## Úvod

V dnešnej dobe plnej informácií a miliárd rôznych dát je čoraz dôležitejšie efektívne vyhľadávanie. Nato aby sme mohli v týchto dátach rýchlo a pohodlne vyhľadávať je potrebné ich správne indexovať. Indexovanie dát sa používa najme pri tvorbe rozsiahlych produktových katalógov a zoznamov, kde je dôraz kladený na efektivitu vyhľadávania a rýchly prístup k požadovanej informácií. Na ilustráciu si predstavíme situáciu kde sa chystáme vytvoriť rozsiahly katalóg áut pričom chceme aby rádovo obsahoval databázu niekoľko tisíc až milión kusov áut. Nato, aby sme v tomto katalógu dokázali rýchlo vyhľadávať, musíme tieto dáta indexovať, lebo v opačnom prípade by sme zakaždým museli prejsť celý katalóg čo pri viacerých kritériách vyhľadávania predsa nemôže byť efektívne. Nato aby sme tieto požiadavky splnili je nutné správne reprezentovanie dát a použitie vhodnej štruktúry ktorá tieto dáta zindexuje.

V súčasnosti poznáme viacero postupov a štruktúr ktoré spĺňajú tieto kritéria, ale v tejto práci sa spolu zaoberáme novým prístupom k tomuto cieľu a to využitím kombinácia priestorového a hashovacieho indexu, čiže **Grid indexu.** Motiváciou výberu tejto práce bol fakt, že ide o súčasnú problematiku ktorá poskytuje priestor na nové pohľady a riešenia. Ďalší faktor ktorý ovplyvnil môj výber je vývojový jazyk Java v ktorom sa práca implementovala. Výhodou aj nevýhodou mojej témy je fakt, že indexácia objektov ešte nebola týmto spôsobom riešená, čiže k tomuto modelu indexu sme nenašli s vedúcim žiadne vedecké články ani výskumy. Pri práci som využil poznatky z iných modelov indexov., najmä pri práci so súborovou štruktúrou. Táto práca mi ponúkla dostatočný priestor na kreativitu pri návrhu riešenia štruktúry indexu. Budeme venovať len formálnej stránke veci čiže nebudeme prácu zameriavať na konkrétny účel.

# Popis problému

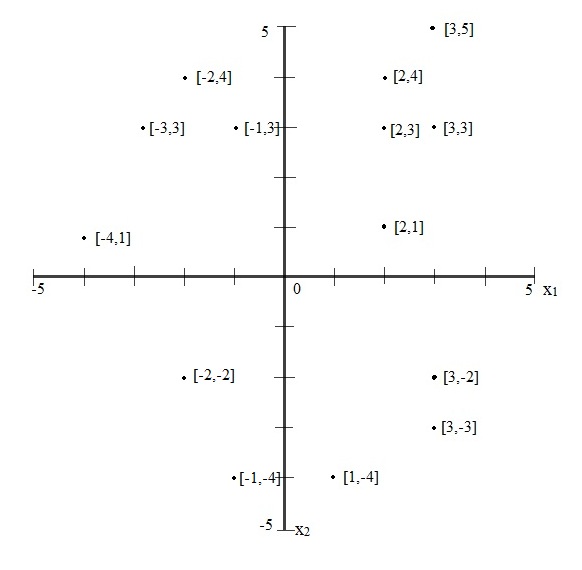
V tejto bakalárskej práci sa ideme pokúsiť o vytvorenie novej štruktúry pomocou ktorej budeme vedieť indexovať a spätne vyhľadávať priestorové n – rozmerné body do súboru. Keďže budeme pracovať s priestorovými bodmi, majú tieto body svoje pevné miesto v priestore dané ich súradnicami. Z geometrického hľadiska potrebujeme rozdeliť oblasť n – rozmerného priestoru v ktorom sa nachádzajú body. Priestor rozdelíme v každom rozmere nadrovinami, so všeobecným predpisom pričom n je počet dimenzii. Nadroviny sú navzájom rovnobežné a kolmé na os tej dimenzie, podľa ktorej priestor delíme. V danej dimenzií je nadrovina definovaná práve jednou súradnicou.

Počet nadrovín ktorými budeme deliť ako aj ich vzdialenosť od seba závisí na konkrétnych bodoch a bude počítaná našim algoritmom. Našou snahou je rozdeliť tento priestor tak, aby sme dosiahli čo najrovnomernejšie rozdelenie bodov do jednotlivých častí priestoru ktoré vzniknú naším delením. Aby sme dosiahli najlepšie riešenie, požadujeme konečný počet bodov.

Následne každej jednej časti priestoru, ktorý sme dostali delením priestoru, prislúcha jedna hash hodnota. Pre každú takúto časť priestoru bez ohľadu na jej veľkosť máme v súbore pripravenú stránku s pevnou veľkosťou, jednotnou pre všetky stránky. Práve hash hodnota danej časti priestoru je kľúčom k ofsetu jej stránky v súbore.

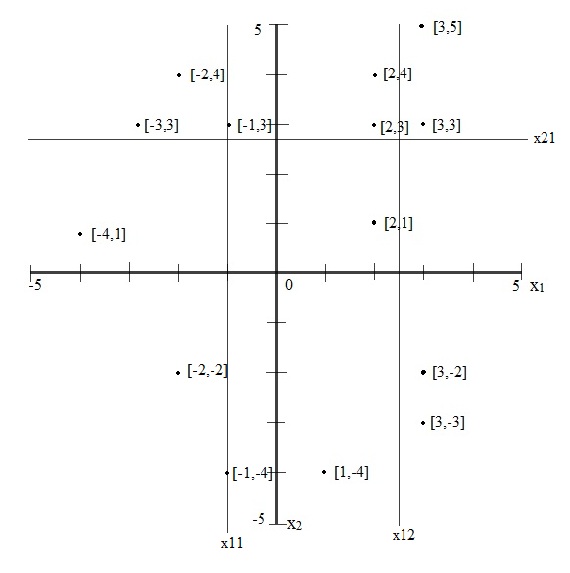
### Ilustračný príklad

Na nasledujúcom príklade si ukážeme ako rozdelíme body v dvoj – rozmernom priestore. Nadrovina dvoj – rozmerného priestoru je priamka ktorej predpis je **.** Na **Obr. 1** sú uvedené body v súradnicovom systéme [x1,x2].



**Obrázok 1.** Body v dvoj – rozmernom priestore pred rozdelením

Následne sa snažíme body čo najrovnomernejšie rozdeliť. Ukážeme si jedno z možných rozdelení a popíšeme ho z geometrického hľadiska. Toto rozdelenie je formálne a nemusí to byť najlepšie rozdelenie.



