36. Prostorové DB (problematika mapování prostoru, ukládání, indexace; využití).

Prostorové databáze se využívají pro ukládání prostorových dat. Rozlišujeme klasická prostorová data (typicky 2D nebo 3D) a virtuální prostorová data (výrazně vyšší počet dimenzí).
Příklady klasických prostorových dat:

- mapové informace, např. ulice, města, silnice, lesy
- struktura chemických sloučenin v prostoru
- struktura vesmíru, galaxií, …

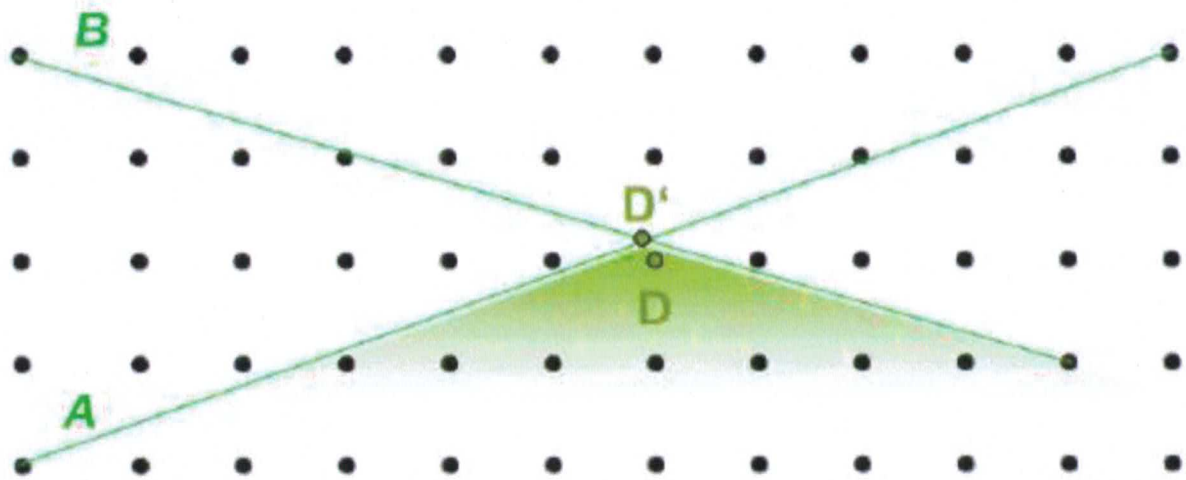
Naopak u virtuálních prostorových dat typicky pracujeme s nějakými vektory charakteristik (např. textu, obrazu, zvuku) nebo vektory nějakých numerických atributů (např. parametry nějakého produktu – notebook má cenu, RAM, velikost, váhu, úhlopříčku displeje, …). Kromě samotného ukládání (a případného indexování) dat chceme typicky také vykonávat určité dotazy nad daty a zkoumat závislosti. Rozdíl mezi prostorovou databází a např. klasickou relační (kde bychom si jistě mohli vytvořit sloupec X a Y a považovat tyto sloupce za souřadnice) je, že prostorová databáze má pro ukládání prostorových dat speciální datové typy a umožňuje nad nimi jednoduché dotazování (to probíhá na úrovni databázového systému, nikoliv na aplikační úrovni).

Základní **datové typy**, které chceme v databázích ukládat:

- **body** (reprezentují města, planety, slunce, atomy, …) – nejméně 2D
- **(lomené) úsečky** – popsány nejméně dvěma body, proměnlivá délka v případě lomené úsečky
- **polygony** = uzavřená lomená úsečka – nejméně 3 body
- **plochy** ve 2D nebo **objemy** ve 3D – proměnlivý počet bodů

Problematika mapování

V reálném světě pro reprezentaci souřadnic využíváme reálná čísla. Při ukládání do počítače tedy nastává problém s převodem spojitého prostoru na diskrétní – dochází k zaokrouhlování a nepřesnostem. Výsledek některých operací tedy nemusí být možné přesně zapsat do naší diskrétní reprezentace, což může změnit výsledek některých operací. Např na následujícím obrázku máme úsečky A, B přesně zaznamenány v naší diskrétní reprezentaci (značena tečkami), ovšem průsečík těchto úseček leží mimo diskrétní prostor – musí dojít k zaokrouhlení. Potom však průsečík neleží přesně ani na jedné z úseček:



Jsou dva způsoby, jak tento problém řešit:

- povolit modelování prostoru pouze z určitých tvarů. Problémem tohoto řešení je, že složitější útvary by bylo nutné manuálně převést do jednodušší reprezentace
- definovat podmínky, které musí být splněny, aby prvek mohl být uložen do databáze přesně

Pro efektivní řešení tohoto problému je základem mít diskrétní prostor navržený dostatečně jemně – pak můžeme předpokládat, že i když nastane nějaká chyba/nepřesnost takového charakteru, bude dostatečně zanedbatelná/pod naši rozlišovací schopnost.

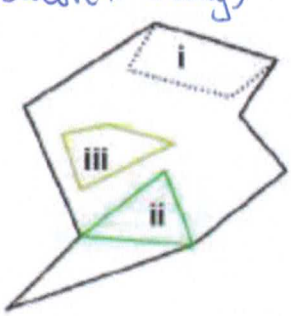
Základem mapování reality do prostorové databáze je **R-cyklus**, který slouží k uložení hranice nějaké oblasti. R-cyklus je polygon uložený v diskrétním prostoru. Polygon je sekvence úseček $s_1, s_2, \ldots, s_n$ taková, že koncový bod úsečky (segmentu) s_i je roven počátečnímu bodu úsečky $s_{(i+1) \bmod n}$ a zároveň se žádné dva segmenty nekříží. Rozlišujeme několik možných vzájemných vztahů R-cyklů:

