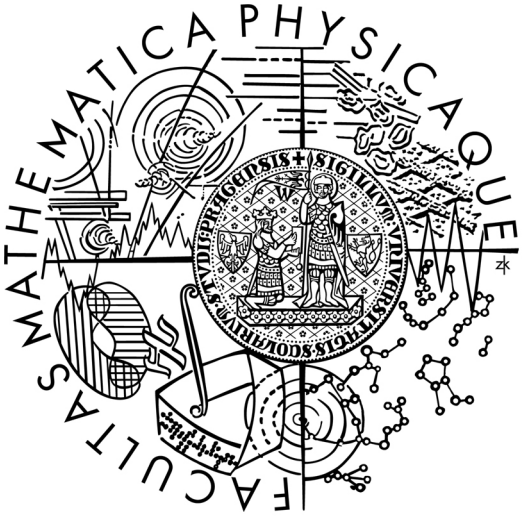
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

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 6. decembra 2010 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 varianty 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 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.

Kapitola 3 sa venuje teoretickému riešeniu top-k problému. Obsahuje detailné popisy už existujúcich a používaných top-k algoritmov. Sú tu taktiež uvedené ich výhody a nevýhody.

Kapitola 4 sa zaoberá spôsobmi indexovania a dátovými štruktúrami, ktoré sú k tomuto potrebné.

Kapitola 5 podrobne popisuje technológie sieťovej komunikácie, ktoré sú využívané pre potreby tejto práce, ako napríklad SOAP alebo RPC.

V kapitole 6 je popísaný spôsob implementácie celého riešenia. 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. Takisto sú tu podrobne popísané aj technológie použité na komunikáciu klienta so serverom, spolu s ich využitím pre účely tejto práce.

Kapitola 7 je venovaná testovaniu vyvinutých riešení. Na tomto mieste sa nachádza porovnanie efektivity jednotlivých algoritmov pri rôznych podmienkach a ich konfigurácii. Cieľom práce bolo optimalizovať celé riešenie hlavne vzhľadom na celkový čas vyhľadávania a v neposlednej rade aj množstvo potrebnej sieťovej komunikácie. Uvedené testy sa preto sústredia hlavne na vplyv rôznych podmienok na tieto veličiny.

V závere sú zhrnuté dosiahnuté výsledky a prínosy tejto práce. Táto kapitola sa taktiež venuje rozšíreniam tejto práce a možnostiam ďalšieho výskumu v tejto zaujímavej a rozsiahlej problematike.

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 z množiny všetkých prvkov P s atribútmi $A_1, A_2, ..., A_m$ a ich hodnotami $a_1^p, a_2^p, ... a_m^p$ 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.

Ako je vidieť, táto práca nepoužíva na ohodnotenie jednotlivých prvkov klasickú dvoj-hodnotovú logiku, ktorá by každému prvku priradila hodnotu 0 alebo 1, podľa toho, či je objekt vhodný alebo nie. Namiesto toho je použitá viac-hodnotová logika, nazývaná tiež *fuzzy* logikou. Fuzzy logika umožňuje ohodnotiť každý objekt hodnotou z intervalu [0, 1], čím určí, nielen ktoré hodnoty sú vhodné a ktoré naopak nie, ale takisto predstavuje aj akúsi mieru vhodnosti pre každú takúto hodnotu.

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

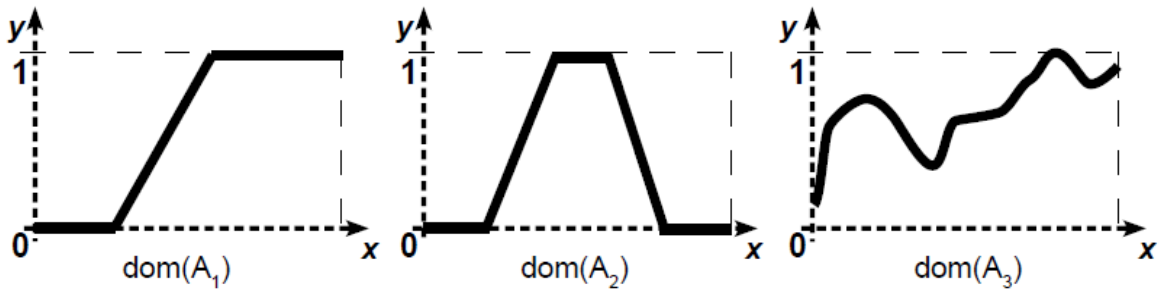
Fuzzy funkcia je zobrazenie f_i^U, pre ktoré platí $f_i^U(a_i^p) = x \in [0,1]$. Čiže je to funkcia, ktorá podľa hodnoty i-teho atribútu prvku p vypočíta jeho takzvanú fuzzy hodnotu, čo je hodnota z intervalu [0, 1], ktorá určuje, nakoľko je hodnota tohto atribútu pre daného užívateľa U vhodná. Podobne, 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.

Nech $adom(A_i)$ je aktuálna doména atribútu A_i, teda množina všetkých hodnôt i-teho atribútu pre všetky prvky p z celkovej množiny prvkov P. Pokiaľ množina $adom(A_i)$ obsahuje len číselné hodnoty, tak existuje interval reálnych čísel R_i, ktorý obsahuje všetky

hodnoty z aktuálnej domény *adom*(A_i). Najmenší takýto interval sa bude nazývať doménou *i*-teho atribútu a značiť *dom*(A_i). V tejto práci sa bude uvažovať len o číselných doménach atribútov, čo umožňuje proces normalizácie, ktorý je popísaný nižšie. Fuzzy funkciu je potom možné chápať ako zobrazenie $f_i : dom(A_i) \rightarrow [0,1]$.

Keďže je použitá spomínaná fuzzy logika, ktorá umožňuje ohodnotiť atribút hodnotu z intervalu [0, 1], tak užívateľ pomocou fuzzy funkcie určí aj akúsi mieru preferencie pre každú takúto hodnotu z domény daného atribútu, teda presne popíše, nakoľko mu ktorá hodnota vyhovuje. Znamená to, že čím je funkčná hodnota atribútu bližšia k 1, tým je reálna hodnota pre daného užívateľa vhodnejšia a naopak.

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.



Obrázok 2.1: Príklady fuzzy funkcií

Na obrázku 2.1 sú zobrazené rôzne typy fuzzy funkcií. Os *x* predstavuje doménu konkrétneho atribútu a os *y* príslušné preferenčné hodnoty podľa daného užívateľa. Vľavo je neklesajúca fuzzy funkcia, keď užívateľ považuje hodnoty atribútu až po určitú hranicu ako úplne neprijateľné a zase od určitej hranice ako ideálne. Pri strednom type existuje interval hodnôt, ktorý užívateľovi vyhovuje najviac, mimo túto oblasť vhodnosť daného atribútu klesá a od určitej spodnej aj vrchnej hranice považuje hodnoty tohto atribútu za nevyhovujúce. Tieto dva typy (monotónna a s extrémom vo vnútri domény) sú pre užívateľské preferencie typické. Vpravo je príklad zložitejšej fuzzy funkcie.

2.1.1 Normalizácia fuzzy funkcií

Pre zjednodušenie práce s jednotlivými fuzzy funkciami je potrebné zjednotiť domény týchto fuzzy funkcií. K tomuto účelu slúži normalizácia domény, ktorá normalizuje túto doménu na interval [0, 1]. Po prevedení normalizácie sú potom všetky fuzzy funkcie tvaru $f_i:[0,1]\rightarrow[0,1]$. Normalizáciou *i*-teho atribútu A_i je potom vlastne zobrazenie $N_i:dom(A_i)\rightarrow[0,1]$, ktoré každej hodnote z domény atribútu A_i priradí hodnotu z intervalu [0, 1].

Normalizácia číselného atribútu je prakticky intuitívna. Stačí zobrať najmenšiu a najväčšiu hodnotu z $dom(A_i)$, ktoré sa budú označovať min_i a max_i. Ak sa potom premietne doména $dom(A_i) = [min_i, max_i]$ na interval [0, 1], tak je normalizovaná. Formálne je možné túto normalizačnú funkciu popísať asi takto:

$$N_i(a_i^p)=\frac{a_i^p-min_i}{max_i-min_i}$$

Toto je samozrejme len jedna z možných normalizačných funkcií, ktorých je nekonečne mnoho. Táto ja však azda najpoužívanejšia a to hlavne kvôli tomu, že zachováva pomer medzi reálnymi a normalizovanými hodnotami, čiže ak je hodnota *x* atribútu A_i dvakrát väčšia ako hodnota *y* toho istého atribútu, tak tomu tak bude aj po normalizácii.

Zdanlivo väčší problém je s normalizáciou atribútov z nečíselnými hodnotami. Tento problém sa však dá vyriešiť v celku jednoducho. Najprv je potrebné doménu takéhoto atribútu, ktorú tvorí vlastne množina nečíselných hodnôt, zotriediť podľa nejakého pevného kritéria. Dostaneme zotriedenú množinu $\{x_i^1, \ldots, x_i^n\}$, ktorá obsahuje všetky hodnoty z $dom(A_i)$. Potom stačí jednotlivé hodnoty rovnomerne rozmiestniť na interval [0, 1]. Formálne bude normalizačná funkcia vyzerať nasledovne:

$$N_i(x_i^j)=\frac{j-1}{n-1}$$

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 atribútov užívateľ vlastne určuje, ktorý atribút je pre neho ako dôležitý. Agregačná funkcia je v tejto práci väčšinou označovaná ako @.

Formálne je možné agregačnú funkciu popísať nasledovne. Nech p je prvok z množiny všetkých prvkov P s hodnotami atribútov $a_1^p, a_2^p, \ldots a_n^p$, pričom uvažujeme len s normalizovanými doménami jednotlivých atribútov (každá doména sa totiž dá normalizovať, ako bolo ukázané v časti 2.1.1). Majme n fuzzy funkcií $f_1^U, f_2^U, \ldots f_n^U$ a ich funkčné hodnoty $h_1, h_2, \ldots h_n$, pre ktoré platí $h_i = f_i^U(a_i^p)$ pre každé $i \in \{1, \ldots n\}$. Agregačná funkcia $@^U$ je potom zobrazenie $@^U : [0,1]^n \rightarrow [0,1]$, pre ktoré platí

$$@^U(p) = @^U(h_1, h_2, \ldots h_n) = @^U(f_1^U(a_1^p), f_2^U(a_2^p), \ldots f_n^U(a_n^p)) .$$

Ako agregačnú funkciu je možné zvoliť prakticky ľubovoľnú funkciu, ktorá vyhovuje definícii, na tomto mieste budú priblížené aspoň niektoré.

Prvým a pravdepodobne aj najpoužívanejším typom agregačnej funkcie je aritmetický priemer $@^U(p) = \frac{\sum_{i=1}^{n} h_i}{n}$, ktorý ma však nevýhodu v tom, že všetky atribúty majú rovnakú váhu a tak sa používa skôr vážený priemer $@^U(p) = \frac{\sum_{i=1}^{n} x_i h_i}{\sum_{i=1}^{n} x_i}$, kde x_i sú nezáporné reálne koeficienty určujúce váhu konkrétneho atribútu. V tomto prípade sa dá vyriešiť aj prípad, keď niektorý atribút je úplne nepodstatný a je teda nežiadúce, aby jeho hodnota ovplyvňovala výslednú celkovú hodnotu agregačnej funkcie. V takomto prípade totiž stačí pre konkrétny atribút nastaviť hodnotu koeficientu na 0. V tejto práci sa využíva práve tento typ agregačnej funkcie, je však prípadne možné jej použitie nahradiť iným typom. Pre výsledky práce nie je typ agregačnej funkcie až tak dôležitý, avšak pre úplnosť bude spomenutých aj zopár ďalších typov.

Zaujímavým typom agregačnej funkcie je minimum, respektíve maximum. Tieto varianty agregačnej funkcie môžu totiž relatívne výrazne ovplyvniť rýchlosť výpočtu

algoritmu pre vyhľadanie najlepších k objektov, ako je to spomínané v [4]. Ako je už intuitívne jasné, jedná sa o funkcie $@^U(p) = min(h_1, h_2, \ldots h_n)$ a $@^U(p) = max(h_1, h_2, \ldots h_n)$.

Okrem spomenutých typov sú veľmi vhodné najrôznejšie priemery, či už s možnosťou priradenia váhy jednotlivým atribútom pomocou akýchsi koeficientov alebo bez nej.

3 Top-k

Úlohou tejto práce je nájsť čo možno najefektívnejší top-k algoritmus vhodný pre manipuláciu s objektami umiestnenými na viacerých vzdialených serveroch. K tomuto účelu je teda nevyhnutné zadefinovať pojem top-k algoritmus. Majme množinu objektov *P*, kde každý objekt $p \in P$ 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*(*p*). Algoritmy, ktoré pre celočíselné $k > 0$ dokážu nájsť *k* prvkov množiny *P* s najväčším ohodnotením *F*(*p*) 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*(*p*) najvyššie.

3.1 Naivný algoritmus

Najjednoduchšie, takzvané naivné riešenie tohto problému, ktorým je možné získať k najlepších objektov je ohodnotiť všetky objekty množiny *P*, 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 $p \in P$
2. Každému takémuto objektu $p \in P$ sa dopočíta jeho ohodnotenie *F*(*p*) (toto je určite možné, keďže sú známe všetky atribúty tohto objektu).
3. Existuje teda množina P' ohodnotených objektov z množiny *P*. 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*(*p*). V takomto prípade je nutné zakaždým prepočítať ohodnotenie pre každý jeden objekt, $p \in P$ 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 $p \in P$ 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 P mohutnosti n, ktorá obsahuje objekty s m atribútmi. Jednotlivé objekty budeme značiť p_1 , ..., p_n a ich atribúty p_i^1, ..., p_i^m. Ďalej je potrebné poznať užívateľskú agregačnú funkciu $@^U : [0,1]^m \rightarrow [0,1]$. Pre názornosť sa bude ohodnotenie objektu p $@^U(f_1^U(p^1), ..., f_m^U(p^m))$ označovať zjednodušene len $@^U(p)$. 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 užívateľskej preferencie konkrétneho atribútu, čiže prvky zoznamu L_i majú tvar $\{p_j, f_i^U(p_j^i)\}$. 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 h s n premennými je vzhľadom na všetky jej premenné neklesajúca práve vtedy, ak platí:

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

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 p_j, teda získanie prvku $\{p_j, f_i^U(p_j^i)\}$ 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 $L_1, \ldots, L_m$ sa postupne vyberajú prvky a ukladajú sa do zoznamu T, až kým v tomto zozname nie je aspoň k objektov s hodnotami všetkých m atribútov známymi.
2. Pre každý nájdený objekt $p \in T$, ktorý nemá hodnoty všetkých m atribútov známe sa načítajú hodnoty chýbajúcich atribútov pomocou priameho prístupu.
3. Dopočíta sa ohodnotenie $@^U(p)$ pre každý nájdený objekt $p \in T$, čím vznikne množina ohodnotených objektov, ktorých počet je minimálne k, čo bolo zabezpečené v kroku 1. 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 $p_1, \ldots, p_k$ takých, že pre každý ďalší doteraz nenájdený objekt p_i sú hodnoty užívateľských preferencií atribútov objektu p_i vo všetkých zoznamoch pod preferenčnými hodnotami príslušných atribútov objektov $p_1, \ldots, p_k$. 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 p_t^i a prahový objekt t, tak platí $t = [p_t^1, ..., p_t^m]$, pričom jednotlivé atribúty prahu typicky nepatria tomu istému reálnemu objektu, ale napríklad atribút p_t^1 by mohol byť v skutočnosti p_5^1 a p_t^2 zase p_2^2, teda prvý atribút by patril v skutočnosti prvku p_5 a druhý prvku p_2.

