Integracja drzew kandydatów w przetwarzaniu zbiorów zapytań eksploracyjnych algorytmem Apriori

Przemysław Grudziński[1], **Marek Wojciechowski**[2]

Streszczenie: Artykuł poświęcony jest problematyce przetwarzania zbiorów zapytań eksploracyjnych dla problemu odkrywania zbiorów częstych algorytmem Apriori. Z dotychczasowych rozwiązań w tym zakresie najlepszą metodą jest Common Counting, czyli współbieżne wykonanie zapytań algorytmem Apriori z integracją odczytów fragmentów bazy danych współdzielonych przez zapytania. W tej pracy proponujemy nową metodę przetwarzania zbiorów zapytań eksploracyjnych dla algorytmu Apriori - Common Candidate Tree, opartą o ściślejszą integrację przetwarzanych zapytań poprzez współdzielenie struktur danych w pamięci operacyjnej. Wyniki eksperymentów pokazują, że Common Candidate Tree jest metodą bardziej wydajną niż Common Counting, a dodatkowo, ze względu na oszczędniejsze gospodarowanie pamięcią operacyjną, może być stosowana dla liczniejszych zbiorów zapytań.

Słowa kluczowe: eksploracja danych, odkrywanie zbiorów częstych, zapytania eksploracyjne.

1. Wprowadzenie

Odkrywanie zbiorów częstych (ang. frequent itemset mining) (Agrawal et al., 1993) to jedna z podstawowych technik eksploracji danych (ang. data mining). Celem jest w tym przypadku odkrycie w kolekcji zbiorów (zwanych transakcjami) wszystkich podzbiorów, których liczba wystąpień przekracza zadany przez użytkownika próg (tzw. próg minimalnego wsparcia). Typowo, odkryte zbiory częste są wykorzystywane do generacji reguł asocjacyjnych (ang. association rules), ale mogą też stanowić docelową formę reprezentacji odkrytej wiedzy. Problem odkrywania zbiorów częstych i reguł asocjacyjnych został pierwotnie sformułowany w kontekście analizy koszyka zakupów w celu analizy współwystępowania produktów w transakcjach klientów, ale szybko znalazł wiele innych zastosowań w takich dziedzinach jak: medycyna, telekomunikacja, czy analiza zachowań użytkowników WWW.

Dotychczas w literaturze zaproponowano wiele algorytmów odkrywania zbiorów częstych. Dwie główne rodziny algorytmów wyznaczają strategie przeglądania przestrzeni poszukiwań. Klasycznym algorytmem realizującym przeszukiwanie wszerz jest Apriori (Agrawal et al., 1994). Z kolei najpopularniejszym algorytmem stosującym przeszukiwanie w głąb jest FP-growth (Han et al., 2000). Apriori rozpoczyna od odkrycia częstych elementów jako jednoelementowych zbiorów częstych,

[1] Wydział Matematyki i Informatyki, Uniwersytet im. Adama Mickiewicza w Poznaniu, Umultowska 87, 61-614 Poznań

[2] Wydział Informatyki i Zarządzania, Politechnika Poznańska, Piotrowo 2, 60-965 Poznań
e-mail: Marek.Wojciechowski@cs.put.poznan.pl

a następnie iteracyjnie generuje zbiory potencjalnie częste, określane mianem kandydatów, w oparciu o zbiory częste odkryte w poprzedniej iteracji. Każda iteracja kończy się pełnym odczytem bazy danych w celu zliczenia wystąpień kandydatów i wybrania spośród nich zbiorów częstych. Dla ułatwienia testowania zawierania się kandydatów w transakcjach odczytywanych z bazy danych, kandydaci umieszczani są w drzewie haszowym w pamięci operacyjnej.

FP-growth, podobnie jak Apriori, buduje większe zbiory częste z mniejszych, ale zamiast generowania kandydatów i zliczania ich wystąpień wykorzystuje ideę projekcji bazy danych. Projekcje są wyznaczane przez odkrywane zbiory częste, a rozrost zbiorów jest realizowany poprzez odkrywanie elementów częstych w projekcjach. Aby ułatwić wielokrotne projekcje, baza danych jest reprezentowana w pamięci operacyjnej w postaci zwięzłej struktury o nazwie FP-tree, do której zbudowania wymagane są dokładnie dwa odczyty źródłowej bazy danych. Algorytm FP-growth jest bardziej efektywny niż Apriori dla niskich progów minimalnego wsparcia oraz w przypadku gęstych zbiorów danych, czyli takich, które zawierają wiele dużych zbiorów częstych. W rzeczywistości jednak analizowane są dane o różnym charakterze i dlatego nie można wskazać algorytmu, który byłby najlepszym rozwiązaniem w każdym przypadku (Zheng et al., 2001).

Odkrywanie zbiorów częstych jest często postrzegane jako przetwarzanie zaawansowanych zapytań do bazy danych, za pomocą których użytkownik specyfikuje dane źródłowe do eksploracji, próg minimalnego wsparcia oraz opcjonalnie dodatkowe kryteria selekcji zbiorów (tzw. ograniczenia) (Imielinski i Mannila, 1996). Efektywne przetwarzanie zapytań eksploracyjnych w kontekście problemu odkrywania zbiorów częstych było przedmiotem wielu prac naukowych, skupiających się głównie na uwzględnianiu ograniczeń w procesie odkrywania (np. Pei i Han, 2000) i wykorzystywaniu wyników poprzednich zapytań (Baralis i Psaila, 1999, Cheung et al., 1996, Meo, 2003, Morzy et al., 2000).

