DESCOPERIREA TIPARELOR FRECVENTE FĂRĂ GENERARE DE CANDIDAȚI

Cornelia GYÖRÖDI, **Robert GYÖRÖDI**

Departmentul de Calculatoare, Facultatea de Electronică şi Informatică,
Universitatea Oradea, România.
Tel: +40 (0) 259 432830 int.: 226 Fax: +40 (0) 259 432789
E-mail: rgyorodi@rdsor.ro

Rezumat: Informaţia implicită din bazele de date, în principal relaţiile interesante de asociere dintre seturi de obiecte, care conduc la formarea de reguli de asociere, ar putea scoate în evidenţă tipare folositoare în activităţile de suport al deciziilor, previziuni financiare, politici de marketing, chiar şi efectuarea de diagnostice medicale precum şi în multe alte aplicaţii. Acest lucru a atras multă atenţie în cercetările recente din domeniul data mining [1]. Aşa cum este arătat în [2], descoperirea regulilor de asociere ar putea necesita scanări iterative ale unor baze mari de date, lucru care este costisitor din punct de vedere al timpului de procesare. Mulţi autori şi-au concentrat cercetările asupra problemei descoperirii eficiente a regulilor de asociere din bazele de date [2], [3]. Un algoritm de descoperire a regulilor de asociere, foarte influent, A priori [2], acest algoritm a fost dezvoltat pentru descoperirea regulilor de asociere, din baze de date tranzacţionale de dimensiuni mari, bazat pe generarea de candidaţi şi eliminarea celor care nu îndeplineau criteriile de *suport* şi *încredere* impuse. Un pas important în îmbunătăţirea performanţelor acestor algoritmi a fost făcut prin introducerea unei structuri de date compacte, numită *arbore de tipare frecvente* sau FP-tree [3], şi a algoritmilor de descoperire asociaţi, FP-growth [3], care se baza pe descoperirea tiparelor frecvente fără generarea unor candidaţi suplimentari. Metoda introdusă de autor în [6] şi [7], numită DynFPGrowth, îmbunătăţeşte performanţele algoritmilor bazaţi pe FP-tree, prin reducerea numărului de scanări de la două la una singură şi prin introducerea unui proces de actualizare dinamică a structurii FP-tree. Lucrarea prezintă, de asemenea, folosirea cadrului de dezvoltare şi de comparare a algoritmilor de descoperire a regulilor de asociere, introdus în [8], în vederea validării metodei introduse şi pe baza datelor obţinute în urma unui studiu de performanţă, care arată avantajele metodei introduse.

Cuvinte cheie: tipare frecvente, reguli de asociere, algoritmi, A priori, FP-Growth, DynFPGrowth

1. Introducere

Descoperirea regulilor de asociere are ca scop aflarea unui set de atribute comune, care aparţin unui număr mare de obiecte dintr-o bază de date. Descoperirea regulilor frecvente de asociere dintr-o bază de date, de dimensiuni mari, este o problemă complexă deoarece spaţiul de căutare creşte exponenţial cu numărul de atribute din baza de date şi cu obiectele bazei de date. Extragerea tiparelor frecvente joacă un rol esenţial în descoperirea regulilor de asociere şi a multor altor activităţi importante din domeniul minării datelor.

Cele mai multe din studiile anterioare, prezentate în [2], [4], [5] au adoptat o abordare bazată pe algoritmul A priori, care este întemeiat pe o euristică A priori antimonotonă [2]: *Dacă oricare tipar de lungime k nu este frecvent în baza de date, atunci supertiparul său de lungime (k+1) nu poate fi frecvent niciodată.* Ideea esenţială este de a genera, în mod iterativ, setul de tipare candidate de lungime (k+1) din setul de tipare frecvente de lungime k (pentru $k >= 1$) şi de a verifica frecvenţa de apariţie a acestora în baza de date.

Euristica A priori atinge performanţe bune prin reducerea dimensiunii seturilor de candidaţi. Cu toate acestea, în situaţii cu multe tipare frecvente, tipare lungi sau praguri de suport minim foarte joase, un algoritm bazat pe A priori poate, totuşi, suferi de două probleme destul de costisitoare:

- este costisitoare manipularea unui număr mare de seturi de candidaţi; de exemplu, dacă există 10^4 seturi de articole frecvente de lungime 1, (1-articolset-uri), algoritmul A priori va trebui să genereze mai mult de 10^7 candidaţi de lungime 2 şi să acumuleze şi să verifice frecvenţele de apariţie ale acestora;
- este anevoioasă scanarea repetată a bazei de date şi verificarea unui număr mare de candidaţi prin potrivire de tipare, în special pentru minarea tiparelor lungi.

Performanţa metodelor gen A priori ar putea creşte dacă s-ar putea evita generarea unui număr foarte mare de seturi de candidaţi. Această problemă a fost abordată de către Han în [3] prin trei aspecte.

În primul rând, este construită o structură de date nouă, denumită *arbore de tipare frecvente - frequent pattern tree* sau *FP-tree*, care este o structură de arbore-prefix extinsă, conţinând informaţii cruciale, cantitative despre tipare frecvente. În arbore, vor exista noduri doar pentru articole frecvente de lungime 1, nodurile fiind aranjate astfel încât tiparele cu apariţii mai frecvente vor avea o probabilitate mai ridicată în a partaja noduri decât cele mai puţin frecvente.

În al doilea rând, este dezvoltată o metodă de creştere a unui fragment de tipar, bazat pe *FP-tree*, care porneşte de la un tipar frecvent de lungime 1 (ca şi un *tipar sufix* iniţial), examinează doar *baza sa condiţională de tipare – conditional pattern base* (o sub-bază de date care constă din setul de articole frecvente care apar împreună cu tiparul sufix), construieşte *FP-tree*-ul său *condiţional* şi efectuează minare, în mod recursiv, pe acesta. Creşterea tiparului este atinsă prin concatenarea tiparului sufix cu noile tipare generate din *FP-tree*-ul

condiţional. Deoarece setul de articole (articolset-ul) frecvent din oricare tranzacţie este întotdeauna codat în calea corespunzătoare a arborilor FP, creşterea tiparelor asigură completitudinea rezultatului. În acest context, metoda nu este gen A priori, adică o generare şi verificare restricţionată, ci este doar o verificare restricţionată. Operaţiile majore ale minării sunt acumularea de frecvenţe şi ajustarea contoarelor pe căile prefix, care sunt, în general, mult mai economice decât operaţiile de generare de candidaţi şi cele de potrivire de tipare efectuate în majoritatea algoritmilor gen A priori.

În al treilea rând, tehnica de căutare folosită în minare este *bazată pe partiţionare*, o metodă mai degrabă tip *divide şi cucereşte,* decât gen A priori, *generare de jos în sus a combinaţiilor de seturi de articole frecvente.* Aceasta reduce, în mod dramatic, dimensiunea *bazei de tipare condiţionale*, generate la fiecare nivel de căutare ca şi, de altfel, dimensiunea arborelui condiţional corespunzător. Mai mult, transformă problema găsirii unor tipare frecvente lungi, în căutarea unora mai scurte şi concatenând sufixele. Foloseşte cele mai puţin frecvente articole ca sufixe, ceea ce oferă o selectivitate bună. Toate aceste tehnici contribuie la reducerea substanţială a costurilor de căutare.