TA algoritmus pracuje nasledovne:

1. Vykoná sa sekvenčný prístup do zoznamu L_i, ktorý vyberie heuristika h. K takto získanému objektu p sa následne pomocou priamych prístupov do zvyšných

zoznamov načítajú hodnoty všetkých *m* atribútov. Vypočíta sa ohodnotenie daného objektu ako @(*p*) 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 *p*, je jasné, že pre každý jeho atribút platí $p^i \leq p_t^i = t_i$, a teda z neklesajúcej vlastnosti agregačnej funkcie aj $@(p) \leq @(t)$. Označme *k*-ty prvok zoznamu *T* ako p_k. Potom podľa ukončovacej podmienky algoritmu musí platiť $@(p) \leq @(t) \leq @(p_k)$ pre všetky zatiaľ nenájdené objekty *p*. 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 *p*, ktoré sa značí *w*(*p*) 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 *p* značené *b*(*p*) 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 $V(p') = \{i_1, \ldots, i_l\} \subseteq \{1, \ldots, m\}$ je množina indexov jeho atribútov so známymi hodnotami. Ďalej majme objekt $b_{p'}$, pre ktorý platí, že $b_{p'}^i = p'^i$ ak $i \in V(p')$ a $b_{p'}^i = t^i$ v opačnom prípade, kde t^i je *i*-ty atribút prahu *t*. Podobne objekt $w_{p'}^i = p'^i$ pre všetky $i \in V(p)$ a $w_{p'}^i = 0$ pre ostatné *i*. Potom $b(p') = @(b_{p'})$ a $w(p') = @(w_{p'})$. 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 *p* známe, tak platí, že $b_p = w_p$ a teda aj $b(p) = w(p) = @(p)$.

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 $\{p_j, f_i^U(p_j^i)\}$. Ak sa objekt p_j už nachádza v zozname *T* alebo *C*, priradí sa mu novonačítaný atribút $f_i^U(p_j^i)$, teda jeho preferečná hodnota. Prepočíta sa $w(p_j)$ a $b(p_j)$.

   - Ak je hodnota $w(p_j)$ väčšia ako w_k, tak: Pokiaľ sa nachádza objekt p_j v zozname T, tak sa presunie na správne miesto tohto zoznamu. Ak sa tento objekt

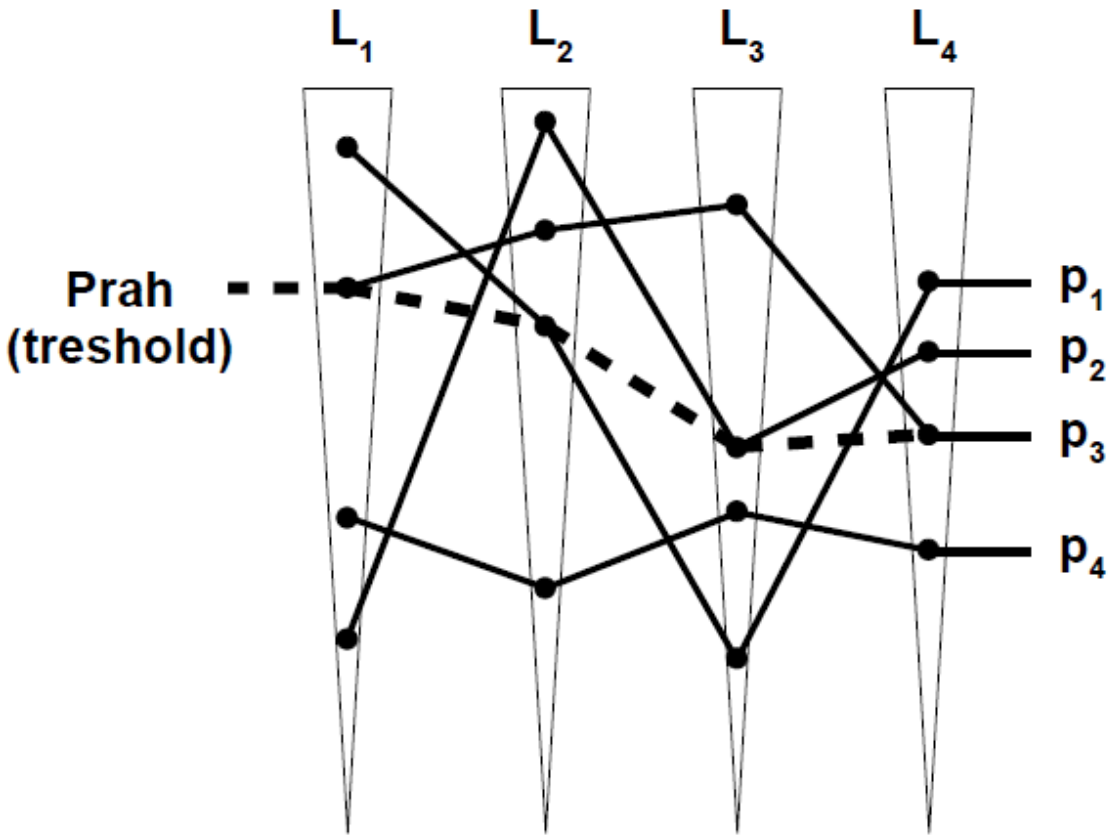
vyskytuje v zozname C, tak bude z tohto zoznamu vyradený a pridaný do zoznamu T. K-ty prvok zoznamu T bude následne z neho odobratý a pokiaľ bude jeho najlepšia hodnota väčšia ako nová hodnota w_k, tak tento objekt zaradíme do zoznamu C.

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

2. Prejde sa celý zoznam C a pre každý jeho prvok p sa dopočíta hodnota $b(p)$ podľa aktuálneho prahu. Ak je pre niektoré takéto x $b(p) \leq w_k$, 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.



Obrázok 3.1: Zotriedené zoznamy pri NRA algoritme

Na obrázku 3.1 je vidieť jednotlivé zotriedené zoznamy uprostred výpočtu. Podľa pozície prahu sú na obrázku prvky v rôznych stavoch. Pri prvku p_3 už algoritmus pozná hodnoty všetkých jeho štyroch atribútov, pretože sú nad úrovňou prahu alebo na nej. Prvky p_1 a p_2 sú známe čiastočne, pretože algoritmus načítal len hodnoty niektorých ich atribútov, ale ešte nie všetky (oba majú jeden atribút neznámy). Prvok p_4 ešte algoritmus NRA neobjavil, pretože sa hodnoty všetkých jeho atribútov nachádzajú pod prahovými hodnotami.

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 článkov [2] a [5]. 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 $b(p)$ 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 p.
2. Pre objekt p sa vypočítajú jeho hodnoty $b(p)$ a $w(p)$ a následne sa tento objekt zaradí na správne miesto v zozname T.
3. Ak je hodnota $b(p)$ 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 p_j zo zoznamu T pre $j > k$ prepočíta algoritmus $b(p_j)$.
2. Ak je $b(p_j) < w_k$, objekt p_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 p.
2. Ak sa objekt p nachádza v zozname T, prepočítajú sa jeho hodnoty $w(p)$ a $b(p)$ a zaradí sa na správne miesto zoznamu T. Ak $b(p) < w_k$, 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 *p* v zozname *T* zmenila hodnota *w*(*p*) 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 Indexovanie pomocou B+ stromov

Indexovanie je v tejto práci založené na dátovej štruktúre zvanej B+ strom. Pre každý atribút existuje vlastný index a teda aj vlastný B+-strom. Cieľom tejto kapitoly je popis ako samotnej dátovej štruktúry, tak samotného indexovacieho algoritmu.

4.1 B strom

B stromy je možné definovať napríklad tak, ako je to uvedené v [6]:

B stromy (rádu m) sú rozvetvené výškovo vyvážené stromy, ktoré spĺňajú nasledujúce podmienky:

1) koreň má najmenej dvoch potomkov, pokiaľ nie je listom,
2) každý uzol okrem koreňa a listov má najmenej $\lceil m / 2 \rceil$ a najviac m potomkov,
3) každý uzol má najmenej $\lceil m / 2 \rceil - 1$ a najviac $m - 1$ dátových záznamov,
4) všetky vetvy sú rovnako dlhé,
5) dáta v uzloch sú organizované nasledovne:

   $$p_0, (k_1, p_1, d_1), (k_2, p_2, d_2), \ldots, (k_n, p_n, d_n), u$$

   kde $p_0, p_1, \ldots, p_n$ sú ukazatele na potomkov, $k_0, k_1, \ldots, k_n$ sú kľúče, $d_0, d_1, \ldots, d_n$ sú asociované dáta prípadne ukazovatele na dáta ležiace mimo strom, u je nevyužitý priestor a (k_i, p_i, d_i) záznamy sú usporiadané vzostupne podľa kľúčov, pričom

   $$\lceil m / 2 \rceil - 1 \leq n \leq m - 1,$$

6) keď zodpovedá ukazovateľu p_i podstrom $U(p_i)$, potom platí:

   a) pre každé k v $U(p_{i-1})$ je $k < k_i$,

   b) pre každé k v $U(p_{i+1})$ je $k < k_i$, kde $i \in \{1, \ldots, n\}$.

B stromy spĺňajúce túto definíciu sa nazývajú *neredundantné*, pretože každý kľúč sa v celom strome nachádza najviac jeden krát. Kvôli minimalizácii výšky stromov sa však častejšie používajú takzvané *redundantné* B stromy, v ktorých sa dáta (respektíve ukazatele na dáta, ktoré ležia mimo strom) nachádzajú len v listových uzloch. Pre túto vlastnosť je

potrebné upraviť podmienky 5 a 6 v pôvodnej definícii B stromu nasledovne:

5) dáta v nelistových uzloch sú organizované nasledovne:

$$p_0, (k_1, p_1, d_1), (k_2, p_2, d_2), \ldots, (k_n, p_n, d_n), u$$

a v listových takýmto spôsobom:

$$(k_1, d_1), (k_2, d_2), \ldots, (k_n, d_n), u$$

kde $p_0, p_1, \ldots, p_n$ sú ukazatele na potomkov, $k_0, k_1, \ldots, k_n$ sú kľúče, $d_0, d_1, \ldots, d_n$ sú asociované dáta prípadne ukazovatele na dáta ležiace mimo strom, u je nevyužitý priestor a (k_i, p_i, d_i) záznamy sú usporiadané vzostupne podľa kľúčov, pričom

$$\lceil m / 2 \rceil - 1 \leq n \leq m - 1,$$

6) keď zodpovedá ukazovateľu p_i podstrom $U(p_i)$, potom platí:

a) pre každé k v $U(p_{i-1})$ je $k \leq k_i$,

b) pre každé k v $U(p_{i+1})$ je $k < k_i$, kde $i \in \{1, \ldots, n\}$.

V redundantných B stromoch majú listové a nelistové uzly značne rozdielnu štruktúru (čo vyplýva z podmienky 5), čo v praxi často krát vedie k odlišnej implementácii týchto dvoch typov uzlov. Redundancia tohto typu B stromu spočíva v tom, že upravená podmienka 6 dovoľuje viacnásobný výskyt jednotlivých kľúčov (nie však v tej istej úrovni).

4.2 B+ strom

V niektorých prípadoch práce s B stromami je potrebné vyhodnotiť takzvaný *rozsahový dotaz*. Tento sa dá definovať nasledovne: pre dve hodnoty a a b vyhľadá rozsahový dotaz všetky prvky, pre ktoré má hodnota ich kľúča k hodnotu spĺňajúcu podmienku $a \leq k \leq b$. Pre tento typ dotazu je klasický B strom definovaný v predchádzajúcej časti pomerne nevýhodný, pretože algoritmus vyhodnocovania rozsahového dotazu v tejto dátovej štruktúre potrebuje využívať prídavný zásobník na prechod stromom, čo je značne neefektívne. Táto komplikácia sa dá riešiť modifikáciou B stromu zvanou B+ strom. Ten sa dá definovať nasledovne:

B+ strom je B strom, pre ktorý navyše platí:

1) uzly každej úrovne sú jednosmerne zreťazené

2) každý uzol obsahuje ukazateľ na svojho pravého suseda

Pre účely tejto práce sa používa ešte trochu upravená verzia redundantného B+ stromu, kde postačuje, aby boli zreťazené len listové uzly, avšak sú potrebné ukazatele na oboch susedov takéhoto uzlu. Takéto stromy sa budú v rámci práce nazývať len B+ stromy a pokiaľ nebude explicitne uvedené inak, tak pod pojmom B+ strom sa bude rozumieť vždy tento druh dátovej štruktúry.

4.3 Využitie B+ stromov pri indexovaní

Faginov algoritmus popísaný v [1] potrebuje pri svojom chode zoznamy L_i dvojíc $\{p_j, f_i^U(p_j^i)\}$ zotriedené podľa hodnôt príslušnej fuzzy funkcie konkrétneho užívateľa U, ktorý sa snaží pomocou tohto algoritmu nájsť vhodné prvky. Všetkých prvkov však môže byť značné množstvo, takže je potrebné uchovávať ich vo vhodnej dátovej štruktúre, ktorá by umožňovala čo možno v najkratšom čase vyhľadať konkrétnu hodnotu. Ak by sme hľadali riešenie len pre tohto jedného užívateľa, bola by situácia jednoduchá, avšak táto práca počíta s viacužívateľským prístupom, kde nie je možné, pre každého jedného užívateľa vytvárať vlastný zoznam podľa jeho konkrétnej fuzzy funkcie. Musí byť teda navrhnutá metóda indexovania s jedným spoločným indexom pre všetkých potenciálnych užívateľov, ktorá by bola úplne nezávislá od užívateľskej fuzzy funkcie.

