STUDIA INFORMATICA 2006
Volume 27 Number 2 (67)

Marcin GORAWSKI, Wojciech GĘBCZYK
Politechnika Śląska, Instytut Informatyki

SYSTEM Z CIĄGŁYMI ZAPYTANIAMI O *k*NN ZŁĄCZEŃ W PRZESTRZENNEJ HURTOWNI DANYCH

Streszczenie. Przedstawiono realizację systemu z ciągłymi zapytaniami klasy *k*NN w przestrzeni euklidesowej, w oparciu o przestrzenną, telemetryczną hurtownię danych. Zapytania są realizowane dla obiektów statycznych oraz obiektów mobilnych. System umożliwia wykonywanie wielu równoczesnych zapytań ciągłych.
Słowa kluczowe: przestrzenna hurtownia danych, zapytanie, złączenie *k*NN, obiekt mobilny, przestrzeń euklidesowa

REALIZATION OF CONTINUOUS *K*NN JOIN QUERIES IN SPATIAL DATA WAREHOUSE

Summary. The paper describes realization on continuous *k*NN join processing in spatial telemetric data warehouse. The queries are evaluated in euclidean space. Continuous *k*NN join queries are evaluated for mobile and static objects. Proposed approach introduces framework that enables evaluation of concurrent continuous *k*NN joins.
Keywords: spatial data warehouse, query evaluation, *k*NN join, mobile object, euclidean space

1. Wprowadzenie

Projektowany system przestrzennej hurtowni danych telemetrycznych i lokacji (ang. *Spatial Location and Telemetric Data Warehouse* (SDW(l/t)) działa w oparciu o system przestrzennej, telemetrycznej hurtowni danych (ang. *Spatial Telemetric Data Warehouse* (SDW(t))), która zbiera dane telemetryczne o pomiarach zużycia mediów (gazu, prądu, wody, ciepła) [2, 3]. System SDW(l/t) jest zasilany danymi telemetrycznymi z systemu zintegrowanego odczytu liczników (IMR) oraz danymi lokalizacji obiektów mobilnych.

Telemetryczny system zintegrowanego odczytu liczników bazuje na technologii IMR (ang. *Integrated Meter Reading*) oraz GSM/GPRS. System ten pozwala przesyłać dane, otrzymywane z liczników zużycia mediów, zlokalizowanych na dużym obszarze geograficznym do serwera telemetrycznego, z wykorzystaniem sieci operatorów telefonii komórkowej GSM w technologii GPRS oraz SMS.

System SDW(l/t) wspomaga podejmowania decyzji taktycznych o wielkości produkcji mediów, na podstawie krótkoterminowych prognoz ich zużycia. Prognozy te sporządzane są w oparciu o analizy danych, zgromadzonych w hurtowni danych zasilanej z serwera telemetrycznego w procesie ekstrakcji ETL.

Projektowany system ma na celu rozszerzenie funkcjonalności SDW(t) o sprawny proces serwisowania liczników systemu IMR. Aplikacja ma za zadanie usprawnienie pracy serwisanta liczników odczytu mediów, poprzez zadawanie ciągłych zapytań o *k* najbliższych sąsiadów, czyli o *k* najbliżej znajdujących się od serwisanta jednostek. Ponadto, dzięki ciągłej aktualizacji informacji na temat lokalizacji serwisantów, będzie można uniknąć dublowania sprawdzania konkretnej jednostki przez więcej niż jednego serwisanta, co pozwoli na zoptymalizowanie i lepszą organizację czasu pracy.

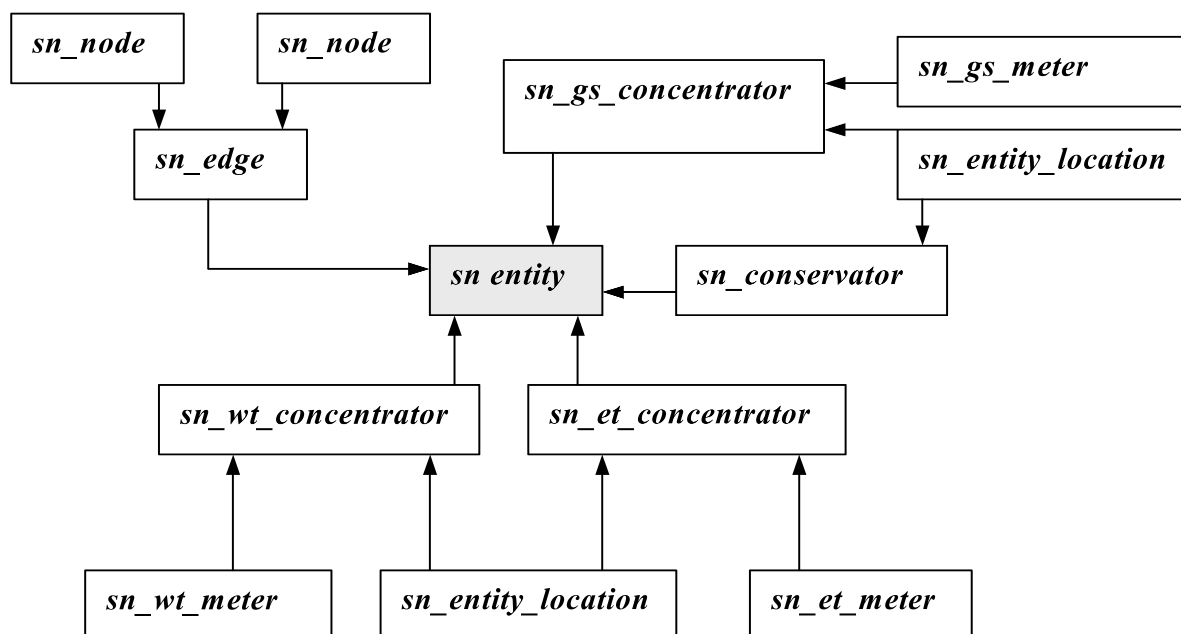
2. Zintegrowany odczyt liczników

Coraz szybszy dostęp do wszelkiego rodzaju informacji był impulsem do powstania dwuwarstwowej infrastruktury telemetrycznej – systemu IMR/SDW(t) [2, 3]. W skład infrastruktury wchodzi zintegrowany odczyt liczników IMR (energii elektrycznej, gazu, wody, ciepła), opierający się na technologii GSM/GPRS oraz system przestrzennych hurtowni danych telemetrycznych (SDW(t)).

System zintegrowanego odczytu liczników IMR (AIUT GSM) umożliwia przesyłanie danych z liczników do systemu SDW poprzez serwer telemetryczny. Zastosowanie technologii GSM/GPRS zapewnia niski koszt użytkowania, szybki czas wdrożenia systemu, dowolną odległość pomiędzy licznikami oraz niewrażliwość na ukształtowanie terenu. Natomiast wadą takiego rozwiązania jest wysoki koszt utrzymania systemu. Przeprowadzono jednak badania, które wykazały, że zakup urządzeń, wykorzystujących komunikację poprzez GSM/GPRS wraz z kosztem transmisji danych, jest znacznie bardziej opłacalnym rozwiązaniem niż zatrudnianie inkasentów, wyposażanie ich w przenośne komputery oraz samochody.

3. Model gwiazdy kaskadowej

Model gwiazdy kaskadowej jest rozwinięciem standardowego modelu gwiazdy i składa się z tablicy faktów oraz wymiarów, z których każdy jest odrębnym, zagnieżdżonym schematem gwiazdy, przechowującym dane wielowymiarowe. Każdy wymiar, oprócz atrybutów zawiera także informacje opisujące te atrybuty.



Rys. 1. Schemat gwiazdy kaskadowej
Fig. 1. Cascade star diagram

Dane, na jakich operuje projektowany system, można zamodelować za pomocą schematu gwiazdy kaskadowej. Schemat prezentowany na rysunku 1 jest rozszerzeniem modelu gwiazdy kaskadowej, zaprezentowanego w pracy [1]. Tabelą faktów całego systemu jest tabela jednostki koncentratora, nazwana *sn_entity*. Tabelami wymiarów, opisującymi koncentrator, są tabele poszczególnych rodzajów koncentratorów *sn_xx_concentrator* wraz z atrybutami oraz tabela krawędzi *sn_edge*, na której znajduje się dana jednostka. Każda krawędź jest opisana przez dwa, tworzące ją węzły. Tabelą wymiarów tabeli krawędzi jest tabela węzłów *sn_node*, zawierająca atrybuty węzła. Tabele poszczególnych koncentratorów są tabelami faktów dla tabel pojedynczych liczników i tabeli położenia koncentratora *sn_entity_location*. Dla przechowywania danych na temat obiektów mobilnych została stworzona tabela konserwatorów *sn_conservator*. Jest ona tabelą wymiarów dla głównej tabeli faktów. W ten sposób otrzymano gwiazdę kaskadową, zagnieżdżoną dwukrotnie. Relacje między tabelami są przejrzyste. Niewielkim kosztem, łącząc tylko niezbędne tabele, można dotrzeć do wszystkich informacji.