**Obrázok 2.** Body v dvoj – rozmernom priestore po rozdelením priamkami na podpriestory

Priestor sme rozdelili tromi priamkami, dvomi v prvej dimenzii a jednou v druhej. Po tomto rozdelení nám vzniklo šesť podpriestorov. Podľa našej definície nadroviny majú priamky predpis :

* Dimenzia 1 s osou x1 : x1,1 = -1, x1,2 = 3
* Dimenzia 2 s osou x2 : x2,1 = 3

### Ciele práce

Aby sme mohli úspešne navrhnúť efektívne fungujúci grid index, potrebujeme sa oboznámiť s problematikou algoritmov slúžiacich na indexovanie n – rozmerných dát. Preto sa v prvom rade pozrieme na už existujúce indexovacie štruktúry a na ich algoritmy aby sme pochopili princípy pri indexovaní n – rozmerných objektov. Aj na základe týchto algoritmov budeme schopný navrhnúť náš algoritmus na vytvorenie Grid indexu.

V druhej časti tejto práce budeme implementovať navrhnutý algoritmus do jazyka Java. Poskytneme rozhranie na vytvorenie indexu, pridávanie ďalších bodov do súboru pomocou indexu a vyhľadávanie nad indexom pomocou rozsahového dopytu. Po dokončení implementácie budú nasledovať testy tohto indexu na generovaných dátach, ktoré budú mať rôzny charakter. Náš index budeme porovnávať s R\* stromom, ktorý je predstaviteľom dynamického indexu. Následné tieto nazbierané výsledky vyhodnotíme. Zapísaním do niekoľkých bodov môžeme ciele zhrnúť nasledovne:

* Preskúmať oblasť indexácie n – rozmerných dát do súborovej štruktúre.
* Navrhnúť a implementovať fungujúci prototyp grid indexu.
* Implementovať vyhľadávací rozsahový dopyt.
* Vykonať porovnávacie testy s R\* stromom na viacerých typoch vstupných dát.

### Prehľad súčasného stavu

Existuje množstvo rôznych, prevažne stromových indexovacích štruktúr. Indexovacie štruktúry môžeme rozdeliť na *dynamické* a *statické*. Statické štruktúry

sú navrhnuté pre jednorazové naplnenie množinou bodov bez možnosti ich ďalšieho

pridávania či odoberania. Dynamické indexovacie štruktúry poskytujú možnosť

kedykoľvek pridávať nové body (prípadne aj odoberať existujúce body) bez narušenia

akékoľvek vlastnosti danej štruktúry.

Stromové indexovacie štruktúry môžeme rozdeliť na *vyvážené* a *nevyvážené*.

Vyvážený strom má bez ohľadu na dáta v ňom uložené vždy všetky vetvy rovnako dlhé (t.j. všetky listy stromu sú na rovnakej úrovni). U takýchto stromov je za každých okolností zaručené, že výška stromu stúpa s počtom jeho bodov logaritmicky. Nemôže sa teda v dôsledku špecificky zvolených dát stať, že sa strom vybuduje nevhodne (zopár krátkych vetiev a jedna veľmi dlhá vetva a pod.) a tým výrazne klesne efektivita pri vyhľadávaní nad takýmto stromom

## Grid index

Grid index je indexovacia štruktúra navrhnutá pre indexovanie n – rozmerných bodov. To znamená, že grid index je možné použiť na indexovanie tých objektov, ktoré je možné reprezentovať ako body n – rozmerného priestoru. Ak chceme na indexovanie bodov použiť grid index, stačí aby sme poznali konečný počet bodov pred začiatkom indexovania. Grid index je statická štruktúra, čo znamená, že je navrhnutá pre jednorazové naplnenie množinou bodov. Pri dodatočnom pridávaní ďalších bodov sa jeho vnútorná štruktúra dynamický nemení a preto pri doindexovaní väčšieho počtu bodov, môže dochádzať k poklesu jeho efektivity, čo sa môže prejaviť pri vyhľadávaní nad ním. Grid index sa najprv musí kompletne vytvoriť, až potom sa doň pridávajú body, čo znamená odlišní prístup ako pri budovaní stromov, kde sa strom buduje počas pridávania bodov postupne.

**Definícia 2.1 G**rid indexom pre n - rozmerne dáta rozumieme štruktúru ktorá obsahuje:

1. V každom z *n* rozmerov niekoľko deliacich nadrovín kolmých na os tohto rozmeru.
2. Usporiadanú n – ticu *H* , pričom každé *Hi*  ,kde *i = 1, ... ,n* vyjadruje počet časti priestoru po rozdelení v *i* – tej dimenzii
3. Usporiadanú n – ticu *N*, pričom *Ni ,*kde *i = 1, .. , n* je n – tica *M*, kde *Mj*, z rozsahu *j = 1, ... ,Hi* vyjadruje súradnicu nadroviny *j* v *i –* tom rozmere, čiže nadrovina   
   *xi,i = Ni,j*.

Na základe tejto definície dokážeme vytvoriť grid index nad ľubovoľnými n – rozmernými dátami. Na výrobu grid indexu majú vplyv aj vstupné parametre indexu, ktoré si dôkladné popíšeme pri algoritme výroby indexu.

**Oknom** vzhľadom na grid index nazývame časť priestoru ktorá je v každom smere ohraničená najbližšou nadrovinou. **Vonkajším oknom** vzhľadom na grid index budeme nazývať časť priestoru, ktorá je ohraničená aspoň jednou nadrovinou. Počet okien a vonkajších okien je rovnaký ako počet vzniknutých časti priestoru. V práci ak nebude napísané, že hovoríme o vonkajšom okne, budeme **okno** aj **vonkajšie okno** nazývať spoločne **okno.** Každý bod ktorý do indexu pridáme patrí na základe svojich súradníc práve do jedného okna. Z definície vyplýva, že dva body s rovnakými súradnicami budú patriť vždy do rovnakého okna. Grafické znázornenie grid indexu len s **vonkajšími oknami** sa nachádza na **Obr. 2.**

**Pásom** vzhľadom na grid index budeme nazývať časť priestoru v jednej dimenzií ohraničenú jednou alebo dvoma nadrovinami.

Všetky indexované body v grid indexe sú uložené v jeho **oknách**. Grid index poskytuje uloženie dát z okien na pevný disk, čiže všetky body z okien sú uložené na pevnom disku a vo forme stránok súboru tvoria elementárne celky ktoré z disku čítame (alebo zapisujeme na disk). Jedným prístupom na disk rozumieme prečítanie jednej alebo viacerých stránok ktoré tvoria okno z disku (alebo ich zapísanie na disk).