Niniejszy artykuł poświęcony jest optymalizacji wykonania zbiorów zapytań eksploracyjnych dla problemu odkrywania zbiorów częstych. Ogólna idea polega na zastąpieniu sekwencyjnego wykonania zapytań wykonaniem współbieżnym, z wykorzystaniem faktu nakładania się danych źródłowych poszczególnych zapytań. Zbiory zapytań eksploracyjnych do wykonania mogą pojawiać się w systemach eksploracji danych pracujących w trybie wsadowym. Możliwe jest również grupowanie zapytań w zbiory do współbieżnego wykonania w interaktywnych, wielodostępnych środowiskach eksploracji danych, na zasadzie zbierania zapytań przychodzących do systemu w ustalonym oknie czasowym. Motywacją dla rozważanych technik może być np. scenariusz w którym wielu użytkowników prowadzących analizy koszyka zakupów poszukuje zbiorów częstych w różnych, nakładających się, przedziałach czasowych zgromadzonej historii transakcji klientów.

Najlepszą z dotychczas zaproponowanych metod wykonania zbioru zapytań eksploracyjnych odkrywających zbiory częste jest Common Counting, czyli współbieżne wykonanie zapytań z integracją odczytów fragmentów bazy danych współdzielonych przez zapytania. Metoda ta została pierwotnie zaproponowana dla algorytmu Apriori, gdzie integracji podlegały odczyty bazy danych do zliczania wystąpień kandydatów (Wojciechowski i Zakrzewicz, 2003). Ze względu na jej uniwer-

salny charakter, została szybko zaadaptowana dla algorytmu FP-growth, redukując liczbę odczytanych bloków dyskowych na etapie budowy struktur FP-tree dla współbieżnie wykonywanych zapytań (Wojciechowski et al., 2005).

Metoda Common Counting, jako sprowadzająca się do integracji odczytów danych z dysku, z pewnością nie wyczerpuje możliwości optymalizacji wykonania zbioru zapytań eksploracyjnych. Dalsza integracja operacji realizowanych przez poszczególne zapytania wymaga technik dedykowanych dla poszczególnych algorytmów (lub przynajmniej klas algorytmów). W tej pracy proponujemy nową metodę przetwarzania zbiorów zapytań eksploracyjnych dla algorytmu Apriori - Common Candidate Tree, realizującą ściślejszą integrację przetwarzanych zapytań poprzez współdzielenie struktur danych w pamięci operacyjnej. Wyniki eksperymentów pokazują, że Common Candidate Tree jest metodą bardziej wydajną niż Common Counting, a dodatkowo, ze względu na oszczędniejsze gospodarowanie pamięcią operacyjną, może być stosowana dla liczniejszych zbiorów zapytań.

2. Optymalizacja wykonania zbioru zapytań eksploracyjnych

2.1. Podstawowe definicje i sformułowanie problemu

Definicja 1. *Zapytanie eksploracyjne w kontekście problemu odkrywania zbiorów częstych to uporządkowana piątka* $dmq = (\mathcal{R}, a, \Sigma, \Phi, minsup)$*, gdzie* $\mathcal{R}$ *jest relacją bazy danych, a jest atrybutem relacji* $\mathcal{R}$*, którego wartościami są zbiory elementów,* Σ *jest wyrażeniem warunkowym dotyczącym atrybutów relacji* $\mathcal{R}$ *określanym mianem predykatu selekcji danych,* Φ *jest wyrażeniem warunkowym dotyczącym odkrywanych zbiorów częstych określanym mianem predykatu selekcji wzorców, a minsup to próg minimalnego wsparcia. Wynikiem zapytania dmq jest kolekcja zbiorów częstych odkrytych w* $\pi_a \sigma_\Sigma \mathcal{R}$*, spełniających predykat* Φ*, o wsparciu* $\geq minsup$ *(*π *i* σ *oznaczają relacyjne operacje projekcji i selekcji).*

Przykład 1. *Załóżmy, że dana jest relacja* $\mathcal{R}_1(a_1, a_2)$*, której atrybut* a_2 *przyjmuje jako wartości zbiory elementów, a atrybut* a_1 *jest typu całkowitoliczbowego. Zapytanie eksploracyjne* $dmq_1 = (\mathcal{R}_1, a_2, a_1 > 5, |itemset| < 4, 3\%)$ *reprezentuje zadanie odkrywania zbiorów częstych w kolekcji zbiorów stanowiących wartości atrybutu* a_2 *z wierszy relacji* $\mathcal{R}_1$ *spełniających warunek* $a_1 > 5$*. Zapytanie zwróci zbiory o wsparciu co najmniej 3% i jednocześnie zawierające mniej niż 4 elementy.*

