Selektivita	ATOM(R)	PG(R)	PG(B^+)	MSSQL(B^{++})	Oracle(B^*)	TB(UB)
1	1	2.4	3.9	0.78	0	1.9	3.3
	2	2.5	3.9	0.78	0	2.2	3.27
	3	2.97	4.69	0.63	0	1.1	3.74
	5	2.34	5.47	0.94	0	0.2	3.74
	10	3.44	5.47	0.94	0	0.16	3.44
	20	1.88	3.9	0.78	0	0.19	3.3
	50	1.72	4.22	1.72	0	0.16	3.59
2	1	1.72	4.22	5	0.66	0.62	3.57
	2	3.13	5.31	5.78	0.46	0.93	3.58
	3	3.28	5.94	4.84	0.34	0.73	3.42
	5	2.5	3.91	4.69	0.31	0.82	3.73
	10	3.75	5.63	4.84	0.43	0.95	3.45
	20	2.97	5.94	6.72	0.85	0.89	3.43
	50	3.28	7.19	3.91	0.57	0.76	3.91
3	1	3.44	5.94	7.81	0.92	1.5	2.51
	2	2.97	4.69	6.25	0.87	1.73	3.88
	3	2.66	5	7.19	1.06	1.76	2.82
	5	1.87	4.06	7.66	1.08	1.63	2.19
	10	2.81	5.31	6.88	0.75	1.87	3.32
	20	2.03	4.53	5.94	0.97	1.36	2.98
	50	2.81	5.94	7.81	0.08	1.49	2.81
4	1	3.28	5.5	6.88	1.49	1.73	3.45
	2	3.91	6.88	6.72	1.6	1.69	3.58
	3	5	7.2	7.66	1.3	1.82	2.19
	5	3.59	5.78	7.19	0.6	2.1	3.91
	10	5	7.66	6.56	1.49	2.04	3.88
	20	4.69	7.81	5.78	1.68	1.44	5
	50	4.84	7.34	6.1	1.3	1.75	4

ATOM(R) je čas pouze ATOMu a PG(R) je celkový čas ATOMu s režií PostgreSQL
Nulový čas značí skutečnost, že ani jeden ze 100 dotazů nepřesáhl minimální časovou hranici

Tabulka 7.7: Porovnání dosažených časů na reálných datech (v ms)

Kapitola 8

Závěr

Cílem práce bylo implementování indexové metody pro podporu víceatributového indexování do některého ze stávajících SŘBD. V rámci práce vznikla implementace indexové struktury R-strom do SŘBD PostgreSQL. Jako externí perzistentní framework byl použit *ATOM*. Navíc vznikl framework, který umožňuje relativně jednoduché zapracování obecné externí indexové metody do PostgreSQL.

Implementovaný R-strom byl porovnáván rozsáhlým testováním proti platformám Oracle, PostgreSQL a Microsoft SQL Server, které používají varianty B-stromu se složenými klíči pro víceatributové indexování a proti platformě Transbase, která používá strukturu UB-strom.

Ukázalo se, že použití R-stromu je ve většině případů efektivnější, než použití libovolné úpravy B-stromu se složenými klíči. Výhoda se obzvláště projevuje ve vyšších dimenzích (především od dimenze 2 do dimenze 5). Míra zlepšení závisí v první řadě na selektivitě příslušného dotazu a dimenzi. Efektivita ve vysokých dimenzích (nad 10) je závislá na velikosti databáze, nad kterou vyhledávání probíhá.

Nejlepší výsledky vykazoval R-strom oproti B-stromům při nízkých selektivitách na velikých databázích. Právě tato kombinace je relevantní v systémech, kde je víceatributový index potřeba (datové sklady, dokumentografické informační systémy, ...).

Při jednoatributovém indexování je R-strom překvapivě minimálně stejně efektivní jako B-stromové metody.

Naopak UB-strom je v porovnání s R-stromem efektivnější při vyšší selektivitě a vysoké dimenzi. R-strom je zase lepší v indexování podle jednoho atributu a potom při velice nízkých selektivitách, což se ukázalo při testování nad reálnými daty. Skutečný rozdíl ale nastává ve vysokých dimenzích, kdy nárůst počtu přístupů je u UB-stromu podobný jako u B-stromových metod, zatímco nárůst počtu přístupu R-stromu je výrazný (což je způsobeno růstem počtu překryvů).