În paragraful 2 din această lucrare, voi prezenta metoda de construcţie a unui arbore a tiparelor frecvente şi voi descrie metoda FP-Growth [3], bazată pe o structură de arbore FP-tree, iar în paragraful 3, voi descrie o nouă metodă de construire a arborelui tiparelor frecvente cu performanţe promiţătoare, introdusă în [6] pe care am denumit-o Dynamic FP-tree. Evaluarea performanţelor celor două metode am realizat-o în paragraful 4, prin folosirea cardului de comparare a algoritmilor de descoperire a regulilor de asociere, introdus în [8]. Concluziile sunt prezentate în paragraful 5.

2. Construcţia arborelui tiparelor frecvente (FP-tree)

Fie $I = \{i_1, i_2, ..., i_m\}$ un set de articole, iar $D = <T_1, T_2, ..., T_n>$ o bază de date tranzacţională, unde T_i ($i \in [1..n]$) este o tranzacţie care conţine un set de articole din I. ***Suportul*** (sau frecvenţa de apariţie) a unui tipar A, care este un set de articole, este numărul de tranzacţii din D care îl conţine pe A. A este un *tipar frecvent* dacă suportul lui A nu este mai mic decât un suport minim *minsup* predefinit.

Fiind dată o bază de date tranzacţională D şi un suport minim, *minsup*, problema *găsirii setului complet de tipare frecvente* este denumită ***problema minării tiparelor frecvente***.

Proiectarea unei structuri de date compactă pentru minarea eficientă a tiparelor frecvente Han, în [3], se bazează pe un exemplu.

Exemplul 1: Fie baza de date tranzacţională D, primele două coloane din 2 – 1 şi fie *minsup*=3.

O structură compactă poate fi proiectată pe baza următoarelor observaţii:

1. deoarece doar articolele frecvente vor juca un rol în minarea tiparelor frecvente, este necesară efectuarea unei scanări a bazei de date D pentru a identifica setul de articole frecvente;
2. dacă stocăm setul de articole frecvente ale fiecărei tranzacţii într-o structură compactă, am putea evita scanarea repetată a bazei de date D;
3. în cazul în care mai multe tranzacţii au în comun un set de articole frecvente, acestea ar putea fi reunite într-un singur set, cu numărul de apariţii înregistrate într-o variabilă *count*; este uşor de verificat dacă două seturi sunt identice dacă articolele frecvente în toate tranzacţiile sunt sortate după o anumită ordine dată;
4. dacă două tranzacţii au în comun un anumit prefix, corespunzător unei ordonări a seturilor frecvente, părţile comune pot fi reunite folosind o structură prefix atât timp cât numărul de apariţii *count* este înregistrat în mod corect. Dacă articolele frecvente sunt sortate descendent după frecvenţa lor de apariţie, există şanse mai mari ca să avem mai multe şiruri prefix în comun.

Tabel 2 - 1. Baza de date cu tranzacţii

TID	Articole cumpărate	Articole frecvente (ordonate)
100	*F, a, c, d, g, i, m, p*	*f, c, a, m, p*
200	*a, b, c, f, l, m, o*	*f, c, a, b, m*
300	*b, f, h, j, o*	*f, b*
400	*b, c, k, s, p*	*c, b, p*
500	*a, f, c, e, l, p, m, n*	*f, c, a, m, p*

Cu aceste observaţii se poate construi un *FP-tree* în următorul mod [3].

În primul rând, derivăm lista articolelor frecvente printr-o scanare a bazei de date D, $\langle(f{:}4), (c{:}4), (a{:}3), (b{:}3), (m{:}3), (p{:}3)\rangle$ (numărul de după: indică suportul), în care articolele sunt ordonate în ordine descrescătoare a frecvenței lor de apariție. Această ordonare este importantă deoarece fiecare cale din arbore va urma această ordonare. Pentru o mai uşoară înțelegere a algoritmului, în Tabel 2 - 1, coloana din dreapta arată această ordonare pentru fiecare tranzacție.

În al doilea rând, creăm rădăcina arborelui etichetat „*root*", după care scanăm baza de date a doua oară. Scanarea primei tranzacții duce la construcția primei căi prin arbore: $\langle(f{:}1), (c{:}1), (a{:}1), (m{:}1), (p{:}1)\rangle$. De notat faptul că articolele frecvente din tranzacție sunt ordonate după ordinea listei articolelor frecvente. Pentru cea de a doua tranzacție, deoarece lista corespunzătoare de articole frecvente (în ordinea respectivă) $\langle f, c, a, b, m\rangle$ partajează un prefix comun $\langle f, c, a\rangle$ cu calea deja existentă $\langle f, c, a, m, p\rangle$, contorul fiecărui nod de-a lungul prefixului este incrementat cu 1, un nou nod $(b{:}1)$ este creat şi legat ca fiu al nodului $(a{:}2)$ şi un alt nod nou $(m{:}1)$ este creat şi legat ca fiu al nodului $(b{:}1)$. Pentru a treia tranzacție, deoarece lista sa de articole frecvente $\langle f, b\rangle$ partajează doar nodul $\langle f\rangle$ cu subarborele prefix f, contorul lui f este incrementat cu 1 şi un nou nod $(b{:}1)$ este creat şi legat cu nodul $(f{:}3)$ ca fiu. Scanarea celei de a patra tranzacții conduce la crearea celei de a doua ramuri $\langle(c{:}1), (b{:}1), (p{:}1)\rangle$. Pentru ultima tranzacție, deoarece lista sa de articole frecvente $\langle f, c, a, m, p\rangle$ este identică cu prima, calea este partajată şi contorul fiecărui nod de-a lungul căii este incrementat cu 1.

Pentru a facilita traversarea arborelui, este construit un tabel antet al articolelor, în care fiecare intrare arată spre apariția acestuia în arbore printr-un pointer la un nod al arborelui (cap de listă). Nodurile cu acelaşi articol sunt înlănțuite prin astfel de pointeri. Arborele rezultat după scanarea tuturor tranzacțiilor este arătat în figura 2-1.

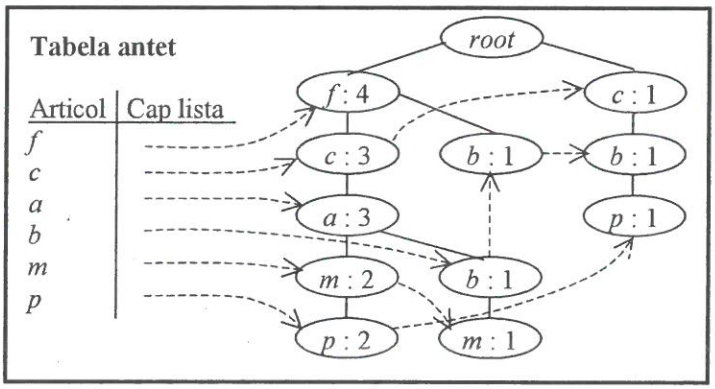

Figura 2 - 1. FP-tree pentru Exemplul 1

Acest exemplu conduce la următorul algoritm de construcție a *arborelui tiparelor frecvente* [3]:

Definiția 1 (FP-tree): Un arbore de tipare frecvente (FP-tree) este un arbore cu structura definită mai jos.

