2 Reguli de asociere şi mulţimi frecvente de articole

Problema coşului de produse presupune că avem un mare număr de articole, e.g. "pâine", "lapte". Cumpărătorii pun în coşul lor de produse anumite submulţimi de articole iar noi vom afla ce articole sunt cumpărate împreună chiar dacă nu şi de cine. Vânzătorii utilizează aceste informaţii pentru aşezarea articolelor în rafturi şi controlează modul în care un cumpărător tipic traversează magazinul.
În afara utilizării în marketing, acelaşi tip de problemă are următoarele utilizări:

1. Coş = documente; articole = cuvinte. Cuvintele aparând frecvent împreuna în documente pot reprezenta fraze sau concepte legate între ele. Poate fi utilizat pentru colectarea de informatii (intelligence).
2. Coş = propoziţii, articole = documente. Două documente conţinând aceleaşi propoziţii pot reprezenta un plagiat sau "mirror sites" pe web.

2.1. Obiective pentru căutarea în coşuri de produse

1. *Regulile de asociere* sunt propoziţii de forma $\{X_1, X_2, \ldots, X_n\} \Rightarrow Y$ însemnând că dacă găsim toate $X_1, X_2, \ldots, X_n$ în coş atunci sunt mari sanse să-l găsim şi pe Y. Probabilitatea de a-l găsi pe Y pentru a accepta această regulă este numită *încrederea* respectivei reguli. În mod normal vom căuta doar reguli care au o încredere peste un anumit prag. Putem de asemenea cere ca încrederea să fie semnificativ mai mare decât în cazul în care articolele ar fi plasate aleator în coş. De exemplu putem găsi o regula ca {*lapte, unt*} $\Rightarrow$ *pâine* doar pentru că foarte multă lume cumpără pâine. Totuşi exemplul bere/scutece arată că regula {*scutece*} $\Rightarrow$ {*bere*} este verificată cu o încredere semnificativ mai mare decât a submulţimii de coşuri continând bere.
2. *Ccauzalitate.* Ideal, am vrea să ştim dacă într-o regulă de asociere prezenţa elementelor $X_1, X_2, \ldots, X_n$ efectiv "cauzează" (determină) cumpararea lui Y. "Cauzalitatea" este însă un concept echivoc. Cu toate acestea, pentru datele de tip coş de produse următorul test arată ce înseamnă cauzalitatea. Dacă scădem preţul scutecelor şi creştem preţul berii putem ademeni cumpărătorii de scutece care au înclinaţia de a cumpăra bere din magazin, acoperind astfel pierderile din vânzarea scutecelor. Această strategie este valabilă deoarece "scutece determină bere". Acţiunea inversă, micşorarea preţului la bere şi mărirea preţului scutecelor nu va determina cumpărătorii de bere să cumpere scutece în număr mare şi vom pierde bani.
3. *Mulţimi frecvente de articole (frequent itemsets).* În multe situaţii (dar nu în toate) ne interesează doar regulile de asociere şi cauzalitatea în ceea ce priveste mulţimi de articole care apar frecvent în coşuri. De exemplu, nu putem conduce o bună strategie de marketing care implică produse pe care oricum nu le cumpără nimeni. Astfel, extragerea de cunoştinţe din date porneşte de la premiza că ne interesează doar mulţimile de articole cu un larg suport; i.e., ele apar împreună în multe coşuri de produse. Găsim apoi regulile de asociere sau cauzalităţile implicând doar articolele cu larg suport (i.e. $\{X_1, X_2, \ldots, X_n, Y\}$ trebuie să apară în cel puţin un anumit procent din coşuri, numit *prag de suport*)

2.2. Cadru pentru căutarea mulţimilor frecvente de articole

Utilizăm termenul *mulţimi frecvente de articole (frequent itemsets)* pentru "o mulţime de articole S care apare în cel puţin a "s"-a parte din coşuri", unde s este o constantă aleasă, de obicei 0.01 sau 1%.