### Vytváranie grid indexu

Na rozdiel od budovania stromových štruktúr ako B – stromy, kde sa strom buduje postupným pridávaním bodov, sa grid index vybuduje na základe všetkých bodov ktoré chceme indexovať. Základnou myšlienkou pri vytváraní grid indexu je snažiť sa čo najlepšie, najrovnomernejšie zabezpečiť rozdelenie bodov do jednotlivých okien indexu. V našej práci sme použili prístup, kde berieme do úvahy počet rôznych hodnôt v jednotlivých rozmeroch bodov. Vychádzame z predpokladu, že v dimenzií kde je viac rôznych hodnôt je potrebné aj jemnejšie rozdelenie danej dimenzie nadrovinami, ako v dimenzií, kde je menší počet rôznych hodnôt. Pseudo algoritmus vytvárania grid indexu *make\_index(store, pageSize, n)*:

*make\_index(store, pageSize, n):*

**Vstup**: *store* = List všetkých bodov, *pageSize* = veľkosť stránky, *n* = počet rozmerov

**Výstup:** Grid index

*štruktúra* = prázdny list zoznamov súradníc nadrovín

*delenia* = prázdny list delení dimenzií

1. *k* = zrátame veľkosť jedného bodu v jednotke ako *pageSize*
2. *m* = *pageSize/k*
3. *počet\_okien* = (počet bodov / *m)* \* 1,3
4. *priemerny\_počet\_deleni* =
5. *počet\_rôznych* = podľa dimenzií usporiadaný zoznam počtu rôznych hodnôt v dimenzií
6. pre každé *i* = 0 do *n* *deleniai* = *priemerny\_počet\_deleni*
7. pokiaľ
   1. *dimenzia* = index( maximum (*počet\_rôznych*))
   2. *deleniadimenzia* += 1
   3. *počet\_rôznychdimenzia* = 0
8. pre každú dimenziu *i*
   1. *suradnice* = prázdny list
   2. *počet\_bodov\_medzi* = počet bodov / *deleniai*
   3. zotrieď body v dimenzii *i*
   4. pre každé *h* = 1 do *deleniai*
      * 1. *jeden* = *store(h \* počet\_bodov\_medzi)*
        2. *druhy* = *store(h \* počet\_bodov\_medzi + 1)*
        3. pridaj (*jeden + druhy*)/2 do *suradnice*
   5. pridaj *suradnice* do *štruktúra*

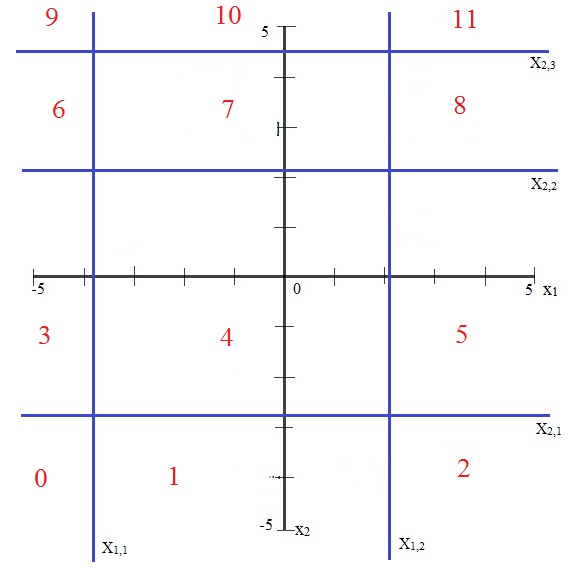
Teraz si podrobnejšie popíšeme uvedení pseudokód. Najprv si popíšeme vstupy a výstupy algoritmu. Na vstupe dostávame zoznam všetkých bodov ktoré chceme indexovať, počet rozmerov indexu a veľkosť jednej stránky v súbore, ktorá je kľúčová pre rozdelenie priestoru. Táto hodnota má vplyv na celkový počet **okien** v indexe. Na výstupe dostávame štruktúru grid indexu.

V kroku 1. zrátame veľkosť jedného bodu, v rovnakej jednotke akou máme zadanú veľkosť stránky. Následné v kroku 2. potrebujeme vyrátať, koľko bodov nám vojde do jednej stránky, aby sme v nasledujúcom kroku vyrátali odhadovaný počet okien v indexe. Pri výpočte *počet\_okien* bola použitá konštanta 1.3, ktorá sa používa vo viacerých výpočtoch tohto charakteru. V kroku 4. spočítavame strednú hodnotu počtu nadrovín pre každú dimenziu, ktorou inicializujeme cely zoznam *priemerny\_počet\_deleni.* Kľúčovým výpočtom na rozdelení priestoru je spočítanie počtu rôznych hodnôt v jednotlivých dimenziách. Tieto hodnoty si uchováme do premennej *počet\_rôznych* usporiadané podľa jednotlivých dimenzií. Následne pokiaľ nie je splnený podmienka v kroku 7. nájdeme dimenziu s najväčším počtom rôznych hodnôt a upravíme hodnotu ktorá zodpovedá počtu deliacich nadrovín danej dimenzie o 1. Všimnime si, že v tomto algoritme je počet deliacich nadrovín pre každú dimenziu rôzny maximálne o jednu.

Po tomto kroku dostávame presný počet deliacich nadrovín pre každú dimenziu priestoru. Posledným krokom k úspešnému vyrobeniu grid indexu je výpočet súradníc nadrovín. Čiže pre každú dimenziu nájdeme zoznam súradníc nadrovín ktoré ju budú deliť. V kroku 8. si najprv vypočítame do premennej *počet\_bodov\_medzi* koľko bodov sa bude nachádzať medzi dvoma susednými nadrovinami. Pri výpočte súradníc nadrovín využijeme utriedený zoznam bodov a priemernú hodnotu medzi dvoma susednými bodmi. Prechádzame celý utriedený zoznam bodov a vždy vyberieme bod ktorý sa nachádza na pozícii násobku *počet\_bodov\_medzi* a jeho suseda sprava. Takto dostávame všetky súradnice nadrovín dimenzie. Po spočítaní súradníc pre každú dimenziu dostávame štruktúru grid indexu, podľa jej definície.