1. el constă dintr-un nod rădăcină, etichetat „root", un set de subarbori prefix de articole ca fii ai rădăcinii şi o tabelă antet pentru articolele frecvente;
2. fiecare nod din subarborele prefix de articole constă din trei câmpuri: *nume-articol*, *contor* şi *înlănțuire-nod*, unde *nume-articol* se referă la articolul pe care acest nod îl reprezintă, *contor* înregistrează numărul de tranzacții reprezentate de porțiunea căii care ajunge la acel nod, iar *înlănțuire-nod* face legătura la următorul nod din FP-tree, care se referă la acelaşi articol sau conține NULL dacă nu mai există un alt astfel de nod;
3. fiecare intrare în *tabela antet pentru articolele frecvente* constă din două câmpuri (1) *nume-articol* şi (2) *capul listei*, care arată spre primul nod din FP-tree care se referă la *nume-articol*.

Pe baza acestei definiții, avem următorul algoritm de construcție a arborelui FP-tree.

Algoritmul FP-tree (construcția FP-tree)

Intrare: O bază de date de tranzacții D şi un suport minim s.

Ieşire: Arborele tiparelor frecvente (FP-tree) corespunzător.

Metoda: Arborele FP-tree este construit în următorii paşi:

1. se scanează baza de date tranzacții D, se colectează setul de articole frecvente F şi suportul corespunzător acestora, se sortează F în ordinea descrescătoare a suportului în L, lista articolelor frecvente;

2. se creează nodul rădăcină al unui arbore FP-tree, T şi se etichetează cu „root"; pentru fiecare tranzacţie *Trans* din D se efectuează următoarele: se selectează şi se ordonează articolele din *Trans*, conform ordonării lui L, adică în ordine descrescătoare a suportului; fie lista articolelor frecvente din *Trans* $[p|P]$, unde p este primul element, iar P este restul listei; se apelează *insert_tree*($[p|P]$, T); funcţia *insert_tree*($[p|P]$, T) este efectuată în felul următor: dacă T are un fiu N astfel încât $N.nume\text{-}articol = p.nume\text{-}articol$, atunci incrementează contorul lui N cu 1, altfel se creează un nou nod, cu contorul iniţializat pe 1, părintele T şi *înlănţuire-nod* înlănţuit cu nodurile având acelaşi *nume-articol*; dacă P nu este gol, se apelează în mod recursiv *insert_tree(P, T)*.

Din procesul de construcţie a arborelui FP-tree putem observa că sunt necesare exact două scanări complete ale bazei de date D. Prima scanare colectează articolele frecvente, iar a doua scanare construieşte arborele FP-tree. Costul inserării unei tranzacţii *Trans* în arbore este $O(|Trans|)$, unde $|Trans|$ este numărul de articole frecvente din *Trans*.

Dimensiunea unui arbore FP-tree este limitată de dimensiunea bazei de date corespunzătoare deoarece fiecare tranzacţie va contribui cu, cel mult, o cale la arborele FP-tree, cu lungimea egală cu numărul de articole frecvente din acea tranzacţie. Deoarece există, de multe ori, multe partajări ale articolelor frecvente între tranzacţii, dimensiunea arborelui este, de cele mai multe ori, mult mai mică decât aceea a bazei de date originală. Spre deosebire de metodele gen A priori, care pot genera un număr exponenţial de candidaţi în cel mai rău caz, în nici un caz, nu va fi generat un FP-tree cu un număr exponenţial de noduri.

FP-tree este o structură foarte compactă, care stochează informaţia pentru minarea tiparelor frecvente. Deoarece o singură cale „$a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_n$" în subarborele a_1-prefix înregistrează toate tranzacţiile ale căror set maximal de articole frecvente este de forma „$a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_k$" pentru orice $1 \leq k \leq n$, dimensiunea arborelui FP-tree este, substanţial, mai mică decât dimensiunea bazei de date şi a dimensiunii setului de candidaţi generaţi pe parcursul minării regulilor de asociere.

Articolele din setul frecvent de articole sunt ordonate descrescător, în funcţie de suportul lor. Articolele care apar mai frecvent sunt aranjate mai aproape de vârful arborelui FP-tree şi, astfel, au o probabilitate mai mare de a fi partajate. Aceasta indică faptul că structura FP-tree este, în general, foarte compactă. De asemenea, experimentele arată că rezultă un arbore FP-tree destul de mic prin comprimarea unei baze de date destul de mari.

2.1. Descoperirea tiparelor frecvente folosind FP-tree. Metoda FP-Growth

În această secţiune, voi prezenta, conform **Error! Reference source not found.**, modul de explorare a informaţiilor compacte, stocate în arbore, şi voi descrie o metodă eficientă pentru minarea setului complet de tipare frecvente.

Algoritmul FP-Growth descoperă setul complet de articolset-uri frecvente, din baza de tranzacţii D. Arborele FP-tree al bazei de date D conţine informaţiile complete ale bazei de date D în legătură cu minarea tiparelor frecvente, care au suportul peste pragul *minsup*.

Algoritmul FP-Growth pentru minarea tiparelor frecvente, utilizând arborele FP-tree, este descris în următoarea secvenţă:

Algoritmul FP-Growth (Minarea tiparelor frecvente cu FP-tree prin creşterea fragmentelor de tipare)

Intrare: Arborele FP-tree construit pe baza algoritmului FP-tree utilizând baza de date D şi un suport minim s.

Ieşire: Setul complet de tipare frecvente.

Metoda: Apelul **FP-Growth(FP-tree**, *null*).

Procedure **FP-Growth**(Tree, α).

{

(1). **if** *Tree* conţine o singură cale P;

(2). **then for each** combinaţie (notată cu β) a nodurilor din calea P **do**;

(3). generează tiparele $\alpha \cup \beta$ cu *support = suportul minim ale nodurilor din* β;

(4). **else for each** a_i din tabela antet *Tree* **do** {

(5). generează tiparele $\beta = a_i \cup \alpha$ cu *support* = a_i.*support*;

(6). construieşte baza condiţională de tipare a lui β şi

apoi arborele FP-tree condiţional a lui β notat $Tree_\beta$;

(7). **if** $Tree_\beta \neq \varnothing$

(8). **then** call **FP-Growth**($Tree_\beta$, β)

}

}

Dacă arborele conţine o singură cale, tiparele generate sunt combinaţii ale nodurilor din cale, iar suportul fiind suportul minim al nodurilor din subcăi. Aceasta se realizează în liniile (1) - (3) ale procedurii FP-Growth. Altfel, se construieşte baza condiţională de tipare şi se minează arborele FP-tree condiţional pentru fiecare articolset frecvent a_i, iar suportul fragmentelor se ia ca fiind suportul articolset-urilor frecvente generate din baza condiţională de tipare.