Tento problém rieši návrh indexovacieho algoritmu v [4]. Uvedený návrh využíva index založený na B+ strome. V tomto strome sú uložené dáta vo forme dvojíc $\{p_j, p_j^i\}$. Tieto dvojice však nie sú uložené v B+ strome tak, ako by to bolo zdanlivo správne a intuitívne, teda že kľúč predstavuje identifikátor objektu p_j a samotné dáta budú tvoriť konkrétne hodnoty atribútu p_j^i. Na počudovanie tomu bude presne naopak, to znamená, že celá štruktúra bude indexovaná pomocou hodnôt atribútu, ktoré budú predstavovať kľúč a samotné identifikátori prvkov budú vlastne dátami. Po menšom zamyslení však vypláva na povrch, že tento spôsob indexovania je naozaj správny, pretože pri sekvenčnom prístupe algoritmus nevie, aký prvok potrebuje, čiže nepozná ani jeho identifikátor a preto podľa neho ani nepotrebuje vyhľadávať.

Dôležitý fakt je, že v strome nie je uložená fuzzy hodnota, ale skutočná hodnota daného atribútu, čím bola dosiahnutá nezávislosť od konkrétneho užívateľa.

Fuzzy funkcia je pre účely tohto algoritmu reprezentovaná ako zoznam úsečiek (je

možné kvôli reprezentácii fuzzy funkcie popísanej v časti 6.2) fuzzy funkcie nazývaných *intervaly*. Každý takýto interval 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 definovaná štruktúra *kurzor*, ktorá 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 *K*. Tento zoznam je zoradený podľa fuzzy hodnôt jednotlivých bodov.

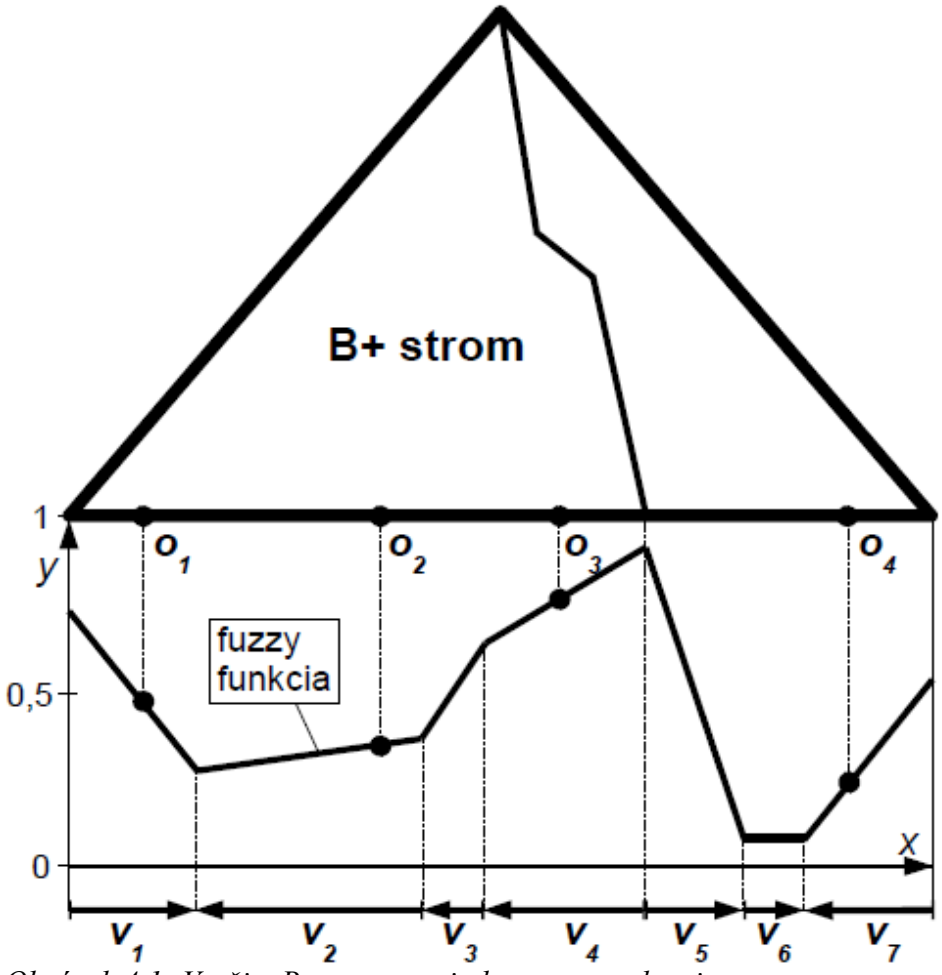
Vstupom algoritmu je fuzzy funkcia značená f_i^U reprezentovaná intervalmi a počet požadovaných prvkov

Samotný indexovací algoritmus je možné popísať nasledovne:

1. Podľa fuzzy funkcie sa vytvorí zoznam *I,* obsahujúci všetky intervaly, ktorý sa zoradí podľa ich maximálnych hodnôt.
2. Vytvorí sa zotriedený zoznam *K*, tak, že pre každý interval z *I* 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 B+ strome bol uložený prvok so skutočnou hodnotou vyššieho konca úsečky a postupovaním po listoch B+ stromu k najbližšiemu objektu v danom smere).
3. Vyberie sa prvý objekt zo zoznamu *K* (teda ten s najvyššou fuzzy hodnotou), odstráni sa z neho a pridá sa do zoznamu výsledných objektov.
4. Od tohto nájdeného prvku postupujeme v rámci konkrétneho intervalu v smere určenom týmto intervalom, čím nájdeme nasledujúci objekt *o*. Je to vlastne len pohyb medzi listami (keďže je to redundantný B+ strom, tak všetku objekty sa nachádzajú v listoch) v smere sklonu fuzzy funkcie v rámci jedného jej intervalu.
5. Pokiaľ predchádzajúci bod skutočne našiel ďalší objekt *o*, čo sa stane vždy, ak sa v danom intervale ešte nejaký objekt nachádza, tak sa tento nový objekt zaradí na správne miesto zoznamu *K*.
6. Ak výsledný zoznam obsahuje dostatočný počet objektov (číslo zadané na

vstupe), koľko bolo požadovaných Faginovým algoritmom, tak tento indexovací algoritmus skončí, pričom výstupom bude zoznam výsledných objektov. V opačnom prípade algoritmus pokračuje v kroku 3.

Je evidentné, že tento algoritmus postupne nájde všetky prvky uložené v B+ strome, pokiaľ fuzzy funkcia je definovaná na celej jej doméne a zároveň zo zotriedenia zoznamu K vyplýva, že sa stále odoberie prvok s najvyššou fuzzy hodnotou, čím sa zabezpečí správne zotriedenie zoznamu L_i, ktoré vyžaduje Faginov algoritmus.



Obrázok 4.1: Využite B+ stromu v indexovacom algoritme

Na obrázku 4.1 je znázornené, akým spôsobom využíva indexovací algoritmus B+ strom pri svojom behu. Na začiatku si algoritmus vytvorí zoznam intervalov, ktoré sú na tomto obrázku znázornené pomocou vektorov v_1, v_2, ... v_7. Potom nájde v strome miesto reprezentujúce maximum fuzzy funkcie a od tohto bodu hľadá objekty o_1 až o_4 s využitím jednotlivých intervalov, ktoré sú na uvedenom obrázku vykreslené ako vektory v_1, v_2, ... v_7.

5 Komunikácia pomocou Webových služieb

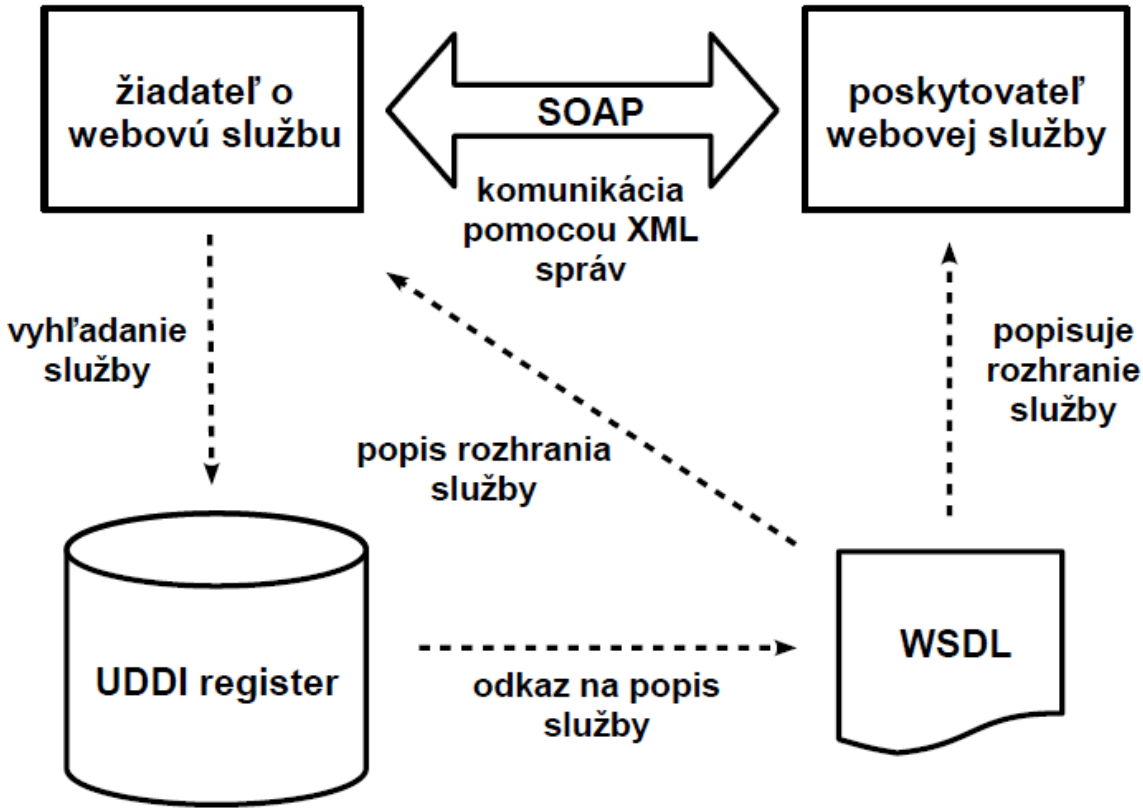
Keďže sa táto práca zaoberá vyhľadávaním *k* najlepších prvkov v distribuovanom prostredí, musí nutne riešiť sieťovú komunikáciu. K tomuto účelu slúžia Webové služby. Využíva sa pri tom SOAP protokol (konkrétne jeho RPC typ), ktorý používa XML a ako prenosový protokol HTTP. Jednotlivé technológie sú popísané v nasledujúcich odstavcoch.

5.1 Webové služby

Webovú službu definuje konzorcium W3C, ktoré vydáva webové štandardy, v [8] nasledovne:

Webová služba je softwarový systém navrhnutý na podporu interakcie medzi dvoma počítačmi cez sieť. Jej rozhranie je popísané v strojovo spracovateľnom formáte (konkrétne WSDL). Ostatné systémy interagujú s Webovou službou spôsobom predpísaným jej popisom pomocou SOAP správ typicky prenášaných použitím HTTP protokolu a XML serializáciou v súčinnosti s ostatnými webovými štandardmi.

Webová služba je abstraktný pojem, ktorý musí byť implementovaný konkrétnym agentom. Agentom je software alebo hardware, ktorý posiela a prijíma správy, zatiaľ čo služba je zdroj charakterizovaný abstraktnou sadou poskytovaných funkcionalít. Behom transakcie je jeden agent žiadajúcim o službu a druhý poskytujúcim túto službu. Štandardy, ktoré špecifikujú webové služby majú pre účely tejto komunikácie zabezpečiť rovnakú sémantiku oboch agentov. Špecifikácia Webovej služby počíta pri jej vykonávaní s tromi účastníkmi. Okrem už spomínaných dvoch (žiadateľ a poskytovateľ) sa objavuje ešte takzvaný register služieb. V tomto registri poskytovateľ zverejňuje definíciu služby pomocou protokolov UDDI a WSDL. Žiadateľ si službu vyhľadá a spojí sa s poskytovateľom využívajúc pri tom definíciu služby poskytnutú registrom služieb. Celý tento proces je znázornený na obrázku 5.1.



Obrázok 5.1: Vzťah základných technológií Webových služieb

5.2 SOAP

Skratka SOAP pôvodne znamenala *Simple Object Access Protocol* (protokol slúžiaci k jednoduchému prístupu k objektom), pretože prvotný zámer bol, ako to už názov napovedá, vytvoriť protokol, ktorý by umožňoval jednoduchý prístup k objektom cez sieť. Neskôr sa však táto požiadavka rozšírila z objektov na komunikáciu prostredníctvom XML správ, čím pôvodná skratka trochu stratila zmysel. Keďže však táto skratka bola už pomerne dobre zaužívaná, rozhodli sa tvorcovia ponechať ju v pôvodnom znení, avšak už ako skratku bez ďalšieho významu.

Protokol SOAP je definovaný konzorciom W3C v [9] nasledovne:

SOAP je jednoduchý protokol určený na výmenu štrukturovaných dát v decentralizovanom distribuovanom prostredí. Používa XML technológie na definovanie rozšíriteľného komunikačného frameworku poskytujúceho konštrukciu správ, ktoré môžu byť vymieňané pomocou rôznych komunikačných protokolov.

Základom tohto riešenia je jeho jednoduchosť, rozšíriteľnosť a nezávislosť od

programovacích modelov. Práve kvôli jednoduchosti chýbajú tomuto systému funkcionality bežné v distribuovaných systémoch ako napríklad zabezpečenie spoľahlivosti, podpora smerovania (*routing*) alebo bezpečnosť komunikácie. Tieto funkcionality je však možné do konkrétneho riešenia doplniť vďaka rozšíriteľnosti tohto systému.

XML správa posielaná pomocou využívajúc protokol SOAP je zabalená v elemente *envelope* (obálka), ktorý je vlastne základným elementom dokumentu vo formáte SOAP. Obálka obsahuje voliteľný element *header* (hlavička) a povinný element *body* (telo). Telo obsahuje samotný obsah správy. Keďže tento element môže obsahovať ľubovoľné množstvo elementov z akéhokoľvek *namespace*, tak práve v ňom sú uložené všetky dáta, ktoré je potrebné poslať. Špecifikácia SOAP ďalej definuje v elemente *body* element *fault*, ktorý slúži na spracovanie výnimiek.

Nasledujúci príklad ukazuje pripomienku uskutočnenú pomocou SOAP správy. Hlavička obsahuje vlastný element *alertcontrol*, ktorý určuje prioritu a čas vypršania notifikácie. V tele správy je potom pomocou vlastného elementu *alert* uvedené znenie konkrétnej pripomienky. V tomto príklade je znázornené aj použitie rôznych *namespace*-ov slúžiacich na definíciu špecifických elementov vo vnútri štandardných elementov SOAP protokolu.

