

Opracowanie do egzaminu		
ZTB	29 I 2020	C2 s.224 10:00

Spis treści

1	Lista tematów	3
2	CouchDB	3
2.1	Przechowywanie danych	3
2.2	Komunikacja z bazą danych	4
2.3	Futon	4
2.4	Tworzenie zapytań - MapReduce	5
2.4.1	Funkcja <i>map</i>	5
2.4.2	Funkcje <i>reduce</i>	7
2.5	Design document	11
2.6	Zadania z poprzednich lat	12
3	PostGIS	14
3.1	Typy przestrzenne w PostGIS	14
3.1.1	Hierarchiczna budowa typów w PostGIS	14
3.2	<i>Geometry</i> i podtypy	15
3.2.1	<i>POINT</i>	15
3.2.2	<i>LINESTRING</i>	16
3.2.3	<i>POLYGON</i>	17
3.3	SRID	18
3.4	Typ <i>Geography</i> i podtypy	19
3.4.1	Różnica pomiędzy <i>geometry</i> i <i>geography</i>	20
3.5	GeoJSON	24
3.6	Funkcje PostGIS	25
3.6.1	<i>ST_GeomFromText</i>	25
3.6.2	<i>ST_GeogFromText</i>	25
3.6.3	<i>ST_Transform</i>	26
3.6.4	<i>ST_GeomFromGeoJSON</i>	26
3.6.5	<i>ST_Centroid</i>	26
3.6.6	<i>ST_X</i> i <i>ST_Y</i>	27
3.6.7	<i>ST_Area</i>	27
3.6.8	<i>ST_Length</i> i <i>ST_Perimeter</i>	27
3.6.9	<i>ST_Covers</i>	28
3.6.10	<i>ST_Distance</i>	28
3.7	Zadania z poprzednich lat	30
4	ACID	34
4.1	Zadania z poprzednich lat	34
5	Indeksy	35
5.1	Ogólny proces wysyłania zapytani SELECT	35
5.2	ANALYZE i EXPLAIN	35
5.3	Tworzenie i usuwanie indeksu	36

5.4	Typy indeksów	36
5.4.1	Btree	36
5.4.2	Hash	36
5.4.3	Inne	37
5.5	Indeksy wielokolumnowe	37
5.6	Indeksy i sortowanie	37
5.7	Indeksy częściowe i indeksy na wyrażeniach	37
5.8	Zadania z poprzednich lat	38
6	XML	39
6.1	Tworzenie xml na podstawie tabeli	39
6.2	Tworzenie zapytań do typu xml	39
7	Neo4j i Cypher	41
7.1	Informacje ogólne	41
7.2	Podstawowe pojęcia neo4j	41
7.3	Konwencje nazewnictwa w Neo4j	42
7.4	Cypher - Podstawy	42
7.4.1	Węzły	42
7.4.2	Relacje	42
7.4.3	Węzły + Relacje = Wzorce	43
7.5	Cypher - CREATE, MATCH, MERGE	43
7.5.1	CREATE	43
7.5.2	MATCH	44
7.5.3	MERGE	45
7.6	Cypher - WHERE i funkcje agregujące	46
7.6.1	WHERE	46
7.6.2	Funkcje agregujące	47
8	Inne	48

1 Lista tematów

1. CouchDB
2. PostGIS
3. Indeksy
4. Postresql i xml
5. ACID
6. Neo4J i Cypher

2 CouchDB

Couch DB to rozwijany przez Fundację Apache wysokowydajny, nierelacyjny (NoSQL) silnik baz danych napisany w języku Erlang zorientowany na dokumenty. Udostępniony został na zasadach licencji Apache License 2.0. Dostęp do systemu realizuje się za pomocą API korzystającego z mechanizmu REST.

Alternatywami dla systemu CouchDB są projekty Cassandra i MongoDB

2.1. Przechowywanie danych

W CouchDB serwer hostuje bazy danych, które zawierają w sobie dokumenty. Dokument jest podstawową jednostką danych w CouchDB. Składa się z dowolnej liczby pól i załączników. Pola to po prostu atrybuty, zapisywane tak jak w pliku JSON (klucz - wartość). O załącznikach możemy myśleć tak jak o załącznikach do email, są to po protu pliki, które dodajemy do dokumentu (np. image, video), z nich jednak nie będziemy raczej korzystać na egzaminie, ale dla kompletności o nich tutaj piszę. Każdy dokument ma nadawane przez bazę unikalne id (o ile nie określmy go sami). Struktura dokumentu odpowiada strukturze pliku typu JSON (klucz - wartość).

Przykładowy dokument:

```
1 {
2   "_id" : "bc2a41170621c326ec68382f846d5764",
3   "_rev" : "2612672603",
4   "item" : "apple",
5   "prices" : {
6     "Fresh Mart" : 1.59,
7     "Price Max" : 5.99,
8     "Apples Express" : 0.79
9   }
10 }
```

Pole id to unikalne id nadane przez bazę, pole _ rev służy do informowania czy dany dokument jest aktualny, omówimy go dokładniej później.

Dokumenty możemy dodawać do bazy korzystając z HTTP i gotowych plików JSON.

2.2. Komunikacja z bazą danych

Z bazą danych CouchDB komunikujemy się poprzez wysyłanie zapytań HTTP. CouchDB zapewnia obsługę API typu REST, dzięki czemu możemy wykonywać typowe zapytania typu GET, POST, PUT, DELETE.

Do wysyłania zapytań najczęściej korzystamy z jakiegoś oprogramowania, najbardziej powszechnym w użyciu wydaje się tutaj CURL, ale możemy korzystać również z dowolnej innej alternatywy.

Listing 1: Testowanie czy CouchDB działa

```
curl http://nosql.kis.agh.edu.pl:5984/
```

```
1 {"couchdb": "Welcome", "version": "0.10.1"}
```

Listing 2: Tworzenie nowej bazy o nazwie dwrobel

```
curl -X PUT http://nosql.kis.agh.edu.pl:5984/dwrobel
```

```
1 {"ok": true}
```

2.3. Futon

Oczywiście wysyłanie zapytań do bazy danych jest bardzo ważne gdy komunikujemy się z bazą z poziomu naszego programu, jednak do celów administracyjnych mamy zapewniony interfejs graficzny dla CouchDB - Futon. Futon to interfejs, który jest wyświetlany w naszej przeglądarce:

Listing 3: Przejście do panelu administratora w przeglądarce

```
http://nosql.kis.agh.edu.pl:5984/_utils/
```

Name	Size	Number of Documents	Update Seq
_replicator	4.1 KB	1	1
_users	4.1 KB	1	1
auzar	56.1 KB	14	14
bgrodek	84.1 KB	16	21
dbart	72.1 KB	8	22
dbazan	44.1 KB	13	19
ddudek	92.1 KB	7	23
dfalana	192.1 KB	14	48
dmynarski	92.1 KB	8	28
dryll	96.1 KB	8	24
dwrobel	140.1 KB	8	35
falana	79 bytes	0	0
ikasprzyk	68.1 KB	8	17
jblak	172.1 KB	8	43
jbogunia	40.1 KB	10	10
jkacorzyk	208.1 KB	9	52
jwiecezorek	24.1 KB	6	6
klesniak	156.1 KB	7	39
kmarchewka	48.1 KB	8	12
kscczrybak	212.1 KB	7	67
mbabrajarsma	84.1 KB	9	21
mtekston	96.1 KB	8	24
mmielus	84.1 KB	21	21

Rysunek 1: Interfejs graficzny do CouchDB - Futon

Przez ten interfejs możemy również tworzyć nowe bazy, dodawać dokumenty, ich pola, itd.

2.4. Tworzenie zapytań - MapReduce

W tradycyjnej bazie danych SQL mamy możliwość odpytywania bazy danych przez tworzenie zapytań typu SELECT, stawiając odpowiednie warunki otrzymujemy określone dane. Jako, że CouchDB jest bazą NoSQL takie podejście nie jest możliwe. Alternatywą dla zapytań w CouchDB jest MapReduce.

MapReduce to styl odpytywania bazy danych, który opiera się na dwóch funkcjach: *map* i *reduce*. Kombinacja funkcji *map* i *reduce* nazywana jest *view* w terminologii CouchDB. Widoki mogą być tymczasowe lub zapisane na stałe. Zapisane na stałe przechowywane są w specjalnym typie dokumentów - *design document*. Podczas rozwoju oprogramowania możemy korzystać z widoków tymczasowych, ale mogą być one o wiele wolniejsze dlatego finalnie należy przenieść widoki do *design documents*. Najpierw omówimy sobie tworzenie *temporary views*, a następnie ich zapisywanie w *design documents*.

Pamiętaj

map i *reduce* to funkcje JavaScript, które są definiowane w *view*.

2.4.1. Funkcja *map*

Funkcja map jest wywoływana jeden raz dla każdego dokumentu. Funkcja może ominąć cały dokument lub wyemitować jedną lub więcej parę klucz-wartość. Funkcje map nie mogą polegać na żadnych informacjach, które znajdują się poza dokumentem. Dzięki temu mogą być wykonywane równolegle.

Pamiętaj

Funkcja map służy do mapowania dokumentu z jego oryginalnej struktury na parę klucz-wartość.

Przykład

Mamy dany informacje o cenach jabłek, bananów i pomarańczy w naszej bazie danych:

```
1 {  
2   "_id" : "bc2a41170621c326ec68382f846d5764",  
3   "_rev" : "2612672603",  
4   "item" : "apple",  
5   "prices" : {  
6     "Fresh Mart" : 1.59,  
7     "Price Max" : 5.99,  
8     "Apples Express" : 0.79  
9   }  
10 }
```

```
1 {  
2   "_id" : "bc2a41170621c326ec68382f846d5764",  
3   "_rev" : "2612672603",  
4   "item" : "orange",  
5   "prices" : {  
6     "Fresh Mart" : 1.99,  
7     "Price Max" : 3.19,  
8     "Citrus Circus" : 1.09  
9   }
```

```

10 }
11
12 {
13     "_id" : "bc2a41170621c326ec68382f846d5764",
14     "_rev" : "2612672603",
15     "item" : "banana",
16     "prices" : {
17         "Fresh Mart" : 1.99,
18         "Price Max" : 0.79,
19         "Banana Montana" : 4.22
20     }
21 }
```

Funkcję *map* tworzymy wybierając z interfejsu graficznego Futon *temporary view*:

Listing 4: Funkcja map emitująca pary klucz:cena - wartość: owoc sklep

```

1 function(doc) {
2     var store , price , value ;
3     if (doc.item && doc.prices) {
4         for (store in doc.prices) {
5             price = doc.prices[store];
6             value = [doc.item , store];
7             emit(price , value);
8         }
9     }
10 }
```

a następnie klikamy *Run*, funkcja map zostanie wykonana na każdym z dokumentów. W trzeciej linijce sprawdzamy czy dokument zawiera pola item oraz prices. W 7 linijce emitujemy parę klucz - wartość.

