AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

**WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI,
INFORMATYKI I INŻYNIERII BIOMEDYCZNEJ**

KATEDRA AUTOMATYKI I INŻYNIERII BIOMEDYCZNEJ

Praca dyplomowa magisterska

Asocjacyjna normalizacja relacyjnych baz danych do postaci asocjacyjnego grafu neuronowego umożliwiającego efektywną eksplorację danych

Associative normalization of relational databases to form a neural associative graph that allows efficient data mining

Autor: Patryk Buzowicz

Kierunek studiów: Automatyka i Robotyka

Opiekun pracy: dr hab. Adrian Horzyk

Kraków, 2016

Uprzedzony o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolnośsci do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystycznego wykonania albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.”, a także uprzedzony o odpowiedzialności dyscyplinarnej na podstawie art. 211 ust. 1 ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym (t.j. Dz. U. z 2012 r. poz. 572, z późn. zm.): „Za naruszenie przepisów obowiązujących w uczelni oraz za czyny uchybiające godności studenta student ponosi odpowiedzialność dyscyplinarną przed komisją dyscyplinarną albo przed sądem koleżeńskim samorządu studenckiego, zwanym dalej «sądem koleżeńskim».”, oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i że nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

Spis treści

Cel pracy 9
- Aplikacja 9
- Badanie 9
- Edukacja 9

Relacyjne bazy danych 11
- Przechowywanie danych w bazie relacyjnej 11
- Relacje 11
- Przeszukiwanie 12
  - Sterta 12
  - Indeks drzewiasty 13
  - Indeks klastrowy 14
- Skalowalność 15
- Postaci normalne relacyjnej bazy danych 15
  - Pierwsza postać normalna (1NF) 15
  - Druga postać normalna (2NF) 16
  - Trzecia postać normalna (3NF) 17
- Inne mechanizmy składowania danych 17
  - Bazy dokumentowe 18
  - Bazy typu klucz-wartość (asocjacyjne) 19
  - Bazy grafowe 19
  - Silniki do wyszukiwania 20

Postać sieci skojarzeniowej AGDS 21
- Struktura sieci 21
  - Atrybuty 22
  - Wartości 23
  - Relacje 23
  - Połączenia 26
- Ograniczenia 26
- Algorytm generowania sieci 26
  - Połączenie z bazą danych 26
  - Wyznaczenie struktury bazy 29
  - Analiza tabeli 30

Kwerendy dla sieci skojarzeniowych 43
- Opis języka 43
  - Słowa kluczowe 43

Atrybuty ... 43
Typy danych ... 43
Wyrażenia warunkowe, filtrowanie ... 44
Operator równości ... 46
Operator nierówności ... 47
Funkcje ... 49
Przetwarzanie wyrażenia warunkowego ... 51
Wyrażenie warunkowe bez zaprzeczeń ... 52
Wyrażenie warunkowe z zaprzeczeniami ... 53
Odczyt danych ... 54
Funkcja podobieństwa ... 55
Dodawanie danych ... 56
Usuwanie danych ... 56
Aplikacja ... 57
Technologie ... 57
Windows Presentation Foundation ... 57
GraphSharp ... 57
Autofac ... 58
xUnit, Fluent Assertions, NSubstitute ... 59
ANTLR ... 60
Architektura ... 60
Intefejs użytkownika ... 61
Mechanizm sieci ... 61
Elementy aplikacji ... 62
Definicje baz danych ... 62
Generacja sieci ... 63
Wizualizacja sieci ... 63
Odpytywanie sieci ... 64
Analiza wydajności ... 67
Wydajność sieci ... 67
Wydajność wybranych zapytań ... 68
Pojedyncze porównanie ... 68
Porównania wykorzystujące operatory logiczne ... 69
Nierówność ... 70
Wartości ekstremalne ... 71
Zaprzeczenie ... 72

Podsumowanie 75
Bibliografia 77
Dodatek A – definicje wybranych typów używanych w aplikacji 81
Interfejs bazy danych 81
Table 81
Column 81
PrimaryKey 81
ForeignKey 81
Row 81
IKeysProvider 82
IDatabaseConnector 82

Cel pracy

Niniejszy rozdział opisuje motywy podjęcia pracy nad tematem związanym z relacyjnymi bazami danych i ich konwersją do asocjacyjnych struktur AGDS.

Aplikacja

Głównym celem tej pracy dyplomowej było utworzenie aplikacji komputerowej, która miała dwa główne zadania: odczyt struktury oraz danych z podanej bazy danych i utworzenie na podstawie tych informacji sieci skojarzeniowej, a także odpytywanie tak utworzonej sieci skojarzeniowej za pomocą specjalnie przygotowanego języka, zbliżonego do języka SQL - *Structured Query Language* [1].

Założeniem była możliwość transformacji bazy danych w trzeciej postaci normalnej. Przyjęto pewne ograniczenia transformowanych baz w celu usprawnienia pracy nad rozwojem aplikacji.

Oprócz tych głównych założeń, na potrzeby badań zostało stworzonych parę pobocznych, mniejszych narzędzi pomocniczych, służących do zadań takich jak importowanie danych z plików CSV czy wypełnianie tabel w podanych bazach losowymi danymi.

Badanie

Kolejnym zadaniem było zbadanie zbudowanego mechanizmu tworzenia sieci. Pod uwagę były brane trzy główne aspekty: proces odczytywania danych z bazy i tworzenia sieci, utrzymywanie wygenerowanej sieci w pamięci oraz odpytywanie sieci. Istotny był zarówno czas trwania poszczególnych procedur, jak i zużycie zasobów komputera do ich przeprowadzenia.

Edukacja

Głównym motywem do podjęcia tej pracy dyplomowej była chęć poszerzenia wiedzy związanej zarówno z relacyjnymi bazami danych, jak i z mechanizmami sieci skojarzeniowych. Została zgłębiona nie tylko wiedza teoretyczna, ale również praktyczne procedury i możliwości wynikające z opisywanej transformacji oraz pewne ograniczenia płynące z badanej struktury.

Niemniej istotna była możliwość wykorzystania mechanizmów baz danych do przeprowadzania zapytań i przygotowywania wyników.

Relacyjne bazy danych

Istnieje wiele mechanizmów przechowywania danych. Najpopularniejszymi z nich są nadal bazy relacyjne (RDBMS – *relational database management system*) [2], choć ostatnimi czasy popularność zyskują silniki bazodanowe oparte na innych zasadach przechowywania i serwowania danych (NoSQL – *not only SQL*). Na końcu niniejszego rozdziału zostały opisane mechanizmy składowania danych inne niż relacyjne bazy danych.

W dalszej części tej pracy rozważania dotyczące relacyjnych baz danych będą oparte o bazę SQL Server firmy Microsoft. Dla porównania natomiast znajdą się również referencje do bazy MySQL.

Przechowywanie danych w bazie relacyjnej

Relacyjne bazy danych organizują dane w tabelach. Każda tabela ma ściśle określony schemat, zdefiniowany podczas projektowania tej tabeli. Schemat ten określa, w jaki sposób tabela przechowuje dane. W ramach tej definicji danych mówi się o atrybutach (kolumnach). Każdy z atrybutów ma ściśle określony typ przechowywanych danych oraz opcjonalnie – w zależności od typu – wielkość danego pola (np. długość pola tekstowego).

System przechowuje dane na dysku w postaci pliku (lub wielu plików). Jednostką przechowywania danych jest *rekord*. W ramach rekordu przechowywany może być *rekord danych* (pojedynczy wiersz danych w tabeli), dane *indeksów*, metadane bazy itd. Rekordy grupowane są w *strony* (strony rekordów danych, rekordów indeksów, metadanych bazy itd.). Wszystkie strony mają tę samą strukturę – składają się z nagłówka, który przechowuje informacje o stronie (zajęte miejsce itp.), właściwej części danych oraz tablicy przesunięcia rekordu – *record offset array,* która zawiera wskaźniki na poprzednią i następną stronę. [3]

Relacje

Relacje są to skojarzenia danych występujących w różnych tabelach. Są one realizowane za pomocą kluczy głównych i obcych, będących kolumnami w tych tabelach. Wartości tych kluczy określają, z jakimi obiektami powiązana jest dana *encja*. [4]

Systemy relacyjnych baz mają wbudowane funkcje umożliwiające tworzenie relacji – klucze. Klucze dzielą się na:

- klucz główny – specjalna kolumna w tabeli, której wartości w poszczególnych encjach są unikatowe. Umożliwia to jednoznaczne rozróżnienie encji w tabeli. Klucz główny może być również określony, jako kilka kolumn w pojedynczej tabeli – wtedy wymuszona jest unikalność kombinacji wartości tych kolumn w poszczególnych wierszach

- klucz obcy – kolumna w tabeli (lub kilka kolumn), która odnosi się do wartości pojedynczej kolumny (lub odpowiedniej kombinacji kilku kolumn) klucza głównego z powiązanej tabeli. Taka sama wartość klucza obcego i głównego oznacza relację pomiędzy dwoma obiektami dwóch różnych tabel

Spójność danych pomiędzy tabelami wymaga, aby każdy obiekt był identyfikowalny – klucz główny każdego obiektu powinien być określony, oraz żeby każda relacja była jasno określona – wartość klucza obcego zawsze powinna odpowiadać istniejącej wartości klucza głównego w skojarzonej tabeli. Opcjonalnie, wartość klucza obcego może być pusta – odpowiada to sytuacji niezdefiniowanej w danej chwili relacji. [5]

Istnieją trzy rodzaje relacji w RDBMS: [6]

- jeden do wielu – pojedynczemu obiektowi (wierszowi) w tabeli A odpowiada wiele różnych obiektów z innej, powiązanej tabeli B. Jednocześnie obiekt tabeli B jest skojarzony dokładnie z jednym obiektem tabeli A. Tabela zawierająca klucz główny (tabela A) jest określana mianem *tabeli nadrzędnej*, zaś tabela z kluczem obcym (tabela B) – *tabeli podrzędnej*. Prócz wyżej opisanych wymagań w tej relacji nie jest narzucane żadne dodatkowe ograniczenie
- jeden do jednego – pojedynczemu obiektowi odpowiada dokładnie jeden obiekt będący w innej bazie. Relacja ta jest realizowana podobnie do w/w relacji „jeden do wielu", jednak dodatkowo kolumnie klucza obcego jest narzucane ograniczenie unikalności
- wiele do wielu – pojedynczemu obiektowi tabeli A odpowiada wiele wierszy tabeli B. Jednocześnie pojedynczemu obiektowi tabeli B odpowiada wiele obiektów tabeli A. Relacja typu „wiele do wielu" wymaga wprowadzenia dodatkowej *tabeli pośredniej*, zawierającej dwa klucze główne – jeden skojarzony z tabelą A, a drugi skojarzony z tabelą B

Przeszukiwanie

Niniejszy podrozdział jest ściśle powiązany ze sposobem składowania danych przez bazę danych. Metody przechowywania danych opisane poniżej wpływają bezpośrednio na wydajność i efektywność pobierania danych z bazy. [3]

Sterta

Sterta jest najprostszym sposobem przechowywania danych w bazie danych. Jest to struktura nieuporządkowana. Podczas dodawania danych, mechanizm bazy umieszcza porcję danych w dowolnym miejscu. Oznacza to, że dodanie danych jest operacją prostą i szybką.

W przypadku wyszukiwania danych, zorganizowanie danych w stertę oznacza jednak, że silnik bazy danych musi dokonać pełnego przeszukania tabeli – *full table scan*[1]. Niezależnie od rodzaju kryterium wyszukiwania danych jest to operacja niewydajna. Gdy od bazy danych oczekuje się większej wydajności wyszukiwania danych, stosuje się indeksy opisane niżej.

Indeks drzewiasty

Ogromnym usprawnieniem operacji wyszukiwania danych jest mechanizm indeksu. Pierwszym omawianym typem indeksu jest *zbalansowane drzewo przeszukiwania* (*balanced search tree* – B-tree). [7]

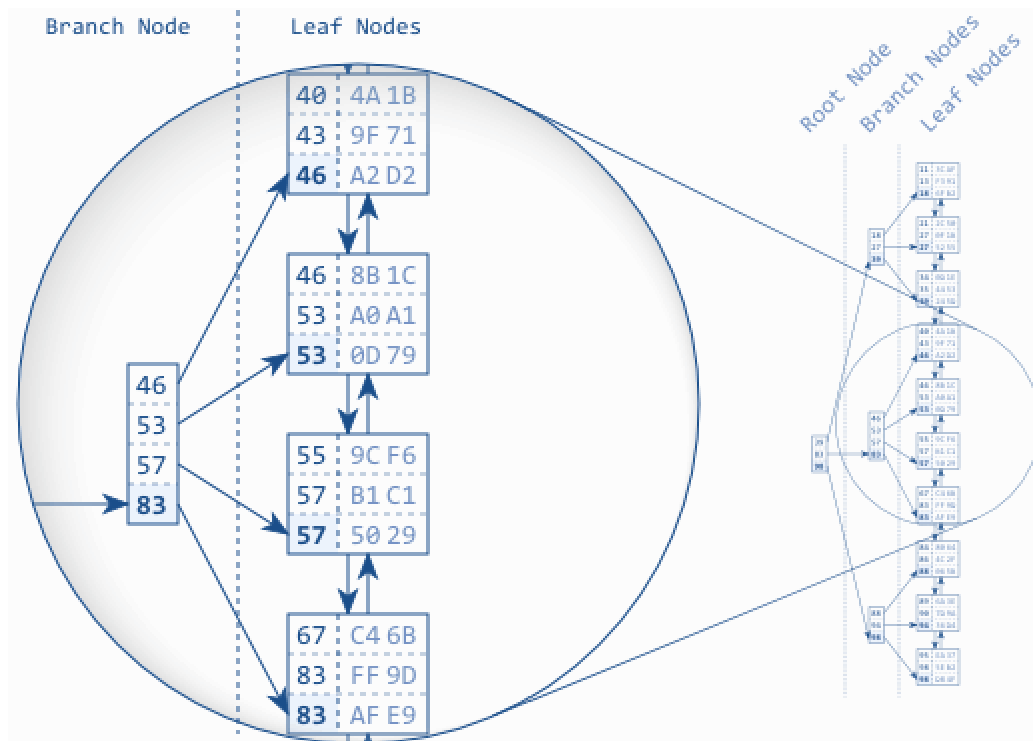
Jest to struktura drzewiasta, definiowana w obrębie pojedynczej tabeli danych. Składa się z elementów grupujących dane tabeli z wybranej kolumny (lub kilku kolumn) zdefiniowanej jako kolumna indeksu. Kilka wierszy z tabeli składa się na jeden element drzewa indeksu. Elementy w drzewie uporządkowane są według kolejności określonej przez wartości odpowiadające tym elementom. Elementy są łączone z elementami sąsiednich poziomów drzewa (zależność rodzic – dzieci) oraz pomiędzy sobą w ramach jednego poziomu (zależność brat – brat).

Wraz z przyrostem ilości danych w bazie, liczba poziomów w indeksie przyrasta logarytmicznie. Podstawa logarytmu związana jest z liczbą pojedynczych wartości, którą może zawierać jeden element drzewa. Oznacza to stosunkowo powolny przyrost liczby poziomów drzewa indeksu przy wzroście ilości danych przechowywanych w tabeli.

Dodatkowa struktura danych do obsługi jest związana z narzutem wydajnościowym przy dodawaniu, edycji i usuwaniu danych. W odróżnieniu od struktury sterty, prócz faktycznej lokalizacji z danymi tabeli aktualizacji wymaga struktura indeksu, by utrzymywać ją zsynchronizowaną z danymi.

Przeszukiwanie tabeli pod kątem atrybutu, dla którego zdefiniowany został ten indeks jest zdecydowanie szybsza [8]. Znalezienie konkretnego rekordu sprowadza się do przeszukania opisanego drzewa, czyli w praktyce przejścia od jego korzenia (najwyższego poziomu) do odpowiedniego liścia (zdefiniowanego przez kryterium wyszukiwania) przez wszystkie jego poziomy. Jak jednak wspomniano wyżej, liczba poziomów zależy logarytmicznie od liczby danych w tabeli, więc wraz z przyrostem danych czas znajdowania danych rośnie wolniej.

[1] https://en.wikipedia.org/wiki/Full_table_scan



Rysunek 1 Przykładowy schemat drzewa B-tree[2]

Indeks klastrowy

Indeks klastrowy to specjalny typ indeksu, który określa, w jakiej kolejności organizowane są dane *fizycznie* w pliku bazy danych. Oznacza to, że optymalizacja wydajności pracy z bazą danych może przebiegać już na etapie zapisu i przechowywania danych. [9]

Wymagania stawiane indeksowi klastrowemu to:

- stały przyrost wartości – mechanizm bazy danych jest najbardziej wydajny podczas zapisywania nowych danych, gdy zapisuje je sekwencyjnie na dysku. W tym celu na indeks klastrowy należy wybierać taki atrybut, o którym wiadomo, że stale będzie rósł w miarę dodawania nowych wierszy,
- unikalność – wartość stanowiąca indeks klastrowy powinna być jednoznaczna w ramach atrybutu, by system był w stanie jednoznacznie identyfikować wiersze. Mimo że nie jest to konieczne z punktu widzenia projektanta bazy, w przypadku duplikatów baza samodzielnie radzi sobie z taką sytuacją dodając unikalną wartość do podanej jako indeks, co w przypadku częstego występowania prowadzi do spadku wydajności,
- jak najmniejszy rozmiar – wartości składające się na indeks klastrowy zajmują miejsce również w indeksie. Im dłuższa wartość atrybutu będącego indeksem klastrowym, tym mniej wartości może zmieścić się w pojedynczym elemencie drzewa (patrz

[2] http://use-the-index-luke.com/img/fig01_02_tree_structure.en.svg

rozdział Indeks drzewiasty), a w konsekwencji zwiększyć się może liczba poziomów drzewa.

Skalowalność

Skalowalność relacyjnych baz danych jest ograniczona jedynie do skalowania wertykalnego. Oznacza to, że w sytuacji, gdy istnieje potrzeba rozszerzania bazy danych (zwiększenie ilości dostępnego miejsca, zwiększenie wydajności pracy bazy), możliwe jest jedynie ulepszanie zasobów pojedynczej maszyny obsługującej system bazy danych – wymianę czy dołożenie procesora, rozszerzenie pamięci lub dysku twardego. Niemożliwym jest skalowanie poziome, tzn. nie da się dokładać podobnych maszyn w celu uzyskania większej ilości pamięci czy zwiększonej wydajności bazy [10] [11].

Postaci normalne relacyjnej bazy danych

W związku z możliwością przechowywania danych w relacyjnych bazach na wiele sposobów, istnieje wiele zagrożeń dla ich poprawności i spójności. Przykładowo – składowanie logicznie różnych encji skojarzonych ze sobą w jednej tabeli niesie ze sobą ryzyko pominięcia niektórych krotek przy niepoprawnie skonstruowanej aktualizacji danych. Z drugiej jednak strony, dane magazynowane w taki sposób są łatwiejsze do uzyskania podczas odpytywania danych, zarówno z punktu widzenia interfejsu bazy danych (łatwiejsza postać zapytania, mniejsza liczba lub brak złączeń tabel) jak i wydajnościowego (odpytywanie mniejszej liczby tabel – mniej kosztownych operacji złączeń tabel).

W związku ze wspomnianymi zagrożeniami, wprowadzono pojęcie postaci normalnych bazy danych oraz normalizacji bazy danych. Normalizacja jest procedurą takiej organizacji schematu bazy danych, by zapewnić bezpieczeństwo, spójność i elastyczność danych, zarazem pozbywając się nadmiarowości danych w bazie.

Wyróżnia się trzy główne postaci normalne relacyjnych baz danych. [12] [13]

Pierwsza postać normalna (1NF)

Baza danych jest w pierwszej postaci normalnej, jeżeli wszystkie dane przechowywane w ramach pojedynczego atrybutu są *atomowe* (niepodzielne). Oznacza to, że każdy wiersz w pojedynczej kolumnie przechowuje dokładnie jedną informację o reprezentowanym obiekcie.

Przykładem tabeli niespełniającej zasady 1NF może być poniższa, przechowująca informacje o dostawcach i produktach przez nich oferowanych.

Tabela 1 Tabela niespełniająca 1NF

Nazwa dostawcy	**Adres dostawcy**	**Produkty**
Firma A	Kraków	Samochody, Motocykle

Firma B	Warszawa	Motocykle, Rury plastikowe, Uszczelki

Należy zauważyć, że powyższa tabela łamie zasadę 1NF, ponieważ w kolumnie Produkty przechowuje kolekcję nazw produktów oferowanych przez poszczególnych dostawców, przez co informacje te nie są atomowe.

Odpowiednikiem powyższej tabeli, która spełnia warunek 1NF jest poniższa tabela. Pojedyncze produkty zostały rozdzielone na wiele wierszy, kojarzących je z daną firmą.

Tabela 2 Tabela w postaci 1NF

Nazwa dostawcy	**Adres dostawcy**	**Produkt**
Firma A	Kraków	Samochody
Firma A	Kraków	Motocykle
Firma B	Warszawa	Motocykle
Firma B	Warszawa	Rury plastikowe
Firma B	Warszawa	Uszczelki

Druga postać normalna (2NF)

Aby baza była w drugiej postaci normalnej, spełnione muszą być dwa warunki: przede wszystkim musi być w postaci 1NF oraz wszystkie kolumny, które nie są kluczami, powinny być zależne funkcjonalnie od istniejących kluczy.

Poniższa tabela, która opisuje oddziały firm w poszczególnych miastach, jest w pierwszej postaci normalnej, ale nie spełnia warunków drugiej postaci normalnej, gdyż właściwość Kraj zależy tylko od jednej z kolumn (Miasto) klucza głównego (Nazwa firmy, Miasto).

Tabela 3 Tabela w postaci 1NF, ale nie 2NF

Nazwa firmy	**Miasto**	**Kraj**	**Produkt**
Firma A	Gdańsk	Polska	Łodzie
Firma B	Warszawa	Polska	Samochody
Firma C	Los Angeles	Stany Zjednoczone	Motorówki

Aby doprowadzić tę tabelę do zgodnej z drugą postacią normalną, należy pozbyć się z niej kolumny Kraj i wyrazić ją w innej tabeli.

Tabela 4 Tabela powstała z Tabela 3, spełniająca warunki 2NF

Nazwa firmy	**Miasto**	**Produkt**
Firma A	Gdańsk	Łodzie
Firma B	Warszawa	Samochody
Firma C	Los Angeles	Motorówki

Tabela 5 Nowa tabela powstała w procesie normalizacji

Miasto	**Kraj**
Gdańsk	Polska
Warszawa	Polska
Los Angeles	Stany Zjednoczone

Trzecia postać normalna (3NF)

Baza jest w trzeciej postaci normalnej, jeżeli spełnia warunki drugiej postaci normalnej oraz wszystkie kolumny, które nie należą do klucza głównego, są od niego bezpośrednio zależne.

Jeżeli w schemacie powstałym z Tabela 4 i

Tabela 5 zmienić podejście i zamiast oddziałów firm w miastach będzie interesujący wprost obiekt firmy, to będzie to schemat w drugiej postaci normalnej, ale nie w trzeciej. Po znormalizowaniu, z Tabela 4 otrzymano dwie tabele, które definiują relacje pomiędzy firmą a lokalizacją oraz pomiędzy firmą a produktami przez nią dostarczanymi.