Definicja 2. *Zbiór elementarnych predykatów selekcji danych dla zbioru zapytań eksploracyjnych* $DMQ = \{dmq_1, dmq_2, ..., dmq_n\}$ *to najmniejszy pod względem liczebności zbiór* $S = \{s_1, s_2, ..., s_k\}$ *predykatów selekcji danych z relacji* $\mathcal{R}$ *taki, że dla każdej pary* u, v $(u \neq v)$ *zachodzi* $\sigma_{s_u}\mathcal{R} \cap \sigma_{s_v}\mathcal{R} = \emptyset$ *i dla każdego zapytania* dmq_i *istnieją takie liczby całkowite* $a, b, ..., m$*, że* $\sigma_{\Sigma_i}\mathcal{R} = \sigma_{s_a}\mathcal{R} \cup \sigma_{s_b}\mathcal{R} \cup .. \cup \sigma_{s_m}\mathcal{R}$*. Zbiór elementarnych predykatów selekcji danych reprezentuje podział bazy danych na partycje wyznaczone przez nakładanie się źródłowych zbiorów danych zapytań.*

Przykład 2. *Załóżmy, że dana jest relacja* $\mathcal{R}_1(a_1, a_2)$ *i trzy zapytania eksploracyjne:* $dmq_1 = (\mathcal{R}_1, a_2, 5 \leq a_1 < 20, \emptyset, 3\%)$*,* $dmq_2 = (\mathcal{R}_1, a_2, 0 \leq a_1 < 15, \emptyset, 5\%)$*,*

$dmq_3 = (\mathcal{R}_1, a_2, 5 \leq a_1 < 15$ or $30 \leq a_1 < 40, \emptyset, 4\%)$. Zbiór elementarnych predykatów selekcji danych w tym wypadku to: $S = \{0 \leq a_1 < 5,\ 5 \leq a_1 < 15,\ 15 \leq a_1 < 20,\ 30 \leq a_1 < 40\}$.

PROBLEM. *Dla danego zbioru zapytań eksploracyjnych $DMQ = \{dmq_1, dmq_2, ..., dmq_n\}$, problem optymalizacji wykonania zbioru zapytań eksploracyjnych polega na wygenerowaniu algorytmu minimalizującego całkowity czas wykonania DMQ.*

W dalszych rozważaniach będziemy przyjmowali, że zbiór zapytań DMQ nie zawiera zapytań identycznych. W praktyce, po zebraniu zbioru zapytań do wspólnego wykonania system powinien usunąć powtarzające się zapytania we wstępnym kroku przetwarzania. Ponadto, w świetle wyników wcześniejszych badań (Morzy et al., 2000), sensowne jest również zastąpienie każdego podzbioru zapytań operujących na identycznym fragmencie bazy danych przez jedno zapytanie, którego wyniki będą mogły być wykorzystane do wygenerowania odpowiedzi na oryginalne zapytania poprzez prostą weryfikację predykatu selekcji wzorców i/lub progu minimalnego wsparcia. Takie nowe zapytanie powinno mieć próg minimalnego wsparcia równy najmniejszemu z progów minimalnego wsparcia zastępowanych zapytań i predykat selekcji wzorców w formie logicznej alternatywy predykatów selekcji wzorców zastępowanych zapytań.

2.2. Common Counting

Metoda Common Counting polega na współbieżnym wykonaniu zbioru zapytań eksploracyjnych algorytmem Apriori z integracją odczytów współdzielonych fragmentów bazy danych. Pseudokod metody Common Counting został przedstawiony na Rys. 1. W zapisie algorytmu *minsup* jest wartością progu minimalnego wsparcia wyrażoną jako bezwzględna liczba transakcji[3].

Metoda Common Counting iteracyjnie generuje i zlicza wystąpienia kandydatów dla wszystkich zapytań eksploracyjnych. Kandydatami jednoelementowymi dla wszystkich zapytań są wszystkie możliwe elementy (linie 1-2). Kandydaci o rozmiarze[4] k (k>1) są generowani ze zbiorów częstych o rozmiarze k-1, oddzielnie dla każdego z zapytań (linie 7-9). Generacja kandydatów (reprezentowana w zapisie algorytmu jako funkcja *generate_ candidates*()) przebiega dokładnie tak jak w oryginalnym algorytmie Apriori. Kandydaci wygenerowani dla każdego z zapytań są umieszczani w odrębnych drzewach haszowych. Iteracyjny proces generacji i zliczania wystąpień kandydatów kończy się gdy dla żadnego z zapytań nie uda się wygenerować kandydatów (warunek w linii 3).

Wystąpienia kandydatów dla wszystkich zapytań są zliczane podczas zintegrowanego odczytu bazy danych w następujący sposób: Dla każdego elementarnego predykatu selekcji danych, odpowiadająca mu partycja bazy danych jest odczytywana transakcja po transakcji. Dla każdej transakcji zwiększane są liczniki kandydatów w niej zawartych dla tych zapytań, które odwołują się do odczytywanej

[3] Typowo, użytkownicy podają wartość progu wsparcia w procentach. W takim wypadku wartość wyrażona procentowo jest przeliczana na bezwzględną wraz z pierwszym odczytem bazy danych realizowanym przez algorytm.

[4] Liczba elementów zbioru jest określana jako jego rozmiar.