Key	Value
0.79 ID: 07909e1c065830be62e30169510f3e7b	["apple", "Apples Express"]
0.79 ID: 07909e1c065830be62e30169510f5048	["banana", "Price Max"]
1.09 ID: 07909e1c065830be62e30169510f3c9c	["orange", "Citrus Circus"]
1.59 ID: 07909e1c065830be62e30169510f3e7b	["apple", "Fresh Mart"]
1.99 ID: 07909e1c065830be62e30169510f3c9c	["orange", "Fresh Mart"]
1.99 ID: 07909e1c065830be62e30169510f5048	["banana", "Fresh Mart"]
3.19 ID: 07909e1c065830be62e30169510f3c9c	["orange", "Price Max"]
4.22 ID: 07909e1c065830be62e30169510f5048	["banana", "Banana Montana"]
5.99 ID: 07909e1c065830be62e30169510f3e7b	["apple", "Price Max"]

Showing 1-9 of 9 rows

View request duration: 00:00:00.068

Rysunek 2: Wynik działania *map*

Pamiętaj

Argumentem funkcji *map* jest obiekt JSON reprezentujący pojedynczy dokument.

Pamiętaj

Funkcja *emit* służy do emitowania par klucz-wartość, jej pierwszy argument to klucz, a drugi to wartość.

Oczywiście funkcję *map* możemy też wywołać poprzez wysłanie HTTP, wtedy wysyłamy JSON'a z polem o nazwie *map*, które ma wartość ustawioną na ciało naszej funkcji.

Przykładowo wysyłając naszą funkcję map przez żądanie POST otrzymamy:

```

1 {"total_rows":9,"offset":0,"rows":[
2 {"id":"07909e1c065830be62e30169510f3e7b","key":0.79000000000000003553,"value":["apple","Apples Express"]},
3 {"id":"07909e1c065830be62e30169510f5048","key":0.79000000000000003553,"value":["banana","Price Max"]},
4 {"id":"07909e1c065830be62e30169510f3c9c","key":1.09000000000000000799,"value":["orange","Citrus Circus"]},
5 {"id":"07909e1c065830be62e30169510f3e7b","key":1.59000000000000000799,"value":["apple","Fresh Mart"]},
6 {"id":"07909e1c065830be62e30169510f3e7b","key":1.98999999999999999911,"value":["orange","Fresh Mart"]},
7 {"id":"07909e1c065830be62e30169510f5048","key":1.98999999999999999911,"value":["banana","Fresh Mart"]},
8 {"id":"07909e1c065830be62e30169510f3c9c","key":3.1899999999999999467,"value":["orange","Price Max"]},
9 {"id":"07909e1c065830be62e30169510f5048","key":4.219999999999997513,"value":["banana","Banana Montana"]},
10 {"id":"07909e1c065830be62e30169510f3e7b","key":5.990000000000002132,"value":["apple","Price Max"]}
11 ]}
12 }
```

Rysunek 3: Wynik zapytania z funkcją *map*

Zauważmy, że id jest zawsze emitowane, nawet jeśli tego nie zrobimy jawnie w kodzie.

2.4.2. Funkcje *reduce*

Funkcja *map* produkuje wiersze z których każdy zawiera parę *klucz-wartość*, te pary mogą zostać opcjonalnie zredukowane do pojedynczej wartości lub do grup wartości, właśnie do tego celu służy nam funkcja *reduce*.

Pamiętaj

Krok (funkcja) *reduce* jest opcjonalna i służy do:

- redukcji wszystkich wartości do jednej wartości lub
- redukcji do wartości pogrupowanych przez klucze

CouchDB ma trzy wbudowane funkcje typu *reduce*, są to: *_sum*, *_count*, *_stats*. Można również pisać swoje własne funkcje reduce, ale nie ma często takiej potrzeby.

- *_sum* - zwraca sumę wartości (tylko dla liczb)
- *_count* - zwraca liczbę wartości w zbiorze, zlicza również wartości null
- *_stats* - zwraca wartości statystyczne dla wartości numerycznych: *sum*, *count*, *min*, *max*.

Przykład

Mamy dane dokumenty z książkami:

```

1 {
2   "_id": "978-0-596-52926-0",
3   "title": "RESTful Web Services",
```

```

4   "subtitle": "Web services for the real world",
5   "authors": [
6     "Leonard Richardson",
7     "Sam Ruby"
8   ],
9   "publisher": "O'Reilly Media",
10  "released": "2007-05-08",
11  "pages": 448
12 }
```

```

1 {
2   "_id": "07909e1c065830be62e30169510f51a8",
3   "_rev": "1-ca7252e7a17b3b8eff123ca80ef5e974",
4   "title": "My Cool Book",
5   "subtitle": "Computer Science",
6   "authors": [
7     "Leo Book",
8     "Ruby Rail"
9   ],
10  "publisher": "Helion",
11  "released": "20010-05-08",
12  "pages": 345
13 }
```

```

1 {
2   "_id": "07909e1c065830be62e30169510f5835",
3   "_rev": "1-1e05d2f9cadad03055d2b6569911aa5",
4   "title": "Java World",
5   "subtitle": "Java Language",
6   "authors": [
7     "Chris Java",
8     "Rob Compiler"
9   ],
10  "publisher": "Javadoc",
11  "released": "20015-05-08",
12  "pages": 231
13 }
```

Wyślemy teraz zapytanie, które najpierw mapuje książki tak, że klucz jest formatem:

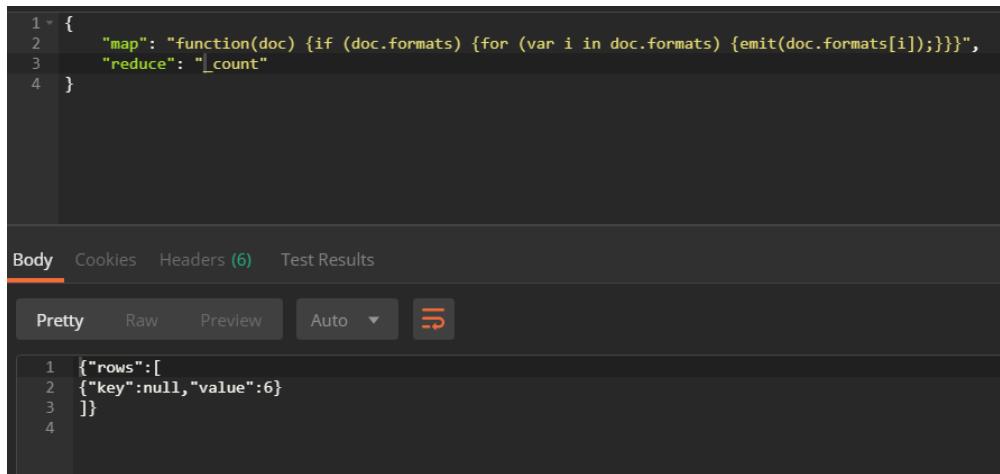
```

1 function(doc) {
2   if (doc.formats)
3   {
4     for (var i in doc.formats) {
5       emit(doc.formats[i]);
6     }
7   }
8 }
```

```
{"total_rows":6,"offset":0,"rows":[
{"id":"07909e1c065830be62e30169510f5161","key":"Ebook","value":null},
 {"id":"07909e1c065830be62e30169510f51a8","key":"Ebook","value":null},
 {"id":"07909e1c065830be62e30169510f5835","key":"Ebook","value":null},
 {"id":"07909e1c065830be62e30169510f51a8","key":"Print","value":null},
 {"id":"07909e1c065830be62e30169510f5835","key":"Print","value":null},
 {"id":"07909e1c065830be62e30169510f51a8","key":"Safari Books Online","value":null}
]})
```

Rysunek 4: Wynik zapytania z funkcją *map*

a następnie dodamy funkcję *reduce*, która zliczy te wartości:



```
1 + {
2   "map": "function(doc) {if (doc.formats) {for (var i in doc.formats) {emit(doc.formats[i]);}}}",
3   "reduce": "_count"
4 }
```

Body Cookies Headers (6) Test Results

Pretty Raw Preview Auto 

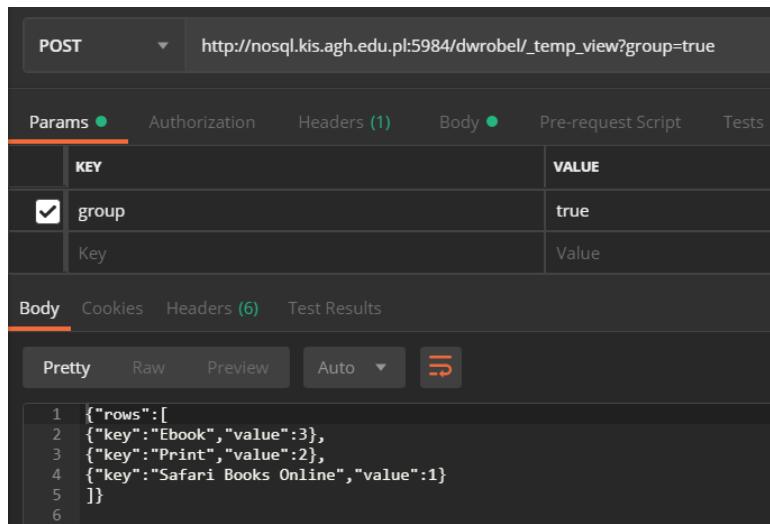
```
1 {"rows": [
2 {"key":null,"value":6}
3 ]}
4 
```

Rysunek 5: Wynik *map* i *reduce*

Jako, że w funkcja *map* zwróciła wszystkie wartości jako *null*, a funkcja *_count* zlicza również null otrzymaliśmy 6 jako wynik. Funkcja *_count* zliczyła wszystkie 'wiersze'.

A co jeśli chcielibyśmy wiedzieć ile książek jest w danym formacie ?

Musimy wtedy zmodyfikować nasze zapytanie, ale nie przez modyfikację ciała JSON'a, ale przez dodanie parametru *group* i ustawienie go na true:



Rysunek 6: Dodanie parametru *group* do zapytania.