Vom presupune că avem o cantitate de date care nu încape în memoria centrală a calculatorului. Fie sunt stocate într-o bază de date relaţională (BDR), de exemplu o relaţie (tabelă) *Coşuri(IdCoş, articol)* fie ca un fişier de înregistrări de forma *(ICoş, articol1, articol2, …, articol-n)*. Când evaluăm timpul de rulare al algoritmilor:

- Numărăm trecerile prin date. Deoarece costul principal este dat adesea de timpul necesar citirii datelor de pe disc, numărul de citiri necesar pentru fiecare dată este adesea cea mai bună masură a timpului de rulare al algoritmului.

Există un principiu cheie, numit *monotonicitate* (monotonicity) sau *principiul a-priori* care ne ajută să găsim mulţimile frecvente de articole:

- Dacă o mulţime de articole S este frecventă (i.e., apare cel puţin în a "s"-a parte a coşurilor), atunci orice submulţime a lui S este de asemenea frecventă.

Pentru a găsi mulţimile frecvente de articole:

1. Procedăm nivel cu nivel, găsind întâi articolele frecvente (mulţimi de dimensiune 1), apoi perechile frecvente, tripletele frecvente, etc. În discuţia noastră ne vom concentra pe găsirea perechilor frecvente deoarece:
   a. Adeseori perechile sunt suficiente.
   b. În multe mulţimi de date partea cea mai grea este găsirea perechilor; continuarea pe nivelele superioare necesită mai puţin timp decât găsirea perechilor frecvente.

   Algoritmii de acest tip utilizează o trecere per nivel.

2. Găsim toate *mulţimile frecvente de articole maximale* (i.e., mulţimile S a.i. nici o mulţime care include strict pe S nu este frecventă) într-o singură trecere sau în câteva treceri.

2.3. Algoritmul a-priori

Acest algoritm procedează nivel cu nivel.

1. Dându-se pragul de suport *s*, în prima trecere găsim articolele care apar în cel puţin a "*s*"-a parte a coşurilor. Această mulţime este notată L_1, mulţimea articolelor frecvente.
2. Perechile de articole din L_1 devin mulţimea C_2 a *perechilor candidate* pentru a doua trecere. Sperăm că dimensiunea lui C_2 nu este atât de mare pentru că altfel nu este suficient spaţiu în memoria centrală pentru un contor numeric întreg al apariţiei fiecărei perechi. Perechile din C_2 al căror contor ajunge sau depăşeşte *s* formează mulţimea L_2 a perechilor frecvente.
3. Tripletele candidate C_3 sunt mulţimile $\{A, B, C\}$ pentru care $\{A, B\}$, $\{A, C\}$ şi $\{B, C\}$ sunt în L_2. În a treia trecere sunt numărate apariţiile tripletelor din C_3; cele al caror contor este cel puţin *s* formează mulţimea L_3 a tripletelor frecvente.
4. Se poate merge oricât de departe se doreste (sau până mulţimile devin vide). L_i conţie mulţimile frecvente de articole de dimensiune i; C_{i+1} este mulţimea candidatelor de dimensiune $i+1$ a.i. fiecare submulţime a lor de dimensiune i este inclusă în L_i.

2.4. De ce este util principiul a-priori

Să considerăm urmatoarea cerere SQL pe o relaţie (tabelă) *Coşuri(IdCoş, articol)* având 10^8 tupluri care conţin date despre 10^7 coşuri de câte 10 articole fiecare. Presupunem existenţa a 100.000 articole diferite (tipic pentru o reţea de magazine ca Wal-Mart de exemplu).

```
SELECT b1.articol, b2.articol, COUNT(*)
FROM Coşuri b1, Coşuri b2
WHERE b1.IdCoş = b2.IdCoş AND b1.articol < b2.articol
GROUP BY b1.articol, b2.articol
HAVING count(*) >= s;
```

Notă: *s* este pragul de suport iar al doilea termen al clauzei WHERE elimină perechile formate din acelaşi produs şi apariţia de două ori a aceleiaşi perechi.