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Header>
  <n:alertcontrol xmlns:n="http://example.org/alertcontrol">
   <n:priority>1</n:priority>
   <n:expires>2010-09-22T14:00:00-05:00</n:expires>
  </n:alertcontrol>
 </env:Header>
 <env:Body>
  <m:alert xmlns:m="http://example.org/alert">
   <m:msg>Pick up Mary at school at 2pm</m:msg>
  </m:alert>
 </env:Body>
</env:Envelope>
```

Vďaka rozšíriteľnosti a nezávislosti SOAP umožňuje výmenu správ použitím veľkého množstva komunikačných protokolov. Najčastejšie využívaný protokol je HTTP, ale je možné použiť aj iné protokoly, ako napríklad SMTP alebo MSMQ. Nasledujúci príklad ukazuje prepojenie SOAP a HTTP podľa príslušnej špecifikácie.

Požiadavka:

```
POST / HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope ...>
...
</soap:Envelope>
```

Odpoveď:

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope ...>
...
</soap:Envelope>
```

5.3 WSDL

Skratka WSDL znamená Web Service Description Language (jazyk popisujúci webové služby). WSDL dokument je vo formáte XML a definuje všetko, čo potrebuje aplikácia vedieť na zavolanie príslušnej webovej služby. V tejto práci je využívaná verzia WSDL 1.1, takže nasledujúci popis sa takisto týka tejto verzie. Dokument WSDL sa skladá z piatich základných sekcií, ktoré sú reprezentované špecifickými elementami, ako je to popísané v [7].

Sekcia *types* definuje dátové typy potrebné na komunikáciu s danou webovou službou.

Časti *message* obsahuje abstraktné definície prenášaných dát. Typicky pre volanie každej webovej služby, rovnako ako pre odpoveď na toto volanie WSDL dokument obsahuje samostatný element *message*. Tento element sa ďalej skladá z častí interpretovanými elementami *part*.

Element *portType* predstavuje sadu operácií, ktoré daná služba poskytuje. Jednotlivé operácie sú interpretované elementami *operation,* pomocou ktorých sú definované vstupy, výstupy a reakcie na chyby.

Sekcia *binding* slúži na popis konkrétnych komunikačných protokolov potrebných

na volanie jednotlivých operácií, ktoré boli definované v časti *portType*. Pre každú operáciu môže existovať viacero podporovaných protokolov.

Element *service* určuje pomocou elementov *port* umiestnenie webovej služby pre každý protokol definovaný v časti *binding*. Na tomto mieste sa nachádza konkrétna adresa a port pre danú webovú službu.

5.4 RPC

RPC znamená Remote Procedure Call (vzdialené volanie procedúr) a táto skratka je naozaj výstižná. Jedná sa totiž o mechanizmus, ktorý umožňuje, aby aplikácia zavolala procedúru na inom (vzdialenom) stroji cez sieť. V ideálnom prípade RPC úplne zakrýva rozdiel medzi volaním lokálnej a vzdialenej procedúry. Volajúci si teda vôbec nemusí byť vedomí, že sa daná procedúra vykonáva na vzdialenom počítači. Celé volanie vzdialenej procedúry prebieha tak, že volajúci zavolá lokálnu procedúru akejsi spojky, ktorá sa nazýva *stub* a táto spojka sa už postará o samotné vzdialené volanie a v prípade potreby vráti výsledok tohto volania. Podobná spojka sa nachádza aj na strane servera (príjemcu vzdialeného volania). Táto serverová spojka prijme požiadavku od klientskej spojky a zavolá z pohľadu samotného servera lokálnu procedúru. Po jej vykonaní upozorní klientskú spojku a vráti jej prípadný výsledok volania procedúry. Z tohto pohľadu je možné sa na celý systém RPC pozerať ako na prídavnú vrstvu sprostredkujúcu sieťovú komunikáciu, ktorú nahradzuje volaním lokálnych procedúr. *Stub* má na starosti okrem nadviazania sieťového spojenia so serverom aj prevedenie všetkých parametrov danej procedúry do tvaru vyhovujúceho sieťovej komunikácii a ich následné pripojenie k správe posielanej na server. Tento úkon sa nazýva *marshalling* alebo tiež *serializing*. Serverová spojka potom správu prijme a vytiahne z nej potrebné parametre. Táto akcia sa analogicky nazýva *demarshalling* alebo *deserializing*.

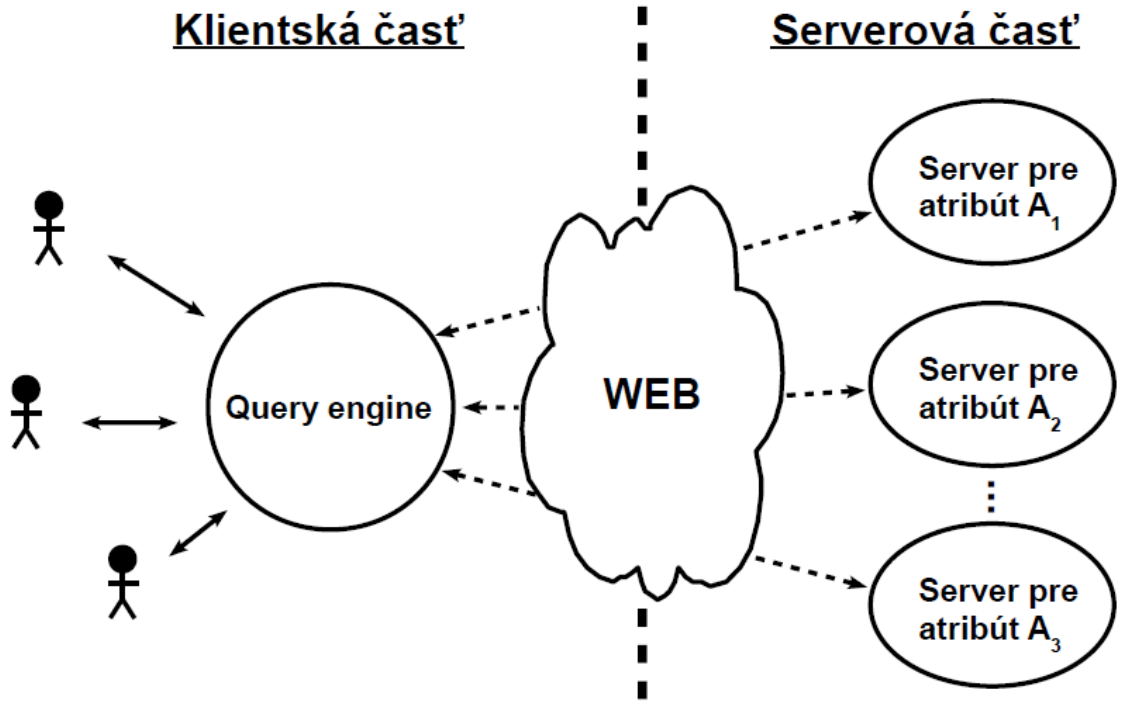
6 Implementácia

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.

6.1 Rozdelenie na klientskú a serverovú časť

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

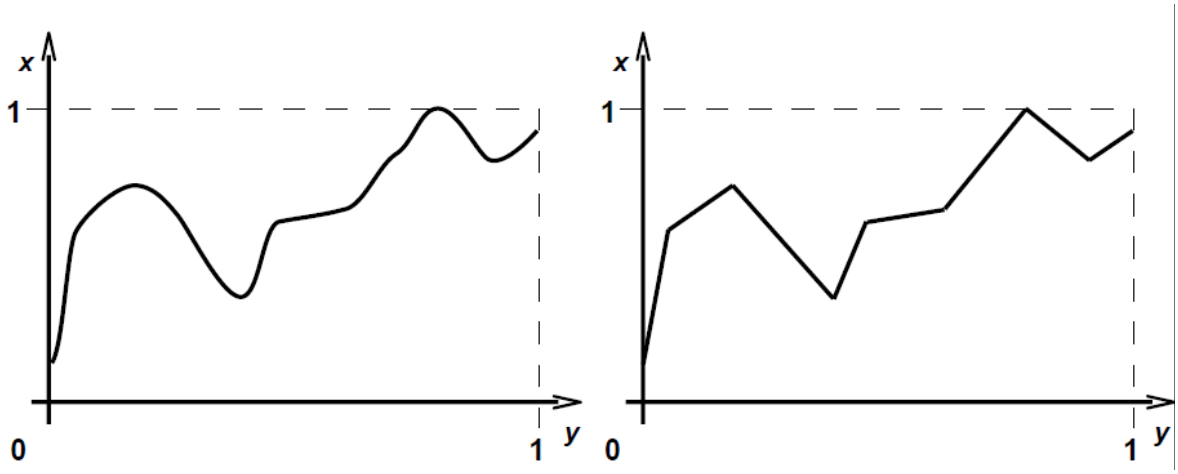


Obrázok 6.1: Rozdelenie aplikácie na klientskú a serverovú časť

Na obrázku 6.1 je prehľadne zobrazené zjednodušené rozdelenie aplikácie na serverovú a klientskú časť. Podrobnejšie sú jednotlivé časti popísané v nasledujúcich kapitolách.

6.2 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.



Obrázok 6.2: Aproximácia fuzzy funkcie

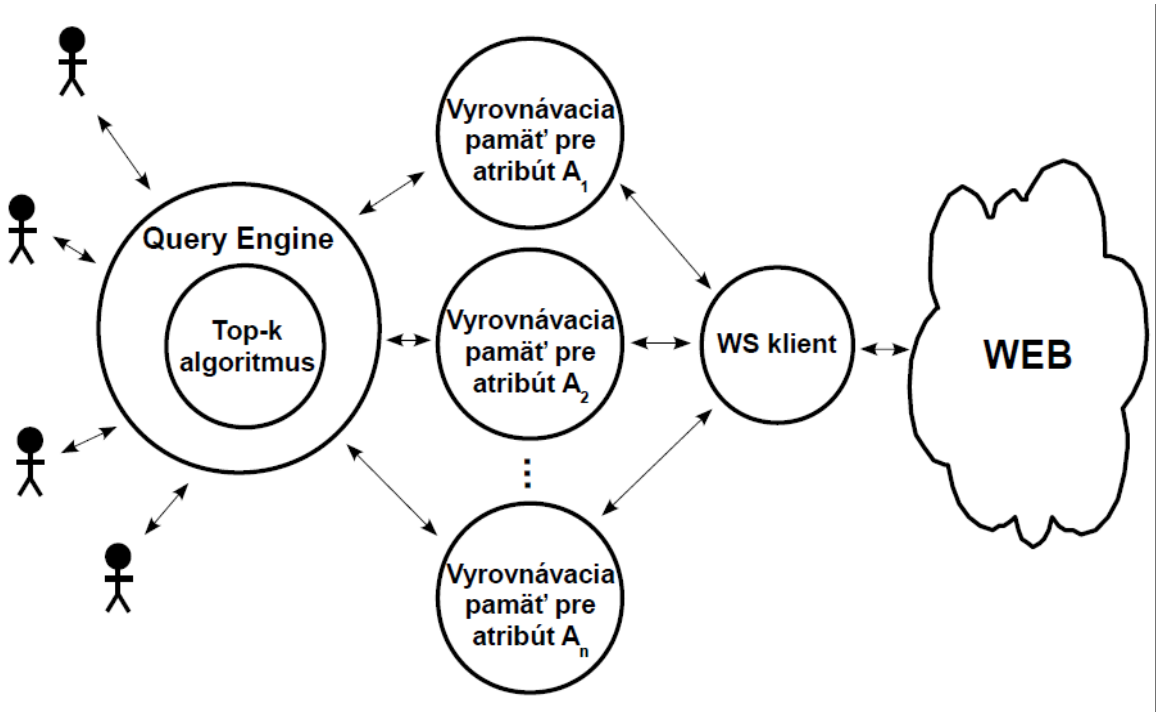
Na obrázku 6.2 je ukázaný príklad aproximácie konkrétnej fuzzy funkcie podľa vyššie popísaného zjednodušenia. Podľa potreby by mohli byť jednotlivé úsečky kratšie, aby boli hodnoty aproximovanej funkcie bližšie reálnym hodnotám alebo naopak dlhšie, aby bola daná reprezentácia jednoduchšia.

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. V práci je implementovaná len jediná agregačná funkcia a to vážený priemer, ktorá je reprezentovaná triedou *AverageAgregationFunction*, ktorá absolútne postačuje účelom práce. Nie je však problém implementovať ďalšie agregačné funkcie, keďže je to abstraktnou triedou *AgregationFunction* umožnené.

6.3 Klient

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



Obrázok 6.3: Klientská časť

Obrázok 6.3 zobrazuje prehľad architektúry klientskej časti. Vstupným bodom je *Query Engine*, ktorý zaobaľuje konkrétny algoritmus na vyhľadávanie *k* najlepších prvkov. Tento využíva vyrovnávacie pamäte jednotlivých atribútov, ktoré prostredníctvom klienta pre Webové služby (*WS klient*) komunikujú cez webové prostredie s príslušnými servermi.

6.3.1 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.

6.3.2 3P-NRA

Najdôležitejšou úlohou pri navrhovaní implementácie jednotlivých algoritmov bola čo najlepšia voľba vhodných dátových štruktúr. Pre zoznam *T*, ktorý počas behu aplikácie obsahuje aktuálne načítané prvky s ich momentálne známymi preferenčnými hodnotami atribútov (viď kapitolu 3.2.4), 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 (viď časť 6.3.5), 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.

6.3.3 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 s algoritmom 3P-NRA. Trieda *AttributeCache* totiž podporuje aj priamy prístup, ale len keď je pripojená na odpovedajúci server.

6.3.4 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, s 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.