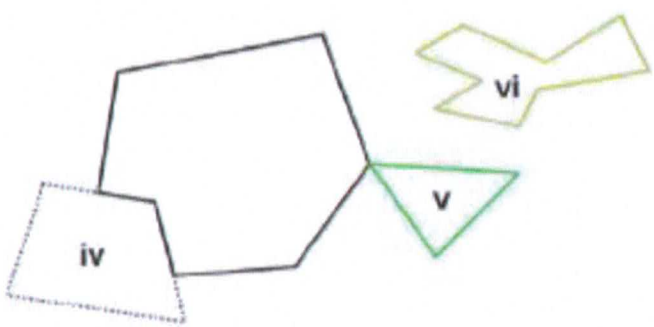
The R-cycle is

- area-nested (inside) - i, ii, iii
- edge-nested (inside) - ii, iii
- vertex/completely-nested (inside) - iii

The R-cycles are

- area-disjointed - iv, v, vi
- edge-disjointed - v, vi
- vertex/completely disjointed - vi





Dále pak definujeme R-plochu, která slouží k uchování ploch (umožňuje "vykousnout" kousek R-cyklu). R-plocha f je dvojice (c, H), kde c je R-cyklus (vnější ohraničení) a H je množina R-cyklů a platí:

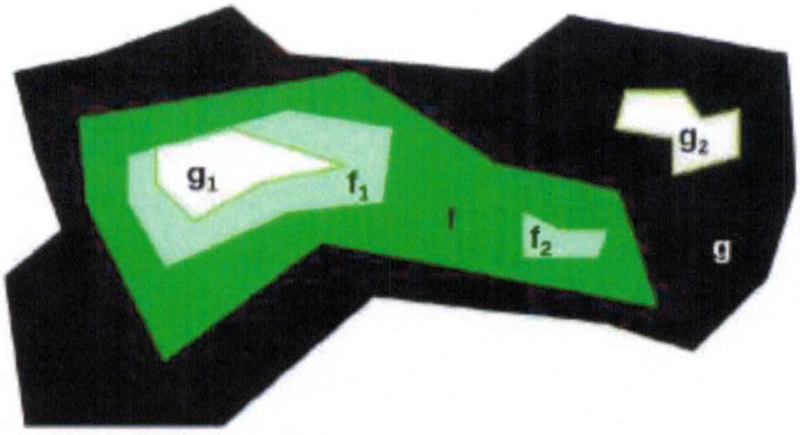
- každý R-cyklus z H je hranově vnořený do c
- každá dvojice R-cyklů z H je hranově disjunktní
- není možné vytvořit jiný cyklus ze segmentů popisující plochu f

Opět můžeme definovat vzájemný vztah R-ploch, konkrétně vzájemné vnoření dvou R-ploch.

Definition:

Let $f=(f_c,F)$ and $g=(g_c,G)$ be two R-areas. We say that f is area-nested in g if and only if:

- f_c is area-nested in g_c and
- $\forall g \in G$:
  - g is area-disjointed with f_c or
  - $\exists f \in F$: g is area-nested in f



Operace nad prostorovými daty

Je mnoho operací, které můžeme chtít vykonávat nad prostorovými daty, např.:

- výpočet charakteristik objektů, např. délka, plocha, objem, obvod, průměr – možné vypočítat už při ukládání
- výpočet těžiště/středu – možné vypočítat už při ukládání
- metriky, např. vzdálenost nebo průměr pro množinu prvků, sousedství – nelze spočítat předem
- vzájemné vztahy – rovnost, vnoření, hledání průsečíku
- tvorba nových objektů – průnik, rozdíl, konvexní obálka

Ukládání dat

Narozdíl od relační databází, kdy řádek má typicky pevnou velikost (a je tedy možné řádky jednoduše naskládat do paměti jeden za druhý) je ukládání dat v prostorových databázích složitější. Body je jednoduché uložit (fixní velikost), stejně tak obdelníky nebo úsečky (uložíme podobně jako v relační DB). Problém nastává u lomených úseček, polygonů, ploch, regionů, R-ploch a podobně – jsou velké a mají proměnlivou velikost.

Data fixní velikosti tedy ukládáme klasicky, jak bylo popsáno výše. Pro data proměnlivé velikosti (polygony, plochy, …) však uložíme pouze nějaké malé množství dat (která třeba mohou být

využita pro indexaci) a zbytek (např. jednotlivé vrcholy) jsou uloženy někde jinde a jsou dostupná přes ukazatel umístěný v hlavní části. Typicky se ukládá nějaká aproximace, např. ohraničující obdelník (minimal bounding box), společně s dalšími zajímavými informacemi – např. délka, plocha apod. Při operacích nad prostorovými daty se nejprve využije rychlý, jednoduchý výpočet nad aproximací, ve kterém mohou být užitečná i předpočítaná data. Až následně probíhá zpřesnění na základě výpočetní geometrie nad původními přesnými daty.

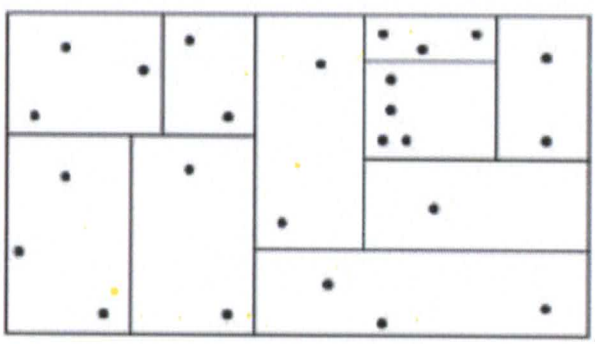
Základy indexování

Indexaci v databázi provádíme, abychom urychlili operace, které chceme provádět. V případě prostorových databází se může jednat o hledání nejbližších sousedů, výpočet průniku, výpočet vzájemné pozice apod.

1D data je jednoduché indexovat, protože je možné je přímo mapovat na celá čísla, se kterými se v počítači dobře pracuje, tzn. je možné např. hashovat nebo využít B+ strom. Nad 1D daty je možné jednoduše definovat vztah předek/následník a sousednost.