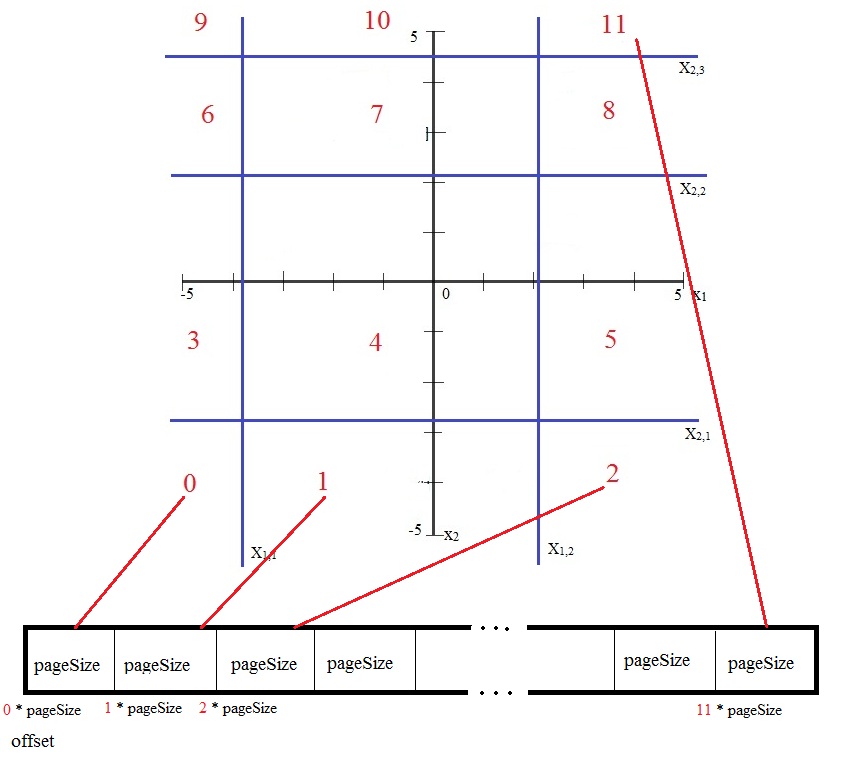
**Použitie Hash funkcie v Grid indexe**

Po algoritme vytvorenia grid indexu, si ukážeme ako funguje hashovacia funkcia v grid indexe. Po tom ako sme rozdelili priestor nadrovinami sme dostali presný počet okien v indexe. Našou úlohou je týmto oknám priradiť nejakú hash hodnotu, podľa ktorej budeme k oknám pristupovať. To je veľmi dôležité ako pri samotnom indexovaní bodov tak aj pri vyhľadávaní v grid indexe, lebo na základe tejto hodnoty pristupujeme k oknám. Každé má jedinečnú hash hodnotu z intervalu 0 až *počet\_okien,* pričom *počet\_okien* = . Okná sú číslovane postupne podľa úrovne na ktorej sa nachádzajú a podľa poradia okna v danej úrovni. Ukážka takto očíslovaného indexu v dvoj rozmernom priestore sa nachádza na **Obr. 3. Vstupom hash funkcie sú súradnice bodu, a výstup hash hodnota okna kam patrí.** Presný algoritmus hash funkcie na výpočet hash hodnoty okna na základe súradníc si popíšeme v nasledujúcej kapitole pri pridávaní bodov do indexu.



**Obrázok 3.** Ukážka očíslovania okien indexu pomocou hash funkcie v dvoj – rozmernom priestoru

Keďže k deleniu priestoru dochádza na začiatku indexácie, nemôže sa stať, že v priebehu indexovania bodov nám pribudnú nové okná a mi budeme potrebovať aj nové hash hodnoty. Takto navrhnutú hash funkciu využijeme pri mapovaní grid indexu do súborovej štruktúry. Hlavná myšlienka spočíva v samotnej hash funkcii, ktorej hodnoty tvoria kľúč k samotným stránkam súboru. To je možné len v prípade, že všetky stránky v súbore majú konštantnú veľkosť, čo pri výrobe grid indexu požadujeme tiež. K ofsetu stránky okna sa dostaneme podľa vzťahu *ofset = pageSize \* hashOkna*. Výhodou tejto hashovacej funkcie je, že potrebujeme toľko pripravených stránok v súbore, koľko máme okien v grid indexe. V praktickej časti si popíšeme aj pretekanie stánok. Ukážku takto hashovaného okna do súboru nájdeme na **Obr. 4.**



**Obrázok 4.** Ukážka hashovania okien grid indexu do súborovej štruktúry bez pretekania stránok

**Pridanie bodu do Grid indexu**

Po vytvorení samotného indexu, je potrebné všetky body, ktoré sa pri výrobe brali do výpočtov aj skutočne zindexovať. Najprv so načrtneme situáciu ktorá predchádza pridaniu prvého bodu do indexu. Priestor máme rozdelený na okná a úlohou algoritmu na pridávanie bodov do indexu je nájsť podľa súradníc bodu okno do ktorého bod priradíme. Tento algoritmus v sebe obsahuje aj hashovaciu funkciu ktorú sme si popísali v predchádzajúcej kapitole. Pseudo algoritmus pre pridanie jedného bodu do grid indexu *add\_Point(index, point).*

*add\_Point(index, point):*

**Vstup:** *index* = Grid index, *point* = bod, ktorý chceme pridať

**Výstup:** žiaden

*result* = hash hodnota okna

*poradie\_v\_dimenzii* = prázdny list poradových čísel súradníc v dimenzií

*pomocne* = prázdny list koeficientov pre každú dimenziu

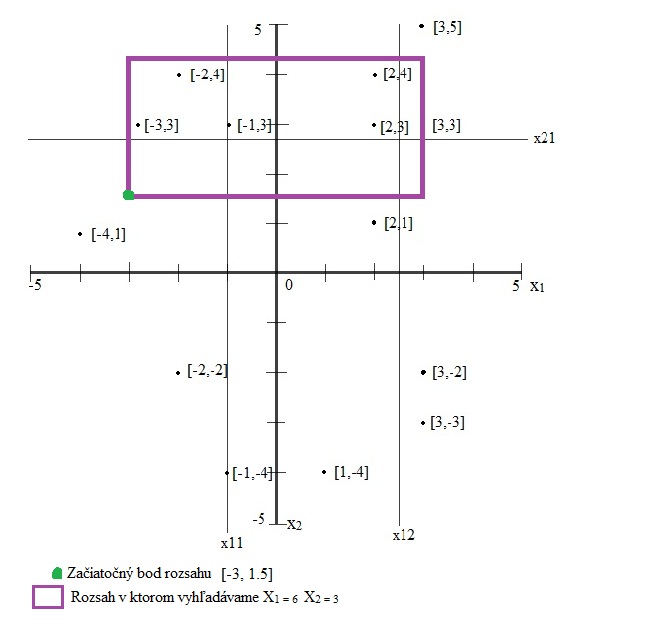
1. *pomocne0 =* 1
2. pre každé *i* = 1 do *n* 
   1. *pomocnei = pomocnei-1 \* deleniai – 1*
3. pre každé *i* = 0 do *n*
   1. pre každé *j* = *deleniai-1* do 0
      1. ak *štruktúrai,j-1* < *pointi*
         1. *poradie\_v\_dimenziii* = *j*
         2. *break*
      2. *break*
4. pre každé *i* = 0 do *n* do
   1. *result* = *result + poradie\_v\_dimenziii*\* *pomocnei*
5. pridaj *point* do *oknoresult*

Keďže tento algoritmus obsahuje v sebe aj hash funkciu, jej vstupom musí byť bod so súradnicami, no keďže náš algoritmus pracuje nad grid indexom, na vstupe musí byť aj jeho inštancia. Algoritmus nemá žiaden výstup, vo vnútri algoritmu obsiahnutý výsledok hash funkcie sa hneď použije, takže úlohou je len pridanie bodu do okna indexu kam patrí.

