Ścieżki dostępu do danych dla algorytmów przetwarzania zbiorów zapytań eksploracyjnych

Piotr Jędrzejczak, Marek Wojciechowski

Politechnika Poznańska, Piotrowo 2, 60-965 Poznań, Polska
Marek.Wojciechowski@cs.put.poznan.pl

Streszczenie. Odkrywanie zbiorów częstych można traktować jako realizację zaawansowanych zapytań do bazy danych, gdzie użytkownik specyfikuje ograniczenia dotyczące selekcji danych źródłowych oraz odkrywanych wzorców. Ponieważ takie zapytania eksploracyjne mogą być wysyłane do bazy danych w trybie wsadowym, np. w czasie niewielkiego obciążenia serwera bazy danych, zaproponowano szereg algorytmów przetwarzania zbiorów zapytań eksploracyjnych. Metody te wykorzystują współdzielenie danych źródłowych między zapytaniami i operują na logicznych partycjach danych wyznaczonych przez nakładanie się źródłowych zbiorów danych. Wszystkie zaproponowane dotychczas algorytmy były testowane jedynie w warunkach dostępności bezpośrednich ścieżek dostępu do poszczególnych partycji, zazwyczaj na plikach płaskich. W praktyce dane poddawane eksploracji są najczęściej składowane w bazach danych, gdzie po pierwsze potencjalnie dostępnych jest wiele różnych ścieżek dostępu do danych, a po drugie dla konkretnych warunków selekcji danych na ogół dostępne są tylko niektóre z nich. Celem niniejszego artykułu jest teoretyczna i praktyczna analiza zachowania się algorytmów przetwarzania zbiorów zapytań eksploracyjnych w kontekście różnych ścieżek dostępu do danych.

1 Wstęp

Odkrywanie zbiorów częstych (ang. frequent itemset mining) jest jedną z podstawowych technik eksploracji danych (ang. data mining), stosowaną zarówno samodzielnie jak i jako pierwszy etap w odkrywaniu reguł asocjacyjnych (ang. association rules) [1]. Problem ten został pierwotnie sformułowany w kontekście analizy koszyka zakupów, ale szybko znalazł liczne inne zastosowania, do których należą m. in. analiza danych medycznych, społecznych i telekomunikacyjnych oraz analiza zachowań użytkowników WWW. W skrócie, odkrywanie zbiorów częstych polega na odkrywaniu najczęściej występujących podzbiorów w bazie danych zawierającej zbiory nazywane transakcjami[1].

Odkrywanie zbiorów częstych może być postrzegane jako realizacja zaawansowanych zapytań do bazy danych, gdzie użytkownik specyfikuje ograniczenia dotyczące selekcji danych źródłowych oraz zawężające zbiór odkrywanych wzorców do tych, które z jego

[1] Nazewnictwo obowiązujące w literaturze poświęconej problemowi odkrywania zbiorów częstych odwołuje się często do pojęć używanych w analizie koszyka zakupów, np. baza danych jako kolekcja transakcji klientów.

punktu widzenia są interesujące [2]. Zapytania reprezentujące zadania odkrywania zbiorów częstych stanowią szczególny przypadek tzw. zapytań eksploracyjnych (ang. data mining queries).

W literaturze zaproponowano wiele algorytmów odkrywania zbiorów częstych, różniących się przyjętym formatem danych źródłowych, wykorzystywanymi strukturami pamięciowymi, technikami redukcji kosztów I/O itp. Dwie najważniejsze klasy algorytmów wyznacza jednak strategia przeszukiwania przestrzeni rozwiązań. Algorytmy rodziny "level-wise", reprezentowane przez klasyczny algorytm *Apriori* [3], stosują strategię przeszukiwania wszerz, podczas gdy metody z rodziny "pattern-growth", spośród których najbardziej znany jest algorytm *FP-growth* [4], przeszukują przestrzeń w głąb.

Chociaż zaproponowano już wiele różnych algorytmów odkrywania zbiorów częstych, efektywne wyekstrahowanie wiedzy z dużych wolumenów danych nie jest zadaniem prostym i proces taki często wymaga dużych nakładów czasowych. Długi czas wykonywania zapytań eksploracyjnych dotyczących odkrywania zbiorów częstych przy pomocy aktualnie znanych algorytmów powoduje, iż zapytania te są nierzadko zbierane i wykonywane wsadowo w dogodnym czasie, gdy obciążenie systemu jest niższe (np. w nocy). Zapytania z takiego zebranego zbioru mogą wykazywać pewne podobieństwa między sobą, np. poprzez odwoływanie się do tych samych zakresów danych. Właśnie to spostrzeżenie stanowi motywację do zastąpienia sekwencyjnego wykonania takiego zbioru zadań wykonaniem współbieżnym, które wykorzystując fakt współdzielenia danych przez poszczególne zadania pozwoli na efektywniejsze przetwarzanie, zmniejszając tym samym całkowity wymagany na znalezienie odpowiedzi czas [5].