Už ve 2D toto však přestává fungovat – mohli bychom teoreticky indexovat jen podle jedné osy, ovšem to nebude fungovat dobře, protože se tímto způsobem zobrazí jako sousedé i prvky, které mohou být v reálu daleko od sebe (jejich pozice je rozdílná v druhé ose). Pro indexování v prostoru se využívají stromy nebo hashovací algoritmy (algoritmy detailně v otázce 37).

Základním přístupem indexování v prostoru je prostor rozdělit na menší díly fixní kapacity. Na základě takové reprezentace je možné i mnohem rychleji vypočítat sousedy – stačí se pro zadaný bod podívat do současného regionu a do sousedních regionů:



Pokud v textu najdete chybu, nebudete něčemu rozumět nebo budete mít dojem, že by bylo vhodné něco doplnit, kontaktujte na discordu uživatele Fifinas.

37. Indexace (nejen) v prostorových DB (kD-Tree a Grid File vč. jejich variant a R-Tree).

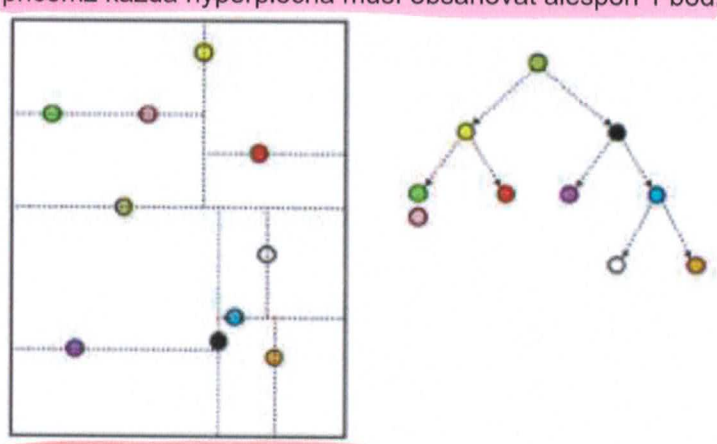
Jak bylo naznačeno v otázce 36, indexace slouží k zefektivnění dotazování nad databázemi, v případě prostorových databází nás typicky zajímají sousedé různých bodů, případně vzájemné vztahy více prostorových objektů (překryv apod.).

Indexace bodů

Základní myšlenkou indexace bodů v prostoru je dekompozice prostoru na menší úseky, dokud počet bodů v každém podprostoru neklesne pod určitou mez, kterou již považujeme za přijatelnou k efektivnímu průchodu při hledání např. sousedních bodů aj.

kD-Tree

Algoritmus spočívá v dělení prostoru hyperplochami (v 2D jsou to úsečky rovnoběžné s osami), přičemž každá hyperplocha musí obsahovat alespoň 1 bod.



Algoritmus pracuje následovně:

1. Nastav počáteční dimenzi (např. podle osy X)
2. Pokud je vstup prázdný, vrať prázdný strom
3. Seřaď body podle aktuální dimenze a vyber prostřední bod(y) a ulož je do stromového uzlu
4. Posuň se do další dimenze a
    a. Pro všechny body, které jsou menší než střední bod proveď znovu (2) a připoj výsledek jako levou větev
    b. Pro všechny body, které jsou větší než střední bod proveď znovu (2) a připoj výsledek jako pravou větev

5. Vrať vytvořený uzel

Můžeme vidět, že postupně se střídají dimenze, podle kterých probíhá půlení prostoru. Vzhledem k tomu, že prostor vždy půlíme, dosáhneme vyváženého stromu, což zajistí efektivní vyhledávání O(log n). Vyhledávání konkrétního bodu (kontrolu existence) můžeme provést následujícím algoritmem:

1. Nastav počáteční dimenzi
2. Když je strom prázdný, vrať False
3. Pokud body uložené v kořeni stromu mají stejnou současnou dimenzi jako vstupní bod, porovnej bod ve všech dimenzích s kořenem. Pokud se shoduje, vrať True, jinak False.
4. Pokud je v současné dimenzi hledaný bod menší než bod v kořeni, nastav další dimenzi a pokračuj bodem (2), ovšem zaměř se už jen na levý podstrom
5. Jinak (hledaný bod je větší než bod v kořeni) nastav další dimenzi a pokračuj bodem (2), ovšem zaměř se už jen na pravý podstrom

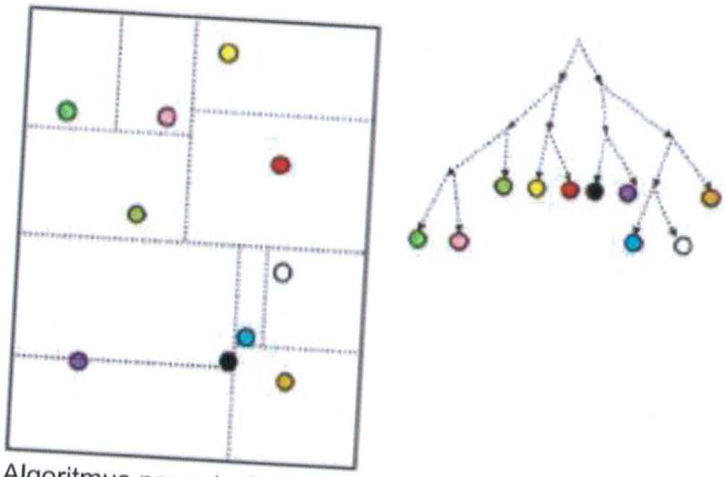
Velmi podobně pak funguje i vkládání, kdy nejprve hledáme místo ve stromě, kam nový bod máme vložit:

1. Nastav počáteční dimenzi
2. Pokud je strom prázdný, vytvoř nový uzel a ulož v něm zadaný bod, vrať True a ten uzel
3. Pokud v současné dimenzi má vkládaný bod stejnou hodnotu jako bod v kořeni, porovnej vkládaný bod s kořenovými body ve všech dimenzích. Pokud se shoduje, vrať False a kořen (uzel už je vložený), jinak přidej vkládaný bod do kořene a vrať True kořen
4. Pokud má vkládaný uzel v současné dimenzi větší hodnotu než kořenový uzel, zvol pravý podstrom, jinak zvol levý podstrom. Posuň se na další dimenzi
5. Zavolej rekurzivně (2) na vybraný podstrom v nové dimenzi:
   - pokud vrátí True + strom, nahraď příslušný podstrom navráceným stromem a vrať True + kořen
   - pokud vrátí False, vrať False + kořen

Odstranění uzlu ze stromu je poměrně problematická operace. Triviálním případem je, pokud odstraňovaný uzel je listovým, pak jej můžeme jednoduše odstranit. Pokud je vnitřním, musíme na zbylých bodech v daném podstromu vybudovat novou stromovou strukturu dle algoritmu pro tvorbu kD-stromu.

Adaptivní kD-Tree

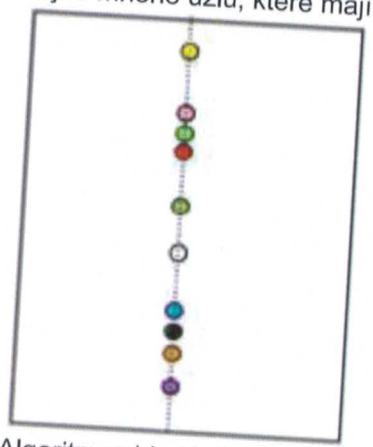
Tato varianta se snaží eliminovat problém nevyváženosti, který vzniká v závislosti na pořadí vkládání dat. Problém odstraňuje tak, že data (body) jsou rozmístěna v listech napříč celým stromem, namísto ve vnitřních uzlech stromů, jak bylo vidět u základní varianty kD-stromu. Hyperplochy (vnitřní body) dělí prostor tak, aby počet bodů na každé straně byl stejný.



Algoritmus pro vytvoření stromu:

1. Nastav počáteční dimenzi
2. Pokud je vstup prázdný, vytvoř prázdný strom
3. Pokud vstup obsahuje jeden bod, vytvoř uzlový strom, ulož do něj bod a vrať jej jako výsledek
4. Najdi minimální a maximální hodnotu vstupních bodu v současné dimenzi
5. Pokud je minimum a maximum stejné, vytvoř uzel, ulož do něj všechny body a vrať výsledek
6. Spočítej střední hodnotu mezi minimem a maximem, ulož ji jako vnitřní uzel
7. Posuň se do další dimenze a
    a. GOTO (3) pro všechny uzly, jejichž hodnota v současné dimenzi je menší nebo rovna střední hodnotě. Připoj výsledek jako levý podstrom
    b. GOTO (3) pro všechny uzly, jejichž hodnota v současné dimenzi je větší než střední hodnota. Připoj výsledek jako pravý podstrom
8. Vrať vytvořený stromový uzel

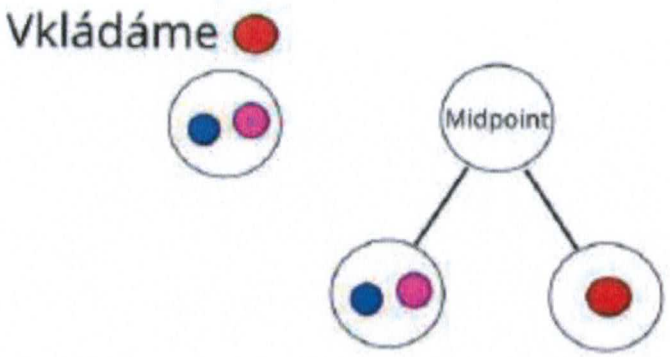
5. bod je volitelný a jedná se o optimalizaci, která se snaží řešit extrémní případy, kdy se nám sejde mnoho uzlů, které mají v jedné dimenzi stejnou hodnotu, např.:



Algoritmus hledání v adaptivním kD-stromě je velmi podobný hledání v základní variantě stromu, ovšem body jsou až v listových uzlech, tzn. porovnání ve všech dimenzích provedeme

až v listových uzlech (jinak opět jdeme doleva, když v současné dimenzi je hodnota menší nebo rovna, jinak doprava). Vkládání je také podobné, ovšem trochu složitější:

1. Nastav počáteční dimenzi
2. Pokud je strom prázdný, vrať nový listový uzel s vloženým bodem, vrať True + uzel
3. Pokud je kořen list, porovnej uložené body s vkládáným bodem na všech dimenzích, pokud se ve všech shodují, vrať False + kořen. (bod už tam máme)
4. Jinak porovnej vkládáný bod v současné dimenzi, pokud se shoduje, vlož bod do kořene a vrať False + kořen.
5. Jinak najdi střední hodnotu v současné dimenzi z uložených bodů a vkládaného bodu. Vytvoř nový uzel, nastav střední hodnotu jako jeho hodnotu. Současný kořen nastav jako patřičný podstrom vytvořeného uzlu. Vytvoř nový listový uzel s vkládaným bodem a připoj jej také jako patřičný podstrom. Vrať True + nově vytvořený uzel.



