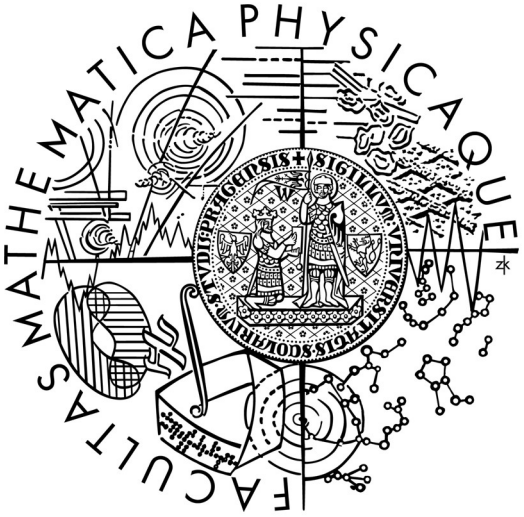
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Erik Horničák

Preferenční dotazování, indexy, optimalizace

Katedra softwarového inženýrství
Vedoucí diplomové práce: Prof. RNDr. Peter Vojtáš, DrSc.
Studijní program: Informatika, softwarové systémy

Na tomto mieste by som rád poďakoval vedúcemu diplomovej práce Prof. RNDr. Petrovi Vojtášovi, DrSc. za jeho rady a pripomienky, ktoré mi pomohli pri vytváraní tejto práce. Ďalej ďakujem rodičom, že ma podporovali po celú dobu štúdia, a všetkým priateľom, s ktorými som túto prácu konzultoval.

Prehlasujem, že som svoju diplomovú prácu napísal samostatne a výhradne s použitím citovaných prameňov. Súhlasím so zapožičiavaním práce.

V Prahe dňa 27. novembra 2007 Erik Horničák

Obsah

Název práce: Preferenční dotazování, indexy, optimalizace
Autor: Erik Horničák
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí diplomové práce: Prof. RNDr. Peter Vojtáš, DrSc.
e-mail vedoucího: Peter.Vojtas@mff.cuni.cz
Abstrakt:

Táto práca sa zaoberá vyhľadávaním k najlepších objektov z pohľadu viacerých užívateľov. Každý užívateľ má vlastné preferencie reprezentované pomocou fuzzy funkcií a agregačnej funkcie. Práca navrhuje a implementuje niekoľko riešení, pomocou ktorých je možné efektívne vyhľadávať k najlepších objektov v prípade, že hodnoty jednotlivých atribútov nie sú uložené lokálne, ale na vzdialených serveroch. Z tohto dôvodu bolo nutné prispôsobiť existujúce algoritmy na tento spôsob získavania dát. Práca využíva rôzne obmeny Faginovho algoritmu, indexáciu pomocou B+ stromov a komunikáciu pomocou webových služieb.

Klíčová slova: užívateľské preferencie, top-k, Faginov algoritmus, B+ strom, webové služby

Title: Preference querying, indexing, optimisation
Author: Erik Horničák
Department: Department of Software Engineering
Supervisor: Prof. RNDr. Peter Vojtáš, DrSc.
Supervisor's e-mail address: Peter.Vojtas@mff.cuni.cz
Abstract:
In this thesis we discuss the issue of searching the best k objects from the multi-users point of view. Every user has his own preferences, which are represented by fuzzy functions and aggregation function. This thesis designs and implements several solutions of searching the best k objects when attributes data are stored on remote servers. It was necessary to modificate existing algorithms for this type of obtaining data. This thesis uses several variants of Fagin algorithm, indexing methods using B+ trees and communication via web services.

Keywords: user preferences, top-k, Fagin algorithm, B+ tree, web services

1 Úvod

Vyhľadávanie tovaru alebo služieb na internete je v dnešnej dobe každodennou záležitosťou. Stále je však pomerne častý jav, že pri snahe užívateľa nájsť to, čo skutočne potrebuje získa buď veľké množstvo nepotrebných informácií, alebo naopak len veľmi málo vhodných výsledkov. Bežné vyhľadávače totiž nerátajú s individuálnym prístupom k jednotlivým užívateľom. Je potrebné teda zaviesť mechanizmus, ktorý zohľadňuje užívateľské preferencie, pretože každý užívateľ má spravidla odlišné nároky na jednotlivé atribúty daného produktu. Takisto sú pre niektorého užívateľa iné atribúty dôležité ako pre druhého.

Táto práca sa zaoberá vyhľadávaním najlepších *k* objektov na základe užívateľských preferencií. V súčasnej dobe existuje viacero algoritmov vhodných na takéto vyhľadávanie. Cieľom tejto práce je nájsť vhodné algoritmy, ktoré by boli schopné efektívne vyhľadávať najlepších *k* objektov, v prípade, že hodnoty jednotlivých atribútov sú uložené na vzdialených serveroch. Súčasťou práce je aj implementácia takýchto mechanizmov. Práca sa takisto podrobne zameriava na optimalizáciu vyhľadávania použitím vhodného indexovania. K tomuto účelu sú v tejto práci využívané B+ stromy.

V kapitole 2 je popísaný model užívateľských preferencií využívaný v tejto práci. Táto kapitola vysvetľuje pojmy ako napríklad fuzzy funkcia alebo agregačná funkcia.

Tretia kapitola sa venuje riešeniu top-k problému. Obsahuje detailné popisy používaných top-k algoritmov. Sú tu taktiež uvedené ich výhody a nevýhody, ako aj spôsoby použitia.

V štvrtej kapitole je popísaný spôsob implementácie vyhľadávania. Sú tu vymenované použité dátové štruktúry, top-k algoritmy, implementované indexovacie algoritmy a podobne. Nechýba detailný popis úprav, ktoré boli nutné na týchto algoritmoch vykonať, aby mohli byť použité na vyhľadávanie s využitím sieťovej komunikácie.

2 Užívateľské preferencie

Ústrednou témou tejto práce je vyhľadávanie najlepších prvkov z nejakej množiny, pričom slovo najlepší prvok je chápané ako objekt, ktorý danému užívateľovi svojimi vlastnosťami najviac vyhovuje. Z tohto dôvodu je potrebné zaviesť systém ohodnocovania jednotlivých prvkov, ktorý k danému objektu priradí hodnotu vhodnosti alebo preferenciu pre konkrétneho užívateľa. K tomuto účelu slúži takzvaná *hodnotiaca funkcia F*, ktorá každému prvku *p* s atribútmi $a_1, a_2, ..., a_m$ priradí hodnotu $F(p) \in [0, 1]$. Táto funkcia je konštruovaná tak, že najmenej vhodnému objektu priradí hodnotu 0, najvhodnejšiemu 1 a pre ľubovoľné dva prvky p_1, p_2 platí $F(p_1) > F(p_2)$ ak p_1 je lepšie ako p_2 a $F(p_1) > F(p_2)$ pre p_1 rovnako vhodné ako p_2. V tejto práci je používaný model užívateľských preferencií, v ktorom je hodnotiaca funkcia *F* rozdelená na agregačnú funkciu a fuzzy funkcie.

