BI-VWM

Matej Murín (murinmat), Václav Lokaj(lokajva1)

Deep Web

Prehľadávanie skrytej databáze

9.5.2020
Fakulta informačních technologií
České vysoké učení technické v Praze

Popis projektu

Cieľom našeho projektu je čo najlepšia rekonštrukcia databáze skrytej za daným uživateľským rozhraním. Máme teda k databáze obmedzený prístup a nie je nám umožnené listovať medzi všetkými výsledkami vyhovujúcim daným filtrom. Je to teda rozhranie vracajúce top-k výsledkov. Daná problematika je teda nasledujúca:

Akým spôsobom môcť za čo najrýchlejší čas získať čo najviac rôznych výsledkov?

Vstup
Vstupom našej implementácie popísanej v ďalšej sekcii je teda daná databáza a jej obmedzené rozhranie. Týmto spôsobom je nám umožnené rôznymi spôsobmi filtrovať dotazy nad danou databázou, a tá nám vráti top-k výsledkov. Dané rozhranie umožňuje taktiež radenie výsledkov, napríklad podľa ceny. Náš algoritmus má rôzne parametre, ktoré sú nastaviteľné pred spusteným samotného vyhľadávania. Prehľadávaná databáza predstavuje v našom priklade **e-shop** ponúkajúci počítače.

Vzorka prehľadávanej databáze

ComputerID	BrandID	MadeIn	Model	Price	ScreenSize	Type	Status
1001	3	5	Ronstring	3631	14	3	1
1002	1	5	Tempsoft	4281	32	1	3
1003	4	5	Alpha	2841	16	2	4
1004	6	9	Greenlam	3078	21	3	1
1005	7	3	Tempsoft	2855	13	3	3

Stĺpce **BrandID**, **MadeIn**, **Type** a **Status** sú foreign keys ukazujúce na ID daných atribútov. Pomocou týcho kľúčov je teda databáza schopná nájsť odpovedajúce reťazce v náležitých tabuľkách, tj **Brands, Countries, Types, Statuses.**

Výstup
Výstupom našeho programu je štatistika prehľadávania a súbor obsahujúci samotné výsledky. Tie sú vo formáte JSON, takže sú ľahko spracovateľné rozličnými nájstrojmi.

Spôsob riešenia

Náš algoritmus musí v prvom rade nejakým spôsobom komunikovať s databázou skrytou za daným rozhraním. Keďže ide o proof-of-concept, tak sme sa zhodli, že najjednoduchšou implementáciou pre komunikovanie s takouto databázou je nejaká trieda obaľujúca danú databázu. Zároveň má táto trieda implementované metódy, ktoré reprezentujú filtrovanie výsledkov. Napríklad metóda **addWantedBrand(String s)** pridá reťazec **s** ako vyžadovanú vlastnosť výsledku v stĺpci **brand**. Týmto spôsobom je teda simulované webové rozhranie typu **multi-select**. Rozhodli sme sa aj pre implementáciu osobitného webového rozhrania, pomocou ktorého je možné robiť rovnaké filtrovanie, aké umožňuje daná trieda. Týmto spôsobom je nám umožnené nahliadnuť na data v databáze v prijateľnej forme.

Samotný algoritmus prehľadávania je založený na metóde **getRandomSample().** Táto metóda postupne filtruje výsledky pomocou daného rozhrania. Nastavovanie parametrov si môžme predstaviť ako rozvetvovanie rozhodovacieho stromu, ktoré eventuelne skončí validným alebo prázdnym výsledkom. Keďže sa v databáze nachádzajú aj numerické hodnoty, náš algoritmus si pred spustením vie zistiť maximálne a minimálne hodnoty daných polí pomocou zoradenia výsledkov bez akýchkoľvek filtrov a vie ich v každom kroku diskrétne rozdeliť na dve časti. Tento spôsob sa dá chápať ako nastavenie atribútu, ktorý má 2 možnosti. Tento spôsob sme schopný aplikovať celkovo maximálne $\lceil \log_2(max - min + 1) \rceil$ krát. Ďalej nám algoritmus umožňuje dva typy radenia toho, aké atribúty sa budú kedy nastavovať.