Mezi B-stromovými metodami v naprosté většině případů je nejefektivnější variace na B^+-strom v MS SQL Serveru, kde oproti klasickému B-stromu bylo navíc přidáno provázání vnitřních uzlů. V porovnání B^+-stromu (PostgreSQL) a B^*-stromu (Oracle) dopadl lépe B^+-strom, ovšem již ne tak výrazně jako variace MS SQL Serveru a v některých případech byl B^*-strom efektivnější.

Problémem R-stromové implementace v PostgreSQL se ovšem ukázal výkon samotného PostgreSQL spolu s ne zcela optimální architekturou pro podporu externího indexování (nutnost udržovat indexové relace). Ačkoli PostgreSQL má, co se počtu přístupů týče, efektivnější implementaci B-stromu než Oracle, tak časy, které vykazuje jsou oproti Oraclu (a obecně proti ostatním testovaným platformám), tomu neodpovídají[1].

K reálnému nasazení R-stromu s podporou *ATOMu* brání neexistence operace mazání prvku ze stromu a také menší problémy při velikostech indexu nad 300 MB, kdy první přístup k indexovému souboru trvá neobvykle dlouho (v porovnání s ostatními přístupy). Nepříjemná je taky doba nutná k zaindexování velkého množství prvků, která výrazně roste s počtem indexovaných prvků. Po odstranění těchto nedostatků je systém zcela použitelný v reálném prostředí.

Myšlenka použití metod vyvinutých primárně pro prostorové databáze se tedy ukázala jako reálná a životaschopná. V porovnání se stávajícími strukturami používanými v současných komerčních databázových systémech vykazuje v závislosti na podmínkách až čtyřnásobné snížení počtu stránkových přístupů.

[1]Což ovšem může být také způsobeno množstvím paměti, kterou si pro sebe rezervuje a které bylo výrazně menší, než množství paměti, kterou využíval Oracle, nebo MSSQL Server. Na druhou stranu Transbase používal srovnatelné množství paměti a jeho výsledky byly výrazně lepší.

Dodatek A

Popis rozhraní pro AM v PostgreSQL

Následuje seznam funkcí, které by měla přístupová metoda podporovat, pokud má být využitelná backendem Postgresu. Ne všechny metody v seznamu je třeba implementovat (další informace lze získat v dokumentaci k PostgreSQL - viz. [17]).

```
void
ambuild (Relation heapRelation,
         Relation indexRelation,
         IndexInfo *indexInfo)
```

- *heapRelation* - relace, nad kterou je index budován
- *indexRelation* - nově vytvořená indexová relace
- *indexInfo* - informace o indexu (atributy, jejich počet, zda podporuje uniqe atp.)

AM využívá tuto funkci pro zaindexování dat a případné další interní úkony, které potřebuje (například vytvoření struktur pro vyhledávání atp.). Při zavolání této metody tedy již indexová relace indexuje, ale není nijak naplněna. O samotné procházení relace (kvůli vložení jejích prvků do indexu) se nemusí implementátor AM starat. Toto je řešeno pomocí funkce IndexBuildHeapScan, která jako jeden ze svých argumentů dostává funkci typu IndexBuildCallback, které jsou předávány jednotlivé n-tice k zaregistrování. Konkrétní deklarace vypdají následovně:

```
double
IndexBuildHeapScan  (Relation heapRelation,
Relation  indexRelation,
IndexInfo *  indexInfo,
IndexBuildCallback  callback,
void *  callback_state )
```

- *heapRelation* - relace, nad kterou je index budován

- *indexRelation* - nově vytvořená indexová relace
- *indexInfo* - informace o indexu (atributy, jejich počet, zda podporuje uniqe atp.)
- *callback* - funkce obstarávající vložení n-tice do indexu
- *callback_state* - paměť vyhrazená pro cokoliv, co AM potřebuje

```
typedef void (*IndexBuildCallback) (Relation index,
HeapTuple htup,
Datum *attdata,
char *nulls,
bool tupleIsAlive,
void *state)
```