2.1 Fuzzy funkcie

Fuzzy funkcia je zobrazenie $f_i^U : a_i^p \rightarrow x \in [0,1]$, čiže funkcia, ktorá podľa hodnoty *i*-teho atribútu prvku *p* vypočíta jeho takzvanú fuzzy hodnotu, čo je hodnota z intervalu [0, 1], ktorá určuje, nakoľko je hodnota tohto atribútu pre daného užívateľa *U* vhodná. Podobne, ako je tomu pri celkovej hodnotiacej funkcii *F*, aj tu hodnota 1 predstavuje ideálnu hodnotu a 0 úplne nevyhovujúcu. Fuzzy funkciu možno chápať, ako nástroj, ktorým užívateľ zvolí preferované hodnoty pre každý atribút zvlášť v rámci domény jednotlivých atribútov. Keďže výsledná hodnota je z intervalu [0, 1], tak užívateľ nielenže určí, ktoré hodnoty mu vyhovujú a ktoré naopak nie, ale takisto zvolí aj mieru preferencie pre každú takúto hodnotu.

Fuzzy funkciu môžeme však chápať aj trochu inak. A síce ako usporiadanie domény atribútu. Z tohto pohľadu nie sú podstatné konkrétne hodnoty fuzzy funkcie pre dané hodnoty atribútu, ale dôležité je len porovnanie hodnôt fuzzy funkcie. Pri vyhľadávaní *k* najlepších objektov však v tejto práci bude potrebné poznať konkrétne hodnoty fuzzy funkcií, a preto sa bude preferovať prvý spôsob vnímania fuzzy funkcie.

2.2 Agregačné funkcie

Pokiaľ máme nejaký komplexný objekt skladajúci sa z niekoľkých atribútov a chceme určiť celkové hodnotenie tohto objektu, nebudú stačiť len fuzzy funkcie. Tieto totiž dokážu ohodnotiť len jednotlivé atribúty daného objektu, avšak z nich nepoznáme

jeho globálne ohodnotenie. K tomuto účelu slúži takzvaná agregačná funkcia. Táto funkcia určí z fuzzy hodnôt jednotlivých atribútov celkové ohodnotenie objektu, pričom váha jednotlivých atribútov môže byť rozdielna. Týmito váhami jednotlivých atribútov užívateľ vlastne určuje, ktorý atribút je pre neho ako dôležitý. Agregačná funkcia je v tejto práci väčšinou označovaná ako @.

Ako agregačná funkcia sa najčastejšie používa aritmetický respektíve vážený priemer, ale je vhodné použiť akýkoľvek iný priemer, maximum, minimum alebo ľubovoľný iný typ funkcie.

3 Top-k

Úlohou tejto práce je nájsť čo možno najefektívnejší top-k algoritmus vhodný pre manipuláciu z objektami umiestnenými na viacerých vzdialených serveroch. K tomuto účelu je teda nevyhnutné zadefinovať pojem top-k algoritmus. Majme množinu objektov X, kde každý objekt $x \in X$ má práve m atribútov $a_1, \ldots, a_m$. V kapitole venujúcej sa užívateľským preferenciám bol popísaný model umožňujúci ohodnotiť tieto objekty pomocou akejsi ohodnocovacej funkcie $F(x)$. Algoritmy, ktoré pre celočíselné $k > 0$ dokážu nájsť k prvkov množiny X s najväčším ohodnotením $F(x)$ sa nazývajú top-k algoritmy. Inak povedané top-k algoritmus je taký, ktorý dostane na vstupe množinu n objektov s hodnotami atribútov, ohodnocovaciu funkciu a celé číslo k a na základe týchto údajov vráti na výstupe k objektov, pre ktoré je ich ohodnotenie $F(x)$ najvyššie.

3.1 Naivný algoritmus

Najjednoduchšie, takzvané naivné riešenie tohto problému, ktorým je možné získať k najlepších objektov je ohodnotiť všetky objekty množiny X, usporiadať ich a nakoniec vrátiť na výstupe prvých k prvkov. Tento algoritmus pracuje teda nasledovne:

1. Načítajú sa hodnoty všetkých m atribútov pre každý objekt $x \in X$
2. Každému takémuto objektu $x \in X$ sa dopočíta jeho ohodnotenie $F(x)$ (toto je určite možné, keďže sú známe všetky atribúty tohto objektu).
3. Existuje teda množina X' ohodnotených objektov z množiny X. Tieto objekty budú zoradené ľubovoľným triediacim algoritmom, čím vznikne zotriedený zoznam T.
4. Na výstup bude vybraných prvých k prvkov tohto zoznamu.

Najväčší problém daného algoritmu sa dá ukázať na príklade viacužívateľského prístupu. Je totiž bežným javom, že top-k algoritmus má byť použitý na vyhľadanie k najlepších prvkov z danej nemennej množiny pre viac užívateľov, z ktorých má však typicky každý rôzne požiadavky alebo preferencie, a teda aj rôznu ohodnocovaciu funkciu $F(x)$. V takomto prípade je nutné zakaždým načítať hodnoty všetkých m atribútov a prepočítať ohodnotenie pre každý objekt $x \in X$, čo je značne neefektívne. Ako sa dá k

tomuto problému pristupovať lepšie je popísané v ďalších kapitolách.

3.2 Faginov algoritmus

Ako už bolo spomenuté, existujú top-k algoritmy, ktoré dokážu daný problém riešiť značne efektívnejšie. Asi najvýznamnejší krok v tejto oblasti bol popis takzvaného Faginovho algoritmu a jeho vylepšení TA (Treshold Algorithm) a NRA (No Random Access) v článku [1]. Hlavnou výhodou týchto algoritmov je fakt, že nepotrebujú k nájdeniu správneho výsledku načítať pre každý objekt $x \in X$ hodnoty všetkých m atribútov. Pre potreby tejto práce sa ďalej budú označovať algoritmy spomínané v článku [1] jednotne Faginov algoritmus, pričom podľa potreby budú bližšie špecifikované.

Majme množinu X mohutnosti N, ktorá obsahuje objekty s m atribútmi. Jednotlivé objekty budeme značiť x_1 , ..., x_n a ich atribúty x_i^1, ..., x_i^m. Ďalej je potrebné poznať užívateľskú agregačnú funkciu $@^U : X \rightarrow [0,1]$. Pre názornosť sa bude ohodnotenie objektu x $@^U(x^1, ..., x^m)$označovať zjednodušene len $@^U(x)$. Faginov algoritmus pracuje so zotriedenými zoznamami prvkov. Pre každý atribút existuje práve jeden takýto zoznam, čiže je potrebné mať m zotriedených zoznamov, ktoré označíme L_1, ..., L_m. Jednotlivé prvky zoznamu tvoria dvojice objektu a hodnoty konkrétneho atribútu, čiže prvky zoznamu L_i majú tvar $\{x_j, x_j^i\}$. Tieto zoznamy sú zotriedené zostupne, takže na vrchu sa budú nachádzať najvhodnejšie objekty, a naopak na spodku objekty, ktoré sú najmenej vhodné.