Dane wejściowe: $DMQ = \{dmq_1, dmq_2, ..., dmq_n\}$,
gdzie $dmq_i = (\mathcal{R}, a, \Sigma_i, \Phi_i, minsup_i)$

(1) **for** (i=1; $i \leq n$; i++) **do**
(2) $\mathcal{C}_1^i$ = all possible 1-itemsets
(3) **for** (k=1; $\mathcal{C}_k^1 \cup \mathcal{C}_k^2 \cup .. \cup \mathcal{C}_k^n \neq \emptyset$; k++) **do begin**
(4) **for each** $s_j \in S$ **do begin**
(5) $\mathcal{CC} = \{\mathcal{C}_k^i : \sigma_{s_j}\mathcal{R} \subseteq \sigma_{\Sigma_i}\mathcal{R}\}$
(6) **if** $\mathcal{CC} \neq \emptyset$ **then** $count(\mathcal{CC}, \sigma_{s_j}\mathcal{R})$ **end**
(7) **for** (i=1; $i \leq n$; i++) **do begin**
(8) $\mathcal{F}_k^i = \{C \in \mathcal{C}_k^i : C.counter \geq minsup_i\}$
(9) $\mathcal{C}_{k+1}^i = generate\_candidates(\mathcal{F}_k^i)$ **end**
(10) **end**
(11) **for** (i=1; $i \leq n$; i++) **do**
(12) $Answer_i = \sigma_{\Phi_i} \bigcup_k \mathcal{F}_k^i$

Rysunek 1. Common Counting for Apriori

partycji (linie 4-6). Sprawdzanie którzy kandydaci zawierają się w danej transakcji odbywa się poprzez testowanie transakcji względem drzew haszowych każdego z zapytań odwołujących się do tej transakcji. Zliczanie wystąpień kandydatów jest reprezentowane w zapisie algorytmu jako funkcja *count*(). Zauważmy, że jeśli dany elementarny predykat selekcji danych jest współdzielony przez kilka zapytań, to przy każdorazowym zliczaniu wystąpień kandydatów odpowiadająca mu partycja bazy danych jest odczytywana tylko raz.

Common Counting nie zajmuje się obsługą predykatów selekcji wzorców Φ, ale pozwala na wykorzystanie do tego celu zaproponowanych dla Apriori metod opartych o modyfikację procedury generacji kandydatów oraz filtracji odkrytych wzorców dla tych ograniczeń, które nie mogą być efektywnie obsłużone w ramach algorytmu Apriori.

2.3. Przegląd pokrewnych problemów

Optymalizacja zbiorów zapytań (Sellis, 1988) była przedmiotem wielu prac w dziedzinie systemów baz danych, ukierunkowanych na wspólną optymalizację wielu zapytań poprzez wyodrębnienie i jednokrotne wykonanie powtarzających się wyrażeń składowych (Alsabbagh i Raghavan, 1994, Jarke, 1985). Główne zadania w optymalizacji zbiorów zapytań w systemach baz danych to identyfikacja wspólnych wyrażeń, takich jak: projekcje, selekcje, połączenia, półpołączenia i przesłania sieciowe, oraz konstrukcja globalnego planu wykonania. Na potrzeby optymalizacji wykonania zbioru zapytań w systemach baz danych zaproponowano wiele algorytmów heurystycznych (np. Roy et al., 2000). Zapytania eksploracyjne wymagają nowych metod optymalizacji wykonania ze względu na specjalistyczne algorytmy wykonania zapytania.

W dziedzinie eksploracji danych, poza rozważanym w niniejszej pracy problemem, optymalizacja zbiorów zapytań była rozważana jedynie w kontekście odkry-

wania wzorców częstych w wielu zbiorach danych (Jin et al., 2005). Celem w tym wypadku była redukcja wspólnych obliczeń pojawiających się w wielu różnych złożonych zapytaniach, z których każde porównywało wsparcie wzorców w różnych rozłącznych zbiorach danych. Stanowi to fundamentalną różnicę w stosunku do naszego problemu, gdzie każde zapytanie odnosi się do jednego zbioru danych i zbiory danych poszczególnych zapytań się nakładają.

Wcześniej, potrzeba optymalizacji wykonania zbioru zapytań została zauważona w pokrewnej do eksploracji danych dziedzinie programowania w logice, co zaowocowało metodą zbliżoną koncepcyjnie do Common Counting, polegającą na łączeniu podobnych zapytań w tzw. paczki zapytań (ang. query packs) (Blockeel et al., 2002).

W pewnym stopniu za wprowadzenie do optymalizacji wykonania zbioru zapytań eksploracyjnych w kontekście odkrywania zbiorów częstych możemy uznać techniki wykorzystywania pośrednich lub końcowych wyników poprzednich zapytań. Metody należące do tej kategorii to: inkrementalne odkrywanie zbiorów częstych (Cheung et al., 1996), przechowywanie w pamięci podręcznej pośrednich wyników poprzednich zapytań (Nag et al., 1999) oraz wykorzystywanie zmaterializowanych kompletnych (Baralis i Psaila, 1999, Meo, 2003, Morzy et al., 2000) lub skondensowanych (Jeudy i Boulicaut, 2002) wyników poprzednich zapytań w oparciu o analizę różnic składniowych między zapytaniami.