Procesul de minare bazat pe algoritmul FP-Growth scanează arborele FP-tree al bazei de date D o dată şi generează o bază de tipare mică B_{ai} pentru fiecare articol frecvent a_i, fiecare constând din seturi de căi prefix transformate ale lui a_i. Minarea tiparelor frecvente este apoi executată recursiv pe baza de tipare mică B_{ai} cu construirea unui arbore FP-tree condiţional pentru B_{ai}. Aşa cum a reieşit din analiza algoritmului FP-tree, un arbore FP-tree al unei baze de date este, în general, mult mai mic decât dimensiunea bazei de date. Similar, deoarece arborele FP-tree condiţional, "FP-tree | a_i", este construit pe baza de tipare B_{ai} el ar trebui să fie mult mai mic şi niciodată mai mare decât B_{ai}. Mai mult, o bază de tipare B_{ai}, este, în general, mult mai mică decât arborele său FP-tree original, deoarece ea constă din căile prefix transformate referitor la numai unul din articolele frecvente, a_i. Astfel, procesul de minare următor lucrează pe un set de baze de tipare şi arbori FP-tree condiţionali mult mai mici, în general. Aceasta este, în general, mult mai puţin costisitor decât generarea şi testarea unui număr foarte mare de tipare candidat, fapt ce conduce la o eficienţă mai mare a algoritmului FP-Growth.

Din algoritmul FP-Growth se observă că procesul de minare este un proces de tipul divide şi cucereşte (divide-and-conquer), şi scala de micşorare este impresionantă. În general, factorul de micşorare este în jur de 20 ~ 100 pentru construirea unui arbore FP-tree dintr-o bază de date.

3. Construcţia unui Dynamic FP-tree

Metoda introdusă în [6], numită Dynamic FP-tree, îmbunătăţeşte performanţele algoritmilor bazaţi pe o structură de arbore, FP-tree, prin reducerea numărului de scanări ale bazei de date, de la două scanări la numai una singură şi, de asemenea, permite un proces de actualizare dinamică a arborelui tiparelor frecvente, care se realizează pe o structură deja existentă.

Pentru a arăta modul de construcţie a unui Dynamic FP-tree voi pleca de la construcţia unui arbore FP-tree pe care am făcut câteva observaţii:

1. pentru o bază de date "logic" identică nu rezultă un arbore FP-tree unic; de exemplu, considerând baza de date cu tranzacţii prezentată în Tabelul 2 – 1 şi arborele corespunzător acesteia din figura figura 2-1, dacă primul articol din prima tranzacţie ar fi c, şi nu f, dar tranzacţia ar conţine aceleaşi articole, arborele FP-tree rezultat ar fi diferit;
2. procesul de construire a arborelui tiparelor frecvente are nevoie de două scanări ale bazei de date.

Pentru a depăşi prima problemă, am propus o modificare a algoritmului de construcţie a unui FP-tree, şi anume aceea că, în pasul 1 al algoritmului de construcţie al arborelui FP-tree, descris mai sus în secţiunea 2, setul de articole frecvente F ar trebui să fie sortat nu numai în ordine descrescătoare a frecvenţei suport, ci şi în ordine lexicografică (aceasta pentru articole care au acelaşi suport) rezultând L, lista articolelor frecvente. Aceasta ar trebui să asigure un arbore FP-tree unic pentru baze de date "logic" identice.

Pentru cea de-a doua problemă, am propus două abordări:

1. aplicarea directă a pasului 2, din algoritmul de construire a arborelui FP-tree, fără a scana prima dată baza de date pentru obţinerea frecvenţelor suport, utilizând numai o ordine lexicografică a articolelor într-o tranzacţie; de asemenea, colectarea frecvenţelor suport într-o tabelă header; suplimentar, avem pasul 3 în care tabela header este ordonată descrescător după suport şi ordonată lexicografic şi se reconstruieşte arborele în conformitate cu noua ordine, dar numai prin scanarea arborelui;
2. proiectarea unui algoritm de reordonare dinamică a FP-tree astfel încât, aplicând direct pasul doi din algoritmul original, cu actualizarea mai întâi a frecvenţelor suport în tabela header, păstrând ordinea

corespunzătoare atât a frecvenţelor suport descrescătoare, cât şi lexicografică a articolelor, iar în cazul în care există o "promovare" a cel puţin un articol, atunci să se efectueze o reordonare a arborelui pentru acele articole care au fost "promovate".

Ambele abordări vor produce un FP-tree care va reflecta toate tranzacţiile din baza de date. Această metodă de construire a arborelui şi de stocare a arborelui este mai eficientă din punct de vedere a timpului consumat, în cazul unui număr mare de interogări cu diferite suporturi minime. Dar uneori arborele rezultat ar putea fi prea costisitor de memorat în întregime în memorie din punct de vedere a spaţiului consumat de acesta.

Utilizând această metodă de construire dinamică a arborelui Dynamic FP-tree, nu trebuie să reconstruim arborele chiar dacă baza de date se actualizează prin adăugarea unor noi tranzacţii. În acest caz, se va apela algoritmul, luând în considerare numai aceste noi tranzacţii. Astfel, arborele se va modifica dinamic, prin adăugarea acestor noi tranzacţii la forma lui anterioară.

Prin această metodă de construire a arborelui Dynamic FP-tree, se oferă un răspuns foarte rapid la orice interogare pe o bază de date, care se actualizează în mod continuu.

În continuare, voi descrie modificările care sunt făcute pe o structură FP-tree originală şi, de asemenea, algoritmul de construcţie a arborelui Dynamic FP-tree.

Deoarece în procesul de reordonare este nevoie să se modifice conţinutul nodurilor din listă, pentru o mai mare uşurinţă, înlocuim lista simplu înlănţuită cu o listă dublu înlănţuită. Din acest motiv, introduc un pointer suplimentar *nume-leg-ant* în fiecare nod din structura de nod, definită în secţiunea 2 (punctul 2 din Definiţia 1), care conţinea *nume-articol*, *contor* şi *înlănţuire-nod*. Aceasta face legătura cu nodul anterior din arbore, care conţine aceeaşi denumire de articol sau, dacă nu este nici unul, atunci va indica spre intrarea corespunzătoare din tabela header.

O altă modificare faţă de algoritmul original este aceea că headerul va avea două tabele, una corespunzătoare tabelei originale şi a cea de-a doua, o tabelă *master* adiţională, care constă dintr-un câmp, *nume-articol* şi un câmp *contor* şi va ţine ordinea curentă de inserare în arborele FP-tree. Inserarea în tabela originală *original-ordered-table* poate fi făcută întotdeauna respectând ordinea impusă de suportul curent (ordine descrescătoare) şi în ordine lexicografică. Orice articol nou va fi adăugat respectând această ordine.

Numim „promovare", mutarea unui nod în arbore datorită creşterii frecvenţei suport corespunzătoare. Orice „promovare" va fi detectată prin compararea tabelei *master* cu tabela originală, ordonată *original-ordered-table*. Pe această structură definesc o operaţie de *checkpoint,* care va sincroniza tabela *master* cu tabela originală, ordonată *original-ordered-table,* adică se copiază conţinutul tabelei originale în tabela master. Aplicând operaţia de ordonare curentă, după inserarea unei tranzacţii, vom obţine un FP-tree actualizat. În conformitate cu această ordine curentă, arborele va fi reordonat dinamic.