Faginov algoritmus kladie jednu dôležitú podmienku aj na agregačnú funkciu, a síce táto musí byť neklesajúca, čo je možné definovať nasledovne:

Funkcia f s n premennými je vzhľadom na všetky jej premenné neklesajúca práve vtedy, ak platí:

$$\forall i \in 1, ..., n : x_i \geq y_i \Rightarrow f(x_1, ..., x_n) \geq f(y_1, ..., y_n)$$

Dôsledkom tejto požiadavky pre Faginov algoritmus je fakt, že pokiaľ sa všetky atribúty nejakého objektu x_1 nachádzajú v jednotlivých zoznamoch L_1, ..., L_m nad atribútmi iného objektu x_2, tak je ohodnotenie objektu x_1 vyššie prípadne rovné ohodnoteniu objektu x_2. Toto vyplýva priamo zo zotriedenia týchto zoznamov a neklesajúcej vlastnosti ohodnocovacej funkcie.

Faginov algoritmus používa dva nezávislé prístupy k jednotlivým zoznamom:

1. *Priamy prístup* umožňuje načítanie hodnoty atribútu pre ľubovoľný objekt x, teda

získanie prvku $\{x, x^i\}$ zo zoznamu L_i, pričom vôbec nezáleží na pozícii tohto prvku v príslušnom zozname.

2. *Sekvenčný prístup* načítava hodnoty iba v poradí v akom sú uložené v zozname a to len smerom zhora nadol, čiže najprv najvhodnejšie prvky a následne tie s nižším ohodnotením.

Tieto dva typy prístupov sú na sebe nezávislé, takže ak bol nejaký prvok zoznamu načítaný priamym prístupom, tak to neznamená, že na neho algoritmus priamym prístupom nenarazí, alebo sa priamym prístupom zmení jeho pozícia v zozname.

3.2.1 Základný Faginov algoritmus

Základný Faginov algoritmus možno popísať nasledovne:

1. Paralelným sekvenčným prístupom do všetkých zoznamov $L_1, ..., L_m$ sa postupne vyberajú prvky, až kým nie je k dispozícii k objektov s hodnotami všetkých m atribútov známymi.
2. Pre každý nájdený objekt $x \in X$, ktorý nemá hodnoty všetkých m atribútov známe sa načítajú hodnoty chýbajúcich atribútov pomocou priameho prístupu.
3. Dopočíta sa ohodnotenie $@^U(x)$ pre každý nájdený objekt $x \in X$, čím vznikne množina ohodnotených objektov, ktorých počet je minimálne k. Výsledkom bude k objektov s najvyšším ohodnotením.

Korektnosť tohto algoritmu vyplýva z predpokladu, že ohodnocovacia funkcia je neklesajúca. Na základe tejto vlastnosti je totiž jasné, že po prvých dvoch krokoch algoritmu vznikne množina obsahujúca k objektov s najvyšším ohodnotením, pretože sa v nej nachádza minimálne k objektov $x_1, ..., x_k$ takých, že pre každý ďalší doteraz nenájdený objekt x_i sú hodnoty atribútov objektu x_i vo všetkých zoznamoch pod hodnotami príslušných atribútov objektov $x_1, ..., x_k$. Existencia k takýchto objektov v tejto množine je daná sekvenčným prístupom k prvkom jednotlivých zoznamov v prvom kroku algoritmu.

Pôvodný Faginov algoritmus uvedený v [1] používa pri sekvenčnom spôsobe pristupovania k prvkom paralelný prístup do všetkých zoznamov naraz. Toto je možné robiť aj inak, a síce pre každý krok algoritmu vybrať zoznamy, z ktorých sa má sekvenčne načítať prvok. Súbor pravidiel, podľa ktorých sa vyberajú zoznamy určené na sekvenčný

prístup v danom kroku sa nazýva *heuristika*. Pôvodne použitá heuristika, ktorá v každom kroku načíta paralelne prvky so všetkých zoznamov sa bude označovať *heuristika TA* (podľa TA algoritmu, ktorý bude popísaný ďalej). Kvôli názornejšiemu pohľadu na využívanie heuristík v jednotlivých algoritmoch budeme uvažovať len o heuristikách, ktoré vyberú zakaždým len jeden zoznam. Heuristika bude v tomto zjednodušení teda akési zobrazenie, ktoré zobrazí množinu krokov algoritmu do množiny $\{1, ..., m\}$. Každá heuristika sa dá pomocou rozdelenia jedného kroku algoritmu na viac krokov previesť na takúto zjednodušenú heuristiku. Z heuristiky TA by takýmto prevodom vznikla heuristika, ktorá v prvom kroku určí k sekvenčnému prístupu zoznam L_1, v druhom kroku L_2, atď. V $(m + 1)$-om kroku by bol zvolený znovu zoznam L_1. Pomocou heuristík je za určitých okolností možné efektívnejšie pristupovať k jednotlivým zoznamom, čo môže urýchliť nájdenie potrebných k najlepších objektov.

3.2.2 TA algoritmus

TA algoritmus, ktorý je tiež nazývaný Prahovým algoritmom je vylepšením základného Faginovho algoritmu. Takisto využíva sekvenčný aj priamy prístup k prvkom zotriedených zoznamov $L_1, ..., L_m$. Výhodou tohto algoritmu oproti základnému je, že spravidla vyžaduje načítať menšie množstvo prvkov a napriek tomu vráti správny výsledok.

Tento algoritmus pracuje so zoznamom T pevnej veľkosti k, takže pamäťová náročnosť je konštantná. Tento zoznam obsahuje k najlepších objektov a je zotriedený podľa ich ohodnotenia. V priebehu chodu algoritmu je nutné udržiavať špeciálny objekt, takzvaný *prah* (treshold), podľa ktorého je aj tento algoritmus pomenovaný. Prah je objekt, ktorého atribúty majú hodnoty rovné hodnotám prvkov naposledy načítaných sekvenčným prístupom z daných zoznamov. Ak teda označíme hodnotu posledne načítaného atribútu zo zoznamu L_i ako x_p^i a prahový objekt t, tak platí $t = [x_p^1, ..., x_p^m]$, pričom jednotlivé atribúty prahu typicky nepatria tomu istému reálnemu objektu, ale napríklad atribút x_p^1 by mohol byť v skutočnosti x_5^1 a x_p^2 zase x_2^2, teda prvý atribút by patril v skutočnosti prvku x_5 a druhý prvku x_2.

TA algoritmus pracuje nasledovne:

1. Vykoná sa sekvenčný prístup do zoznamu L_i, ktorý vyberie heuristika h. K takto získanému objektu x sa následne pomocou priamych prístupov do zvyšných zoznamov načítajú hodnoty všetkých m atribútov. Vypočíta sa ohodnotenie daného

objektu ako *@(x)* a následne sa tento objekt zaradí na správne miesto zoznamu *T*, ktorý je zoradený podľa ohodnotenia agregačnou funkciou. Ak v zozname *T* je po zaradení objektu *x k+1* objektov, bude zo zoznamu vyradený posledný objekt, čiže ten s najnižším ohodnotením.