Jak dotąd opracowano kilka metod optymalizacji wykonania zbioru zapytań eksploracyjnych dotyczących odkrywania zbiorów częstych, wykorzystujących nakładanie się na siebie zakresów danych źródłowych poszczególnych zapytań: *Mine Merge* - niezależna od wykorzystywanego algorytmu odkrywania zbiorów częstych [5], *Common Counting* [5] i *Common Candidate Tree* [6] przeznaczone dla algorytmu *Apriori* oraz *Common Building* i *Common FP-tree* [7] opracowane z myślą o algorytmie *FP-growth*. Dla metod tych, oprócz niezbędnej analizy teoretycznej, przeprowadzone zostały także praktyczne testy wydajnościowe z wykorzystaniem plików płaskich przy założeniu dostępności bezpośrednich ścieżek dostępu do partycji danych wyznaczonych przez nakładanie się danych źródłowych zapytań. W rzeczywistości jednak dane poddawane eksploracji są najczęściej przechowywane w bazach danych, gdzie po pierwsze potencjalnie dostępnych jest wiele różnych ścieżek dostępu do danych, a po drugie dla konkretnych warunków selekcji danych na ogół dostępne są tylko niektóre z nich.

Celem niniejszego artykułu jest teoretyczna i praktyczna analiza zachowania się zaproponowanych dotychczas algorytmów przetwarzania zbiorów zapytań eksploracyjnych dotyczących odkrywania zbiorów częstych w kontekście różnych ścieżek dostępu do danych. Ze względu na fakt, że metody dla algorytmu *FP-growth* są adaptacjami metod wcześniej opracowanych dla algorytmu *Apriori*, analiza będzie dokonana jedynie w kontekście algorytmu *Apriori*, który jest najpowszechniej implementowanym algorytmem odkrywania zbiorów częstych, a ponadto ze względu na wielokrotne odczyty danych źródłowych powinien w większym stopniu niż *FP-growth* uwypuklić wpływ ścieżek dostępu na całkowity czas procesu eksploracji danych.

Artykuł wpasowuje się w tematykę optymalizacji zapytań eksploracyjnych, skupiając się na rozwiązaniach, które można interpretować jako optymalizację zbiorów zapytań (ang. multiple-query optimization). Optymalizacja zbiorów zapytań była przedmiotem intensywnych badań w kontekście systemów baz danych [8], których celem było zbudowanie globalnego planu wykonania wykorzystującego współdzielenie operacji poszczególnych zapytań. Na gruncie eksploracji danych, poza problemem poruszanym w niniejszym artykule, optymalizacja zbiorów zapytań rozważana była jedynie w znacząco różnym problemie odkrywania zbiorów częstych w wielu zbiorach danych [9]. Podobnych rozwiązań do przedstawianych w tej pracy można natomiast doszukać się w pokrewnej eksploracji danych dziedzinie programowania w logice, gdzie zaproponowano metodę zbliżoną koncepcyjnie do metody *Common Counting* [10].

2 Podstawy teoretyczne

2.1 Podstawowe pojęcia

Niech $I = \{i_1, i_2, ..., i_n\}$ będzie pewnym zbiorem literałów, nazywanych dalej *elementami*. *Transakcją* T nazywamy taki zbiór elementów, że $T \subseteq I$ i $T \neq \emptyset$. Mówimy, że transakcja T *wspiera* element $x \in I$, jeżeli $x \in T$. Transakcja T wspiera zbiór $X \subseteq I$, jeżeli wspiera każdy element ze zbioru X, tj. $X \subseteq T$. *Wsparciem bezwzględnym* zbioru X w pewnym zbiorze transakcji D nazywamy liczbę transakcji $T \in D$ wspierających X. *Wsparciem względnym* zbioru X w pewnym zbiorze transakcji D nazywamy stosunek liczby transakcji $T \in D$ wspierających X do liczby wszystkich transakcji należących do D i wyrażamy w procentach. Zbiór $X \subseteq I$, którego wsparcie jest niemniejsze od pewnej przyjętej wartości *minimalnego wsparcia* (ang. minimum support), nazywamy *zbiorem częstym*. *Rozmiarem* zbioru X nazywamy liczbę znajdujących się w nim elementów.

2.2 Sformułowanie problemu

Zapytaniem eksploracyjnym w kontekście problemu odkrywania zbiorów częstych nazywamy uporządkowaną piątkę $dmq = (R, a, \Sigma, \Phi, minsup)$, gdzie R jest relacją bazy danych, a jest atrybutem relacji R, którego wartościami są zbiory elementów, Σ jest wyrażeniem warunkowym dotyczącym atrybutów relacji 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 $\pi_a \sigma_\Sigma R$, spełniających predykat Φ, o wsparciu $\geq minsup$ (π i σ oznaczają relacyjne operacje projekcji i selekcji).

Zbiór elementarnych predykatów selekcji danych dla danego 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 R taki, że dla każdej pary u, v ($u \neq v$) zachodzi $\sigma_{s_u} R \cap \sigma_{s_v} R = \emptyset$ i dla każdego zapytania dmq_i istnieją takie liczby całkowite $a, b, ..., m$, że $\sigma_{\Sigma_i} R = \sigma_{s_a} R \cup \sigma_{s_b} R \cup .. \cup \sigma_{s_m} R$. Zbiór elementarnych predykatów selekcji danych reprezentuje podział bazy danych na partycje wyznaczone przez nakładanie się źródłowych zbiorów danych zapytań.

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.