Tabela 6 Jedna z tabel powstała w efekcie normalizacji do 3NF

Nazwa firmy	**Lokalizacja**
Firma A	Gdańsk
Firma B	Warszawa
Firma C	Los Angeles

Tabela 7 Druga tabela jako rezultat normalizacji do 3NF

Nazwa firmy	**Produkt**
Firma A	Łodzie
Firma B	Samochody
Firma C	Motorówki

Inne mechanizmy składowania danych

Systemy NoSQL w większości są zorientowane na skalowalność zarówno pionową, jak i poziomą. Oznacza to, że w miarę dodawania kolejnych urządzeń do przechowywania i

obsługi danych, system bazy danych jest w stanie efektywnie zarządzać stosunkowo większą ilością danych. Różnią się jednak zasadniczo od systemów relacyjnych, przede wszystkim pojęciem relacji, ale i samym przechowywaniem danych. Inną istotną różnicą między NoSQL a bazami relacyjnymi jest brak konieczności definiowania struktury (schematu) danych, co oznacza, że dane przechowywane w bazie NoSQL są dynamiczne i mogą mieć różną strukturę w obrębie pojedynczego odpowiednika znanej z RDBMS tabeli.

Poniżej krótko opisano wybrane ze względu na popularność mechanizmy składowania inne niż RDBMS.

Bazy dokumentowe

Najpopularniejszą bazą dokumentową jest MongoDB (dane na lipiec 2016). Ta baza dokumentowa przechowuje dane w postaci obiektów – dokumentów w formacie BSON – *binary JSON*[3]. W praktyce pojedynczy dokument to obiekt JSON stanowiący zestaw kluczy (nazwa atrybutu) i wartości. Kluczem może być dowolna wartość liczbowa lub łańcuch znaków (string). Wartością zaś może być dowolny obiekt: zarówno typu prymitywnego (liczbowy, łańcuch znaków, wartość logiczna) jak i inny obiekt JSON lub tablica (zbiór obiektów).

Relacje w ramach tej bazy są realizowane na zasadzie *hierarchii* – każdy obiekt (rodzic) w bazie może jako atrybut posiadać inny obiekt lub listę obiektów (dzieci), a te z kolei mogą przechowywać jeszcze inne obiekty. W ten sposób dokumentowa baza logicznie wspiera dwa z trzech rodzajów relacji znanych z relacyjnych baz danych: relację *jeden do jednego*, gdy rodzic posiada jako atrybut jeden obiekt – dziecko, oraz relację *jeden do wielu*, gdy rodzic posiada tablicę obiektów - dzieci. Należy jednak pamiętać, że wspomniane relacje nie odpowiadają dokładnie relacjom z RDBMS – na przykład brakuje możliwości bezpośredniego dostępu do obiektów podrzędnych – dzieci.

Odpowiednikiem tabeli z relacyjnej bazy danych jest *kolekcja obiektów*. W ramach kolekcji zdefiniowana jest jej nazwa oraz możliwe jest również utworzenie indeksów z istniejących atrybutów obiektów. Jak wspomniano w poprzednim rozdziale, kolekcja nie definiuje struktury danych, które przechowuje. [14]

```
{
  "birthdate": "05-02-1977",
  "first_name": "John",
  "last_name": "Connor",
  "children": [
    {
      "birthdate": "01-04-2001",
      "first_name": "Anna",
      "last_name": "Connor"
    },
    {
      "birthdate": "09-08-2003",
```

[3] JSON – JavaScript Object Notation - http://www.json.org/json-pl.html

```
      "first_name": "Deryl",
      "last_name": "Connor"
    }
  ]
}
```

Listing 1 Przykład obiektu JSON będącego dokumentem w bazie

Bazy typu klucz-wartość (asocjacyjne)

Innym ciekawym typem bazy danych jest baza typu klucz-wartość. Popularnym przykładem implementacji tej bazy jest Redis[4]. Redis jest systemem, który przechowuje dane w pamięci podręcznej.

Kluczem w strukturze może być dowolny zestaw danych binarnych: od łańcucha znaków po zdjęcie czy utwór muzyczny. W praktyce, ze względów wydajnościowych, najlepiej używać kluczy mniejszych niż 1 kilobajt danych. Wartością przechowywaną w bazie Redis może być dowolny złożony typ danych: łańcuchy znaków, listy, zbiory, bitmapy etc.

Ze względu na architekturę tej bazy (przechowywanie danych w pamięci RAM) system ten znajduje zastosowanie przede wszystkim jako implementacja mechanizmów *cache* czy szybkiej komunikacji pomiędzy innymi systemami (np. wykorzystując zaimplementowany tam wzorzec *publish/subscribe*). [15]

Bazy grafowe

Rosnącą popularność zyskuje również grafowe podejście do przechowywania danych. Najpopularniejszym przykładem jest system Neo4j[5], wykorzystywany przez takie firmy jak UBS, Cisco czy eBay.

Neo4j jest systemem napisanym w technologii Java, jednak zapewnia interfejs HTTP umożliwiający wykorzystanie go w dowolnej innej technologii klienckiej. Do przechowywania i zarządzania danymi, Neo4j bazuje na podstawowych konceptach związanych z teorią grafów: wierzchołki i krawędzie. W takim podejściu za pomocą wierzchołków reprezentowane są obiekty (np. pojedynczy użytkownik, produkt), a za pomocą krawędzi – relacje i zależności pomiędzy obiektami (np. użytkownik *kupił* produkt). Zarówno wierzchołki, jak i krawędzie takiego grafu mogą posiadać właściwości opisujące dany obiekt (np. nazwa użytkownika, data zakupu). Ponadto, za pomocą *etykiet* (labels) grupowane są wierzołki tego samego typu – można je potraktować jako odpowiednik tabel (np. użytkownicy, produkty). [16]

Opisywany system przechowywania danych definiuje własny, dedykowany język do zarządzania danymi – Cypher. Podobnie jak SQL, jest to język deklaratywny, przy czym jest

---

[4] http://redis.io/
[5] https://neo4j.com/

zorientowany na zarządzanie relacjami w grafie danych. Poniższe zapytanie zwraca tytuły filmów (`movie.title`) zaczynających się na literę „T" wraz z całą ich obsadą (`actor.name`)

```
MATCH (actor:Person)-[:ACTED_IN]->(movie:Movie)
WHERE movie.title STARTS WITH "T"
RETURN movie.title AS title, collect(actor.name) AS cast
ORDER BY title ASC LIMIT 10;
```

Cypher umożliwia również dodawanie nowych danych (CREATE), modyfikację (`MATCH...SET`) i usuwanie (`MATCH...DELETE`). [17]

Łatwość odczytywania relacji ze struktury grafowej umożliwia sprawne przeprowadzanie operacji rozpoznawania wzorców i wyciąganie informacji na podstawie zależności pomiędzy obiektami, które nie są natychmiast widoczne w typowej relacyjnej bazie danych (lub trudno osiągane przez konieczność wielokrotnego, kosztownego łączenia tabel). Przykładowym zastosowaniem jest wykrywanie zachowań niepożądanych, np. prób oszustwa. [18]

Na potrzeby zarządzania danymi z podejścia grafowego korzysta również aplikacja Facebook[6], jednak nie wykorzystuje ona wspomnianej bazy Neo4j. Zamiast tego, na potrzeby zarządzania ogromną ilością danych i przeszukiwania ich, utworzony został inteligentny mechanizm interpretowania zapytań użytkowników i przeszukiwania grafowej struktury połączeń systemu, by zwrócić wyniki. Przykładowe zapytanie możliwe do przekazania do silnika bazy: „ZNAJOMI MOICH ZNAJOMYCH KTÓRZY SĄ MĘŻCZYZNAMI I INTERESUJĄ SIĘ KOBIETAMI" [19]

Silniki do wyszukiwania

Ostatnim mechanizmem wartym wspomnienia jest silnik wyszukiwania. Taki mechanizm również przechowuje dane, jednak są one organizowane w taki sposób, by zawarty w nich tekst można było łatwo przeszukiwać. Przykładem może być serwis Allegro[7], którego wyszukiwarka wspierana jest takimi silnikami do wyszukiwania[8].

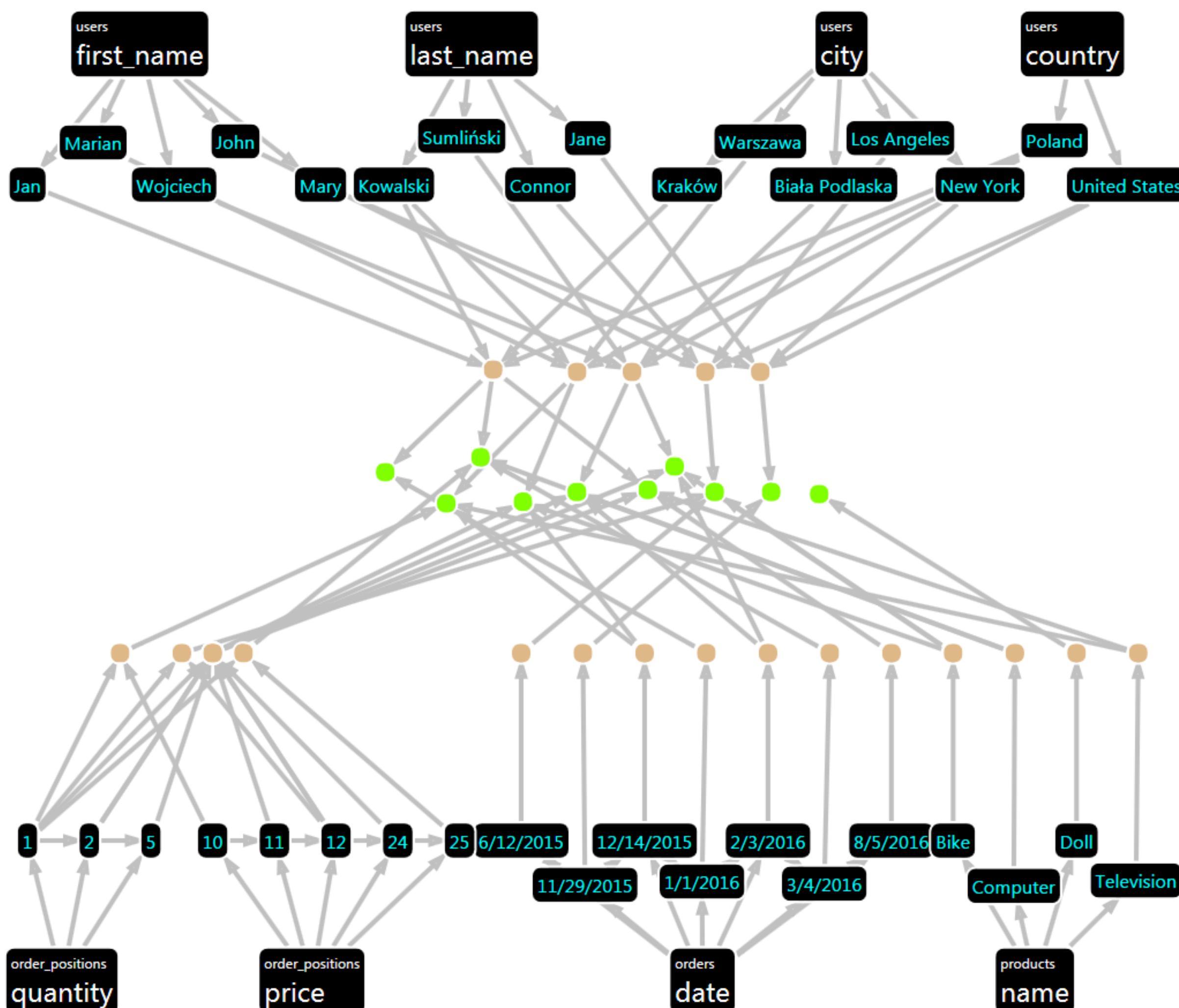
Dane przechowywane w takiej bazie są poddawane analizie leksykalnej, łańcuchy znaków są analizowane przez wiele rodzajów analizatorów i procesorów zarówno przy dodawaniu, jak i przy przeprowadzaniu zapytań. Te procedury służą do efektywnego przeszukiwania danych pod kątem pożądanych fraz.

Przykładami takich baz są Lucene[9] czy ElasticSearch[10].

[6] https://www.facebook.com/
[7] http://allegro.pl/
[8] http://allegro.tech/2015/02/search-engines-at-allegro.html
[9] https://lucene.apache.org/
[10] https://www.elastic.co/

Postać sieci skojarzeniowej AGDS

Utworzona w ramach niniejszej pracy sieć skojarzeniowa ma ściśle określoną strukturę. Owa struktura wraz z algorytmem tworzenia i ograniczeniami zostanie opisana szczegółowo w tym rozdziale. [20] [21]



Rysunek 2 Przykładowa sieć wygenerowana przez aplikację

Struktura sieci

Sieć oparta jest o graf skojarzeniowy AGDS (*associative graph data structure*) (18-23), zawiera jednak również kilka własnych elementów i uogólnień. Wprowadzono je głównie ze względu na prostotę rozwiązania, ale i usprawnienie mechanizmu odpytywania zbudowanej sieci.

Sieć składa się kilku kluczowych elementów:

- węzły atrybutów – *sensins*,
- węzły wartości – *value neurons,*
- węzły relacji w tabeli – *table relations*,

- główne węzły relacji – *main relations*.

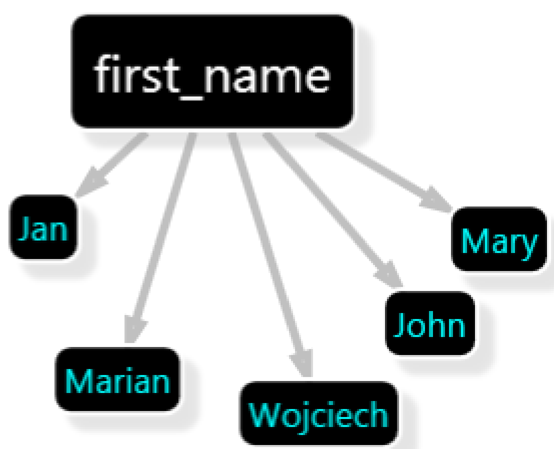
Wszystkie te elementy zostały bardziej szczegółowo opisane w kolejnych podrozdziałach.

Atrybuty

Podstawowym elementem grafu jest węzeł skojarzony z pojedynczym atrybutem (kolumną) tabeli bazy danych. Każda kolumna w schemacie bazy danych, która nie jest kluczem głównym ani obcym, ma odpowiadający jej węzeł atrybutu. Jest on połączony ze wszystkimi węzłami wartości, które w bazie reprezentują wartość odpowiadającej kolumny. Zdefiniowano dwa typy atrybutów. Przynależność wartości do poszczególnych rodzajów atrybutów jest określana przez typ tych wartości.

Artybuty niesortowalne

Jednym z typów węzłów atrybutów jest prostszy typ – atrybuty wartości, które nie są sortowalne. Wartości odpowiadające temu atrybutowi są nieuporządkowanym zbiorem wartości. Dostęp do poszczególnych wartości z tego atrybutu odbywa się jedynie bezpośrednio. Stąd też wynika jedynie połączenie atrybutu z wartością.



Rysunek 3 Atrybut z nieuporządkowanymi wartościami

Typem wartości, dla którego generowany jest atrybut łączący wartości niesortowalne jest typ łańcucha znaków (*string*).

Atrybuty sortowalne

Drugim typem, nieco bardziej skomplikowanym, jest atrybut o wartościach uporządkowanych. W przypadku tego atrybutu prócz połączenia wartości z atrybutem, wartości są również uporządkowane i łączone ze sobą zgodnie z kolejnością.

Rysunek 4 Atrybut z uporządkowanymi wartościami

Dzięki takiemu dodatkowemu połączeniu zyskujemy możliwość przede wszystkim bezpośredniego wyznaczenia wartości minimalnych i maksymalnych, jak i łatwo jesteśmy w stanie przechodzić po kolejnych wartościach w kierunku rosnącym i malejącym.

Typami wartości, które kojarzone są z atrybutem sortowalnym, są przede wszystkim typy liczbowe (typ całkowity, zmiennoprzecinkowy). Prócz tego, wartości związane z datą i czasem (*DateTime*).

Wartości

Węzły wartości przechowują przede wszystkim wartość pobraną z bazy danych. Pojedynczy węzeł zawiera połączenia do wszystkich węzłów relacji tabeli powstałych z wierszy, w których odpowiadająca mu wartość występuje. Oznacza to również, że w obrębie pojedynczego węzła atrybutu nie występują duplikaty wartości (patrz Rysunek 5 i Tabela 8).

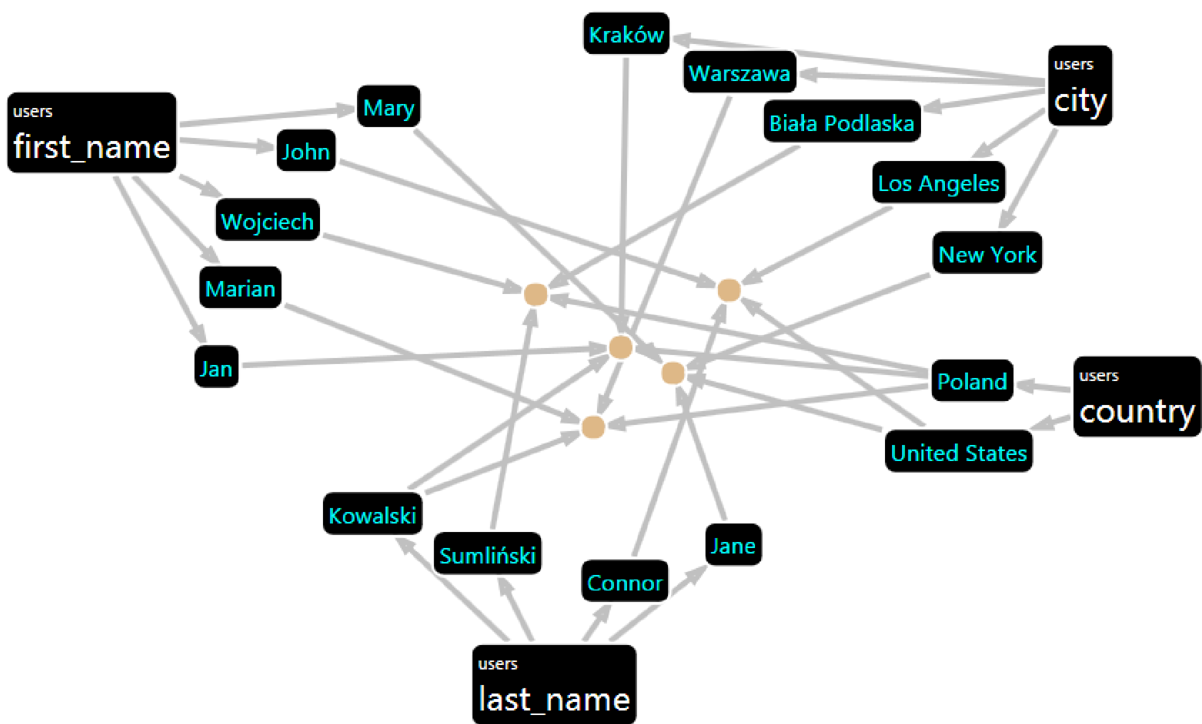
W zależności od typu wartości, jeżeli wartości są sortowalne, posiadają one również połączenia do sąsiadujących z nimi innych węzłów wartości, wynikające z ich kolejności. Stosowane są one do późniejszego wykonywania zapytań na tych wartościach.

Relacje

W sieci na potrzeby kojarzenia wartości ze sobą zostały zdefiniowane dwa rodzaje relacji: relacje tworzone na podstawie poszczególnych wierszy (*table relations*) i relacje główne, które łączą relacje pomiędzy tabelami (*main relations*).

Relacje tabeli

Pojedyncza relacja tabeli powstaje z jednego wiersza danych. Jak wspomniano wyżej, w obrębie jednego atrybutu w sieci nie ma żadnych duplikatów wartości, więc jeżeli wiersze dzielą w ramach jednego atrybutu tę samą wartość, odpowiadające im relacje przypisane są do tego samego, pojedynczego węzła wartości odpowiadającego temu duplikatowi. Zobrazowane jest to na poniższym rysunku.



Rysunek 5 Fragment grafu dla pojedynczej tabeli

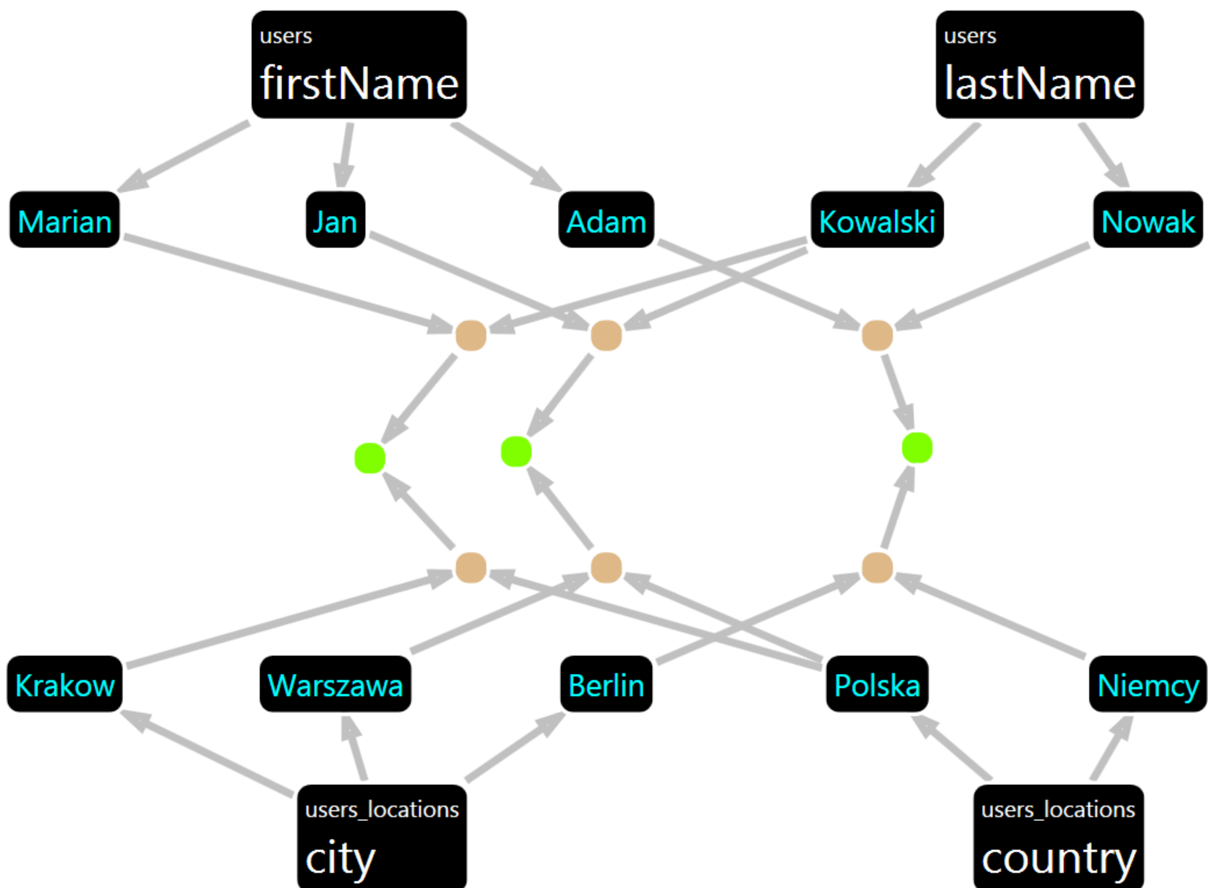
Na rysunku (Rysunek 5) można zauważyć pięć relacji tabeli (oznaczonych jako ●) odpowiadającym pięciu wierszom w tabeli źródłowej bazy danych. Warto zwrócić uwagę na wartości „Poland”, „United States” czy nazwisko „Kowalski” – każda z tych wartości ma więcej niż jedno połączenie do relacji. Jest to równoznaczne z wielokrotnym występowaniem tych wartości w swoich kolumnach. Poniżej przedstawiono omawianą tabelę wraz z danymi.