3. Common Candidate Tree: wspólne drzewo kandydatów dla zbioru przetwarzanych zapytań eksploracyjnych

Metoda Common Counting optymalizuje jedynie odczyty współdzielonych przez zapytania fragmentów bazy danych, realizując pozostałe operacje algorytmu Apriori oddzielnie dla każdego zapytania. Aby zwiększyć stopień współdzielenia przetwarzania między współbieżnie wykonywanymi zapytaniami, proponujemy nową metodę: Common Candidate Tree, opartą o współdzielenie pamięciowej struktury drzewa haszowego do składowania kandydatów. Takie rozwiązanie zachowuje integrację odczytów współdzielonych danych z dysku, a dodatkowo pozwala na integrację operacji testowania zawierania się kandydatów z poszczególnych zapytań w transakcji odczytanej z bazy danych.

Struktura drzewa haszowego w Common Candidate Tree pozostaje niezmieniona w stosunku do Common Counting i oryginalnego Apriori. Aby możliwe było współdzielenie jednego drzewa przez wszystkie zapytania, konieczne jest jedynie rozszerzenie struktury kandydatów, tak aby z kandydatem związany był nie pojedynczy licznik wystąpień, a wektor liczników (*counters*[]) - po jednym liczniku dla każdego zapytania. Ponadto, każdy kandydat będzie posiadał wektor flag logicznych (*fromQuery*[]) do oznaczenia, które zapytania wygenerowały danego kandydata. Flagi te będą ustawiane podczas łączenia zbiorów kandydatów poszczególnych zapytań w jeden zbiór kandydatów, który następnie będzie umieszczany we wspólnym drzewie haszowym.

Pseudokod metody Common Candidate Tree został przedstawiony na Rys. 2. Różnicą w stosunku do Common Counting, jest to, że zliczaniu podlega zintegro-

Dane wejściowe: $DMQ = \{dmq_1, dmq_2, ..., dmq_n\}$,
gdzie $dmq_i = (\mathcal{R}, a, \Sigma_i, \Phi_i, minsup_i)$

(1) $\mathcal{C}_1$ = all possible 1-itemsets
(2) **for** (k=1; $\mathcal{C}_k \neq \emptyset$; k++) **do begin**
(3) **for each** $s_j \in S$ **do begin**
(4) $\mathcal{CC} = \{C \in \mathcal{C}_k : \exists i\ C.fromQuery[i] = true \wedge \sigma_{s_j}\mathcal{R} \subseteq \sigma_{\Sigma_i}\mathcal{R}\}$
(5) **if** $\mathcal{CC} \neq \emptyset$ **then** $count(\mathcal{CC}, \sigma_{s_j}\mathcal{R})$ **end**
(6) **for** (i=1; $i \leq n$; i++) **do begin**
(7) $\mathcal{F}_k^i = \{C \in \mathcal{C}_k : C.counters[i] \geq minsup_i\}$
(8) $\mathcal{C}_{k+1}^i = generate\_candidates(\mathcal{F}_k^i)$ **end**
(9) $\mathcal{C}_{k+1} = \mathcal{C}_{k+1}^1 \cup \mathcal{C}_{k+1}^2 \cup .. \cup \mathcal{C}_{k+1}^n$
(10) **end**
(11) **for** (i=1; $i \leq n$; i++) **do**
(12) $Answer_i = \sigma_{\Phi_i} \bigcup_k \mathcal{F}_k^i$

Rysunek 2. Common Candidate Tree

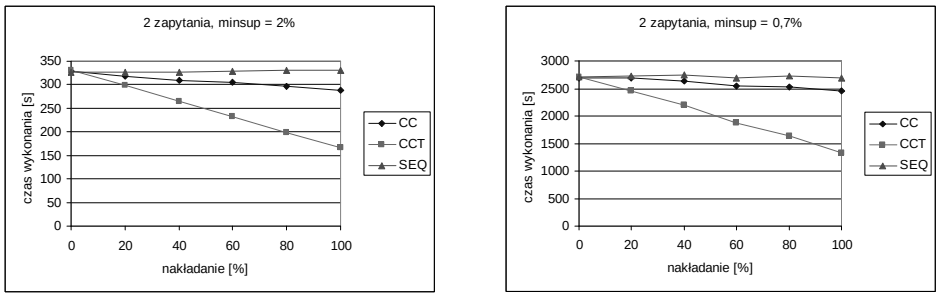
wany zbiór kandydatów (linie 1 i 9), choć generacja kandydatów i selekcja zbiorów częstych są w dalszym ciągu realizowane odrębnie dla poszczególnych zapytań (linie 6-8). W fazie zliczania wystąpień kandydatów, podczas odczytu danej partycji bazy danych uwzględniani są tylko ci kandydaci, którzy zostali wygenerowani przez któreś z zapytań odwołujących się do tej partycji i jednocześnie w przypadku zawierania się kandydata w odczytanej transakcji zwiększane są jedynie liczniki kandydatów związane z takimi zapytaniami (linie 3-5).