6. Rekurzivně postupuj stejně jako v neadaptivní variantě (rekurzivně se zanoř do levého nebo pravého podstromu)

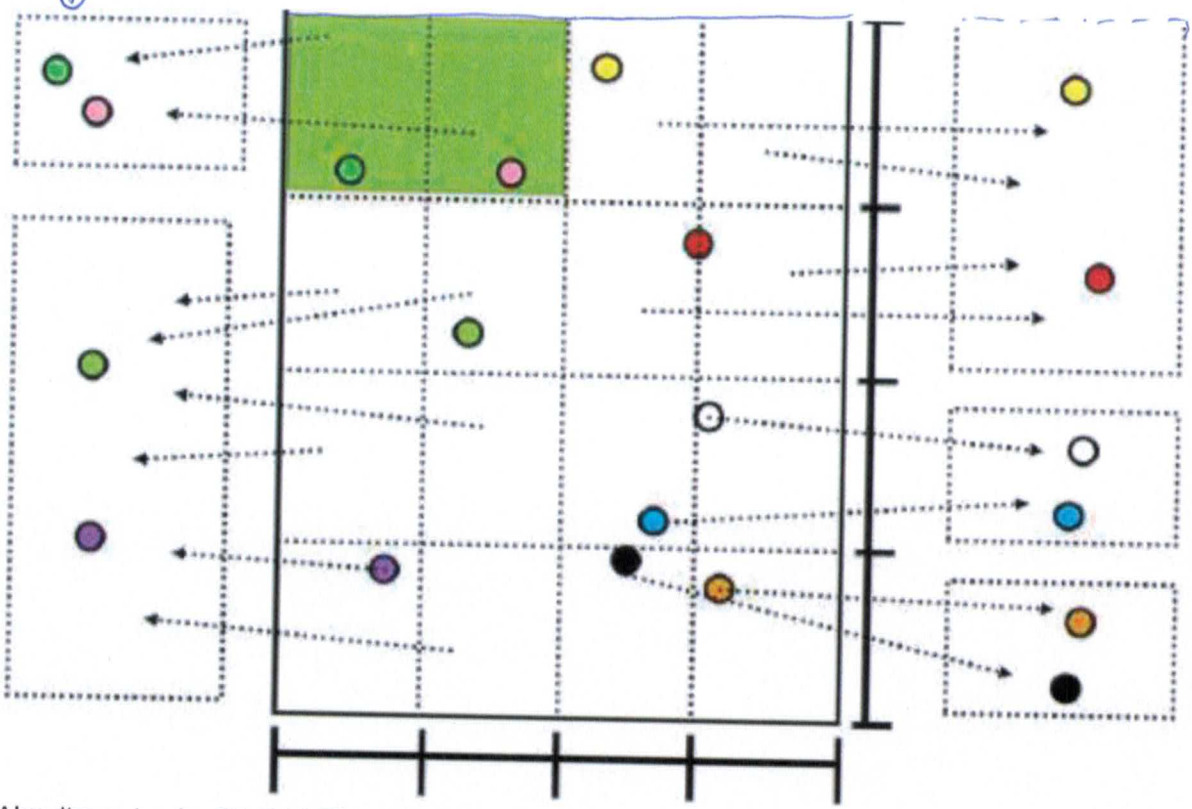
Existují další varianty kD-stromu, např. BSP-strom (hyperplochy nemusí být vodorovné s osami) nebo QuadTree (rozdělujeme prostor v každé úrovni na kvadranty).

Grid file

Struktura Grid file vychází z adaptivního hashování. Principem je, že prostor je pokryt hashovatelnou mřížkou. Vyhledávání je pak index-sekvenční – z nějakého bodu dokážeme velmi rychle najít buňku mřížky a uložiště pro danou buňku pak procházíme sekvenčně. Problém této metodě dělají shluky bodů.

Prostor dělíme obecně n-dimenzionální mřížkou, která nemusí být pravidelná. Adresář každé buňce mřížky přiřazuje datové uložiště (bucket), kde jsou uloženy body. Na obrázku můžeme vidět, že dvě zelené buňky ukazují na stejný bucket bodů. Adresář i mřížka jsou uloženy na disku – vyhledání bodu vyžaduje tedy nejvýše dva přístupy na disk

bucket



Algoritmus tvorby 2D Grid File – jeho vstupem je velikost mřížky M, N, rozsah prostoru <Xmin, Xmax> a <Ymin, Ymax> a seznam bodů:

1. Vytvoříme mřížku z M, N a rozsahu prostoru (pole/seznam)
2. Nechť B je nový bucket/datová jednotka
3. Procházej mřížku nějakým vhodným způsobem
    a. Přidávej body do B, dokud B nezaplníš
    b. Nahraď B novým bucketem
    c. Pokračuj prohledávání, dokud nejsou projity všechny buňky mřížky
4. Vrať mřížku a buckety

Hledání v mřížce pak probíhá velmi jednoduše:

1. Na základě souřadnice hledaného bodu urči buňku mřížky
2. Načti odpovídající bucket z mřížky
3. Projdi sekvenčně body v bucketu, pokud je tam shodný bod, vrať True, jinak False.

Vkládání:

1. Na základě souřadnice vkládáného bodu určit buňku mřížky
2. Načti odpovídající bucket z mřížky
3. Projdi seznam bodů v bucketu, pokud se tam bod už nachází, vrať False
4. Pokud má bucket volné místo, vlož do něj bod a vrať True
5. Jinak pokud se dají buňky ještě rozdělit (každá buňka jde rozdělit maximálně jednou – je nutné rozdělit "celý řádek"):