2. Prepočíta sa hodnota prahu *@(t)*. Pokiaľ je ohodnotenie *k*-teho prvku zoznamu *T* väčšie ako prahová hodnota, tak už žiadny ďalší objekt nemôže mať vyššie ohodnotenie ako *k*-ty objekt z *T*, a preto algoritmus skončí, pričom jeho výstupom bude zoznam *T*.

Algoritmus vráti skutočne k najlepších objektov kvôli tomu, že agregačná funkcia je neklesajúca. Ak totiž máme nejaký doteraz nenájdený objekt *x*, je jasné, že pre každý jeho atribút platí $x_i \leq x_p^i = t_i$, a teda z neklesajúcej vlastnosti agregačnej funkcie aj $@(x) \leq @(t)$. Označme *k*-ty prvok zoznamu *T* ako x_k. Potom podľa ukončovacej podmienky algoritmu musí platiť $@(x) \leq @(t) \leq @(x_k)$ pre všetky zatiaľ nenájdené objekty *x*. Ďalej je ešte nutné overiť, či v prvom kroku nemôžeme vylúčiť zo zoznamu objekt, ktorý by mohol patriť medzi *k* najlepších. Tento fakt vyplýva z toho, že ak niektorý objekt zo zoznamu *T* vyhadzujeme, poznáme už všetky jeho atribúty a navyše je jasné, že v tomto zozname existuje práve *k* objektov, ktoré majú ohodnotenie vyššie ako tento objekt, keďže je vždy vylúčený len *k+1*-vý objekt a zoznam *T* je zoradený podľa ohodnotení jednotlivých objektov.

3.2.3 NRA algoritmus

NRA algoritmus (No Random Access), ako už názov napovedá, pracuje bez použitia priameho prístupu. Táto vlastnosť je dosť výhodná, pretože vo väčšine používaných dátových štruktúrach je priamy prístup časovo podstatne náročnejší ako sekvenčný. Keďže však priamy prístup nie je možný, tak je algoritmus nútený pracovať s objektami, ktoré nemajú všetky atribúty známe. U takýchto objektov však nie je možné vypočítať ich ohodnotenie. Z tohto dôvodu sa v NRA algoritme zavádzajú ďalšie dve ohodnotenia. *Najhoršie možné ohodnotenie* objektu *x*, ktoré sa značí *w(x)* sa vypočíta tak, že sa v agregačnej funkcii namiesto hodnôt atribútov, ktoré nie sú zatiaľ známe dosadí najhoršia možná hodnota, teda 0. *Najlepšie možné ohodnotenie* objektu *x* značené *b(x)* by sa mohlo podobne vypočítať dosadením najlepšej možnej hodnoty, čo je 1, avšak existuje ešte striktnejšie kritérium, ktoré takisto ohraničí ohodnotenie objektu zhora. V tomto

algoritme sa totiž rovnako ako v TA algoritme udržiava hodnota t, čiže prah. Keďže všetky hodnoty atribútov, ktoré zatiaľ algoritmus pomocou sekvenčného prístupu nezískal musia byť zákonite nižšie ako príslušné hodnoty atribútov prahu, môžeme pri výpočte najlepšieho možného ohodnotenia objektu x dosadiť za chýbajúce atribúty hodnoty prahu. Nech teda x' je nekompletný objekt a $V(x')=\{i_1,\dots,i_n\}\subseteq\{1,\dots,m\}$ je množina indexov jeho atribútov so známymi hodnotami. Ďalej majme objekt $b_{x'}$, pre ktorý platí, že $b_{x'}^i = x'^i$ ak $i\in V(x')$ a $b_{x'}^i = t^i$ v opačnom prípade, kde t^i je i-ty atribút prahu t. Podobne objekt $w_{x'}^i = x'^i$ pre všetky $i\in V(x)$ a $w_{x'}^i = 0$ pre ostatné i. Potom $b(x') = @(b_{x'})$ a $w(x') = @(w_{x'})$. Ako vyplýva už z definície, je najhoršie možné ohodnotenie objektu vždy menšie alebo rovné ako ohodnotenie tohto objektu pri všetkých známych hodnotách atribútov a takisto aj najlepšie možné hodnotenie je vždy vyššie alebo rovné ako výsledné ohodnotenie. Ak sú hodnoty všetkých m atribútov objektu x známe, tak platí, že $b_x = w_x$ a teda aj $b(x) = w(x) = @(x)$.

NRA algoritmus používa rovnako ako TA algoritmus zoznam T pre k aktuálne najlepších prvkov. Keďže v tomto prípade nemusia byť známe hodnoty všetkých atribútov jednotlivých objektov, nie je zoznam T zoradený podľa ohodnotenia, ale len podľa najhoršieho možného ohodnotenia. Navyše ak má dva alebo viac prvkov zoznamu T rovnaké najhoršie možné ohodnotenie, tak sú ďalej zotriedené podľa najlepšieho možného ohodnotenia. Oproti algoritmu TA je však naviac potrebný zoznam kandidátov, ktorý označme C a ktorý obsahuje objekty momentálne sa nenachádzajúce medzi k najlepšími (nie sú teda v zozname T), ale napriek tomu majú šancu sa počas chodu programu medzi k najlepších dostať. Pre potreby tohto algoritmu sa bude najhoršia možná hodnota k-teho prvku zoznamu T značiť w_k a jeho najlepšia možná hodnota ako b_k.

Algoritmus NRA možno popísať nasledovne:

1. Pomocou heuristiky sa vyberie zoznam L_i a sekvenčným prístupom sa z neho získa prvok $\{x_j, x_j^i\}$. Ak sa objekt x_j už nachádza v zozname T alebo C, priradí sa mu novonačítaný atribút x_j^i. Prepočíta sa $w(x_j)$ a $b(x_j)$.
   - Ak je hodnota $w(x_j)$ väčšia ako w_k, tak: Pokiaľ sa nachádza objekt x_j v zozname T, tak sa presunie na správne miesto tohto zoznamu. Ak sa tento objekt vyskytuje v zozname C, tak bude z tohto zoznamu vyradený a pridaný do zoznamu T. K-ty prvok zoznamu T bude následne z neho odobratý a pokiaľ bude jeho najlepšia hodnota väčšia ako nová hodnota w_k, tak tento objekt

zaradíme do zoznamu C.

- Ak je hodnota $b(x_j) > w_k$, a x_j ešte nie je v zozname C, tak sa do neho pridá.
- Ak $b(x_j) \leq w_k$ a x_j sa nachádza v C, tak sa z neho odstráni.

2. Prejde sa celý zoznam C a pre každý jeho prvok x sa dopočíta hodnota $b(x)$ podľa aktuálneho prahu. Ak je pre niektoré takéto x $b(x) \leq w_k$, tak bude tento objekt odstránený z C. Pokiaľ ja v zozname T aspoň k objektov a zoznam C je prázdny, algoritmus skončí a výsledkom bude zoznam T. V opačnom prípade chod algoritmus pokračuje opäť v bode 1.