- **Náhodné radenie**

  - v každom kroku “náhodne” rozhodne, aký filter sa bude nastavovať ako nasledujúci

  - pred každým spustením spustením sa vytvorí zoradený list daných nastavení filtrov s náhodnou prioritou, a tento list sa postupne prechádza

- **Fixné radenie**

  - fixné radenie je založené na preferovaní rozvetvovania rozhodovacieho stromu v rastúcom poradí

  - najväčšiu priority v každej chvíli budú mať teda atribúty, ktoré v danej situácii majú najmenšie rozvetvenie

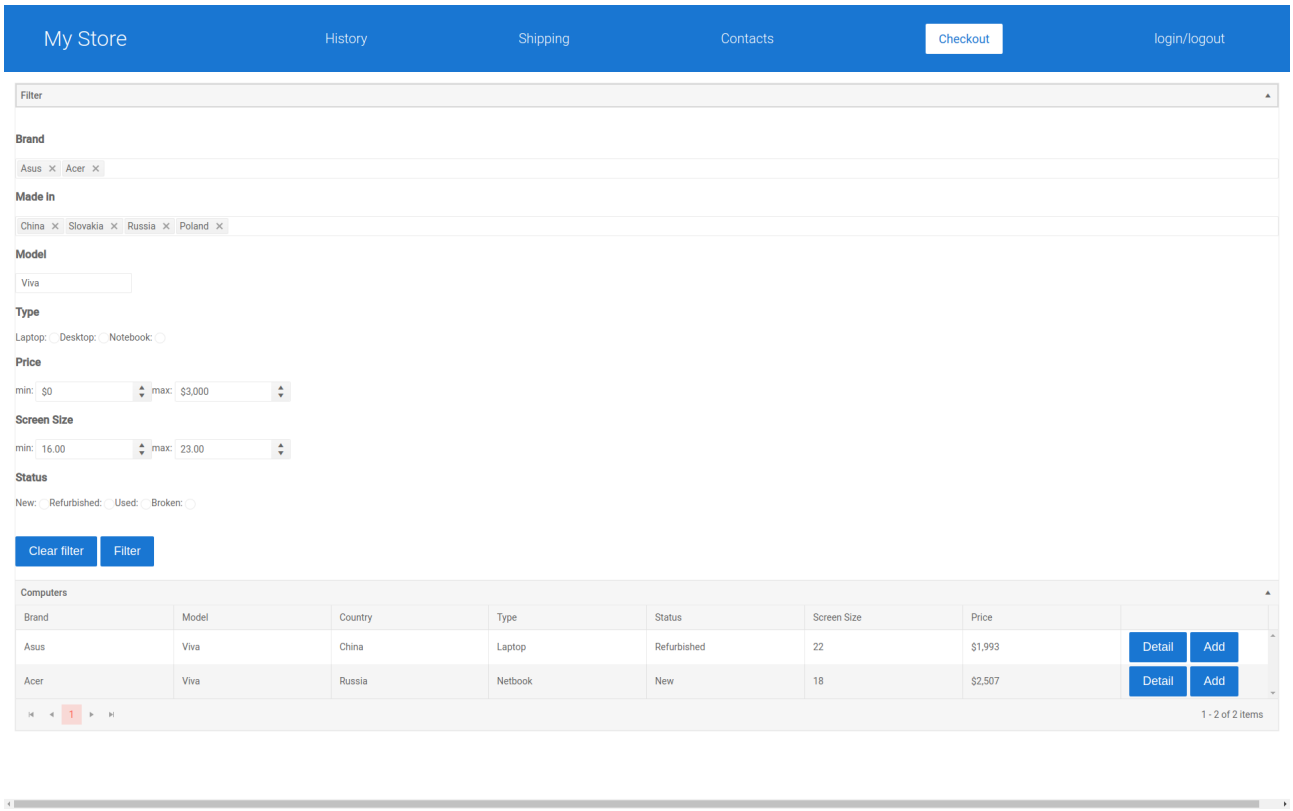
Nastáva však nasledujúci problém. Nie je dané, že distribúcia produktov v databáze je rovnomerná. Môže existovať atribút s relatívne nízkym rozvetvením, ktorých podstromy obsahujú podstatne menej validných výsledkov ako iné podstromy. Preto sa teda naša implementácia obsahuje takzvaný **acceptance factor**. Označme túto hodnotu **a(t).** Táto hodnota je vypočítaná ako