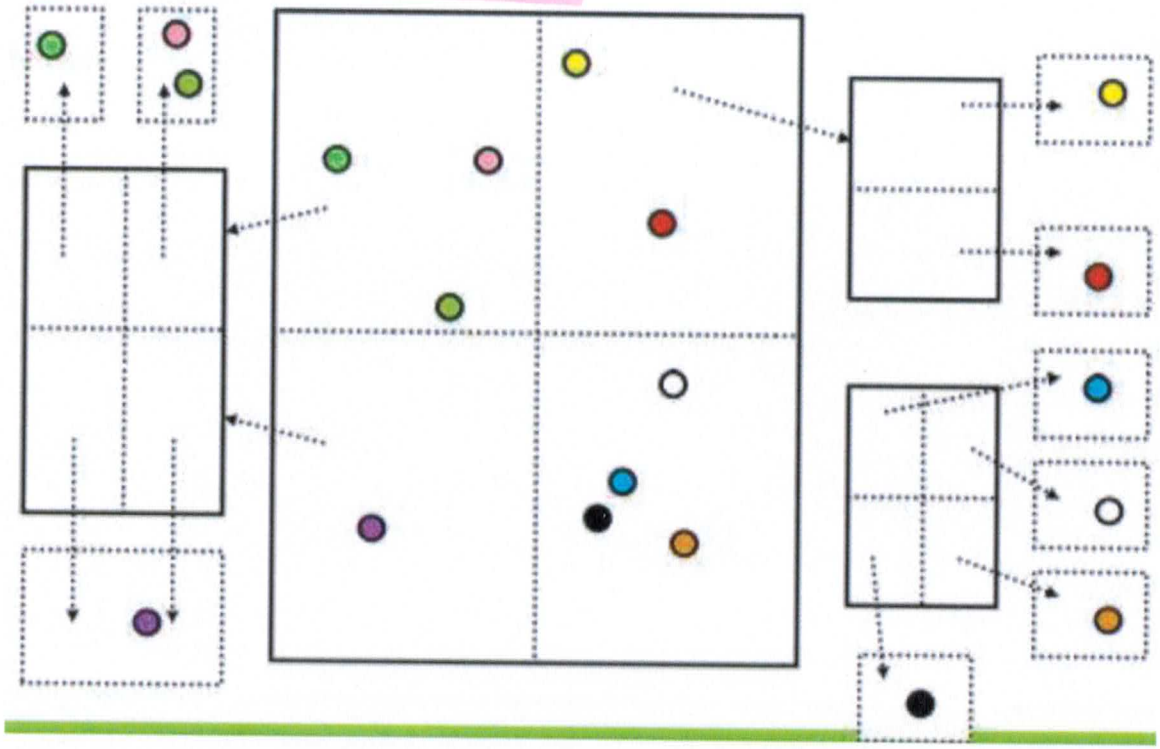
a. Vytvoř nový bucket
b. Rozděl buňky
c. Vlož bod do příslušného bucketu a vrať True

6. Fail (nedostatek místa)

Two-level grid file

- výhoda: rozsahy jsou většinou lokální (na 2. úrovni)

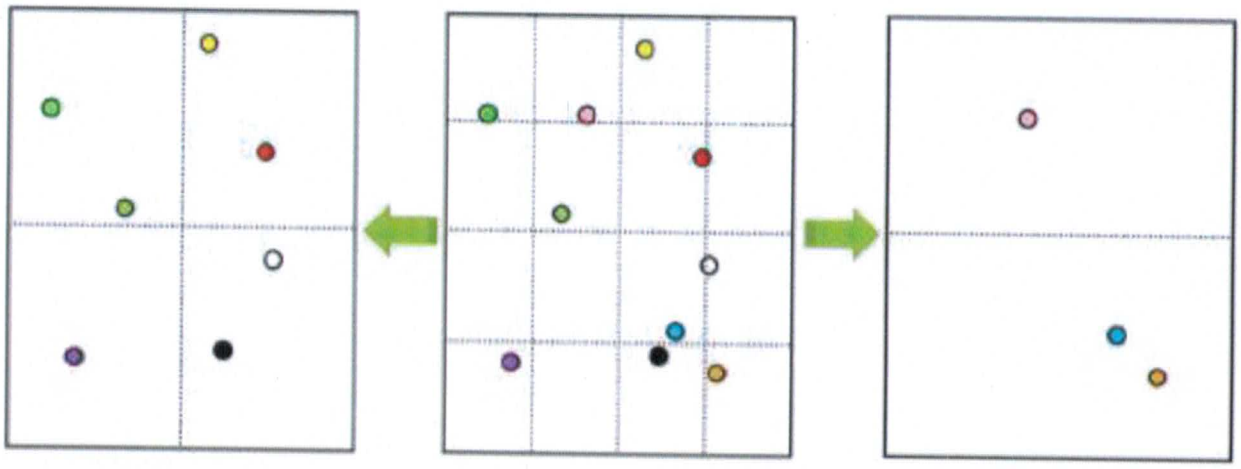
Zavádí dvě úrovně mřížek, buňky v kořenové mřížce odkazují na podaresář (jiná mřížka). Buňky v mřížce druhé úrovně už odkazují na bucket.



Twin grid file

Zdvojení mřížky, nevzniká však hierarchie. Na počátku jsou body dle nějakého kritéria rozděleny mezi mřížky. Jedna mřížka je brána jako primární, druhá jako mřížka pro případ přetečení (když dojde místo). Dosahuje lepšího využití prostoru než začne selhávat na nedostatek místa. Umožňuje navíc rozdělit shluky mezi mřížky, což pomůže algoritmu vyhledávání.