Tabela 8 Zawartość źródłowej tabeli

Imię	**Nazwisko**	**Miasto**	**Kraj**
Marian	Kowalski	Warszawa	Poland
Mary	Jane	New York	United States
Wojciech	Sumliński	Biała Podlaska	Poland
Jan	Kowalski	Kraków	Poland
John	Connor	Los Angeles	United States

Relacje pomiędzy tabelami

Krotki w poszczególnych tabelach łączą się w relacje, w aplikacji przedstawione jako relacje główne. Pojedyncza relacja główna odpowiada pojedynczemu obiektowi otrzymanemu przez złączenie wszystkich tabel bazy za pomocą zdefiniowanych kluczy głównych i obcych. W aplikacji oznaczone zostały jako zielone kropki ●.



Rysunek 6 Prosta sieć AGDS

Powyższa sieć skojarzeniowa odpowiada następującemu schematowi bazy danych:

Tabela 9 Tabela "users"

id	firstName	lastName
1	Marian	Kowalski
2	Jan	Kowalski
3	Adam	Nowak

Tabela 10 Tabela "users_locations"

userId	City	country
1	Kraków	Polska
2	Warszawa	Polska
3	Berlin	Niemcy

Warto zwrócić uwagę, że po złączeniu Tabela 9 i Tabela 10 (i odrzuceniu kolumn kluczy) otrzymamy następującą tabelę:

Tabela 11 Tabela będąca rezultatem złączenia Tabela 9 i Tabela 10

firstName	lastName	Country	city
Marian	Kowalski	Kraków	Polska
Jan	Kowalski	Warszawa	Polska
Adam	Nowak	Berlin	Niemcy

Nietrudno zauważyć analogię pomiędzy wierszami powyższej Tabela 11 i głównymi relacjami z Rysunek 6. Idąc od lewej, kolejne relacje główne odpowiadają kolejnym wierszom z powyższej tabeli.

Połączenia

W sieci zdefiniowano połączenia pomiędzy elementami poszczególnych warstw sieci, które są ze sobą skojarzone. Ponadto, jak wspomniano w podrozdziale Atrybuty sortowalne, połączenia występują również pomiędzy sortowalnymi wartościami w obrębie pojedynczego atrybutu.

W praktyce, w aplikacji każde połączenie jest dwukierunkowe, to znaczy podczas przechodzenia po elementach sieci, można się poruszać w dowolnym kierunku pomiędzy połączonymi ze sobą elementami.

Ograniczenia

W trakcie rozwoju aplikacji na wejściową bazę danych zostało narzucone ograniczenie - klucz główny może składać się z dokładnie jednej kolumny. Jednocześnie, klucz obcy wskazujący na pojedynczą krotkę w innej tabeli, może mieć co najwyżej jedną kolumnę.

Algorytm generowania sieci

Aplikacja automatycznie generuje sieć na podstawie danych bazy podanych na wejście. W tym rozdziale przedstawione zostały kroki algorytmu generowania sieci wraz z ich szczegółowym omówieniem. W celu ułatwienia zrozumienia algorytmu, w ramach algorytmu zostanie przedstawiony przykład wraz z prostą wizualizacją, utworzoną za pomocą programu MS Visio[11].

Połączenie z bazą danych

Pierwszym etapem jest ustanowienie połączenia z konkretną bazą danych. W dalszej części pracy przedstawiony jest sposób definiowania połączenia wewnątrz aplikacji (podrozdział Definicje baz danych).

Algorytm korzysta z abstrakcyjnego interfejsu połączenia z bazą danych. Na podstawie danych dostarczanych przez implementację tego interfejsu przeprowadza dalsze kroki

[11] https://products.office.com/pl-pl/visio/flowchart-software

tworzenia sieci. Szczegóły interfejsu są przedstawione w podrozdziale Interfejs bazy danych rozdziału Dodatek A – definicje wybranych typów używanych w aplikacji.

W aplikacji zostały zastosowane trzy różne implementacje połączenia dla różnych źródeł danych.

Baza SQL Server

Implementacja połączenia do bazy SQL Server jest oparta o dane systemowe systemu bazy danych, z których odczytywane są informacje o wybranym schemacie. Przy podawaniu danych użytkownika do połączenia ważne jest, aby dany użytkownik miał uprawnienia dotyczące odczytywania struktury schematu. Gdy dane logowania nie zostaną podane, aplikacja podejmie próbę zalogowania się w kontekście użytkownika Windows, który ma uruchomioną aplikację (za pomocą flagi *Integrated Security=True*[12]).

Sam schemat powinien mieć ściśle określone relacje za pomocą kluczy głównych i obcych, gdyż to na ich podstawie budowana jest struktura tabel.

Baza MySQL

Podobnie, połączenie z bazą MySQL opiera się na danych systemowych mechanizmu bazy. Tu również ważne jest, by podany użytkownik miał uprawnienia do odczytu takich danych. W przypadku niezdefiniowania danych użytkownika, zostanie podjęta próba wykorzystania domyślnego użytkownika (o nazwie root, bez hasła).

Pliki CSV

Niestandardowym rozwiązaniem jest zdefiniowana implementacja wykorzystująca schematy zbudowane z plików CSV. Za jej pomocą można definiować schemat oparty na relacyjnej bazie wraz z danymi, tj.

- tabele,
- kolumny,
- klucze główne,
- klucze obce.

Kolumny mogą również definiować typ przechowywanych danych.

Tabele w ten sposób tworzonej bazy są definiowane przez pliki znajdujące się w określonym katalogu. Pod uwagę brane są pliki z rozszerzeniem CSV. Nazwa pliku jest odpowiednikiem nazwy tabeli. Poniższy Rysunek 7 przedstawia bazę CSV złożoną z pięciu tabel

---

[12] https://msdn.microsoft.com/en-us/library/bsz5788z.aspx

Name	Date modified	Type	Size
orders.csv	6/28/2016 7:54 PM	Microsoft Excel C...	1 KB
orders_positions.csv	6/28/2016 7:54 PM	Microsoft Excel C...	1 KB
product_categories.csv	6/28/2016 7:54 PM	Microsoft Excel C...	1 KB
products.csv	6/28/2016 7:54 PM	Microsoft Excel C...	1 KB
users.csv	6/28/2016 7:54 PM	Microsoft Excel C...	1 KB

Rysunek 7 Przykładowy schemat tabel z plików CSV

Kolumny w ramach pojedynczego wiersza rozdzielone są znakiem średnika. Pierwszy wiersz pliku definiującego pojedynczą tabelę zawiera informacje o kolumnach tej tabeli i rozpoczyna się od znaku *hash* (#). Możliwe formaty wpisów o kolumnie są następujące:

- `nazwa_kolumny*` - definicja kolumny będącej kluczem głównym tabeli,
- `nazwa_kolumny` – zwykła, niekluczowa kolumna danych,
- `nazwa_kolumny_fk`**`[nazwa_tabeli(nazwa_kolumny_pk)]`** – definicja kolumny będącej kluczem obcym
  - `nazwa_tabeli` – tabela będąca rodzicem definiowanej tabeli
  - `nazwa_kolumny_pk` – nazwa kolumny w tabeli `nazwa_tabeli` będącej kluczem głównym tej tabeli.

Określenie typu kolumny jest opcjonalne. Można to zrobić, poprzedzając nazwę kolumny nazwą typu w następujący sposób:

`[`**`nazwa_typu`**`] nazwa_kolumny`

gdzie `nazwa_typu` to jedna z czterech wartości:

- liczby całkowite – „int”,
- liczby rzeczywiste – „double”,
- data i czas – „datetime”,
- łańcuch znaków – „string” (typ domyślny).

Ponadto, prócz typu możliwe jest podanie konwertera wartości kolumny. Format jest następujący:

`[nazwa_typu `**`nazwa_konwertera`**`] nazwa_kolumny`

Obecnie na potrzeby przykładów zdefiniowane zostały dwa konwertery:

- currency – parsuje za pomocą natywnej funkcji platformy .NET wartość zapisaną w formacie waluty do liczby zmiennoprzecinkowej,
- datetimeoryearconverter – próbuje parsować ciąg znaków jako strukturę daty i czasu platformy .NET. W przypadku niepowodzenia zakłada, że podano jedynie rok w

postaci pojedynczej liczby i na jej podstawie tworzy obiekt daty z ustawionym rokiem zgodnym z podaną liczbą.

Wyznaczenie struktury bazy

Na podstawie listy tabel w bazie danych oraz kluczy głównych i obcych dla każdej z tabel, tworzona jest struktura grafowa określająca relacje pomiędzy tabelami. Dla każdej tabeli ustalane są tabele-rodzice oraz jednocześnie tabele-dzieci.

W ramach przykładu zostanie omówiony algorytm przetwarzający bazę danych o strukturze i zawartych w niej danych (Tabela 12 - Tabela 15) ,który może reprezentować prostą wersję aplikacji do przeprowadzania transakcji.

Tabela 12 Tabela "users"

Id	first_name	last_name	City	Country
1	Jan	Kowalski	Kraków	Poland
2	Marian	Kowalski	Warszawa	Poland
5	Mary	Jane	New York	United States

Tabela 13 Tabela "products"

Id	Name
1	Bike
2	Computer
4	Television

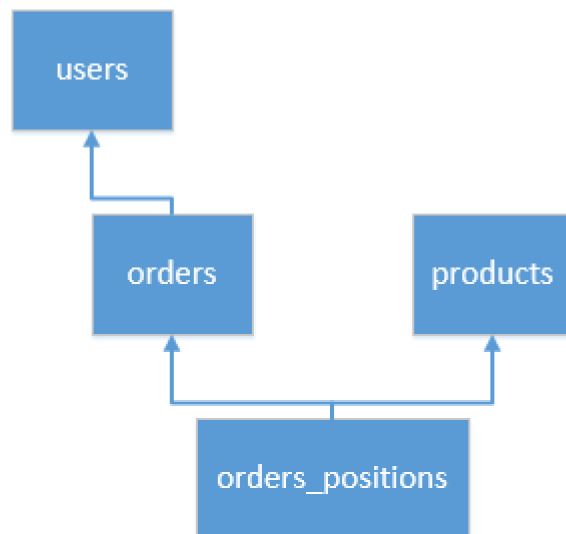
Tabela 14 Tabela "orders"

Id	userId	Date
1	1	2016-08-05
4	1	2016-03-04
5	2	2015-12-14
6	5	2015-11-29

Tabela 15 Tabela "orders_positions"

Id	orderId	productId	Quantity	Price
4	1	4	5	11
5	5	4	1	10
6	5	1	2	12
7	4	2	1	25

Z powyższego schematu bazy danych określane są zależności pomiędzy tabelami w postaci grafu (Rysunek 8):



Rysunek 8 Hierarchia zależności tabel przykładowej bazy

Za pomocą tak zbudowanej struktury zależności określana jest kolejność tabel do analizy. Pożądana kolejność tabel jest taka, że każda tabela występuje na liście przed wszystkimi innymi tabelami, które od niej zależą (to znaczy których rodzicem jest ta tabela). Innymi słowy, każda tabela występuje w liście po wszystkich tabelach, od których zależy (od wszystkich swoich rodziców). W związku z tym, w poprawnym z punktu widzenia aplikacji schemacie na początku tak wyznaczonej listy jest co najmniej jedna tabela, która nie ma żadnych rodziców – to one są analizowane w pierwszej kolejności. Następnie, lista może zawierać te z tabel, których rodzice już zostali uwzględnieni w liście, itd.

Zgodnie z tymi zasadami, poprawną kolejnością tabel do analizy jest: `users`, `products`, `orders`, `orders_positions` (nie jest to oczywiście jedyna poprawna konfiguracja).

Wyznaczona w ten sposób struktura zależności tabel jest również wykorzystywana na dalszym etapie tworzenia sieci – patrz podrozdział Tworzenie relacji głównych.

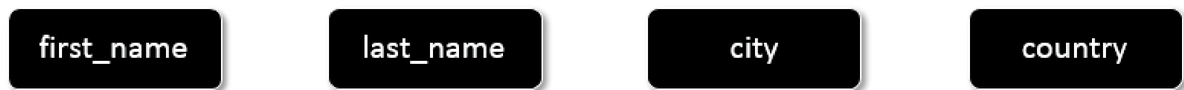
Analiza tabeli

Następnym etapem jest iteracja po wszystkich tabelach we wcześniej opisanej kolejności i utworzenie elementów kolejnych warstw sieci.

Tworzenie węzłów atrybutów

Najpierw, na podstawie kolumn tabeli tworzone są węzły atrybutów. Pod uwagę brane są *kolumny danych*, to znaczy takie kolumny, które nie są ani kluczami głównymi, ani obcymi. Dla każdej takiej kolumny tworzony jest pojedynczy atrybut.

Pierwszą analizowaną tabelą jest tabela `users`. Najpierw tworzone są atrybuty na podstawie kolumn danych:

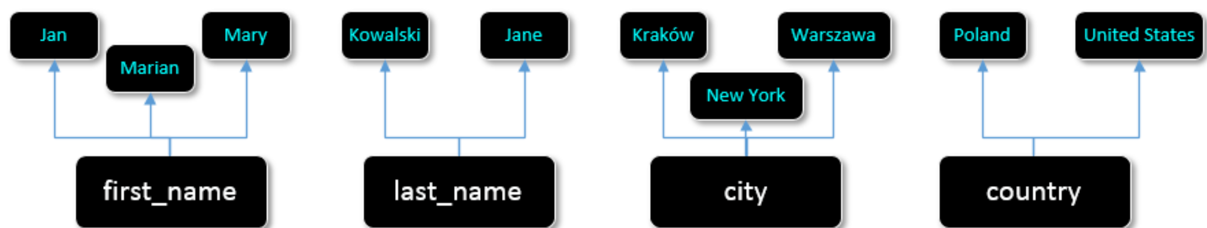


Rysunek 9 Węzły atrybutów na podstawie tabeli "users"

Tworzenie wartości

Dalej, następuje iteracja po wszystkich krotkach w tabeli. Na podstawie kolumn danych tworzony jest zestaw węzłów wartości. Są one łączone z odpowiednimi węzłami atrybutów na podstawie kolumn. Jeżeli w ramach danego atrybutu został już wcześniej utworzony węzeł o takiej samej wartości, to zamiast tworzenia duplikatu zwracany jest istniejący węzeł wartości.

Na podstawie krotek tabeli `users` tworzone są wartości:

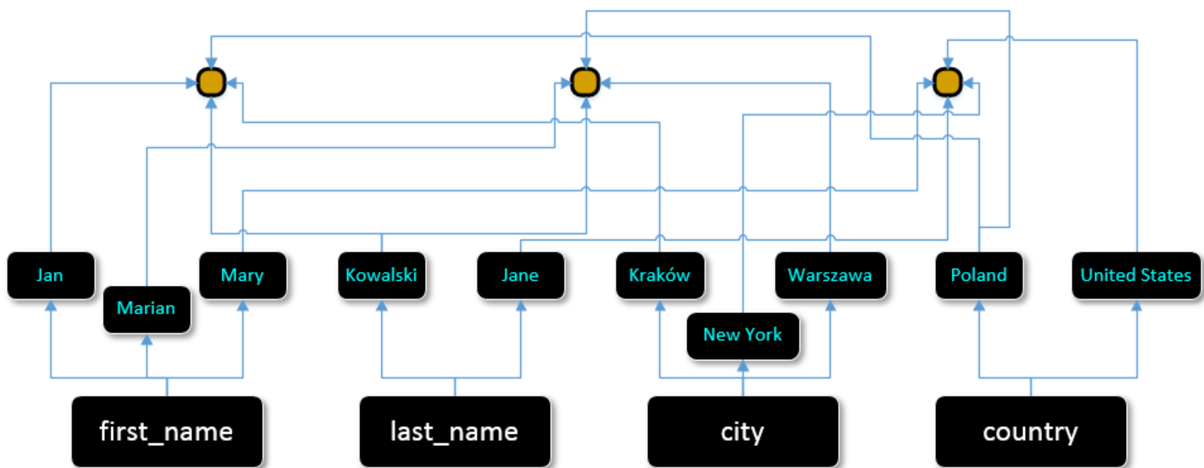


Rysunek 10 Sieć po utworzeniu wartości dla tabeli "users"

Tworzenie relacji tabeli

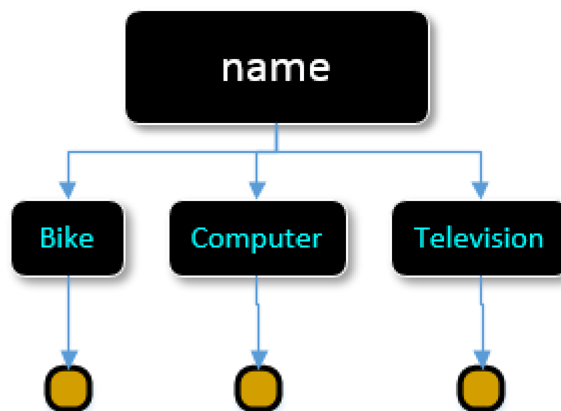
Uzyskana kolekcja wartości jest łączona nowo utworzoną relacją tabeli. Relacja tabeli jest następnie zapisywana w wewnętrznym słowniku D1 i jest identyfikowana za pomocą tabeli i wartości klucza głównego. Słownik D1 jest wykorzystywany jedynie na potrzeby algorytmu tworzenia sieci i służy do przechowywania utworzonych relacji tabeli pomiędzy kolejnymi iteracjami po wszystkich tabelach, by móc je łączyć z relacjami głównymi.

Dla tabeli `users` tworzone są relacje tabeli:



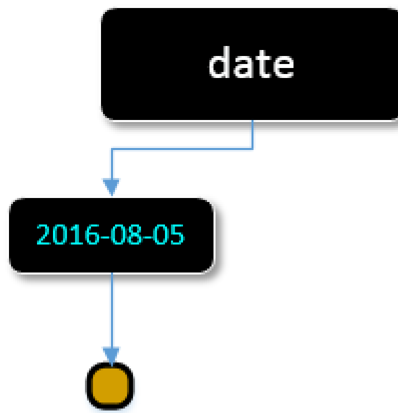
Rysunek 11 Sieć po utworzeniu relacji tabeli dla tabeli "users"

Na tym etapie kończy się przebieg iteracji dla tabeli `users` (ponieważ nie zawiera ona kluczy obcych). Te same kroki przeprowadzane są dla następnej tabeli – `products`. W niej również nie zdefiniowano żadnych kluczy obcych, więc iteracja kończy wykonanie na tym samym etapie, jak w przypadku tabeli `users`.



Rysunek 12 Fragment sieci na podstawie tabeli "products"

Kolejna tabela do analizy to tabela `orders`. Zawiera ona klucz obcy odnoszący się do tabeli `users`, dlatego w ramach iteracji po pierwszym wierszu danych, algorytm wygeneruje relację główną, co zostanie przedstawione w następnym podrozdziale (Tworzenie relacji głównych). Posiada tylko jedną kolumnę danych (`date`). Dla pierwszego wiersza tworzona jest wartość oraz relacja tabeli:



Rysunek 13 Fragment sieci dla tabeli "orders" dla pierwszej iteracji po wierszach

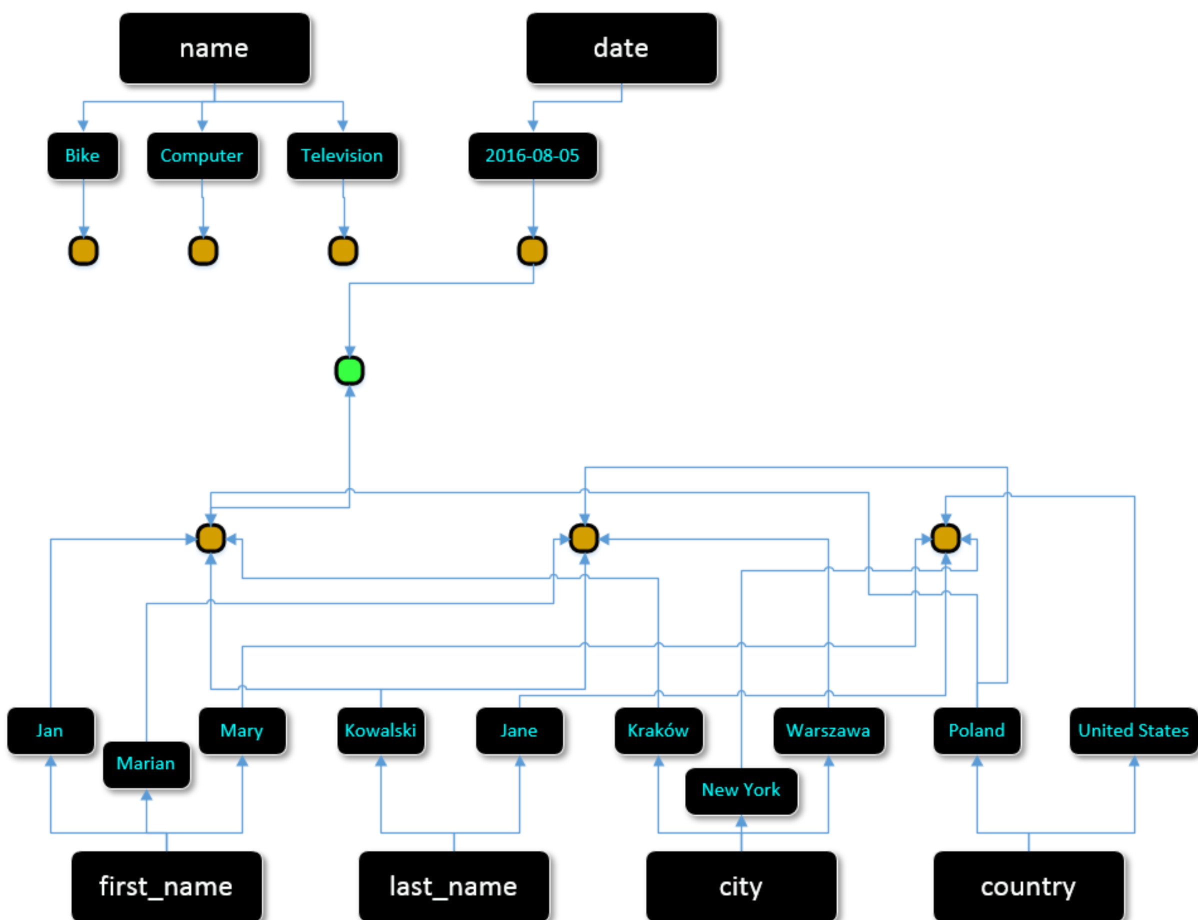
Tworzenie relacji głównych

Dla pojedynczego wiersza tabeli następuje iteracja po wszystkich jego kluczach obcych. Na podstawie informacji o tabeli obcej oraz wartości klucza głównego krotki skojarzonej z danym wierszem, podejmowana jest próba pobrania wcześniej zapisanej relacji głównej. W przypadku niepowodzenia, tworzona jest nowa relacja główna i dodawana jest do niej utworzona wcześniej relacja tabeli. Następnie, ta relacja główna jest zapisywana w wewnętrznym słowniku D2, identyfikowana na podstawie danej tabeli i wartości klucza głównego. Słownik D2, podobnie jak słownik relacji tabeli D1, służy jako struktura pomocnicza wyłącznie na rzecz algorytmu generowania sieci. Przechowuje relacje w celu ich późniejszego odczytu i aktualizacji w trakcie dalszych kroków algorytmu.

Przy pierwszej iteracji po kluczach głównych, jeżeli dla *obcej* krotki udało się pobrać istniejącą relację główną, to ona jest w dalszej części modyfikowana. Następuje sprawdzenie, czy pobrana relacja zawiera jakąkolwiek relację tabeli pochodzącej z aktualnie analizowanej tabeli. Jeżeli nie, to ta relacja główna jest modyfikowana i dodawana jest do niej relacja tabeli utworzona dla aktualnie analizowanego wiersza i zapisywana w słowniku (identyfikowana aktualną tabelą i kluczem głównym aktualnego wiersza). Jeżeli zaś relacja główna zawiera relację tabeli, która pochodzi z aktualnie analizowanej tabeli (na przykład z innego, wcześniej analizowanego wiersza), to ta relacja główna jest kopiowana w taki sposób, że tworzona jest nowa relacja główna i przypisywane do niej są relacje tabeli pochodzące z kopiowanej relacji głównej. Do tego przypisania z kopiowanej relacji głównej wybierane są tylko takie relacje tabeli, które pochodzą z tabel będących *przodkami* aktualnej tabeli (patrz Wyznaczenie struktury bazy). Na koniec iteracji po pierwszym kluczu obcym, rozważana relacja główna jest zapamiętana na czas iterowania po pozostałych kluczach obcych dla aktualnego wiersza.