6.3.5 Vyrovnávacia pamäť

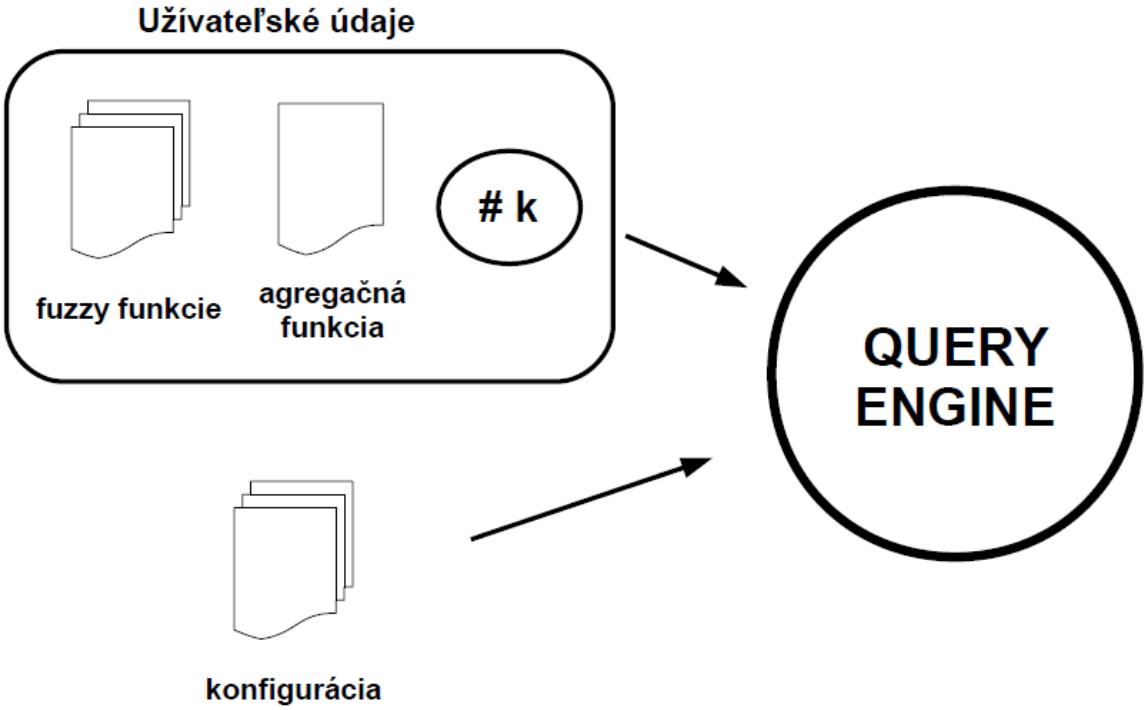
Keďže táto aplikácia funguje na princípe klient-server architektúry a je potrebné, aby bolo vyhľadávanie čo najrýchlejšie, tak nie je prípustné, aby pri sekvenčnom prístupe vyhľadávací algoritmus čakal na každý jeden prvok každého zoznamu, kým sa načíta zo servera. Z tohto dôvodu algoritmus nepristupuje 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 pamäte 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*(), ktorá predstavuje sekvenčný prístup, vyžiada ďalšiu hodnotu, načíta požadovanú hodnotu zo zdielaného poľa (ktoré už obsahuje zotriedené hodnoty užívateľských preferencií pre daný atribút) 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.

6.3.6 Užívateľské vstupy

Vstupný bod celej aplikácie z pohľadu užívateľa tvorí *Query Engine*. Ten potrebuje od užívateľa dostať predovšetkým jednotlivé fuzzy funkcie a agregačnú funkciu. Za užívateľský vstup sa dá takisto považovať aj konfiguračný súbor, ktorý určuje, aký algoritmus sa má pri vyhľadávaní použiť, všetky parametre špecifické pre tento algoritmus (ako napríklad heuristiky), veľkosť vyrovnávacej pamäte, adresy serverov, na ktorých sa hodnoty jednotlivých atribútov nachádzajú a kde sú uložené jednotlivé fuzzy funkciu rovnako ako aj agregačná funkcia. *Query Engine* dostane taktiež na vstupe informáciu o tom, koľko najlepších prvkov požaduje užívateľ vlastne vyhľadať.



Obrázok 6.4: Užívateľské vstupy

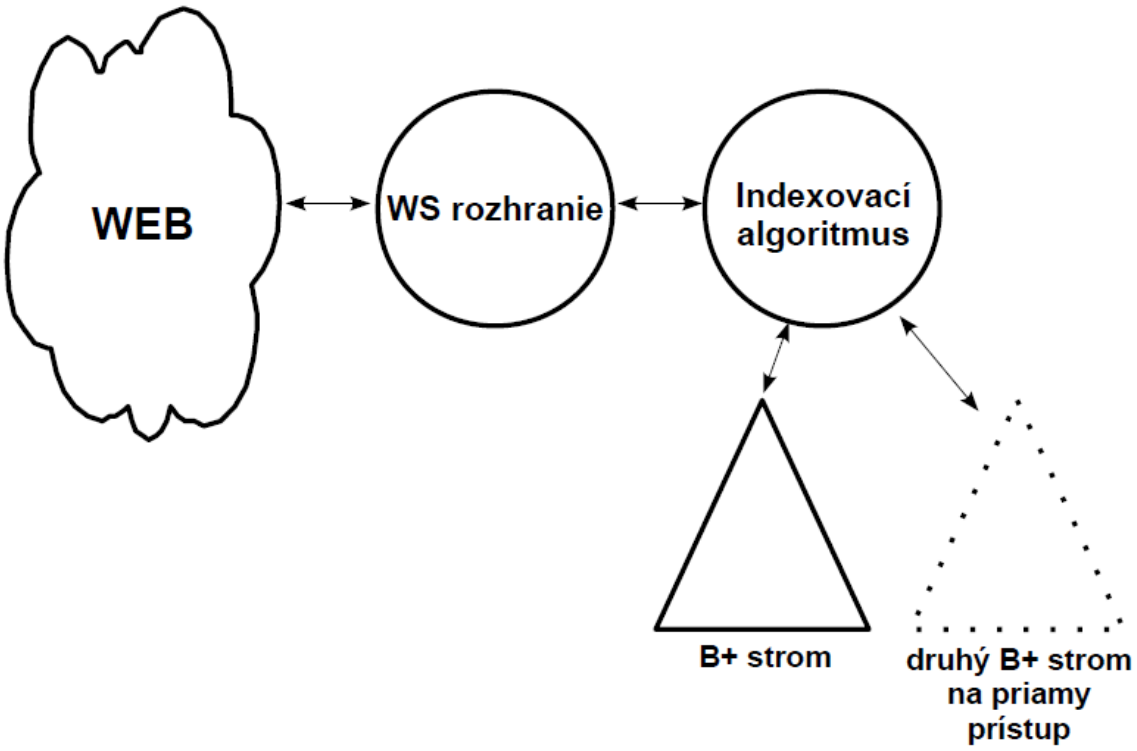
Obrázok 6.4 ukazuje vstupy potrebné pre beh algoritmu. Sú rozdelené medzi údaje zadávané priamo užívateľom a konfiguráciu aplikácie.

6.4 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.



Obrázok 6.5: Serverová časť

Na obrázku 6.5 je znázornená architektúra serverovej časti aplikácie. Jadrom celej tejto časti je indexovací algoritmus, ktorý pracuje s úložiskom dát reprezentovaným jedným (algoritmus 3P-NRA) alebo dvomi B+ stromami (v prípade TA algoritmu). Indexovací algoritmus komunikuje cez webové prostredie s klientom pomocou rozhrania pre webovú službu (*WS rozhranie*).

6.4.1 Indexovací algoritmus pre 3P-NRA

Pre algoritmus 3P-NRA sa používa indexovací algoritmus vychádzajúci z článku [4]. Tento algoritmus (jeho popis je uvedený v 4.3) 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.

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 indexovacieho algoritmu.

Jednotlivé intervaly fuzzy funkcie popísané v časti 4.3 sú reprezentované triedou *Interval* kurzory triedou *MovingPoint*. Na získanie nasledujúceho objektu daného intervalu slúži metóda *getNextMP*().

Z dôvodu požiadavky na pokračovanie v predchádzajúcom výpočte algoritmu pribudol k už spomínaným vstupom aj zoznam pozícií počiatočných bodov. Samotný algoritmus na výstupe vráti spolu s výslednými hodnotami aj zoznam bodov, ktoré poslúžia ako počiatočné body pri potrebe získania ďalších hodnôt. V prípade, že sa jedná o prvé načítanie hodnôt, je na vstupe algoritmu tento zoznam prázdny.

V samotnom indexovacom algoritme potom dôjde len k dvom zmenám.

Prvou je vytvorenie zoznamu kurzorov (reprezentovaných triedou *MovingPoint*). Tento zoznam sa totiž v zmenenom algoritme vytvára zo zoznamu intervalov len pri prvom volaní, to znamená, keď nie sú k dispozícii počiatočné body. Ak však algoritmus dostane tieto body na vstupe, vytvorí potrebný zoznam K z nich, a to tak, že ich len vyhľadá pomocou prechodu stromom.

Druhá zmena sa týka výstupu algoritmu. Ako už bolo spomínané, tak táto modifikácia tohto algoritmu musí na výstupe vrátiť spolu s požadovanými hodnotami aj zoznam počiatočných bodov pre ďalšie volanie algoritmu. V tomto však nie je nič komplikované, pretože tento zoznam vlastne tvoria prvky zoznamu K pri ukončení algoritmu v kroku 6, ktorý sa preto pošle na výstup.

6.4.2 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 v klientskej časti dopočítať fuzzy hodnotu.

6.5 Komunikácia klienta so serverom

Ako už bolo spomínané, pre potreby sieťovej komunikácie je v tejto práci využívaná technológia webových služieb, konkrétnejšie RPC typ SOAP protokolu komunikujúci pomocou HTTP.

Serverová časť beží na aplikačnom serveri *Tomcat* verzie 6.0 a použitý je framework pre webové služby nazývaný *Axis2*. Tieto technológie boli zvolené kvôli relatívne jednoduchému používaniu a dobrej stabilite. Navyše keďže oba tieto softwarové produkty sú vyvíjané tou istou spoločnosťou, tak sú aj optimalizované na vzájomnú spoluprácu, čím sa šanca na stabilné fungovanie aplikácie samozrejme zvyšuje.

6.5.1 Implementácia webovej služby

V tejto práci sú je implementovaná jedna webová služba s dvoma operáciami, ktoré zodpovedajú dvom rôznym spôsobom potrebným pre algoritmy na vyhľadávanie *k* najlepších prvkov. Jedna operácia je určená na sekvenčný prístup a druhá na priamy.

Na strane klienta sa o komunikáciu pomocou webových služieb stará trieda *PrefQuerWSClientImpl*, ktorá týmto plní úlohu klientskej spojky (*stub*) z pohľadu RPC. Táto trieda implementuje rozhranie *PrefQuerWSClient*, ktoré definuje spomenuté dve operácie. Pre sekvenčný prístup obsahuje metódu *getItems*(), ktorá ako parametre potrebuje celočíselný počet načítavaných hodnôt, fuzzy funkciu reprezentovanú vektorom bodov (viď popis tejto reprezentácie v 6.2) a vektor takzvaných štartovacích bodov (viď kapitolu 6.4.1). Podobne pre priamy prístup je tu metóda *getValuesByID*(), ktorá má jediný parameter, a to vektor identifikátorov prvkov, ktorých preferencie potrebuje algoritmus zo servera načítať. Tento *stub* takisto zabezpečuje *marshalling* parametrov, ktoré je potrebné poslať na server a zároveň *demarshalling* návratových hodnôt pre oba typy prístupov. Variant s manuálne vytvoreným *stubom* namiesto vygenerovania pomocou príslušných nástrojov bol zvolený kvôli potrebe vlastnej serializácie komplexných parametrov, ako aj kvôli nízkej spokojnosti s výsledkom generovania tohto *stubu*, ktorá bola zrejme z časti

zapríčinená aj nedostatočnými skúsenosťami s danými nástrojmi.

Serverovú časť komunikácie tvorí hlavne trieda *PrefQuerWSImpl*, ktorá implementuje rozhranie *PrefQuerWS*. Toto rozhranie definuje podobne ako rozhranie *PrefQuerWSClient* v klientskej časti metódy pre jednotlivé typy prístupov. Z triedy *PrefQuerWSImpl* je generovaný aj WSDL dokument popisujúci implementovanú webovú službu. Táto trieda okrem spomenutých metód pre priamy a sekvenčný prístup, ktoré priamo volajú metódy jednotlivých indexovacích algoritmov, obsahuje aj metódy na *marshalling* návratových hodnôt posielaných klientovi a *demarshalling* správ prijatých od klienta. Toto je analogické z klientským *stubom*. Dôležitou súčasťou tejto triedy je zároveň metóda slúžiaca na inicializáciu indexovacích algoritmov. Jedná sa o metódu *initPrefQuerAlgorithm()* a jej úloha spočíva v nastavení parametrov indexovacieho algoritmu podľa konfiguračného súboru servera a načítania samotného indexu, ktorý tvorí B+ strom, respektíve dva B+ stromy, pokiaľ sa jedná o indexovanie pre algoritmus TA (viď kapitoly 6.4.1 a 6.4.2). Táto metóda sa spúšťa staticky pri štarte samotného servera, čím je zabezpečené vytvorenie indexu pred jeho použitím.

Kvôli prehľadnosti je potrebné zjednodušene zhrnúť priebeh celej komunikácie klienta so serverom:

1) Top-k algoritmus potrebuje načítať ďalšiu fuzzy hodnotu pre konkrétny atribút, vypýta si ju teda od vyrovnávacej pamäte (*AttributeCache*).
2) Pokiaľ má vyrovnávacia pamäť má potrebnú hodnotu už načítanú a uloženú lokálne v zásobníku, tak ju danému algoritmu vráti a ku žiadnej komunikácii klienta so serverom nedochádza. V opačnom prípade zavolá príslušnú metódu triedy *PrefQuerWSClientImpl*.
3) Spojka (*stub*) *PrefQuerWSClientImpl* prevedie pomocou *marshallingu* parametre volania do potrebnej podoby a pošle správu na server.
4) Trieda *PrefQuerWSImpl* príjme správu, pomocou *demarshallingu* získa potrebné parametre a pomocou príslušného indexovacieho algoritmu získa hľadané hodnoty. Výsledok zabalí do správy vhodnej pre potreby sieťovej komunikácie pomocou *marshallingu* a pošle ho klientovi.
5) Klient, konkrétne trieda *PrefQuerWSClientImpl* výsledok pomocou *demershallingu* rozkóduje a poskytne ho vyrovnávacej pamäti.

6) Vyrovnávacia pamäť reprezentovaná triedou *AttributeCache* následne poskytne vyžadovanú hodnotu top-k algoritmu, z ktorého pohľadu išlo len o lokálne volanie načítania príslušnej hodnoty. Vyrovnávacia pamäť sa mu totiž javí ako zásobník, ktorý obsahuje tieto hodnoty a nie ako prostredník pri sieťovej komunikácii so serverom.

7 Testovanie implementovaných algoritmov

Jedným z cieľov tejto práce bolo aj otestovanie implementovaných algoritmov s rôznymi parametrami a pri rozličných podmienkach. Z pohľadu efektívnosti daného riešenia sú v tomto prípade najdôležitejšie tieto veličiny:

- celkový čas výpočtu
- množstvo sieťovej komunikácie

Pod celkovým časom sa rozumie len čas, ktorý algoritmus potrebuje na nájdenie *k* najlepších prvkov, čiže sa do tohto intervalu nepočíta doba strávená inicializáciou ako napríklad načítanie konfigurácie a podobne. Keďže čas výpočtu závisí vo veľkej miere od konkrétneho hardwaru, slúži táto veličina len na porovnávanie vhodnosti algoritmov alebo ich parametrov.

Keďže celé riešenie funguje v prostredí webu, je azda najdôležitejším kritériom práve množstvo sieťovej komunikácie. Túto veličinu predstavuje v tomto prípade počet volaní webovej služby.

Testy prebiehali nad množinou náhodne vygenerovaných dát, kde hodnoty jednotlivých atribútov boli rovnomerne rozložené po celých svojich doménach. Dáta boli vygenerované pomocou konfigurovateľného generátora vyvinutého pre účely tejto práce.