Algoritmus je korektný, pretože keď skončí, tak žiadny objekt, na ktorý zatiaľ pri sekvenčnom prístupe nenarazil nemôže mať najlepšiu možnú hodnotu vyššiu ako najhoršia možná hodnota posledného objektu v zozname T a zároveň žiadny objekt vylúčený zo zoznamu C sa už z rovnakého dôvodu nemôže dostať do výsledného zoznamu T. Formálnejší dôkaz korektnosti tohto algoritmu je uvedený v článku [1].

Tento algoritmus má jednu drobnú nevýhodu, a síce fakt, že výsledný zoznam spravidla obsahuje objekty, ktoré nemajú hodnoty všetkých atribútov známe. Toto je spôsobené tým, že NRA nepoužíva priamy prístup, takže nemôže jednoducho získať chýbajúce hodnoty atribútov. Je to však možné vyriešiť tým, že nakoniec, keď už algoritmus má uzavretý zoznam T (keď sa už do neho nebudú prijímať nové objekty), algoritmus sekvenčným prístupom bude prechádzať zoznamy $L_1, \ldots, L_m$ a len v nich vyhľadávať príslušné hodnoty, až kým nebudú všetky objekty kompletné.

3.2.4 3P-NRA algoritmus

Algoritmus 3P-NRA (3-phased no random access) je vylepšením algoritmu NRA a nie je obsiahnutý v článku [1] ako predchádzajúce algoritmy, ale je prevzatý z článku [2]. Algoritmus NRA má totiž jednu výraznú slabinu. Musí totiž prerátavať najlepšie možné hodnoty pre celý zoznam C, zakaždým, keď sa zmení hodnota prahu, čo je prakticky po každom sekvenčnom prístupe. Keďže zoznam C nemá na rozdiel od zoznamu T pevnú veľkosť, mohol by za určitých podmienok značne narásť, čo by následne zvýšilo celkovú časovú náročnosť algoritmu. 3P-NRA používa namiesto dvoch zoznamov T a C len jediný zoznam T, ktorý je vlastne ich zlúčením a je zoradený podľa najhoršej možnej hodnoty. Problém s častým prepočítavaním $b(x)$ pre celý zoznam je tu vyriešený rozdelením

algoritmu na tri fázy, pričom k prepočítaniu najlepšej možnej hodnoty dochádza len vo fáze číslo 2, do ktorej sa algoritmus snaží dostať čo možno najzriedkavejšie. Ďalšou veľkou výhodou tohto algoritmu je použitie dvoch rôznych heuristík h_1 a h_2 vo fázach 1 a 3, čo umožňuje vhodným výberom týchto heuristík rýchlejšie nájdenie najlepších k-objektov.

Popis 3P-NRA algoritmu:

Fáza I:

1. Pomocou heuristiky h_1 sa vyberie zoznam L_i a z neho sa sekvenčným prístupom získa objekt x.
2. Pre objekt x sa vypočítajú jeho hodnoty $b(x)$ a $w(x)$ a následne sa tento objekt zaradí na správne miesto v zozname T.
3. Ak je hodnota $b(x)$ menšia ako w_k, čiže najhoršia možná hodnota k-teho prvku zoznamu T, tak bude tento prvok zo zoznamu T odstránený.
4. Ak je hodnota $@(t) < w_k$ a súčasne zoznam T obsahuje aspoň k objektov, algoritmus prejde na fázu II, v opačnom prípade sa vráti na prvý bod fázy I.

Fáza II:

1. Pre každý objekt x_j zo zoznamu T pre $j > k$ prepočíta algoritmus $b(x_j)$.
2. Ak je $b(x_j) < w_k$, objekt x_j bude vyhodený zo zoznamu T.
3. Ak zostalo v zozname T viac ako k objektov, algoritmus pokračuje vo fáze III, v opačnom prípade skončí a vráti ako výsledok zoznam T.

Fáza III:

1. Pomocou heuristiky h_2 sa vyberie zoznam L_i a z neho sa sekvenčným prístupom získa objekt x.
2. Ak sa objekt x nachádza v zozname T, prepočítajú sa jeho hodnoty $w(x)$ a $b(x)$ a zaradí sa na správne miesto zoznamu T. Ak $b(x) < w_k$, tak bude vyradený zo zoznamu T.
3. Ak je počet objektov zoznamu T rovný k, ukončí sa výpočet a výsledkom bude zoznam T.
4. Ak sa presunom prvku x v zozname T zmenila hodnota $w(x)$ alebo ak klesla

hodnota *@(t)* a zároveň sa už zopakovalo *b* cyklov fázy III bez toho, aby algoritmus vošiel do fázy II, tak algoritmus skočí naspäť do fázy II. V opačnom prípade výpočet pokračuje v kroku 1 fáze III.

Vo fáze III bol pri rozhodovaní vo štvrtom kroku použitý parameter *b*. *b je* ľubovoľné kladné celé číslo a je to vlastne parameter 3P-NRA algoritmu, ktorý určuje, ako často sa vykonáva fáza II. Táto fáza sa môže vykonať po každom cykle fázy III (ak *b* je rovné 1) alebo sa nemusí vykonať ani raz (ak b je väčšie ako celkový počet cyklov fázy III potrebných na získanie k najlepších prvkov), pretože v tomto prípade bude objekt, ktorý by bol vyradený v tejto fáze odstránený vo fáze III. Nájdenie vhodného parametra *b* pre tento algoritmus je možné len experimentálne, pretože jeho vhodnosť do značnej miery závisí od vstupných dát.

Tento algoritmus je len obmenou algoritmu NRA a dôkaz, že nájde korektne *k* najlepších objektov je uvedený v článku [2].

Rovnako ako je tomu v algoritme NRA, ani tento algoritmus nemusí vrátiť zoznam objektov zo všetkými atribútami známymi. Aby sa tohto docielilo, je potrebné takisto ako v prípade NRA doplniť ďalšiu fázu, v ktorej sa sekvenčným prístupom načítajú chýbajúce hodnoty atribútov. Pri tomto sekvenčnom prístupe je vhodné použiť rozdielnu heuristiku, ktorá by vynechávala zoznamy z ktorých nie je potrebné načítať žiadnu hodnotu. Rýchlosť výpočtu v rámci tejto štvrtej fázy je možné ďalej zvýšiť nenačítavaním hodnôt atribútov zo zoznamov, kde už bola objavená nulová hodnota, pretože tieto atribúty budú vďaka zotriedeniu zoznamov nutne takisto nulové.

4 Implementácia

Cieľom tejto práce je implementovať top-k algoritmy a pomocné mechanizmy, ktoré by boli vhodné na vyhľadávanie *k* najlepších objektov podľa užívateľských preferencií. Táto práca sa konkrétne zaoberá situáciou, kde jednotlivé hodnoty atribútov sú uložené na vzdialených serveroch a je preto nutné zvolené algoritmy optimalizovať pre sieťovú komunikáciu. Navyše je potrebné zvoliť vhodné dátové štruktúry a algoritmy, ktoré majú na starosti indexovanie dát také, aby boli čo možno najlepšie adaptovateľné na sieťovú komunikáciu.