4. Złączenie *k*NN – *Gorder*

Operacja złączenia *k*NN (ang. *k Nearest Neighbour*) łączy każdy punkt jednego zbioru danych z jego *k* najbliższymi sąsiadami w innym zbiorze. *Gorder* jest metodą łączenia bloków przy użyciu pętli zagnieżdżonych, wykorzystującą sortowanie, planowanie łączeń, filtrację obliczania odległości oraz redukcję kosztów I/O oraz CPU [4]. Metoda ta najpierw sortuje zbiory danych wejściowych, według G-porządkowania (porządek bazujący na gridzie), tak aby zbiór danych mógł zostać podzielony na bloki, które są następnie poddawane zaplanowanemu złączeniu z użyciem pętli zagnieżdżonych do znalezienia *k* najbliższych sąsiadów dla każdego punktu danych. Jest to metoda prosta, a ponadto skuteczna i pozwalająca utrzymywać skutecznie wysokowymiarowe dane.

Metoda uzyskuje efektywność dzięki następującym własnościom: (1) dziedziczy moc łączenia bloków z użyciem pętli zagnieżdżonych, co pozwala na redukcję losowych odczytów, (2) odrzuca bloki danych, których obliczanie nie ma szans powodzenia, poprzez wykorzystywanie właściwości G-uporządkowanych danych, co pozwala zaoszczędzić na operacjach I/O oraz obniżyć koszt obliczania podobieństw, (3) wykorzystuje dwupoziomową strategię podziału do zoptymalizowania obciążenia I/O oraz czasu wykorzystania CPU, (4) redukuje koszt obliczania odległości, poprzez odrzucanie nadmiernych obliczeń bazując na odległości mniejszych wymiarów.

4.1. Definicja złączenia *k*NN

Definicja 1. (złączenie *k*NN) [4]

Mając dane dwa zbiory danych R i S, liczbę całkowitą *k* oraz metrykę podobieństwa *dist*(), złączenie *k*NN R i S, oznaczone jako R $\ltimes_{KNN}$ S zwróci pary punktów (p_i, q_j) takie, że p_i jest ze zbioru zewnętrznego R, a q_j ze zbioru wewnętrznego S oraz q_j jest jednym z *k* najbliższych sąsiadów p_i.

Zasadniczo, złączenie *k*NN łączy każdy punkt zewnętrznego zbioru danych R z *k* najbliższymi sąsiadami z wewnętrznego zbioru danych S. Odległość metryczna określana jest następująco:

$$dist(p,q) = \left(\sum_{i=1}^{d} \left| p.x_i - q.x_i \right|^{\rho} \right)^{\frac{1}{\rho}}, \ 1 \le \rho \le \infty .$$

W projektowanym podejściu wykorzystano metrykę kwadratową ($\rho = 2$), nazywaną „odległością euklidesową”. Zaproponowana technika może być przystosowana do innych metryk L_ρ, takich jak odległość *Manhattan* (L_1) oraz odległość maksymalna (L_∞).

Złączenie *k*NN posiada następujące własności:

1. Jest asymetryczne, tzn. ($R \ltimes_{KNN} S \neq S \ltimes_{KNN} R$). Spowodowane jest to asymetrycznością metody *k* najbliższych sąsiadów.
2. Moc zbioru odpowiedzi złączenia *k*NN jest przewidywalna, ponieważ złączenie zwraca *k* najbliższych sąsiadów dla każdego punktu z R.
3. Odległość od każdego punktu R do jego najbliższych sąsiadów jest nieznana apriori.

Własność druga sprawia, że złączenie *k*NN jest bardziej przydatne niż inne złączenia podobieństw oraz złączenia zakresowe w sytuacjach, gdy zakres ε nie może być w łatwy sposób określony. Złączenie zakresowe zwraca pary punktów z dwóch zbiorów danych, których odległość podobieństwa nie przekracza pewnej wartości. Jedną z trudności zastosowania złączenia zakresowego podobieństw w rzeczywistych aplikacjach jest fakt, że rozproszenie punktów danych jest często nieznane i wskazywanie odpowiedniego przedziału odległości podobieństw pomiędzy punktami jest sprawą trudną, o ile nie niemożliwą. A zatem wyniki zakresowego złączenia podobieństw są nieprzewidywalne, co wymaga od aplikacji działania w trybie „próba-i-błąd”.

4.2. G-porządkowanie

W relacyjnych bazach danych sortowanie jest wykorzystywane nie tylko do rozmieszczenia krotek według pewnego porządku, ale także do grupowania krotek z takimi samymi wartościami atrybutu złączenia, dla ułatwienia przetwarzania, bazującego na partycjach. Podobnie w *Gorder* projektowane jest porządkowanie bazujące na gridzie, nazywane G-porządkowaniem, do grupowania ze sobą sąsiednich punktów danych, tak aby podczas fazy planowania złączeń bloków z użyciem zagnieżdżonych pętli bylibyśmy w stanie zidentyfikować partycję bloku G-uporządkowanych danych i zaplanować dla niej złączenie.

G-porządkowanie składa się z dwóch kroków – transformacji PCA (ang. *Prinicipal Component Analisys* – analiza głównego komponentu) oraz sortowania według porządku grida.

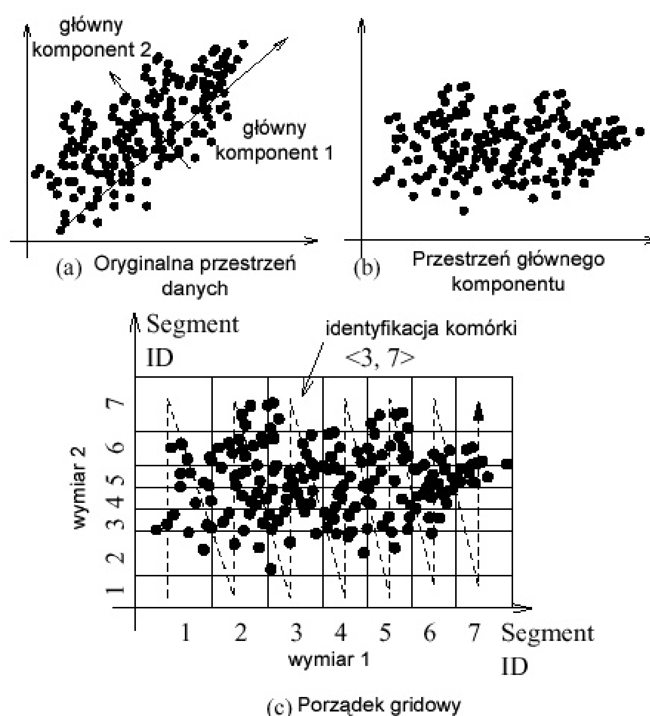
4.2.1. Transformacja PCA

Transformacja to sposób na wykrycie wzorców w danych oraz przedstawienie danych w taki sposób, aby podkreślić podobieństwa i różnice pomiędzy nimi. Ponieważ czasami ciężko znaleźć wzorce w wysoko wymiarowych danych, których nie można przedstawić graficznie, stosuje się w tym celu transformację PCA. Wykonuje ona analizę głównego komponentu na wejściowych zbiorach danych R i S oraz przekształca oryginalne dane w przestrzeń komponentu głównego. PCA przechwytuje rozbieżności w zbiorze danych i określa kierunki, wzdłuż których dane wykazują dużą rozbieżność. Po przetwarzaniu PCA większość informacji z oryginalnej przestrzeni danych jest skondensowana w kilka pierwszych wymiarów, wzdłuż których rozbieżności w rozproszeniu danych są największe.

4.2.2. Porządek gridowy

Drugi krok G-porządkowania sortuje zbiory danych R i S według porządku gridowego. Porządek gridowy nakłada grid na przestrzeń danych i dzieli ją na l^d komórek kwadratowych, gdzie l jest liczbą segmentów przypadających na wymiar grida. Długość komórki grida może być równa lub zmienna. W założeniach systemu ustalono, że komórki mają taką samą długość, ustalaną przez użytkownika aplikacji.

Na rys. 2 a przedstawiono oryginalną przestrzeń danych, a następnie przestrzeń komponentu głównego po przeprowadzeniu transformacji PCA (rys. 2 b). Ostatni z rysunków pokazuje nałożenie grida na zbiór danych dwuwymiarowych.



Rys. 2. Transformacja PCA oraz porządek gridowy
Fig. 2. PCA transformation and grid ordering

Jednym z podstawowych pojęć, związanych z nakładaniem na dane porządku gridowego jest określenie *wektora identyfikacji*. Jest to d-wymiarowy wektor $v=<s_1,...,s_d>$, gdzie s_i jest numerem segmentu, do którego należy komórka w i-tym wymiarze. W przypadku projektowanego systemu wektory identyfikacji będą dwuwymiarowe.

Na rys. 2 c przedstawiony został przykładowy wektor identyfikacji komórki. Pierwszym składnikiem wektora jest numer segmentu komórki z wymiaru 1, a drugim składnikiem numer segmentu z wymiaru drugiego.

Posiadając wiedzę na temat wektorów identyfikacji możemy przystąpić do zdefiniowania porządku gridowego:

Definicja 2. (Porządek gridowy $\prec_g$) [4]

Mając dany grid, który dzieli d-wymiarową przestrzeń danych na l^d kwadratowych komórek, punkty $p_m \prec_g p_n$, wtedy gdy $v_m \prec v_n$, gdzie v_m (v_n) jest komórką otaczającą punkt p_m.