Oceniając metodę Common Candidate Tree należy zwrócić uwagę, że ściślejsza integracja przetwarzania zapytań nie jest jej jedyną zaletą w porównaniu z Common Counting. Istotną wadą Common Counting, ograniczającą jej stosowalność dla dużych zbiorów zapytań, jest konieczność jednoczesnego utrzymywania w pamięci wielu drzew haszowych. Problem ten był dotychczas rozwiązywany poprzez podział oryginalnego zbioru zapytań na rozłączne podzbiory i uruchomienie metody Common Counting oddzielnie dla każdego z podzbiorów (Boiński et al., 2006, Wojciechowski i Zakrzewicz, 2005). Common Candidate Tree wykorzystuje jedno drzewo haszowe, o niezmienionej strukturze węzłów pośrednich, a jedynie rozbudowanych kandydatach, co powinno zwiększyć zakres jego stosowalności bez konieczności podziału oryginalnego zbioru zapytań.

Podobnie jak Common Counting, Common Candidate Tree nie zajmuje się obsługą predykatów selekcji wzorców Φ, ale pozwala na wykorzystanie do tego celu technik zaproponowanych dla Apriori, gdyż metoda generacji kandydatów nie podlega optymalizacji przez Common Candidate Tree.

4. Wyniki eksperymentów

W celu przetestowania wydajności i wymagań pamięciowych algorytmu Common Candidate Tree na tle dotychczas najwydajniejszego algorytmu Common Counting oraz sekwencyjnego wykonania zapytań, przeprowadziliśmy szereg ekspery-



Rysunek 3. Czasy wykonania algorytmów dla 2 zapytań i różnych stopni nakładania

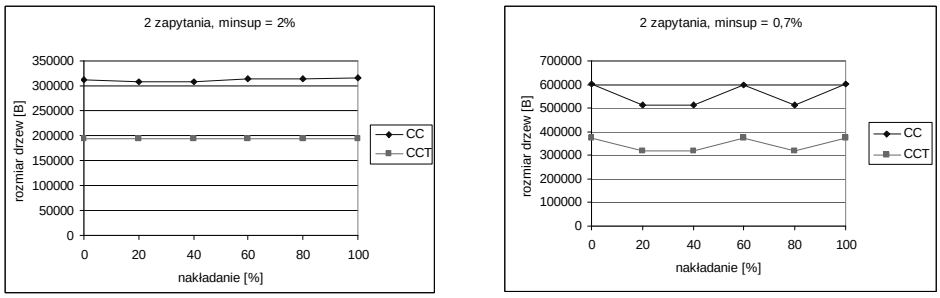
mentów na syntetycznym zbiorze danych wygenerowanym za pomocą generatora GEN (Agrawal et al., 1996). Przy generacji zbioru danych zostały wykorzystane następujące wartości parametrów generatora: liczba transakcji = 1000000, średnia liczba elementów w transakcji = 8, liczba różnych elementów = 1000, liczba wzorców (tj. zbiorów częstych) = 1500, średnia liczba elementów we wzorcu = 4. Zbiór danych został zapisany w pliku na dysku lokalnym, zajmując około 97 MB. Eksperymenty zostały zrealizowane na komputerze klasy PC z procesorem Athlon 1700+ i 512 MB pamięci RAM, pracującym pod kontrolą systemu operacyjnego Microsoft Windows XP.

W trakcie eksperymentów zmienialiśmy liczbę współbieżnie wykonywanych zapytań, próg minimalnego wsparcia oraz stopień nakładania się źródłowych zbiorów danych dla zapytań. Zbiór danych źródłowych każdego z zapytań stanowił podzbiór 500000 kolejnych transakcji z wygenerowanego zbioru danych. Choć żadna z porównywanych metod tego nie wymaga, we wszystkich eksperymentach wszystkie zapytania stanowiące zbiór zapytań do wspólnego wykonania wykorzystywały ten sam próg minimalnego wsparcia, aby jego potencjalny wpływ na zachowanie poszczególnych metod i różnica w wydajności między Common Counting i Common Candidate Tree były łatwiejsze do zaobserwowania[5].

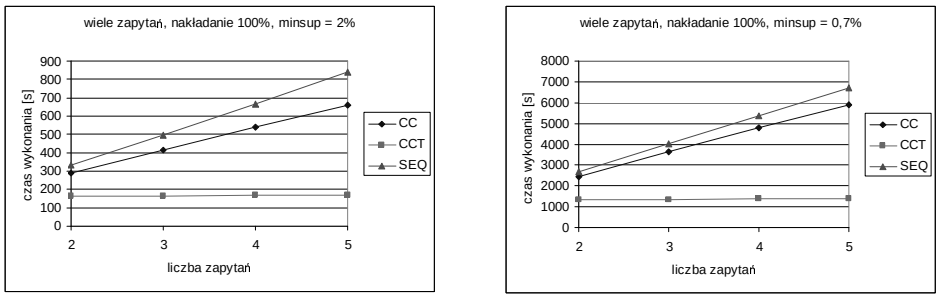
Celem pierwszej serii eksperymentów było sprawdzenie wpływu stopnia nakładania się źródłowych zbiorów danych zapytań na czasy wykonania algorytmów Common Counting (CC) i Common Candidate Tree (CCT) w porównaniu z wykonaniem sekwencyjnym (SEQ). Jednocześnie w celu porównania wymagań pamięciowych metod CC i CCT mierzony był rozmiar drzew haszowych (węzły + kandydaci). Eksperymenty zostały wykonane dla dwóch nakładających się zapytań przy dwóch różnych wartościach progów wsparcia: 2% i 0,7%. Progi te zostały dobrane tak, aby prowadziły do znacząco różnej liczby iteracji Apriori (2 iteracje dla progu 2% i 7-8 iteracji dla progu 0,7%).