Przy iteracjach po kolejnych kluczach obcych, pobrane obce relacje główne z danego klucza obcego są kopiowane na podobnej zasadzie jak przy pierwszym kluczu obcym i *złączane* z zapamiętaną relacją główną w pierwszej iteracji.

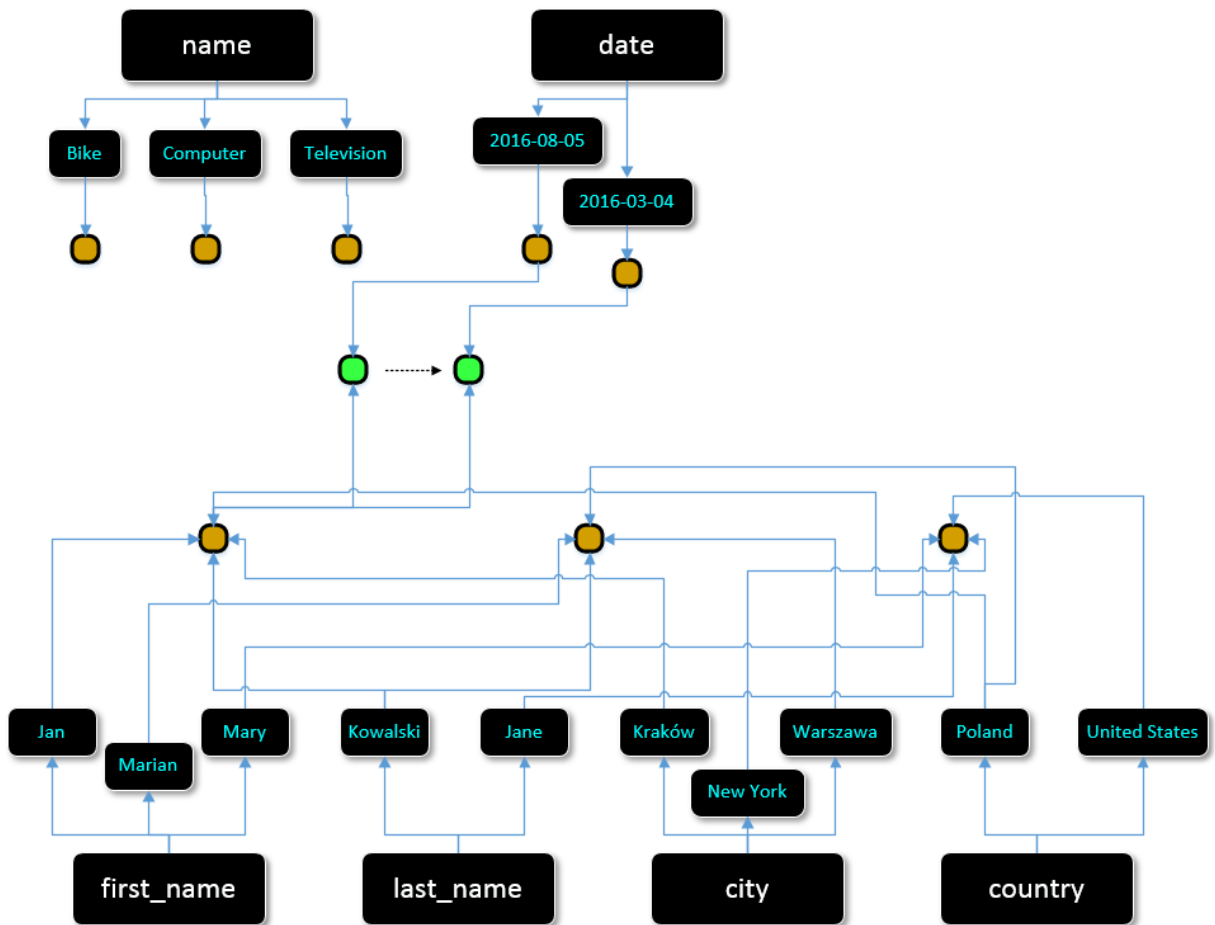
Będąc w trakcie pierwszego przebiegu pętli po wszystkich wierszach tabeli `orders`, przeprowadzana jest iteracja po wszystkich kluczach obcych tej tabeli. Dla jedynego znalezionego klucza (odnosi się do krotki o kluczu głównym 1 tabeli `users`) podejmowana jest próba pobrania istniejącej relacji głównej, skojarzonej z tą krotką. Dotychczas nie została utworzona żadna relacja, więc algorytm tworzy nową, dodaje do niej zarówno krotkę obcą jak i aktualny wiersz tabeli oraz zapisuje do wewnętrznego słownika D2 (zarówno identyfikując na podstawie obcej krotki jak i aktualnej).



Rysunek 14 Utworzenie pierwszej relacji głównej

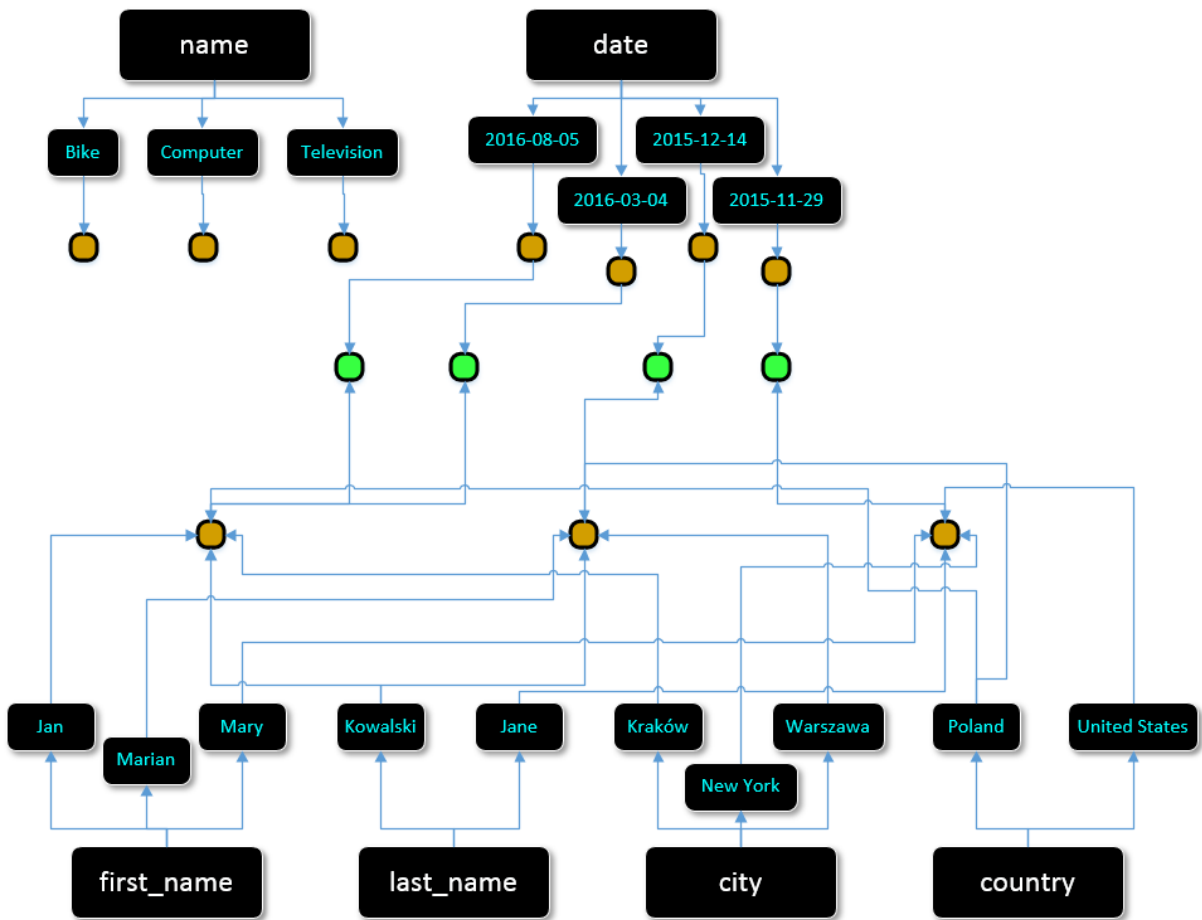
W przypadku drugiego wiersza algorytm również tworzy wartość i relację tabeli oraz sprawdza, czy dla podanego klucza głównego (wartość 1) w tabeli `products` istnieje już relacja główna. Tym razem stworzona przy poprzednim wierszu relacja zostaje pobrana i okazuje się, że jest już przypisana do innej relacji tabeli pochodzącej z tej tabeli. Tworzona jest więc kopia tej relacji głównej wraz ze wszystkimi relacjami tabeli pochodzącymi z tabeli zależnych. W tym wypadku jest to jedynie tabela `users`. Do tej relacji głównej dodana jest

właśnie utworzona relacja tabeli. Dla uproszczenia na rysunkach obrazujących sieć nie są uwzględniane połączenia pomiędzy sortowalnymi wartościami.



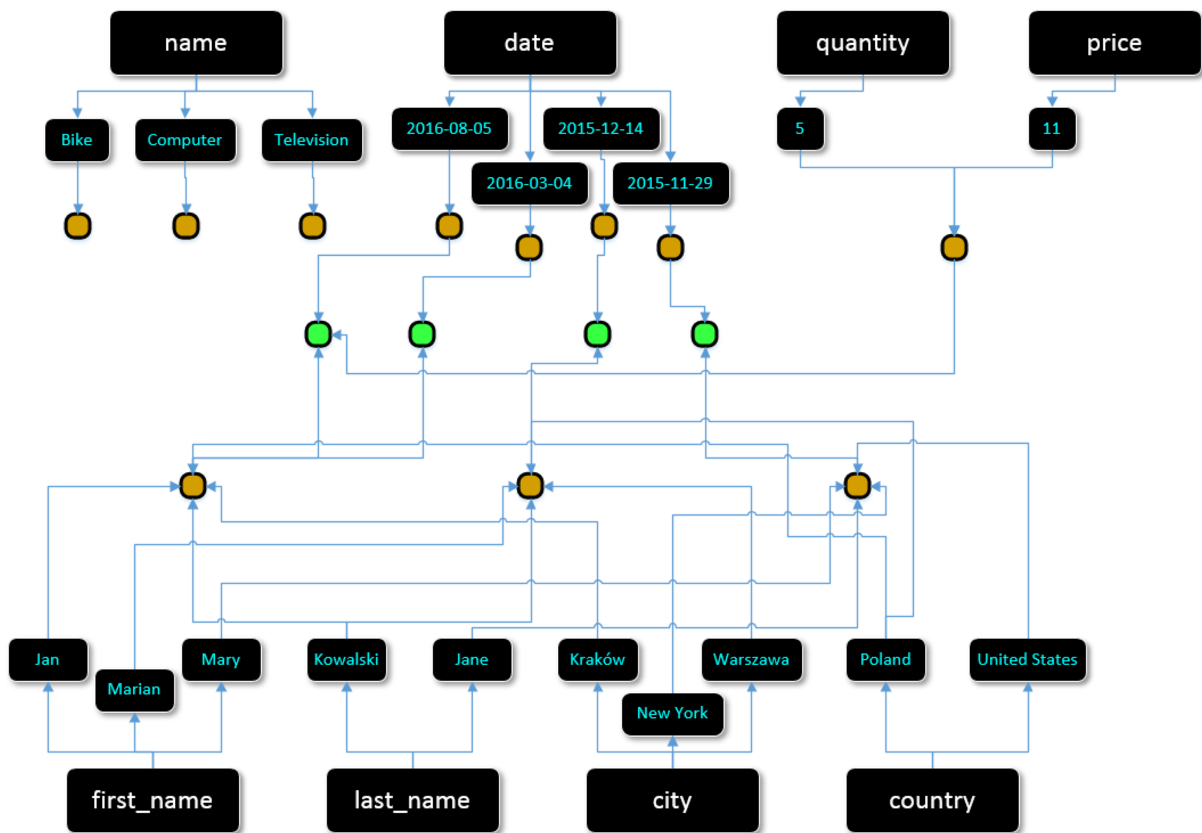
Rysunek 15 Utworzenie kopii relacji głównej i przypisanie nowo utworzonej relacji tabeli

Pozostałe wiersze generują elementy sieci na podobnej zasadzie jak pierwszy wiersz tabeli `orders`.



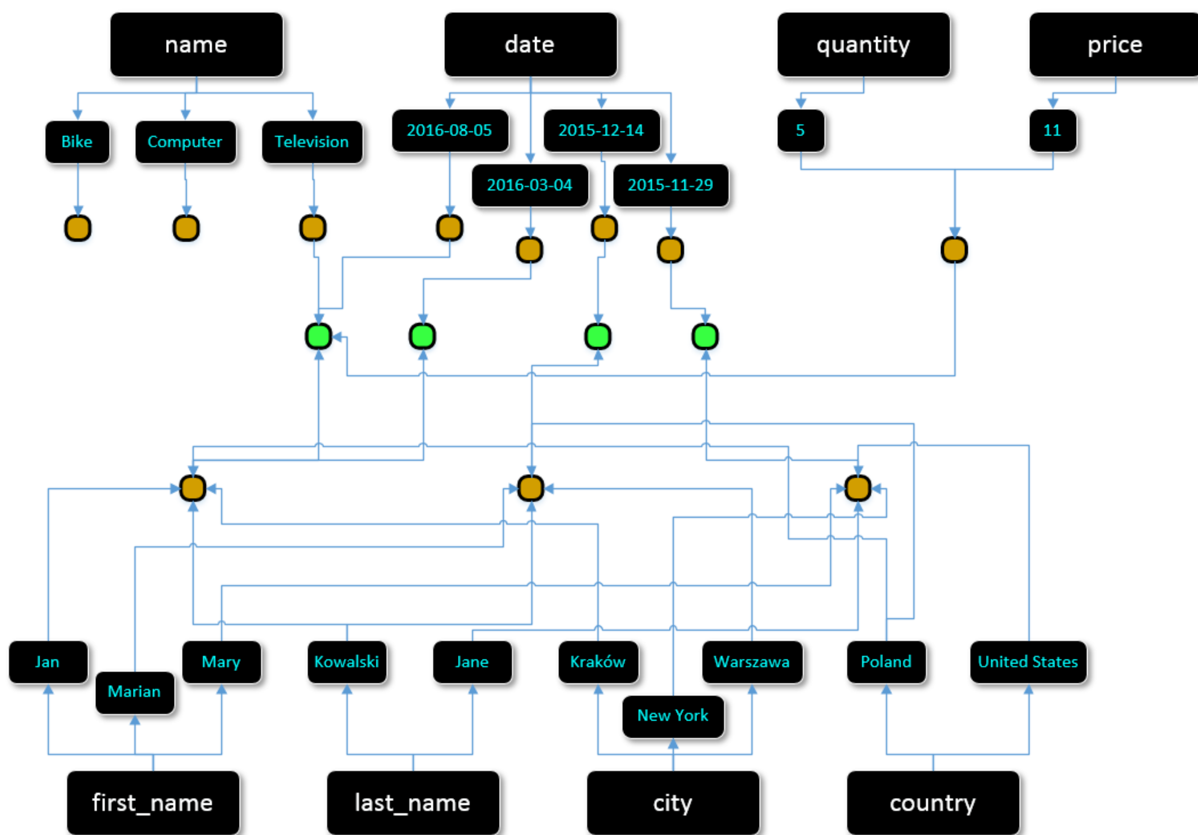
Rysunek 16 Sieć po zakończonej analizie tabeli "orders"

Algorytm przechodzi do ostatniej tabeli, `orders_positions`. Przy pierwszym wierszu tabeli i pierwszym kluczu obcym (tabela `orders`) do istniejącej relacji głównej dołączana jest nowo utworzona relacja tabeli z `orders_positions`. Ta relacja główna jest zapamiętywana na kolejne iteracje kluczy obcych.



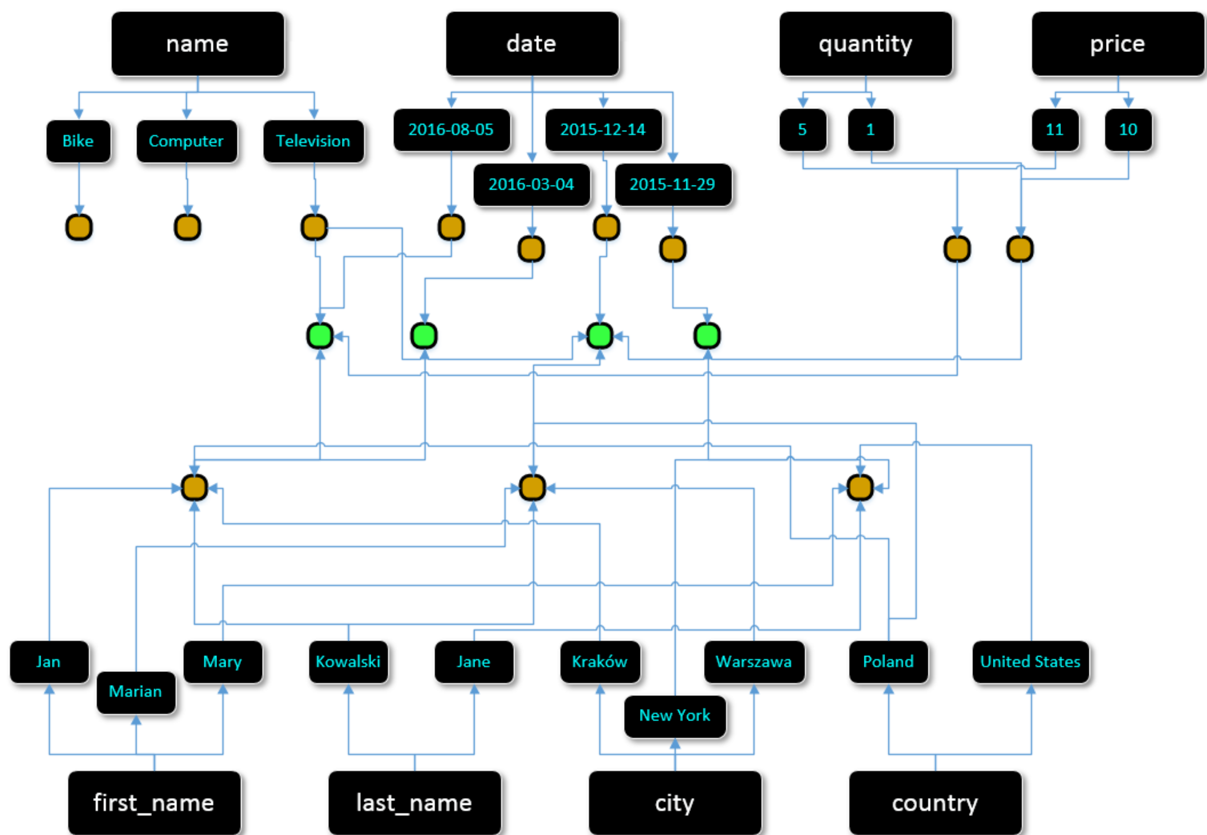
Rysunek 17 Sieć w trakcie iteracji po pierwszym wierszu tabeli "orders_positions" - pierwszy klucz obcy

Następnie, algorytm przechodzi do drugiego klucza obcego (tabela `products`). Dla tej tabeli nie zostały jeszcze utworzone żadne relacje główne, więc algorytm pobiera skojarzoną przez aktualny klucz obcy krotkę z tabeli `products` i dołącza ją do zapamiętanej relacji głównej.



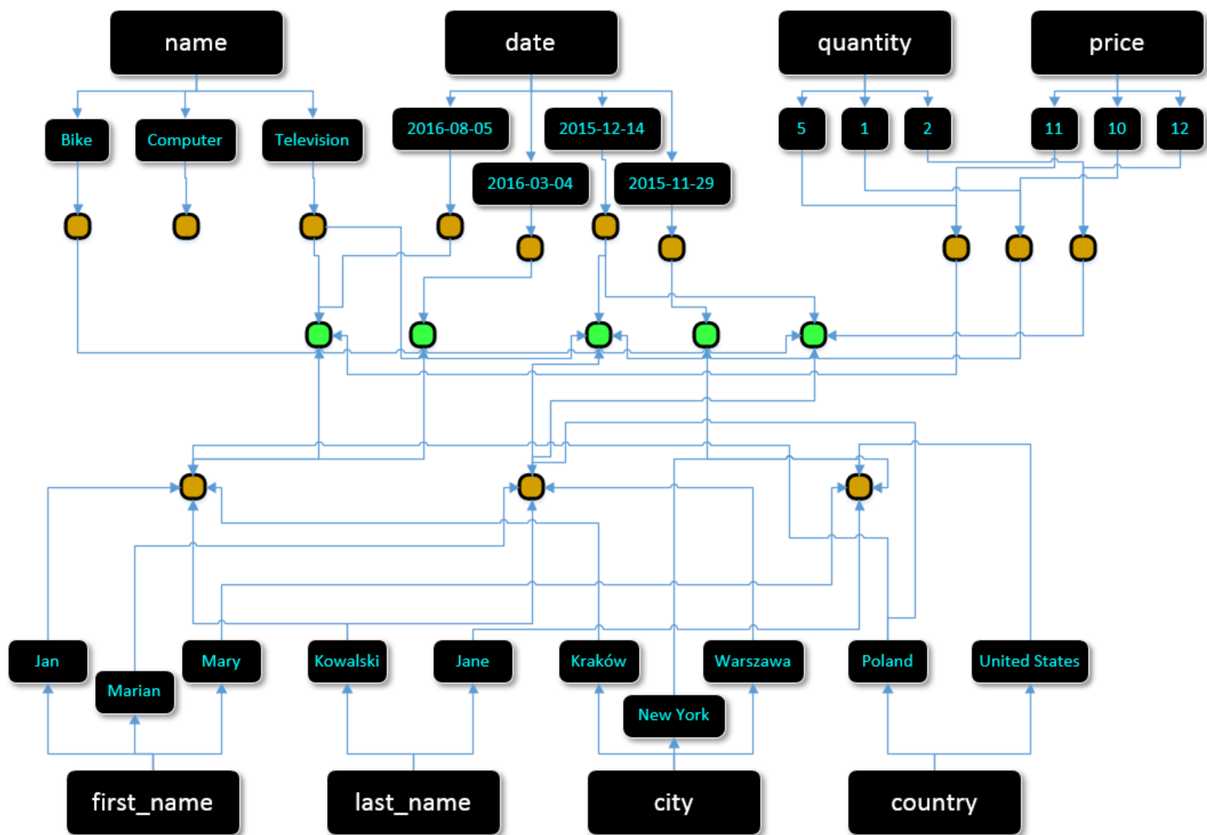
Rysunek 18 Sieć w trakcie iteracji po pierwszym wierszu tabeli "orders_positions" - drugi klucz obcy

Dalej, rozpoczyna się analiza drugiego wiersza tabeli. Pierwszy klucz obcy odnosi się do krotki tabeli `orders`, która jest już skojarzona z istniejącą relacją główną (*), więc utworzona relacja tabeli `orders_positions` dla drugiego wiersza jest dołączana do tej relacji głównej. Z kolei drugi klucz obcy (tabela `products`) odnosi się do tej samej krotki, do której odnosił się pierwszy wiersz tabeli `orders_positions`, więc dla tej obcej relacji tabeli istnieje już skojarzona relacja główna. Z tej istniejącej relacji głównej zostają zatem pobrane wszystkie te relacje tabeli, które pochodzą od tabel zależnych od tabeli `products` (i niej samej) – jest to tylko jedna relacja tabeli z tabeli `products`. Ta pobrana relacja tabeli zostaje dołączona do istniejącej, zapamiętanej relacji głównej (*).