Algoritmul de construcţie a arborelui tiparelor frecvente Dynamic FP-tree este următorul:

Algoritmul Dynamic FP-tree (algoritmul de construire dinamică a arborelui tiparelor frecvente):

Intrare: O bază de date tranzacţională D şi un suport minim s.

Ieşire: Arborele tiparelor frecvente (Dynamic FP-tree) corespunzător.

Metoda: Arborele tiparelor frecvente este construit în următorii paşi:

1. Se creează nodul rădăcină T, al unui arbore FP-tree, şi se etichetează cu „root". Pentru fiecare tranzacţie *Trans* din baza de date, D execută următoarele:
   a. adaugă articolele din tranzacţia *Trans* într-o tabelă header;
   b. selectează şi sortează articolele din tranzacţia *Trans* conform cu ordinea din tabela *master*; fie [p | P] lista sortată de articole frecvente din *Trans*, unde p este primul element şi P restul listei; apelează funcţia *insert_tree*([p | P], T);
   c. funcţia *insert_tree*([p | P], T) este executată astfel: dacă T are un fiu N astfel că *N.item-name = p.item-name* atunci incrementăm contorul lui N cu 1; altfel creăm un nod nou N şi iniţializăm contorul lui pe 1, legătura părinte a lui va indica spre T, iar legătura nodului va indica la nodurile cu aceeaşi denumire *item_name* prin structura de listă de noduri; dacă mai avem elemente în P, atunci se apelează recursiv funcţia *insert_tree*(P, N);
   d. dacă se "promovează" un nod, atunci este nevoie de o reordonare şi se apelează funcţia *reorder(*dynFPtree*)* pe arbore.

3.1. Reordonarea arborelui Dynamic FP-tree

Funcţia de reordonare a arborelui, *reorder(*dynFPtree*)* se execută după cum urmează:

1. colectează toate articolele „promovate" (articolele care au frecvenţa suport modificată) într-o listă *reorderList*, ordonată după suport descrescător şi în ordine lexicografică;
2. apelează *checkpoint ()* pentru a actualiza ordinea de inserare în FP-tree;
3. pentru fiecare articol din *reorderList* parcurge lista nodurilor şi, pentru fiecare dintre aceste noduri, apelează funcţia *moveUp(node)* pentru a pune acel nod în poziţia corectă în FP-tree, în conformitate cu tabela *master*.

Funcţia *moveUp(node)* se execută astfel:

1. repetă următoarea procedură până când nodul (*node*) şi părintele său (*parent*) sunt în ordinea corectă:
   a. preia părintele indicat de nodul părinte al nodului curent (*pparent*);
   b. dacă *parent* are aceeaşi frecvenţă ca şi *node*, elimină *parent* din lista *childNodes* a părintelui acestuia (*pparent*) şi asignează-l lui *newNode*;
   c. altfel efectuează următoarele acţiuni:
      i. creează un nou nod *newNode* cu acelaşi articol ca şi *parent*, dar având frecvenţa nodului *node*;
      ii. inserează nodul *newNode* în lista nodurilor cu acelaşi articol;
      iii. ajustează frecvenţa nodului *parent* prin scăderea frecvenţei nodului *node*;
      iv. elimină *node* din lista *childNodes* a lui *parent*.
   d. înlocuieşte lista fiilor din *newNode* cu lista fiilor lui *node* şi actualizează legăturile părinte ale noilor fii cu noul părinte (*newNode*);
   e. setează legătura părinte a lui *node* spre *pparent* (părintele original al părintelui nodului curent *node*), iniţializează lista fiilor *childNodes* cu nodul *newNode* şi setează legătura părinte a acestuia la *node*;
   f. inserează nodul *node* în lista fiilor lui *pparent*.

3.2. Reducerea dimensiunii Dynamic FP-tree

Algoritmul prezentat asigură performanţe superioare în detrimentul consumului de memorie. Acest lucru se datorează faptului că, în funcţia *moveUp* la pasul 1.f la un anumit nivel din FP-tree, am putea avea mai mult de un singur nod care să se refere la acelaşi articol. Acest lucru nu constituie nici o problemă pentru algoritmul de minare, dar duce la un arbore cu mai multe noduri decât necesar pentru reprezentarea compactă a tranzacţiilor din baza de date.

Din acest motiv, am dezvoltat un algoritm pentru combinarea a doi subarbori care au rădăcinile reprezentând acelaşi articol (*merge*).

Acest algoritm ar fi folosit în pasul 1.f din algoritmul *moveUp*, pasul în acest caz devenind:

   f. dacă avem, deja, un nod existent *existingNode* pentru articolul reprezentat de *node* în lista fiilor lui *pparent*, atunci apelează *merge(existingNode, node)* şi continuă cu nodul *existingNode* ca şi nod curent (*node*);
   g. altfel, inserează nodul *node* în lista fiilor lui *pparent*.

Funcţia *merge(existingNode, node)* este implementată în următorul mod:

1. combină informaţia de frecvenţă prin incrementarea frecvenţei suport a nodului existent *existingNode* cu frecvenţa nodului *node*;
2. elimină *node* din lista nodurilor cu acelaşi articol, dar fără modificarea legăturilor acestuia (acestea fiind necesare pentru continuarea algoritmului prin parcurgerea listei nodurilor corespunzătoare);
3. combină fiii nodului *node* cu fiii nodului existent *existingNode* prin executarea următoarelor acţiuni pentru fiecare fiu *child* al nodului:
   a. dacă există un nod *sameItem* pentru acelaşi articol în nodul fiu al nodului *existingNode*, atunci combină cei doi prin apelarea recursivă a funcţiei *merge(sameItem, child)*;

b. altfel, inserează nodul fiul *child* în lista fiilor nodului *existingNode* şi setează părintele nodului fiu *child* la nodul *existingNode*.

4. Evaluarea experimentelor şi studierea performanţelor

Luând în considerare rezultatele raportate în [8] şi folosind cadrul de dezvoltare şi comparare, am adaptat implementarea a doi algoritmi de descoperire a regulilor de asociere la acest cadru pentru a-l putea folosi şi pentru a compara rezultatele obţinute şi pentru alţi algoritmi, ca de exemplu A priori [2]. Primul algoritm implementat se baza pe o structură FP-tree, algoritm numit *FP-Growth* [3], iar celălalt bazat pe structura propusă Dynamic FP-tree, numit *DynFP-Growth*. Folosind acest cadru, am realizat câteva testări pe diverse baze de date. Pe algoritmul *DynFP-Growth* am realizat o optimizare prin folosirea funcţiei *merge*, descrisă în secţiunea 3.2, care are rolul de a reduce dimensiunea arborelui şi am denumit algoritmul *DynFP-GrowthMerge*.

```
public class FPGrowth extends ARMiner {
  /** Arborele FP-tree */
  protected FPTree fpTree;
  /** Implementarea efectiva a algoritmului */
  public long mineFrequentItemsets() {
    ItemSet itemSet;
    ArrayList rawItemSet;

    frequentItemSets.clear();
    try {
      first();
      while((itemSet = getNextItemSet()) != null) {
        fpTree.addItems(itemSet);
      }
      fpTree.sortHeader();
      if (applyMinSupportToHeader)
        fpTree.hdr.applyMinSupport(getMinSuppCount());
      first();
      while((rawItemSet = getNextRawItemSet()) != null) {
        fpTree.addItemSet(rawItemSet);
      }
      fpGrowth(fpTree, frequentItemSets, new ArrayList(), getMinSuppCount());
    } catch (SQLException e) {
      e.printStackTrace();
    }
    frequentItemSets.internalSort();
    return 2;
  }
  protected void fpGrowth(FPTree fpTree, ItemSets frequentItemSets,
    ArrayList itemSet, long minSuppCount) {
    if (fpTree.singlePath()){
      /* Genereaza toate combinatiile din cale care satisfac cerintele */
```

```
    }
    else {
      /* Aplică recursiv algoritmul pe bazele condiționale de tipare */
    }
  }
  /* ... */
}
```