- *index* - indexová relace, do které se má vložit prvek
- *htup* - vkládaný prvek (odkaz na něj)
- *attdata* - data, která se indexují (obsah indexovaných sloupců)
- *nulls* - seznam null hodnot
- *tupleIsAlive* - atribut použitelný při podpoře současné práce více uživatelů
- *state* - paměť vyhrazená pro cokoliv, co AM potřebuje

```
bool
aminsert (Relation indexRelation,
          Datum *values,
          bool *isnull,
          ItemPointer heap_tid,
          Relation heapRelation,
          bool check_uniqueness)
```

- *indexRelation* - indexová relace, do které se n-tice vkládá
- *values* - hodnoty vkládaní n-tice
- *isnull* - seznam null hodnot
- *heap_tid* - odkaz na n-tici učenou k zaindexování
- *heapRelation* - relace, nad kterou je index budován
- *check_uniqueness* - informace, zda je index unique

Tato funkce implementuje vložení n-tice do indexu. Tedy je volána při příkazu *INSERT*. V závislosti na tom, zda metoda podporuje *unique index* (lze zjistit ze sloupce *pg_am.amcanunique*) se také může stát, že vložení bude neúspěšné (AM si musí zkontrolovat, zda v indexu již danou n-tici nemá) a informace o tomto se propaguje pomocí návratové hodnoty do backendu PostgreSQL.

```
IndexBulkDeleteResult *
ambulkdelete (Relation indexRelation,
              IndexBulkDeleteCallback callback,
              void *callback_state)
```

- *indexRelation* - aktuální indexová relace
- *callback* - funkce pro zjištění, zda prvek do ní vložený má být smazán
- *callback_state* - parametr

Funkce bulkdelete není volána, jak by se mohlo zdát tehdy, když je odstraněn prvek z relace. Tato funkce je určena k projití celé relace, zjištění, zda prvek ještě stále v tabulce je a v případě že není, tak ho smazat i z indexu, případně ho v indexu ponechat a nastavit mu nějaký příznak (závisí na dané AM). Z toho, že je procházen celý index plyne, že funkce je volána při operaci VACUUM, která je volána jednou za čas a která volá právě funkci ambulkdelete a navíc příslušně upravuje statistiky tabulek a indexů pro efektivnější práci plánovače.

Na rozdíl od funkce ambuild, zde musí AM procházet index sama.

Rozhodnutí o tom, zda prvek je ještě aktuální náleží funkci typu IndexBulkDeleteCallback, která má následující deklaraci:

```
typedef bool (*IndexBulkDeleteCallback) (ItemPointer itemptr,
 void *state)
```

- *itemptr* - odkaz do tabulky na prvek, na který ukazuje aktuální indexový záznam
- *state* - viz. *callback_state* u ambulkdelete

```
IndexBulkDeleteResult *
amvacuumcleanup (Relation indexRelation,
                 IndexVacuumCleanupInfo *info,
                 IndexBulkDeleteResult *stats)
```

- *indexRelation* - aktuální indexová relace
- *info* - bližší informace o mazání (úroveň statistického reportu atd.)
- *stats* - informace o tom, zda se poslední volání ambulkdelete navrátilo

Tato funkce je volána v rámci příkazu VACUUM. To znamená, že je volána po jednom, nebo více voláních funkce ambulkdelete. Může to být tedy místo, kde lze přeuspořádat index tak, aby využíval místo, které bylo uvolněno při operacích mazání z indexu (není-li toto implementováno již v ambulkdelete).

Tato funkce nemusí být implementována.

Dále následují funkce pro podporu vyhledávání, tj. toho, co je hlavním účelem AM.

```
IndexScanDesc
ambeginscan (Relation indexRelation,
             int nkeys,
             ScanKey key)
```

- *indexRelation* - procházená indexová relace
- *nkeys* - počet argumentů, podle kterých se bude vyhledávat
- *key* - pole struktur *ScanKeyData* o délce nkeys, kde ScanKeyData obsahuje informace o i-tém argumentu podle kterého se indexuje

Začíná nový scan. Vždy, když je zavolán select, který využívá danou AM, tak je zavolána tato funkce, jejíž hlavním cílem je vytvořit nový scan. Co všechno je při tomto volání děláno závisí na indexové metodě. Obvykle je scan zařazen do seznamu scanů, které si AM drží a navíc je naplněna jeho opque struktura (struktura typu void*)libovolnými daty v závislosti na AM. Návratová hodnota vzniká voláním RelationGetIndexScan(). Tato funkce být volána musí, protože Postgres v ní inicializuje IndesScanDesc strukturu, přičemž tato struktura je velice důležitá pro průběh vyhledávání v indexu.

```
boolean
amgettuple (IndexScanDesc scan,
            ScanDirection direction)
```