Widzimy, że tym razem otrzymujemy redukcję do grup określonych przez klucze.

Analogicznie możemy korzystać z funkcji `_stats` oraz `_sum`, przykładowo dla `_stats`:

```

1  {
2    "map": "function(doc) {if (doc.formats) {for (var i in doc.formats) {emit(doc.formats[i], doc.pages);}}}",
3    "reduce": "_stats"
4  }

```

Status: 200 OK

Body	Cookies	Headers (6)	Test Results
<pre> 1 {"rows":[2 {"key":"Ebook", "value":{"sum":1024, "count":3, "min":231, "max":448, "sumsqr":373090}}, 3 {"key":"Print", "value":{"sum":576, "count":2, "min":231, "max":345, "sumsqr":172386}}, 4 {"key":"Safari Books Online", "value":{"sum":345, "count":1, "min":345, "max":345, "sumsqr":119025}} 5]} 6 </pre>			

Rysunek 7: Przykład dla `_stats`

Możemy również pisać nasze własne funkcje `reduce`, ale tam gdzie to tylko możliwe powinniśmy korzystać z funkcji wbudowanych. Szkielet naszej własnej funkcji `reduce` wygląda następująco

```

1 function(keys, values, rereduce) {
2
3 }

```

- `keys` - tablica kluczy i dokumentów id, każdy z elementów tej tablicy jest w formie $[key, id]$, gdzie id to id dokumentu

- values - to tablica zawierająca zmapowane wartości
- rereduce - określa czy funkcja została wywołana poraz kolejny po tym jak dla wielkiego zbioru danych wywołano kilka funkcji reduce (to nam niewiele mówi, wyjaśnienia do tego pod spodem)

rereduce jest ustawiane na *true* lub *false*. Chodzi o to, że funkcja *map* może wyprodukować ogromną liczbę danych, wtedy optymalniej jest takie dane przetworzyć przez wiele funkcji *reduce* równolegle, a na koniec skleić w całość i tak też zachowuje się CouchDB. Jeśli *rereduce* jest ustawione na *true* to znaczy, że mamy do czynienia z wywołaniem w którym musimy skleić wszystkie częściowe wartości, wtedy tablica *keys* ustawiona jest na null, a tablica *values* zawiera częściowe wyniki wywołań *reduce*, które działały na częściowych danych.

Dla zobrazowania tego zobaczymy jak będzie wyglądać implementacja funkcji *_count*:

```

1 function(keys , values , rereduce) {
2   if (rereduce) {
3     return sum(values);
4   } else {
5     return values.length;
6   }
7 }
```

Czyli jeśli jesteśmy już w kroku łączenia wyników częściowych *reduce*, to sumujemy wszystkie wyniki, a jeśli jesteśmy w częściowym *reduce*, to zwracamy długość tablicy *values*.

Oczywiście funkcja *reduce* z parametrem *rereduce* ustawionym na *true* nie zawsze jest wywoływana, dzieje się to tylko gdy *map* wyprodukuje bardzo dużo danych.

2.5. Design document

Wiemy już jak korzystać w naszej bazie z *temporary views*, teraz chcielibyśmy móc zapisać nasze *views* w bazie tak aby były reużywalne. Do tego służą nam *design documents*, czyli specjalny typ dokumentu przeznaczony na przetrzymywanie *views*, dokument taki może przechowywać wiele *views*.

Design documents muszą mieć id, które będzie się zaczynało na *_design/nazwa*, przykładowo:

"07909e1c065830be62e30169510f51a8" ID: 07909e1c065830be62e30169510f51a8	{rev: "2-e83b0f34a564edad05f9d773d23ce7df")
"07909e1c065830be62e30169510f5835" ID: 07909e1c065830be62e30169510f5835	{rev: "2-7523e3b9922074a2711a72a9f6336638")
"_design/accouncement" ID: _design/accouncement	{rev: "1-7ca6ae01cdcd9f1609fed9443ebaef9")
"_design/letters" ID: _design/letters	{rev: "28-a3e26eb858901272a81e670a3b0d127c")

Rysunek 8: Design documents i zwykłe dokumenty w bazie CouchDB

Design documents tworzymy tak samo jak temporary views, z tą różnicą, że teraz je zapisujemy. Możemy to zrobić przez interfejs graficzny lub przez HTTP, w wyniku otrzymujemy dokument, który zawiera tablicę *views*, przykładowo:

Field	Value
_id	"_design/letters"
_rev	"28-a3e26eb858901272a81e670a3b0d127c"
language	"javascript"
views	<pre> countLetters map "function(doc){var contentLetters = ''; for(var key in doc) { if(key != '_id' & key != '_rev') { contentLetters += doc[key] } } var letters = contentLetters.split(''); reduce '_sum'"</pre>

Showing revision 28 of 28

Rysunek 9: Design document

Następnie możemy wyświetlić wynik wybranego przez nas *view* z tego dokumentu przez zapytanie:

Listing 5: Testowanie design documents

```
curl http://nosql.kis.agh.edu.pl:5984/dwrobel/_design/letters/_view/countLetters
```

2.6. Zadania z poprzednich lat

Egzamin

1. (6 pkt.) W bazie CouchDB znajdują się dokumenty opisujące serwisowanie drukarek:

```
{ "_id": "pr01", "typ": "laser", "wydruki": 3211, "data": "2018-01-01", "uszkodzona": false }
```

Napisz odpowiednie funkcje w JavaScript implementujące widok, który zwróci średnią wartość atrybutu **wydruki** dla drukarek, które nie są uszkodzone (wartość logiczna atrybutu **uszkodzona: false**), dla po poszczególnych typów.

```

1 // map
2 function(doc){
3   if (!doc.uszkodzona && doc.wydruki){
4     emit(doc.typ , doc.wydruki)
5   }
6 }
7
8 // reduce
9 function(keys , values , rereduce) {
10
11   // sytuacja gdy jedna funkcja przetwarza WSZYSTKIE dane
12   if (!rereduce){
13
14     valuesLength = values.length;
15     valuesAvg = sum(values) / valuesLength;
16     return [valuesAvg , valuesLength];
17   }
18
19   // sytuacja gdy kilka funkcji rownolegle przetworzylo czesci danych
20   if(rereduce) {
21     var allLength = sum(values.map(function(v){ return v[1]; }));
22     var allAvg = sum(values.map(function(v){ return v[0]*(v[1]/allLength); }));
23     return [allAvg , allLength];
24   }
25 }
```

²⁷ }

3 PostGIS

PostGIS to rozszerzenie geograficzne bazy PostgreSQL, które daje możliwość zapisywania danych geograficznych wprost do bazy danych.

3.1. Typy przestrzenne w PostGIS

Rozszerzenie PostGIS wprowadza do naszej bazy możliwość używania nowych typów, które dedykowane są do przechowywania danych przestrzennych. Są cztery główne typy danych:

- *Geometry* - najbardziej popularny model, opiera się na współrzędnych kartezjańskich, jest podstawą dla innych modeli
- *Geography* - typ sferyczny, linie i wielokąty są rysowane na zakrzywionej powierzchni, co odzwierciedla rzeczywisty kształt ziemi

O dwóch ostatnich typach tylko sobie wspomnimy, nie są one tak powszechnie jak dwa pierwsze:

- *Raster* - siatka prostokątnych komórek
- *Topology* - modeluje świat jako sieć powiązanych węzłów, krawędzi i powierzchni

Wszystkie te typy możemy oczywiście przechowywać razem w bazie danych, możemy je także przechowywać w różnych kolumnach tej samej tabeli.

Pamiętaj

Dwa najczęściej używane w PostGIS typy danych to *geometry* i *geography*. *geometry* modeluje dane w dwóch wymiarach według współrzędnych kartezjańskich (wada: płaska ziemia, zaleta: szybkość obliczeń). *geography* modeluje dane w trzech wymiarach według współrzędnych sferycznych (wada: wolne obliczenia, zaleta: dokładność reprezentacji)

UWAGA: Sam PostgreSQL (bez PostGIS) również miał od początku wbudowane typy przestrzenne: *point*, *polygon*, *lseg*, *box*, *circle*, *i path*. Nie są one jednak praktyczne, ponieważ praktycznie brak jest dla nich wsparcia w narzędziach do wizualizacji danych, zostały więc prawie całkowicie wyparte przez typy PostGIS.

3.1.1. Hierarchiczna budowa typów w PostGIS

W PostGIS typy przestrzenne mają swoje podtypy, możemy o tym myśleć tak jak o dziedziczeniu w programowaniu, do klasy bazowej możemy przypisywać dowolny z jej podtypów, analogicznie jest w PostGIS.

Typy *geometry* i *geography* mają swoje podtypy. Jeśli w bazie danych mamy kolumnę typu *geometry*, to możemy do niej wstawić dane o dowolnym z podtypów dla *geometry*, analogicznie dla *geography*. Możemy także utworzyć kolumnę, która stricte określa typ jako jeden z podtypów *geometry*.

Pamiętaj

W PostGIS typy mają strukturę hierarchiczną, najbardziej ogólne typy to *geometry* i *geography*. Do kolumn o tych typach można wstawać dane o dowolnym z podtypów dla typu kolumny.

3.2. *Geometry* i podtypy

Typ *Geometry* służy do reprezentowania obiektów w dwóch wymiarach, traktuje świat jako płaską siatkę współrzędnych kartezjańskich, tak jak robimy to na mapach. Model ten jest intuicyjny i łatwo na nim wykonywać obliczenia, ale ma jedną poważną wadę, którą jest płaska ziemia.

Geometry ma wiele podtypów, do najbardziej popularnych należą:

- *POINT* - reprezentacja pojedynczego punktu
- *LINESTRING* - reprezentacja linii, ścieżki
- *POLYGON* - reprezentacja wielokątu

Istnieją również podtypy dla tych podtypów, przykładowo *POINT* ma podtypy, które umożliwiają dodanie trzeciego wymiaru:

- *POINT* - 2D
- *POINTZ* - 3D
- *POINTM* - 2D z miarą
- *POINTZM* - 3D z miarą

Dla nas jednak wystarczą tylko te podstawowe podtypy, warto jednak pamiętać, że na tym hierarchia się nie kończy.

3.2.1. *POINT*

Służy do reprezentacji pojedynczego punktu na mapie, składa się z dwóch współrzędnych (dla punktu w 2D).