Figura 4 - 1. Implementare FPGrowth

```
public class DynFPGrowth extends FPGrowth {
  /** Implementarea efectiva a algoritmului */
  public long mineFrequentItemsets() {
    ArrayList rawItemSet;
    long cnt;
    long crtSuppCnt;
    DynFPTree dynFPTree = (DynFPTree)fpTree;

    frequentItemSets.clear();
    try {
      cnt = 0;
      first();
      cnt++;
      while((rawItemSet = getNextRawItemSet()) != null) {
        crtSuppCnt = Math.round(cnt * getMinSupport());
        dynFPTree.addItems(new ItemSet(rawItemSet));
        dynFPTree.addItemSet(rawItemSet);
        if (!performReorderAtEnd && ((cnt % performReorderAtEvery) == 0)) {
          if (dynFPTree.isReorderingNeeded(crtSuppCnt)) {
            dynFPTree.reorder(crtSuppCnt);
          }
        }
        cnt++;
      }
      if (performReorderAtEnd) {
        if (dynFPTree.isReorderingNeeded(getMinSuppCount())) {
          dynFPTree.reorder(getMinSuppCount());
        }
      }
      // Nu trebuie sa sortam Header-ul deoarece este deja sortat !!!
      fpGrowth(dynFPTree, frequentItemSets, new ArrayList(), getMinSuppCount());
    } catch (SQLException e) {
```

```
            e.printStackTrace();
        }
        frequentItemSets.internalSort();
        return 1;
    }
    /* ... */
}
```

Figura 4 - 2. Implementare DynFPGrowth

Calculatorul pe care am realizat testele a fost un Pentium 4 la 1.7 GHz, cu 256 MBRAM sub sistemul Windows 2000. Pentru o evaluare cât mai realistă, am optat pentru baze de date, stocate pe servere SQL (Microsoft SQL 2000), accesul fiind realizat prin interfețele ODBC standard.

Pentru studierea performanțelor algoritmilor şi pentru studierea scalabilității, am generat seturi de date, de la 10 000 până la 500 000 de tranzacții, şi am folosit factori suport, începând de la 5% până la 40%. Orice tranzacție poate conține mai mult decât un articolset frecvent. Tranzacțiile pot avea un număr diferit de articole, de asemenea, şi dimensiunea unui articolset frecvent diferă. Unele articolset-uri pot avea mai puține articole, altele pot avea mai multe. Având în vedere aceste observații, am generat seturi de date ținând cont de numărul de articole ale unei tranzacții, de numărul de articole dintr-un articolset frecvent etc. Parametrii necesari pentru a genera setul de date de test sunt definiți în Tabelul 4 – 1.

Tabel 4 - 1. Parametrii utilizați pentru generarea setului de date

|D|	Numărul de tranzacții
|T|	Media aritmetică a dimensiunii unei tranzacții
|L|	Numărul maxim de articolset-uri frecvente
N	Numărul de articole

Seturile de date de test sunt generate prin setarea numărului de articole la N = 100 şi a numărului maxim de articolset-uri frecvente la |L| = 3000. De asemenea, se alege media aritmetică a dimensiunii unei tranzacții, |T| = 10.

Câteva rezultate comparative ale algoritmilor A priori, FP-Growth, DynFP-Growth, FP-GrowthMinSup şi DynFP-GrowthMerge obținute pe diferite baze de date de test, raportate la un factor suport de 5%, sunt prezentate în Tabelul 4 – 2.

Tabel 4 - 2. Rezultate raportate la suport de 5%

Tranzacții (în mii)	Timp(secunde)				
	Minsuport 5%				
	A priori	DynFP-Growth	DynFP-GrowthMerge	FP-Growth	FP-GrowthMinSup
10	13.94	2.32	1.78	3.76	3.06
20	21.98	3.98	3.38	6.88	6.30
30	48.37	8.23	8.07	14.63	13.73
40	66.50	12.10	11.50	20.90	19.80
50	107.65	19.50	18.65	34.30	32.95
80	198.30	37.90	36.20	64.80	63.70
110	1471.40	55.00	53.30	95.50	93.90
150	3097.20	98.90	97.80	174.60	170.30
190	5320.60	152.70	150.00	273.60	268.00
300	9904.80	284.00	281.80	526.70	511.90
400	17259.20	458.10	449.80	849.70	849.20
520	20262.60	610.20	601.40	1150.70	1142.40

Rezultatele din Tabelul 4 – 2 arată că timpul de execuţie al algoritmilor creşte o dată cu dimensiunea bazei de date. Cea mai bună performanţă o au algoritmii DynFP-GrowthMerge şi DynFP-Growth. O performanţă bună prezintă şi algoritmul FP-Growth comparativ cu A priori, dar se observă că performanţa lui DynFP-Growth este cu câteva ordine de mărime mai bună.

Din figura 4 – 3, se observă că timpul de execuţie al algoritmilor FP-Growth şi DynFP-Growth este constant raportat la aceeaşi bază de date, pentru un factor suport, care descreşte de la 40% la 5%, pe când timpul de execuţie al algoritmului A priori creşte o dată cu descreşterea factorului suport. Pentru un factor suport mai mare de 30% pe o bază de date de 40 000 de tranzacţii algoritmul A priori are performanţe asemănătoare cu DynFP-Growth şi mai bune decât FP-Growth, dar pentru un suport mai mic de 20% performanţa lui scade foarte mult. Astfel, timpul de execuţie al algoritmului A priori ajunge să crească de până la 3 ori mai mult decât timpul de execuţie al algoritmului FP-Growth şi de până la 5 ori mai mult faţă de DynFP-Growth, pentru un suport de 5%.

Se observă că algoritmii FP-Growth şi DynFP-Growth au o performanţă bună din punct de vedere al timpului de execuţie pentru diferite valori ale factorului suport, în timp ce algoritmul Apriori are o performanţă mult mai slabă în special pentru factor suport mai mic de 20 %.