Najprv sa budeme zaoberať hash funkciou vo vnútri algoritmu. Potrebujeme si vypočítať pomocný zoznam *pomocne*, ktorý obsahuje počet okien na jednotlivých úrovniach okrem poslednej, čiže *pomocnei*  obsahuje počet všetkých okien na úrovni *i-1*. Táto hodnota predstavuje akúsi váhu, ktorú pridelíme jednotlivým dimenziám pri výpočte poradia okna v rámci celého indexu. V prvom kroku do *pomocne0* priradíme 1, lebo po prvej dimenzii sa posúvame o jedno okno. V ďalšom kroku dopočítame hodnoty pre zvyšné dimenzie. Ďalším krokom je zistiť pre každú dimenziu poradie najbližšej nadroviny ktorej súradnica je menšia ako súradnica bodu. To dosiahneme tak, že pre každú dimenziu prechádzame zoznamom súradníc nadrovín, od tej z najväčšou hodnotou súradnice až po tú s najmenšou hodnotou. . Ak nájdeme prvú nadrovinu ktorej súradnica je menšia, hodnotu si zapamätáme do premennej *poradie\_v\_dimenzii* a cyklus prerušíme, lebo inač by už v každej ďalšej iterácií bola podmienka splnená, a my hľadáme najbližšiu hodnotu. Ak podmienka v kroku 3 nie je splnená ani raz, znamená to, že súradnica žiadnej nadroviny nie je menšia ako súradnica bodu, a teda poradové číslo nadroviny je 0. V kroku 4 zrátame do premennej *result* výslednú hash hodnotu okna, do ktorého bod patrí. Princíp tohto výpočtu je v tom, že berieme do úvahy poradie bodu v každej dimenzií a váhu tejto dimenzie. Posledným krokom algoritmu je priradenie bodu do správneho okna v indexe.

**Rozsahový dopyt**

Rozsahový dopyt je bežná databázová operácia, ktorá vracia objekty z určeného rozsahu hodnôt medzi hornou a dolnou hranicou. Tento dopyt dopredu potrebuje vedieť presný rozsah v ktorom má vyhľadávať. Nevieme však vopred povedať, koľko objektov, ak vôbec nejaký nám tento dopyt vráti, vieme len povedať, že body výsledného zoznamu budú v požadovanom rozsahu. Ukážku rozsahového dopytu nad grid indexom nájdeme na **Obr. 5,** kde je ako vstupný parameter dopytu obdĺžnik reprezentovaný bodom, a dĺžkou strán v každom rozmere. Také isté vstupné parametre budeme používať aj v rozsahovom dopyte nad grid indexom.



**Obrázok 5.** Ukážka rozsahového dopytu nad Grid indexom

**Algoritmus rozsahového dopytu nad Grid indexom**

Podmienkou pre korektný rozsahový dopyt je zadaný nadobdlžník s rovnakým počtom rozmerov ako grid index, pripúšťame aj hodnotu dĺžky strany 0. Preto dokážeme pri vyhľadávaní niektoré súradnice hľadaných bodov nechať rovnaké s počiatočným bodom a len niektoré variabilné v istom rozsahu. Rozsahový dopyt prebieha v dvoch fázach. Teraz si napíšeme pseudokódy oboch algoritmov, ktoré si následne aj slovne popíšeme. Pseudokód rozsahového dopytu *Range\_query(point, dlzky, index)*, ktorý predstavuje prvú fázu*:*

*Range\_query(point, dlzky):*

**Vstup:** *point* = začiatočný bod nadobdlžníka, dolné hranica dopytu, *dlzky* = list dĺžok hrán nadobdlžníka, *index* = inštancia Grid indexu

**Výstup:** *result* = zoznam hľadaných bodov dopytu

*d\_h* = prázdny list dolných hraníc hodnôt

*h\_h* = prázdny list horných hraníc hodnôt

*pomocne* = prázdny list koeficientov pre každú dimenziu

*result* = prázdny list výsledných bodov

*postupnost* = pole postupnosti

1. *pomocne0 =* 1
2. pre každé *i* = 1 do *n* 
   1. *pomocnei = pomocnei-1 \* deleniai – 1*
3. pre každú súradnicu *i*
   1. pre každé *j* = *deleniai – 1* do 0
      1. ak *štruktúrai,j-1* < *pointi*
         1. *d\_hi* = *j*
         2. *break*
   2. pre každé *j* = *deleniai – 1* do 0
      1. ak *štruktúrai,j-1* < *pointi* + *dlzkyi*
         1. *h\_hi* = *j*
         2. *break*
      2. *break*
4. *generuj\_range(0, h\_h, d\_h, postupnost, pomocne, result)*
5. pre každý bod *b* v *result*
   1. *platí* = true
   2. pre každú súradnicu *i*
      1. ak *bi* >= *pointi* a zároveň *bi* <= *pointi* + *dlzkyi*
         1. *platí* = true
      2. ináč  
         1. *platí* = false
         2. break
   3. ak *platí* == false
      1. odober bod *b* z *result*
6. vráť *result* na výstupe

Teraz si popíšeme uvedený pseudokód. Na vstupe algoritmus dostane nadobdlžník ktorý predstavuje rozsah dopytu vo vhodnom tvare, čiže začiatočný bod dopytu *point* a zoznam dĺžok jeho strán *dlzky.* Na vstupe musí byť aj inštancia Grid indexu, keďže práve nad ním sa chystáme vyhľadávať. Výstupom algoritmu je očakávaný zoznam bodov ktoré obsahuje uvedený nadobdlžník.

V prvom a druhom kroku algoritmu sa zaoberáme výpočtom pomocného zoznamu *pomocne,* ktorý vypočítame rovnakým spôsobom ako sme ukázali v algoritme vloženia bodu do Grid indexu. V nasledujúcom kroku potrebujeme zistiť rozsah pásov v ktorom budeme následne generovať okná, ktoré majú spoločný prienik s nadobdlžníkom. Aby sme to dosiahli, potrebujeme prejsť cez každú dimenziu a v nej hľadáme dva pásy, prvý, ktorý obsahuje hodnotu vstupného bodu v danej dimenzií a druhý, ktorý obsahuje túto hodnotu sčítanú s dĺžkou hrany nadobdlžníka v danej dimenzií. Poradové čísla pásov obsahujúcich začiatočný bod uložíme do listu *d\_h* a tie ktoré obsahujú sčítanú hodnotu do listu *h\_h.* Popis algoritmu na hľadanie pásov bol popísaný pri vkladaní bodu do Grid indexu. Tieto dva listy sú následne vstupom do metódy *generuj\_range(n, h\_h, d\_h, postupnost, pomocne, result),* ktorú si rozoberieme neskôr. Táto metóda nám do premennej *result* vráti všetky body, ktoré obsahujú okná zasiahnuté nadobdlžníkom, avšak potrebujeme ešte skontrolovať presné hranice nadobdlžníka, lebo *result* môže obsahovať aj body ktoré patria oknu ale nepatria nadobdlžníku. V kroku päť prevádzame kontrolu hraníc každého bodu a to tým spôsobom, že v každej dimenzií skontrolujeme či je hodnota zodpovedajúca danej dimenzií väčšia ako dolná hranica nadobdlžníka *pointi*a zároveň menšia ako horná hranica nadobdlžníka *pointi + dlzkyi*. na kontrolu využívame premennú *platí*, ktorá sa postará, že v prípade situácie kde narazíme počas testovania na dimenziu v ktorej podmienka neplatí, daný bod vyhlásime premennou *platí* za nepatriaci a vyradíme ho z výsledného zoznamu *result.* Po skontrolovaní všetkých bodov na výstupe vraciame výsledok dopytu uložený v zozname *result.*