- *scan* - informace o daném scanu
- *direction* - informace o tom, zda jde scan od posledního začátku ke konci, nebo obráceně

Získává další (ve smyslu daném parametrem direction) prvek v indexu, který splňuje kritéria pro vyhledávání. Funkce vrací hodnotu typu boolean, z čehož plyne, že ona sama nevrací ukazatel na daný prvek. Pouze zjistí, zda prvek v indexu je a v parametru scan nastaví příslušný prvek id n-tice reprezentující indexový záznam odkazující na n-tici nalezeného prvku. V případě, že žádný (další) prvek neodpovídá podmínkám pro vyhledávání, pak funkce vrací false a aktuální scan je ukončen.

```
boolean
amgetmulti (IndexScanDesc scan,
            ItemPointer tids,
            int32 max_tids,
            int32 *returned_tids)
```

- *scan* - informace o daném scanu
- *tids* - navrácené záznamy
- *max_tids* - mazimální počet navrácených záznamů
- *returned_tids* - aktuální počet navrácených záznamů

Pracuje podobně jako amgettuple s tím rozdílem, že může vrátit najednou max_tids záznamů. Tyto záznamy se ukládají do tids. Tato funkce nemůže být implementována, je-li implementována funkce amgettuple. Smysl této funkce je v tom, že AM může více vyhovovat vracet záznamy v určitých "balících", protože to více odpovídá fyzickému uložení záznamů (například jsou-li následné záznamy uloženy v jedné stránce) v indexu a takovýto způsob bude rychlejší, než vyhledávání po jednotlivých prvcích. Návratová hodnota říká Postgresu, stejně jako u amgettuple, zda má daný scan pokračovat, či skončit.

```
void
amrescan (IndexScanDesc scan,
          ScanKey key)
```

- *scan* - informace o daném scanu
- *key* - pole struktur ScanKeyData o délce nkeys, kde ScanKeyData obsahuje informace o i-tém argumentu podle kterého se indexuje

Funkce, která je volána v situaci, kdy je scan startován, nebo restartován. Rozlišení těchto dvou případů je realizováno parametrem key, který je null v případě, že nejde o nový scan a tedy jsou použita stejná struktura atributů pro vyhledávání. Je-li scan startován, pak je funkce volána v rámci funkce RelationGetIndexScan(), která je volána v ambeginscan (viz. výše). Naopak restartování je v praxi uplatňováno tehdy, když je AM použita v "nested-loop" joinu, je-li použita nová vnější n-tice pro join a tudíž je scan použit se stejnou strukturou atributů pro vyhledávání.

```
void
amendscan (IndexScanDesc scan)
```

- *scan* - informace o daném scanu

Z názvu plyne, že amendscan je volána při ukončení scanu. Je určena pro uvolnění prostředků, které byly v průběhu scanu alokovány (jde především o opaque struktury daného scanu).

```
void
ammarkpos (IndexScanDesc scan)
```

- *scan* - informace o daném scanu

Zaznamená současnou pozici ve scanu. V rámci jednoho scanu si každá AM pamatuje pouze jednu pozici (informace o této pozici si ukládá právě do proměnné scan, která má k tomuto účelu vyhrazené místo).

```
void
amrestrpos (IndexScanDesc scan)
```

- *scan* - informace o daném scanu

Nastaví scan na poslední zaznamenanou pozici.

```
void
amcostestimate (PlannerInfo *root,
                IndexOptInfo *index,
                List *indexQuals,
                Cost *indexStartupCost,
                Cost *indexTotalCost,
                Selectivity *indexSelectivity,
                double *indexCorrelation);
```

- *root* - informace o aktuálním dotazu používaná plánovačem
- *index* - informace o aktuálním indexu
- *indexQuals* - seznam kvalifikátorů (podmínek ve WHERE klauzuli)
- **indexStartupCost* - cena inicializace AM při daném dotazu
- **indexTotalCost* - celková cena průběhu indexu
- **indexSelectivity* - selektivita indexu
- **indexCorrelation* - korelace mezi pořadím prvků v indexu a pořadím prvků v tabulce