2.3 Algorytm Apriori

Wprowadzony w [3] algorytm *Apriori* działa iteracyjnie, każdorazowo odkrywając zbiory częste o coraz większym rozmiarze. Opiera się on na spostrzeżeniu, iż każdy podzbiór zbioru częstego jest także zbiorem częstym. Podczas pierwszej iteracji zliczana jest liczba wystąpień każdego pojedynczego elementu i na jej podstawie wyznaczane są 1-elementowe zbiory częste. Kolejne iteracje składają się z dwóch faz: fazy generowania zbiorów kandydackich na podstawie zbiorów częstych odkrytych w poprzedniej iteracji oraz fazy ich weryfikacji, podczas której zliczane są wystąpienia i odrzucane są te zbiory, które nie posiadają wymaganego minimalnego wsparcia. Proces ten powtarzany jest do momentu, aż nie zostaną odkryte nowe zbiory częste. Wpływ na wydajność algorytmu *Apriori* mają przede wszystkim odczyty danych z dysku oraz zliczanie kandydatów. W celu efektywnego sprawdzenia, które ze zbiorów kandydackich zawierają się w danej transakcji, zbiory te przechowywane są w drzewie haszowym.

3 Metody przetwarzania zbioru zapytań eksploracyjnych dla algorytmu Apriori

3.1 Wykonanie sekwencyjne

Metoda naiwna wykonania zbioru zapytań eksploracyjnych dla problemu odkrywania zbiorów częstych sprowadza się do wykonania tych zapytań sekwencyjnie przy użyciu podstawowego algorytmu odkrywania zbiorów częstych np. *Apriori*. Każde zapytanie wykonywane jest osobno, zatem jakiekolwiek podobieństwa wykazywane przez te zapytania są pomijane. Chociaż metoda ta nie jest efektywnym rozwiązaniem postawionego problemu, stanowi naturalny punkt odniesienia przy ocenie pozostałych metod i uwzględnienie jej w eksperymentach pomiarowych pozwoli uogólnić wyciągnięte odnośnie ścieżek dostępu wnioski.

3.2 Algorytm Common Counting

Algorytm *Common Counting*, zaprezentowany w [5], koncentruje się na zmniejszeniu liczby odczytów danych poprzez zlikwidowanie potrzeby wielokrotnego odczytu wspólnych partycji bazy danych, tzn. takich, do których odwołuje się więcej niż jedno zadanie ze zbioru. Wszystkie zadania wykonywane są równocześnie w sposób zbliżony do *Apriori*, czyli poprzez iteracyjne powtarzanie faz generowania zbiorów kandydackich oraz faz weryfikacji aż do momentu, gdy dla żadnego z zadań nie zostaną odkryte nowe zbiory częste. W trakcie fazy generowania zbiorów kandydackich generowane są zbiory kolejno dla wszystkich wykonywanych zadań. Sam proces generowania zbiorów dla pojedynczego zadania przebiega identycznie jak w *Apriori*. Wygenerowane dla poszczególnych zadań zbiory kandydackie przechowywane są w osobnych drzewach haszowych.

Faza weryfikacji oblicza wsparcie zbiorów kandydackich dla wszystkich zadań jednocześnie podczas jednego wspólnego odczytu danych. Odczytywane są wszystkie partycje, do których odwołuje się chociaż jedno zadanie ze zbioru. Każda partycja bazy danych odczytywana jest w pojedynczej iteracji algorytmu dokładnie raz, skutecznie redukując w ten sposób liczbę wykonywanych operacji wejścia-wyjścia w porównaniu z wykonaniem sekwencyjnym.

3.3 Algorytm Common Candidate Tree

Algorytm *Common Counting* optymalizuje tylko odczyty bazy danych, wykonując pozostałe operacje osobno dla każdego zadania. Algorytm *Common Candidate Tree* [6] rozszerza *Common Counting* o uwspólnienie wykorzystywanych struktur danych. Tak jak w przypadku *Common Counting*, wszystkie zadania wykonywane są jednocześnie i każda partycja bazy danych odczytywana jest tylko raz w pojedynczej iteracji algorytmu. Różnica polega na zastosowaniu jednego wspólnego drzewa haszowego dla wszystkich zadań. Sama struktura drzewa haszowego w *Common Candidate Tree* jest identyczna jak w przypadku *Common Counting* i oryginalnego *Apriori*. Zmodyfikowana została natomiast struktura samych zbiorów kandydackich, tak aby z każdym z nich związana była tablica liczników – po jednym dla każdego zapytania oraz tablica zmiennych boolowskich, pozwalająca zidentyfikować zadania, które dany zbiór kandydacki wygenerowały. Faza generowania zbiorów kandydackich przebiega analogicznie jak w algorytmie *Common Counting*. Dodatkowo, po wygenerowaniu zbiorów dla wszystkich zadań, zbiory te są scalane do opisanej powyżej rozszerzonej reprezentacji i umieszczane we wspólnym drzewie haszowym. Faza weryfikacji sprowadza się do odczytu wszystkich transakcji z wszystkich partycji bazy danych i przeszukania drzewa dla każdej z nich. Dla każdego znalezionego zbioru kandydackiego inkrementowane są tylko liczniki powiązane z tymi zadaniami, które odwołują się do aktualnie odczytywanej partycji i dla których dany zbiór został wygenerowany.