Przykład - Deklaracja typu dla kolumny: *geometry* z podtypem *POINT*

geometry(POINT, 4326)

- *geometry* - typ danych
- *POINT* - podtyp
- *4326* - SRID (o tym później)

Przykład - utworzenie bazy danych z typem geometry(POINT, 4326)

```
CREATE TABLE miasta (
    id SERIAL PRIMARY KEY,
    nazwa VARCHAR(64),
    location geometry(POINT,4326)
);
```

Przykład z projektu:

```
CREATE TABLE public.punkty
(
    id bigint NOT NULL DEFAULT nextval('punkty_id_seq'::regclass),
    geom geometry(Point,4326),
    label integer,
    CONSTRAINT punkty_pkey PRIMARY KEY (id)
)
```

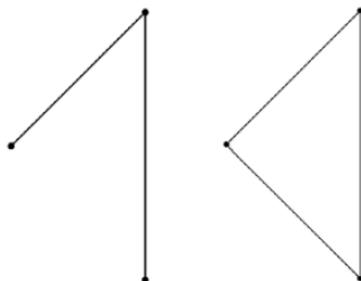
Przykład Wstawianie danych

```
INSERT INTO miasta (nazwa, location) VALUES ('Krakow', ST_GeomFromText('POINT(19.938333 50.061389
)', 4326));
INSERT INTO miasta (nazwa, location) VALUES ('Warszawa', ST_GeomFromText('POINT(21.008333 52.2322
22)', 4326));
```

Korzystamy tu z funkcji ST_GeomFromText, patrz rozdział *Funkcje PostGIS*.

3.2.2. LINESTRING

Służy do reprezentacji ciągu punktów, które są ze sobą pod pewnym względem spójne, przykładowo, rzeki, ulice, trasy, rury itp.



Rysunek 10: Geometria typu Linestring

Przykład - *LINESTRING*

```

CREATE TABLE ch02.my_linestrings (
    id serial PRIMARY KEY,
    name varchar(20),
    my_linestrings geometry(LINESTRING)
);

INSERT INTO ch02.my_linestrings (name, my_linestrings)
VALUES
    ('Open', ST_GeomFromText('LINESTRING(0 0, 1 1, 1 -1)'),           ↪ ② Insert open
     ('Closed', ST_GeomFromText('LINESTRING(0 0, 1 1, 1 -1, 0 0)')));   ↪ ③ Insert closed linestring
                                                               ↪ ① Create table

```

3.2.3. *POLYGON*

To typ, który jest podobny do zamkniętego *LINESTRING* (takiego, który ma punkt startowy taki jak końcowy), z tą różnicą, że tutaj, cała przestrzeń wewnętrz *POLYGON* uznawana jest jako jego część. Granice *POLYGON* określane są w terminologii PostGIS terminem *ring*.



Rysunek 11: Geometria typu Polygon

Przykład - *POLYGON*

```

ALTER TABLE ch02.my_geometries ADD COLUMN my_polygons geometry(POLYGON);
INSERT INTO ch02.my_geometries (name, my_polygons)
VALUES
    ('Triangle',
     ST_GeomFromText('POLYGON((0 0, 1 1, 1 -1, 0 0))'))
;

```

POLYGON może być również reprezentowany jako kilka zamkniętych *LINESTRING*, wtedy, pierwszy z nich określa zewnętrzne granice (outer ring), a każdy kolejny wycina z niego skrawek (inner ring).

Przykład - *POLYGON*

```
INSERT INTO ch02.my_geometries (name,my_polygons)
VALUES (
    'Square 2 holes',
    ST_GeomFromText('POLYGON(
        (-0.25 -1.25,-0.25 1.25,2.5 1.25,2.5 -1.25,-0.25 -1.25),
        (2.25 0,1.25 1,1.25 -1,2.25 0),(1 -1,1 1,0 0,1 -1))
    )');
)
```

**3.3. SRID**

SRID to skrót od *Spatial Reference ID*, czyli id jednego z *Spatial Reference System* (układu współrzędnych).

Chodzi o to, że w zależności od tego jaki obszar przyjmiemy do analizy przestrzennej, nasze współrzędne będą się różnić. Przykładowo jeśli ujmiemy na mapie całą ziemię i utworzymy dla takiej reprezentacji układ współrzędnych, to będzie on mniej dokładny niż gdybyśmy analizowali tylko i wyłącznie sam obszar Polski.

Powszechnie stosowanym układem jest tzw. WGS 84, dla którego SRID to 4326, a więc ten który stosowaliśmy w przykładzie. Układ ten obejmuje obszarem cały glob, ale jest najmniej dokładny. Jest to układ odniesienia, który bazuje na stopniach (długość i szerokość geograficzna) i chyba jest on każdemu znany z geografii. Co ważne, układ ten jest podstawą dla typu *Geography*, który zakłada, że wszystkie dane są zapisane w tym układzie.

Najważniejsze jest jednak dla nas, że aby móc obliczać relacje pomiędzy dwoma obiektami, przykładowo odległość pomiędzy dwoma punktami, musimy mieć dwa obiekty o współrzędnych z tym samym SRID. Dlatego przed wykonaniem takich operacji musimy zawsze się upewnić, że operujemy na danych o tym samym SRID, ewentualnie zastosować konwersję na wybrany SRID.

Pamiętaj - obliczanie relacji a SRID

Jeśli dwa różne obiekty przestrzenne mają to samo SRID, wtedy możemy obliczyć relację pomiędzy nimi (np. odległość, zawieranie w sobie). Jeśli mamy do czynienia z dwoma różnymi SRID, ale są one nam znane, to możemy wykonać konwersję jednego z nich tak aby do siebie pasowały.

Po zainstalowaniu PostGIS w naszej bazie, mamy automatycznie dodawaną tabelę z dostępnymi SRS:

The screenshot shows a PostgreSQL Query Editor interface. At the top, there are tabs for 'Query Editor' and 'Query History'. Below them, a code editor contains the following SQL query:

```

1 SELECT * FROM public.spatial_ref_sys
2

```

Below the code editor, there are tabs for 'Data Output', 'Explain', 'Messages', and 'Notifications'. The 'Data Output' tab is selected, displaying a table with 145 rows. The table has columns: srid [PK] integer, auth_name character varying (256), auth_srid integer, srtext character varying (2048), and proj4text character varying (2048). The data consists of various EPSG codes and their corresponding definitions.

srid [PK] integer	auth_name character varying (256)	auth_srid integer	srtext character varying (2048)	proj4text character varying (2048)
131	4201 EPSG	4201	GEOGCS["Adindan",DATUM["Adl... +proj=longlat +ellps=clrk80 +to...	
132	4202 EPSG	4202	GEOGCS["AGD66",DATUM["Aust... +proj=longlat +ellps=aust_SA +...	
133	4203 EPSG	4203	GEOGCS["AGD84",DATUM["Aust... +proj=longlat +ellps=aust_SA +...	
134	4204 EPSG	4204	GEOGCS["Ain el Abd",DATUM["A... +proj=longlat +ellps=intl +towg...	
135	4205 EPSG	4205	GEOGCS["Afgooye",DATUM["Aga... +proj=longlat +ellps=krass +to...	
136	4206 EPSG	4206	GEOGCS["Agadez",DATUM["Aga... +proj=longlat +a=6378249.2 +b...	
137	4207 EPSG	4207	GEOGCS["Lisbon",DATUM["Lisb... +proj=longlat +ellps=intl +towg...	
138	4208 EPSG	4208	GEOGCS["Aratu",DATUM["Aratu"... +proj=longlat +ellps=intl +towg...	
139	4209 EPSG	4209	GEOGCS["Arc 1950",DATUM["Ar... +proj=longlat +a=6378249.145 ...	
140	4210 EPSG	4210	GEOGCS["Arc 1960",DATUM["Ar... +proj=longlat +ellps=clrk80 +to...	
141	4211 EPSG	4211	GEOGCS["Batavia",DATUM["Bat... +proj=longlat +ellps=bessel +to...	
142	4212 EPSG	4212	GEOGCS["Barbados 1938",DAT... +proj=longlat +ellps=clrk80 +to...	
143	4213 EPSG	4213	GEOGCS["Beduaram",DATUM["... +proj=longlat +a=6378249.2 +b...	
144	4214 EPSG	4214	GEOGCS["Beijing 1954",DATUM... +proj=longlat +ellps=krass +to...	
145	4215 EPSG	4215	GEOGCS["Belge 1950",DATUM["... +proj=longlat +ellps=intl +no_d...	

Rysunek 12: Tabela z SRS w PostGis.

3.4. Typ *Geography* i podtypy

Kiedy mamy do zamodelowania obiekty, które są od siebie bardzo oddalone geograficznie, często konieczne jest wzięcie pod uwagę krzywizny ziemi, w takich przypadkach korzystamy z typu *Geography*. Przykładowo dla modelowania tras lotów pasażerskich oczywistym wyborem jest typ *Geography*. Dokładniejszą reprezentację dostajemy kosztem dłuższego czasu obliczeń, który jest spowodowany bardziej złożoną matematyką dla tego modelu.

Pamiętaj

Typ *geography* zakłada, że dane są przechowywane w układzie WGS 84 (długość i szerokość geograficzna). Jeśli nie określmy żadnego układu dla *geography*, to defaultowo przyjęty zostanie WGS84.