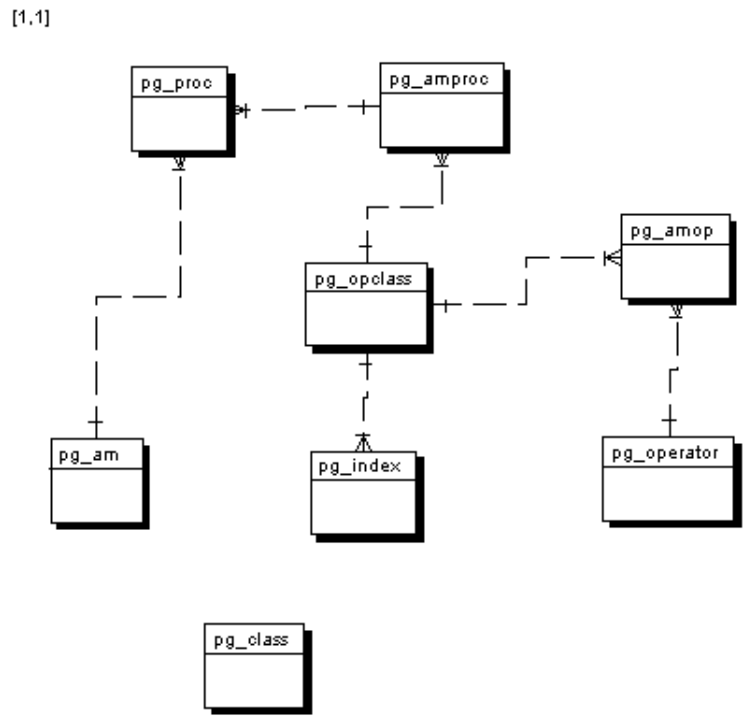
Odhaduje cenu scanu. Tato funkce nemusí být implementována

Dodatek B

Katalog PostgreSQL vztahující se k indexům a přístupovým metodám

Stejně jako prakticky každá databázová platforma udržuje i PostgreSQL katalog obsahující metadata týkající se tabulek, jejich sloupců atd. PostgreSQL jde ovšem v tomto ohledu co nejdále a snaží se v katalogových tabulkách udržovat maximum informací. Udržuje v nich tedy i informace o přístupových metodách a jejich vlastnostech, odkazy na dynamické knihovny obsahující implementace funkcí a další. Na obrázku B.1 můžeme vidět tu část katalogu, která obsahuje metadata relevantní k přístupovým metodám a indexům obecně. Popišme si tedy podrobněji důležité atributy tabulek, které se ve zmíněné části katalogu vyskytují (nemá smysl popisovat zde všechny sloupce).

- *pg_am*
  - Obsahuje informace o přístupových metodách. Pro každou přístupovou metodu zde existuje jeden řádek.
    Relevantní atributy jsou:
    * *amname* - jméno přístupové metody
    * *amstrategies* - počet operátorových strategií
    * *amsupport* - počet podpůrných metod
    * *amorderstrategy* - nula, když metoda nepodporuje třídění, jinak číslo strategie, která má třídění starosti
    * *amcanunique* - zda metoda podporuje unikátní indexy
    * *amcanmulticol* - zda metoda podporuje víceatributové indexování
    * *amoptionalkey* - zda metoda podporuje takový víceatributový index, kde není v podmínce uveden první sloupec
    * *amindexnulls* - zda metoda indexuje null hodnoty
    * *amconcurrent* - zda metoda podporuje souběžné updaty
    * *aminsert* - funkce pro vkládání (odkaz do *pg_proc*)
    * *ambeginscan* - funkce pro inicializaci scanu (odkaz do *pg_proc*)