- jde paralelní vyhledávání v obou grid filech



Indexace ploch

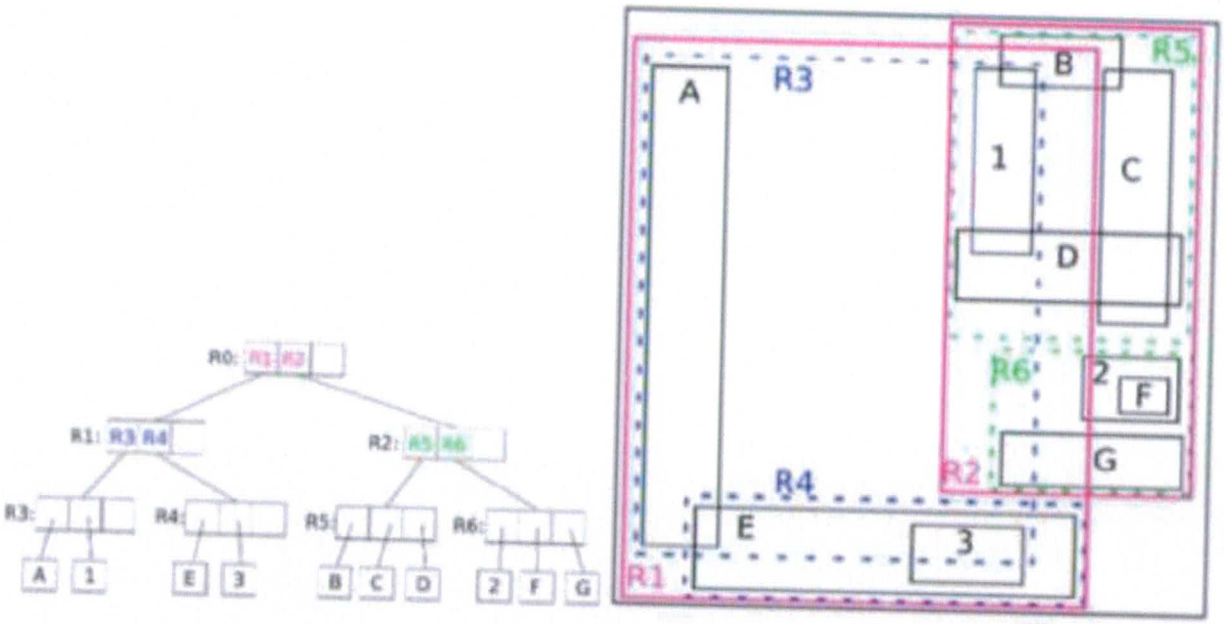
Indexace ploch je principielně složitější než indexace bodů. Namísto obecný ploch se však budeme zaměřovat pouze na indexaci obdelníků – pro složitější útvary spočítáme tzv. Minimal Bounding Box, což je obdelník, který plochu zcela obaluje a je kolmý/rovnoběžný na osy.

Základními principy používanými k indexaci ploch jsou:

- transformace – mapování objektů z jednoho prostoru do jiného. Např. obdelník ve 2D můžeme reprezentovat 2 body, což můžeme brát jako bod ve 4D prostoru (nefunguje příliš dobře)
- překryv – indexovací struktury se mohou překrývat (na tomto principu staví R-Tree)
- ořezání – objekt může být duplikován ve více indexovacích strukturách

R-Tree

Struktura inspirovaná B-stromem, tzn. je to vyvážený strom a každý uzel má pevný počet následníků. Jak bylo naznačeno výše, staví na tom, že se indexační struktury mohou překrývat, což efektivně znamená, že existuje více vyhledávacích cest, které může být nutné prozkoumat. Jednotlivé uzly ve stromě odpovídají obalujícím obdelníkům, které obalují své potomky (jedná se o tzv. minimum bounding rectangle):



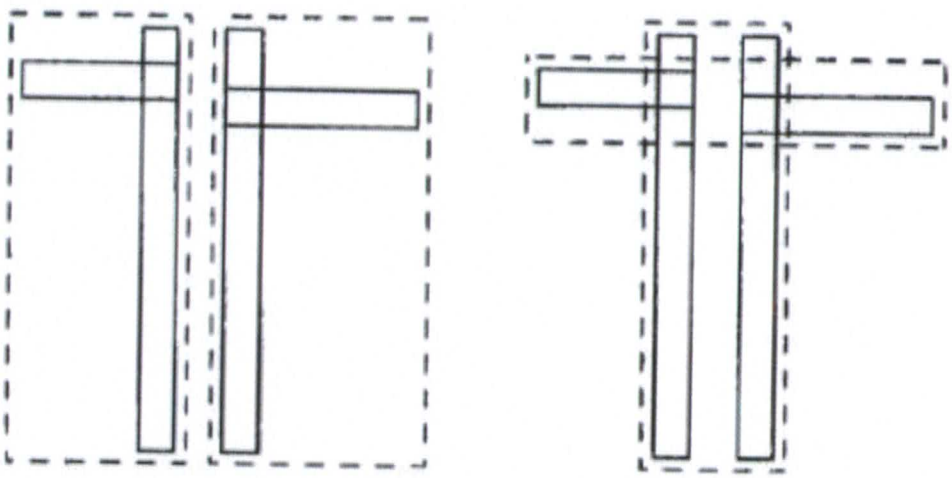
Cílem vyhledávání je najít všechny objekty, které se překrývají se zadaným obdelníkem:

1. Pokud kořen není listový, zkontroluj všechny záznamy v kořeni, jestli se překrývají s hledaným obdelníkem. Pro všechny překrývající se obdelníky rekurzivně zavolej vyhledávání
2. Pokud kořen je list, zkontroluj všechny záznamy, jestli se překrývají s hledaným obdelníkem. Pokud ano, překrývající se záznam vrať

Vkládání je složitější – po vložení je nutné zvětšit rodičovský obdelník tak, aby obaloval i nově vložený obdelník:

1. Zavolej ChooseLeaf (viz dále) pro výběr uzel, kam se má obdelník vložit
2. Pokud zvolený uzel má místo, vlož do něj vkládaný obdelník, jinak zavolej SplitNode pro obdržení částí L a LL
3. Zavolej AdjustTree na L
4. Pokud se rozdělil kořen, vytvoř nový kořen, jehož děti jsou dva vytvořené uzly.

ChooseLeaf – hledáme takový listový uzel, který způsobí nejmenší zvětšení, abychom zahrnuli vkládaný obdelník
AdjustTree – upraví bounding rectangle tak, aby co nejtěsněji obklopoval všechny obdelníky, co jsou následníci ve stromě
SplitNode – snaží se minimalizovat prázdný prostor:



Bad split **Good split**

- naivně bychom museli vyzkoušet všechny možné kombinace rozdělení, kterých je extrémně mnoho (exponenciálně), proto se využívá "heuristika", tzv. algoritmus Quadratic split nebo Linear split:
  a. Vybereme dva počáteční obdelníky, tzv. semínka (algoritmus PickSeeds), každé je přiřazeno do vlastní skupiny.
  b. Pokud všechny prvky byly přiřazeny, stop. Pokud jedna skupina má tak málo prvků, že všechny zbývající obdelníky musí být přiřazeny do ní, aby bylo dosaženo minimální zaplnění B stromu, přiřaď je a konči.
  c. Vyber další obdelník (algoritmus PickNext) a přiřaď jej do skupiny, která bude muset být nejméně zvětšena pro přidání tohoto obdelníku. Remízu řeš přidáním obdelníku do skupiny s menší plochou. Pak pokračuj na bod (b).