Struktura podtypów dla *geography* imituje tą dla *geometry* dlatego wszystkie typy, które omówiliśmy dla **geometry** (*POINT*, *LINESTRING*, *POLYGON*) są nadal dostępne i aktualne dla *geography*. Jedyną różnicą jest tutaj zamiana terminu *geometry* na *geography*:

Przykład

```
CREATE TABLE ch02.my_geogs (
    id serial PRIMARY KEY,
    name varchar(20),
    my_point geography(POINT)
);
INSERT INTO my_geogs (name, my_point)
VALUES
    ('Home',ST_GeogFromText('POINT(0 0)'),),
    ('Pizza 1',ST_GeogFromText('POINT(1 1)'),),
    ('Pizza 2',ST_GeogFromText('POINT(1 -1)'));
```

3.4.1. Różnica pomiędzy *geometry* i *geography*

Jeśli więc mamy te same podtypy i możemy reprezentować je w tych samych układach współrzędnych, to ktoś może się zastanawiać jaka jest w końcu różnica pomiędzy *geometry* i *geography*. Na pierwszy rzut oka różnica jest rzeczywiście niewidoczna, chodzi jednak o to, że inaczej przetwarzają dane. Porównajmy. Obliczmy dystans pomiędzy Krakowem i Warszawą najpierw dla *geography*, a potem dla *geometry*:

Przykład - *geography*

```
-- Zadanie 2.1

CREATE TABLE geomiasta (
    id SERIAL PRIMARY KEY,
    nazwa VARCHAR(64),
    location geography(POINT,4326)
);

-- Zadanie 2.2

INSERT INTO geomiasta (nazwa, location) VALUES ('Krakow', ST_GeomFromText('POINT(19.938333 50.061
389)', 4326));
INSERT INTO geomiasta (nazwa, location) VALUES ('Warszawa', ST_GeomFromText('POINT(21.008333 52.2
32222)', 4326));

-- zadanie 2.3

SELECT ST_DISTANCE(
    krakow.location, warszawa.location
) as metdistance
FROM geomiasta krakow, geomiasta warszawa
WHERE krakow.nazwa = 'Krakow' AND warszawa.nazwa = 'Warszawa';
```

Otrzymujemy dystans w metrach (252 km):

```
wrobdm1=> \i zad2_3.sql
      metdistance
-----
 252840.13360491
(1 row)
```

Przykład - *geometry* - niepoprawny wynik

```
-- Zadanie 1.1

CREATE TABLE miasta (
    id SERIAL PRIMARY KEY,
    nazwa VARCHAR(64),
    location geometry(POINT,4326)
);

-- Zadanie 1.2

INSERT INTO miasta (nazwa, location) VALUES ('Krakow', ST_GeomFromText('POINT(19.938333 50.061389
)', 4326));
INSERT INTO miasta (nazwa, location) VALUES ('Warszawa', ST_GeomFromText('POINT(21.008333 52.2322
22)', 4326));

-- zadanie 1.3

SELECT ST_DISTANCE(
    krakow.location, warszawa.location
) as degDistance
FROM miasta krakow, miasta warszawa
WHERE krakow.nazwa = 'Krakow' AND warszawa.nazwa = 'Warszawa';
```

Otrzymujemy dystans w stopniach (prosta geometria pitagorejska)

```
wrobdom1=> \i zad1_3.sql
      degdistance
-----
      2.42020989046178
(1 row)
```

Aby dostać poprawny wynik musimy zastosować konwersję:

Przykład - *geometry* - poprawy wynik

```
SELECT ST_Distance(
    ST_Transform(
        krakow.location, 2178
    ),
    ST_Transform(
        warszawa.location, 2178
    )
) as meterDistance
FROM miasta krakow, miasta warszawa
WHERE krakow.nazwa = 'Krakow' AND warszawa.nazwa = 'Warszawa';

wrobdm1=> \i zad1_4.sql
meterdistance
-----
252826.599011547
(1 row)
```

Funkcja ST_Transform - patrz FunkcjePostGIS.

Pamiętaj

Funkcje mierzące w PostGIS (np. dystans itp.) dla typu *geometry* zawsze dają wynik w takich jednostkach w jakich zapisane są współrzędne. Jeśli nasze współrzędne są w stopach, to długości dostaniemy w stopach a pola w stopach kwadratowych, dlatego tak ważne jest dopasowanie odpowiednich współrzędnych zwłaszcza przy korzystaniu z tego typu funkcji.

3.5. GeoJSON

Geometry JavaScript Object Notation to format bazujący na JSON. GeoJSON rozszerza JSON o specyfikację zapisu obiektów przestrzennych. Przykład

```
1 { "type": "FeatureCollection",
2   "features": [
3     { "type": "Feature",
4       "geometry": { "type": "Point", "coordinates": [102.0, 0.5] },
5       "properties": { "prop0": "value0" }
6     },
7     { "type": "Feature",
8       "geometry": {
9         "type": "LineString",
10        "coordinates": [
11          [102.0, 0.0], [103.0, 1.0], [104.0, 0.0], [105.0, 1.0]
12        ]
13      },
14      "properties": {
15        "prop0": "value0",
16        "prop1": 0.0
17      }
18    },
19    { "type": "Feature",
20      "geometry": {
21        "type": "Polygon",
22        "coordinates": [
23          [ [100.0, 0.0], [101.0, 0.0], [101.0, 1.0],
24            [100.0, 1.0], [100.0, 0.0] ]
25        ]
26      },
27      "properties": {
28        "prop0": "value0",
29        "prop1": { "this": "that" }
30      }
31    }
32  ]
33 }
```

3.6. Funkcje PostGIS

Tutaj omówimy sobie najważniejsze funkcje PostGIS, które służą do operowania na danych geograficznych.

3.6.1. ST_GeomFromText

To funkcja, która tworzy obiekt typu geometry na podstawie tekstu, jako pierwszy argument przyjmuje tekst, który ma reprezentować obiekt geometry, a jako drugi SRID dla tego obiektu.

Przykład - ST_GeomFromText

- *POINT*

```
INSERT INTO ch02.my_points (p, pz, pm, pzm, p_srid)
VALUES (
    ST_GeomFromText('POINT(1 -1)'),
    ST_GeomFromText('POINT Z(1 -1 1)'),
    ST_GeomFromText('POINT M(1 -1 1)'),
    ST_GeomFromText('POINT ZM(1 -1 1 1)'),
    ST_GeomFromText('POINT(1 -1)', 4269)
) ;
```

- *LINESTRING*

```
INSERT INTO ch02.my_linestrings (name, my_linestrings)
VALUES
    ('Open', ST_GeomFromText('LINESTRING(0 0, 1 1, 1 -1)'), 2),
    ('Closed', ST_GeomFromText('LINESTRING(0 0, 1 1, 1 -1, 0 0)'), 3);
```

- *POLYGON*

```
INSERT INTO ch02.my_geometries (name, my_polygons)
VALUES (
    'Triangle',
    ST_GeomFromText('POLYGON((0 0, 1 1, 1 -1, 0 0))')
) ;
```

3.6.2. ST_GeogFromText

Odpowiednik ST_GeomFromText dla typu *geography*, ale tutaj jeśli nie podamy układu współrzędnych, to przyjęty zostanie WGS 84 (4326).

Przykład - ST_GeogFromText

```
INSERT INTO my_geogs (name, my_point)
VALUES
    ('Home', ST_GeogFromText('POINT(0 0)'), 1),
    ('Pizza 1', ST_GeogFromText('POINT(1 1)'), 2),
    ('Pizza 2', ST_GeogFromText('POINT(1 -1)'), 3);
```

3.6.3. ST_Transform

Zwraca obiekt typu *geometry*, który został przekonwertowany na układ którego SRID podaliśmy jako drugi argument.

```
geometry ST_Transform(geometry g1, integer srid);
```

Funkcja ta nie ma odpowiednika dla typu *geography*. Możemy natomiast wykonywać prostą konwersję *geometry* na *geography* i odwrotnie, przykładowo:

```
SELECT
    ST_Transform(
        ST_ClosestPoint(
            ST_Transform(geog::geometry, 32618),
            ST_Transform(
                'SRID=4326;LINESTRING(-73 41,-72 42)' ::geometry, 32618
            )
        ),
        4326
    ) ::geography;
```

Funkcja ST_ClosestPoint przyjmuje jako argument tylko typ *geometry*.

3.6.4. ST_GeomFromGeoJSON

Funkcja tworzy obiekt *geometry* na podstawie GeoJSON, który możemy przekazać jako plik lub tekst. Jeśli nie określmy inaczej, otrzymany obiekt *geometry* będzie miał SRID odpowiadające WGS 84.

Przykład

```
SELECT ST_AsText(ST_GeomFromGeoJSON('{"type":"Point","coordinates":[-48.23456,20.12345]'}')) As wkt;
wkt
-----
POINT(-48.23456 20.12345)
```

3.6.5. ST_Centroid

Funkcja oblicza środek ciężkości danego obiektu *geometry*. Nie ma odpowiednika dla **geography**, ale można to ominąć przez rzutowanie na *geometry*.

```
geometry ST_Centroid(geometryg1);
```

```
SELECT
    ST_Transform(
        ST_ClosestPoint(
            ST_Transform(geog::geometry, 32618),
            ST_Transform(
                'SRID=4326;LINESTRING(-73 41,-72 42)' ::geometry, 32618
            )
        ),
        4326
    ) ::geography;
```

3.6.6. ST_X i ST_Y

To funkcje, których można użyć do uzyskania współrzędnej x oraz y danego punktu dla typu *geometry*, nie ma odpowiednika dla *geography*, ale możemy to ominąć przez rzutowanie.

3.6.7. ST_Area

To funkcja, która służy do obliczania pola powierzchni obiektu o podtypie *Polygon*. Funkcja ta działa zarówno dla typu *geometry* jak i *geography*.

- Używając funkcji na typie *geography* zawsze dostaniemy wynik w metrach kwadratowych
- Używając funkcji na typie *geometry* zawsze dostaniemy wynik w takich jednostkach jak układ współrzędnych (dla WGS 84 w stopniach), dlatego musimy upewnić się, że nasze dane są odpowiednio zapisane lub przekonwertować je na odpowiedni układ współrzędnych w celu wykonania obliczeń

Oczywiście pamiętajmy także, że działając na typie *geography* uwzględnimy krzywiznę ziemi, a działając na typie *geometry* nie.

Przykład

```
wrobdm1=> SELECT ST_Area(ST_Transform(krakow.geom, 2180)) FROM admin_krakow;
          st_area
-----
 326399267.424197
(1 row)

wrobdm1=> SELECT ST_Area(krakow.geo) FROM admin_krakow;
          st_area
-----
 326816324.798849
(1 row)
```

3.6.8. ST_Length i ST_Perimeter

To funkcje, które służą do obliczania odpowiednio długości obiektu typu *LINESTRING* (ST_Length) oraz obwodu obiektu typu *POLYGON*.

Obie te funkcje są zdefiniowane zarówno dla typu *geometry* jak i *geography*. Podobnie jak dla *ST_Area*:

- Używając funkcji na typie *geography* zawsze dostaniemy wynik w metrach
- Używając funkcji na typie *geometry* zawsze dostaniemy wynik w takich jednostkach jak układ współrzędnych (dla WGS 84 w stopniach), dlatego musimy upewnić się, że nasze dane są odpowiednio zapisane lub przekonwertować je na odpowiedni układ współrzędnych w celu wykonania obliczeń

Przykład

```
wrobdm1=> SELECT SUM(ST_LENGTH(r.geom)) FROM roads r, admin a WHERE ST_Covers(a  
.geom, r.geom);  
      sum  
-----  
 6675812.12702389  
(1 row)  
  
wrobdm1=> SELECT SUM(ST_LENGTH(r.geo)) FROM roads r WHERE class='motorways';  
      sum  
-----  
141987.110175961  
(1 row)
```

3.6.9. ST_Covers

To funkcja, która sprawdza czy jej pierwszy argument w całości zawiera jej drugi argument i zwraca *true*(jeśli tak) lub *false*(jeśli nie). Funkcja ta jest zdefiniowana dla typów *geometry* i *geography*. Funkcja korzysta z *bounding box* do wykonania obliczeń.