$v_m \prec v_n$ wtedy i tylko wtedy, gdy istnieje wymiar k, to $v_m.s_k < v_n.s_k$ oraz $v_m.s_j = v_n.s_j$ dla $\forall j < k$.

Porządek gridowy ma na celu posortowane obiektów zgodnie z komórką otaczającą obiekt, tak aby po drugiej fazie G-porządkowania, obiekty znajdujące się wewnątrz tej samej komórki były razem zgrupowane.

Na rys. 3 przedstawiono przestrzeń danych dwuwymiarowych, na którą nałożono grid. Płaszczyzna XY została podzielona na segmenty (komórki grida) o takim samym rozmiarze. Mając tak podzielony układ współrzędnych można zauważyć, że każdy obiekt znajduje się, w którejś z komórek grida. Autorzy *Gorder* w celu identyfikacji położenia obiektu wykorzystują wektory identyfikacji. Ponieważ w przypadku omawianego projektu takie podejście wymagałoby przechowywania wielu dodatkowych struktur danych, postanowiono, że wektory identyfikacji zostaną zastąpione identyfikatorem położenia *id_grid*, który jest alfanumerycznym odpowiednikiem wektora. Każdy obiekt znajdujący się na mapie aplikacji ma swój własny *id_grid*, przechowywany w odpowiednich tabelach. Dla obiektów statycznych wyznaczany jest on tylko raz, przy ładowaniu mapy do programu. Dla obiektów mobilnych identyfikator jest uaktualniany za każdym razem, gdy obiekt przekroczy granicę komórki grida, w której się dotychczas znajdował.

Na rys. 3 widzimy zakreślony obiekt w komórce o wektorze identyfikacji *v*=<2, 1>, zakładając, że wymiarem o większym priorytecie będzie wymiar X. Poczynimy założenie, że liczba segmentów, przypadająca na wymiar, wynosi 5. Aby wyznaczyć unikalny identyfikator położenia dla komórki grida o takim wektorze identyfikacji, wykonujemy następujące działanie:

$$id_grid = sx + sy * seg + 1,$$

gdzie: $sx = 2$, $sy = 1$, natomiast $seg = 5$. Dla takich danych otrzymujemy identyfikator równy 8.

Mając przyporządkowany *id_grid* dla każdego obiektu można przystąpić do posortowania wejściowych zbiorów danych według porządku gridowego. W algorytmie *Gorder* wykorzystywane są dwa zbiory danych, które będą ulegać złączeniu. Należało w pewien sposób wyodrębnić te dwa zbiory oraz zastanowić się nad optymalnym definiowaniem zapytań pod względem oszczędności I/O oraz CPU. Zostało przyjęte, że mogą być zadawane cztery rodzaje zapytań. Każde zapytanie odpowiadałoby jednemu zestawowi danych – zapytania o koncentratory wody, gazu, elektryczności lub o wszystkie koncentratory równocześnie. Wtedy dla przykładowo trzech obiektów mobilnych, dla których byłyby zdefiniowane zapytania te-

go samego typu, możemy zdefiniować tylko jedno zapytanie, które w jednym zbiorze danych będzie posiadało informacje o obiektach mobilnych, natomiast w drugim zbiorze danych znalazłyby się informacje na temat wybranego zestawu danych.

Aby wykonać sortowanie musimy ustalić, który z wymiarów ma większą wagę. Założono, że większą wagę ma wymiar X. Stąd sortowanie punktów odbywa się najpierw po identyfikatorze grida, następnie po współrzędnej X, a na koniec po współrzędnej Y.

4.2.3. Prostokąty ograniczające

Do zdefiniowania prostokąta ograniczającego niezbędne jest przedstawienie definicji aktywnego wymiaru.

Definicja 3. (Aktywny wymiar G-uporządkowanych danych) [4]

Załóżmy, że v_1 (v_m) jest wektorem identyfikacji komórki otaczającej p_1 (p_m). Wymiar α jest aktywnym wymiarem G-uporządkowanych danych *B*, jeśli:

(1) $v_1.s_\alpha < v_m.s_\alpha$,

(2) $v_1.s_j = v_m.s_j \quad \forall j < \alpha$.

Dosłownie mówiąc, α jest pierwszym wymiarem takim, że $v_1.s_j < v_m.s_j (1 \le j \le d)$. Dla przykładu z rys. 3. należy wyznaczyć współrzędne wektorów identyfikacji pierwszego oraz ostatniego obiektu z bloku danych, do których należy zakreślony obiekt. Porównując otrzymane współrzędne wyznaczamy aktywny wymiar.

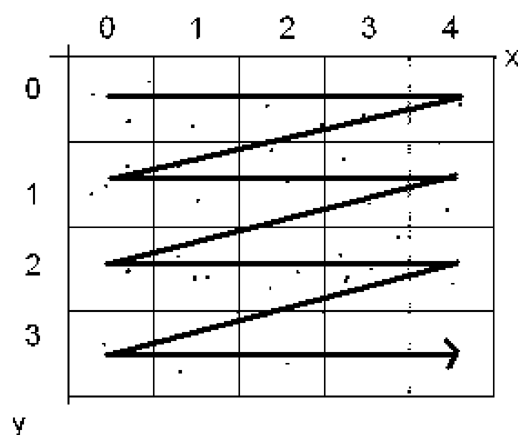
W omawianym przypadku dane są dwuwymiarowe. Ponadto, poczyniono wcześniej założenie, że wymiar X będzie miał większą wagę od wymiaru Y.

Prostokąt ograniczający dla *B* jest reprezentowany poprzez dolny lewy punkt E = <$e_1, ..., e_d$> oraz prawy górny punkt T = <$t_1, ..., t_d$>.

$$e_k = \begin{cases} (v_1.s_k - 1)\frac{1}{l} & dla \quad 1 \le k \le \alpha \\ 0 & dla \quad k > \alpha, \end{cases}$$

$$t_k = \begin{cases} v_m.s_k \frac{1}{l} & dla \quad 1 \le k \le \alpha \\ 0 & dla \quad k > \alpha, \end{cases}$$

gdzie α jest aktywnym wymiarem bloku danych. System działa w oparciu o dwa wymiary, X oraz Y, więc punkty przyjmą następującą postać: E = <*ex*, *ey*> oraz T = <*tx*, *ty*>. W zależności od tego, który wymiar będzie wymiarem aktywnym istnieją dwie możliwości wyznaczania współrzędnych tych punktów.



Rys. 3. Kierunek sortowania danych
Fig. 3. Direction of data ordering

Jeżeli wymiarem aktywnym jest wymiar X, współrzędne przyjmują następującą postać:

$ex = (sx1\text{-}1)*(1/seg)$; $ey = 0$;

$tx = sx2*(1/seg)$; $ty = 1$.

Jeżeli wymiarem aktywnym jest wymiar Y, współrzędne przedstawiają się następująco:

$ex = (sx1\text{-}1)*(1/seg)$; $ey = (sy1\text{-}1)*(1/seg)$;

$tx = sx2*(1/seg)$; $ty = sy2*(1/seg)$;

$sx1$, $sy1$ – współrzędne wektora identyfikacji pierwszego obiektu ze zbioru danych B,
$sx2$, $sy2$ – to współrzędne wektora identyfikacji ostatniego obiektu ze zbioru danych B,
seg – liczba segmentów (komórek grida), przypadająca na wymiar.

Dla przykładu z rys. 3 przyjęto, że blok danych składa się z pierwszych sześciu, kolejnych komórek grida (zgodnie z G-porządkowaniem). Współrzędne wektora identyfikacji pierwszego obiektu przyjmą wartości (2, 0), a ostatniego (2, 1). Dla $seg = 5$ otrzymano, dla aktywnego wymiaru X, współrzędne prostokąta ograniczającego [(0.2, 0);(1, 1)].

4.3. Złączenie G-uporządkowanych bloków danych

W drugiej fazie *Gorder*, G-uporządkowane dane ze zbiorów R i S są analizowane pod względem złączenia. Etap złączenia jest charakteryzowany przez dwie własności:

- *Gorder* stosuje dwupoziomową strategię podziału do zoptymalizowania czasu użycia I/O oraz czasu wykorzystania CPU.
- Gorder planuje dane dla złączenia, w celu optymalizacji przetwarzania *k*NN.

Partycjonowanie pierwszego poziomu jest zoptymalizowane pod względem czasu wykorzystania I/O. *Gorder* dzieli G-uporządkowane zbiory danych wejściowych na bloki, składające się z kilku stron fizycznych. Przypuśćmy, że alokujemy n_r oraz n_s strony buforowe dla danych z R i S, dzielimy R i S na bloki o rozmiarze zaalokowanych buforów. Bloki dla R są umieszczane w pamięci sekwencyjnie oraz iteracyjnie, blok po bloku. Bloki dla S są ładowa-

ne do pamięci w zaplanowanym porządku, bazując na ich podobieństwie do danych z R znajdujących się w buforze. To ładowanie wielokrotnych stron w danej chwili czasu jest efektywne pod względem czasu wykorzystania I/O, ponieważ znacząco redukuje obciążenie, spowodowane wyszukiwaniem. Dodatkowo, aby zoptymalizować przetwarzanie *k*NN, bloki dla S są planowane w taki sposób, aby jak najszybciej można było wydobyć *k* najbliższych sąsiadów, załadować ich do pamięci oraz złączyć z danymi z R, znajdującymi się już wcześniej w buforze.