$$a(t) = \frac{currentAverageBranching^{currentDepth}}{overallAverageBranching^{overallAverageDepth - C}}$$

kde **C** je parameter nastaviteľný pred spustením programu. Existencia tohoto parametru nám umožňuje ovplyvňovať šance, s akými sú nájdené validné výsledky prijaté. Nastavenie **C** bude podrobnejšie preskúmané v experimentálnej sekcii. Hodnota **a(t)** je potom porovnaná s pseudo-náhodným **r**, ktoré je medzi 0 a 1. Pokiaľ **a(t) > r,** výsledok je prijaný. V opačnom prípade sa algoritmus **getRandomSample()** opakuje. Takýto prístup by nám mal priniesť čo najširšie množstvo produktov nájdených v databáze. Ako uvidíme v experimentálnej sekcii, sú prípady, kedy je preferované radenie atribútov náhodné, inokedy fixné.

Implementácia

Aby sme sa pri našom riešení čo najviac priblížili fungujúcemu obchodu, vystavili sme API napísané v **Jave,** konkrétne používajúce technológiu servlet. Taktiež sme navrhli webovu aplikaciu ktora pristupuje k databáze pomocou daného API a simuluje obchod. Webová aplikácia je napísaná vo frameworku JavaScriptu, **Angular 2** používajúca **Kendo UI** komponenty. Následujúci obrázok ukazuje filter zboží vo webovej aplikácii.



Pre databázu sme použili lokálnu **mysql** databázu. Na začiatku bola naplnená 1000 produktmi, no neskôr sme sa ju rozhodli zväčšiť na 10 000.

Samotná aplikácia ktorá vykonáva prehľadávanie databáze je napísaná v **Jave 8**. Objektovo-orientovaný jazyk nám prišiel ako správna voľba, keď že práve on nám umožňuje zapúzdrovanie rôznych funkcionalít. Presne týmto spôsobom sme obalili komukovanie s databázou do jednoduchého **API**, ktoré simuluje filtrovanie prístupné z webového rozhrania. Môžme si to predstaviť ako Java API poskytované daným obchodom. Jazyk bol taktiež zvolený pre jeho jednoduchú komunikáciu s **mysql** databázou pomocou **jdbc driveru**. Ďalej nám umožňuje uloženie skompilovaného kódu do **jar** súboru, ktorý je potom ľahko opakovane spustiteľný.

Zaznamenané výsledky sú potom exportované vo formáte **JSON**, ktoré sú dobre čitateľné ako pre ľudí, tak aj pre stroje.

Pre opakované spúšťanie aplikácie, zaznamenávanie výsledkov pre rôzne parametre pri spúšťaní a vizualizáciu štatisktík sme sa rozhodli používať jazyk **Python, verziu 3.7.6.** Využili sme ho s pomocou technológie **Jupyter Notebook**, vďaka ktorej sa jednoduchšie zaznamenávajú a analyzujú výsledky.

Príkľad výstupu

Na vzorovovom spustení programu sa nastaví maximálne trvanie aplikácie na **60 sekúnd**. Toto je horná hranica času, v ktorom nechávame program bežať. Po ubehnutí daného časového intervalu sa program zastaví, nehľadiac na výsledky. **Wanted sample** je počet výsledkov, ktoré od programu vyžadujeme. Pokiaľ teda prehľadávanie databáze nájde daný počet rôznych produktov, predčasne sa úspešne zastaví. Táto hodnota bola nastavená na **1000**. Parameter **C** bol nastavený na **3**, čo sa experimentálne zistilo byť celkom vysoká hodnota, takže väčšina validných nájdených výsledkov bude prijatá. Parameter **report-frequency** nastavuje dobu čakania v sekundách pre reporter, ktorý postupne ukladá počet doposiaľ získaných produktov. Týmto spôsom stačí pri experimentácii spustiť program na dlhšiu dobu a je nám dostupná aj informácia, za aký čas sa získalo koľko produktov. To všetko bez toho, aby sme museli aplikáciu spúšťať na rôzne časové intervaly. Táto hodnoda bola nastavená na **0.2**, čo nám dá 5 reportov na sekundu. Parameter –**output-stats** určuje názov výstupného súboru, do ktoréhu budú zaznamenané štatistiky z pokusu, ako aj parametre, pre ktoré bol program spustený. Parameter **request limit** je hodnota **k** pre **top-k** rozhranie zobrazovania výsledkov. V danom spustení bol nastavený na **5. Ordering** bol nastavený na **random**. Ako je vidieť, aplikácia bežala reálne **9.021 sekúnd.** Bola teda schopná za danú dobu nájsť požadované množstvo produktov.