Przykład

```
wrobdm1=> SELECT SUM(ST_LENGTH(r.geom)) FROM roads r, admin a WHERE ST_Covers(a  
.geom, r.geom);  
      sum  
-----  
 6675812.12702389  
(1 row)
```

3.6.10. ST_Distance

To funkcja, która oblicza najkrótszy dystans pomiędzy dwoma obiektami *geometry* lub *geography* (dwa obiekty muszą być tego samego typu). W odróżnieniu od ST_Length i ST_Perimeter, funkcja ta działa na dowolnych podtypach, tzn. możemy mierzyć odległość punktu od linestring, linestring od polygon itd.

Oczywiście również tutaj obwiązuje nas te same zasady, przypomnijmy:

- Używając funkcji na typie *geography* zawsze dostaniemy wynik w metrach
- Używając funkcji na typie *geometry* zawsze dostaniemy wynik w takich jednostkach jak układ współrzędnych (dla WGS 84 w stopniach), dlatego musimy upewnić się, że nasze dane są odpowiednio zapisane lub przekonwertować je na odpowiedni układ współrzędnych w celu wykonania obliczeń

Przykład

```
wrobdm1=> SELECT TYPE, count(*)  
  FROM amenities  
 WHERE ST_DISTANCE(  
    ST_TRANSFORM( ST_GeomFromText('POINT(19.945901 50.065920)', 4326), 2180),  
    ST_TRANSFORM(amenities.geom, 2180)  
  ) <= 2000  
 GROUP BY TYPE ORDER BY count desc;  
   type | count  
-----+-----  
 school | 23  
 university | 8  
 fuel | 7  
 library | 5  
 police | 3  
 townhall | 2  
 hospital | 1  
(7 rows)
```

3.7. Zadania z poprzednich lat

Egzamin

3. (15 pkt.) W osobnych plikach dostarczono granice Krakowa oraz Polski, zapisane jako obiekty `POLYGON` w formacie GeoJSON – same obiekty zgodne z formatem dla pola `geometry` w GeoJSON. Funkcja `ST_GeomFromGeoJSON(text geojson)` zwraca obiekt typu `geometry` bez układu odniesienia. Napisz kod SQL realizujący poniższe czynności. Jeżeli nie pamiętasz odpowiednich wartości SRID, zastąp je wybranymi stałymi i opisz co oznaczają.
1. Tworzy tabelę i wypełnia ją danymi z plików, przechowując współrzędne w kolumnie typu `geometry` wraz z semantyką wartości. Zawartość plików zastąp odpowiednio symbolami `*A*` oraz `*B*`.
 2. Oblicza powierzchnię Krakowa w m^2 , nie korzystając bezpośrednio z innych typów PostGIS.
 3. Oblicza powierzchnię Polski w km^2 , nie korzystając bezpośrednio z funkcji przekształcających współrzędne.

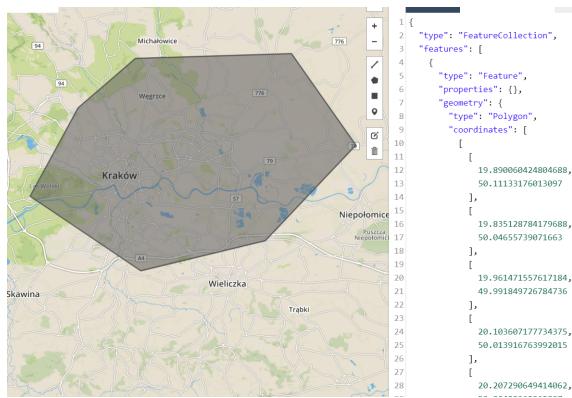
Tworzymy tabelę:

```

1 CREATE TABLE granice (
2     id SERIAL PRIMARY KEY,
3     nazwa VARCHAR(64),
4     granica geometry(POLYGON, *4326*)
5 );

```

Weźmy sobie takie dane dla Krakowa (z geojson.io):



I takie dla Polski:

ZAAWANSOWANE TECHNOLOGIE BAZODANOWE - EGZAMIN



Wstawiamy dane do tabeli (oczywiście zgodnie z poleceniem wstawimy w miejsce JSONA nazwy plików):

```

1 INSERT INTO granice (nazwa, granica) VALUES ('Krakow', ST_GeomFromGeoJSON('{
2   "type": "Polygon",
3   "coordinates": [
4     [
5       [
6         [
7           [
8             [
9               [
10              [
11                [
12                  [
13                    [
14                      [
15                        [
16                          [
17                            [
18                              [
19                                [
20                                  [
21                                    [
22                                      [
23                                        [
24                                          [
25                                            [
26                                              [
27                                                [
28                                                  [
29                                                    [
30                                                      [
31
32
33
34
35
36
37
38
39

```

```

1 INSERT INTO granice (nazwa, granica) VALUES ('Polska', ST_GeomFromGeoJSON('{
2   "type": "Polygon",
3   "coordinates": [
4     [

```

```

5      [
6          19.51171875,
7          54.648412502316695
8      ],
9      [
10         14.2822265625,
11         54.25238930276849
12     ],
13     [
14         14.17236328125,
15         51.91716758909015
16     ],
17     [
18         18.369140624999996,
19         49.33944093715546
20     ],
21     [
22         23.752441406249996,
23         49.33944093715546
24     ],
25     [
26         24.41162109375,
27         53.51418452077113
28     ],
29     [
30         19.51171875,
31         54.648412502316695
32     ],
33   ]
34 }
35 ) );

```

Tworzymy zapytania, najpierw nie możemy korzystać z innych typów PostGIS a mamy obliczyć w metrach kwadratowych, nasze współrzędne są w WGS 84, a więc mamy stopnie, musimy zatem skorzystać z konwersji na jakiś inny układ, który zwróci nam pole w metrach, przykładowo 2178.



The screenshot shows a PostgreSQL Query Editor window. The query in the editor is:

```
1 SELECT ST_Area(ST_Transform(granica, 2178)) FROM granice WHERE id = 1;
```

The results pane shows a single row of data:

st_area
double precision
1 319493019.5658099

A według danych google to $327000000m^2$ więc jest ok.

W drugim zapytaniu nie możemy korzystać z funkcji przekształcających współrzędne więc możemy chytrze przekonwertować się na *geography*:



The screenshot shows a PostgreSQL Query Editor window. The query in the editor is:

```
1 SELECT ST_Area(granica::geography) / 1000000 FROM granice WHERE id = 2;
```

The results pane shows a single row of data:

?column?
double precision
1 331760.4060560581

A według danych google to $312679 km^2$ więc też jest ok.

4 ACID

4.1. Zadania z poprzednich lat

Egzamin

Opisać czym jest ACID.

Zbiór właściwości gwarantujących poprawne przetwarzanie transakcji w bazach danych. ACID jest skrótwcem od angielskich słów atomicity, consistency, isolation, durability, czyli niepodzielność, spójność, izolacja, trwałość.

- atomicity - każda transakcja albo zostanie wykonana w całości, albo w ogóle – na przykład jeśli w ramach jednej transakcji odbywać się ma przelew bankowy (zmniejszenie wartości jednego konta i powiększenie innego o tę samą kwotę), to nie może dojść do sytuacji, że z jednego konta ubędzie pieniędzy, a kwota na koncie docelowym będzie bez zmian
- consistency - zapewnia, że każda transakcja może tylko zmienić stan bazy danych z prawidłowego na prawidłowy, tzn. po wykonaniu transakcji, wszystkie dane nadal poprawne względem ograniczeń, triggerów itd.
- isolation - transakcje, które są wykonywane równolegle nie wpływają na siebie nawzajem, baza jest pozostawiana w takim stanie jakby transakcje były wykonywane sekwencyjnie
- durability - gwarantuje, że dane, które zostały zapisane w bazie danych pozostały w niej, nawet w przypadku awarii systemu

5 Indeksy

Indeksy są nieodłącznie powiązane z wykonywaniem zapytań na bazie, bo służą do optymalizacji wyciągania z bazy danych przy pomocy zapytania typu SELECT.

5.1. Ogólny proces wysyłania zapytani SELECT

Zanim przejdziemy stricte do omówienia indeksów przedstawmy sobie jaki proces przechodzi zapytanie SELECT od momentu jego wysłania do momentu gdy dostajemy dane z bazy.

- Aplikacja ustanawia połączenie z bazą danych
- Aplikacja wysyła zapytanie do bazy danych
- Baza danych sprawdza poprawność zapytania i buduje na jego podstawie drzewo zapytania
- Następnie działa tzw. *rewriter*, który bierze drzewo zapytania i transformuje go według pewnych reguł o ile jest taka potrzeba
- **Planner/Optimizer bierze przepisane drzewo zapytania i tworzy na jego podstawie plan zapytania, który będzie wejściem do executor, który wykona zapytanie** - to właśnie na tym etapie tworzone są wszystkie możliwe ścieżki prowadzące do uzyskania rezultatu zapytania, np. jeśli na relacji jest indeks, to mamy dwie możliwe ścieżki (sekwencyjna + ta z indeksem), wybierana jest ta najbardziej optymalna i plan dla niej jest przekazywany do *executor*
- *executor* wykonuje plan zapytania i zwracane są wyniki

5.2. ANALYZE i EXPLAIN

Wiemy już o działaniu plannera, ale skąd mamy wiedzieć jaki optymalny plan on sobie wybrał dla naszego zapytania ? Do tego właśnie służy nam polecenie *EXPLAIN*, które zwraca dla danego zapytania, to co planner uznał za optymalne.

Jeśli dodamy po poleceniu EXPLAIN również polecenie *ANALYZE* po którym wstawimy nasze zapytanie, to dostaniemy zarówno to co planner zaplanował jak i to co rzeczywiście miało miejsce, zapytanie bowiem zostanie naprawdę wykonane, a dane z plannera i dane rzeczywiste ukażą się nam w celu porównania.