7.1 Sieťové prostredie

Všetky experimenty v tejto práci prebiehali na jednom počítači. Komunikácia síce využívala klasické volania webových služieb, ktoré boli nasadené na aplikačnom serveri, avšak ako klient, tak aj server bežali fyzicky na rovnakom počítači. Nebola teda priamo použitá sieťová vrstva a priepustnosť siete bola nastavovaná pomocou serverovej premennej, ktorá je nastaviteľná v konfiguračnom súbore. Samozrejme funkčnosť aplikácie bola overená aj v reálnom sieťovom prostredí, avšak samotné experimenty prebiehali lokálne. Toto bolo výhodné kvôli konštantnej a teda nekolísavej latencii siete, ktorá by za iných okolností mohla negatívne ovplyvniť výsledky experimentov. Cieľom tejto práce totiž nebol návrh aplikácie, ktorá by si poradila s najrôznejšími chybami a výpadkami sieťového pripojenia, ale skôr fungujúce riešenie *top-k* vyhľadávania v distribuovanom prostredí, čo sa týka návrhu, prípadne úpravy algoritmov na

vyhľadávanie *k* najlepších prvkov podľa užívateľských preferencií. K tomuto účelu je testovanie aplikácie zvoleným spôsobom nie len postačujúce, ale dokonca vhodnejšie.

7.2 Parametre algoritmov

Jednotlivé algoritmy sú konfigurovateľné pomocou viacerých parametrov, ktoré výrazne ovplyvňujú rýchlosť výpočtu, množstvo sieťovej komunikácie aj počet načítaných prvkov. Jednou z úloh testovania je aj praktická ukážka ako tieto parametre vplývajú na výkonnosť systému. Zoznam týchto parametrov je nasledovný:

- *veľkosť vyrovnávacej pamäte* – je to vlastne veľkosť jedného z dvoch zásobníkov, ktoré má vyrovnávacia pamäť reprezentovaná triedou *AttributeCache* (viď časť 6.3.5) k dispozícii, čiže doslovne vzaté je to vlastne polovica z celkovej veľkosti vyrovnávacej pamäte. Zároveň je to aj počet prvkov načítavaných zo servera pri jednom volaní webovej služby pri sekvenčnom prístupe. Tento parameter je spoločný pre oba použité algoritmy (NRA aj TA). Čím je jeho hodnota vyššia, tým menší je počet volaní webovej služby, avšak môže sa zvýšiť celkový počet načítaných prvkov. Navyše narastie aj veľkosť jednotlivých správ posielaných zo serveru.
- *b* – je špecifický parameter pre algoritmus 3P-NRA popísaný v 3.2.4. Určuje koľko krát má prebehnúť fáza 3 tohto algoritmu, kým sa vykoná kód fázy 2. Táto fáza je výpočtovo dosť náročná, pretože sa v nej prepočítavajú najhoršie a najlepšie možné ohodnotenia značného množstva prvkov. Na druhej strane, táto fáza musí občas prebehnúť, aby sa odstránili nepotrebné prvky, ktoré už nemôžu byť vo výslednej množine, čím sa zmenší počet prvkov, s ktorými algoritmus počíta, a tým sa samotný výpočet urýchli.
- *N* – tento parameter využíva len algoritmus TA popísaný v 3.2.2. Určuje počet prvkov, ktoré by sa mali načítavať pri jednom priamom prístupe. Toto číslo vlastne predstavuje minimálny počet sekvenčných načítaní, ktoré je potrebné uskutočniť pred jedným priamym prístupom. Aby nevznikol zmätok je nutné na tomto mieste pripomenúť, že zatiaľ čo sekvenčný prístup nemusí nutne znamenať komunikácia s webovou službou, u priameho prístupu je nutná sieťová komunikácia. Hodnoty pre sekvenčný prístup sú totiž už väčšinou uložené vo vyrovnávacej pamäti, avšak u priameho prístupu nie je podľa tohto riešenia možné vedieť vopred, ktoré prvky

bude výpočet algoritmu potrebovať. Priamy prístup je preto časovo najnáročnejšou operáciou algoritmu TA.

- *M* – parameter podobný *N* predstavuje maximálny počet sekvenčných načítaní, ktoré sa uskutočnia pred jedným priamym prístupom. Sekvenčným prístupom totiž algoritmus často dostane prvok, ktorý už má načítaný (načítal ho z iného zoznamu sekvenčne a priamym prístupom k nemu dohľadal chýbajúce hodnoty). Môže sa stať, že takýchto „zbytočných" hodnôt bude príliš veľa a tento parameter slúži na to, aby v tomto podobnom prípade algoritmus postupoval vo svojom výpočte ďalej, aj keď za cenu načítania menšieho množstva prvkov pri priamom prístupe. Táto situácia totiž spravidla nastáva ku koncu výpočtu, keď už v mnohých prípadoch nie je potrebný veľký počet nových načítaných prvkov a algoritmus by sa tam „zasekol" na zbytočne dlhú dobu.

Všetky uvedené parametre sú konfigurovateľné pomocou príslušného konfiguračného súboru. Veľkosť vyrovnávacej pamäte je v týchto súboroch označovaná ako *cacheSize*, u ostatných parametroch sa ich značenie zhoduje s tým, ktoré je uvedené v prechádzajúcom odstavci.

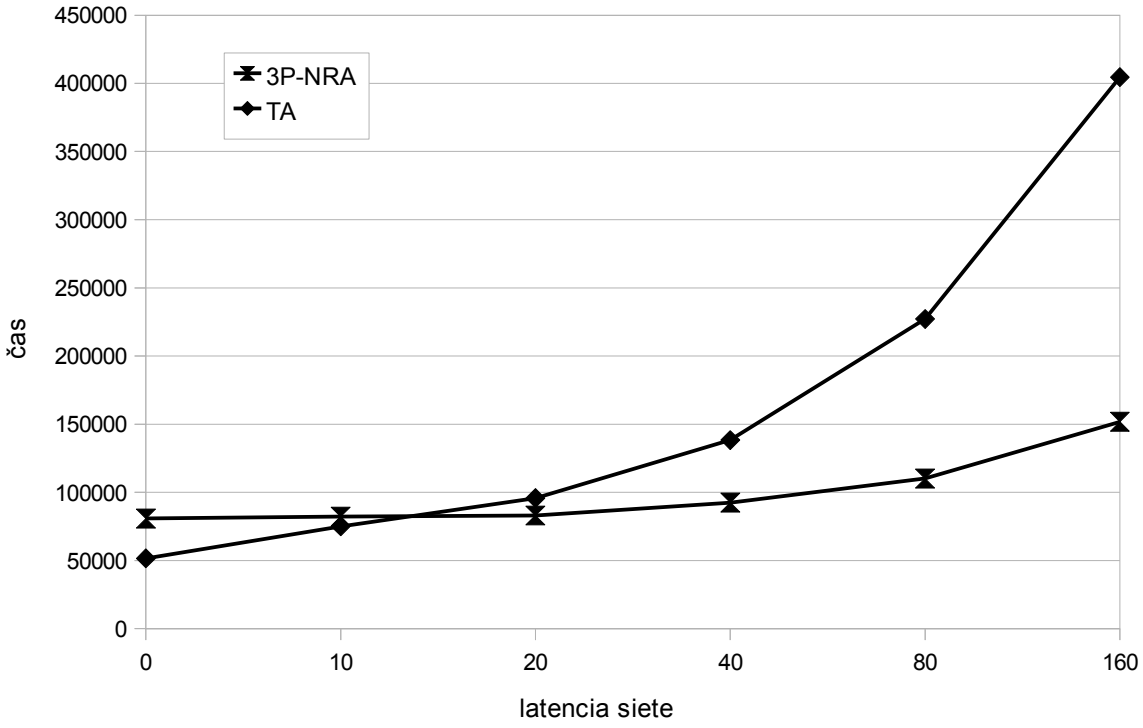
7.3 Závislosť času výpočtu od latencie siete

Keďže klient so serverom komunikujú cez sieť, tak jej nižšia priechodnosť a teda väčšia latencia negatívne ovplyvňuje výkon algoritmu. Tento jav by mala čiastočne obmedziť vyrovnávacia pamäť, avšak len pri sekvenčnom prístupe. Nasledujúci experiment ukazuje, aký je vlastne vplyv latencie na celkový čas výpočtu.

Horšia priechodnosť siete je pre účely tohto testu simulovaná jednoduchým zastavením výpočtu na strane serveru na určený čas. Latencia je uvádzaná v milisekundách a je možné ju nastaviť pomocou konfiguračného súboru pri štarte servera.

Z uvedeného experimentu vyplýva, že pomalšia sieť oveľa viac ovplyvňuje chod algoritmu TA oproti 3P-NRA. Ako už bolo spomenuté, je to spôsobené vyrovnávacou pamäťou používanou pri sekvenčnom prístupe. Pri priamom prístupe totiž nie je možné skonštruovať podobnú vyrovnávacia pamäť, a tým predísť výraznému spomaleniu výpočtu pri vyššej latencii siete. Dôvodom je fakt, že samotný algoritmus nedokáže efektívne predikovať, ktoré prvky bude potrebovať načítať priamym prístupom v nasledujúcich krokoch behu.

Test prebiehal na množine 100000 prvkov, ktoré majú štyri atribúty. Algoritmy vyhľadávali 10 najlepších prvkov pri veľkosti vyrovnávacej pamäte 100 a hodnotami ostatných parametrov b = 100, N = 100 a M = 1000. V tomto experimente bola zvolená geometricky rastúca latencia siete, kvôli vyzdvihnutiu jej rozdielneho pôsobenia na jednotlivé algoritmy.



Graf 7.1: Závislosť času výpočtu algoritmu od latencie siete

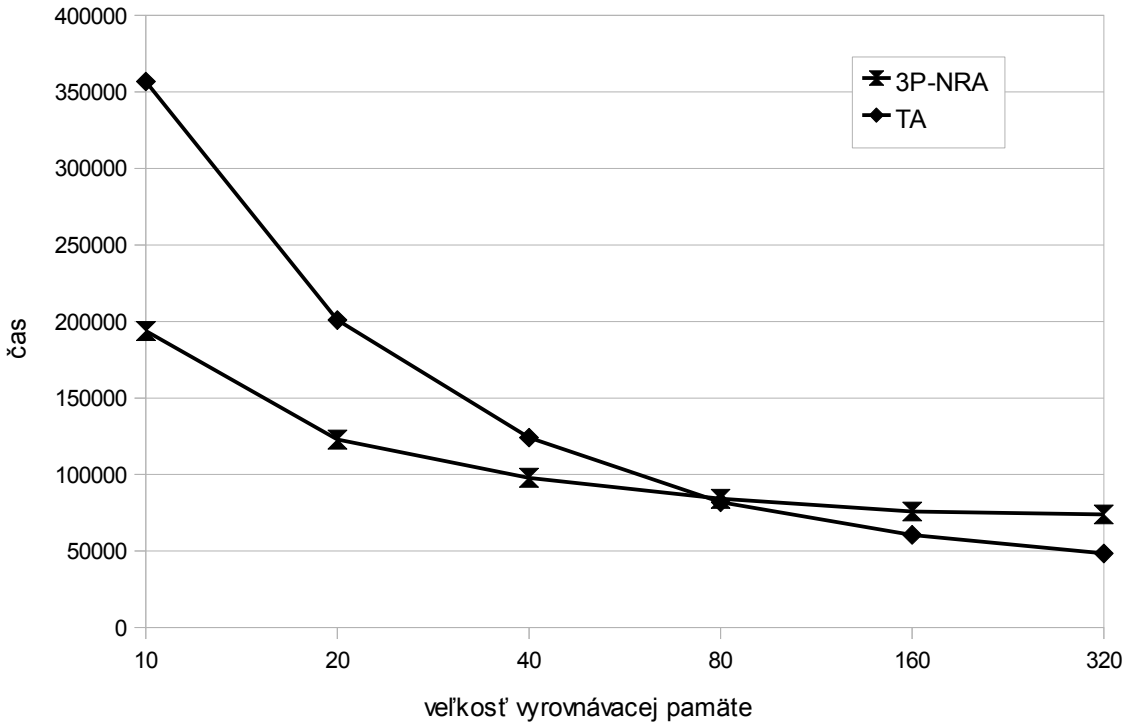
7.4 Testovanie vplyvu vyrovnávacej pamäte

Tento experiment slúži na ukážku vplyvu veľkosti vyrovnávacej pamäte na čas výpočtu a počet volaní webovej služby.

Znovu bolo vyhľadávaných 10 prvkov z celkového počtu 100000, pričom každý mal 4 atribúty. Latencia bola nastavená na 10 milisekúnd, parameter b na 100, parameter N sa vždy rovnal veľkosti vyrovnávacej pamäte a M mal hodnotu desaťnásobne väčšiu.

Na veľkosti vyrovnávacej pamäte záleží jednak množstvo vopred načítaných prvkov a jednak počet prvkov, ktoré sa pri jednom volaní webovej služby načítajú zo servera. Zvýšením tejto hodnoty sa teda výrazne zníži množstvo komunikácie klienta so

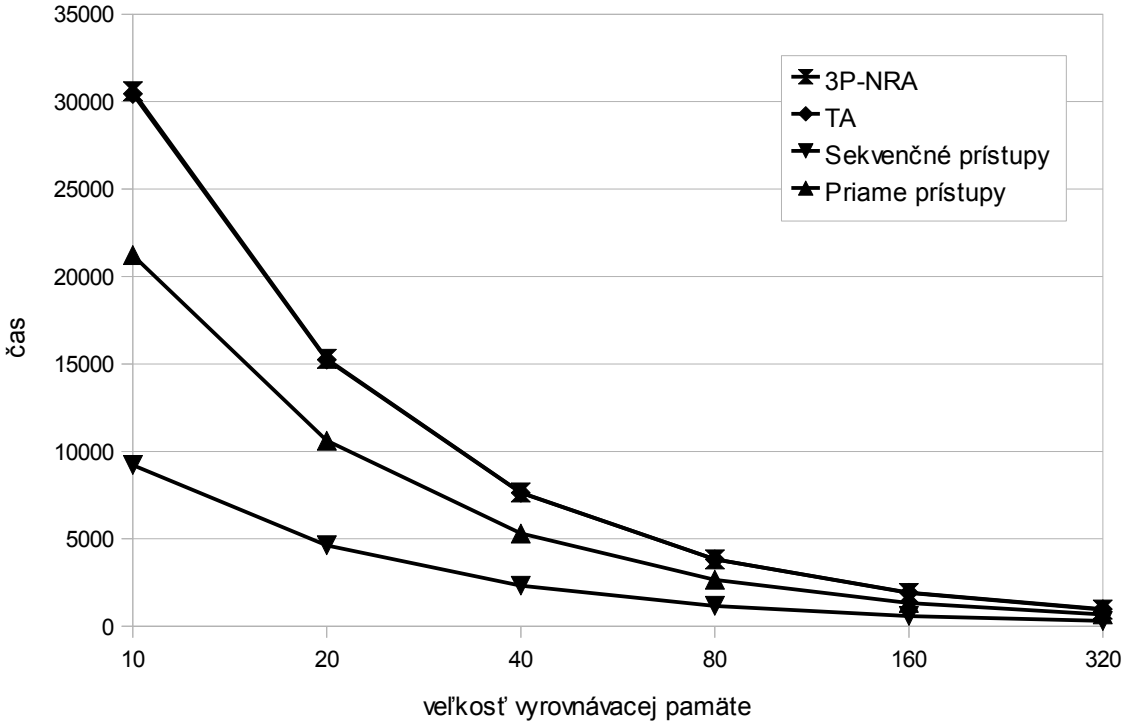
serverom, avšak zvýši sa objem prenášaných dát pre jednotlivé volania. Zároveň sa tým minimalizuje potreba aktívneho čakania na dáta, ktoré nastáva vtedy, keď príslušný top-k algoritmus spotrebuje všetky prvky z vyrovnávacej pamäte a musí sa preto jeho výpočet pozastaviť do chvíle, kedy budú k dispozícii ďalšie prvky. Toto je vlastne hlavný účel tejto vyrovnávacej pamäte, avšak úplne sa vyhnúť tomuto problému nie je možné, najmä pri vysokej latencii siete.