Partycjonowanie poziomu drugiego segmentuje dane z R i S, znajdujące się w pamięci na bloki o znacznie mniejszych rozmiarach (podbloki). Optymalny rozmiar podbloków to 20-50 obiektów, nawiązując do wyników eksperymentów wykonanych przez autorów *Gorder*. Ponieważ w omawianym przypadku 20-50 obiektów to rozmiar normalnego bloku danych, zrezygnowano z dzielenia bloków na podbloki, ponieważ spowodowałoby to tylko większe zużycie pamięci oraz zwiększyłoby czas wykonywania obliczeń.

Podobieństwo dwóch bloków G-uporządkowanych danych mierzone jest poprzez odległość pomiędzy ich prostokątami ograniczającymi. Jak przedstawiono w poprzednim rozdziale, prostokąt ograniczający może być obliczony przez analizę pierwszego i ostatniego obiektu bloku G-uporządkowanych danych.

Minimalna odległość pomiędzy dwoma blokami G-uporządkowanych danych B_r oraz B_s oznaczana $MinDist(B_r, B_s)$, jest definiowana jako minimalna odległość pomiędzy ich prostokątami ograniczającymi:

$$MinDist(B_r, B_s) = \sum_{k=1}^{d} d_k^2 , \qquad d_k = \max(b_k - u_k, 0),$$

$$b_k = \max(B_r.e_k, B_s.e_k), \qquad u_k = \min(B_k.t_k, B_s.t_k).$$

Bloki z takim samym *MinDist* są sortowane według ich maksymalnej odległości. Maksymalna odległość pomiędzy dwoma blokami G-uporządkowanych danych B_r oraz B_s, oznaczana $MaxDist(B_r, B_s)$, jest definiowana jako maksymalna odległość pomiędzy ich prostokątami ograniczającymi:

$$MaxDist(B_r, B_s) = \sum_{k=1}^{d} (u_k - b_k)^2 ,$$

$$b_k = \min(B_r.e_k, B_s.e_k), \qquad u_k = \max(B_k.t_k, B_s.t_k).$$

Można zauważyć, że *MinDist* jest dolną granicą odległości dowolnych dwóch punktów z bloków R i S:

$$\forall p_r \in B_r, p_s \in B_s, \quad MinDist(B_r, B_s) \le dist(p_r, p_s).$$

Bazując na powyższym wniosku można stworzyć dwie strategie odrzucania:

1. Jeśli MinDist(Br,Bs) > odległości przycinania dla obiektu p, Bs, nie zawiera żadnych obiektów należących do k najbliższych sąsiadów obiektu p, dlatego też obliczenia odle-

głości pomiędzy p oraz obiektami z Bs mogą zostać odfiltrowane. Odległość przycinania dla obiektu p jest odległością pomiędzy obiektem p oraz kandydatem na jego k-tego najbliższego sąsiada. Początkowo wartość odległości przycinania jest ustawiona na ∞.

2. Jeśli MinDist(Br,Bs) > odległości przycinania dla Br, Bs, nie zawiera żadnych obiektów należących do k najbliższych sąsiadów jakiegokolwiek obiektu z Br. Skutkiem tego złączenie Br oraz Bs może być odrzucone. Odległość przycinania bloku dla R jest maksymalną odległością przycinania obiektów, znajdujących się wewnątrz R.

Algorytm 1. Złącz_Dane_Uporządkowane_Gridowo(R,S).

Wejście: R i S są dwoma zbiorami G-uporządkowanych danych, które zostały podzielone na bloki.

Opis:

1. Dla każdego bloku B_r є R wykonaj

    WczytajBlok(B_r);

    SortujBloki(S, B_r);

    Dla każdego B_s є *NieOdrzucone*(S, B_r) wykonaj

        WczytajBlok(B_s);

        Złączenie_w_Pamięci(B_r, B_s);

    WyjścieKNN(B_r);

Algorytm złączenia bloków G-uporządkowanych danych został przedstawiony jako algorytm 1. Ładuje on sekwencyjnie bloki dla R do pamięci (linie 1-2). Dla bloku B_r dla R znajdującego się w pamięci, bloki dla S są sortowane według ich odległości od B_r w porządku rosnącym (linia 3). W tym samym czasie bloki z *MinDist*(B_r B_s) większe od odległości przycinania B_r są odrzucane (strategia odrzucania (2)). A zatem, tylko pozostałe bloki są ładowane do pamięci jeden po drugim (linie 4-5). Dla każdej pary bloków z R i S, złączamy je w pamięci, poprzez wywołanie funkcji *Złączenie_w_Pamięci* (linia 6). Po tym jak już wszystkie nieodrzucone bloki dla S zostały przeanalizowane z B_r, zbiór kandydatów *k*NN dla obiektów z B_r jest zwracany jako wynik działania złączenia (linia 7).

4.3.1. Wczytywanie danych

Wczytywanie danych, zaimplementowane w systemie SDW(l/t), różni się od tego proponowanego przez autorów *Gorder*. W ich podejściu bloki danych były wyczytywane sekwencyjnie. Najpierw jeden blok, na którym wykonywany był cały algorytm *Gorder*, a następnie kolejny itd. Jest to zrozumiałe podejście w przypadku, gdy mamy do czynienia z ogromną liczbą danych. W zaprojektowanym podejściu nie będzie ich aż tak wiele, dlatego wszystko jest ładowane jednorazowo do pamięci. Jest to podejście wykorzystujące na pewno więcej pamięci, natomiast koszt operacji I/O, wykonywanych przez algorytm, jest o wiele mniejszy.

Wczytywanie danych jest wykonywane automatycznie, przy wywoływaniu metody rozpoczynającej działanie algorytmu. Do bazy danych wysyłane są dwa zapytania. Pierwsze z nich pobiera informacje na temat wszystkich konserwatorów, czyli obiektów mobilnych. Przekładając to na symbolikę stosowaną w powyższych opisach, jest to zbiór danych R. Drugie zapytania zwraca nam informacje na temat wszystkich obiektów wybranego przez użytkownika typu. Mogą to być także wszystkie możliwe typy koncentratorów oraz obiekty mobilne. Jest to zbiór danych S.

W trakcie definiowania zapytania lub ustawiania parametrów programu należy określić, jak pojemne będą bloki danych, tzn. ile obiektów ze zbiorów R oraz S trafi do pojedynczego bloku. Następnie przechodzimy do rozmieszczenia obiektów w odpowiednich blokach danych. Ponieważ dane są już posortowane gridowo, umieszczanie ich odbywa się poprzez prosty podział – pierwsze *n* obiektów do pierwszego bloku, kolejne *n* obiektów do drugiego bloku itd., aż do wyczerpania zasobów zbiorów R oraz S.

4.3.2. Sortowanie bloków danych

Algorytm sortowania bloków zwraca listę podbloków B_s, które mogą być złączone z danym podblokiem B_r. Podbloki są posortowane względem odległości od podbloku B_r, natomiast te podbloki, których $MinDist(B_r,B_s)$ jest większe od odległości przycinania B_r są odrzucane. Dla podbloków z takim samym *MinDist* obliczane jest *MaxDist*, a następnie wykonywane jest sortowanie według tej wartości.

Algorytm 2. SortujBloki(B_r).

Wejście: Podblok zbioru R, B_r.

Opis:

1. Ustaw $x = 0$ oraz *min_dist* = 0
2. Zainicjuj listę elementów *temp*
3. Dla każdego podbloku B_s
    - A. *min_dist* = odległość pomiędzy B_r *i* B_s
    - B. Dla każdego obiektu *t* z listy *temp*
        - I. Oblicz odległość obiektu *t* od podbloku B_s
        - II. Jeśli *min_dist* = obliczonej odległości
            1. Oblicz *max_dist* dla B_r i B_s oraz B_r i *t*
            2. Posortuj bloki względem *max_dist*
        - III. Jeśli *min_dist* > obliczonej odległości
            1. x = indeks obiektu $t + 1$
    - C. Jeśli *min_dist* < odległości przycinania B_r
        - I. Dodaj B_s do listy *temp* z indeksem *x*
    - D. x = 0, *min_dist* = 0

4.3.3. Złączenie bloków danych w pamięci

Oryginalny algorytm złączenia w pamięci został przedstawiony jako algorytm 3. Zarówno bloki dla R, jak i bloki dla S są dzielone na podbloki (linia 1). Dla każdego podbloku dla R B'_r podbloki dla S są porządkowane ze względu na ich odległość do B'_r. Ponownie zostaje wykorzystana strategia odrzucania 2 do odrzucenia tych podbloków dla S, których $MinDist(B'_r,B'_s)$ jest większa od odległości przycinania B'_r. Nie odrzucone podbloki dla S biorą udział w złączeniu z podblokami dla R, jeden po drugim (linie 4-5). Aby złączyć podbloki B'_r oraz B'_s, każdy obiekt p_r z B'_r jest porównywany z B'_s. Dla każdego obiektu p_r w B'_r analizujemy czy $MinDist(B'_r,B'_s)$ jest większe od odległości przycinania p_r. Jeśli tak, to korzystając ze strategii odrzucania 1, B'_s nie może zawierać żadnych obiektów, które są k najbliższymi sąsiadami p_r, a zatem B'_s może zostać pominięte (linie 6-7). W innym przypadku wywoływana jest funkcja *ObliczOdległość* dla obiektu p_r i każdego obiektu p_s z B'_s (linia 8). Funkcja *ObliczOdległość*, opisana w kolejnym podrozdziale, wstawia do zbioru kandydatów *k*NN obiektu p_r te obiekty p_s, których $dist(p_r, p_s)$ jest mniejsze od odległości przycinania p_r. d_α^2 jest odległością pomiędzy prostokątami ,ograniczającymi B'_r oraz B'_s w α-tym wymiarze, gdzie $\alpha = \min(B'_r.\alpha, B'_s.\alpha)$.