În joinul *Coşuri* ⋈ *Coşuri* fiecare coş contribuie cu $C^2_{10} = 45$ de perechi astfel încât joinul are $4{,}5 \times 10^8$ tupluri. A-priori "împinge clauza HAVING în jos pe arborele expresiei", determinându-ne în primul rând să înlocuim *Coşuri* cu rezultatul cererii:

```
SELECT *
FROM Coşuri
GROUP BY articol
HAVING COUNT(*) >= s;
```

Dacă s = 0,01 atunci cel mult 1000 de grupuri de articole pot trece de clauza HAVING. Motivul: în relatia Coşuri sunt 10^8 linii conţinând articole iar fiecare articol are nevoie de 0,01 x 10^7 = 10^5 dintre acestea pentru a apărea în 1% din coşuri.

- Deşi 99% dintre articole sunt eliminate de algoritmul a-priori nu trebuie să concluzionăm că relaţia *Coşuri* care rezultă are doar 10^6 tuple. În fapt, *toate* tuplele pot fi pentru prodse cu larg suport. Totuşi, în situaţiile reale, micşorarea relaţiei *Coşuri* este substanţială şi dimensiunea joinului scade cu pătratul acestei micşorări.

2.5. Îmbunătăţiri ale algoritmului a-priori

Sunt de două tipuri:

1. Micşorarea dimensiunii mulţimilor candidat C_i pentru i ≥ 2. Această opţiune este importantă chiar şi pentru găsirea perechilor frecvente deoarece numărul de elemente trebuie să fie suficient de mic pentru ca un contor de apariţii pentru fiecare să fie ţinut în memoria centrală.
2. Contopirea încercărilor de găsire a L_1, L_2, L_3, … în doar una sau două treceri în loc de o trecere per nivel.

2.6. Algoritmul PCY

Park, Chen şi Yu au propus, utilizând o tabelă de dispersie, să determine la prima trecere (când este calculat L_1) că multe perechi nu sunt perechi frecvente posibile. Profită de faptul că memoria centrală este uzual *mult* mai mare decât numărul de articole. În timpul celor două faze pentru găsirea lui L_2 memoria centrală este ocupată ca în Fig. 1.

Figura 1: Două treceri ale algoritmului PCY

Presupunem că datele sunt stocate într-un fişier de înregistrari constând dintr-un identificator IdCoş şi o listă cu articolele sale.

1. Trecerea 1:
   a. Se număra apariţiile fiecarui articol.
   b. Pentru fiecare coş constând din articolele $\{i_1, \ldots, i_k\}$, se aplica funcţia de dispersie fiecărei perechi asociind-o unei intrări a tabelei de dispersie şi se incrementează contorul acesteia cu 1.
   c. La sfârşitul trecerii, se determină L_1, articolele cu contorul cel puţin s.
   d. De asemenea la sfârşitul trecerii se determină acele intrări ale tabelei de dispersie cu contor cel puţin s.
      - Punct cheie: o pereche (i, j) nu poate fi frecventă decât dacă este dispersată într-o intrare frecventa astfel încât perechile care sunt dispersate în alte intrări nu vor fi candidate în C_2.

      Se înlocuieşte tabela de dispersie cu un bitmap având un bit per buchet: 1 dacă buchetul a fost frecvent, 0 altfel.

2. Trecerea 2:
   a. Memoria centrală conţine o listă cu toate articolele frecvente, i.e. L_1.
   b. Tot memoria centrală conţine un bitmap reprezentând rezultatele dispersiei din prima trecere.
      - Punct cheie: intrările tabelei de dispersie utilizează 16 sau 32 de biţi pentru un contor dar sunt comprimate la un singur bit. Astfel, chiar dacă tabela de dispersie ocupă aproape întreaga memorie centrală la prima trecere, bitmapul său nu ocupă mai mult de 1/16 din memoria centrală la trecerea 2.
   c. În final, memoria centrală conţine de asemenea o tabelă cu toate perechile candidat şi contorii asociaţi lor. O pereche *(i, j)* poate fi candidată în C_2 doar dacă *toate* condiţiile următoare sunt adevărate:
      i. *i* este în L_1.
      ii. *j* este în L_1.
      iii. *(i, j)* este dispersată într-o intrare frecventă

      Ultima condiţie diferenţiază PCY de a-priori clasic şi reduce necesarul de memorie în trecerea 2.
   d. În timpul trecerii 2 luăm în considerare fiecare coş şi fiecare pereche de articole din el, efectuând testul de mai sus. Dacă o pereche îndeplineşte toate cele trei condiţii, se incrementează contorul acesteia din memorie sau se crează unul dacă acesta nu exista încă.