Pseudokód rekurzívneho algoritmu *generuj\_range(n, h\_h, d\_h, postupnost, pomocne, result),* ktorý predstavuje druhú fázu rozsahového dopytu a využil sa v predchádzajúcom algoritme na generovanie okien:

*generuj\_range(n, h\_h, d\_h, postupnost, pomocne, result):*

**Vstup:** *n* = pozícia ktorú generujem, *d\_h* = list dolných hraníc hodnôt, *h\_h* = list horných hraníc hodnôt, *postupnost* = pole postupnosti, *pomocne* = list koeficientov pre každú dimenziu, *result* = list výsledných bodov

**Výstup:** žiaden

1. pre každé *i* = *d\_hn* do *h\_hn* 
   1. *postupnostn* = *i*
   2. ak (*n* + 1 == počet súradníc)
      1. pre každé súradnicu *j*
         1. *hash\_okna* = *hash\_okna + postupnostj \* pomocnej*
      2. pridaj body z *oknohash\_okna* do *result*
   3. ináč
      1. *generuj\_range(++n, h\_h, d\_h, postupnost, pomocne, result)*
      2. *n* = *n­* – 1

Teraz si slovne priblížime najdôležitejšiu časť algoritmu rozsahového dopytu. Tento algoritmus je rekurzívny a generuje čísla z rozsahu horných a dolných hodnôt pre každú pozíciu výslednej postupnosti. Na vstupe teda dostávame zoznamy horných hodnôt *d\_h* a dolných hodnôt *h\_h*, zoznam *pomocne* na výpočet okna a premennú *n*, ktorá signalizuje na ktorú pozíciu postupnosti *postupnost* sa chystáme vložiť číslo. Čiže v kroku jeden nám bež cyklus v intervale hornej a dolnej hodnoty pre pozíciu *n*. V druhom kroku do postupnosti na pozíciu *n* vložíme hodnotu riadiacej premennej cyklu. Nasleduje podmienka kde testujeme či už sme dosiahli úplnú postupnosť veľkosti počtu dimenzií. V prípade, že podmienka je splnená, tak pomocou finálnej postupnosti *postupnost* a pomocného zoznamu *pomocne* vyrátame hash hodnotu okna. Z tohto okna následne do premennej *result* pridáme všetky body. Ak podmienka naopak nebola splnená, znamená to, že ešte nemáme celú postupnosť a teda rekurzívne zavoláme metódu *generuj\_range(++n, h\_h, d\_h, postupnost, pomocne, result),* pričom premennú *n* zvýšime o jeden, aby sme generovali nasledujúcu pozíciu postupnosti. Na konci ešte v nasledujúcom kroku znížime premennú *n* o jeden, aby sme vygenerovali všetky možné kombinácie. Po poslednom dobehnutí tejto metódy bude premenná *result* obsahovať všetky body zo všetkých okien ktoré majú spoločný prienik s nadobdlžníkom.

**Praktická časť – návrh programu**

Základným cieľom programátorskej časti práce je implementovať navrhnutú indexovaciu štruktúru Grid indexu so všetkými možnosťami, vlastnosťami a algoritmami popísanými v teoretickej časti. Táto kapitola popisuje návrh a implementáciu Grid indexu podľa presne definovaných požiadaviek.

**Požiadavky na aplikáciu**

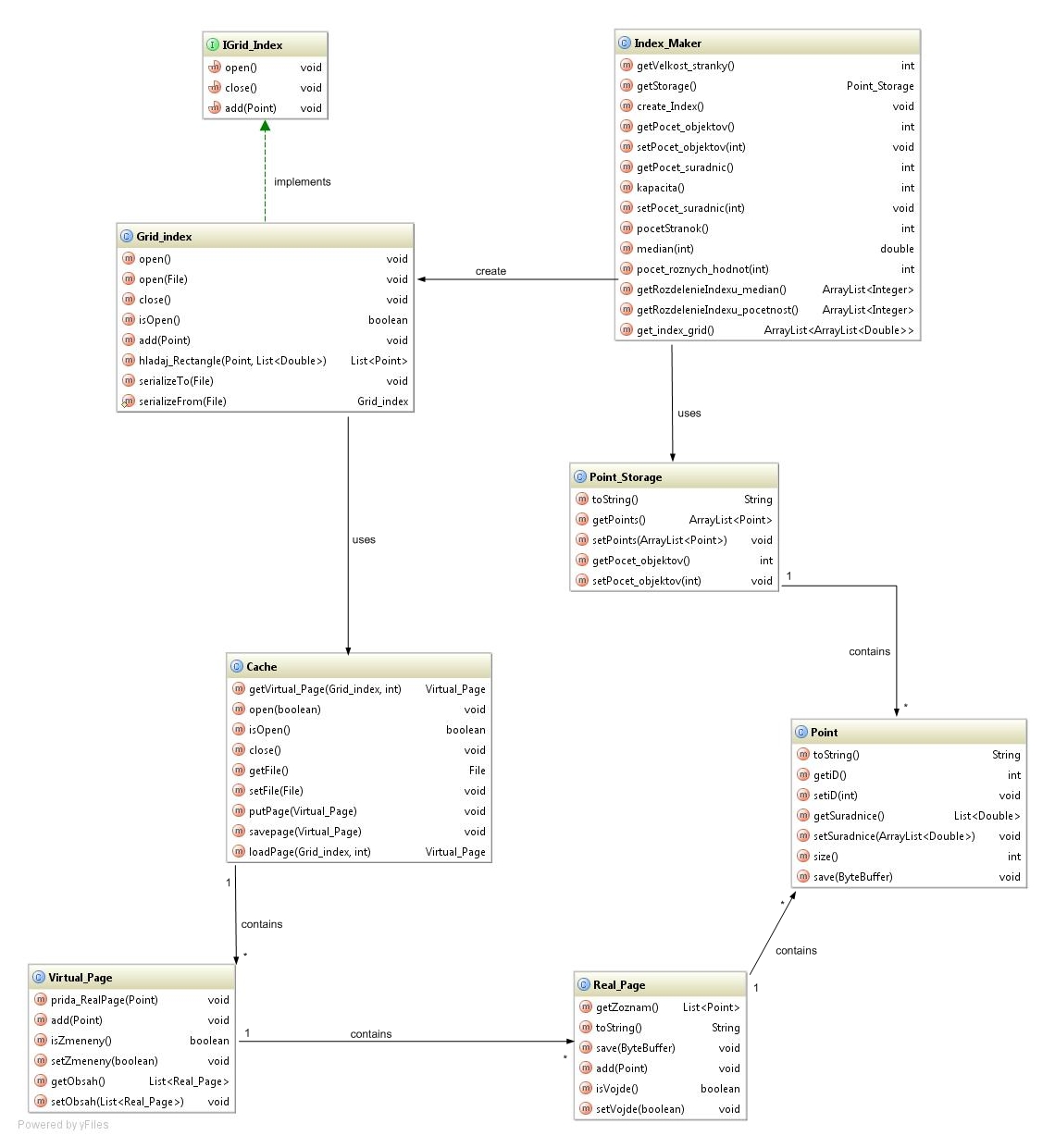
Cieľom je vytvoriť knižnicu, ktorá poskytne plnú funkcionalitu Grid indexu. Snažíme sa implementovať algoritmy uvedené v teoretickej časti čo najefektívnejšie, aby index netvoril slabý článok aplikácie ktorá ho využíva.