Graf 7.2: Graf závislosti dĺžky behu programu od veľkosti vyrovnávacej pamäte

Z grafu vyplýva, že veľkosť vyrovnávacej pamäte naozaj výrazne ovplyvňuje čas, potrebný na výpočet algoritmu, avšak s vyššími hodnotami vplyv tejto premennej postupne klesá. Rovnako sa ukazuje, že na nej podstatne viac závisí rýchlosť výpočtu pri algoritme bez priameho prístupu. Toto je implikované faktom, že pri algoritme TA je nutné aktívne čakať na načítanie prvkov priamym prístupom, pri ktorom vyrovnávacia pamäť rýchlosť neovplyvní.

V experimente bola zvolená geometricky rastúca veľkosť vyrovnávacej pamäte, aby lepšie vynikol fakt, že ďalším zvyšovaním tejto premennej sa čas výpočtu programu neskracuje až tak výrazne, ako pri nižších hodnotách, aj keď je jej hodnota neustále zdojnásobovaná.



Graf 7.3: Graf závislosti počtu volaní webovej služby od veľkosti vyrovnávacej pamäte

Na grafe 7.3 sú znázornené počty volaní webovej služby pri behu jednotlivých algoritmov. Je vidieť, že algoritmus 3P-NRA a algoritmus TA si vymenili so serverom, na ktorom bežala webová služba takmer rovnaké množstvo správ a teda grafy reprezentujúce túto komunikáciu prakticky splývajú. Na obrázku sú zároveň rozdelené počty volaní služby reprezentujúce priamy a nepriamy prístup pre algoritmus TA. Ich pomer ovplyvňujú viaceré faktory, ako napríklad počet atribútov a parametre *M* a *N*. V uvedenom experimente tvorili volania pre priamy prístup približne 70% z celkového počtu. Keďže medzi grafmi jednotlivých algoritmov sú z obrázku zle postrehnuteľné rozdiely, tak hodnoty zanesené v predchádzajúcom grafe sú znázornené aj v nasledujúcej tabuľke. Stĺpce tabuľky predstavujú veľkosť vyrovnávacej pamäte a riadky algoritmus a druh prístupu, pre ktorý je daný počet volaní webovej služby zaznamenaný (zhora najprv všetky volania pre algoritmus 3P-NRA, potom všetky volania pre TA a ďalej len priame, respektíve iba sekvenčné prístupy algoritmu TA).

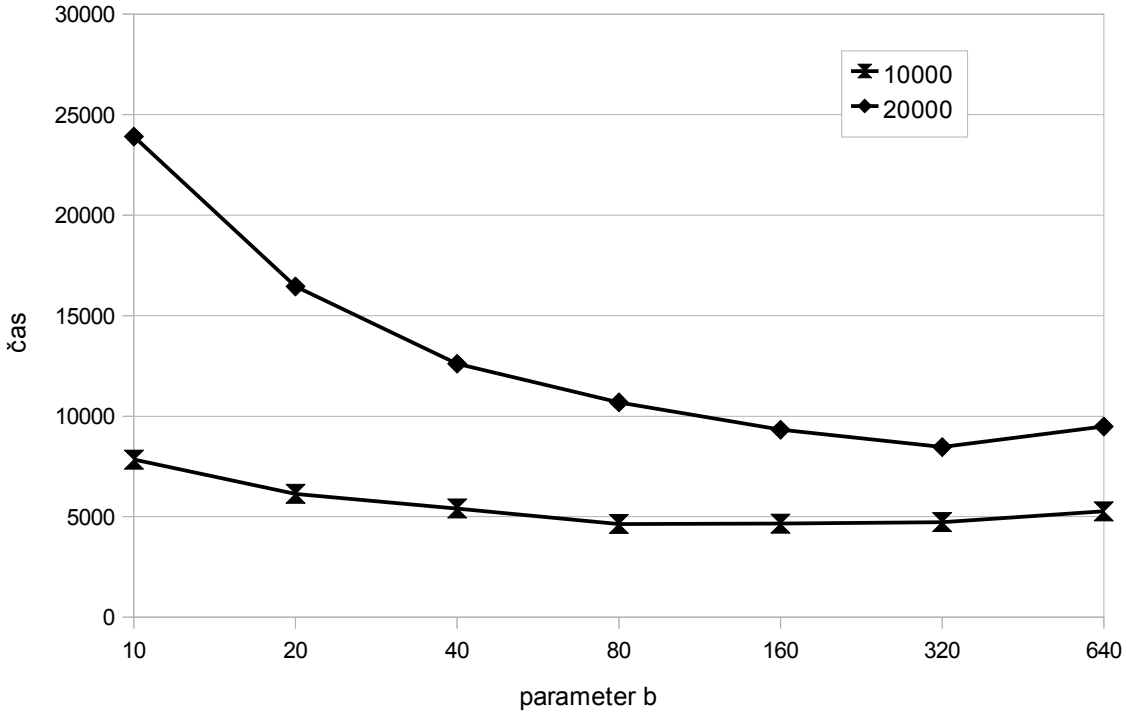
	10	20	40	80	160	320
3P-NRA	30587	15294	7648	3826	1910	957
TA - celkovo	30433	15240	7632	3830	1924	981
TA - priamy prístup	21222	10614	5310	2661	1335	675
TA - sekvenčný prístup	9221	4626	2322	1169	589	306

Z uvedeného experimentu vyplýva, že veľkosť vyrovnávacej pamäte vplýva na počet volaní webovej služby až do takej miery , že jej zdvojnásobením narastie množstvo komunikácie medzi klientom a serverom taktiež na dvojnásobok, čiže tieto dve veličiny sú priamo úmerné.

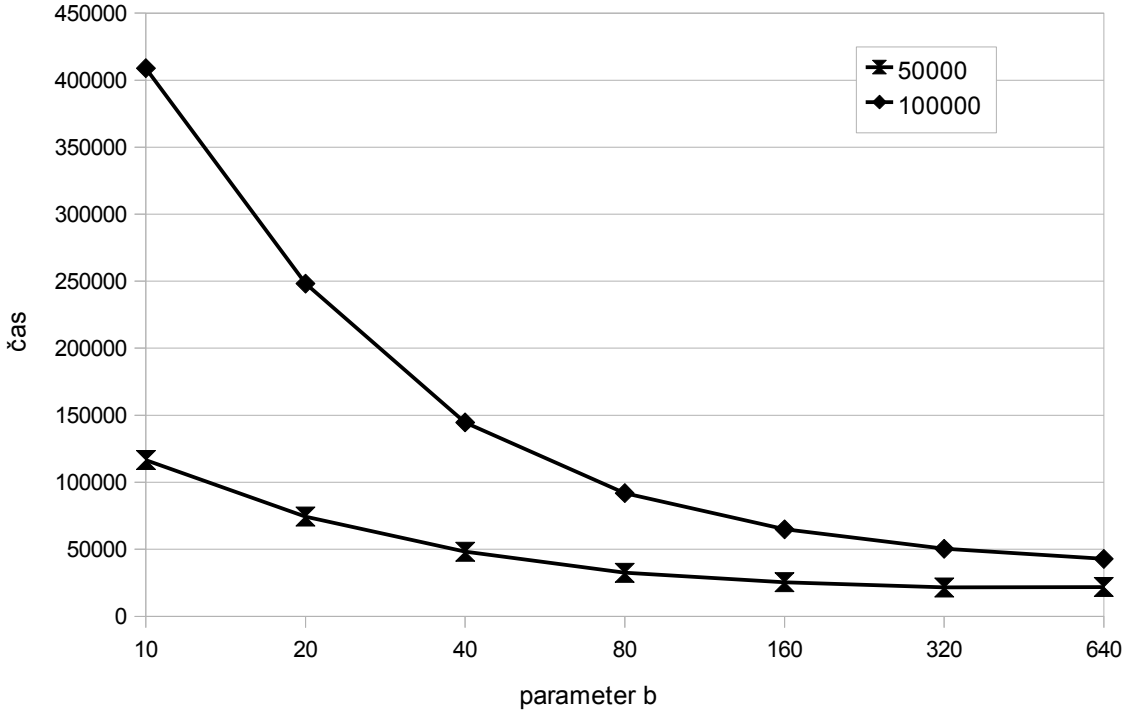
7.5 Testovanie vplyvu parametra *b*

Ako už bolo spomínané v časti 3.2.4 aj 7.2, parameter *b* takisto ovplyvňuje výkonnosť programu. Tento test ukazuje, akým spôsobom závisí celkový čas výpočtu a počet sieťových požiadaviek na tomto parametri.

Opäť je vyhľadávaných najlepších 10 prvkov, avšak tentokrát sa bude meniť ich celkový počet, ktorý bude nadobúdať hodnoty 10000, 20000, 50000 a 100000. Táto zmena bola vyžiadaná faktom, že výber vhodnej hodnoty tohto parametru závisí na veľkosti celkovej množiny prvkov. Latencia bola určená na 10 milisekúnd a veľkosť vyrovnávacej pamäte na 100 prvkov.



Graf 7.4: Závislosť času potrebného na nájdenie najlepších prvkov od parametru b pri nižších počtoch prvkov



Graf 7.5: Závislosť času potrebného na nájdenie najlepších prvkov od parametru b pri vyšších počtoch prvkov

Z grafov sa dá odpozorovať, že čím je celkový počet objektov vyšší, tým viac parameter *b* ovplyvňuje čas potrebný na nájdenie *k* najlepších prvkov. Rovnako je vidno aj fakt, že postupným zvyšovaním daného parametru 3P-NRA algoritmu dochádza od určitej hodnoty k javu, keď ďalšie zvyšovanie *b* už nijako neurýchľuje beh programu. Toto je zjavné hlavne z grafu 7.4, teda pri nižších celkových počtoch objektov.

Geometricky rastúci parameter *b* je v tomto experimente zvolený kvôli názornejšiemu poukázaniu na klesajúci pozitívny vplyv jeho zvyšovania pri vyšších hodnotách.

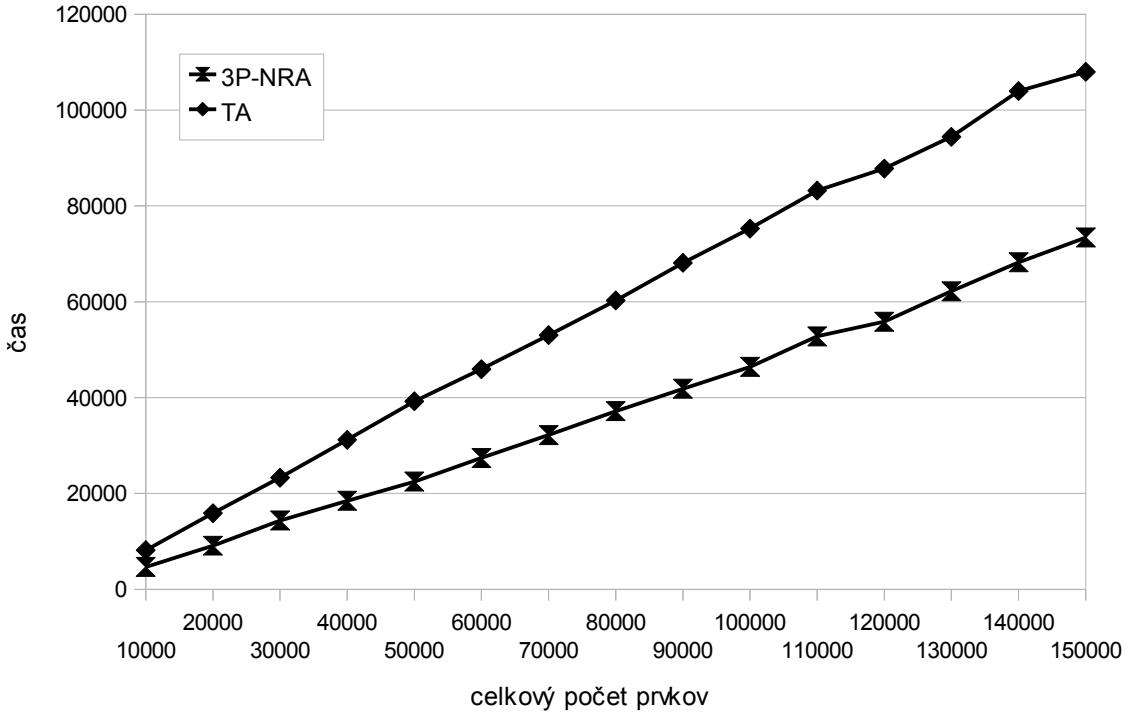
7.6 Porovnanie použitých algoritmov

Jedným z cieľov tejto práce je určenie algoritmu, ktorý je pre daný problém najvhodnejší. Rozhodnúť, ktorý algoritmus je lepší na nájdenie *k*-najlepších objektov podľa preferencií pre konkrétneho užívateľa nie je však až také jednoduché.

Prvým problémom je zvoliť kritérium, podľa ktorého sa budú tieto algoritmy porovnávať. Pre potreby tejto práce sa považuje za rozhodujúce kritérium čas potrebný na nájdenie správneho výsledku.

Nemenej dôležité je aj voľba podmienok, pri ktorých sú jednotlivé algoritmy porovnávané. Ako vyplýva z výsledku experimentu skúmajúceho závislosť času výpočtu od latencie siete, tak pri nižšej priepustnosti siete dosahuje algoritmus 3P-NRA lepšie výsledky, ako algoritmus TA. Existuje ale ešte jedna veličina, ktorá výrazne odlišne ovplyvňuje rýchlosť jednotlivých algoritmov. Jedná sa o celkový počet objektov, medzi ktorým je vyhľadávaných *k* najlepších.