3.4 Algorytm *Mine Merge*

Algorytm *Mine Merge* prezentuje zupełnie inne podejście do wykonywania zbioru zadań niż wszystkie poprzednie metody. Podstawę zasady jego działania stanowi fakt, iż w przypadku bazy danych podzielonej na rozłączne partycje, zbiór będący zbiorem częstym w całej bazie danych musi być częsty w co najmniej jednej z tych partycji. Przebieg algorytmu jest następujący. Najpierw z wejściowego zbioru zadań generowane są zapytania pośrednie, z których każde opiera się na pojedynczym elementarnym predykacie selekcji (czyli każde z nich odwołuje się do pojedynczej partycji). Następnie zapytania te wykonywane są sekwencyjnie (np. przy użyciu *Apriori*), a z odkrytych przez nie zbiorów częstych tworzona jest globalna lista zbiorów kandydackich dla każdego z oryginalnych zadań (lista taka powstaje poprzez zsumowanie wszystkich zbiorów częstych z wszystkich partycji, do których dane zadanie się odwołuje). Na koniec wykonywany jest pojedynczy odczyt danych, podczas którego następuje obliczenie wsparć poszczególnych zbiorów kandydackich i wyznaczenie zbiorów częstych będących rozwiązaniem każdego z oryginalnych zapytań.

4 Dostęp do danych

4.1 Struktury danych i ścieżki dostępu

Dostępne na rynku systemy zarządzania bazą danych oparte o model relacyjny oferują zbliżoną podstawową funkcjonalność w zakresie składowania danych i dostępu do nich, przez co niniejsze rozważania mają w dużym stopniu charakter uniwersalny. Ponieważ każdy z nich posiada jednak swoją specyfikę, dalsza analiza przeprowadzona będzie w oparciu o jeden konkretny system zarządzania bazą danych - Oracle 11g. Ten sam system wykorzystany zostanie podczas testów wydajnościowych.

W bazie danych Oracle domyślnie tabela zorganizowana jest wewnętrznie jako sterta (ang. heap), co oznacza że nie jest gwarantowane żadne fizyczne uporządkowanie rekordów tabeli na dysku. Dlatego też, bez pomocy dodatkowych towarzyszących tabeli struktur danych, w celu odszukania rekordów spełniających podane przez użytkownika w zapytaniu kryteria selekcji system musi dokonać pełnego przeglądu tabeli. Co więcej, nie można również zakładać żadnego logicznego porządku wierszy zwracanych przez zapytanie SQL, jeśli nie zlecono w nim żadnego porządku jawnie poprzez klauzulę ORDER BY. Opcjonalnie z tabelą powiązane mogą być indeksy, które w odpowiednich przypadkach umożliwiają szybszy dostęp do danych. Utworzenie indeksu na jednej bądź kilku kolumnach tabeli pozwala na odczyt podzbioru rekordów znajdujących się w tabeli bez konieczności przeglądania wszystkich bloków danych – dla danej zaindeksowanej wartości indeks wskazuje bezpośrednie lokalizacje wszystkich rekordów, które tę wartość zawierają. Podstawowym rodzajem indeksów w Oracle są indeksy o strukturze B-drzewa, choć dostępne są również indeksy bitmapowe oraz zaawansowane specjalistyczne struktury indeksowe wykorzystywane do indeksowania specyficznych rodzajów danych np. tekstowych czy przestrzennych.

Typowy sposób wykorzystania indeksu polega na odszukaniu w indeksie identyfikatorów/adresów rekordów (w Oracle – ROWID) spełniających warunki selekcji zapytania, a następnie bezpośrednim dostępie do rekordów tabeli w oparciu o informacje z indeksu. W tym względzie dostęp do danych z wykorzystaniem indeksu stanowi alternatywę dla pełnego przeglądu tabeli i jest opłacalny pod warunkiem odpowiedniej selektywności indeksu[2]. W rzeczywistości system zarządzania bazą danych Oracle różnicuje ścieżki dostępu do indeksu w zależności od unikalności/nieunikalności kluczy indeksu oraz rodzaju warunku selekcji (równościowy/zakresowy), a ponadto w pewnych scenariuszach realizuje pełny przegląd indeksu (w porządku indeksowym lub nie). Niuanse związane ze ścieżkami dostępu do indeksu w systemie Oracle wykraczają poza zakres niniejszego artykułu. We wszystkich eksperymentach indeks będzie użyty do wyszukania odpowiednich rekordów tabeli w oparciu o warunki zakresowe dotyczące nieunikalnego atrybutu.

Ciekawą alternatywą dla domyślnej organizacji tabeli w postaci sterty jest w systemie Oracle tabela zorganizowana jako indeks (IOT, ang. index-organized table). Rekordy są

[2] Dostęp do rekordów tabeli w oparciu o informacje z indeksu w przeciwieństwie do pełnego przeglądu tabeli nie jest odczytem sekwencyjnym. Ze względu na sposób działania twardych dysków sekwencyjny odczyt danych jest znacznie szybszy od odczytu pojedynczych rekordów rozmieszczonych na dysku w sposób nieciągły i dlatego optymalizator stanowiący jeden z modułów systemu zarządzania bazą danych skorzysta z indeksu tylko gdy zapytanie wybiera niewielki procent rekordów tabeli.

wówczas przechowywane bezpośrednio w indeksie o strukturze B-drzewa utworzonym na kolumnach klucza głównego tabeli. Odczyt danych zawartych w tabeli zorganizowanej jako indeks przebiega podobnie jak w przypadku zwykłego dostępu za pomocą indeksu, tyle że wynikiem wyszukiwania są kompletne dane, a nie jedynie wskaźnik do nich. Ma to szczególne znaczenie dla zapytań wykonywanych przez zakresowy przegląd indeksu, które w tabelach zorganizowanych jako indeks wymagają zdecydowanie mniej odczytów niż w typowym modelu złożonym z indeksu i tabeli w postaci sterty. Tabela zorganizowana jako indeks posiada również inną ważną zaletę. Rekordy składowane są w strukturze indeksu zbudowanego na kolumnach klucza głównego, w kolejności zdefiniowanej przez wartości tego klucza. W związku z tym wykonanie zapytań, dla których wynikowe dane zwrócone mają zostać posortowane po kolumnach z dowolnej części wiodącej klucza głównego nie wymagają przeprowadzenia dodatkowej operacji sortowania.