Požiadavky na knižnicu Grid indexu:

1. Použitie moderného programovacieho jazyka Java
2. Použitie Grid indexu pre ľubovoľné priestorové dáta
3. Uloženie okien indexu do stránok súboru
4. Požívanie cache pamäte pre naposledy prečítanie okna zo stránok, čím minimalizujeme počet prístupov na pevný disk
5. Možnosť uložiť celý hotový Grid index do súboru a tým zachovať jeho znovu použiteľnosť pri ďalšom spustení aplikácie ktorá ho využíva
6. Možnosť zadávať manuálne veľkosť cache pamäte pre naposledy prečítane okná
7. Možnosť zadávať veľkosť stránok v súbore
8. Vyhľadávanie pomocou rozsahového dopytu

**Implementácia Grid indexu**

V tejto časti si povieme základné informácie o samotnej implementácií Grid indexu podľa požiadaviek. Samotný diagram tried aplikácie nájdeme na **Obr. 6.** V diagrame sú znázornené triedy aplikácie aj vzťahy medzi samotnými triedami. Triedy popisujú len metódy, inštančné premenné sme z dôvodu prehľadnosti vynechali.

****

**Obrázok 6.** Diagram tried implementácie Grid indexu

Základnou triedou pre prácu s Grid indexom je trieda *Grid\_index*, ktorá implementuje rozhranie *IGrid\_index*. To deklaruje ktoré metódy musí inštancia *Grid\_index* poskytovať*.* Inštancia tejto triedy predstavuje samotný Grid index. Ten po jeho vytvorení dovoľuje pridávať body pomocou metódy *Add()*. Keďže Grid index ukladá body do súborových stránok, metódy *open()* a *close()* slúžia na riedenie prístupu k tomuto súboru. Metóda *hladaj\_Rectangle()* poskytuje rozsahový dopyt nad Grid indexom. Na vytvorení inštancie Grid indexu sa výrazne podieľa trieda *Index\_maker,* ktorá na základe všetkých bodov, ktoré sa chystáme pridať do indexu, vyráta celú štruktúru Grid indexu. Body n – rozmerného priestoru sú reprezentované triedou *Point*, ktorá obsahuje *id* každého bodu a zoznam súradníc. Pomocou tejto triedy dokážeme reprezentovať akékoľvek priestorové dáta.

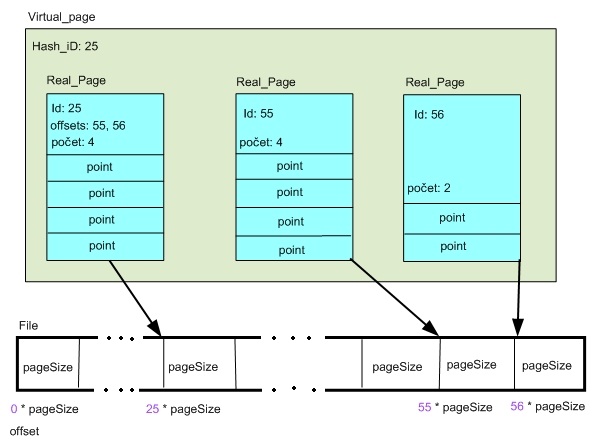
Ďalšou dôležitou požiadavkou je to, aby Grid index používal pre uloženie okien súborové stránky a neuchovával všetky dáta v primárnej pamäti. Jedným z parametrov konštruktora triedy *Grid\_index* je meno súboru, do ktorého sa majú okná ukladať, a veľkosť jednej stránky súboru v bajtoch, tá je dôležitá aj pre samotné rozdelenie indexu. Do zadaného súboru sa okná ukladajú za sebou tak, že každá stránka zaberá presne toľko bajtov, koľko bolo zadané pri vytváraní inštancie triedy *Grid\_index* (je to spoločná hodnota pre všetky stránky súboru). Pre identifikáciu jednotlivých okien indexu je použitá hash funkcia ktorej hodnota je priamo kľúčom ich ofsetu v súbore. Do súboru sa na vyhradené miesto binárne zapisujú primitívne javovské dátové typy, nie celé inštancie. Pri čítaní sa následne vytvárajú nové inštancie, ktorých atribúty sú inicializované na hodnoty prečítané zo súboru. Aby bolo možné zapísať do súboru indexované body, objekty triedy *Point* poskytujú metódu *save()*. O čítanie týchto hodnôt zo súboru sa postará trieda *Cache*, ktorú si popíšeme neskôr. Ešte dodáme, že Grid index vyžaduje fixnú kapacitu stránky, t.j. maximálny možný počet bodov, ktorý je možné uložiť do jednej stránky. Samozrejme Grid index nie je obmedzený počtom bodov v jednom okne, preto podporuje aj pretečenie stránok, ktoré si vysvetlíme pri triedach *Virtual\_page* a *Real\_page*. Na výpočet kapacity stránky potrebujeme vedieť, koľko bajtov zaberá jedna inštancia triedy *Point*. Táto trieda poskytuje aj informáciu o veľkosti jedného bodu metódou s*ize(),* kde jeden atribút *int*(to je 4B pre iD) a *n* atribútov *double*(t.j. 8B) za každú súradnicu bodu takto :

public int size(){

return 4 + 8\*suradnice.size();

}

Okná Grid indexu sú reprezentované triedou *Virtual\_page*. Jedna inštancia tejto triedy zodpovedá jednému oknu indexu. Keďže povoľujeme pretekanie stránok v súbore, tak aj tie majú svoju triedu a to *Real\_page*, kde jedna inštancia tejto triedy reprezentuje jednu stránku v súbore na pevnom disku. Preto jedna inštancia *Virtual\_page* môže obsahovať viacero inštancií *Real\_page,* ktoré obsahujú body, čiže inštancie triedy *Point,* lebo okna Grid indexu nemajú obmedzenú kapacitu. Ak zavoláme metódu pre pridanie bodu *add()* nad *Virtual\_Page,* tá zavolá metódu *add()* nad jednou *Real\_page,* ktorá ešte nemá plnú kapacitu, ak žiadna taká už nie je, tak vytvorí metódou *pridaj\_RealPage* novú inštanciu *Real\_Page*, do ktorej následné pridá bod. Taktiež aj trieda *Real\_page* poskytuje metódu *save()*, pričom sa do stránok súboru uloží informácia o počte bodov v stránke a zavolá sa metóda *save()* nad každým bodom. Na **Obr. 7** nájdeme ukážku uloženia na pretečených stránok do súboru.