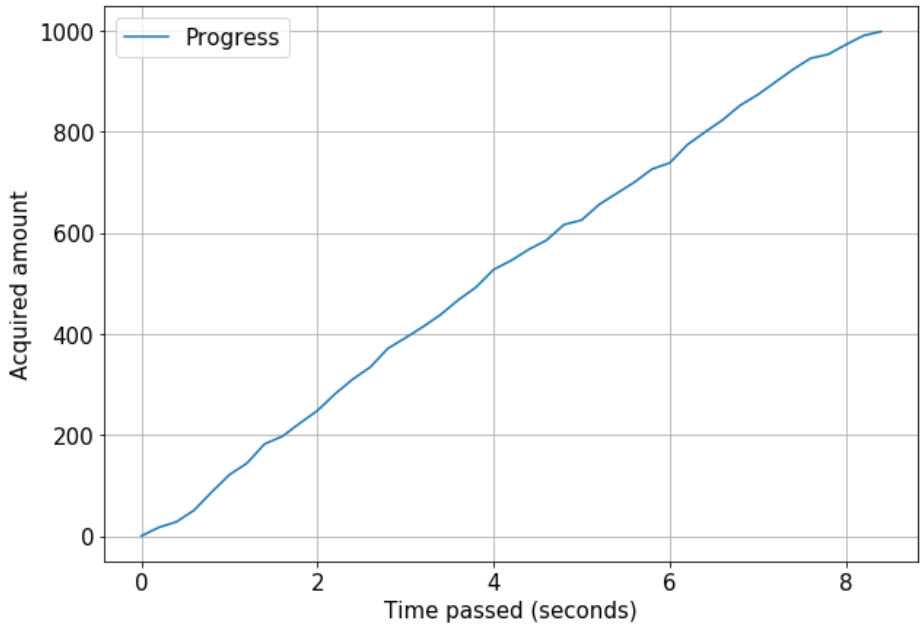
```
(base) gorq@gorq-Laptop:~/Desktop/VWM/walker_2.0/out/artifacts/walker_2_0_jar$ time java -jar walker_2.0.jar \
> --db-url localhost:3306 \
> --db-name market \
> --username gorq \
> --password hesloheslo \
> --timeout 60 \
> --wanted-sample 1000 \
> --set-c 3 \
> --report-frequency 0.2 \
> --output-stats test_stats.json \
> --ordering random \
> --request-limit 5 \
> --get-products test_products.json
Starting to work!
Starting overseer.
Got the wanted sample size!
Overseer got interrupted!
Working overseer stopping work.
Working ending.
Got 1001 samples.
Average depth: 11.615776
Unaccepted: 72

real    0m9,021s
user    0m4,244s
sys     0m0,235s
(base) gorq@gorq-Laptop:~/Desktop/VWM/walker_2.0/out/artifacts/walker_2_0_jar$
```

Zobrazenie výsledného reportu je dobre viditeľné, keď za prevedie do napr. **pandas** tabulky.

	Timeout	Limit	Ordering	C	Average depth	Average branching	Report frequency	Progress list	Acquired amount	Unaccepted amount
0	60	5	random	3.0	11.615776	3.445293	0.2	[17, 28, 51, 87, 121, 144, 182, 197, 223, 248,...	1001	72