- Când este mai performant PCY decât a-priori? Când sunt prea multe perechi de articole din L_1 pentru a încăpea într-o tabelă de perechi candidate şi de contori asociaţi în memoria centrală iar numărul de intrări frecvente ale tabelei de dispersie din algoritmul PCY este suficient de mic pentru a reduce dimensiunea lui C_2 suficient pentru a încăpea în memoria centrală (chiar şi fără 1/16 din ea consumată de bitmap).
- Când o mare parte a intrărilor tabelei de dispersie nu vor fi frecvente în PCY? Când sunt puţine perechi frecvente iar cea mai mare parte a perechilor sunt atât de puţin frecvente încât chiar dacă contorii tuturor perechilor care sunt dispersate într-o aceeaşi intrare sunt adunaţi nu sunt mari şanse să se obţină o valoare egală sau mai mare ca *s*.

2.7. Extensia "Iceberg" a PCY

1. *Tabele de dispersie multiple*: se împarte memoria între două sau mai multe tabele de dispersie, ca în Fig. 2. La trecerea 2 se reţine în memorie câte un bitmap pentru fiecare dintre acestea; de notat că spaţiul necesar pentru aceste bitmapuri este exact acelaşi cu cel necesar în bitmapul unic din PCY deoarece numărul total de intrări reprezentate este acelaşi. Pentru a fi candidata la C_2 o pereche:
   a. Constă din articole din L_1, şi
   b. Este dispersată intr-o intrare frecventă în *fiecare* tabelă de dispersie.

Figura 2: Utilizarea memoriei pentru tabele de dispersie multiple

2. Tabele de dispersie iterate *Multistage:* În locul verificării candidatelor în trecerea 2 se crează o altă tabelă de dispersie (altă funcţie de dispersie) în trecerea 2 dar sunt dispersate doar acele perechi care îndeplinesc conditiile pentru PCY; i.e., ambele articole sunt din L_1 şi sunt dispersate într-un buchet frecvent în prima trecere. În a treia trecere, păstrăm câte un bitmap pentru fiecare tabela de dispersie şi tratăm o pereche ca o candidată din C_2 doar dacă:

    a. Ambele articole sunt în L_1.
    b. Perechea a fost dispersată într-o intrare frecventă în prima trecere.
    c. Perechea a fost dispersată de asemenea într-o intrare frecventă în trecerea 2.

Figura 3 arată utilizarea memoriei. Aceasta schemă poate fi extinsă la mai multe treceri, dar există o limită deoarece eventual memoria devina plină de bitmapuri şi nu mai putem contoriza perechile candidat.

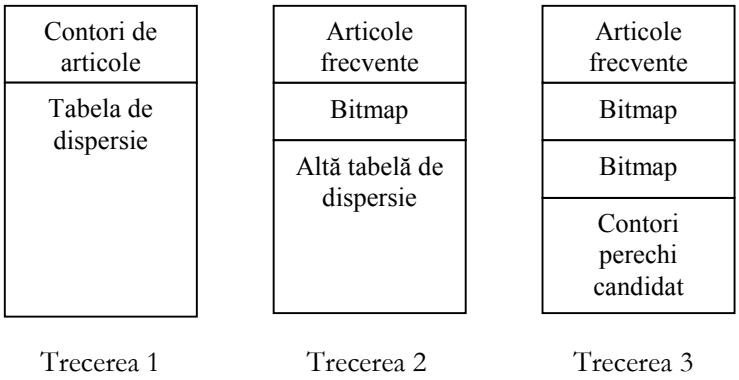

Figura 2: Utilizarea memoriei pentru tabele de dispersie Multistage

- Când sunt utile tabelele de dispersie multiple? Când cele mai multe dintre intrările tabelei de dispersie de la prima trecere a PCY au contori cu mult sub pragul *s*. Atunci putem dubla contorii intrărilor şi totuşi cele mai multe dintre ele vor fi sub prag.
- Când sunt utile tabelele de dispersie Multistage? Când numărul intrărilor frecvente din prima trecere este mare, dar nu toate (e.g. 50%). Atunci, a doua dispersie cu unele dintre perechile ignorate poate reduce numărul de intrări frecvente semnificativ.