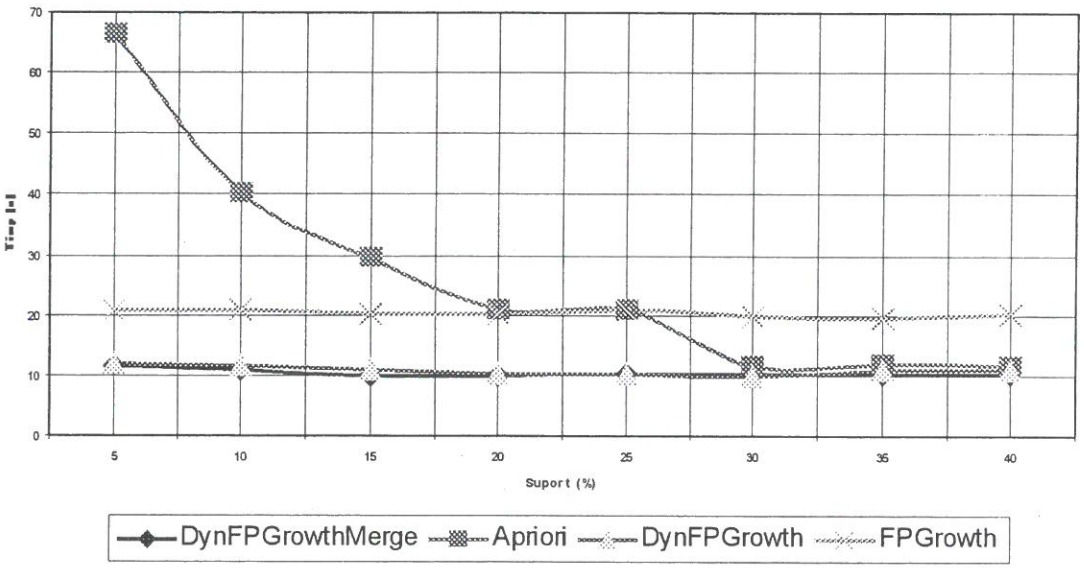

Figura 4 - 3. Scalabilitatea raportată la suport (D1 40K)

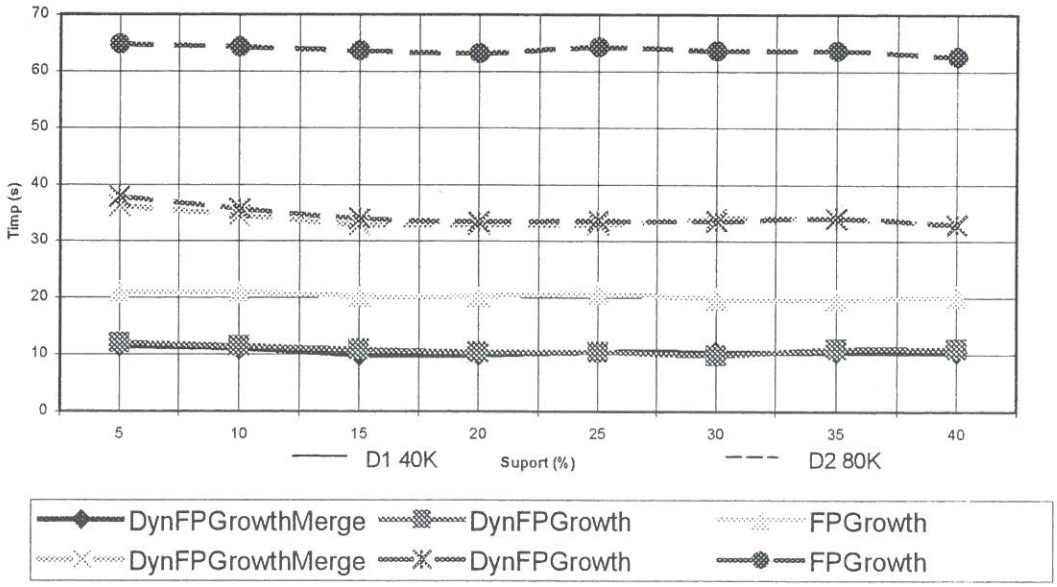

Figura 4 - 4. Scalabilitatea raportată la suport (D1 40K, D2 80K)

Scalabilitatea raportată la diferite valori ale suportului pe două baze de date (D1 = 40 000 şi D2 = 80 000 de tranzacţii) este prezentată în figura 4 – 4. Se observă că timpul de execuţie al celor doi algoritmi depinde de dimensiunea bazei de date şi nu de valoarea suport. Astfel, o dată cu creşterea dimensiunii bazei de date creşte şi timpul de execuţie al algoritmilor, dar performanţa algoritmului propus, DynFP-Growth este mai bună decât cea a algoritmului FP-Growth.

Din figura 4 – 5, se poate observa că atât algoritmul FP-Growth, cât şi DynFP-Growth, au o performanţă bună chiar şi pentru un factor suport mic (minsup = 5%), şi asta chiar pentru baze de date mari.