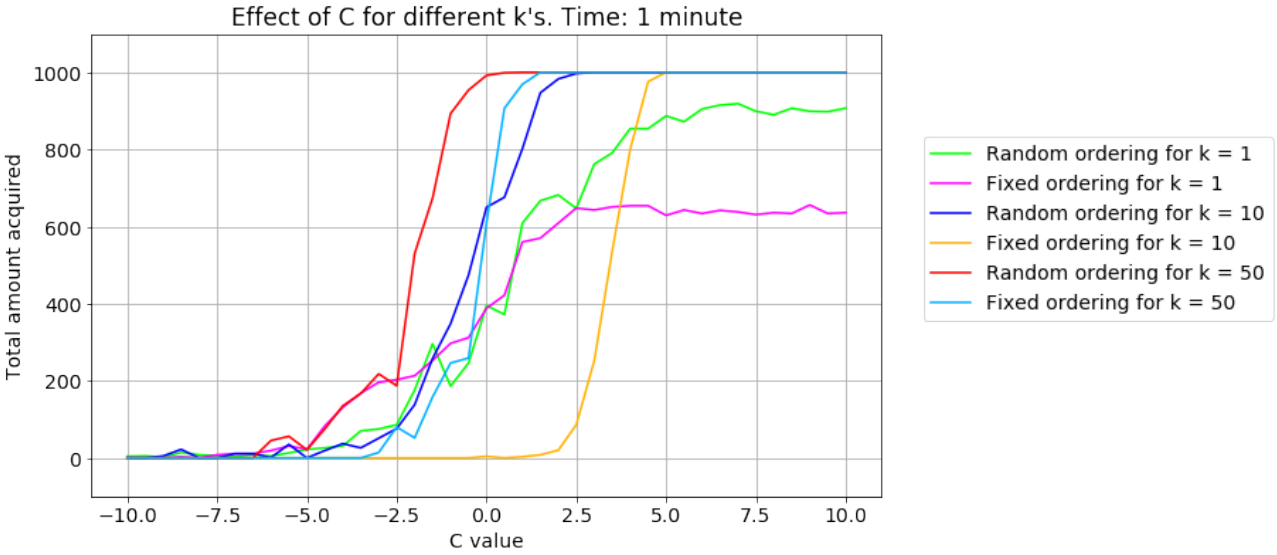
Taktiež obsahuje informácie o premernej hĺbke zanorenia v rozhodovacom strome, v ktorej boli nájdené validné výsledky, ako aj priemerné rozvetvenie rozhodovacieho stromu.

Report obsahuje taktiež **progress list,** v krotom je postupne zaznamenaný celkový počet získaných produktov. Keď že vieme, že reportov za sekundu je 5, vieme už jednoducho vykresliť tento progress relatívne k dobe behu programu.

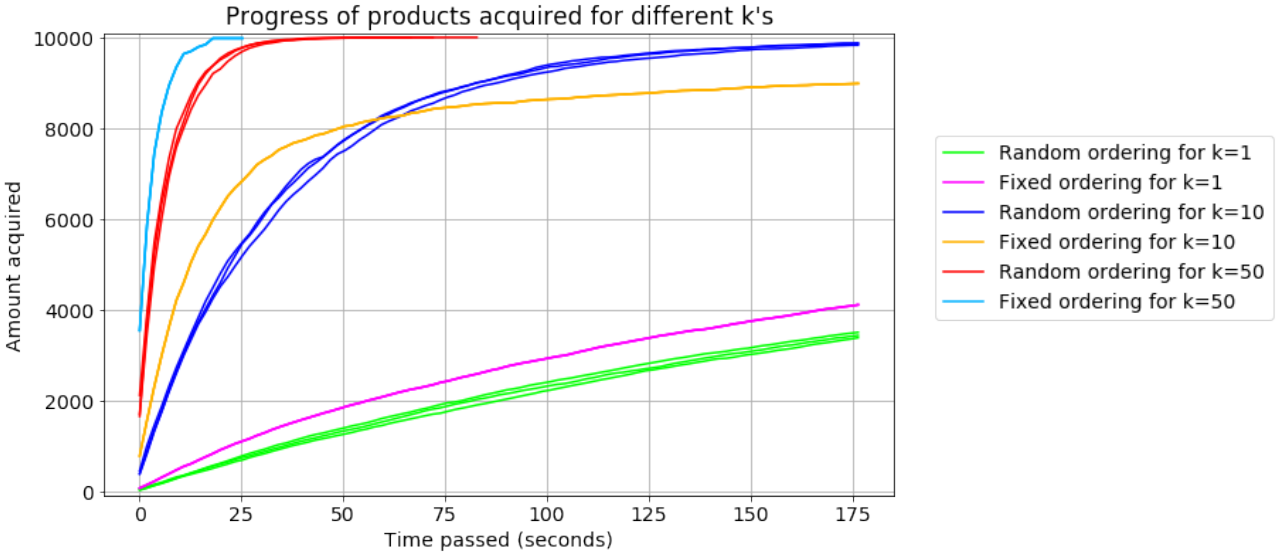


Experimentálna sekcia

V prvom rade bol preskúmaný efekt parametra **C** na celkový počet získaných produktov. Pripomeňme si, že C nám určuje, v akom pomere budeme prijímať, respektíve odmietať nájdené validné výsledky. Spustené experimenty vyhľadávali v databáze s **1000** produktmi. Ako je zobrazené na nasledujúcom grafe, optimálna hodnota **C** sa pohybovala okolo čísla **2.5.**