Przykład

```

6. wróblom1=> EXPLAIN ANALYZE VERBOSE SELECT * FROM zamówienia WHERE idkompozycji = 'buk1';
7.                                         QUERY PLAN
8.
9.   Seq Scan on public.zamówienia  (cost=0.00..167.19 rows=424 width=52) (actual time=0.023..2.63
8 rows=424 loops=1)
10.   Output: idzamówienia, idklienta, idodbiorcy, idkompozycji, termin, cena, zapłacone, uwagi
11.   Filter: (zamówienia.idkompozycji = 'buk1'::bpchar)
12.   Rows Removed by Filter: 7591
13.   Planning time: 0.114 ms
14.   Execution time: 2.710 ms
15. (6 rows)
16.

```

Planner zwraca również operacje, które zostają wykonane na podstawie zapytania, w powyższym przykładzie mamy np. operacje *filter*, ponieważ w klauzuli where zawarliśmy porównanie.

5.3. Tworzenie i usuwanie indeksu

Indeks tworzymy przy pomocy polecenia:

```
CREATE INDEX test1_id_index ON test1 (id);
```

a usuwamy przy pomocy polecenia:

```
DROP INDEX test1_id_index ;
```

Defaultowy typ tworzonego indeksu to b-tree.

5.4. Typy indeksów

PostgreSQL zapewnia kilka różnych typów indeksów, każdy z nich używa innego algorytmu, który jest odpowiedni dla pewnego rodzaju zapytań.

5.4.1. Btree

To defaultowy indeks, który jest tworzony jeśli nie określmy jawnie typu indeksu. B-tree może obsługiwać zapytania bazujące a porównywaniu i zakresach, a więc takie w których indeksowana kolumna podlega któremuś z operatorów:

- <
- <=
- =
- >=
- >

Inne operatory, które są tworzone na podstawie tych operatorów (jak np. *BETWEEN*, *IN*, *IS NULL*) mogą również być używane z indeksem Btree.

Oprócz tego *optimizer* może również używać indeksu Btree dla zapytań, które zawierają dopasowywanie do wzorców, czyli dla operatorów takich jak *LIKE* czy (ale uwaga, są dwa warunki !), pierwszy jest taki, że tylko wtedy gdy wzorzec jest stały i znajduje się na początku wzorca. Przykładowo:

- *LIKE foo%*

ale już nie

- *LIKE %bar*

a drugi jest taki, że jeśli nasza baza nie używa C locale, to musimy dodać przy tworzeniu indeksu klasę operatorów:

```
CREATE INDEX zamowienia_uwagi ON zamowienia (uwagi varchar_pattern_ops);
```

5.4.2. Hash

Kolejny typ indeksów to tzw. *HASH*, tworząc taki indeks musimy jawnie określić jego typ, bo nie jest to defaultowe:

```
CREATE INDEX name ON table USING HASH (column);
```

Hashe są o wiele prostsze od Btree i mogą działać tylko wtedy gdy mamy do czynienia operatorem porównania =, a więc mają bardzo ograniczone zastosowanie.

5.4.3. Inne

Oczywiście jest też wiele innych, ale my głównie skupiamy się na dwóch powyższych, opis pozostałych w dokumentacji.

5.5. Indeksy wielokolumnowe

Jeśli mamy tabelę typu:

```
1 CREATE TABLE test2 (
2     major int ,
3     minor int ,
4     name varchar
5 );
```

i często wykonujemy zapytania typu:

```
1 SELECT name FROM test2 WHERE major = constant AND minor = constant;
```

to dobrze jest założyć wtedy indeks wielokolumnowy:

```
1 CREATE INDEX test2_mm_idx ON test2 (major , minor);
```

5.6. Indeksy i sortowanie

Chodzi o to, że jak już zdefiniujemy indeks, to dane układane są według pewnej kolejności określonej przez ten indeks i według takiej kolejności są wyszukiwane. Dla indeksu Btree defaultowa kolejność to *ASC NULLS LAST*.

Jeśli więc sortujemy dane wyjściowe przez kolumnę na której mamy indeks, to wtedy krok sortowania może być pominięty, ponieważ dane wyciągane są w kolejności takiej jak są ułożone przez indeks. My mamy natomiast kontrolę nad tym jak nasze dane będą przechowywane według indeksu (malejąco, rosnąco, nulle najpierw lub na końcu). Dzięki temu możemy oszczędzić bazie kroku sortowania i mieć dane według kolejności którą chcemy.

Przykład

```
CREATE INDEX test2_info_nulls_low ON test2 (info NULLS FIRST);
CREATE INDEX test3_desc_index ON test3 (id DESC NULLS LAST);
```

5.7. Indeksy częściowe i indeksy na wyrażeniach

Indeksy możemy również zakładać na części danych:

Przykład

```
wrobdm1=> CREATE INDEX indIdKlient ON zamowienia(idklienta) WHERE zaplacone;
CREATE INDEX
```

Jak również na wyrażeniach, zamiast na kolumnach:

Przykład

```
wrobdm1=> CREATE INDEX indexMiasto ON klienci (lower(miasto) varchar_pattern_ops);  
CREATE INDEX
```

5.8. Zadania z poprzednich lat

Egzamin

4. (6 pkt.) Napisz zapytanie, które w tabeli **samochody** zakłada indeks oparty o B-drzewa na atrybutie **marka**(varchar), tak aby mógł on być wykorzystywany do sortowania wyników malejąco według marki. Dla każdego z podanych warunków klauzuli WHERE odpowiedz czy indeks ten może zostać użyty i dlaczego:
a) marka ILIKE 'M%', b) marka LIKE 'M%', c) marka LIKE '%d', d) marka >= 'L'.

```
| CREATE INDEX ind ON samochody USING BTREE (marka varchar_pattern_ops DESC NULLS LAST); |
```

- *ILIKE M%* - nie, ponieważ dla ILIKE wzorzec może rozpoczynać się tylko od znaku na którego konwersja do małych / dużych liter nie ma wpływu
- *LIKE M%* - tak, o ile dodamy klasę operatorów jak w przykładzie powyżej
- *LIKE %d* - nie, bo wzorzec nie rozpoczyna się od stałej
- marka $\geq L$ - tak, Btree działa poprawnie dla operatora \geq

6 XML

PostgreSQL umożliwia zarówno generację dokumentów XML na podstawie danych relacyjnych, jak i gromadzenie i przetwarzanie dokumentów XML. Omówimy sobie tutaj krótko jak z istniejącej tabeli uzyskać dane w formie xml, a także jak odpytywać dane, które mamy w bazie w formie xml.

6.1. Tworzenie xml na podstawie tabeli

Załóżmy, że mamy w bazie danych zwykłą tabelę, bez żadnego xml i chcemy z niej wyciągnąć dane, które będą reprezentowane w strukturze xml. Posłuży nam do tego funkcja xmlelement.

```
1 xmlelement(NAME nazwa_elementu_xml, xmlattributes(column1 as atrybut1, column2 as
    atrybut2, ...), 'content1', 'content2', ... )
```

- Pierwszy atrybut tej metody zawsze rozpoczyna się od NAME, a po nim następuje nazwa elementu xml
- Drugi z argumentów jest opcjonalny i określamy w nim atrybuty dla naszego elementu xml, po *as* określamy jak atrybut ma się nazywać
- Kolejne argumenty są również opcjonalne, po przecinku dodajemy zawartość naszego elementu xml, w szczególności mogą to być kolejne funkcje xmlelement tworzące zagnieżdżone elementy xml

Przykład

```
1. SELECT xmlelement (
2.   NAME flowers,
3.   xmlelement(NAME bouquet, xmlattributes (k.idkompozycji AS id, k.stan AS
    quant, k.cena AS price),
4.     xmlelement(NAME NAME, k.nazwa),
5.     xmlelement(NAME description, k.opis)
6.   )
7. ) FROM kompozycje k
8. WHERE stan > 4;
```

```
<flowers>
<bouquet id="ko2" quant="12" price="120.00">
<name>Kosz rozyczek</name>
<description>tuzin czerwonych rozyczek, molucella, gips, sizal, koszyk
czerwony z palakiem
</description>
</bouquet>
</flowers>
```

6.2. Tworzenie zapytań do typu xml

Bazę danych z kolumną xml możemy utworzyć w bardzo prosty sposób:

```
1. CREATE TABLE printers (
2.     id SERIAL PRIMARY KEY,
3.     NAME VARCHAR(50) NOT NULL,
4.     description XML
5. );
```

Teraz chcemy utworzyć zapytanie, które wyciągnie z bazy danych jakieś konkretne dane z naszego xmla.

W tym celu korzystamy z narzędzia *xpath*, pierwszym argumentem do tej funkcji jest ścieżka określająca położenie elementu w dokumencie xml, a drugim kolumna z xml:

Przykład

```
SELECT NAME, Xpath('/printer/mechanism/resolution/dpi/x/text()', description) FROM printers;
```

name	xpath
-----+-----	

Jeśli chcemy wybrać zamiast elementu wartość jakiegoś atrybutu, to nazwę atrybutu poprzedzamy znakiem małpy @.

Kolejną funkcją z której możemy korzystać jest *path_exists*. Funkcja ta zwraca true lub false w zależności od tego czy dany element znajduje się w xml.

Przykład

```
SELECT name FROM printers WHERE Coalesce(Nullif(Btrim(Xpath('/printer/mechanism/re solution/dpi/x/text()', description) :: text, '{}'), ''), '0') :: INT > 1200 AND X path_exists('//printer/mechanism/color', description);
```

name

```
Brother-HL-4070CDW
```

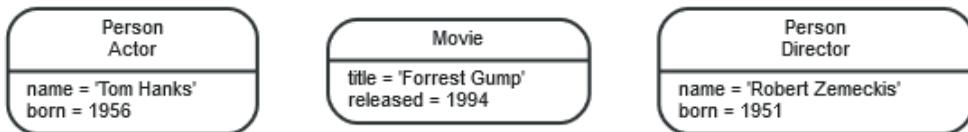
7 Neo4j i Cypher

7.1. Informacje ogólne

- **Cypher** - język zapytań przeznaczony dla grafów, który pozwala użytkownikom na manipulacje danymi w graficznej bazie danych
- **Grafowa baza danych** - to baza danych w której danych nie dostosowujemy do konkretnego modelu, zamiast tego dane są łączone z innymi danymi relacjami, które tak jak w grafie pokazują połączenie lub relację pomiędzy nimi. Połączenia między danymi przechowywane są jako dane w modelu danych. Zapytania na grafowej bazie danych są bardzo wydajne, działają w czasie stałym i pozwalają na przeszukiwanie nawet milionów połączeń

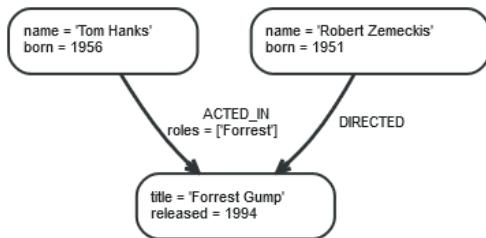
7.2. Podstawowe pojęcia neo4j

- **Węzły** - używane są w neo4j do reprezentowania pojedynczego węzła, czyli danych
- **Etykiety** - przypisywane są do węzłów i używane są w celu pogrupowania węzłów w zbiory, np. wszystkie węzły użytkowników mogłyby mieć przypisaną etykietę User. Etykiety mogą być dodawane i usuwane po dodaniu węzła. Węzeł może mieć 0 lub więcej etykiet.