Rys. 3 przedstawia wykresy ilustrujące zależność czasów wykonania porówny-

[5] Im większa różnica między wartościami progu wsparcia dla poszczególnych zapytań, tym większej różnicy w liczbach iteracji Apriori dla zapytań można oczekiwać. Zarówno Common Counting jak i Common Candidate Tree redukują czas przetwarzania tylko w tych iteracjach Apriori, w których wykonywane są co najmniej 2 zapytania.



Rysunek 4. Średnie sumaryczne rozmiary drzew haszowych dla 2 zapytań i różnych stopni nakładania



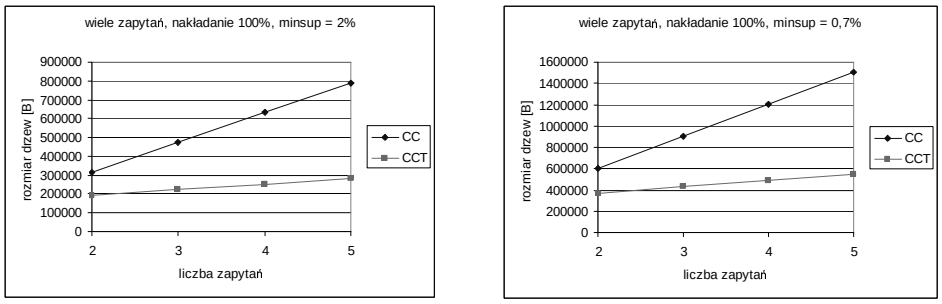
Rysunek 5. Czasy wykonania algorytmów dla wielu identycznych zapytań

wanych algorytmów od stopnia nakładania się danych źródłowych zapytań. Czasy wykonania CC i CCT maleją liniowo wraz ze wzrostem stopnia nakładania, przy czym CCT jest zdecydowanie bardziej wydajny niż CC, a względne różnice w czasach wykonania algorytmów są zbliżone dla obu testowanych progów wsparcia.

Na Rys. 4 przedstawione zostały średnie sumaryczne rozmiary drzew haszowych dla CC i CCT (suma rozmiarów wszystkich drzew ze wszystkich iteracji podzielona przez liczbę iteracji). CCT dla dwóch zapytań zredukował wykorzystanie pamięci w porównaniu z CC o 33% do 40%. Zaobserwowane różnice w wymaganiach pamięciowych obu algorytmów dla wsparcia 0,7% przy różnych stopniach nakładania wynikały z różnej charakterystyki poszczególnych fragmentów bazy danych.

Druga seria eksperymentów miała na celu przetestowanie skalowalności metod CC i CCT względem rosnącej liczby wspólnie wykonywanych zapytań. Ze względu na wielość możliwych konfiguracji nakładania się źródłowych zbiorów danych w przypadku więcej niż dwóch zapytań, aby uchwycić rzeczywisty wpływ liczby zapytań zdecydowaliśmy się na przeprowadzenie testów tylko dla zbiorów identycznych zapytań (stopień nakładania się zawsze równy 100%).

Rys. 5 przedstawia wykresy ilustrujące zależność czasów wykonania porównywanych algorytmów od liczby zapytań. Czas wykonania CCT rośnie nieznacznie wraz ze zwiększaniem liczby zapytań, podczas gdy czas wykonania CC rośnie nie-



Rysunek 6. Średnie sumaryczne rozmiary drzew haszowych dla wielu identycznych zapytań

wiele wolniej niż w przypadku wykonania sekwencyjnego.

Na Rys. 6 przedstawione zostały sumaryczne rozmiary drzew haszowych dla CC i CCT przy zwiększającej się liczby zapytań. Wzrost wymagań pamięciowych CCT wraz ze wzrostem liczby zapytań jest zdecydowanie mniejszy niż w przypadku CC. Oczywiście należy pamiętać, że charakter eksperymentu (identyczne zapytania) faworyzował CCT, gdyż każde dodatkowe zapytanie powodowało dodanie nowego drzewa w przypadku CC, a jedynie zwiększenie rozmiarów wektorów związanych z kandydatami w przypadku CCT.

5. Podsumowanie

W niniejszej pracy rozważaliśmy przetwarzanie zbiorów zapytań eksploracyjnych dla problemu odkrywania zbiorów częstych. Zaproponowaliśmy nową metodę wykonania zbioru zapytań eksploracyjnych algorytmem Apriori, która została nazwana Common Candidate Tree z racji wykorzystywania wspólnej struktury drzewa haszowego dla współbieżnie wykonywanych zapytań. Wyniki eksperymentów pokazały, że Common Candidate Tree jest metodą znacznie bardziej wydajną, lepiej skalującą się wraz ze wzrostem liczby zapytań do wykonania, a ponadto oszczędniej wykorzystującą pamięć operacyjną niż dotychczas najlepsza metoda Common Counting.

W przyszłości planujemy kontynuować pracę nad coraz bardziej ścisłą integracją operacji realizowanych podczas wykonywania zbiorów zapytań eksploracyjnych zarówno rozważanym w niniejszej pracy algorytmem Apriori jak i konkurencyjnym wobec niego algorytmem FP-growth.