Powyższa analiza możliwości sytemu Oracle wskazuje na konieczność przetestowania w kontekście problemu przetwarzania zbiorów zapytań eksploracyjnych trzech poniższych ścieżek dostępu do danych:

– pełen przegląd tabeli zorganizowanej jako sterta,
– dostęp do danych tabeli zorganizowanej jako sterta z użyciem indeksu,
– dostęp do tabeli zorganizowanej jako indeks.

4.2 Specyfika dostępu do danych dla badanych algorytmów

Rozważając wpływ ścieżek dostępu do danych na działanie poszczególnych algorytmów przetwarzania zbiorów zapytań eksploracyjnych, na pewnym poziomie ogólności można dwie ostanie z wyżej wymienionych ścieżek potraktować łącznie jako dostęp selektywny, stanowiący alternatywę dla pełnego odczytu danych.

Punktem odniesienia dla oceny poszczególnych zaproponowanych metod optymalizujących wykonanie zbioru zapytań eksploracyjnych zarówno w analizie teoretycznej jak i następnie w analizie eksperymentalnej będzie wykonanie sekwencyjne. W przypadku sekwencyjnego wykonania zapytań algorytmem *Apriori* ilość odczytanych danych z dysku jest oczywiście równa sumie danych odczytanych przez poszczególne zapytania.

Algorytm *Mine Merge* w pierwszej fazie wykonuje zadania pośrednie w sposób sekwencyjny przy użyciu *Apriori*, w drugiej fazie natomiast wymaga pojedynczego odczytu wszystkich partycji bazy danych. Zapytania pośrednie *Mine Merge* mogą wymagać więcej lub mniej iteracji *Apriori* niż zapytania oryginalne w zależności od konkretnego rozkładu wartości w bazie danych, przez co teoretyczne porównanie kosztów odczytów dyskowych *Mine Merge* z kosztami wykonania sekwencyjnego w ogólności nie jest możliwe. Na potrzeby niniejszych rozważań przyjmiemy więc, że wszystkie zapytania, zarówno oryginalne jak i pośrednie *Mine Merge*, wymagają tej samej liczby iteracji *Apriori*[3]. W takim wypadku dla ścieżki selektywnej *Mine Merge* w pierwszej fazie odczytuje mniej danych z dysku niż przy wykonaniu sekwencyjnym, gdyż każdy fragment bazy danych jest w tej metodzie odczytywany co najwyżej przez jedno zapytanie. Zysk jest tym większy im bardziej nakładają się oryginalne zapytania. Należy jednak zwrócić

[3] Przyjęte założenie jest uzasadnione przy założeniu zbliżonego rozkładu wartości w poszczególnych partycjach bazy danych.

uwagę, że *Mine Merge* w drugiej fazie odczytuje raz sumę danych źródłowych wszystkich zapytań, przez co kluczowe dla wydajności *Mine Merge* jest aby zysk z pierwszej fazy przewyższał koszt dodatkowego odczytu danych w drugiej fazie.

Jeśli do odczytu partycji danych odpowiadających poszczególnym zapytaniom konieczne są pełne przeglady tabeli, *Mine Merge* wymaga niestety większej liczby odczytów danych z dysku, gdyż liczba zapytań pośrednich jest większa od liczby zapytań oryginalnych. W tym sensie *Mine Merge* należy uznać za metodę zdecydowanie polegającą na dostępności selektywnych ścieżek dostępu do danych. Należy tu jednak podkreślić, że zwiększony koszt dotyczy tylko odczytów danych z dysku na poziomie serwera bazy danych, a koszt przesłania danych do eksploracji (szczególnie w architekturze klient-serwer) oraz koszt przetwarzania odczytanych danych przez algorytm *Apriori* będzie niższy dla *Mine Merge* niż dla wykonania sekwencyjnego. Dlatego też w konkretnej architekturze *Mine Merge* może okazać się szybszy od wykonania sekwencyjnego nawet gdy jedyną dostępną ścieżką dostępu do danych będzie pełen odczyt tabeli z danymi.