Zaujímavé bolo taktiež sledovať vplyv hodnoty **k** v **top-k** zobrazovaní výsledkov**.** Následujúci graf zobrazuje správanie algoritmu pre rôzne k, fixné aj náhodné radenie nastavovania atribútov, a pre **C = 5.** Tieto experimenty boli spustené naopak nad databázou s **10 000** produktimi, pre lepšie znázornenie správania.



Ako je vidieť, vplyv spôsobu radenia má značný vplyv na počet získaných produktov, ako aj rýchlosť ich získavania. Pre malé **k = 1** bolo preferované fixné radenie, a naopak pre **k = 10** onci viac produktov náhodné radenie. Určite by bolo zaujímavé detailne preskúmať, aké spôsoby radenia sú vhodné pre rôzne typy databáz.

Diskusia

Počas experimentovania s parametrami sa ukázalo, že najväčší vlpyv na správanie a výsledky programu majú parametre:

- **C**
  - parameter nám mal pomôcť s problématikou príliš nízkeho prijímania validných výsledkov pomocou vypočítania **acceptance factor**
  - čím višší bol C parameter nastavený, tým menej produktov bolo odmietnutých
  - experimentálne sa ukázalo, že hodnota **C** je vhodná niekde medzi **-1** až **6**
- **k in top-k**
  - tento parameter mal jasný vplyv na rýchlosť nachádzania výsledkov, čím vyššia hodnota K, tým bol algoritmus rýchlejšie schopný násjť produkty
  - po experimentácii sa ukázalo, že tento parameter hrá aj rolu pri spôsobe radenia atribútov
  - pre k=1 sa ukázalo, že fixné radenie je preferované, naopak pre k=10 dosahovalo lepšie výsledky náhodné radenie
- **Ordering**
  - spôsob radenie mal vplyv na rýchlosť nachádzania produktov, ako aj dostupnosť nájdenia nejakých produktov
  - počas experimentov bolo zistené, že fixné radenie je schopné nachádzať produkty rýchlejšie, avšak niektoré produkty sú mu potenciálne nedostupné práve z dôvodu fixného radenia

Typ databáze

Naša databáza, nad ktorou boli experimenty uskutočnené obsahovala pomerne rovnomerne rozptýlené dáta do atribútov. Bolo by určite zaujímavé preskúmať, ako sa algoritmus správa pri databázach rozložených rôznymi spôsobmi. Nerovnomerne rozložené dáta by mohli potenciálne priniesť problémy pri výpočte **acceptace factor.** V týchto prípadoch mohol teoreticky pomôcť, avšak optimálne hodnoty parametru **C** by pravdepodobne boli iné.

Vplyv radenia

V našej implementácii boli preskúmané dva typy radenia. Každé sa ukázalo mať určité výhody a nevýhody. Stálo by za to preskúmať aj iné typy radenia, ako napríklad fixné, ktoré by ale bolo založené na inom princípe, ako velkosť rozvetvovania. Napríklad by sa do tohto spôsobu radenia mohla zaviesť nejaká randomizácia. Jeden spôsob by mohol byť náhodné pričítanie alebo odčítanie pseudonáhodného čísla v nejakej modularite, aby sa atribúty s podobnou velkosťou rozvetvovania raz za čas prestriedali.

Záver

Počas práce sme preskúmavali spôsoby, akými je možné prehľadávať databáze, ktoré sú pre nás skryté za istou formou uživateľského rozhrania. Boli predvedené spustenia nami popísaného riešenia pre rôzne parametre. Ukázalo sa, že zjavne najväčší vlpyv na celkový počet získania informácií o databáze je počet prvkov, ktoré nám dané rozhranie poskytuje (**k** in **top-k** style).