Ze względu na fakt, że w zaproponowanym podejściu nie wykonuje się operacji dzielenia bloków na podbloki, algorytm złączenia w pamięci uległ modyfikacjom. Zostało pominięte dzielenie bloków na podbloki i skupiono się bezpośrednio na obiektach z bloków B_r.

Algorytm 3. Złączenie_w_Pamięci(B_r, B_s).

Wejście: Podblok zbioru R, B_r oraz podblok zbioru S, B_s.

Opis:

1. Podziel B_r oraz B_s na podbloki;
2. Dla każdego podbloku $B'_r \in B_r$ wykonaj
   - *SortujBloki*(B_s, B'_r);
   - Dla każdego podbloku $B'_s \in NieOdrzucone(B_s, B'_r)$ wykonaj
     - Dla każdego obiektu $p_r \in B'_r$ wykonaj
       - Jeśli $MinDist(B'_r, B'_s) \leq$ OdleglośćPrzycinania(p_r), to
         - A. Dla każdego obiektu $p_s \in B'_s$ wykonaj
           - *ObliczOdległość*(p_s, p_r, d_α^2)

4.3.4. Obliczanie odległości

Prostokąty ograniczające G-uporządkowane dane posiadają specjalne właściwości, które można wykorzystać do zredukowania obliczeń odległości.

Krawędź prostokąta ograniczającego blok G-uporządkowanych danych B rozciąga się na cały zakres od 0 do 1 w wymiarze j ($j > B.\alpha$), gdzie $B.\alpha$ jest bieżącym wymiarem B. Własność

ta jest bezpośrednią obserwacją obliczania prostokątów ograniczających. Dlatego, gdy obliczamy podobieństwo dwóch bloków G-uporządkowanych danych, należy wziąć pod uwagę tylko pierwsze α wymiarów, gdzie $\alpha = \min(B_1.\alpha, B_2.\alpha)$ oraz $B_1.\alpha$ ($B_2.\alpha$) jest bieżącym wymiarem B_1 (B_2). W wyniku czego, obliczenia *MinDist* oraz *MaxDist* redukują się do następujących wzorów:

$$MinDist(B_1, B_2) = MinDist(B_1.\alpha, B_2.\alpha),$$

$$MaxDist(B_1, B_2) = MaxDist(B_1.\alpha, B_2.\alpha) + d - \alpha,$$

gdzie $B_1.\alpha$ ($B_2.\alpha$) jest projekcją B_1 (B_2) na pierwsze α wymiarów.

Ponadto, projekcja prostokąta ograniczającego blok G-uporządkowanych danych B, zawierającego m obiektów $p_1, ..., p_m$ na pierwsze $B.\alpha - 1$, wymiarów odpowiada komórce grida w pierwszych $B.\alpha - 1$ wymiarach. Własność ta wskazuje, że projekcje wszystkich obiektów w bloku G-uporządkowanych danych B na pierwsze $B.\alpha - 1$ wymiarów, znajdują się wewnątrz jednej komórki grida w pierwszych $B.\alpha - 1$ wymiarach. Skutkiem tego, dla jakichkolwiek obiektów p i q z bloków B_1 oraz B_2, $MinDist(B_1.\alpha - 1, B_2.\alpha - 1)$ może być wykorzystane do zaproksymowania odległości pomiędzy projekcją dla p oraz q na pierwszych $\alpha-1$ wymiarach, gdy grid posiada drobną ziarnistość. Aproksymowana odległość jest dolną granicą rzeczywistej odległości. To znaczy:

$$MinDist(B_1.\alpha - 1, B_2.\alpha - 2) \approx dist(p_{\alpha-1}, q_{\alpha-1}),$$

gdzie $p_{\alpha-1}(q_{\alpha-1})$ jest projekcją p (q) na pierwsze α-1 wymiarów.

Bazując na powyższych własnościach można zdefiniować trzecią strategię odrzucania, bazującą na aproksymowanej odległości. Formalnie przedstawia się ona następująco:
dla każdego obiektu p oraz q z G-uporządkowanych bloków B_r oraz B_s, jeśli $MinDist(B_1.\alpha - 1, B_2.\alpha - 1) + dist(p_{\{\alpha,k\}}, q_{\{\alpha,k\}})(\alpha \le k \le d)$ jest większe od odległości przycinania dla p, q nie może być kandydatem na k najbliższego sąsiada p, gdzie $\alpha = \min(B_r.\alpha, B_s.\alpha)$ *oraz* $p_{\{i,j\}}(q_{\{i,j\}})$ jest projekcją p (q) na wymiary od i do j.

Algorytm 4 jest zarysem algorytmu redukcji obliczeń odległości. Oblicza on najpierw $MinDist(B_1.\alpha - 1, B_2.\alpha - 1)$ z $MinDist(B_r, B_s)$ (linia 2). Następnie sumuje odległości pomiędzy p oraz q od wymiaru α, gdzie $\alpha = \min(B_r.\alpha, B_s.\alpha)$ (linie 3-4). Zawsze, kiedy *pdist* jest większe od odległości przycinania p, q nie może być jednym z k najbliższych sąsiadów p i może zostać odrzucony (linia 8). Jeśli q nie może zostać odrzucone przez aproksymowaną odległość, usuwamy czynnik aproksymacji (linia 4) oraz obliczamy jego rzeczywistą odległość (linie 5-6). Jeśli $dist(p, q)$ jest mniejsze od odległości przycinania p, q jest wstawiane do zbioru kandydatów *k*NN punktu p.

Algorytm 4. ObliczOdległość(B_r, B_s, p_s, p_r, *dist*).

Wejście:

B_r – blok zbioru R, B_s – blok zbioru S,

p_s – obiekt z bloku B_s, p_r – obiekt z bloku B_r,
dist (d_{α}^{2}) – jest odległością pomiędzy prostokątami ograniczającymi B_r oraz B_s na α-tym wymiarze.

Opis:

1. *dim* = minimalny aktywny wymiar spośród aktywnych wymiarów B_r oraz B_s
2. $pdist = MinDist(B_r, B_s) - dist$
3. Dla $k = dim$ do $k <= 2$

   Jeśli $k = 1$, to oblicz odległość w wymiarze X

   $pdist = pdist + (p_r.x - p_s.x)^2$

   Jeśli $k = 2$, to oblicz odległość w wymiarze Y

   $pdist = pdist + (p_r.y - p_s.y)^2$

   Jeśli *pdist* jest większe od odległości przycinania obiektu p_r, to obiekt p_s nie może być kandydatem na najbliższego sąsiada p_r Zwróć -1.
4. Jeśli p_s nie może zostać odrzucone przez aproksymowaną odległość, usuwamy czynnik aproksymacji $MinDist(B_r,B_s) - dist$.
5. Dla $k = 1$ do $k < 2$ (czyli praktycznie jeden raz)

   Jeśli $k = 1$, to oblicz odległość w wymiarze X

   $pdist = pdist + (p_r.x - p_s.x)^2$

   Jeśli $k = 2$, to oblicz odległość w wymiarze Y

   $pdist = pdist + (p_r.y - p_s.y)^2$

   Jeżeli *pdist* jest większe od odległości przycinania obiektu p_r, to p_s nie może być kandydatem na najbliższego sąsiada p_r. Zwróć -1.
6. Zwróć *pdist.*

4.4. Proces Gorder

Dla jak najszybszego wykonywania algorytmu *Gorder,* zaprojektowano osobny proces, wykonywany równolegle z innymi procesami aplikacji. Założenia były następujące:

- Proces był inicjowany w trakcie ładowania mapy z licznikami, jego stan pracy ustawiany jest na wartość *false*, więc pozostaje on w stanie spoczynku.
- Po zdefiniowaniu pierwszego zapytania zmieniany jest stan procesu, jego stan pracy ustawiany jest na *true.*
- Po usunięciu ostatniego zdefiniowanego zapytania, do procesu jest wysyłana informacja o zmianie jego stanu pracy ponownie na *false,* i jego akcja ulega zawieszeniu.

Jakie zadania ma proces? Przede wszystkim jest on odpowiedzialny za stałe przetwarzanie istniejących zapytań i jak najszybsze uaktualnianie odpowiedzi. Oprócz tego proces grupuje zapytania. Każde zdefiniowane zapytanie jest opisane w bazie danych, w tabeli *SN_GORDER*. Wpis określa, dla jakiego obiektu mobilnego zostało zdefiniowane zapytanie, ilu najbliższych sąsiadów należy wyszukać oraz jakiego typu mają to być sąsiedzi (woda/gaz/elektryczność/wszystko). Proces pobiera z bazy danych informacje na temat typów zapytań. Następnie dla każdego z typów uruchamiany jest algorytm *Gorder*, który znajduje najbliższych sąsiadów dla tych wszystkich obiektów mobilnych, dla których zdefiniowane jest zapytanie o aktualnie przetwarzany typ koncentratora. Następnie uaktualniane są odpowiedzi w tabeli wyników.