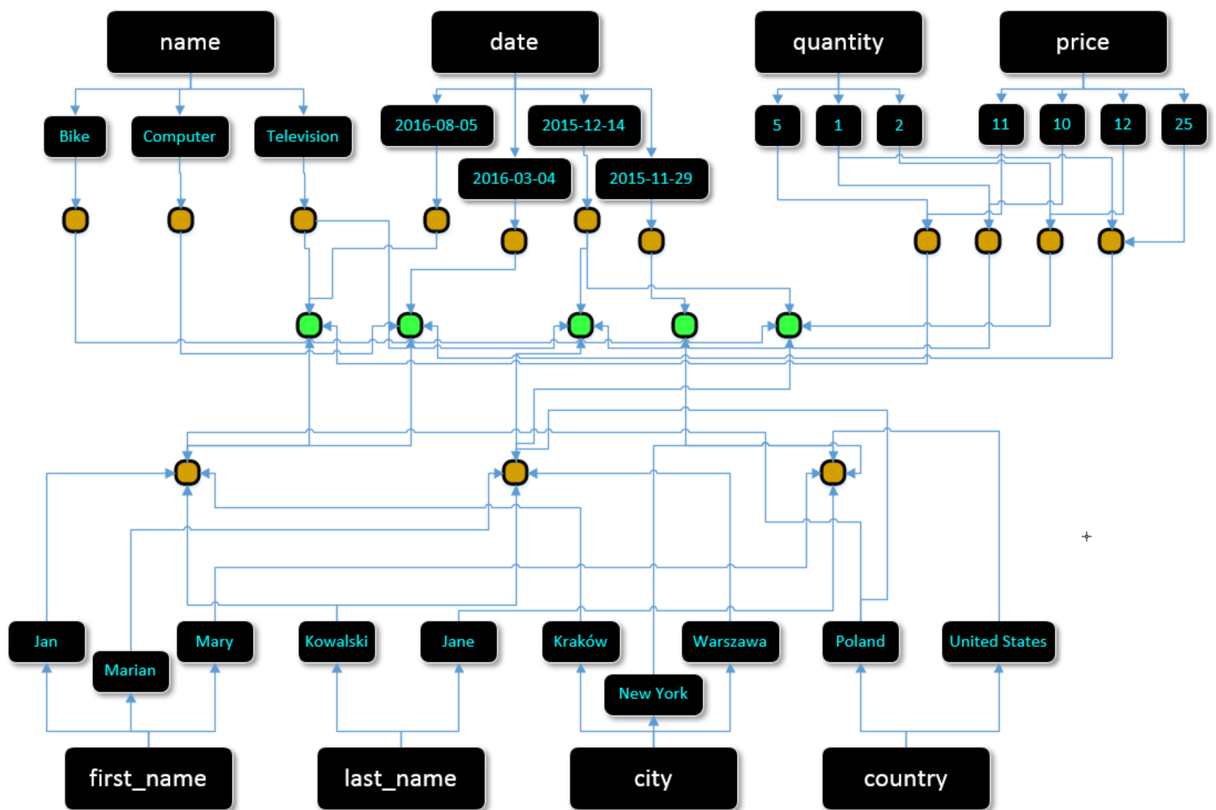
Rysunek 19 Sieć po iteracji po drugim wierszu tabeli "orders_positions"

Przy iteracji po trzecim wierszu tabeli orders_positions okazuje się, że pierwszy klucz obcy (z tabeli `orders`) odpowiada relacji tabeli, która ma skojarzoną główną relację. W tym wypadku jednak ta relacja główna posiada już taką relację tabeli, która pochodzi od tabeli `orders_positions`. W związku z tym ta relacja główna jest kopiowana i uwzględniane są relacje tabeli pochodzące od tabel zależnych od `orders` (wraz z nią samą). Są to zatem tabele `orders` i `users`. Do tej skopiowanej relacji głównej dołączana jest nowo utworzona relacja tabeli `orders_positions` dla aktualnego, trzeciego wiersza. Relacja tabeli pochodząca z drugiego klucza obcego (tabela `products`) dla aktualnego wiersza nie posiada skojarzonej relacji głównej – ta relacja tabeli jest więc dołączana do zapamiętanej przy pierwszym kluczu obcym relacji głównej.



Rysunek 20 Sieć po iteracji po trzecim wierszu tabeli "orders_positions"

Dla wiersza czwartego tabeli `orders_positions` tworzenie relacji i połączeń odpowiadających obu kluczom obcym przebiegają podobnie jak dotychczas.



Rysunek 21 Ostateczna postać sieci

Kwerendy dla sieci skojarzeniowych

W ramach niniejszej pracy zrealizowany został dedykowany język służący do przeprowadzania zapytań do sieci skojarzeniowej. Poniżej znaleźć można szczegółowy opis języka.

Opis języka

Składnia utworzonego języka zbliżona jest do języka SQL. Zdefiniowano trzy rodzaje operacji: odczyt danych, dodawanie nowych relacji oraz usuwanie istniejących. Podobnie jak SQL, utworzony język jest niewrażliwy na wielkość znaków.

Słowa kluczowe

Język posiada następujące słowa kluczowe:

- `SELECT`
- `INSERT`
- `DELETE`
- `WHERE`
- `OR`
- `AND`
- `NOT`

Atrybuty

Wiele operacji wymaga zdefiniowania atrybutu (kolumny) do działania. Nazwy atrybutów odpowiadają nazwom tabel i kolumn źródłowej bazy danych. Składnia wyboru atrybutu jest następująca:

```
nazwa_tabeli.nazwa_kolumny
```

Konieczne jest podanie obu nazw – w przeciwnym razie aplikacja nie będzie w stanie poprawnie określić atrybutu. W przypadku podania nieistniejącej nazwy, aplikacja wypisze komunikat o błędnej nazwie kolumny.

Typy danych

W kwerendach konieczna jest możliwość podawania wartości. Utworzony język umożliwia podanie następujących typów w podanych formatach (przedstawionych jako wyrażenia regularne[13]):

- Typ: liczba całkowita
  Opis: opcjonalny znak (plus lub minus), co najmniej jedna cyfra
  Format: `(-|+)? [0-9]+`

[13] https://pl.wikipedia.org/wiki/Wyra%C5%BCenie_regularne

Przykłady: `1; 134; +12; -766`
Typ .NET: `int`

- Typ: liczba rzeczywista
  Opis: opcjonalny znak (plus lub minus), co najmniej jedna cyfra i separator (kropka) ALBO separator i co najmniej jedna cyfra ALBO co najmniej jedna cyfra, separator, co najmniej jedna cyfra. W przypadku braku liczby po jednej ze stron separatora, aplikacja zakłada wartość zerową (części całkowitej lub dziesiętnej)
  Format: `(-|+)? ([0-9]+\.|\.[0-9]+|[0-9]+\.[0-9]+)`
  Przykłady: `1., 5.7, -7.1, -90., -.7`
  Typ .NET: `double`
- Typ: data
  Opis: trzy liczby rozdzielone kropkami, oznaczające: dzień, miesiąc i rok
  Format: `[0-9]{1,2}\.[0-9]{1,2}\.[0-9]{4}`
  Przykłady: `1.2.2012, 06.1.1990, 12.02.1992`
  Typ .NET: `DateTime`
- Typ: łańcuch znaków
  Opis: dowolny zestaw znaków pomiędzy dwoma podwójnymi cudzysłowami
  Format: `\".*\"`
  Przykłady: `"mama", "", "234fsdfsdf", "abc@def", "w_mas-of^der"`
  Typ .NET: `string`

Warto nadmienić, że wartości liczbowe mimo ewentualnego niedopasowania typu (na przykład liczba całkowita (`int`) porównywana z liczbą rzeczywistą) są konwertowane, gdy jest taka potrzeba (tj. przed przeprowadzeniem porównania itp.). Wspomniana konwersja opiera się na standardowej konwersji typu z biblioteki .NET[14].

Wyrażenia warunkowe, filtrowanie

Na potrzeby wybierania relacji z całej sieci zdefiniowany został system wyrażeń warunkowych. Dzięki nim można określić filtr zawężający zbiór relacji.

Składnia definiowania wyrażeń filtrujących jest następująca[15]:

[14] https://msdn.microsoft.com/en-us/library/system.convert.changetype(v=vs.110).aspx
[15] https://pl.wikipedia.org/wiki/Notacja_BNF

```
<wyrażenie> ::=           <wyrażenie> AND <wyrażenie>
                          | <wyrażenie> OR <wyrażenie>
                          | NOT ( <wyrażenie> )
                          | <funkcja>
                          | <wyrażenie_proste>
<funkcja> ::=             | nazwa (parametry)
<wyrażenie_proste> ::=    <atrybut> <operator_relacji> <wartość>
                          | <wartość> <operator_relacji> <atrybut>
<atrybut> ::=             nazwa . nazwa
<wartość> ::=             liczba | data | łańcuch znaków
<operator_relacji> ::=    < | <= | > | >= | =
```

Wyrażenia mogą być grupowane w nawiasy okrągłe – w ten sposób można narzucić priorytet ich wykonywania.

Z powyższej gramatyki wynika, że wyrażenia filtrowania mogą być definiowane rekurencyjnie i dowolnie zagnieżdżane i grupowane. Język definiuje trzy podstawowe operatory logiczne do łączenia warunków:

- iloczyn logiczny (AND) – zwraca relacje, które spełniają zarówno jeden jak i drugi warunek operatora,
- suma logiczna (OR) – zwraca relacje, które spełniają co najmniej jeden warunek operatora,
- zaprzeczenie (NOT) – zwraca relacje, które nie spełniają warunku.

Całe wyrażenie warunkowe buduje drzewo warunków łączonych ze sobą spójnikami logicznymi i tak też jest reprezentowane i przetwarzane w aplikacji.

Jak wspomniano wyżej, filtrowanie służy do ograniczania zbioru relacji. W istocie mechanizm filtrowania służy do pobudzania odpowiednich węzłów i zwraca zestaw relacji głównych, które spełniają określone warunki i które są później wyświetlane jako wyniki (przy wykorzystaniu SELECT) lub usuwane (dzięki DELETE). Sposób wybierania odpowiednich relacji zależy od wykorzystanego operatora.

Wyrażenie proste z powyższej gramatyki to wyrażenie postaci „atrybut operator wartość" (lub odwrotnie).

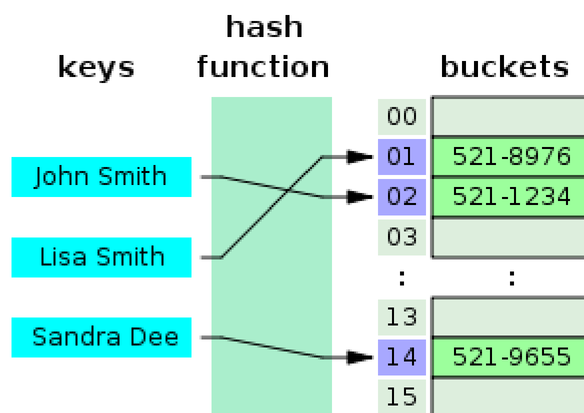
Operator równości

Gdy wyrażenie proste filtru ma postać sprawdzenia równości atrybutu i pewnej wartości, algorytm przeszukiwania relacji próbuje bezpośrednio z atrybutu pobrać węzeł odpowiadający tej wartości. Jeżeli wartość zostanie odnaleziona w ten sposób, wtedy wartość ta wzbudza odpowiadające jej relacje, aż do relacji głównych. W sytuacji jednak, gdy atrybut nie zwróci węzła dla podanej wartości, wyrażenie to nie pobudzi, a w konsekwencji nie zwróci żadnej relacji.

Dostęp do wartości (o ile wartość istnieje) jest natychmiastowy, ponieważ atrybut przechowuje wartości w strukturze *tablicy mieszającej*. Zapewnia to dostęp do danych w czasie O(1) (pod warunkiem braku *kolizji*). Tablica mieszająca jest strukturą danych umożliwiającą kojarzenie wartości z innymi na zasadzie relacji klucz-wartość. Zrealizowana jest w oparciu o tablicę danych, przechowującą wartości, a indeksowaną za pomocą liczby będacej wynikiem *funkcji mieszającej* dla klucza. Dostęp do danych skojarzonych z danym kluczem odbywa się następująco: najpierw dla klucza funkcja mieszająca wylicza wartość liczbową indeksu w tablicy, następnie za pomocą wyliczonego indeksu następuje odwołanie do odpowiadającego mu elementu w tablicy. W ten sposób jest możliwe natychmiastowe pobranie wartości skojarzonej z konkretnym kluczem.

W praktyce wystąpić mogą sytuacje, kiedy dla dwóch różnych kluczy funkcja mieszająca oblicza ten sam indeks w tablicy – jest to zjawisko kolizji. Istnieje kilka sposobów rozwiązywania problemu kolizji: metoda łańcuchowa, adresowanie otwarte etc. [22]

W ramach aplikacji, wykorzystano gotową implementacji tablicy haszującej – klasę generyczną `Dictionary` biblioteki standardowej .NET.

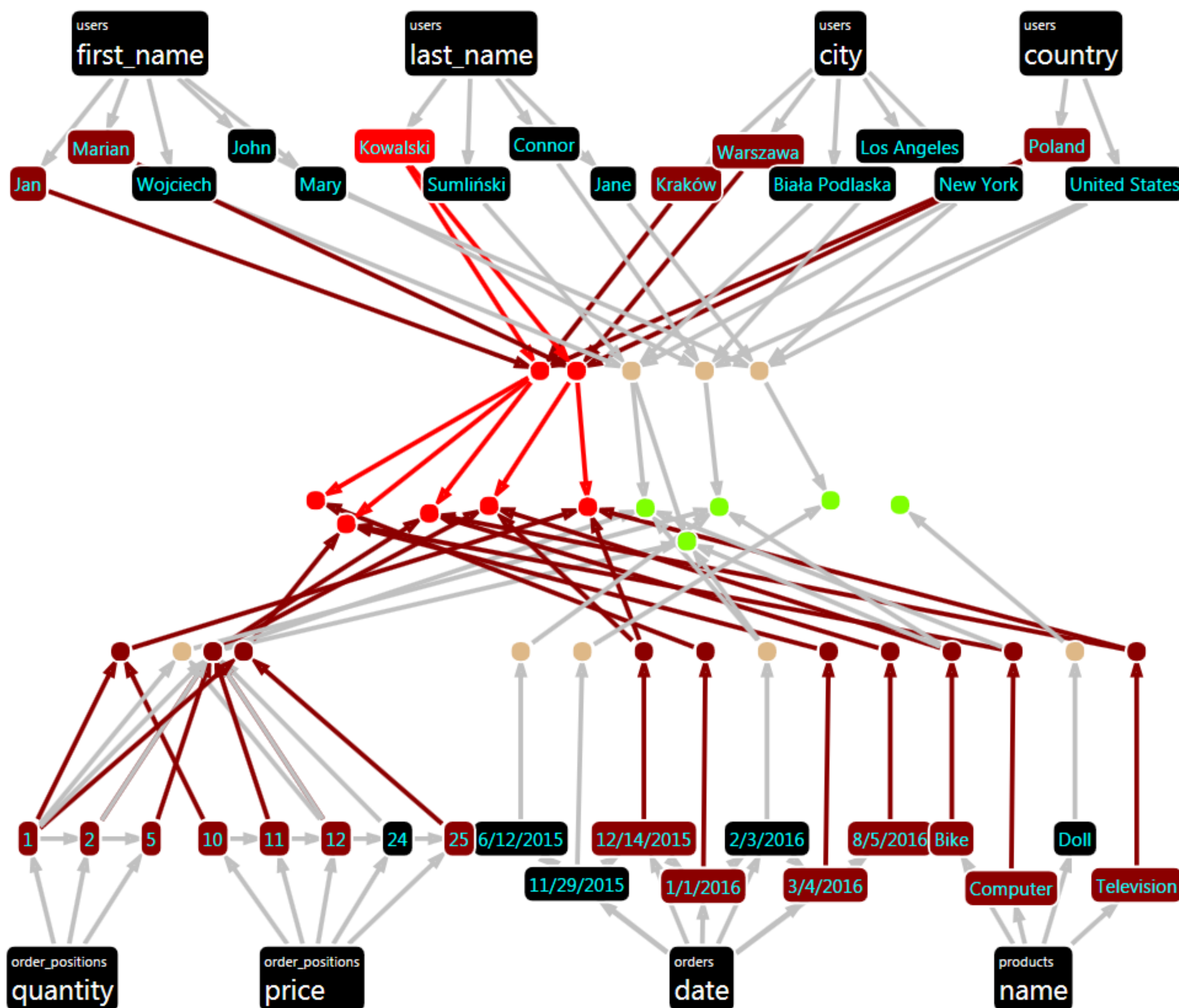


Rysunek 22 Wizualizacja działania tablicy mieszającej[16]

Na przykład, podanie prostego filtru dla sieci z poniższego Rysunek 23 postaci `users.last_name = "Kowalski"` wzbudza pojedynczą wartość „Kowalski. Jednocześnie,

[16] https://upload.wikimedia.org/wikipedia/commons/thumb/7/7d/Hash_table_3_1_1_0_1_0_0_SP.svg/315px-Hash_table_3_1_1_0_1_0_0_SP.svg.png

ta wartość pobudza dwie relacje tabeli (ponieważ wartość „Kowalski" występuje w dwóch różnych wierszach tabeli), a z kolei te dwie relacje tabeli pobudzają dalsze pięć relacji głównych.



Rysunek 23 Wzbudzenie w sieci asocjacyjnej wartości „Kowalski" powoduje reakcje powiązanych neuronów.

Ścieżka wzbudzenia *pierwotnego* od neuronu pobudzanego na początku do relacji głównych jest zaznaczona kolorem czerwonym, natomiast pobudzenia *wtórne*, to znaczy wywołane przez pobudzone wcześniej relacje główne, zostały zaznaczone kolorem ciemnoczerwonym (kierunki strzałek odwzorowują kierunek dodawania węzłów w grafie a nie przepływ informacji w wyniku działania algorytmu). Pobudzenie pierwotne poprzez relacje tabeli działa na relacje główne, w ten sposób wybierając krotki do zwrócenia przez aktywację. Z kolei pobudzenie wtórne służy do faktycznego wybrania węzłów danych do zwrócenia.

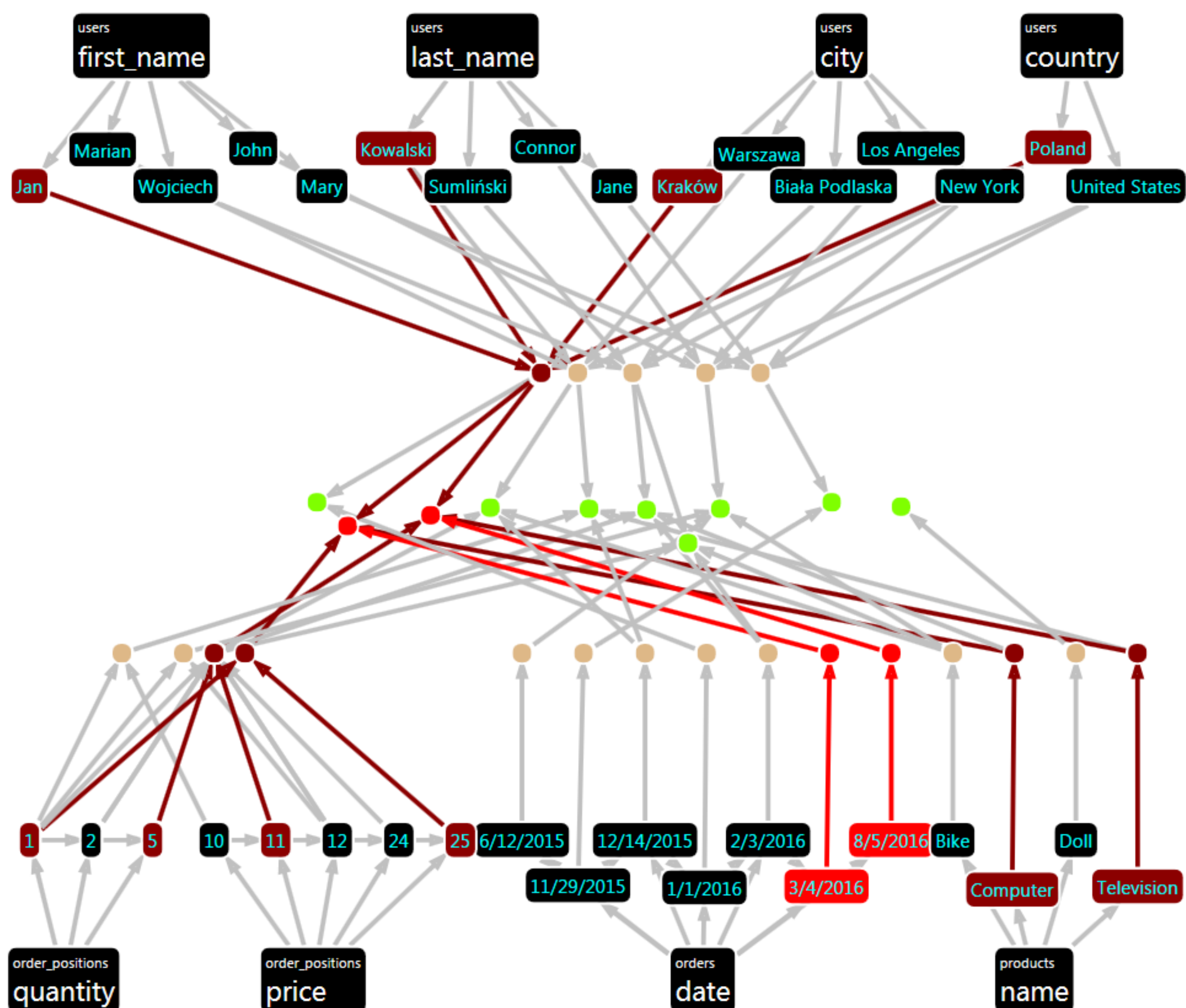
Operator nierówności

W przypadku operatora nierówności proces pobudzania wygląda nieco inaczej. Przede wszystkim, operator nierówności może być zastosowany do atrybutów, które przechowują typy sortowalne. To założenie umożliwia łatwe przechodzenie pomiędzy posortowanymi wartościami.

W zależności od typu nierówności wybierana jest wartość startowa dla dalszej części procedury filtrowania: gdy wyrażenie jest postaci „atrybut mniejszy od wartości” (jednoznaczne z „wartość większa od atrybutu”), wartością startową jest minimalna wartość dla atrybutu. Przeciwnie dla wyrażenia „atrybut większy od wartości” (lub „wartość mniejsza od atrybutu”) – wtedy wartością startową jest wartość maksymalna.

Następnie, algorytm rozpoczyna pobudzanie kolejnych węzłów wartości, poruszając się w zależności od typu nierówności w kierunku rosnącym (dla nierówności „atrybut mniejszy od wartości”) lub malejącym (dla przypadku „atrybut większy od wartości”). Oczywiście w przypadku, gdy żadna wartość nie spełnia warunku, wspomniane pobudzanie nie odbędzie się w ogóle.

Warto wspomnieć, że możliwe jest również określenie warunku w postaci nierówności słabej („mniejsze lub równe”, „większe lub równe”). Wtedy różnica w zachowaniu polega na tym, że algorytm uwzględnia przy sprawdzaniu warunku również te wartości, które są równe podanej w warunku.