Nasledujúci experiment je preto zameraný práve na závislosť veľkosti množiny všetkých objektov a času potrebného na výpočet jednotlivých algoritmov. Latencia siete bola nastavená na hodnotu 10 ms, veľkosť vyrovnávacej pamäte na 100 prvkov, parameter *N* na 100, *M* na 1000 a *b* na 500. Vyhľadávalo sa 10 najlepších prvkov.



Graf 7.6: Celkové porovnanie času výpočtu algoritmov pri rôznych počtoch objektov

Z experimentu je zrejmé, že lepších výsledkov dosahuje algoritmus 3P-NRA. Časy výpočtov jednotlivých algoritmov však rastú približne lineárne vzhľadom k počtu prvkov, takže majú rovnakú asymptotickú zložitosť.

Je nutné podotknúť, že situácia by bola výrazne odlišná pri nesprávnej voľbe parametra *b* pre algoritmus bez priameho prístupu (ak by bola jeho hodnota príliš nízka). V tom prípade by bol naopak algoritmus TA podstatne rýchlejší.

7.7 Zhrnutie výsledkov experimentov

Experimenty názorne ukázali dôležité vlastnosti použitých algoritmov a ich rozdielne správanie pri rôznych podmienkach.

Asi najpodstatnejší je poznatok, že pri vhodných parametroch je použitý algoritmus 3P-NRA rýchlejší ako TA. Je však nutné poznamenať, že priebeh výpočtu týchto dvoch algoritmov je značne rozdielny. Algoritmus bez priameho prístupu totiž výrazne spomaľujú použité dátové štruktúry a operácie s nimi, ako aj lokálny beh algoritmu, čo má za následok veľké vyťaženie hardware počítača, na ktorom beží klientská aplikácia. Naopak rýchlosť algoritmu TA záleží viac od aktívneho čakania na výsledok volania webovej služby, ktoré je pri priamom prístupe potrebné. Tento rozdiel sa dal odpozorovať jednak

z experimentu 7.3, ktorý skúma vplyv veľkosti latencie siete na jednotlivé algoritmy, a jednak zo sledovania vyťaženia systémových prostriedkov (hlavne procesora) v priebehu experimentov. Aspekt vyťažovania procesora bol viditeľný hlavne pri zvýšenej väčšej veľkosti vyrovnávacej pamäte.

Ďalším dôležitým výsledkom testov je, že preukázali veľmi pozitívny dopad vyrovnávacej pamäte na dĺžku výpočtu. Najmä čas potrebný na nájdenie k najlepších prvkov pomocou algoritmu 3P-NRA sa dá do určitej hranice výrazne vylepšiť zväčšením veľkosti tejto vyrovnávacej pamäte. To znamená, že spĺňa svoj účel a skutočne minimalizuje časovú réžiu spojenú s komunikáciou so vzdialenými servermi pomocou webových služieb pri sekvenčnom prístupe.

Zaujímavý je taktiež veľký vplyv na prvý pohľad takmer bezvýznamného parametra b. Jeho nesprávnou voľbou je možné prekvapivo výrazne negatívne ovplyvniť výkon algoritmu bez priameho prístupu.

8 Záver

Cieľom tejto práce bol návrh a implementácia systému, ktorý by bol vhodný na efektívne vyhľadávanie *k* najlepších objektov podľa užívateľských preferencií. Riešenie pritom malo byť použiteľné v sieťovom prostredí, kde dáta jednotlivých atribútov sú uložené na vzdialených serveroch a zároveň toto riešenie muselo byť nezávislé na konkrétnom užívateľovi.

V tejto práci boli navrhnuté a implementované dve rozdielne riešenia tohto problému. Prvé riešenie je postavené na algoritme 3P-NRA, ktoré využíva výhradne sekvenčný prístup k jednotlivým atribútom, naproti tomu základ druhého riešenia tvorí algoritmus TA používajúci ako sekvenčný, tak aj priamy prístup. Obe riešenia obsahujú indexovanie na strane servera s využitím B+ stromov, ktoré bolo upravené pre potreby distribuovaného prostredia a umožňuje podporu užívateľských preferencií.

Hlavným prínosom tejto práce je vytvorenie nových riešení distribuovaného vyhľadávania *k* najlepších prvkov. Nemenej prínosné je však aj experimentálne overenie funkčnosti tohto riešenia a porovnanie efektivity spomenutých dvoch prístupov. Z experimentov vyplynulo, že o niečo vhodnejšie riešenie je to založené na algoritme využívajúcom výhradne sekvenčný prístup k hodnotám atribútov a to hlavne vďaka dobrej predikovateľnosti behu algoritmu, ktorá je dôležitá pre neblokujúcu komunikáciu medzi klientskou a serverovou časťou aplikácie.

Vyvinuté riešenie bolo navrhované tak, aby bolo v budúcnosti čo najjednoduchšie rozšíriteľné. Pre tento účel je celá aplikácia implementovaná v jazyku Java a na sieťovú komunikáciu boli použité štandardné technológie, ako napríklad webové služby, SOAP, RPC a podobne. Za ďalší prínosom práce sa dá považovať implementácia náhodného generátora dát, ktorý bol používaný na testovacie účely.

Keďže problém, ktorý rieši táto práca je dosť komplexný, bolo by možné výskum rozšíriť v niekoľkých smeroch. V prvom rade by bolo vhodné aplikáciu doplniť spoľahlivým systémom pre riešenie všetkých možných sieťových problémov, akými sú výpadky jednotlivých serverov, ako aj ich rozdielne rýchle odozvy. Toto by bolo dobré riešiť vylepšením vyrovnávacej pamäte, ktoré by umožnilo rozdielny prístup k týmto serverom.

Ďalším možným smerom vývoja by mohlo byť zdokonalenie algoritmu využívajúceho priamy prístup tak, aby mohol do určitej miery predvídať, ktoré hodnoty bude v svojom ďalšom behu potrebovať k výpočtu, čím by bolo možné obmedziť aktívne čakanie na webové služby a výrazne tým tento algoritmus urýchliť. Takisto by bolo zaujímavé skúmať vplyv rôznych heuristík na výkon algoritmov vzhľadom k distribúcii a povahe testovacích dát.

9 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.

[5] P. Gurský: Towards Better Semantics in the Multifeature Querying. In Proc. Of the Dateso 2006, Czech Republic, April 26-28,2006. Edited by: Vaclav Snasel, Karel Richta, Jaroslav Pokorný Published on CEUR-WS: Vol-176

[6] Pokorný, J., Žemlička, M.: Základy implementace souborů a databází. Skripta UK, Vydavatelství Karolinum, 2004.

[7] Richta, K.: Standardy pro webové služby WSDL, UDDI. In: Moderní databáze 2003. Praha: Komix, 2003, s. 19-30. ISBN 80-239-0753-0

[8] W3C Working Group: Web Services Architecture http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/

[9] W3C Working Group: SOAP Version 1.2 Part 1: Messaging Framework (Second Edition) http://www.w3.org/TR/soap12-part1/

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
- `\data`
  - Obsahuje testovacie dáta
- `\pref_quer\client`
  - Obsahuje spustiteľnú aplikáciu na vyhľadávanie najlepších objektov
- `\pref_quer\server`
  - Obsahuje skompilovanú serverovú časť, ktorú je možné spustiť na príslušnom webovom serveri
- `\pref_quer\generator`
  - Obsahuje spustiteľný generátor dát
- `\install`
  - Obsahuje potrebné inštalačné súbory

B Užívateľská dokumentácia

Inštalácia a používanie aplikácie na vyhľadávanie *k* najlepších objektov podľa užívateľských preferencií, ktorej implementácia bola hlavným cieľom tejto práce, nemusí byť na prvý pohľad intuitívna. Preto je nutné popísať všetky úkony potrebné na spustenie tejto aplikácie.

B.1 Inštalácia

Aplikácia vyžaduje nainštalované JAVA prostredie na klientskom počítači, rovnako ako aj na všetkých serverových. Na počítačoch, na ktorých bude pustená serverová časť programu musí byť navyše nainštalovaný Tomcat.

B.1.1 Java

Na správnu funkčnosť aplikácie je potrebné nainštalovať JRE (Java Runtime Environment), čo je prostredie, ktoré slúži na spúšťanie Java programov. Toto prostredie je možné nainštalovať pomocou spustiteľného súboru, ktorý sa nachádza aj priamo na priloženom CD:

```
\install\java\jre-6u23-windows-i586.exe
```

Je to verzia určená pre operačný systém Windows, takže v prípade potreby spustenia aplikácie pod iným operačným systémom je nutné stiahnuť príslušnú verziu JRE na adrese: http://www.oracle.com/technetwork/java/javase/downloads/index.html .

B.1.2 Tomcat

Serverová časť aplikácie potrebuje k svojmu chodu aplikačný server Apache Tomcat verzie minimálne 6.0. Na jeho inštaláciu stačí rozbaliť archív, nachádzajúci sa na priloženom CD na miesto zvolené pre inštaláciu. Tento archív je možné nájsť na:

```
\install\tomcat\apache-tomcat-6.0.29-windows-x86.zip
```

Tento archív je určený pre operačný systém Windows, takže v prípade potreby je nutné stiahnuť verziu pre iný operačný systém na adrese: http://tomcat.apache.org/download-60.cgi .

B.1.3 Server

Na spustenie webovej služby je nutné serverovú časť aplikácie spustiť na aplikačnom serveri Tomcat. K tomuto kroku potrebujeme nahrať súbor

```
\pref_quer\server\PREF_QUER_WEB.war
```

do koreňového adresáru programu Tomcat. Cesta k tomuto adresáru bude mať nasledujúci tvar:

```
'ADRESAR_KDE_JE_NAINSTALOVANY_TOMCAT'\webapps\   .
```

Server musí byť počas tejto operácie vypnutý.

Serverová aplikácia musí mať konfiguračný súbor nakopírovaný adresári

```
c:\data .
```

Konfiguračný súbor sa nachádza na CD:

```
\pref_quer\server\ServerConfiguration.properties .
```

B.1.4 Klient a generátor dát

Na inštaláciu klientskej časti aplikácie a generátora dát stačí adresár

```
\pref_quer\client\
```

respektíve

```
\pref_quer\generator\
```

skopírovať na pevný disk.

B.2 Konfigurácia

Klientská aj serverová časť aplikácie sú konfigurovateľné pomocou konfiguračných súborov. Tieto súbory však majú kvôli odlišným povahám klienta a servera rozdielnu štruktúru.

B.2.1 Server

Konfigurácia serverovej časti je pomerne jednoduchá. Obsahuje len tieto tri parametre:

- *values* – cesta k súboru s dátami, čiže samotnými hodnotami atribútov.
- *latency* – umelá latencia siete. Tento parameter slúži na simulovanie nižšej priechodnosti siete.
- *b* – rád B-stromu, ktorý je v časti 4.1 označovaný *m*.

B.2.2 Klient

Konfigurácia klienta sa trochu líši podľa algoritmu, ktorý konfiguruje. Preto rozdelíme parametre na 3 skupiny: spoločné pre oba implementované algoritmy, parametre, ktoré využíva len algoritmus 3P-NRA a parametre pre algoritmus TA.

Spoločné parametre:

- *attributesNames* – názvy atribútov oddelené znakom „|“. Tento parameter zatiaľ nemá poriadne využitie, pretože pre účely experimentov boli používané len číselne identifikátory atribútov.
- *attributesServers* – adresy webových služieb pre jednotlivé atribúty oddelené znakom „|“. Jednotlivé adresy sú tvaru „Adresa_serveru:port/Cesta_k_webovej_službe_na_serveri“.
- *fuzzyFunctions* – cesty k súborom obsahujúcim fuzzy funkcie k jednotlivým atribútom oddelené znakom „|“.
- *algorithm* – názov algoritmu. Tento parameter môže nadobúdať dve hodnoty: *ThreePhaseNRA* pre algoritmus 3P-NRA a *TresholdAlgorithm* pre algoritmus TA.
- *agregationFunction* – cesta k súboru obsahujúcemu reprezentáciu agregačnej funkcie.
- *cacheSize* – veľkosť vyrovnávacej pamäte (jednotkou je počet prvkov).
- *statisticFilePath* – cesta k súboru, do ktorého sa uložia štatistiky výpočtu.

Parametre pre algoritmus 3P-NRA:

- *b* – parameter *b* popisovaný v 3.2.4 a 7.2.
- *heuristic1* – heuristika používaná v prvej fáze algoritmu. V experimentoch bola využívaná len heuristika cz.cuni.mff.pref_quer.alg.SimpleNRAHeuristic.

- *heuristic2* – heuristika používaná v tretej fáze algoritmu. V experimentoch bola využívaná len heuristika cz.cuni.mff.pref_quer.alg.SimpleNRAHeuristic.

Parametre špecifické pre algoritmus TA:

- *N* – parameter *N* popisovaný v 7.2.
- *M* – parameter *M* popisovaný v 7.2.
- *heuristic* – heuristika používaná algoritmom TA. V experimentoch bola využívaná len heuristika cz.cuni.mff.pref_quer.alg.SimpleNRAHeuristic.

B.3 Spustenie aplikácie

B.3.1 Server

Po správnom nainštalovaní serverovej časti aplikácie stačí spustiť Tomcat pomocou spustiteľného súboru

```
'ADRESAR_KDE_JE_NAINSTALOVANY_TOMCAT'\bin\tomcat6.exe .
```

Serverová časť sa v prípade potreby rozbalí z archívu a spustí spolu s aplikačným serverom. Pri každej zmene konfigurácie alebo testovacích dát je nutné server reštartovať, aby sa tieto zmeny prejavili.

B.3.2 Klient

Klient sa spúšťa z príkazového riadku príkazom

```
TopkLauncher K CESTA_KU_KONFIGURAČNÉMU_SÚBORU
```

v adresári, ktorý obsahuje klientskú aplikáciu, kde *K* je počet hľadaných objektov.

Príklad spustenia klienta:

```
TopkLauncher 10 .\conf\LauncherConfiguration3PNRA.properties
```

Klientskú aplikáciu je taktiež možno priamo spustiť pomocou spustiteľného súboru run_topk_3PNRA.bat respektíve run_topk_TA.bat a podľa potreby v ňom zmeniť parametre.

B.3.3 Generátor dát

Generátor testovacích dát sa spúšťa z príkazového riadku príkazom

```
DataGeneratorLauncher POČET_OBJEKTOV POČET_ATRIBÚTOV PRESNOSŤ CIEĽOVÝ_ADRESÁR
```

v adresári, v ktorom sa nachádza generátor dát.

Takže napríklad príkaz

```
DataGeneratorLauncher 100000 4 7 c:/data/
```

vygeneruje 4 súbory, z ktorých každý bude obsahovať 100000 hodnôt s presnosťou na 7 desatinných miest a nakopíruje ich do adresára c:/data/. Jednotlivé vygenerované súbory majú tvar „att_N_M.txt", kde *N* je poradové číslo atribútu a *M* počet hodnôt v tomto súbore.