W przypadku algorytmów *Common Counting* i *Common Candidate Tree* należy uwzględnić fakt, iż w każdej iteracji odwołują się one do wielu partycji (zliczanie następuje wspólnie dla wszystkich zapytań eksploracyjnych). W każdej fazie weryfikacji kandydatów pobrany z bazy danych musi zostać pewien z góry znany zbiór partycji, a kolejność zwracanych transakcji tak naprawdę nie ma znaczenia – zamiast sprawdzać przynależność transakcji do konkretnego zadania na poziomie partycji, ten sam test może zostać wykonany dla każdej pojedynczej transakcji. W związku z powyższym, dla wspomnianych algorytmów można zintegrować pobieranie danych z poszczególnych logicznych partycji wyznaczonych przez nakładanie się zapytań eksploracyjnych poprzez wysłanie jednego zapytania SQL, które zwróci transakcje z wszystkich potrzebnych partycji. O ile usprawnienie to w nieznaczny sposób powinno wpłynąć na całkowity koszt odczytu danych przy odczytach selektywnych (szczególnie w przypadku warunków zakresowych), to będzie miało kluczowe znaczenie gdy wykorzystywane będą pełne odczyty tabeli. W tym drugim scenariuszu metody *Common Counting* i *Common Candidate Tree* będą wymagały dokładnie tyle odczytów danych z dysku co pojedyncze zapytanie eksploracyjne wykonujące największą liczbę iteracji *Apriori*. Metody te, w przeciwieństwie do *Mine Merge*, nie tylko więc nie powinny tracić na wydajności w stosunku do wykonania sekwencyjnego gdy nie są dostępne selektywne ścieżki dostępu do partycji danych, ale wręcz wydają się szczególnie predysponowane do zastosowania w przypadkach gdy konieczne są pełne odczyty tabeli z danymi.

5 Wyniki eksperymentów

Do eksperymentów wykorzystany został syntetyczny zbiór danych wytworzony za pomocą generatora GEN z projektu Quest [11]. Parametry wejściowe były następujące: 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. Dane umieszczone zostały w postaci znormalizowanej (tj. jako pary `<identyfikator transakcji, element>`) w bazie danych Oracle 11g Enterprise Edition, uruchomionej na komputerze z procesorem Athlon64 3800+ pracującym na systemie operacyjnym SuSE Linux (wersja jądra 2.6.27.7-9). Aplikacja testowa na-

pisana została w języku Java i uruchamiana była na komputerze z procesorem Intel Core2Duo 2.2GHz pracującym na systemie operacyjnym Mac OS X 10.6.1. Połączenie z bazą danych realizowane było za pomocą biblioteki JDBC przez sieć Ethernet.

Przeprowadzone zostały dwa eksperymenty: w pierwszym z nich zbiór zawierał dwa zadania o stałym rozmiarze, a zmienną był stopień ich nakładania; w drugim całkowity odczytywany zakres bazy danych był stały, zmieniała się natomiast liczba zadań w zbiorze. Każde zadanie w obu zbiorach odwoływało się do pojedynczego, ciągłego zakresu transakcji. W obu eksperymentach mierzono czasy wykonania sekwencyjnego (oznaczonego jako SEQ), algorytmu *Common Counting* (CC), algorytmu *Common Candidate Tree* (CCT) oraz algorytmu *Mine Merge* (MM) dla wszystkich trzech badanych ścieżek dostępu.

Wyniki pierwszego eksperymentu dla dwóch zapytań o rozmiarze 500 000 transakcji oraz progu minimalnego wsparcia 0.7% i stopnia nakładania od 0% do 100% z krokiem co 20% przedstawia Rys. 1. Zgodnie z oczekiwaniami, w przypadku pełnego przeglądu tabeli algorytm *Mine Merge* osiąga czasy znacznie słabsze od pozostałych badanych algorytmów, wygrywając jedynie z wykonaniem sekwencyjnym w przypadku wysokiego stopnia nakładania. Dla przeglądu tabeli z wykorzystaniem indeksu oraz przeglądu tabeli zorganizowanej jako indeks jego strata nie jest już tak duża, a dla zapytań o wysokim stopniu nakładania *Mine Merge* działa efektywniej od *Common Counting*.

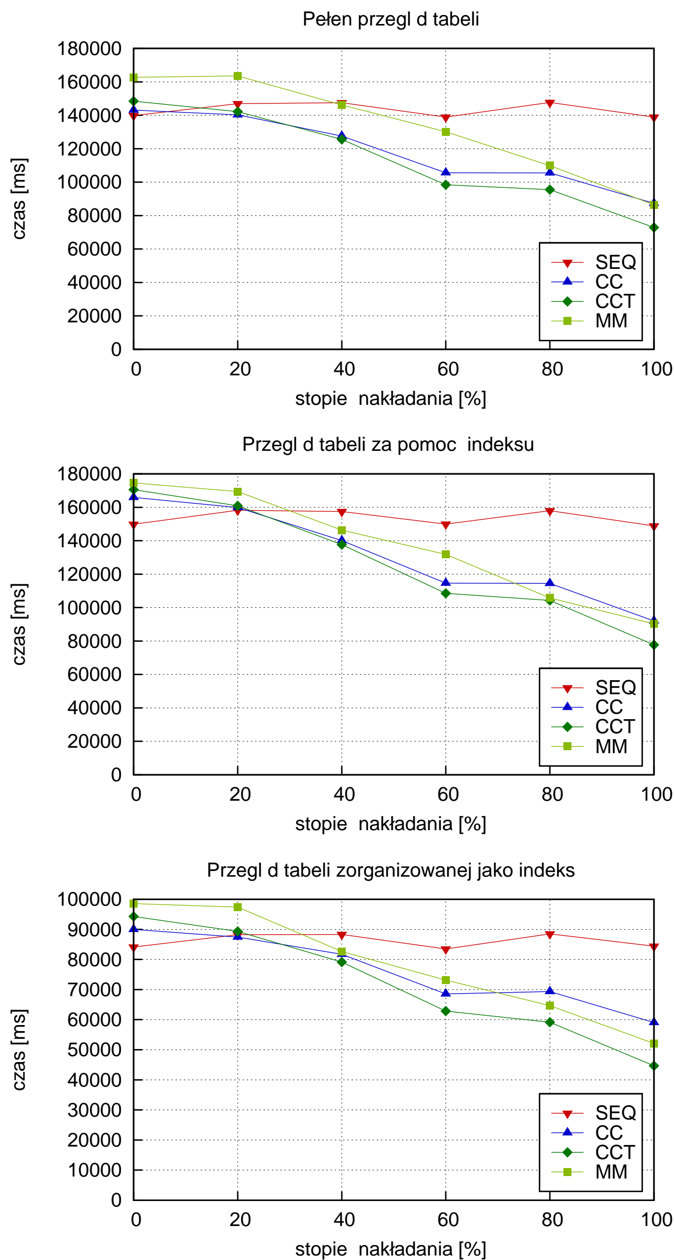
Niezależnie od ścieżki dostępu, algorytmy *Common Counting* oraz *Common Candidate Tree* osiągają zbliżone do siebie czasy dla niskiego stopnia nakładania. Przewaga CCT staje się wyraźna dopiero dla stopnia nakładania większego od 40%, w których to przypadkach uzyskuje on najlepsze wyniki spośród wszystkich badanych metod.