Dodatkowym elementem, wpływającym na szybkość procesu łączenia zapytań, jest mechanizm zaimplementowany w algorytmie zapytania *Gorder*. Pozwala on na sprawdzanie czy dany obiekt mobilny zmienił swoją lokalizację od ostatniej chwili czasu, w której było wyliczane dla niego zapytanie. Pozwala to na wyeliminowanie zbędnych obliczeń dla obiektów, których wynik zapytania i tak nie ulegnie zmianie.

5. Testy

Wszystkie testy zostały przeprowadzone na komputerze z procesorem *Athlon XP 2400+* oraz 512 MB pamięci RAM. Na stanowisku zainstalowany był system *Windows XP Professional*, środowisko *Java* firmy *Sun* w wersji 1.5 oraz *Oracle 10g*.

5.1. Testy dla jednego zapytania Gorder

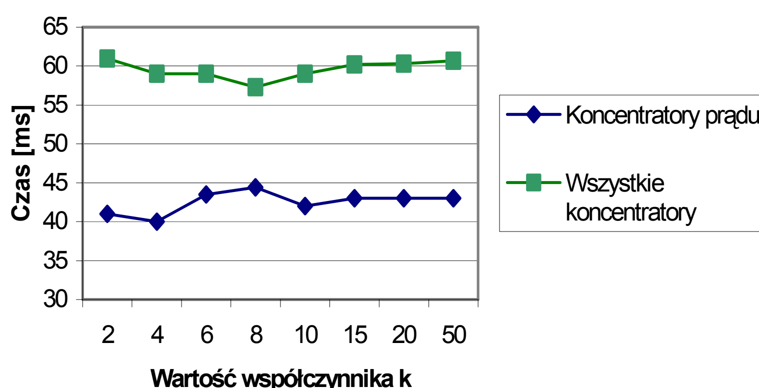
Wszystkie testy, oprócz ostatniego, wykonywane dla pojedynczego zapytania *Gorder* zostały przeprowadzone na mapie o rozmiarach 15x15 kilometrów. Mapa została wygenerowana dla 50 węzłów, przypadających na 100 km^2 oraz łącznie 150 koncentratorów, przypadających na 100 km^2. Skrypt, tworzący mapę, składał się z 2881 zapytań, a jego wykonanie spowodowało wzrost wykorzystywanej pamięci o 130 MB. Po mapie porusza się jeden obiekt mobilny, którego lokalizacja odświeżana jest co 1 sekundę.

5.1.1. Wpływ parametru k

Na rys. 4 przedstawiony został wykres wpływu wartości współczynnika *k* na czas wykonywania się zapytania *Gorder*. Zapytanie o koncentratory prądu (1) wykonywane było na zbiorze 50 koncentratorów/100 km^2, natomiast zapytanie o wszystkie koncentratory (2) opie-

rało się o zbiór 150 koncentratorów/100 km^2. Liczba segmentów, przypadających na wymiar wynosiła 10, a bloki danych zawierały po 50 elementów.

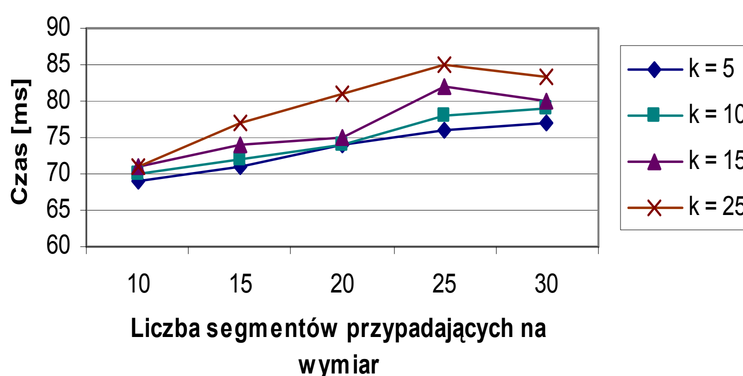
Na wykresie można zauważyć, że dla zapytania (1) czas obliczania utrzymuje się mniej więcej na stałym poziomie. Zauważalne są pewne odchylenia dla wartości 6 oraz 8 parametru *k*, ale są to odchylenia bardzo niewielkie, liczone w milisekundach. Dla zapytania (2) czas wykonywania obliczeń rośnie wraz ze wzrostem wartości parametru *k*, zaczynając od wartości 8, w którym to miejscu osiągnięte zostało minimum. Należy jednak zauważyć, że jest to wzrost o milisekundy. Można zatem stwierdzić, że algorytm *Gorder* dla jednego zapytania daje zadowalające rezultaty, jeżeli bierzemy pod uwagę zmiany wartości parametru *k*.



Rys. 4. Wpływ współczynnika *k* na czas wykonywania się zapytania
Fig. 4. *k* ratio's influence on query execution time

5.1.2. Wpływ liczby segmentów

Rysunek 5 przedstawia wykres wpływu liczby segmentów, przypadających na wymiar na czas, jaki jest potrzebny na obliczenie zapytania *Gorder*. W trakcie eksperymentów zostały wykonane cztery serie testów dla różnych wartości parametru *k*. Rozmiar bloków danych wynosił 50 elementów.

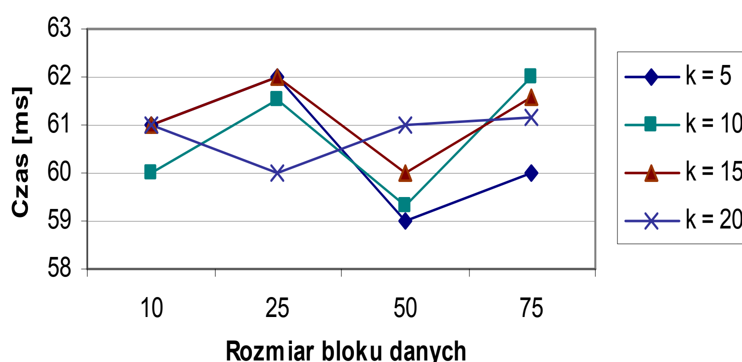


Rys. 5. Wpływ liczby segmentów przypadających na wymiar na czas wykonywania się zapytania
Fig. 5. Segments quantity's influence on query execution time

Dla wszystkich serii optymalną liczbą segmentów okazała się wartość 10. W miarę jak liczba ta rosła, rósł też czas obliczania zapytania. Dla *k* równego 5 czas rósł prawie liniowo. Dla *k* równego 10 można zauważyć już małe odchylenie przy 25 segmentach. Dla zapytania o 15 najbliższych sąsiadów wyniki są najbardziej zróżnicowane. Zauważalne jest zarówno gwałtowne wydłużenie się czasu (dla 25 segmentów), jak i nagły spadek (dla 30 segmentów). Przy *k* równym 25 czas narasta liniowo aż do 30 segmentów, gdzie gwałtownie spada. Warto też zauważyć, że wraz ze wzrostem liczby segmentów zwiększała się odległość pomiędzy krzywymi, co oznacza wzrost czasu obliczania dla zapytań o takim samym współczynniku *k*.

5.1.3. Wpływ rozmiaru bloków danych

Na rys. 6 został przedstawiony wpływ rozmiaru bloków danych na czas wykonywania się zapytania *Gorder*.

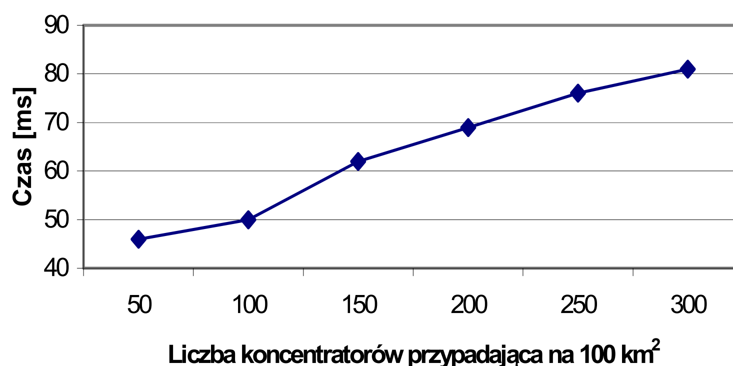


Rys. 6. Wpływ rozmiaru bloku danych na czas wykonywania się zapytania *Gorder*
Fig. 6. Data block size influence on *Gorder* query execution time

Testy zostały przeprowadzone dla 10 segmentów, przypadających na wymiar dla czterech wartości parametru *k*. Zgodnie z oczekiwaniami, otrzymane rezultaty wykazały, że rozmiar bloku danych nie ma większego wpływu na czas obliczania zapytania. Wyniki nieznacznie różniły się od siebie, lecz były to różnice rzędu kilku milisekund, które można pominąć.