Rysunek 24 Wzbudzenie dat większych niż 1 marca 2016

Na powyższym Rysunek 24 wykorzystany został warunek nierówności postaci `orders.date > 1.03.2016`. Wywołało to wybranie maksymalnej daty spośród wszystkich zdefiniowanych, czyli 5 sierpnia 2016 oraz pobudzanie kolejnych dat podążając w kierunku coraz mniejszych dat. Gdy napotkano pierwszą datę niespełniającą warunku, czyli 3 lutego 2016, pobudzanie zostało przerwane i zostały zwrócone dwie relacje główne.

Analogicznie, gdyby wybrać warunek `order_positions.price <= 11`, zostałyby pobudzone dwie wartości, począwszy od 10, a skończywszy na 11.

Funkcje

Sporym usprawnieniem w stosunku do klasycznego języka SQL jest wprowadzenie prostych, ale funkcjonalnych funkcji służących do filtrowania danych. W obecnej wersji aplikacji zdefiniowano kilka funkcji filtrujących dane.

Minimum, maksimum

Mowa tu o dwóch funkcjach: `min` i `max`. Służą one do pobudzania odpowiednio najmniejszej i największej wartości w ramach atrybutu. Można ich użyć do atrybutów sortowalnych, a ich

wysoka wydajność jest osiągana przez fakt natychmiastowej dostępności wartości minimalnej i maksymalnej z poziomu atrybutu.

Różnica w stosunku do tradycyjnego SQL polega na tym, że nie ma wbudowanego sposobu na zwracanie rekordów ze względu na wartość ekstremalną danego atrybutu. Możliwe jest jednak zbudowanie zapytania w taki sposób, by najpierw pobrać wartość ekstremalną rozważanego atrybutu, a potem pobrać wszystkie krotki mające dokładnie tę wartość atrybutu. Przykładowo:

```
SELECT * FROM orders WHERE price = (SELECT MAX(price) FROM ORDERS)
```

zwróci wszystkie zamówienia o maksymalnej kwocie. W aplikacji równoważne zapytanie wygląda następująco:

```
SELECT * WHERE MAX(orders.price)
```

Funkcja daje również możliwość zwrócenia kolejnych wyników, począwszy od ekstremalnego. Za pomocą parametru `top` można ograniczyć liczbę zwróconych wyników do pierwszych N relacji, na przykład poniższe zapytanie zwróci co najwyżej trzy pierwsze wyniki o maksymalnej kwocie:

```
SELECT * WHERE MAX(orders.price, top=3)
```

Prócz tego, zamiast arbitralnego ustalenia liczby wyników do zwrócenia, można określić maksymalne odchylenie od wartości ekstremalnej. Służy do tego parametr `maxDeviation`. Wartość podana za pomocą tego parametru powinna być z zakresu [0; 1] – określa ona maksymalny stosunek różnicy wartości atrybutu relacji i wartości ekstremalnej do różnicy wartości ekstremalnych atrybutu:

$$D_{max} = \frac{|a - v_{ext}|}{v_{max} - v_{min}}$$

Równanie 1 Wzór odchylenia wartości atrybutu

gdzie:
a – wartość atrybutu dla danej relacji
v_{ext} – wartość ekstremalna: maksymalna dla funkcji max, minimalna dla funkcji min
v_{max} – wartość maksymalna
v_{min} – wartość minimalna

Poniższe zapytanie zwraca wszystkie zamówienia, których cena mieści się w pierwszych 10% wartości maksymalnych:

```
SELECT * WHERE MAX(orders.price, maxDeviation=0.1)
```

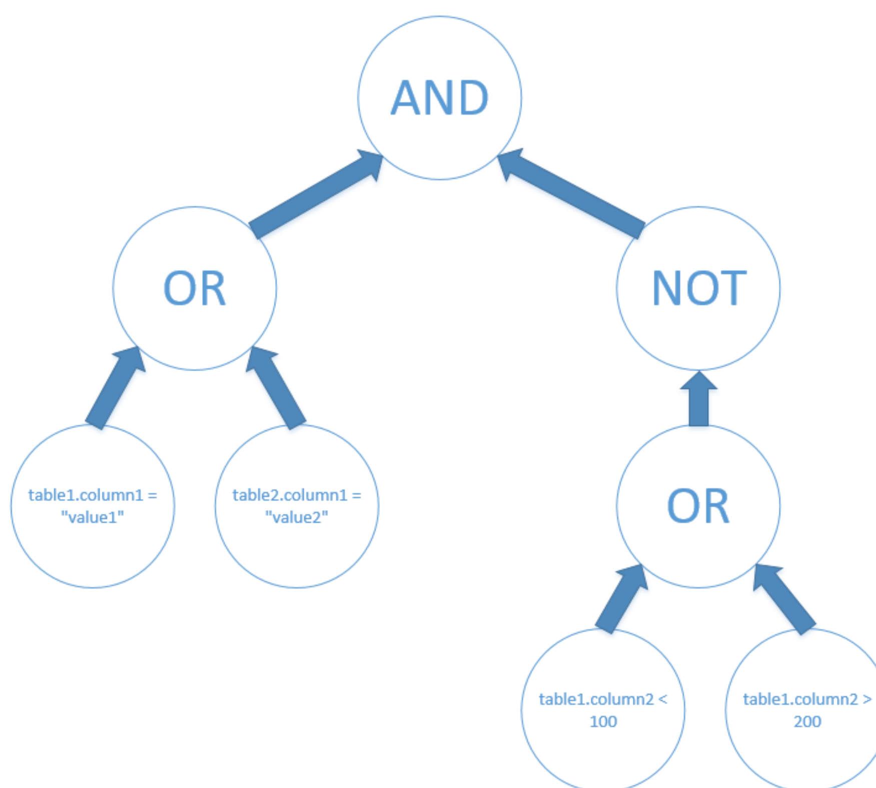
Przetwarzanie wyrażenia warunkowego

Wyrażenie warunkowe może być przedstawione w formie drzewa. Liściami drzewa są operacje *aktywujące*, tzn. takie operacje, które dokonują aktywacji relacji poprzez wartości. Operacje aktywujące to np. operator porównania równościowego, nierówność czy funkcje warunkowe. Z kolei elementami pośrednimi drzewa są operacje logiczne łączące inne operacje, tj. suma logiczna (OR), iloczyn logiczny (AND) i zaprzeczenie (NOT).

Przykładowe złożone wyrażenie logiczne może mieć następującą postać:

```
table1.column1 = "value1" OR table2.column1 = "value2" AND NOT
(table1.column2 < 100 OR table1.column2 > 200)
```

Powyższe wyrażenie może zostać opisane za pomocą następującego drzewa wyrażenia:



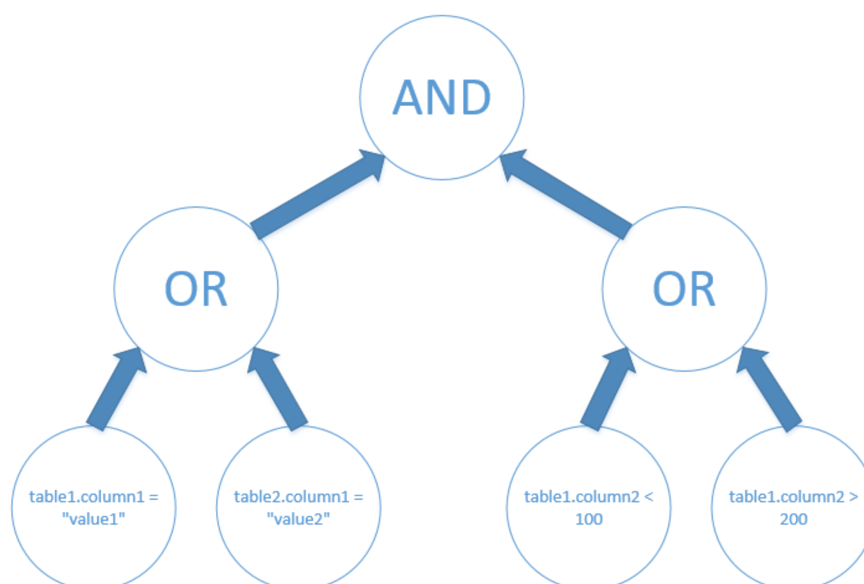
Rysunek 25 Wizualizacja złożonego wyrażenia logicznego

Dwa kolejne podrozdziały opisują mechanizm ewaluacji złożonego wyrażenia warunkowego i zbierania wyników w dwóch różnych przypadkach – gdy wyrażenie nie zawiera żadnych zaprzeczeń oraz gdy zawiera co najmniej jedno. Oba przypadki różnią się od siebie w zasadniczy sposób.

Opisywany mechanizm bazuje na nieco zmodyfikowanym algorytmie asocjacyjnej ewaluacji złożonych warunków logicznych na strukturach AGDS. Oryginalny algorytm został opracowany przez promotora Pana dr hab. Adriana Horzyka i przedstawiony mi na potrzeby realizacji tej pracy.

Wyrażenie warunkowe bez zaprzeczeń

Podczas parsowania zapytania zawierającego wyrażenie w zdefiniowanym języku, każda operacja otrzymuje unikalny identyfikator (są to kolejne liczby całkowite począwszy od 0). Przed ewaluacją zapytania, każdy węzeł relacji inicjalizuje tablicę aktywacji o długości równej liczbie pojedynczych operacji wyrażenia (liczbie węzłów drzewa wyrażeń). Dla drzewa wyrażenia przedstawionego poniżej, będącego w istocie poprzednim wyrażeniem (Rysunek 25) zmodyfikowanym przez usunięcie zaprzeczenia, liczba węzłów drzewa wyrażenia wynosi 7 – tyle zatem elementów mają tablice aktywacji relacji głównych sieci przed wykonaniem zapytania.



Rysunek 26 Drzewo wyrażenia warunkowego bez zaprzeczeń

Spełnienie pojedynczego warunku (równości, nierówności itp.) przez wartość jest jednoznaczne z aktywacją tej wartości. Głównym elementem ewaluacji złożonego wyrażenia jest przejście po wszystkich operacjach aktywujących i wzbudzenie relacji powiązanych z wartościami wskazanymi przez operacje aktywujące.

Pobudzenie relacji w ten sposób skutkuje zapisaniem informacji o spełnieniu danej operacji w tablicy aktywacji węzła (pod indeksem równym identyfikatorowi danej operacji aktywującej) oraz propagacją w górę drzewa wyrażenia (zgodnie ze strzałkami z Rysunek 25 i Rysunek 26) faktu aktywowania danej relacji. W ten sposób informacja o aktywacji jest przekazywana do jednego z dwóch rodzajów operacji: sumy albo iloczynu logicznego.

Operacja sumy zaraz po otrzymaniu informacji o aktywowaniu danej relacji, przekazuje ją dalej w górę oraz zapisuje fakt aktywacji w tablicy relacji pod indeksem równym swojemu identyfikatorowi. Taka natychmiastowa aktywacja wynika wprost z zasady działania sumy logicznej – przynajmniej jeden z argumentów musi być spełniony.

W przypadku gdy informacja o aktywowaniu relacji trafia do operacji iloczynu logicznego, dalsza propagacja w górę nie dzieje się od razu. Zamiast tego operacja AND sprawdza, czy obie operacje będące jej argumentami zostały również aktywowane (w ramach drzewa wyrażenia istnieją połączenia zarówno w górę drzewa, jak i w dół, niemniej jedynie operacja AND wykorzystuje połączenia w dół). Dopiero gdy ten warunek zostanie spełniony, operacja zapisuje aktywację w tablicy aktywacji węzła relacji i przekazuje informację o aktywacji w górę.

W celu efektywnego zbierania wyników, operacja logiczna będąca korzeniem drzewa jest łączona ze specjalnym blokiem, do którego przekazuje informacje o aktywowaniu poszczególnych relacji. Blok ten dodaje otrzymane relacje do listy i zwraca je po zakończeniu ewaluacji. Rozwiązanie to, w celu zapewnienia efektywności, wymaga jednak, aby wszystkie operacje aktywowały poszczególne relacje co najwyżej raz. Spełnienie takiego warunku było łatwe – wystarczyło do operacji sumy dodać sprawdzenie uniemożliwiające wielokrotne aktywowanie tego samego węzła relacji.

Wyrażenie warunkowe z zaprzeczeniami

Wprowadzenie zaprzeczenia do zapytania zdecydowanie zmniejsza wydajność zapytania. Dzieje się tak, ponieważ wykonanie zaprzeczenia wymaga wybrania i aktywacji wszystkich relacji, które nie spełniają podanego warunku, który jest zaprzeczany. Następnie tak wybrany zbiór relacji będący różnicą pomiędzy zbiorem wszystkich relacji i tych, które do operacji NOT trafiły jako aktywowane przez operację-argument zaprzeczenia, jest aktywowany w taki sam sposób jak dotychczas (tj. ustawiana jest flaga w tablicy aktywacji poszczególnych węzłów relacji i następuje propagacja w górę drzewa wyrażenia).

Na potrzeby obsługi negacji do mechanizmu przetwarzania został dodany dodatkowy mechanizm informowania o zakończeniu aktywowania wszystkich relacji przez operację – jest to sygnał, że dana operacja nie będzie już dokonywać żadnych aktywacji.

Wszystkie operacje prócz zaprzeczenia, wchodzące w skład drzewa warunku, przeprowadzają procedurę aktywacji w taki sposób, że w miarę napływania do nich informacji o aktywowaniu danej relacji, sprawdzają, czy dla tej relacji zachodzą warunki aktywowania przez tę operację, a następnie w przypadku spełnienia tych warunków natychmiast aktywują tę relację. Operacja NOT natomiast w odpowiedzi na informacje o aktywacjach, nie dokonuje aktywacji natychmiast. Zamiast tego, zbiera wszystkie relacje, które są przekazywane z operacji-argumentu i dodaje do listy operacji *zanegowanych*. Dopiero w momencie uzyskania komunikatu o zakończeniu aktywacji, wybiera ze wszystkich relacji w sieci te, które należy aktywować, dokonuje aktywacji tych relacji i przekazuje informacje o zakończeniu aktywacji do operacji nadrzędnej.

Odczyt danych

Do odczytu danych z sieci, podobnie jak w SQL, służy słowo kluczowe SELECT. Składnia zapytania SELECT jest następująca:

```
<zapytanie> ::=   SELECT <kolumny> WHERE <wyrażenie>

<kolumny> ::=     <kolumna>

                  | <kolumna> , <kolumny>

<kolumna> ::=     nazwa kolumny

                  | funkcja

                  | *

<wyrażenie> ::=   [patrz rozdział Wyrażenia warunkowe, filtrowanie]
```

Od klasycznego zapytania SELECT znanego z SQL powyższą implementację odróżniają istotne szczegóły:

- klauzula WHERE wraz z podaniem funkcji filtrujących jest obligatoryjna dla zapytania. Bez niej aplikacja zwróci komunikat o błędzie zapytania;
- brakuje części FROM, służącej w SQL do wybierania pojedynczej tabeli (lub kilku tabel) do pobrania danych;
- nie zdefiniowano części złączających tabele JOIN.

Mimo powyższych różnic, mechanizm odczytu danych z sieci działa w sposób zbliżony do pobierania danych z relacyjnej bazy danych utworzonej z tabel. Za pomocą algorytmu filtrowania i na podstawie podanych w zapytaniu warunków, uzyskuje zestaw relacji głównych, z których odczytuje określony w zapytaniu zestaw kolumn. Format uzyskanych danych jest sprowadzany do postaci tabeli, aby można je było wyświetlić w sposób czytelny.

Przykładowo, poprawnym zapytaniem jest poniższe, wykorzystujące filtr zobrazowany na Rysunek 24:

```
SELECT users.first_name, users.last_name, products.name
WHERE orders.date > 1.03.2016
```

Zapytanie to zwróci dwa wiersze:

Tabela 16 Wyniki zapytania o trzy kolumny z filtrem "większy niż"

users.first_name	users.last_name	products.name
Jan	Kowalski	Computer
Jan	Kowalski	Television

Warto zwrócić uwagę, że w wynikach w kolumnach users.first_name i users.last_name uzyskano zdublowane dane, mimo że na Rysunek 24 pobudzony jest jeden Jan Kowalski, przypisany do jednej relacji tabeli, a więc jedyny Jan Kowalski istniejący w bazie źródłowej.

Zamiast podawania wszystkich kolumn w ramach zapytania, możliwe jest użycie znaku gwiazdki, znanego z SQL. Spowoduje to wyświetlenie wszystkich atrybutów.

Funkcja podobieństwa

Możliwe jest również podanie funkcji jako dodatkowego wyniku do zwrócenia. W obecnej wersji zdefiniowana jest jedna funkcja – funkcja *podobieństwa*. Składnia tej funkcji jest następująca (produkcje `<atrybut>` i `<wartość>` zdefiniowane zostały już w podrozdziale Wyrażenia warunkowe, filtrowanie):

```
<funkcja_podobieństwa> ::=  similar ( <wyrażenia> )

<wyrażenia> ::=             <wyrażenie>

                            | <wyrażenie> , <wyrażenia>

<wyrażenie> ::=             <atrybut> = <wartość>
```

Produkcja `<wyrażenia>` umożliwia zdefiniowanie nowego (lub istniejącego) obiektu w kontekście struktury sieci poprzez podanie zestawu par atrybut-wartość. Tak zdefiniowany obiekt służy dalej za wzorzec do ustalania podobieństwa. Ustalenie podobieństwa polega na obliczeniu wartości funkcji

$$P = \frac{\sum_{i=1}^{n} P_i}{n} \quad (1)$$

gdzie

$$P_i = 1 - \frac{|a_i - v_i|}{R_i} \quad (2)$$

a_i – wartość wzorcowa i-tego atrybutu,
v_i – wartość i-tego atrybutu rozważanego obiektu
R_i – różnica pomiędzy maksymalną a minimalną wartością w ramach i-tego atrybutu

Jako porównywane atrybuty można zdefiniować zarówno atrybuty sortowalne, jak i niesortowalne. W przypadku atrybutów niesortowalnych, funkcja wartości podobieństwa jest postaci

$$P_i = \begin{cases} 1 & gdy\ wartość\ jest\ taka\ sama\ jak\ wzorcowa \\ 0 & gdy\ wartość\ jest\ różna\ od\ wzorcowej \end{cases} \quad (3)$$

Przykładowo, możliwe jest określenie stopnia podobieństwa dla obiektu o określonych wartościach:

```
SELECT employees.first_name, employees.last_name,
similar  (  employees.birth_date  =  01.05.1980,  employees.salary  =
5000)
WHERE employees.hire_date > 01.01.2000
```

W powyższy sposób można określić podobieństwo wszystkich pracowników zatrudnionych od 2000 roku do wzorcowego pracownika urodzonego 1 maja 1980 roku i zarabiającego 5000.

Dodawanie danych

Język umożliwia dodawanie nowych danych do zbudowanej sieci. W ramach pojedynczego zapytania możliwe jest dodanie jednej relacji. Składnia kwerendy dodającej relację danych jest następująca:

```
<zapytanie> ::=           INSERT VALUES ( <przypisania> )

<przypisania> ::=         <przypisanie>

                          | <przypisanie> , <przypisania>

<przypisanie> ::=         kolumna = wartość
```

Podobnie jak procedura tworząca sieć, kwerenda dodawania danych łączy istniejące wartości lub tworzy nowe, gdy podane w zapytaniu nie istnieją jeszcze w sieci.

Usuwanie danych

Aplikacja pozwala również na usunięcie relacji, które zostały utworzone w sieci. Składnia zapytania usuwającego relacje wygląda następująco

```
<zapytanie> ::=     DELETE WHERE <wyrażenie>

<wyrażenie> ::=     [patrz rozdział Wyrażenia warunkowe, filtrowanie]
```

Procedura usuwania relacji za pomocą filtrów wybiera relacje główne, które zostały przeznaczone do usunięcia. Następnie usuwa te relacje oraz połączenia ze skojarzonymi relacjami głównymi. Aby maksymalnie zaoszczędzić miejsce, mechanizm sprawdza te relacje tabeli, które dotychczas były skojarzone z usuniętymi relacjami głównymi. Jeżeli znajdzie wśród nich takie, które po usunięciu relacji głównych nie mają już skojarzonych żadnych innych relacji, automatycznie usuwa je. Podobnie, sprawdzone zostają wartości skojarzone z usuniętymi relacjami tabeli – jeżeli wartość pozostanie bez skojarzonej relacji tabeli, zostaje ona usunięta. Te mechanizmy odpowiadają mechanizmom usuwania kaskadowego stosowanego w mechanizmach relacyjnych baz danych.

Aplikacja

W tym rozdziale zostały opisane techniczne aspekty związane z utworzoną aplikacją. Wymienione i opisane zostały technologie użyte w systemie, jak również ogólna architektura całej aplikacji i lokalne wzorce zastosowane do rozwiązania lokalnych problemów. Zawarty na końcu podrozdział Elementy aplikacji jest prostą instrukcją wykorzystania samego interfejsu utworzonej aplikacji.

Technologie

Cała aplikacja wraz ze wszystkimi komponentami została napisana na platformie .NET firmy Microsoft. Wykorzystane do tego zostało środowisko programistyczne Visual Studio 2015 wraz z dodatkiem Resharper, ogromnie ułatwiającym pisanie i zarządzanie kodem. Sam kod wykorzystywał wszelkie zalety nowej wersji języka C# 6.

W aplikacji zostało użyte szerokie spektrum różnych bibliotek pomocniczych i całych frameworków opisanych poniżej.

Windows Presentation Foundation

Interfejs użytkownika został oparty o framework Windows Presentation Foundation (w skrócie WPF) firmy Microsoft. Jest to cały zestaw bibliotek programistycznych wspierających tworzenie kompleksowych aplikacji klienckich typu *desktop*. WPF jest następcą bardzo popularnego frameworka o podobnych zastosowaniach – Windows Forms.

Największą różnicą pomiędzy WPF a Windows Forms jest możliwość pisania kodu widoków za pomocą deklaratywnego języka XAML (*eXtensible Application Markup Language*) [23] bazującego na języku XML, a stworzonego również przez firmę Microsoft. Technologia XAML jest wykorzystana również w innych produktach Microsoft, np. Windows Phone/Windows Mobile czy Xamarin.

Framework WPF cechuje się dużą elastycznością i stosunkową łatwością tworzenia własnych widoków, począwszy od prostych (formularze, tabelki), a skończywszy na skomplikowanych komponentach (grafy, wykresy). Wykorzystując WPF można w prosty sposób definiować bardziej skomplikowane zachowania interfejsu jak animacje, a wydajność takich operacji jest zwiększona dzięki architekturze WPF wykorzystującej efektywnie procesor karty graficznej. [24]

GraphSharp

GraphSharp[17] jest biblioteką ułatwiającą wizualizację grafowych struktur danych. Projekt ten nie jest już aktywnie wspierany[18], jednak został wybrany ze względu na relatywną łatwość

[17] https://graphsharp.codeplex.com/

wykorzystania we własnym projekcie. Ponadto, brakuje innych, ogólnodostępnych i niekomercyjnych bibliotek umożliwiających łatwe przedstawianie grafów na graficznym interfejsie użytkownika.

Biblioteka zawiera kilka predefiniowanych algorytmów umożliwiających ułożenie węzłów i krawędzi grafu, jednak żaden z nich nie wyglądał zadowalająco z punktu widzenia przedstawienia grafu AGDS. W związku z tym, został napisany prosty algorytm definiujący wygląd wygenerowanej sieci. Ogólny schemat wyglądu sieci narysowanej za pomocą omawianego algorytmu jest następujący:

Atrybuty (a więc i wartości, i relacje) były grupowane wg tabel źródłowej bazy w celu równomiernego rozmieszczenia relacji i połączeń z wartościami. Jako że omawiany algorytm nie jest perfekcyjny, po wygenerowaniu sieci możliwe jest też ręczne przesuwanie węzłów grafu.

Autofac

Autofac[19] jest biblioteką ułatwiającą stosowanie wzorca wstrzykiwania zależności (*Dependency Injection* - DI), który jest jednym z „dobrych praktyk" programowania obiektowego i realizacją zasady *odwrócenia zależności* (*Inversion of Control* - IoC), będącej jedną z zestawu pięciu zasad programowania obiektowego SOLID[20].