Czasy uzyskane dla dostępu do danych poprzez przegląd tabeli zorganizowanej jako indeks są dla wszystkich metod zdecydowanie krótsze niż czasy uzyskane dla pełnego przeglądu tabeli oraz przeglądu tabeli za pośrednictwem indeksu. Słaba efektywność tego ostatniego jest z kolei spowodowana niską selektywnością zapytań (pojedyncze zadanie eksploracyjne odwołuje się do 50% transakcji w tabeli).

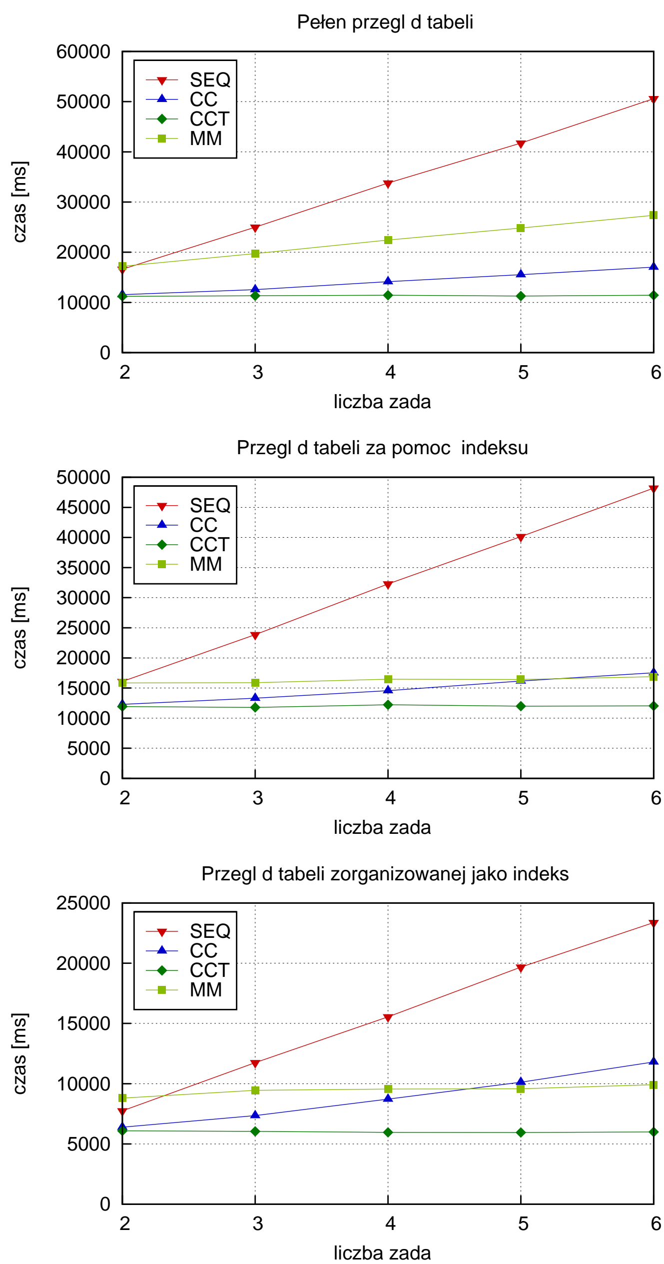
W drugim eksperymencie całkowity zakres danych, do których odwoływały się zadania, miał dla wszystkich punktów pomiarowych stały rozmiar. Zbiór zadań w każdej serii pomiarowej zawierał od 2 do 6 zadań o rozmiarze 200 000 transakcji, rozmieszczonych równomiernie na obszarze pierwszych 300 000 transakcji z bazy danych (za każdym razem pierwsze zadanie odwoływało się do transakcji o identyfikatorach z zakresu od 0 do 200 000, a ostatnie – od 100 000 do 300 000). Wyniki przedstawia Rys. 2.

Podobnie jak podczas pierwszego eksperymentu, efektywność *Mine Merge* dla pełnego przeglądu tabeli jest dość niska, a algorytm ten osiąga czasy lepsze jedynie od wykonania sekwencyjnego[4]. Gdy możliwy jest dostęp selektywny, liczba zadań w zbiorze ma niewielki wpływ na czas wykonania *Mine Merge*, co pozwala (przy odpowiednio dużej liczbie zadań) na osiągnięcie lepszych rezultatów od *Common Counting*. Algorytm *Common Candidate Tree*, niezależnie od ścieżki dostępu, pozostaje niewrażliwy na liczbę zadań w zbiorze i uzyskuje najlepsze czasy spośród wszystkich badanych metod.