Celá aplikácia bola implementovaná v jazyku Java a bola navrhnutá tak, aby bolo možné čo najjednoduchšie zmeniť jej súčasti, napríklad dodatočne implementovať ďalší top-k algoritmus, nové heuristiky alebo nové typy agregačných funkcií.

4.1 Rozdelenie na klientskú a serverovú časť

Keďže hodnoty atribútov sú uložené na vzdialených serveroch, bolo nevyhnutné zvoliť architektúru klient-server.

4.2 Fuzzy funkcie a agregačná funkcia

Fuzzy funkcie sú reprezentované triedou *FuzzyFunction*. Keďže je implementačne veľmi náročné počítať so všeobecnou fuzzy funkciou, tak táto práca používa len zjednodušenú podobu. Každá fuzzy funkcia je tu popísaná len ako zoznam bodov, ktoré sú navzájom spojené úsečkami. Táto reprezentácia je plne postačujúca, pretože každá spojitá funkcia sa dá na danom intervale dostatočne aproximovať pre účely vyhľadávania *k* najlepších prvkov. Navyše fuzzy funkcie typicky nezvyknú byť natoľko zložité, aby to takúto aproximáciu sťažilo. Trieda *FuzzyFunction* obsahuje teda len zoznam bodov, ktoré predstavujú akési zlomy tejto funkcie. Táto trieda ďalej implementuje metódu *getFunctionValue()*, ktorá k normalizovanej hodnote atribútu dopočíta jeho fuzzy hodnotu.

Všeobecná agregačná funkcia je reprezentovaná abstraktnou triedou *AgregationFunction*. Všetky konkrétne agregačné funkcie musia byť potomkami tejto triedy. Táto abstraktná trieda prikazuje potomkom implementovať len jedinú metódu a to *calculate()*, ktorá dostane ako parameter fuzzy hodnoty jednotlivých atribútov a vráti hodnotu agregačnej funkcie pre tieto atribúty, čiže vlastne celkové ohodnotenie celého objektu.

4.3 Klient

Kostru klientskej aplikácie tvorí akýsi vyhľadávací mechanizmus nazývaný *Query Engine* (QE). QE spúšťa vyhľadávanie najlepších k objektov pomocou nastaveného top-k algoritmu.

4.3.1 Použité top-k algoritmy

V tejto práci boli zvolené dva top-k algoritmy, ktoré sa zdali byť pre daný problém najvhodnejšie. Ako zástupca algoritmov, ktoré používajú priamy prístup bol vybratý algoritmus TA. Ďalej bol zvolený ešte jeden algoritmus, ktorý nevyužíva priamy prístup, ale funguje výlučne na sekvenčnom prístupe a to konkrétne 3P-NRA. Oba algoritmy sú podrobnejšie popísané v kapitole 3. Všetky algoritmy musia byť potomkami abstraktnej triedy *TopKAlgorithm*, ktorá prikazuje implementovať metódu *findTopK()*, slúžiacu na vyhľadanie *k* najlepších objektov. Táto abstraktná trieda taktiež implementuje niekoľko tried a metód, ktoré môžu byť pre jednotlivé algoritmy užitočné.

4.3.2 3P-NRA

Najdôležitejšou úlohou pri navrhovaní implementácie jednotlivých algoritmov bola čo najlepšia voľba vhodných dátových štruktúr. Pre zoznam *T* sa zvolila reprezentácia pomocou tried *TreeSet* a *HashMap*, ktoré sú súčasťou jazyka Java. Trieda *TreeSet* umožňuje totiž triedenie jednotlivých prvkov aj pomocou vlastných triediacich kritérií a udržiava ich v potrebnom poradí. Trieda *HashMap* zase podporuje rýchly prístup k jednotlivým prvkom pomocou jednoznačného identifikátora. Spolu teda máme nástroj, s ktorým môžeme efektívne udržiavať usporiadanie tohto zoznamu a zároveň je možné rýchlo pristupovať k jednotlivým objektom.

Ďalej je v tomto algoritme používaný vektor zotriedených zoznamov $L_1, \ldots, L_m$. Tie sú reprezentované triedou *AttributeCache*, ktorá je popísaná neskôr. Aj keď je v tejto triede ukrytá ďalšia netriviálna logika, pre algoritmus sa javí ako dátová štruktúra, ktorá dokáže vrátiť nasledujúci prvok obsahujúci identifikátor objektu a hľadanú hodnotu atribútu.

Do implementácie tohto algoritmu boli pridané ešte pomocné štruktúry, ktoré slúžia na urýchlenie výpočtu. Jedná sa napríklad o zoznam už odstránených objektov, ktorý v prípade, že algoritmus sekvenčným prístupom získa hodnotu atribútu zo zoznamu predtým odstráneného, umožňuje tento prvok ignorovať a pokračovať vo výpočte.

Bola implementovaná aj štvrtá fáza, ktorá slúži na načítanie chýbajúcich hodnôt

atribútov, pretože pri algoritme 3P-NRA sa spravidla stane, že po skončení algoritmu a nájdení najlepších *k* objektov nie sú všetky tieto objekty kompletné, teda nemajú známe hodnoty všetkých atribútov, čo má za následok to, že ich ohodnotenie nie je presné. Na urýchlenie tejto fázy slúži špeciálna heuristika, ktorá prechádza len zoznamy, v ktorých sa nachádzajú chýbajúce hodnoty a ukončí výpočet, keď sú hodnoty všetkých atribútov známe.

4.3.3 TA algoritmus

TA algoritmus narozdiel od 3P-NRA potrebuje využívať aj priamy prístup, a preto musí komunikovať so serverom, ktorý podporuje indexovanie umožňujúce tento druh prístupu. Na prvý pohľad by sa mohlo zdať, že tento algoritmus je pre sieťovú komunikáciu úplne nevyhovujúci, kvôli faktu, že pre každý prvok získaný sekvenčným prístupom je potrebné načítať z každého zoznamu priamym prístupom príslušný atribút. Keďže sa však hodnota každého atribútu nachádza na inom serveri, bolo by nutné kvôli každému jednému objektu načítanému sekvenčne použiť *m-1* volaní serverovej operácie. Toto sa dá však vyriešiť tým, že namiesto toho, aby sa pre každý prvok posielala požiadavka na server, počká sa, kým sa naakumuluje väčší počet prvkov a tieto sa načítajú všetky naraz. Sieťové operácie sú totiž rádovo pomalšie ako výpočet algoritmu. Priebeh algoritmu je potom rovnaký, až na to, že sa načítava väčšie množstvo prvkov a následne sa ich aj viac zo zoznamu *T* odstráni.

Základné dátové štruktúry použité v tomto algoritme sú totožné z algoritmom 3P-NRA. Trieda *AttributeCache* totiž podporuje aj priamy prístup, ale len keď je pripojená na odpovedajúci server.