2.8. Toate mulţimile frecvente de articole în două treceri

Metodele de mai sus sunt cele mai bune când dorim perechile frecvente, cazul cel mai comun. Dacă dorim toate mulţimile frecvente maximale de articole, inclusiv mulţimi mari, sunt necesari prea mulţi paşi. Există mai multe abordări pentru obţinerea tuturor mulţimilor frecvente de articole în două treceri sau mai puţin. Fiecare se bazează într-un fel sau altul pe faptul că datele sunt aleatoare.

1. *Abordarea simplă:* Se ia un eşantion de date de dimensiunea memoriei centrale. Se rulează un algoritm nivel cu nivel în memoria centrală (deci nu sunt costuri de I/O) şi se speră că eşantionul ne va conduce la adevăratele mulţimi frecvente.
    - De notat că pragul *s* trebuie scalat prin micşorare; e.g. dacă eşantionul este de 1% din date, se utilizează s/100 ca prag de suport.
    - Se poate face o trecere completă prin date pentru a verifica dacă mulţimile frecvente de articole ale eşantionului sunt cu adevărat frecvente, dar vor fi pierdute mulţimile de articole care sunt frecvente în ansamblul datelor dar nu şi în eşantion.
    - Pentru a minimiza falsele negative se poate scădea puţin pragul pentru eşantion găsindu-se mai multe candidate pentru trecerea prin ansamblul datelor. Risc: prea multe candidate pentru a încăpea în memoria centrală.

2. *SON95* (Savasere, Omniecinski şi Navathe în VLDB 1995; referit de Toivonen). Se citesc submulţimi ale datelor în memoria centrală aplicându-se abordarea simplă pentru descoperirea mulţimilor candidat. Fiecare coş este parte a uneia dintre aceste submulţimi. În a doua trecere o mulţime este candidată dacă a fost identificată ca şi candidată în una sau mai multe submulţimi ale datelor.
   - Punct cheie: O mulţime nu poate fi frecventă în ansamblul datelor dacă nu este frecventă în cel puţin o submulţime a acestora.

3. *Algoritmul lui Toivonen:*

   a. Se ia un eşantion care încape în memoria centrală. Se rulează abordarea simplă pe aceste date dar cu un prag micşorat astfel încât sa fie improbabilă pierderea vreunei adevărate mulţimi frecvente de articole (e.g. dacă eşantionul este de 1% din date se foloseşte s/125 ca prag de suport).
   b. Se adaugă candidatelor din eşantion *marginea negativă:* acele mulţimi de articole *S* astfel încât *S nu* este identificată ca frecventă în eşantion dar *orice* submulţime strictă maximală a lui *S* este identificată astfel. De exemplu, dacă *ABCD* nu este frecventă în eşantion dar *ABC, ABD, ACD* şi *BCD* sunt frecvente în eşantion, atunci *ABCD* este în marginea negativă.
   c. Se face o trecere prin date, numărând toate mulţimile frecvente de articole şi marginea negativă. Dacă nici o mulţime din marginea negativă nu este frecventă în ansamblul datelor, atunci mulţimile frecvente de articole sunt exact acele candidate care sunt deasupra pragului.
   d. Din pacate, dacă există o mulţime din marginea negativă care devine frecventă, atunci nu stim dacă vreuna dintre mulţimile care o contin nu este de asemenea frecventă, astfel încât întregul proces trebuie sa fie repetat (sau acceptăm că există şi nu ne interesează câteva fals negative).