Literatura

Agrawal, R., Imielinski, T. i Swami, A (1993) Mining Association Rules Between Sets of Items in Large Databases. *Proceedings of the 1993 ACM SIGMOD Int'l Conf. on Management of Data*, 207–216.

Agrawal, R., Mehta, M., Shafer, J., Srikant, R., Arning, A. i Bollinger, T. (1996) The Quest Data Mining System. *Proc. of the 2nd Int'l Conf. on Knowledge Discovery in Databases and Data Mining*, 244–249.

Agrawal, R. i Srikant, R. (1994) Fast Algorithms for Mining Association Rules. *Proc. of the 20th Int'l Conf. on Very Large Data Bases*, 487–499.

Alsabbagh, J.R. i Raghavan, V.V. (1994) Analysis of common subexpression exploitation models in multiple-query processing. *Proceedings of the 10th ICDE Conference*, 488–497.

Baralis, E. i Psaila, G. (1999) Incremental Refinement of Mining Queries. *Proceedings of the 1st DaWaK Conference*, 173–182.

Blockeel, H., Dehaspe, L., Demoen, B., Janssens, G., Ramon, J. i Vandecasteele, H. (2002) Improving the Efficiency of Inductive Logic Programming Through the Use of Query Packs. *Journal of Artificial Intelligence Research* **16**, 135–166.

Boiński, P., Wojciechowski, M. i Zakrzewicz, M. (2006) A Greedy Approach to Concurrent Processing of Frequent Itemset Queries. *Proc. of the 8th International Conference on Data Warehousing and Knowledge Discovery*, 292–301.

Cheung, D.W., Han, J., Ng, V. i Wong, C.Y. (1996) Maintenance of Discovered Association Rules in Large Databases: An Incremental Updating Technique. *Proceedings of the 12th ICDE Conference*, 106–114.

Han, J., Pei, J. i Yin, Y. (2000) Mining frequent patterns without candidate generation. *Proceedings of the 2000 ACM SIGMOD Int'l Conference on Management of Data*, 1–12.

Imielinski, T. i Mannila, H. (1996) A Database Perspective on Knowledge Discovery. *Communications of the ACM* **39**, 11, 58–64.

Jarke, M. (1985) Common subexpression isolation in multiple query optimization. *Kim, W., Reiner, D.S. (Eds.), Query Processing in Database Systems*, 191–205.

Jeudy, B. i Boulicaut, J-F. (2002) Using condensed representations for interactive association rule mining. *Proceedings of the 6th European Conference on Principles and Practice of Knowledge Discovery in Databases*, 225–236.

Jin, R., Sinha, K. i Agrawal, G. (2005) Simultaneous Optimization of Complex Mining Tasks with a Knowledgeable Cache. *Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 600–605.

Meo, R. (2003) Optimization of a Language for Data Mining. *Proceedings of the ACM Symposium on Applied Computing - Data Mining Track*, 437–444.

Morzy, T., Wojciechowski, M. i Zakrzewicz, M. (2000) Materialized Data Mining Views. *Proceedings of the 4th PKDD Conference*, 65–74.

Nag, B., Deshpande, P.M. i DeWitt, D.J. (1999) Using a Knowledge Cache for Interactive Discovery of Association Rules. *Proceedings of the 5th KDD Conference*, 244–253.

Pei, J. i Han, J. (2000) Can We Push More Constraints into Frequent Pattern Mining? *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 350–354.

Roy, P., Seshadri, S., Sundarshan, S. i Bhobe, S. (2000) Efficient and Extensible Algorithms for Multi Query Optimization. *Proceedings of the 2000 ACM SIGMOD Int'l Conference on Management of Data*, 249–260.

Sellis, T. (1988) Multiple-query optimization. *ACM Transactions on Database Systems* **13**, 1, 23–52.

Wojciechowski, M., Gałęcki, K. i Gawronek, K. (2005) Concurrent Processing of Frequent Itemset Queries Using FP-Growth Algorithm. *Proceedings of the 1st ADBIS Workshop on Data Mining and Knowledge Discovery*, 35–46.

Wojciechowski, M. i Zakrzewicz, M. (2003) Evaluation of Common Counting Method for Concurrent Data Mining Queries. *Proc. of the 7th ADBIS Conference*, 76–87.

Wojciechowski, M. i Zakrzewicz, M. (2005) On Multiple Query Optimization in Data Mining. *Proceedings of the 9th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, 696–701.

Zheng, Z., Kohavi, R. i Mason, L. (2001) Real world performance of association rule algorithms. *Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 401–406.

Integration of Candidate Hash Trees in Concurrent Processing of Frequent Itemset Queries Using Apriori

In this paper we address the problem of processing of batches of frequent itemset queries using the Apriori algorithm. The best solution of this problem proposed so far is Common Counting, which consists in concurrent execution of the queries using Apriori with the integration of scans of the parts of the database shared among the queries. In this paper we propose a new method - Common Candidate Tree, offering a more tight integration of the concurrently processed queries by sharing memory data structures, i.e., candidate hash trees. The experiments show that Common Candidate Tree outperforms Common Counting in terms of execution time. Moreover, thanks to smaller memory consumption, Common Candidate Tree can be applied to larger batches of queries.