4.3.4 Heuristiky

Ako abstrakcia heuristiky slúži abstraktná trieda *Heuristic*. Všetky implementované heuristiky musia byť potomkom tejto triedy. Táto abstraktná trieda prikazuje jednotlivým heuristikám implementovať metódu *getColumn()*, ktorá vráti identifikátor zoznamu, z ktorého sa má sekvenčne načítať hodnota. Trieda *Heuristic* ďalej obsahuje odkaz na konkrétnu inštanciu použitého top-k algoritmu, ktorá je potrebná k prístupu ku konkrétnym dátam, z ktorými daný algoritmus pracuje. Väčšina heuristík (až na niektoré veľmi jednoduché) totiž potrebuje poznať agregačnú funkciu, prípadne počty chýbajúcich hodnôt pre jednotlivé zoznamy. Konkrétna implementácia metódy *getColumn()* ako aj dáta, ktoré sú potrebné pre výpočet identifikátora nasledujúceho zoznamu sú závislé na konkrétnej

heuristike.

4.3.5 Vyrovnávacia pamäť

Keďže táto aplikácia funguje na princípe klient-server architektúry a je potrebné, aby bolo vyhľadávanie čo najrýchlejšie, tak nie je prípustné, aby pri sekvenčnom prístupe vyhľadávací algoritmus čakal na každý jeden prvok každého zoznamu, kým sa načíta zo servera. Z tohto dôvodu algoritmus nepristupuje priamo na server, ale len cez vyrovnávaciu pamäť. Túto vyrovnávaciu pamäť implementuje trieda *AttributeCache*. Top-k algoritmus k nej pristupuje ako k nejakému zoznamu a pomocou metódy *getNextValue()* z nej získava nasledujúci prvok vo forme usporiadanej dvojice obsahujúcej jednoznačný identifikátor objektu a hodnotu hľadaného atribútu.

Hlavnú časť vyrovnávacej pamäti tvoria dve paralelne bežiace vlákna a medzi nimi zdielané pole hodnôt načítaných zo servera, ktoré je zoradené od najlepšej hodnoty po najhoršiu. Kým prvé vlákno, nazvime ho *konzument*, sa stará výhradne o obsluhovanie top-k algoritmu, druhé vlákno, ktoré budeme označovať *producent,* má na starosti načítavanie dát zo servera. Činnosť konzumenta spočíva len v tom, že čaká na to, kým si top-k algoritmus metódou *getNextValue()* vyžiada ďalšiu hodnotu, načíta požadovanú hodnotu zo zdielaného poľa a vráti ju algoritmu na ďalšie spracovanie. Hlavná náplň práce producenta je starať sa o to, aby bolo v zdielanom poli vždy dostatočné množstvo hodnôt, aby na ne konzument a teda aj top-k algoritmus nemusel čakať. Implementované je to tak, že akonáhle konzument zistí, že sa pole začína vyprázdňovať (počet prvkov klesne pod vopred určenú úroveň), zobudí producenta, ktorý získa zo servera ďalšie hodnoty a doplní tak zdielané pole. V ideálnom prípade sa teda na server nebude čakať, pretože počas načítavania ďalších hodnôt zo servera producentom je konzument stále schopný dodávať top-k algoritmu potrebné hodnoty. Keď producent doplní zdielané pole, znovu sa uspí a čaká, kým mu dá konzument opäť pokyn na doplnenie hodnôt. Ak je však komunikácia so serverom tak pomalá, že producent nestihne doplniť pole skôr, ako ho konzument úplne vyprázdni, konzument sa uspí a čaká na zobudenie od producenta, ktorý mu tým oznámi, že sú ďalšie hodnoty k dispozícii. Ak sa tento jav stáva príliš často, je vhodné konfiguračne zväčšiť veľkosť zdielaného poľa, prípadne posunúť hranicu, pri ktorej dáva konzument producentovi pokyn na doplnenie poľa. Možno by bolo vhodnejšie v budúcnosti upraviť logiku vyrovnávacej pamäte tak, aby v prípade potreby sama menila tieto nastavenia, avšak takýto mechanizmus je pomerne implementačne náročný a nebol teda do triedy

AttributeCache zakomponovaný.

Ako už bolo spomenuté pri implementácii TA algoritmu, táto trieda podporuje aj priamy prístup. Nie je to kvôli tomu, že by boli pri priamom prístupe tieto hodnoty vopred načítané, pretože nie je možné predvídať, hodnoty akých objektov bude algoritmus vyžadovať. Dôvodom, prečo je táto funkčnosť zahrnutá vo vyrovnávacej pamäti, je len skutočnosť, že táto trieda zároveň obsahuje všetku komunikáciu zo serverom, teda navonok ako by ani žiadny server neexistoval, naopak, trieda *AttributeCache* sa všetkým top-k algoritmom javí, akoby obsahovala všetky hodnoty atribútov. Priamy prístup je teda sprostredkovaný len poslaním požiadavky na server a vrátením výsledku top-k algoritmu, pričom sa vždy jedná kvôli efektivite o požiadavok na viacero prvkov naraz.

4.4 Server

Serverová časť sa stará o uloženie dát a rýchly prístup k nim. Je implementovaná tak, aby mohla fungovať bezstavovo, a teda si neukladá žiadne dodatočné dáta pre nejaké pripojenie. Návrh počítal s komunikáciou pomocou webových služieb, takže nie je potrebné udržiavať stále spojenie s klientom. Z tohto dôvodu sa klienti ani nijako nerozlišujú a ku každej požiadavke sa pristupuje, akoby prišla od nového klienta. Kvôli tomuto je nutné všetky informácie potrebné na získanie dát odosielať s každou požiadavkou. Týchto dát nie je ale nejako prevratne veľa, pretože sa vlastne jedná len o reprezentáciu fuzzy funkcie, ktorá je vzhľadom na množstvo potrebných dát dosť úsporná a ešte reprezentáciu počiatočného stavu, ktorej je venovaný priestor v popise indexovacích algoritmoch.

Nebola implementovaná žiadna autentifikácia ani autorizácia, pretože to nie je predmetom tejto práce. V reálnom využití podobného vyhľadávania by to však asi bolo nutné.

Serverovú časť tvorí hlavne úložisko dát, indexovacie algoritmy a trieda komunikujúca s klientskou časťou. Na túto komunikáciu slúži trieda *ServerAttributeCache*, ktorá je vlastne len rozhraním medzi klientskou časťou a indexovacím algoritmom. Obsahuje metódy *getItems()*, ktorá slúži na sekvenčný prístup k prvkom a *getValuesByID()*, ktorá sa naopak stará o priamy prístup.

4.4.1 Indexovací algoritmus pre 3P-NRA

Pre algoritmus 3P-NRA sa používa indexačný algoritmus vychádzajúci z článku

[4]. Tento algoritmus musel však prejsť niekoľkými úpravami, aby bol prispôsobený pre sieťovú komunikáciu. Jedná sa hlavne o umožnenie tomuto algoritmu vrátiť sa na predtým dosiahnuté pozície v strome, namiesto toho, aby pri každej požiadavke začínal vyhľadávanie od začiatku.