Obrázek B.1: ER schéma části katalogu PostgreSQL

  * *amgettuple* - funkce pro získání n-tice (odkaz do *pg_proc*)
  * *amgetmulti* - funkce pro získání definovaného počtu n-tic (odkaz do *pg_proc*)
  * *amrescan* - funkce pro znovu započetí scanu (odkaz do *pg_proc*)
  * *amendscan* - funkce pro ukončení scanu (odkaz do *pg_proc*)
  * *ammarkpos* - funkce pro označení aktuální pozice ve scanu (odkaz do *pg_proc*)
  * *amrestrpos* - funkce pro vrácení se k označené pozici ve scanu (odkaz do *pg_proc*)
  * *ambuild* - funkce pro postavení indexu na existující tabulce (odkaz do *pg_proc*)
  * *ambulkdelete* - funkce pro hromadné mazání z indexu (odkaz do *pg_proc*)
  * *amvacuumcleanup* - funkce pro volitelné operace prováděné po volání příkazu VACUUM (odkaz do *pg_proc*)
  * *amcostestimate* - funkce pro odhad doby trvání vrácení n-tice (odkaz do *pg_proc*)

- *pg_amop*
  - Obsahuje informace o operátorech asociovaných s přístupovými metodami. Relevantní atributy jsou:

    - *amopclaid* - určuje, ke které třídě operátorů tento operátor patří (odkaz do *pg_opclass*)
    - *amopsubtype* - rozlišuje mezi více operátory k jedné třídě operátorů (nula znamená defaultní)
    - *amopstrategy* - číslo strategie
    - *amopopr* - operátor (odkaz do tabulky *pg_operator*)

- *pg_amproc*
  - Obsahuje informace o podpůrných metodách asociovaných s přístupovými metodami.
    Relevantní atributy jsou:
    - *amopclaid* - určuje, ke které třídě operátorů tato metoda patří (odkaz do *pg_opclass*)
    - *amprocnum* - číslo podpůrné procedury
    - *amproc* - funkce (odkaz do tabulky *pg_proc*)

- *pg_class*
  - Obsahuje informace o objektech, které mají strukturu tabulek (tedy obsahují sloupce) a tabulky samotné.
    Relevantní atributy jsou:
    - *relname* - jméno tabulky, indexu, ...
    - *reltype* - odkaz do tabulky *pg_type*
    - *relam* - je-li to index, pak je to informace o použité přístupové metodě (odkaz do *pg_am*)
    - *relfilenode* - jméno souboru na disku obsahující danou relaci (co relace, to jeden diskový soubor)
    - *reltuples* - počet řádek
    - *relhasindex* - jde-li o tabulku, pak informace, zda obsahuje , nebo obsahovala index (informace se zde nemaže při DROP INDEX, nýbrž při volání VACUUM)
    - *relkind* - znak udávající typ relace (r = tabulka, i = index, s = sekvence, ...)
    - *relnatts* - počet atributů
    - *relhaspkey* - jde-li o tabulku, pak informace o tom, zda obsahuje primární klíč

- *pg_index*
  - Obsahuje část informací o indexech (zbytek je v *pg_class*).
    Relevantní atributy jsou:

    * *indexrelid* - třída pro daný index (odkaz do tabulky *pg_class*)
    * *indrelid* - třída pro tabulku nad kterou je daný index (odkaz do tabulky *pg_class*)
    * *indnatts* - počet atributů, nad kterými je index vystavěn
    * *indisunique* - zda je index unikátní
    * *indisclustered* - zda je index klastrovaný
    * *indkey* - pole o délce indnatts značící, pro které atributy tabulky je index nadefinován (jednotlivé prvky pole jsou odkazy do tabulky *pg_attribute*)
    * *indclass* - pole značící pro každý atribut, která třída operátorů se má použít (odkaz do tabulky *pg_opclass*)
    * *indexprs* - strom výrazu, neboli string, který obsahuje výraz z WHERE podmínky pro sloupce, nad kterými je index (null pokud jde pouze o jednoduché restrikce)

- *pg_opclass*
  - Obsahuje informace o třídách operátorů asociovaných s přístupovými metodami. Třída operátoru definuje sémantiku pro indexový sloupec určitého typu určité přístupové metody.
    Relevantní atributy jsou:
    * *opcamid* - přístupová metoda, pro kterou je daná třída operátorů definována (odkaz do tabulky *pg_amop*)
    * *opcname* - jméno třídy operátorů
    * *opcintype* - typ dat, které daná třída operátorů indexuje (odkaz do tabulky *pg_type*)
    * *opcdefault* - zda je daná třída operátorů defaultní pro typ opcintype