Zgodnie z zasadą odwrócenia zależności, implementacja pojedynczej części logiki systemu (na przykład klasa A) nie powinna zależeć bezpośrednio od implementacji innych części logiki (na przykład klasa B), z których korzysta. Zamiast tego, klasa B powinna mieć ściśle określony interfejs operacji, które dostarcza, a klasa A powinna zależeć od wspomnianego interfejsu klasy B, bez znajomości szczegółów realizacji operacji tego interfejsu.

[18] https://www.nuget.org/packages/GraphSharp/ - ostatnia aktualizacja pod koniec 2012 roku (dane na 3 sierpnia 2016)
[19] https://autofac.org/
[20] https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)

Istnieje kilka sposobów realizacji wzorca wstrzykiwania zależności: Service Locator (statyczna fabryka konkretnych instancji potrzebnego interfejsu), wstrzykiwanie przez metody piszące do obiektu (*mutator*[21]) oraz wstrzykiwanie przez konstruktor. Z tych trzech podejść najwygodniejszym wydaje się być ostatnie ze względu na całkowitą niezależność od konkretnego mechanizmu wstrzykiwania i zachowanie czystości oraz przejrzystości kodu. Autofac obsługuje wszystkie z podanych podejść wstrzykiwania zależności.

Zastosowanie tego wzorca istotnie wpłynęło na dwa aspekty wytwarzania aplikacji. Przede wszystkim, większość kodu była łatwa do automatycznego przetestowania za pomocą testów jednostkowych, a kluczowe implementacje, które ewoluowały dość intensywnie podczas rozwoju aplikacji, były stale weryfikowane, dzięki czemu nastąpiła znaczna oszczędność czasu związana z ewentualnymi błędami – były one natychmiast wykrywane i usuwane. Ponadto, dzięki wsparciu w wybieraniu konkretnych implementacji do zastosowania podczas działania aplikacji, jakie zapewnia Autofac, pojedyncze komponenty mogły być łatwo zastępowane nowszymi wersjami w miarę rozwoju, bez krytycznych zmian innych fragmentów programu, niezwiązanych z daną zmianą.

xUnit, Fluent Assertions, NSubstitute

Jest to zestaw bibliotek wspierających pisanie czytelnych i efektywnych testów jednostkowych.

xUnit jest frameworkiem zapewniającym podstawowe klasy pomocne przy definiowaniu testów jednostkowych. Dzięki xUnit, od programisty wymagana jest minimalny nakład pracy w celu zdefiniowania testu jednostowego – sprowadza się to do określenia *atrybutu*[22] przy implementacji testu w kodzie C#.

Fluent Assertions to zestaw rozszerzeń dla bibliotek do testów jednostkowych (nie jest powiązany z xUnit – można użyć dowolnego innego frameworka). Usprawnia on pisanie i zarządzanie stworzonymi testami poprzez zdecydowane poprawienie czytelności kodu testów jednostkowych. Dzięki Fluent Assertions kod testów, a zwłaszcza przeprowadzanie *asercji* rezultatów testowanych implementacji jest pisany w składni naturalnej dla człowieka. Przykładowo, bez użycia Fluent Assertions porównanie wartości wyniku testu z oczekiwaną wartością może wyglądać tak:

```
Assert.AreEqual(receivedResult, expectedResult);
```

Mimo że zrozumienie takiej asercji nie jest zadaniem trudnym, linia kodu odpowiedzialna za to porównanie nie jest możliwa do przeczytania przez człowieka w sposób naturalny. Zdecydowanie łatwiej przeczytać można poniższą asercję, która zapisana została przy pomocy Fluent Assertions:

[21] https://msdn.microsoft.com/en-us/library/aa287786(v=vs.71).aspx
[22] https://msdn.microsoft.com/en-us/library/aa288454(v=vs.71).aspx

```
receivedResult.Should().Be(expectedResult);
```

co łatwo można przeczytać jako: „rezultat powinien być równy oczekiwanemu”.

NSubstite jest biblioteką wspierającą tworzenie dublerów (mocks) na potrzeby przeprowadzania testów jednostkowych. Dzięki niej bardzo uproszczone jest szybkie przygotowanie testowych instancji interfejsów, od których zależy testowana implementacja. Dostarczanie podstawowego zachowania dublowanego interfejsu sprowadza się do wywołania pojedynczej metody i dostarczenia pożądanej implementacji. Podobnie, bardzo prosto można przeprowadzić nawet złożone weryfikacje ścieżek wykonania kodu.

ANTLR

Kluczowym narzędziem do zaimplementowania mechanizmu zapytań było interpretowanie (parsowanie) zapytań wprowadzanych do aplikacji. Ze względu na fakt rozwijania definiowanego języka zapytań na bieżąco wraz z programem, nie najlepszym rozwiązaniem byłaby autorska implementacja parsera. W związku z tym wykorzystano gotową bibliotekę umożliwiającą generowanie kodu parserów na podstawie dostarczonego kodu gramatyki języka. Wybraną aplikacją był ANTLR – *ANother Tool for Language Recognition*[23].

W podstawowej wersji, ANTLR napisany został w języku Java. Powstała jednak implementacja dla platformy .NET umożliwiająca zintegrowanie aplikacji z biblioteką pozostającą w technologii Java. Dzięki takiemu wsparciu, na podstawie przygotowanej i na bieżąco rozwijanej gramatyki zgodnej ze składnią ANTLR, biblioteka automatycznie generowała kod odpowiedniego parsera w języku C#, gotowy do skompilowania i wykorzystania w aplikacji.

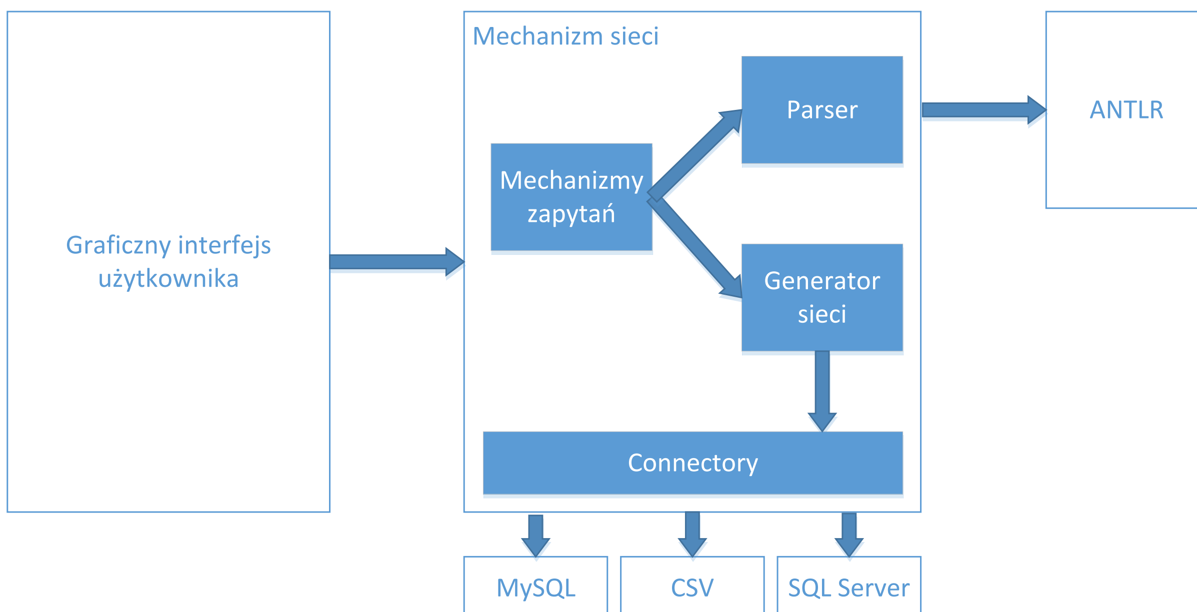
Prócz parsera, ANTLR dostarcza również zestaw implementacji wspierających interpretowanie danych wejściowych do parsera. Dzięki temu, dane wejściowe przetransformowane przez ANTLR do postaci *drzewa wyrażenia*, mogły być łatwo obsłużone za pomocą bazowej, rozszerzalnej wersji wzorca *wizytatora*[24].

Dzięki całkowitej integracji generowania kodu parsera z kompilacją aplikacji, jednoczesne rozwijanie zagadnień sieci i implementacji mechanizmu zapytań było bezproblemowe.

Architektura

Aplikacja została rozdzielona na dwa zasadnicze elementy: interfejs graficzny oraz silnik sieci.

[23] http://www.antlr.org/
[24] https://pl.wikipedia.org/wiki/Odwiedzający



Rysunek 27 Schemat architektury aplikacji

Intefejs użytkownika

W ramach części interfejsu zdefiniowane zostały wszelkie elementy umożliwiające interakcję z programem: widoki, kontrolki, okna itp. Zgodnie z podejściem promowanym przez framework WPF, widoki (i ich logika) i kod implementacji zostały od siebie wyraźnie rozdzielone, dzięki czemu zarządzalność systemu pozostała na wysokim poziomie. Połączenie tych dwóch elementów aplikacji zostało osiągnięte przez realizację wspieranego w pełni przez WPF wzorca *Model-View-View Model* (MVVM), będącego jedną z wielu pochodnych wzorca *Model-View-Controller* (MVC). Dzięki MVVM osiągnięto maksymalną separację kodu, a w ten sposób zapewniono pełną przenośność (na przykład pomiędzy potencjalnie różnymi widokami tego samego modułu) implementacji *widoków-modeli* (view models), będących odpowiednikami *kontrolerów* ze wzorca MVC, a zawierających elementy (lub całość) logiki biznesowej.

Jednym z najbardziej pomocnych mechanizmów WPF w kontekście MVVM jest wbudowane (i bardzo promowane) *wiązanie* (binding) elementów widoku z logiką biznesową w taki sposób, że to widok definiuje zależność od konkretnej logiki biznesowej, podczas gdy implementacja tejże logiki nie ma żadnej wiedzy o widokach ją wykorzystujących. Prócz tego w wielu miejscach wykorzystano wsparcie zmiany formatu danych z logiki biznesowej na potrzeby odpowiedniego wyświetlenia ich na widoku. To wsparcie zostało zrealizowane za pomocą powiązanych z mechanizmem binding *konwerterów*.

Mechanizm sieci

Fundamentem systemu, który wykorzystywany jest przez zbudowany interfejs graficzny, jest część odpowiadająca za zarządzanie siecią. W ramach tej części aplikacji odbywa się komunikacja z dostarczoną bazą danych, odczyt jej schematu i danych, a następnie

utworzenie sieci na podstawie tych danych. Prócz tego to właśnie tu znajduje się kod odpowiedzialny za modyfikację sieci poprzez dodawanie nowych elementów do już utworzonej sieci, jak i usuwanie określonych jej węzłów.

Ta część aplikacji stanowi zarówno pośrednika pomiędzy biblioteką ANTLR do parsowania zapytań, jak i implementuje wizytatora, tłumaczącego drzewo wyrażeń na implementację konkretnych filtrów i pobierania danych z sieci. Owe filtry też zawarte są w części mechanizmu sieci.

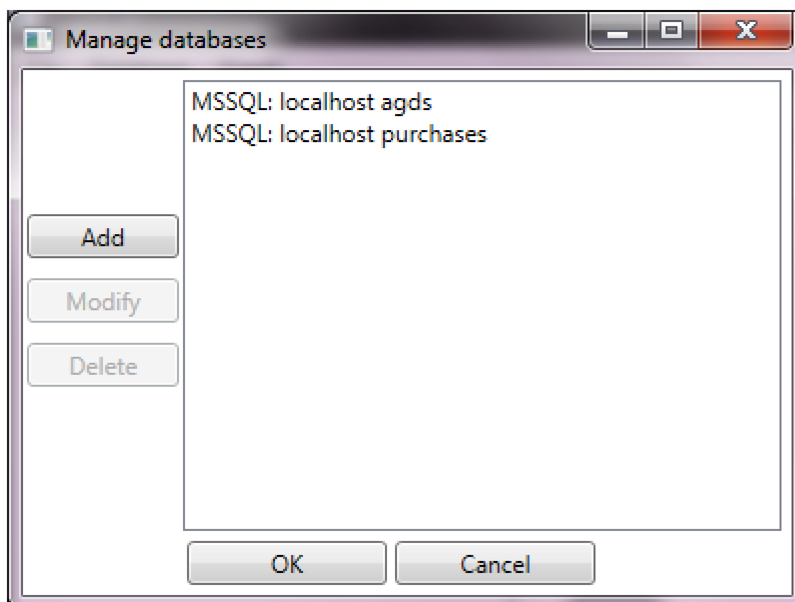
Dzięki takiej separacji, prostym zadaniem byłoby ewentualne wykorzystanie zaimplementowanej części obsługi sieci do stworzenia np. serwera danych z przeznaczeniem dla wielu klientów.

Elementy aplikacji

Poniżej zostanie opisana aplikacja od strony interfejsu użytkownika.

Definicje baz danych

Jako że kluczowym elementem aplikacji jest interakcja z bazami danych, zasadniczą funkcją była możliwość definiowania i zarządzania połączeniami z systemem bazy danych. Do tego celu stworzony został *manager* definicji baz danych. Jest on dostępny z menu aplikacji Databases > Manage databases. Po jego wybraniu mamy do dyspozycji okno z opcjami zarządzania bazami.



Rysunek 28 Manager baz danych

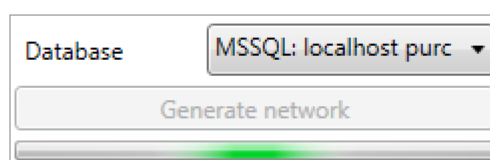
Możliwe jest dodanie nowego, edycja lub usunięcie istniejącego wpisu o bazie danych. Widok dodawania i edycji pojedynczego wpisu umożliwia podanie wymaganych do połączenia danych. Dla bazy danych są to: nazwa hosta komputera z bazą danych, nazwa bazy danych na wskazanym komputerze, użytkownik i hasło. Natomiast dla mechanizmu

odczytu plików CSV jest to po prostu ścieżka do katalogu ze schematem CSV na lokalnym komputerze.

Aplikacja zapisuje wprowadzone dane w pliku binarnym obok głównego pliku wykonywalnego aplikacji. W ten sposób wprowadzone zmiany są zachowywane pomiędzy kolejnymi uruchomieniami.

Generacja sieci

Mając zdefiniowaną przynajmniej jedno źródło danych, można przejść do utworzenia sieci. Do tego celu służy panel umieszczony po lewej stronie okna programu. Najpierw z listy rozwijanej *Databases* należy wybrać konkretną bazę danych zdefiniowaną wcześniej. Następnym krokiem jest kliknięcie w przycisk *Generate network*. Na tym etapie aplikacja generuje sieć, o czym informuje animowanym paskiem dostępu pod przyciskiem.



Rysunek 29 Widok tworzenia sieci

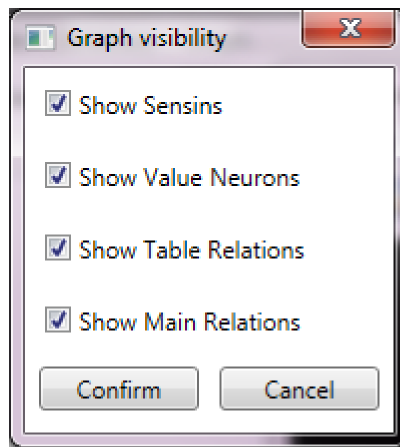
Po ukończeniu procesu generacji sieci, użytkownik jest informowany o statusie: „Network generated” w przypadku powodzenia albo „An error has occured”, jeżeli podczas tworzenia wystąpi błąd.

Wizualizacja sieci

Gdy sieć zostanie utworzona, istnieje możliwość wygenerowania poglądowego schematu sieci (większość rysunków sieci wykorzystanych w tej pracy powstała właśnie dzięki tej funkcji).

Zakładka Graph posiada jeden przycisk, który zostaje uaktywniony po utworzeniu sieci. Jego kliknięcie powoduje narysowanie grafu odpowiadającego sieci skojarzeniowej.

Dla wygody możliwe jest sterowanie widocznością poszczególnych typów węzłów w grafie. Ta opcja jest dostępna z poziomu głównego menu aplikacji (Graph > Adjust visibility). Ukrycie poszczególnych typów węzłów powoduje również ukrycie krawędzi, które są z nimi połączone.

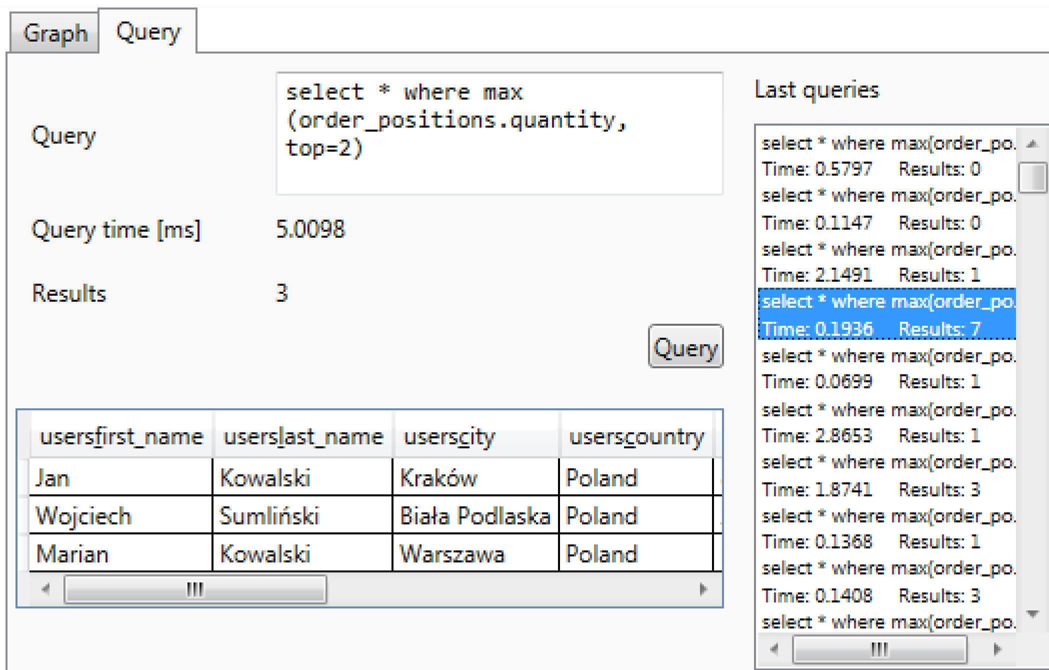


Rysunek 30 Menu zarządzania widocznością grafu

Ważne jest jednak, by nie wywoływać wizualizacji dla sieci dużych rozmiarów, gdyż spowoduje to znaczne spowolnienie programu. Czas generowania wizualizacji może być zdecydowanie wydłużony, a jej końcowy efekt – niezadowalający ze względu na dużą liczbę elementów, a w konsekwencji minimalną czytelność.

Odpytywanie sieci

Aplikacja oferuje dedykowany panel do przeprowadzania zapytań do sieci. Znajduje się on pod zakładką *Query*, obok zakładki wizualizacji grafu.



Rysunek 31 Widok panelu zapytań

Kluczowym elementem panelu zapytań jest pole do wprowadzania treści zapytania. Używając klawiatury, można do niego wprowadzić kod, po czym wcisnąć klawisz *Query*. Można również wykorzystać skrót klawiaturowy *Control+Enter*, będąc w polu zapytania. Wywołanie jednej z tych dwóch komend spowoduje wykonanie zapytania.

Po zakończeniu zapytania, panel przedstawia jego rezultat: informacja o błędzie (wraz ze szczegółami w przypadku podania niepoprawnego zapytania) w wypadku jego wystąpienia lub tabelę z wynikami, gdy proces odpytywania sieci zakończy się sukcesem. Prócz tabeli z wynikami, widoczne są również dodatkowe informacje (Rysunek 31): przybliżony czas wykonania zapytania (w milisekundach) oraz liczba relacji głównych spełniających warunek podany w zapytaniu.

Po prawej stronie panelu zaimplementowana została lista ostatnio przeprowadzonych zapytań *Last queries*. Lista ta skojarzona jest z aktualnie wygenerowaną siecią ze względu na jej strukturę. Oznacza to, że lista przechowuje różne historyczne kwerendy dla każdej sieci. Dane zapisywane są w plikach, dzięki czemu zapytania są przechowywane pomiędzy różnymi uruchomieniami programu. Oznacza to również, że w przypadku zmiany struktury sieci związanej z daną bazą (na przykład poprzez zmianę schematu źródłowej bazy), tworzona jest nowa lista historycznych zapytań. Pojedynczy wpis zawiera treść zapytania jak i czas jego przeprowadzenia (w milisekundach) i liczbę rezultatów zwróconych przez filtr tego zapytania. Dwukrotne kliknięcie w pojedyncze zapytanie powoduje jego ponowne wykonanie. Zaznaczony wpis można usunąć klawiszem *Delete* (alternatywnie można użyć menu kontekstowego dostępnego po wciśnięciu prawego klawisza myszy).

Analiza wydajności

Ten rozdział opisuje efektywność sieci z różnych perspektyw. Przede wszystkim zwracano uwagę na czas wykonywania programu, zarówno w momencie budowania sieci, jak i przeprowadzania zapytań do istniejącej sieci. Drugim istotnym aspektem było zużycie zasobów komputera, na którym uruchamiana była aplikacja.

Testy przeprowadzane były na komputerze o następujących parametrach technicznych:

- procesor CPU: Intel Core i7, 2.8GHz, 8 rdzeni,
- pamięć RAM: 16GB,
- dysk twardy SSD.

Analiza została oparta o bazę danych utworzoną na podstawie pliku CSV. Rozmiar bazy SQL Server wynosił 92MB, zaś po utworzeniu indeksów zwiększył się do 182MB. Baza danych była zlokalizowana na tym samym komputerze, na którym została uruchomiona aplikacja, więc opóźnienia związane z transportem danych zostały zminimalizowane. Baza zawierała pojedynczą tabelę o 7 kolumnach i 731381 wierszach.

Wydajność sieci

Proces generowania sieci skojarzeniowej jest stosunkowo długim i złożonym procesem, gdyż wymaga utworzenia nowej grafowej struktury skojarzeniowej zawierającej odpowiednie powiązania pomiędzy wszystkimi danymi. Poniżej zostały przedstawione czasy dla pięciu prób utworzenia takiej sieci:

- 67 327 ms
- 71 389 ms
- 75 157 ms
- 71 732 ms
- 72 050 ms

Średni czas generacji sieci skojarzeniowej dla w/w tabeli na podstawie powyższych pięciu prób wynosi 71 531 ms.

Po utworzeniu sieci skojarzeniowej, wszystkie jej dane są przechowywane w pamięci. W związku z tym, program w pamięci fizycznej wraz z utworzoną grafową strukturą danych zajmował około 710MB. Przed wygenerowaniem sieci, program potrzebował około 40MB pamięci operacyjnej, więc obiekty związane z wygenerowaną siecią i danymi zajęły około 670MB z dostępnej pamięci. Taka ilość zajętej pamięci mimo pewnej oszczędności wynikającej z usuwania duplikatów danych wynika z faktu, że prócz przechowywania w pamięci samych danych (łańcuchów znaków, liczb itp.) pamięć jest zajmowana przez elementy struktury sieci skojarzeniowej: węzły, połączenia. Dla omawianej sieci

skojarzeniowej utworzono 3654695 węzłów wartości, co w stosunku do ilości danych (7 kolumnach i 731381 wierszach = 5119667) oznacza ok. 71% kompresję na skutek usunięcia duplikatów. Każdy węzeł wartości przechowuje listę wskaźników (połączenia) do skojarzonych z nim relacji tabeli i odwrotnie – każda relacja tabeli zawiera listę wskaźników do powiązanych węzłów wartości. Podobnie, takie połączenia zdefiniowane są pomiędzy warstwą relacji tabeli i relacji głównych.