5.1.4. Wpływ liczby koncentratorów

Test wpływu liczby koncentratorów na czas wykonywania się zapytania *Gorder* został przeprowadzony dla rozmiaru bloków danych równych 50, 10 segmentom przypadającym na wymiar oraz dla parametru *k* o wartości 20. Otrzymane wyniki wykazały, że wraz ze wzrostem liczby koncentratorów rośnie czas wykonywania się zapytania. Czas nie rośnie jednak bardzo gwałtownie. Po sześciokrotnym zwiększeniu liczby koncentratorów czas wzrósł o około 77%.

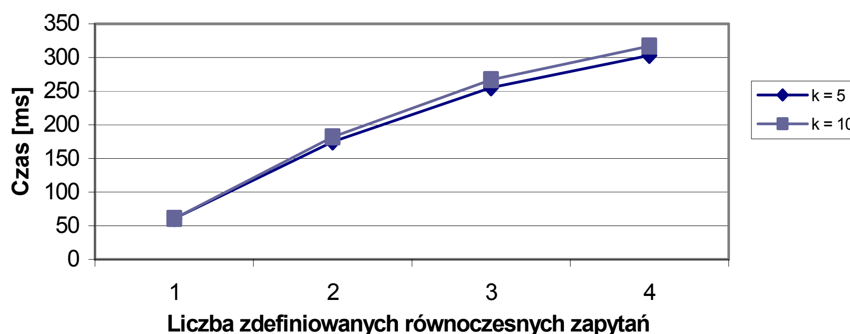


Rys. 7. Wpływ liczby koncentratorów na czas wykonywania się zapytania *Gorder*
Fig. 7. Concentrators quantity's influence on *Gorder* query execution time

5.2. Testy dla równoczesnych zapytań Gorder

Testy dla równoczesnych zapytań *Gorder* zostały przeprowadzone na mapie o wymiarach 15x15 kilometrów, o zagęszczeniu węzłów wynoszącym 50 węzłów/100 km^2 oraz zagęszczeniu koncentratorów, wynoszącym 150 koncentratorów/100 km^2. Wszystkie mobilne obiekty definiowane były z odświeżaniem jednosekundowym. Rozmiar bloków danych wynosił 50, a liczba segmentów, przypadających na wymiar miała wartość 10. W trakcie testów mierzono czas wykonywania się pełnego cyklu procesu *Gorder*, czyli czas obliczenia wszystkich zdefiniowanych zapytań.

5.2.1. Wpływ liczby zapytań



Rys. 8. Wpływ liczby równoczesnych zapytań na czas wykonywania się procesu *Gorder*
Fig. 8. Simultaneous queries fluence on *Gorder* process execution time

Na rys. 8 przedstawiono wpływ liczby zdefiniowanych równoczesnych zapytań na czas wykonywania się jednego cyklu procesu *Gorder*. Wszystkie zapytania były definiowane dla takiego samego typu koncentratorów, co powodowało, że obliczenie wszystkich zapytań odbywało się w trakcie jednego wywołania algorytmu *Gorder*.

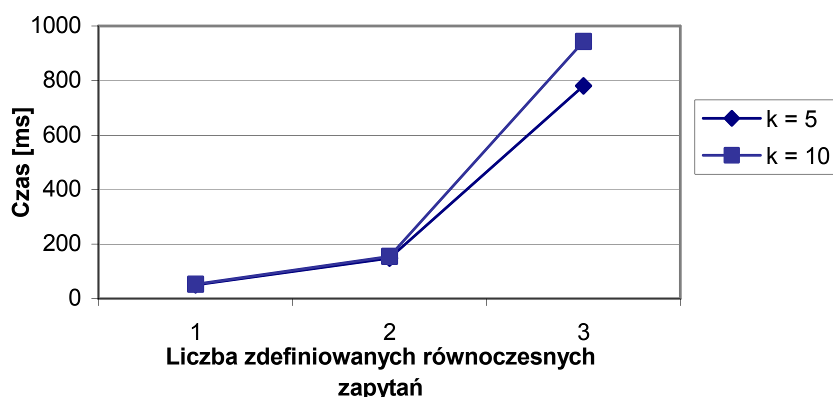
Zgodnie z wynikami, otrzymanymi w poprzednich testach, wartość parametru *k* nie miała znaczącego wpływu na otrzymywane rezultaty. Natomiast wraz ze wzrostem liczby równo-

czesnych zapytań, zaczął rosnąć czas wykonywanych obliczeń. Każde kolejne zapytanie powodowało znaczne wydłużenie się czasu oczekiwania na odpowiedź zapytania.

5.2.2. Wpływ liczby równoczesnych zapytań na czas wykonywania się procesu Gorder

Rysunek 9 przedstawia wyniki otrzymane dla testu sprawdzającego wpływ zróżnicowania równoczesnych zapytań na czas wykonywania się procesu *Gorder*. Każde z zapytań wykonywane było na zbiorze danych o takiej samej liczności.

Zróżnicowanie zapytań spowodowało, że w trakcie jednego cyklu procesu *Gorder* algorytm obliczania zapytania był wywoływany osobno dla każdego typu zapytania. A zatem, dla trzech zapytań, z których każde zdefiniowane było dla innego rodzaju koncentratorów (elektryczność/woda/gaz), proces trzy razy wywoływał algorytm *Gorder.* Otrzymane wyniki wykazały, że wraz ze wzrostem liczby zróżnicowanych zapytań rośnie czas wykonywania się procesu. O ile dla dwóch zapytań czas nie zwiększył się aż tak znacząco, to dla trzech różnych zapytań przyrost czasu jest znaczny. Pozwala to sądzić, że zaprojektowany mechanizm równoczesnego obliczania zapytań ciągłych nie spełnia naszych oczekiwań i należy pomyśleć nad innym rozwiązaniem lub rozwiązanie wymaga większych zasobów sprzętowych.

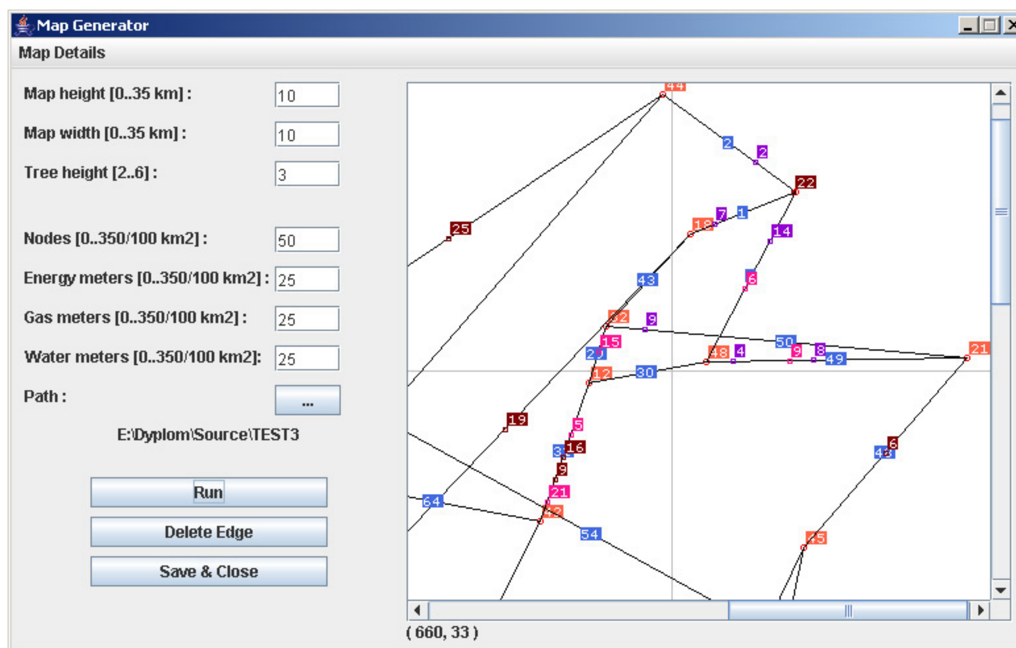


Rys. 9. Wpływ zróznicowania zapytań na czas wykonywania się procesu *Gorder*
Fig. 9. Queries disparity's influence on *Gorder* process execution time