[4] Dla obu metod czas zależy liniowo od liczby zadań, jednak w przypadku *Mine Merge* każde kolejne zadanie powoduje jedynie dodatkowe odczyty zawartości bazy danych po stronie serwera; wykonanie sekwencyjne z każdym kolejnym zadaniem przesyła do aplikacji i przetwarza dodatkowe 200 000 transakcji.



Rys. 1. Czasy wykonania dwóch zadań o rozmiarze 500 000 w zależności od ich stopnia nakładania przy minimalnym wsparciu 0.7%.



Rys. 2. Czasy wykonania badanych metod w zależności od liczby zadań przy minimalnym wsparciu 2%.

Dostęp do danych poprzez tabelę zorganizowaną jako indeks, tak jak w poprzednim eksperymencie, jest znacznie szybszy niż pozostałe metody odczytu. Analiza planów zapytania wykazała, iż w przeciwieństwie do pozostałych dwóch ścieżek, w przypadku tej ścieżki nie występuje operacja sortowania, ponieważ wymagana przez zapytanie kolejność rekordów jest zgodna z kolejnością ich występowania w indeksie. Co więcej, wykonywane zapytanie jest typowym przykładem zapytania zakresowego, szczególnie dobrze wspieranego właśnie przez tabele zorganizowane jako indeks. Pamiętać należy jednak, że tabela zorganizowana jako indeks posiada znaczne ograniczenie – aby mogła ona efektywnie wykonać dane zapytanie, wykorzystane atrybuty selekcji muszą znajdować się na początku jej klucza głównego. Nie może być ona zatem efektywnie wykorzystana np. w przypadku, kiedy w tabeli występuje wiele atrybutów selekcji, a zadania odwołują się do różnych ich kombinacji.

6 Podsumowanie

Artykuł poświęcony jest metodom przetwarzania zbiorów zapytań eksploracyjnych dla problemu odkrywania zbiorów częstych. Celem artykułu była teoretyczna i eksperymentalna analiza zachowania się zaproponowanych dotychczas w literaturze metod w kontekście różnych ścieżek dostępu do danych. Analiza teoretyczna wykazała, że o ile metoda *Mine Merge* traci w stosunku do wykonania sekwencyjnego w przypadku gdy do odczytu partycji danych konieczne są pełne odczyty tabeli, to metody *Common Counting* i *Common Candidate Tree* relatywnie zyskują na wydajności. Seria eksperymentów z wykorzystaniem systemu Oracle 11g potwierdziła wnioski z analizy teoretycznej, a ponadto wykazała, że spośród dwóch testowanych ścieżek oferujących selektywny dostęp do danych: dostępu do tabeli z wykorzystaniem indeksu i dostępu do tabeli zorganizowanej jako indeks zauważalnie lepsza jest ta druga.

Literatura

[1] R. Agrawal, T. Imielinski, and A. N. Swami, "Mining association rules between sets of items in large databases," in *SIGMOD Conference* (P. Buneman and S. Jajodia, eds.), pp. 207–216, ACM Press, 1993.

[2] T. Imielinski and H. Mannila, "A database perspective on knowledge discovery," *Commun. ACM*, vol. 39, no. 11, pp. 58–64, 1996.

[3] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *VLDB* (J. B. Bocca, M. Jarke, and C. Zaniolo, eds.), pp. 487–499, Morgan Kaufmann, 1994.

[4] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," in *SIGMOD Conference*, pp. 1–12, ACM, 2000.

[5] M. Wojciechowski and M. Zakrzewicz, "Methods for batch processing of data mining queries," in *Proceedings of the Fifth International Baltic Conference on Databases and Information Systems (DBIS 2002)*, pp. 225–236, 2002.

[6] P. Grudzinski and M. Wojciechowski, "Integration of candidate hash trees in concurrent processing of frequent itemset queries using apriori," in *Proceedings of the 3rd ADBIS Workshop on Data Mining and Knowledge Discovery (ADMKD'07)*, pp. 71–81, 2007.

[7] M. Wojciechowski, K. Galecki, and K. Gawronek, "Three strategies for concurrent processing of frequent itemset queries using fp-growth," in *Knowledge Discovery in Inductive Databases. 5th International Workshop, KDID 2006 Berlin, Germany, September 18, 2006 Revised Selected and Invited Papers*, pp. 240–258, Springer, 2007.

[8] T. K. Sellis, "Multiple-query optimization," *ACM Trans. Database Syst.*, vol. 13, no. 1, pp. 23–52, 1988.

[9] R. Jin, K. Sinha, and G. Agrawal, "Simultaneous optimization of complex mining tasks with a knowledgeable cache," in *KDD*, pp. 600–605, ACM, 2005.

[10] H. Blockeel, L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, and H. Vandecasteele, "Improving the efficiency of inductive logic programming through the use of query packs," *J. Artif. Intell. Res. (JAIR)*, vol. 16, pp. 135–166, 2002.

[11] R. Agrawal, M. Mehta, J. C. Shafer, R. Srikant, A. Arning, and T. Bollinger, "The quest data mining system," in *KDD*, pp. 244–249, 1996.