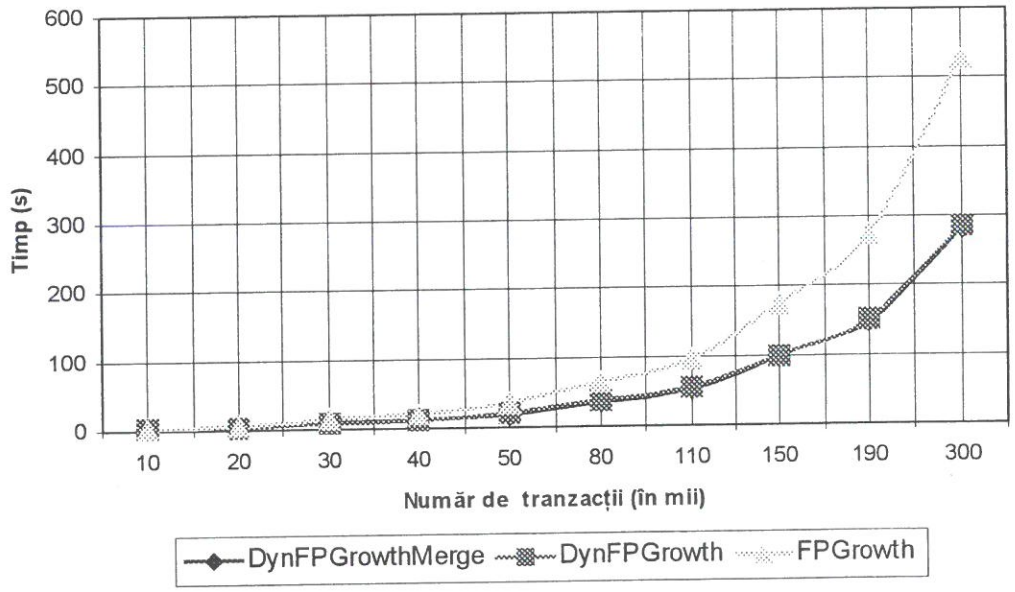

Figura 4 - 5. Timp de execuţie pentru un factor suport de 5%

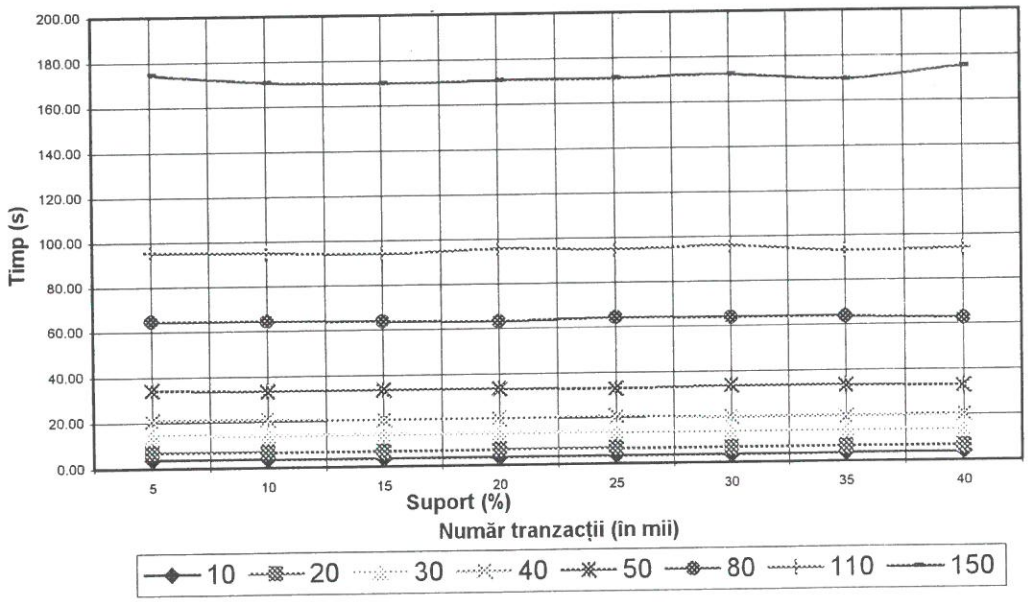

Figura 4 - 6. Scalabilitatea FP-Growth raportată la tranzacţii/suport

Din figura 4 – 6, se observă că timpul de execuţie al metodei FP-Growth nu depinde de suport, ci numai de dimensiunea bazei de date. În cazul metodei FP-GrowthMinSup, se ia în considerare suportul minim în construcţia arborelui tiparelor frecvente, ceea ce face ca, pentru un factor suport mare, algoritmul FP-GrowthMinSup să fie mai performant, iar pentru un suport mic, de exemplu 5%, performanţa lui să fie aproape identică cu al lui FP-Growth.

Metoda de construcţie a arborelui articolset-urilor frecvente nu ia în considerare suportul, astfel arborele va conţine toate tranzacţiile din baza de date şi, în funcţie de suport, se vor extrage din arbore articolset-urile frecvente,

care au frecvenţa mai mare decât suportul specificat. Această modalitate de construcţie a arborelui permite aflarea articolset-urilor frecvente, pentru diferite suporturi, fără a mai necesita o nouă construcţie a arborelui.

Prin aplicarea metodei propuse, de construire a arborelui tiparelor frecvente, DynFP-Growth, putem actualiza arborele prin adăugarea unor noi tranzacţii (care sunt adăugate în baza de date ulterior, după ultima minare) la arborele deja construit, fără a reconstrui arborele din nou.

Numai o singură modificare trebuie făcută în algoritmul propus pentru o minare dinamică şi anume, în pasul 1 al algoritmului Dynamic FP-tree:

1. încarcă arborele FP-tree, *T*, pentru fiecare tranzacţie nouă *Trans* din *DB*, execută paşii începând de la **a.** până la **d.** din algoritmul Dynamic FP-tree.

După modificarea arborelui tiparelor frecvente, putem aplica acelaşi algoritm FP-Growth pentru a efectua minarea articolset-urilor frecvente.

5. Concluzii

Din experimentele şi studiile de performanţă efectuate rezultă că algoritmul propus DynFP-Growth are performanţe mai bune decât algoritmul FP-Growth chiar şi la baze de date mari. Timpii de execuţie ai algoritmilor FP-Growth şi DynFP-Growth nu depind de suport, ci numai de dimensiunea bazei de date: astfel, timpul creşte o dată cu dimensiunea bazei de date, în timp ce performanţa algoritmului A priori depinde atât de suport, cât şi de dimensiunea bazei de date. Rezultatele minării, obţinute prin metoda propusă DynFP-Growth, sunt compatibile cu cele obţinute cu algoritmul FP-Growth pe o structură FP-tree originală **Error! Reference source not found.** Astfel, se obţin aceleaşi seturi de articole frecvente, dar ceea ce diferă este timpul de execuţie a celor doi algoritmi, care este mai mic în cazul lui DynFP-Growth, acest lucru rezultând din experimentele prezentate în secţiunea 4.

Metoda propusă are performanţe mai bune deoarece se face numai o singură parcurgere a bazei de date, ceea ce minimizează timpul de creare a arborelui tiparelor frecvente. Această modalitate de construcţie dinamică a arborelui Dynamic FP-tree poate fi folosită pentru a adăuga noi tranzacţii la arborele deja existent, ceea ce metodele existente, bazate pe o structură de arbore nu o permit. Acesta este unul din avantajele principale, pe care o aduce noua metodă propusă, de construire dinamică a arborelui Dynamic FP-tree.

Abordarea propusă este mai potrivită în următoarele cazuri:

- când sunt baze de date mari, care trebuie interogate într-un timp scurt, şi memoria principală nu este un factor determinat;
- când avem un arbore al tiparelor frecvente construit şi ar trebui să-l actualizăm cu noi tranzacţii fără să reconstruim întreg arborele tiparelor frecvente.

Bibliografie

1. **HAN, J., M. KAMBER**: Data Mining Concepts and Techniques, Morgan Kaufmann Publishers, San Francisco, USA, 2001, ISBN 1558604898.
2. **AGRAWAL R., R. SRIKANT**: Fast Algorithms for Mining Association Rules in Large databases. În Proc. of 20th Int'l conf. on VLDB, Santiago, Chile, September 1994, pp. 487-499.
3. **HAN, J., J. PEI, Y. YIN**: Mining Frequent Patterns without Candidate Generation. În: Proc. of ACM-SIGMOD, May 2000, ACM Press.
4. **PARK, J. S., M.S. CHEN, P.S. YU**: An Effective Hash-based Algorithm for Mining Association Rules. În: SIGMOD'95, pp. 175-186.
5. **SAVASERE, A., E. OMIECINSKI; S. NAVATHE**: An Efficient Algorithm for Mining Association Rules in Large Databases. În: VLDB'95, pp. 432-443.
6. **GYORODI, C.**: Contribuţii la dezvoltarea sistemelor de descoperire a cunoştinţelor. Teză de doctorat, Universitatea Tehnică "Politehnica" Timişoara, Romania, 2003.
7. **GYORODI, C., R. GYORODI, T. COFFEY, S. HOLBAN**: Mining Association Rules Using Dynamic FP-trees. În: Proc. of The Irish Signal and Systems Conference, University of Limerick, Limerick, Ireland, 30th June-2nd July, ISBN 0-9542973-1-8, pp. 76-82.
8. **GYORODI, R.**: A Comparative Study of Iterative Algorithms in Association Rules Mining". În: Studies in Informatics and Control, Volume 12 Number 3, Romania, 2003, ISSN 1220-1766, pp. 205-215.