- *pg_proc*
  - Obsahuje informace o funkcích.
    Relevantní atributy jsou:
    * *proname* - jméno funkce
    * *prolang* - jazyk v kterém je procedura napsána (odkaz do tabulky *pg_lang*)
    * *proisagg* - zda jde o agregační funkci
    * *pronargs* - počet argumentů
    * *prorettype* - návratový typ (odkaz do tabulky *pg_type*)
    * *proargtypes* - pole typů vstupních parametrů, tedy určuje volací signaturu funkce (prvky pole jsou odkazy do tabulky pg_type)
    * *proallargtypes* - pole typů všech parametrů (prvky pole jsou odkazy do tabulky *pg_type*)

* *prosrc* - podle typu funkce je zde buď zdrojový kód interpretované funkce, jméno souboru, nebo cokoliv jiného podle jazykové/volací konvence dané funkce
* *probin* - dodatečná informace o tom, jak danou funkci volat (závisí na jazykové/volací konvenci dané funkce)

Seznam obrázků

Seznam tabulek

Literatura

[1] Korry Douglas, Susan Douglas. PostgreSQL - A comprehensive guide to building, programming, and administerring PostgreSQL databases.*Sams Publishing*, 2003

[2] Volker Markl, Frank Ramsak, Roland Pieringer, Robert Fenk, Klaus Elhardt, Rudolf Bayer. The Transbase Hypercube RDBMS: Multidimensional Indexing of Relational Tables. *ICDE Demo Sessions*, 2001: 4-6

[3] Rudolf Bayer, Edward M. McCreight. Organization and Maintenance of Large Ordered Indexes. *SIGFIDET Workshop* 1970: 107-141

[4] Rudolf Bayer, Edward M. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica 1*: 173-189 (1972)

[5] Antonin Guttman.R-trees: A Dynamic Index Structure for Spatial Searching. *Proceedings of the ACM SIGMOD, Boston, MA*, June 1984

[6] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, Bernhard Seeger.The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. *SIGMOD Conference* 1990: 322-331

[7] R. Bayer. The Universal B-Tree for multidimensional indexing. *Institut für Informatik, TU München*, 1996

[8] Volker Markl. MISTRAL: Processing Relational Queries using a Multidimensional Access Technique. *Institut für Informatik, TU München.* 1999

[9] Amphora Research Group. Amphora Tree Object Model Book. `http://arg.vsb.cz`, 2006

[10] Michal Krátký, Tomáš Skopal, Václav Snášel. Porovnání některých metod pro vyhledávání a indexování multimediálních dat. *Kybernetika - história, perspektívy, teória a prax, Žilinská univerzita*, 2002

[11] Rudolf Bayer, Volker Markl. The UB-Tree: Performance of Multidimensional Range Queries. *Technical Report TUMI9814, Institut für Informatik, TU München*, 1997

[12] Cui Yu. High-Dimensional Indexing: Transformational Approaches to High-Dimensional Range and Similarity Searches. *Lecture Notes in Computer Science, Volume 2341, Springer*, 2002

[13] Jolly Chen, Andrew Yu: The Postgres Implementation Guide (chapter 7). *University of California at Berkeley*, 1995 (`http://pluto.iis.nsk.su/postgres95/impl-guide/`, 2006)

[14] Marek Matula. Diplomová práce: Vyhledávání ve vícerozměrných datech pomocí Pyramid-stromu. *VŠB - Technická Univerzita Ostrava*, 2004

[15] John H. Miller and Henry Lau: Microsoft SQL Server 2000 RDBMS Performance Tuning Guide for Data Warehousing. *Microsoft Technet*, 2001, (`http://www.microsoft.com/technet/prodtechnol/sql/2000/maintain/rdbmspft.mspx`, 2006)

[16] Raghu Ramakrishnan,Johannes Gehrke: Database Management Systems. *McGraw-Hill Science/Engineering/Math; 3 edition*, August 14, 2002

[17] Dokumentace PostgreSQL. `http://dev.mysql.com/doc/refman/5.1/en/custom-engine-index.html`, 2006

[18] Dokumentace MySQL. `http://dev.mysql.com/doc/refman/5.1/en/custom-engine-index.html`, 2006

[19] Emulace Linuxu pro Windows. `http://www.mingw.org`, 2006

[20] DBLP - Digital Bibliography & Library Project. `http://dblp.uni-trier.de/`, 2006