Biorąc pod uwagę wersję wyjściowej bazy danych, która posiadała indeksy, stosunek zajętego miejsca przez dane w sieci do danych pliku bazy wynosi przeszło 3,5. Stosunek ten zależy od liczby wystąpień duplikatów wartości w oryginalnej bazie danych. Gdyby zastosować bazę danych z większą liczbą zduplikowanych danych (np. bazę w ogóle nie znormalizowaną lub w pierwszej postaci normalnej), iloraz ten byłby mniejszy, co oznaczałoby oszczędność miejsca.

Wydajność wybranych zapytań

W tym podrozdziale zaprezentowane zostały przykładowe zapytania przeprowadzone zarówno na sieci, jak i na odpowiadającej jej bazie. Odczytane również były czasy przeprowadzonych zapytań w obu wersjach.

Pojedyncze porównanie

Podstawowym zadaniem jest wybranie krotek przez porównanie wartości w ramach pojedynczej kolumny.

Sieć skojarzeniowa AGDS	**SQL Server**
`select * where purchases.vendorname = "CITY TREASURER"`	`select * from purchases where vendorname = "CITY TREASURER"`
0,03 ms	34 ms
0,03 ms	48 ms
0,01 ms	36 ms
0,02 ms	22 ms
0,02 ms	50 ms
Średnia = 0,02 ms	**Średnia = 38 ms**

Powyższe zapytanie zwróciło 5 krotek. Czasy odpowiedzi sieci skojarzeniowej AGDS są wyraźnie mniejsze od czasu odpowiedzi serwera SQL. W związku z natychmiastowym dostępem do konkretnej wartości dzięki przechowywaniu ich za pomocą tablic mieszających, gdyż taki filtr ma złożoność obliczeniową O(1).

Różnica ta wynika prawdopodobnie jednak z konieczności transportu danych pomiędzy klientem bazy a serwerem w przypadku SQL Server, podczas gdy sieć jest dostępna

natychmiast w pamięci aplikacji. W związku z tym trudno o rzetelne porównanie obu mechanizmów na podstawie tego prostego testu.

Porównania wykorzystujące operatory logiczne

Język zapytań umożliwia łączenie warunków logicznych za pomocą operatorów logicznych. Poniżej przedstawiono wyniki złączenia trzech warunków operatorem alternatywy OR.

Sieć skojarzeniowa AGDS	**SQL Server**
`select * where purchases.contractnumber = "DV" or purchases.vendorname = "BLUE CROSS & BLUE SHIELD" or purchases.vendorname = "THE BANK OF NEW YORK"`	`select * from purchases where contractnumber = 'DV' or vendorname = 'BLUE CROSS & BLUE SHIELD' or vendorname = 'THE BANK OF NEW YORK'`
340 ms	3609 ms
383 ms	3354 ms
308 ms	3377 ms
411 ms	3636 ms
302 ms	3508 ms
Średnia = 348 ms	**Średnia = 3496 ms**

Zapytanie zwróciło większość rezultatów – 545 527 krotek – przez co zapytanie było zdecydowanie wolniejsze od poprzedniego w obu przypadkach. Widać jednak, że sieć skojarzeniowa odpowiada w czasie wyraźnie krótszym niż serwer SQL, mamy więc poprawę wyniku.

Przeanalizowano również inne zapytanie podobnego typu, w praktyce różniące się od poprzedniego zmienionym pierwszym operatorem OR na AND.

Sieć skojarzeniowa AGDS	**SQL Server**
`select * where purchases.contractnumber = "DV" and purchases.vendorname = "BLUE CROSS & BLUE SHIELD" or purchases.vendorname = "THE BANK OF NEW YORK"`	`select * from purchases where contractnumber = 'DV' and vendorname = 'BLUE CROSS & BLUE SHIELD' or vendorname = 'THE BANK OF NEW YORK'`
406 ms	115 ms
431 ms	92 ms
420 ms	82 ms
321 ms	100 ms
354 ms	66 ms
Średnia = 386 ms	**Średnia = 91 ms**

Tak zmodyfikowane zapytanie zwraca jedynie 16 relacji. Zauważalnie jednak zapytanie tego typu jest wykonywane szybciej przez system SQL Server niż przez sieć AGDS. Powodem, dla którego czas odpowiedzi sieci skojarzeniowej jest tak duży jest fakt, że jedna z operacji aktywujących zawarta w warunku logicznym (tj. `purchases.contractnumber = "DV"`) aktywuje aż 545 465 relacji. Mimo że całe wyrażenie zwraca jedynie 16 relacji po ewaluacji pozostałych operacji aktywujących i warunków, wspomniana operacja jest przyczyną takiego wyniku czasowego.

Jawne pogrupowanie warunków za pomocą nawiasów w taki sposób, aby to dwa ostatnie warunki były wykonywane razem, przedstawione jest poniżej:

Sieć skojarzeniowa AGDS	**SQL Server**
`select * where purchases.contractnumber = "DV" and (purchases.vendorname = "BLUE CROSS & BLUE SHIELD" or purchases.vendorname = "THE BANK OF NEW YORK")`	`select * from purchases where contractnumber = 'DV' and (vendorname = 'BLUE CROSS & BLUE SHIELD' or vendorname = 'THE BANK OF NEW YORK')`
421 ms	78 ms
340 ms	64 ms
390 ms	38 ms
259 ms	142 ms
286 ms	91 ms
Średnia = 339 ms	**Średnia = 82 ms**

Zapytanie po takiej modyfikacji zwraca dokładnie te same rekordy. Niestety, nie poprawia to w znaczący sposób wydajności sieci skojarzeniowej – wyrażenie nadal zawiera tę samą operację aktywującą przeszło pół miliona relacji. Najwyraźniej posługiwanie się wyrażeniami, które po drodze aktywują dużą liczbę węzłów, jest pewnym ograniczeniem mechanizmu odpytywania sieci skojarzeniowej.

Nierówność

Dalej przeanalizowane zostały warunki nierównościowe wskazujące pewne zakresy wartości.

Sieć skojarzeniowa AGDS	SQL Server
`select * where purchases.amount > 50000000`	`select * from purchases where amount > 50000000`
9 ms	76 ms
8 ms	81 ms
4 ms	45 ms
7 ms	132 ms
10 ms	87 ms
Średnia = 7 ms	**Średnia = 84 ms**

Powyższe zapytanie zwraca 143 rekordy. Na podstawie czasów odpowiedzi widać przewagę sieci AGDS w zapytaniu tego typu. Wynika to z metody działania filtru nierównościowego – jego złożoność sprowadza się jedynie do pobudzania kolejnych wartości począwszy od ekstremalnej. Dlatego też w pesymistycznym przypadku konieczności pobudzenia wszystkich wartości złożoność tego filtru to O(n), gdzie n jest liczbą wartości w ramach danego atrybutu. Widać to na poniższym zapytaniu, które jest zmodyfikowaną wersją poprzedniego.

Sieć skojarzeniowa AGDS	SQL Server
`select * where purchases.amount > 5000`	`select * from purchases where amount > 5000`
408 ms	1290 ms
418 ms	1010 ms
432 ms	1032 ms
445 ms	1070 ms
274 ms	1054 ms
Średnia = 395 ms	**Średnia = 1091 ms**

Zmieniona wersja zwraca 149713 rekordów. Zauważalnie wzrósł czas przeprowadzania zapytania w obu strukturach danych.

Warto zwrócić uwagę na zależność czasu odpowiedzi od liczby aktywowanych relacji i stopnia skomplikowania warunku – podczas gdy relacyjne bazy danych są od lat udoskonalane pod kątem wydajności przeprowadzanych zapytań, sieć AGDS nie posiada algorytmów optymalizujących zapytania w żaden sposób, dlatego nie jest „odporna” na wiele rodzajów zapytań, z którymi baza relacyjna radzi sobie dobrze.

Wartości ekstremalne

Przetestowane również były filtry wybierające krotki o wartościach ekstremalnych.

Sieć skojarzeniowa AGDS	SQL Server
`select * where max(purchases.amount)`	`select * from purchases where amount = (select max(amount) from purchases)`
10 ms	61 ms
7 ms	101 ms
7 ms	83 ms
7 ms	60 ms
7 ms	42 ms
Średnia = 8 ms	**Średnia = 69 ms**

Oba mechanizmy radzą sobie dobrze z tym zadaniem. Sprawdzone również zostały czasy wykonania zapytań wybierających krotki o wartości atrybutu będącej N najbliższymi wartościami począwszy od wartości ekstremalnej, czyli wyszukiwanie danych o N najmniejszych lub N największych wartościach danego atrybutu:

Sieć skojarzeniowa AGDS	SQL Server
`select * where max(purchases.amount, top=2)`	`SELECT * FROM  purchases p1 WHERE (2-1) <= ( SELECT COUNT(DISTINCT(p2.amount)) FROM purchases p2 WHERE p2.amount > p1.amount)`
7 ms	ponad 10 min
7 ms	ponad 10 min
9 ms	ponad 10 min
7 ms	ponad 10 min
7 ms	ponad 10 min
Średnia = 7 ms	**Średnia = ponad 10 min**

Powyższe zapytanie ma na celu zwrócenie krotek dla dwóch największych wartości atrybutu. Sieć radzi sobie z tym zadaniem bez problemu, natomiast struktura zapytania możliwego do uruchomienia na SQL Server sprawia, że wykonanie zapytania trwało ponad 10 minut.

Zaprzeczenie

Przeprowadzone zostały warunki zawierające zaprzeczenia. W związku z tym, że operacje negacji były najmniej wydajne, liczba zastosowanych negacji mocno wpływała na wyniki przeprowadzonych zapytań.

Najpierw w prosty sposób sprawdzono wydajność samego zaprzeczenia. Jako że zapytania nie są optymalizowane ze względu na ich strukturę, są one wykonywane dokładnie tak, jak są podane do aplikacji. Najpierw dla porównania wybrano prosty warunek, który można w prosty sposób zaprzeczyć – warunek nierównościowy.

Sieć skojarzeniowa AGDS	**SQL Server**
`select * where purchases.amount > 4000000`	`select * from purchases where purchases.amount > 4000000`
5 ms	129 ms
6 ms	94 ms
5 ms	140 ms
7 ms	93 ms
6 ms	98 ms
Średnia = 6 ms	**Średnia = 111 ms**

To zapytanie zwraca 3065 relacji. Widać, że mimo sporej liczby wyników, sieć AGDS radzi sobie dobrze. Sprawdzone zatem zostało zapytanie równoważne.

Sieć skojarzeniowa AGDS	**SQL Server**
`select * where not (purchases.amount <= 4000000)`	`select * from purchases where not (purchases.amount <= 4000000)`
1354 ms	131 ms
1303 ms	132 ms
1168 ms	91 ms
1082 ms	129 ms
1094 ms	96 ms
Średnia = 1200 ms	**Średnia = 115 ms**

Widać wyraźny spadek wydajności zapytania. Nadal zwraca ono 3065 relacji, ale teraz operacja nierówności aktywuje prawie wszystkie relacje, które przekazuje do zaprzeczenia. Operacja NOT znajduje różnicę pomiędzy wszystkimi relacjami a aktywowanymi przez jej argument i te wynikowe relacje aktywuje. Jest to operacja dużo bardziej złożona czasowo, potencjalnie jednak możliwa do zoptymalizowania, na przykład poprzez wstępną interpretację wyrażenia warunkowego i ewentualne przekształcenie go do prostszej postaci równoważnej.

Następne zapytanie to zapytanie z podrozdziału Porównania wykorzystujące operatory logiczne zmodyfikowany w taki sposób, że warunek (`purchases.contractnumber = "DV"`) aktywujący uprzednio przeszło pół miliona relacji został zaprzeczony.

Sieć skojarzeniowa AGDS	**SQL Server**
`select * where not(purchases.contractnumber = "DV") and purchases.vendorname = "BLUE CROSS & BLUE SHIELD" or purchases.vendorname = "THE BANK OF NEW YORK"`	`select * from purchases where not(contractnumber = 'DV') and vendorname = 'BLUE CROSS & BLUE SHIELD' or vendorname = 'THE BANK OF NEW YORK'`
759 ms	93 ms
622 ms	65 ms
672 ms	101 ms
628 ms	46 ms
653 ms	94 ms
Średnia = 666 ms	**Średnia = 80 ms**

Sprawdzane zapytania zwracają 72 relacje. Widoczny jest spory spadek wydajności w stosunku do oryginalnego zapytania, podobnie jak w poprzednim przypadku użycia NOT.

Podsumowanie

W ramach niniejszej pracy znacznie wzbogaciłem zasób wiadomości związanych z relacyjnymi bazami danych. W oparciu o bazę SQL Server zapoznałem się z niskopoziomowymi szczegółami związanymi z przechowywaniem danych i przeprowadzaniem zapytań. Prócz tego poznałem i wykorzystałem podstawowe elementy sieci skojarzeniowych do zaimplementowania statycznej asocjacyjnej grafowej struktury danych AGDS. [20]

Kluczowym aspektem tej pracy dyplomowej była praktyczna realizacja struktury sieci AGDS oraz implementacja mechanizmów umożliwiających efektywne wydobywanie danych z utworzonej sieci w sposób podobny do wykorzystywanego w przypadku szeroko stosowanych relacyjnych baz danych. W tym celu prócz poznania systemów skojarzeniowych, zgłębiono wiedzę ściśle technologiczną, związaną z utworzonym oprogramowaniem, jak framework Windows Presentation Foundation na potrzeby graficznego interfejsu użytkownika czy ANTLR do parsowania i interpretowania zapytań wpisywanych przez użytkownika.

Zaprezentowane zostały podstawowe testy wydajnościowe zbudowanego mechanizmu. Porównano je do analogicznych kwerend dla relacyjnej bazy danych pod kątem czasu wykonywania.

Zbudowany system zaledwie powierzchownie porusza zagadnienie sieci skojarzeniowych. Stanowi on jednak dobry fundament do dalszego rozwoju i optymalizacji zarówno sieci, jak i zapytań. Warte dalszego rozważenia jest przede wszystkim zoptymalizowanie wykorzystania pamięci przez sieć w trakcie lub po jej wygenerowaniu poprzez np. odpowiedni jej zapis i późniejszy odczyt na dysku twardym. Znacznie zwiększyłoby to skalowalność pionową zaimplementowanego rozwiązania, która w obecnej wersji bazuje wyłącznie na pamięci RAM. Inną istotną kwestią jest optymalizacja już zaimplementowanych operatorów logicznych i warunków filtrowania – w obecnej wersji brak jest optymalizacji przeprowadzanych zapytań.

Ponadto, w obecnej wersji rozbudowa mechanizmu filtrowania o nowe funkcje jest stosunkowo łatwa i nie wymaga dużego nakładu pracy, lecz tylko drobnych zmian w samym mechanizmie parsowania zapytań. Największym wyzwaniem jest zdefiniowanie i implementacja odpowiedniej funkcji filtrującej działającej na strukturze AGDS.

Bibliografia

[1] „SQL,” [Online]. Available: https://en.wikipedia.org/wiki/SQL. [Data uzyskania dostępu: 17 lipiec 2016].

[2] „DB-Engines Ranking,” [Online]. Available: http://db-engines.com/en/ranking. [Data uzyskania dostępu: 17 lipiec 2016].

[3] M. S. Rasmussen, „SQL Server Storage Internals 101,” 9 październik 2013. [Online]. Available: https://www.simple-talk.com/sql/database-administration/sql-server-storage-internals-101/. [Data uzyskania dostępu: 17 lipiec 2016].

[4] „Relational Database Management Systems,” [Online]. Available: http://rdbms.opengrass.net/2_Database%20Design/2.1_TermsOfReference/2.1.2_Keys.html. [Data uzyskania dostępu: 11 wrzesień 2016].

[5] „Data Integrity,” [Online]. Available: https://www.techopedia.com/definition/811/data-integrity-databases. [Data uzyskania dostępu: 17 lipiec 2016].

[6] „Types of Table Relationships,” Microsoft, [Online]. Available: https://technet.microsoft.com/en-us/library/ms190651(v=sql.105).aspx. [Data uzyskania dostępu: 17 lipiec 2016].

[7] „The Search Tree (B-Tree) Makes The Index Fast,” 24 październik 2011. [Online]. Available: http://use-the-index-luke.com/sql/anatomy/the-tree. [Data uzyskania dostępu: 17 lipiec 2016].

[8] S. Combaudon, „Full table scan vs full index scan performance,” 23 listopad 2012. [Online]. Available: https://www.percona.com/blog/2012/11/23/full-table-scan-vs-full-index-scan-performance/. [Data uzyskania dostępu: 17 lipiec 2016].

[9] M. Ufford, „Effective Clustered Indexes,” 6 styczeń 2011. [Online]. Available: https://www.simple-talk.com/sql/learn-sql-server/effective-clustered-indexes/. [Data uzyskania dostępu: 17 lipiec 2016].

[10] „Relational Databases Are Not Designed For Scale,” 9 listopad 2015. [Online]. Available: http://www.marklogic.com/blog/relational-databases-scale/. [Data uzyskania dostępu: 17 lipiec 2016].

[11] „Scalability,” [Online]. Available: https://en.wikipedia.org/wiki/Scalability. [Data uzyskania dostępu: 17 lipiec 2016].

[12] „Opis podstaw normalizacji bazy danych,” [Online]. Available: https://support.microsoft.com/pl-pl/kb/283878. [Data uzyskania dostępu: 24 lipiec 2016].

[13] A. Ligęza, „Bazy danych,” [Online]. Available: http://home.agh.edu.pl/~ligeza/lectures.html. [Data uzyskania dostępu: 24 lipiec 2016].

[14] „MongoDB Documentation,” [Online]. Available: https://docs.mongodb.com/manual/. [Data uzyskania dostępu: 11 wrzesień 2016].

[15] „Introduction to Redis,” [Online]. Available: http://redis.io/topics/introduction. [Data uzyskania dostępu: 11 wrzesień 2016].

[16] „Graph Data Modeling Guidelines,” [Online]. Available: https://neo4j.com/developer/guide-data-modeling/. [Data uzyskania dostępu: 5 wrzesień 2016].

[17] „Cypher query language,” [Online]. Available: https://neo4j.com/docs/developer-manual/current/cypher/. [Data uzyskania dostępu: 5 wrzesień 2016].

[18] G. S. &. P. Rathle, „Fraud Detection: Discovering Connections with Graph Databases,” [Online]. Available: http://info.neo4j.com/rs/neotechnology/images/Fraud%20Detection%20Using%20GraphDB%20-%202014.pdf. [Data uzyskania dostępu: 5 wrzesień 2016].

[19] L. E. Rasmussen, „Under the Hood: Building Graph Search Beta,” [Online]. [Data uzyskania dostępu: 17 lipiec 2016].

[20] A. Horzyk, Sztuczne systemy skojarzeniowe i asocjacyjna sztuczna inteligencja, Warszawa: Akademicka Oficyna Wydawnicza EXIT, 2013.

[21] A. Horzyk, „Asocjacyjne grafowe struktury danych AGDS,” [Online]. Available: http://home.agh.edu.pl/~horzyk/lectures/aas/SSS-AGDS.pdf. [Data uzyskania dostępu: 28 sierpień 2016].

[22] „Tablice z haszowaniem,” [Online]. Available: http://edu.pjwstk.edu.pl/wyklady/asd/scb/asd10/main10_p4.html. [Data uzyskania dostępu: 6 wrzesień 2016].

[23] „What is XAML?,” [Online]. Available: https://msdn.microsoft.com/en-us/library/cc295302.aspx. [Data uzyskania dostępu: 3 sierpień 2016].

[24] „Windows Presentation Foundation,” [Online]. Available: https://en.wikipedia.org/wiki/Windows_Presentation_Foundation. [Data uzyskania dostępu: 3 sierpień 2016].

[25] T. Parr, The Definitive ANTLR 4 Reference, The Pragmatic Programmers, LLC, 2012.

[26] A. Saltarello i D. Esposito, Architecting Applications for the Enterprise, Microsoft Press, 2014.

[27] A. Horzyk, *Innovative Types and Abilities of Neural Networks Based on Associative Mechanisms and a New Associative Model of Neurons - referat na zaproszenie na międzynarodowej konferencji ICAISC, pp. 26-38,* Springer Verlag, 2015.

[28] A. Horzyk, Human-Like Knowledge Engineering, Generalization and Creativity in Artificial Neural Associative Systems, pp. 39-51, Springer Verlag, 2016.

[29] A. Horzyk, *How Does Generalization and Creativity Come into Being in Neural Associative Systems and How Does It Form Human-Like Knowledge?, pp. 238-257,* Elsevier, 2014.

[30] A. Horzyk, Information Freedom and Associative Artificial Intelligence, pp. 81-89, Springer Verlag, 2012.

Dodatek A – definicje wybranych typów używanych w aplikacji

Dodatek ten zawiera opis pewnych grup typów implementowanych w aplikacji, ważnych z punktu widzenia rozszerzania aplikacji lub innych aspektów.

Interfejs bazy danych

Poniżej został przedstawiony kompletny interfejs sterownika wykorzystywanego do ustanowienia połączenia i odczytu danych z bazy.

Table

Typ reprezentujący pojedynczą tabelę bazy danych. Zawiera pojedynczą właściwość:

- `Name: string` – nazwa tabeli z bazy

Column

Odpowiednik pojedynczej kolumny zawartej w tabeli bazy danych. Ma następujące właściwości:

- `Name: string` – nazwa kolumny w tabeli
- `Table: Table` – tabela, do której należy kolumna
- `Type: Type` (.NET typ) – typ danych .NET reprezentowany przez tę kolumnę

PrimaryKey

Zawiera informacje o kluczu głównym zdefiniowanym w ramach tabeli. Ten typ również ma jedną właściwość:

- `Columns: IEnumerable<Column>` - kolekcja kolumn tabeli tworzących klucz główny

ForeignKey

Przechowuje dane związane z kluczem obcym występującym w tabeli. Te dane to:

- `Column: Column` – kolumna tabeli-dziecka, będąca kluczem głównym
- `ParentTable: Table` – tabela-rodzic, do której odnosi się relacja określona tym kluczem obcym
- `ReferencedColumn: Column` – kolumna będąca kluczem głównym tabeli ParentTable, do której odnosi się kolumna klucza obcego

Row

Ten typ odpowiada pojedynczej krotce w tabeli bazy danych. Zawiera *indekser* przyjmujący kolumnę jako parametr, a zwracający wartość w krotce, która odpowiada podanej kolumnie.

IKeysProvider

Interfejs ten jest podinterfejsem wydzielonym z niżej opisanego `IDatabaseConnector`. Służy do pobierania informacji o kluczach (głównym i obcym) zawartych w bazie. Zawiera następujące metody:

- `PrimaryKey GetPrimaryKey(Table table)`
  zwraca obiekt `PrimaryKey` dla tabeli, będący zestawem informacji o kluczu głównym podanej tabeli `table`.
- `ForeignKey[] GetForeignKeys(Table table)`
  zwraca tablicę wszystkich kluczy obcych, które zdefiniowane są w tabeli `table`.

IDatabaseConnector

Interfejs `IDatabaseConnector` rozszerza powyższy interfejs do zarządzania informacjami o kluczach. Zaimplementowanie tego interfejsu zapewnia całkowitą obsługę nowego typu bazy danych wewnątrz aplikacji. Prócz metod z interfejsu `IKeysProvider`, definiuje wiele własnych, opisanych poniżej:

- `Table[] GetTables()`
  zwraca tablicę wszystkich tabel zdefiniowanych w bazie danych
- `Column[] GetColumns(Table table)`
  dla podanej tabeli `table` zwraca tablicę wszystkich kolumn w tej tabeli
- `IEnumerable<Row> GetRows(Table table)`
  dla podanej tabeli zwraca sekwencję wszystkich wierszy w tej tabeli, gotową do iteracji przez algorytm.