Rysunek 13: Węzły w Neo4j.

- **Relacje** - relacje łączą dwa węzły ze sobą, co tworzy struktury w naszym grafie. Relacja może mieć tylko jeden typ, ale za to może mieć 0 lub więcej własności. Relacje mogą być nieskierowane (z żadnej strony nie ma strzałki) lub skierowane tylko w jednym kierunku (strzałka tylko w jedną stronę). Relacja może być w szczególności skierowana do węzła z którego wychodzi.



Rysunek 14: Jedna z relacji jest typu DIRECTED, a druga ACTED_IN, relacja ACTED_IN ma dodatkowo przypisaną własność, która jest tablicą.

- **Własności** - To pary nazwa:wartość, które są używane do dodania informacji do węzłów, a także relacji.

7.3. Konwencje nazewnictwa w Neo4j

Graph entity	Recommended style	Example
Node label	Camel case, beginning with an upper-case character	:VehicleOwner rather than :vehice_owner
Relationship type	Upper case, using underscore to separate words	:OWNS_VEHICLE rather than :ownsVehicle
Property	Lower camel case, beginning with a lower-case character	firstName rather than first_name

Rysunek 15: Konwencje nazewnictwa w Neo4j.

7.4. Cypher - Podstawy

W tym podrozdziale omówimy sobie język Cypher stosowany do manipulowania danymi w Neo4j.

7.4.1. Węzły

- Węzły reprezentujemy w nawiasach, które mają przypominać okrągły węzeł grafu

Przykłady

```
{}
matrix
(:Movie)
(matrix:Movie)
(matrix:Movie {title: "The Matrix"})
(matrix:Movie {title: "The Matrix", released: 1997})
```

Rysunek 16: Przykładowe reprezentacje węzłów w Cypher.

- Najprostszy węzeł to po prostu dwa nawiasy
- W drugiej linii widzimy węzeł ze zmienną, zmienną możemy wprowadzić do węzła jeśli chcemy się do niego odwoływać w innym miejscu
- W kolejnych liniach widzimy dodanie do węzła etykiety Movie
- Własności zapisujemy w nawiasach wąsatych w formacie nazwa:wartość, oddzielone przecinkami

7.4.2. Relacje

- Relacja reprezentowana jest jako dwa znaki myślnika (-), możemy dodatkowo dodać grot strzałki aby uformować relację skierowaną (->) lub (<-).

```
-->
-[role]->
-[:ACTED_IN]->
-[:role:ACTED_IN]->
-[:role:ACTED_IN {roles: ["Neo"]}]>
```

Rysunek 17: Przykładowe reprezentacje relacji w Cypher.

- Nawiasy kwadratowe pomiędzy myślnikami używane są do dodawania do relacji zmiennych, etykiet oraz własności.

7.4.3. Węzły + Relacje = Wzorce

- Wzorce tworzymy poprzez połączenie składni węzłów oraz relacji

Przykład

```
(keanu:Person:Actor {name: "Keanu Reeves"} )  
-[role:ACTED_IN {roles: ["Neo"] } ]->  
(matrix:Movie {title: "The Matrix" })
```

Rysunek 18: Przykładowy wzorzec w Cypher.

7.5. Cypher - CREATE, MATCH, MERGE

- Wiedząc jak tworzyć wzorce w języku Cypher możemy teraz zacząć tworzyć zdania, które mają wykonywać określone zadania w naszej bazie danych takie jak tworzeniu czy wyszukiwanie.

7.5.1. CREATE

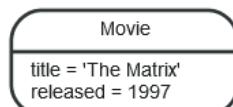
- Create to operacja, która pozwala na utworzenie nowych danych w bazie, mogą to być węzły, relacje, a także węzły wraz z relacjami.

Przykładowo wyrażenie

```
CREATE (:Movie { title:"The Matrix",released:1997 })
```

Rysunek 19: Utworzenie węzła o etykiecie Movie z dwiema własnościami.

jest równoważne utworzeniu takiego węzła w grafie:



Rysunek 20: Utworzony węzeł.

Mogimy też przypisać wartość do nowo tworzonego węzła, np. po to aby go wyświetlić:

```
CREATE (p:Person { name:"Keanu Reeves", born:1964 })
RETURN p
```

This is what gets returned:

```
+-----+
| p |
+-----+
| Node[1]{name:"Keanu Reeves",born:1964} |
+-----+
1 row
Nodes created: 1
Properties set: 2
Labels added: 1
```

Rysunek 21: Tworzenie węzła z przypisaniem zmiennej.

tutaj przykład tworzenia węzłów wraz z relacją:

```
CREATE (a:Person { name:"Tom Hanks",
  born:1956 })-[r:ACTED_IN { roles: ["Forrest"] }]->(m:Movie { title:"Forrest Gump",released:1994 })
CREATE (d:Person { name:"Robert Zemeckis", born:1951 })-[:DIRECTED]->(m)
RETURN a,d,r,m
```

Rysunek 22: Tworzenie węzłów z relacją między nimi.

7.5.2. MATCH

- Aby utworzyć zapytanie, które zwróci informacje z bazy danych, tworzymy również Wzór, który przekazujemy do MATCH
- Szczególnie przydatne tutaj są zmienne, dzięki którym możemy wyświetlić konkretną informację

```
MATCH (p:Person { name:"Tom Hanks" })-[r:ACTED_IN]->(m:Movie)
RETURN m.title, r.roles
```

```
+-----+
| m.title | r.roles |
+-----+
| "Forrest Gump" | ["Forrest"] |
+-----+
1 row
```

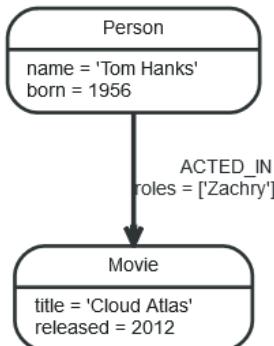
Rysunek 23: Stosowanie MATCH.

MATCH możemy też stosować do aktualizowania bazy danych w taki sposób:

```

MATCH (p:Person { name:"Tom Hanks" })
CREATE (m:Movie { title:"Cloud Atlas",released:2012 })
CREATE (p)-[r:ACTED_IN { roles: ['Zachry']}]->(m)
RETURN p,r,m
  
```

Here's what the structure looks like in the database:



Rysunek 24: Aktualizacja bazy danych.

- Należy pamiętać, że MATCH zwraca jeden wiersz na dopasowanie dane, dlatego korzystając z połączenia MATCH-CREATE musimy wziąć pod uwagę, że CREATE może wykonać się wiele razy ze względu na to, że MATCH zwróci do zmiennej wiele wierszy.

Możemy też stosować MATCH razem z SET w celu dodania atrybutu do istniejących węzłów:

```
MATCH(n) WHERE n.name="S-1" SET n.function ="service";
```

Rysunek 25: Aktualizacja bazy danych.

7.5.3. MERGE

- MERGE działa jak *MATCH or CREATE*, tzn. poszukuje wzorca, jeśli go znajdzie, to go zwraca, jeśli go nie znajdzie, to go tworzy

Przykład

```

MERGE (m:Movie { title:"Cloud Atlas" })
ON CREATE SET m.released = 2012
RETURN m
  
```

m
Node[5]{title:"Cloud Atlas",released:2012}
1 row

Rysunek 26: Aktualizacja bazy danych.

7.6. Cypher - WHERE i funkcje agregujące

7.6.1. WHERE

- WHERE możemy używać podobnie jak w bazach relacyjnych do filtrowania otrzymanych rezultatów, możemy korzystać z wyrażeń regularnych, warunków logicznych i wielu innych opcji

Przykład

```
MATCH (m:Movie)
WHERE m.title = "The Matrix"
RETURN m
```

m
{:Movie {title: "The Matrix", released: 1997}}

1 row

Rysunek 27: Przykład użycia WHERE.

co jest równoważne:

```
MATCH (m:Movie { title: "The Matrix" })
RETURN m
```

można też jednak stosować bardziej skomplikowane rozwiązania:

```
MATCH (p:Person)-[r:ACTED_IN]->(m:Movie)
WHERE p.name =~ "K.+"
OR m.released > 2000
OR "Neo" IN r.roles
RETURN p,r,m
```

Rysunek 28: Przykład użycia WHERE.

zwłaszcza interesujące jest stosowanie wzorców w połączeniu z WHERE:

```
MATCH (p:Person)-[ :ACTED_IN ]->(m)
WHERE NOT (p)-[:DIRECTED]->()
RETURN p,m
```

p	m
{:Person {name: "Tom Hanks", born: 1956}}	{:Movie {title: "Cloud Atlas", released: 2012}}
{:Person {name: "Tom Hanks", born: 1956}}	{:Movie {title: "Forrest Gump", released: 1994}}

2 rows

Rysunek 29: Stosowanie wzorców z WHERE.

7.6.2. Funkcje agregujące

Podobnie jak w zwykłym SQL możemy również stosować funkcję agregującą:

```
MATCH (:Person)
RETURN count(*) AS people
```

```
+-----+
| people |
+-----+
| 3      |
+-----+
1 row
```

Rysunek 30: Przykład zastosowania funkcji agregujących.

8 Inne

Egzamin

4. (3 pkt.) Podaj scenariusz operacji, którego wykonanie zakończy się błędem wyjątkiem przy izolacji transakcji *serializable*.

SET TRANSACTION ISOLATION LEVEL serializable;

BEGIN

}