**Obrázok 7.** Ukážka uloženia okna s pretečenými stránkami do súboru

Na **Obr. 7** vidíme uloženie okna Grid indexu v ktorom sa nachádza viac bodov ako je kapacita jednej stránky v súbore, číže jedna inštancia *Virtual\_page* obsahuje tri inštancie *Real\_page.* Z obrázku je viditeľné, že prvá inštancia *Real\_page* s Id = 25 má túto hodnotu zhodnú s Id hodnotou *Virtual\_page*  a teda hash hodnotou okna grid indexu. Taktiež obsahuje aj ofsety všetkých ostatných *Real\_page* inštancií. Z toho vyplýva, že nato aby sme pri čítaní zo súboru prečítali všetky stránky, musíme najprv prečítať tú, ktorá ma Id a teda aj ofset kľúč zhodný s hash hodnotou okna. Id zvyšných *Real\_page* inštancií získame z triedy *Grid\_index* ktorá obsahuje informáciu o poslednom pridelenom Id, čiže pri vytváraní inštancie sa len táto hodnota inkrementne a použije.

Pre efektívne čítanie a zapisovanie do súboru je použitý bufferovaný prístup.

Veľkosť buffera je zhodná s veľkosťou stránky súboru a jeden prístup do súboru tak vykoná prečítanie (alebo zápis) jednej stránky zo (do) súboru. Ako požitý buffer sme zvolili inštanciu triedy *ByteBuffer* z balíka *java.nio*, vďaka čomu je rýchlejší ako bežné javovské polia bajov použité pre bufferovaný prístup s využitím iba balíka *java.io*. Pre flexibilný pohyb po súbore a s tým súvisiace čítanie alebo zapisovanie potrebného počtu bajtov od požadovaného ofsetu je použitá inštancia trieda *RandomAccessFile* z balíka *java.io* s parametrom súboru s ktorým pracujeme.

Prístup k oknám a teda stránkam uloženým v súbore je centralizovaný a riadený samotnou cache pamäťou stránok. To znamená, že keď samotný Grid index potrebuje pristúpiť k svojim oknám, nemôže ho začať čítať sám z disku. Musí zavolať metódu *getVirtual\_page()* inštancie triedy *Cache* s parametrom hash hodnoty okna ktoré chce čítať a táto metóda mu vráti požadované okno priamo ako inštanciu *Virtual\_page* so všetkými *Real\_page* inštanciami. Metóda *getVirtual\_page()* po obdŕžaní požiadavky na okno s danou hash hodnotou overí, či toto okno nemá uložené v cache pamäti. Ak áno, jednoducho vráti okno ako *Virtual\_page* z pamäti a skončí. Ak nie, prečíta požadované okno z disku, vloží ho v podobe inštancie *Virtual\_page* do cache pamäti (ak jecache plná, posledné okno z cache sa vyhodí, ak bolo po čítaní z disku zmenené,zapíše ho na disk), vráti na výstup a skončí. Poradie uzlov v cache podľa toho ako boli docache pridávané zabezpečuje inštancia triedy *LinkedList* balíka *java.util,* obsahujúci hash hodnoty okien uložených v pamäti cache. Mapovanie reálnych inštancií *Virtual\_page* okien na ichhash hodnoty a rýchle vyhľadávanie okien v cache pamäti podľa hash hodnoty zabezpečuje trieda *HashMap*  balíka *java.util*.

Cache pamäť musí mať kapacitu aspoň pre jedenú stránku, vtedy sa v pamäti udržiava len aktuálne okno s ktorým sa práve pracuje. Pri požiadavke na ďalšie okno sa aktuálne okno nahradí aktuálne prečítaným oknom. Kapacita cache pamäte sa udáva

v počte okien ktoré môže naraz udržiavať. Je dôležité zvoliť vhodnú kapacitu, aby nedošlo k prekročeniu pamäte pridelenej pre JVM. Inštancie javovských tried zaberajú v operačnej pamäti viac miesta ako presne špecifikovaný počet bajtov, ktoré zaberajú okna keď sú uložené v stránkach súboru.

Jednou z požiadaviek na aplikáciu bolo, aby sa dal celý Grid index uložiť na disk a mohol byť znovu použitý pri ďalšom spustení aplikácie bez nutnosti vytvárania novej inštancie Grid indexu. Túto požiadavku môžeme graciózne splniť pomocou serializácie inštancií, ktorú java triedam poskytuje. Týmto spôsobom dokážeme jedným príkazom do súboru uložiť celú inštanciu. Aby bolo možné, týmto spôsobom inštancie zapísať na disk a znovu ich jedným príkazom prečítať, musia implementovať rozhranie *Serializable* ktoré java poskytuje. Implementácia tohto rozhrania je jednoduchá a ide len o formalitu, keďže toto rozhranie nedeklaruje žiadne metódy ktoré je potrebné prekryť. Keďže inštancia triedy Grid index obsahuje množstvo ďalších inštancií v podobe svojich atribútov, všetky ich triedy musia tiež implementovať rozhranie *Serializable*.

Pred samotným serializovaním inštancie triedy Grid index do súboru je dôležité si uvedomiť, že táto inštancia pracuje so súborom pre uloženie okien. Tento súbor pre uloženie okien indexu drží otvorený v exkluzívnom režime pre zápis pomocou inštancie triedy *RandomAccessFile*, ktorá neimplementuje rozhranie *Serializable*. Preto skôr ako inštanciu triedy *Grid\_index* zapíšeme do súboru, musíme uzavrieť jej prístup k súboru pre uloženie okien. Nato slúži metóda *close()*, ktorá uzavrie súbor pre uloženie okien a príslušný atribút pre prístup k tomu súboru t.j. inštanciu triedy *RandomAccessFile* nastaví na hodnotu *null*. Podobne pri načítaní inštancie triedy *Grid\_index* zo súboru alebo pri vytvorení novej inštancie tejto triedy, treba túto inštanciu otvoriť pomocou metódy *open()*. Táto metóda vytvorí novú inštanciu triedy *RandomAccessFile* a otvorí súbor pre uloženie okien.