Tento indexačný algoritmus používa ako zdroj dát B+ strom, čo je dátová štruktúra umožňujúca rýchle vyhľadanie objektu podľa kľúča a zároveň efektívne pracuje s dátami uloženými na disku. Pre potreby tejto práce stačí, aby bol tento strom uložený v pamäti, ale za normálnych okolností by to malo byť samozrejme na disku. Dáta sú v tejto štruktúre uložené len v listoch stromu, pričom nelistové uzly slúžia len na prechod stromom. Každý list navyše obsahuje odkazy na susedné listy, pretože je to potrebné pre tento indexačný algoritmus. Dáta uložené v tomto strome sú vlastne dvojice obsahujúce identifikátor objektu a hodnotu atribútu (skutočnú a nie fuzzy hodnotu). Kľúčom však je hodnota atribútu, aby bolo možné vyhľadávanie podľa hodnôt a nie podľa identifikátorov. V tejto práci je B+ strom reprezentovaný triedou *BPlusTree*, ktorá však pôvodne nebola vyvíjaná pre účely tejto práce a je teda prevzatá a len pozmenená, aby vyhovovala potrebám tohto indexačného algoritmu.

Fuzzy funkcia je pre účely tohto algoritmu reprezentovaná ako zoznam úsečiek fuzzy funkcie nazývaných *intervaly*. Tieto úsečky zase reprezentujú inštancie triedy *Interval*. Táto trieda obsahuje počiatočný a koncový bod úsečky a dva pomocné údaje: smer (či je daná úsečka klesajúca, rastúca alebo konštantná) a hodnotu najvyššieho bodu (hodnota počiatočného alebo koncového bodu podľa smer úsečky). Tieto pomocné údaje slúžia len nato, aby sa predišlo zbytočnému opakovanému výpočtu týchto informácií.

Na určenie aktuálnej pozície na danej úsečke je použitá trieda *MovingPoint*, ktorá predstavuje akýsi kurzor, ktorý obsahuje odkaz na objekt v B+ strome, fuzzy hodnotu, smer pohybu a odkaz na interval, v ktorom sa nachádza. Algoritmus používa zoznam týchto bodov, ktorý bude ďalej označovaný ako *MP*. Tento zoznam je zoradený podľa fuzzy hodnôt jednotlivých bodov.

Vstupom algoritmu je fuzzy funkcia reprezentovaná zoznamom bodov, počet požadovaných prvkov a pozície počiatočných bodov. V prípade, že sa jedná o prvé načítanie hodnôt, je tento zoznam prázdny, inak obsahuje hodnoty, ktoré algoritmus vrátil pri predchádzajúcom volaní. Výstupom tohto algoritmu je totiž okrem zoznamu požadovaných dvojíc identifikátorov objektov a hodnôt ich atribútov aj zoznam týchto počiatočných bodov, ktoré slúžia algoritmu na nadviazanie vyhľadávania na predošlé

hľadanie bez nutnosti hľadať od začiatku.

Popis tohto algoritmu vyzerá nasledovne:

1. Podľa fuzzy funkcie sa vytvorí zoznam intervalov, ktorý sa zoradí podľa ich maximálnych hodnôt.
2. V prípade, že zoznam počiatočných bodov je prázdny, tak sa vytvorí *MP*, tak, že pre každý interval sa nájde prvok s najvyššou fuzzy hodnotou. Vykoná sa to vyhľadaním prvého objektu od najvyššieho bodu úsečky v smere jej sklonu (táto operácia je vlastne len nájdením miesta, kde by v strome bol uložený prvok so skutočnou hodnotou vyššieho konca úsečky a postupovaním po listoch B+ stromu k najbližšiemu objektu v danom smere). V prípade, že sú určené počiatočné body, tak sa zoznam *MP* naplní týmito bodmi nájdenými v B+ strome.
3. Vyberie sa prvý objekt zo zoznamu *MP*, odstráni sa z neho a pridá sa do zoznamu výsledných objektov.
4. Pomocou metódy *getNextMP()* získame novú inštanciu triedy *MovingPoint*, ktorá zodpovedá nasledujúcemu objektu v danom intervale. Nájdenie tohto objektu je vlastne len pohybom po listoch B+ stromu v rámci jedného intervalu, samozrejme smerom, ktorý určuje tento interval.
5. Pokiaľ predchádzajúci bod skutočne našiel ďalší objekt, čo sa stane vždy, ak sa v danom intervale nejaký objekt nachádza, tak sa tento nový objekt zaradí na správne miesto zoznamu *MP*.
6. Ak výsledný zoznam obsahuje dostatočný počet objektov, koľko bolo požadovaných klientom, algoritmus skončí, pričom výstupom bude zoznam výsledných objektov a zoznam počiatočných objektov pre budúce vyhľadávanie, ktorý obsahuje prvky zoznamu *MP*. V opačnom prípade algoritmus pokračuje v kroku 3.

4.4.2 Indexovací algoritmus pre TA

Pre sekvenční prístup sa pre algoritmus TA používa rovnaké indexovanie ako pre algoritmus 3P-NRA. Tento fakt vyplýva aj z toho, že ani v klientskej časti nie je rozdiel medzi sekvenčným prístupom týchto dvoch algoritmov.

Pre priamy prístup sa používa indexovanie opísané v článku [3]. Tento indexovací mechanizmus využíva ďalší B+ strom, v ktorom sú však prvky uložené presne naopak ako v prípade B+ stromu pre algoritmus 3P-NRA. Kľúčom jednotlivých prvkov sú teda identifikátory objektov a hodnotami skutočné hodnoty atribútov. Tento spôsob uloženia umožňuje rýchly prístup k objektom pomocou ich identifikátora, čo je presne to, čo priamy prístup vyžaduje. K nájdenej hodnote potom stačí dopočítať fuzzy hodnotu a výsledok poslať klientovi.

5 Literatúra

[1] Fagin, R., Lotem, A., Naor, M.: Optimal aggregation algorithms for middleware. Journal of Computer and System Sciences 66 (2003) 614–656.

[2] Gurský, P.: Algoritmy na vyhľadávanie najlepších k objektov bez priameho prístupu. Proceedings of Znalosti 2006, pp. 95-105.

[3] Gurský P., Vojtáš P.: Multikriteriálne vyhľadávanie najlepších objektov s podporou viacerých užívateľov. Proceedings of Znalosti 2007, pp. 52-62

[4] Eckhardt, A., Pokorny, J., Vojtas, P.: A system recommending top-k objects for multiple users preference. In Proc. of 2007 IEEE International Conference on Fuzzy Systems, July 24-26, 2007, London, England, pp. 1101-1106.

A Obsah CD

Priložený CD disk obsahuje text diplomovej práce a zdrojové kódy implementovanej aplikácie.

Štruktúra adresárov CD:

- `\text`
  - Obsahuje text diplomovej práce vo formáte pdf. Názov súboru je `dp.pdf`
- `\src`
  - Obsahuje kompletné zdrojové kódy