Algoritmy PickSeeds a PickNext závisí na "heuristice", kterou využíváme, zda Quadratic nebo Linear split, např. Quadratic split vybírá tak, aby se maximalizovala plocha mezi počátečními obdelníky (naopak Linear vybere obdelníky, které jsou nejdále od sebe):

Quadratic split:

PickSeeds:

Input: set of entries where seeds are to be found
Output: pair of entries composing seeds

1. For each pair of entries E_1 and E_2, compose a rectangle J including E_1I and E_2I. Calculate d = area(J) – area(E_1I) – area(E_2I).
2. Choose the pair with the largest d.

~~PickNext~~:

PickNext:

Input: Two groups of entries, set of entries not being in any of the two group
Output: Entry to be added in a next step to a group

1. For each entry E not yet in a group, calculate d_1 = the area increase required in the covering rectangle of Group 1 to include EI. Calculate d_2 similarly for Group 2.
2. Chose an entry with the maximum difference between d_1 and d_2.

Linear split: $O(n)$

PickSeeds: ① V každé z dimenzí najdi dva od sebe nejvzdálenější obdélníky
② Normalizuj tyto vzdálenosti šířkou dané dimenze
③ Vyber pár s největším normal. vzdálen. vyjdeš. dim

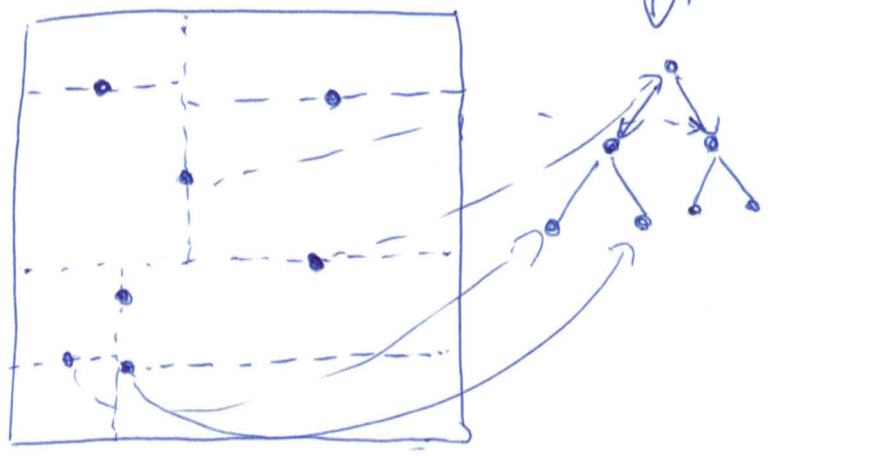
PickNext: ① Vyber náhodně obdélník

13/14

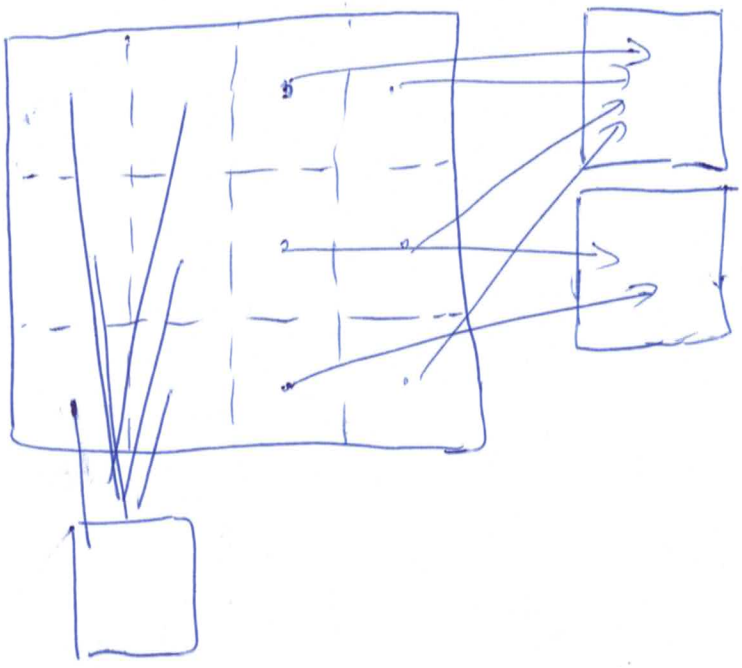
Odmazání staví na podobném principu, ovšem po odstranění zkusíme smrsknout strom operací CondenseTree. Pokud na konci má kořen pouze jednoho následníka, kořen odmažeme a onoho následníka uděláme kořenem.

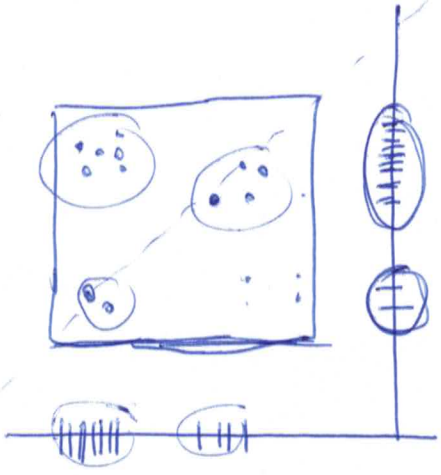
Pokud v textu najdete chybu, nebudete něčemu rozumět nebo budete mít dojem, že by bylo vhodné něco doplnit, kontaktujte na discordu uživatele Fifinas.

kD - Tree
- indexace bodů



Grid File - indexace bodů





- nelze efektivně mapovat do 1D
tak, aby byla zachována sousednost
bodů, čehož většinou využívají
B+ stromy k indexaci