6. System SDW(l/t)

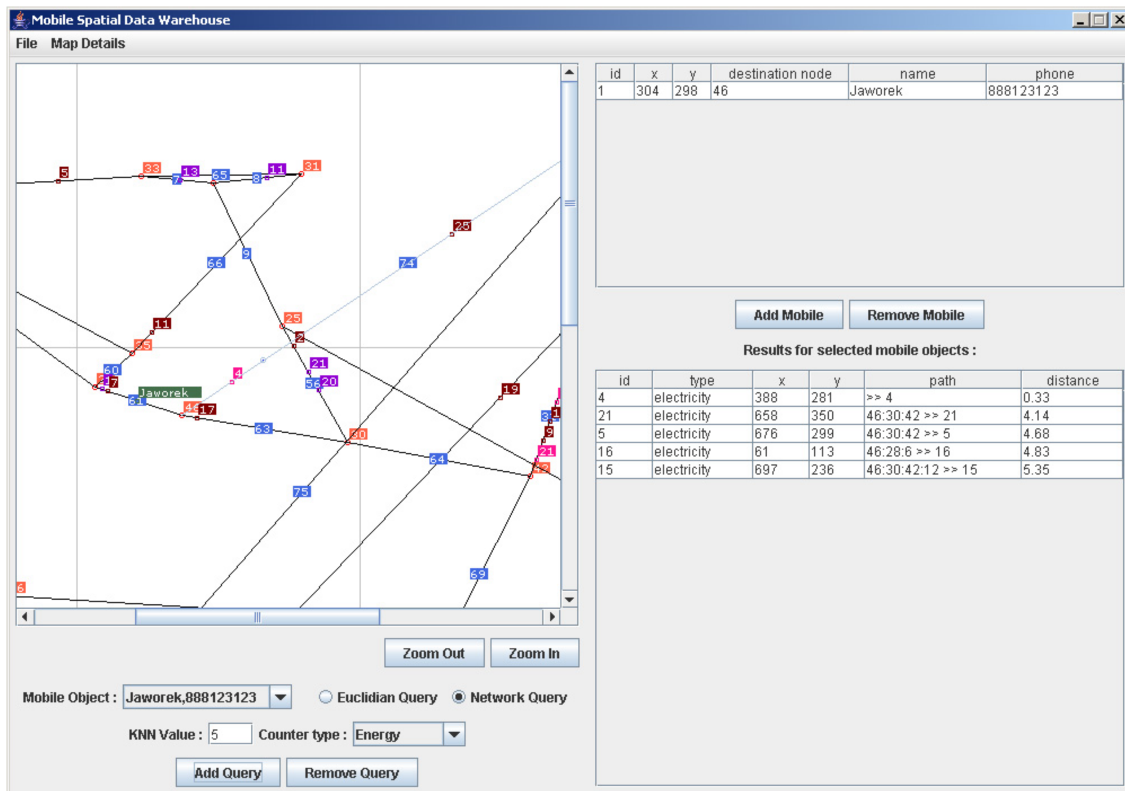
System SDW(l/t) składa się z dwóch zintegrowanych ze sobą aplikacji. Pierwsza z nich to generator danych, przedstawiony na rys. 10. Ponieważ SDW(l/t) ulega ciągłym aktualizacjom, co jakiś czas wprowadzane są modyfikacje, mające na celu udoskonalenie całej struktury oraz potrzebny był pewien zbiór danych, które nie byłyby poddawane tym operacjom. W ten oto sposób powstała idea generatora liczników telemetrycznych. Aplikacja umożliwia wygenerowanie wirtualnego schematu ulic wraz z rozmieszczonymi na nich licznikami. Użytkownik chcąc wygenerować nową mapę podaje takie parametry jak: gęstość węzłów

(skrzyżowań), tworzących krawędzie na mapie (ulice) oraz gęstość liczników, występujących na danej jednostce powierzchni mapy. Tak wygenerowane dane zapisywane są w postaci skryptu w języku SQL, który następnie wprowadza dane do bazy danych.



Rys. 10. Generator danych testowych
Fig. 10. Generator for test data

Jeżeli posiada się w bazie danych wygenerowane wcześniej informacje, można w pełni korzystać z głównego okna aplikacji, przedstawionego na rys. 11. Okno aplikacji podzielone jest na cztery części. W lewej górnej części okna widoczna jest mapa, przedstawiająca schematy ulic oraz rozmieszczenie liczników. Pod nią znajduje się część okna odpowiedzialna za definiowanie zapytań *k*NN. W prawej górnej części okna znajduje się tabela z informacjami na temat zdefiniowanych przez użytkownika obiektów mobilnych. Istnieje także możliwość definiowania nowych obiektów oraz edycja już istniejących. Ostatnia część aplikacji to tabela z wynikami zapytań.



Rys. 11. Główne okno aplikacji SDW(l/t)
Fig. 11. Main form of SDW (l/t)

7. Podsumowanie

Stworzony system SDW(l/t) bazuje na algorytmie *Gorder,* efektywnej metodzie przetwarzania złączenia *k*NN. *Gorder* opiera się na łączeniu bloków danych przy użyciu pętli zagnieżdżonych, wykorzystując sortowanie danych, planowanie łączeń, filtrację obliczania odległości oraz redukcję kosztów I/O i procesora.

Testy, dla pojedynczego zapytania *Gorder* w systemie SDW(l/t), wykazały skuteczność zaimplementowanego algorytmu. Zgodnie z oczekiwaniami parametr *k* oraz rozmiar bloku danych nie mają wpływu na czas wykonywania obliczeń. Różne wyniki dla testu, badającego wpływ liczby segmentów, przypadających na jeden wymiar, na czas wykonywania się algorytmu pozwoliły określić optymalną wartość tego parametru.

Rezultaty otrzymane dla testów, badających wpływ równoczesnego wykonywania się zapytań w SDW(l/t) okazały się niezadowalające. Można wymienić kilka czynników, które mogły mieć na to wpływ:

- Nieoptymalne rozwiązanie procesu przetwarzania równoczesnych zapytań ciągłych. Jest to pierwsze podejście do tej tematyki, zatem można je traktować jako pewnego rodzaju

eksperyment, którego rezultaty staną się podstawą do dalszych badań i porównań z kolejnymi podejściami.

- Znaczne obciążenie systemu spowodowane jest przez obsługę obiektów mobilnych. W rzeczywistych aplikacjach wystarczyłoby pobierać informacje o aktualnym położeniu obiektu. W naszym przypadku musimy zadbać o wprawienie obiektów w ruch, o ich poprawne poruszanie się oraz wizualizację. Wszystkie te elementy wpływają na obciążenie procesora oraz wzrost wykorzystywanej pamięci.

Zaprojektowany system SDW(l/t) jest doskonałym rozwiązaniem pilotażowym do dalszych badań nad tematyką zapytań ciągłych oraz obiektów mobilnych. Wykorzystanie stale powstających, nowych technik przetwarzania równoczesnych zapytań ciągłych może przynieść oczekiwane rezultaty. Należy jednak pamiętać o wyborze takiego mechanizmu, który nie spowoduje wzrostu obciążenia systemu SDW(l/t), a wręcz przeciwnie, pozwoli na przyspieszenie całości.

W celu poprawienia efektywności omawianego systemu można wykorzystać paradygmat rozdzielonego wykonywania operacji, jako sposób na osiągnięcie skalowalności podczas równoczesnego wykonywania ciągłych zapytań przestrzenno-czasowych. Główną ideą jest tu grupowanie podobnych zapytań w tabelę zapytań, a następnie obliczanie zbioru ciągłych, przestrzenno-czasowych zapytań jest rozpatrywane jako przestrzenne złączenie pomiędzy poruszającymi się obiektami i poruszającymi się zapytaniami. Podobne pomysły na rozdzielenie wykonywania zapytań zostały zastosowane w NiagaraCQ [6] dla zapytań web'owych, PSoup [7], oraz [8] dla zapytań potokowych.

LITERATURA

1. Gorawski M., Wróbel G.: Realizacja zapytań klasy *k*NN w przestrzennej telemetrycznej hurtowni danych. Studia Informatica, vol.26, nr 2(63), s. 1÷22.
2. Gorawski M., Malczok R.: Distributed Spatial Data Warehouse Indexed with Virtual Memory Aggregation Tree. 5th Workshop on Spatial-Temporal DataBase Management (STDBM_VLDB'04), Toronto, Canada 2004.
3. Gorawski M., Gabryś M.: Telemetryczny system zintegrowanego odczytu liczników. W: Współczesne problemy sieci komputerowych, WNT, Warszawa 2004, s. 203÷ 211.
4. Xia C., Lu H., Chin Ooi B., Hu J..: GORDER: An Efficient Method for KNN Join Processing.
5. Böhm C., Braunmüller B., Krebs F., Kriegel H. P.: Epsilon Grid Order: An Algorithm for the Similarity Join on Massive High-Dimensional Data. Proc. ACM SIGMOD INT. Conf. on Management of Data, Santa Barbara, CA, 2001.

6. Chen J., DeWitt D. J., Tian F., Wang Y.: NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In SIGMOD, 2000.
7. Chandrasekaran S., Franklin M. J.: Streaming Queries over Streaming Data. In VLDB, 2002.
8. Hammad M. A., Franklin M. J., Aref W. G., Elmagarmid A. K.: Scheduling for shared window joins over data streams. In VLDB, 2003.

Recenzent: Prof. dr hab. inż. Henryk Rybiński

Wpłynęło do redakcji 8 lutego 2006 r.

Abstract

Paper describes realization of continuous *k*NN (*k* Neared Neighbor) join processing in spatial telemetric data warehouse. *k*NN operation combines each point of one dataset with it's *k*NN's in the other dataset and provides more precise query results than the range similarity join. Such an operation is useful for data mining and similarity search. System bases on *Gorder* method, which is block nested loop join algorithm that exploits sorting, join scheduling and distance computation filtering and reduction to reduce both I/O and CPU costs. It sorts input datasets into the *G-order* and applies the scheduled block nested loop join on the G-ordered data. Continuous *k*NN-join queries are evaluated for mobile and static objects. Proposed approach introduces environment which enables evaluation of simultaneous continuous *k*NN joins. It introduces user-friendly environment. Paper also describes experiments on synthetic datasets and illustrates tests results.

Adresy

Marcin GORAWSKI: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-101 Gliwice, Polska, Marcin.Gorawski@polsl.pl
Wojciech GĘBCZYK: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-101 Gliwice, Polska, Wojciech.Gebczyk@polsl.pl