

Neo4J

Grafowa baza danych

Lipka Magdalena

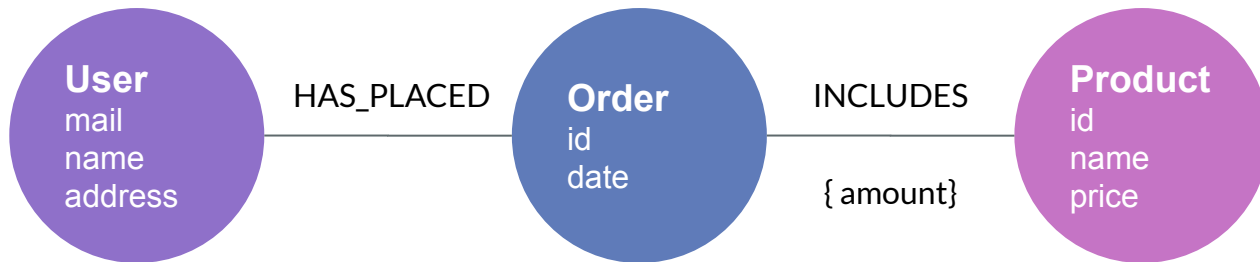
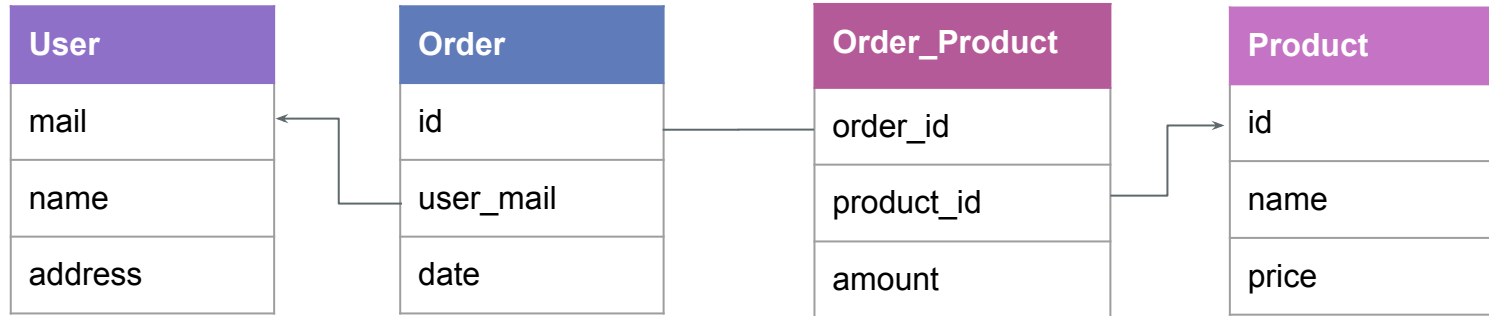
Czym jest grafowa baza
danych?

Grafowe bazy danych - jak sama nazwa mówi - to bazy, w których informacja trzymana jest w grafie. Obiekty poza swoimi atrybutami mogą mieć też krawędzie do innych obiektów. Bazy tego typu są niezwykle użyteczne do modelowania i analizowania struktur w których dane są ze sobą silnie powiązane.

Bazy relacyjne a bazy grafowe

Mimo należenia do grupy “baz nierelacyjnych” bazy grafowe są tak naprawdę bazami przykładającymi największą wagę do relacji między danymi.

Należy zauważyć, że “relacje” w tradycyjnej bazie relacyjnej są tak naprawdę kwestią umowną - tworzy się je sztucznie za pomocą kluczy obcych lub dodatkowych tablic relacji. Taka relacja nie jest obiektem istniejącym w bazie danych, a jedynie założeniem, że fakt identyczności wartości pól oznacza relację między obiektami. Dla bazy grafowej relacja między obiektami jest obiektem i może posiadać własne atrybuty.



Przewaga baz grafowych nad relacyjnymi

Ponieważ w bazach relacyjnych połączenia między danymi w dwóch różnych tabelach nie istnieją jako obiekt, to podczas wykonywania zapytań należy używać klauzuli “JOIN” (lub innych odpowiednich dla języka) co powoduje wczytanie obu tabel i połączenie ich w pamięci do nowej tymczasowej tabeli. Podczas każdego wykonania takiego zapytania owa tymczasowa tabela jest ponownie przeliczana w pamięci. Bazy grafowe rozwiązują ten problem poprzez brak potrzeby obliczania połączeń, w momencie znalezienia jednego wierzchołka można szybko przejść do innych pozostających z nim w relacji bez potrzeby wczytywania i przeszukiwania pozostałych wierzchołków.

Kiedy użyć bazy grafowej

Grafowe bazy są idealnym rozwiązaniem, gdy zajmujemy się problemami związanymi z wzorami, połączeniami, szukamy odpowiedzi na pytania typu, kto kogo zna, między jakimi kontami bankowymi przepływają nielegalne pieniądze, jakie produkty w sklepie są często kupowane razem. Bazy te nie sprawdzą się równie dobrze co standardowe bazy relacyjne do odpowiadania na pytania o średnie zarobki pracowników czy koszt produktów na największym zamówieniu.

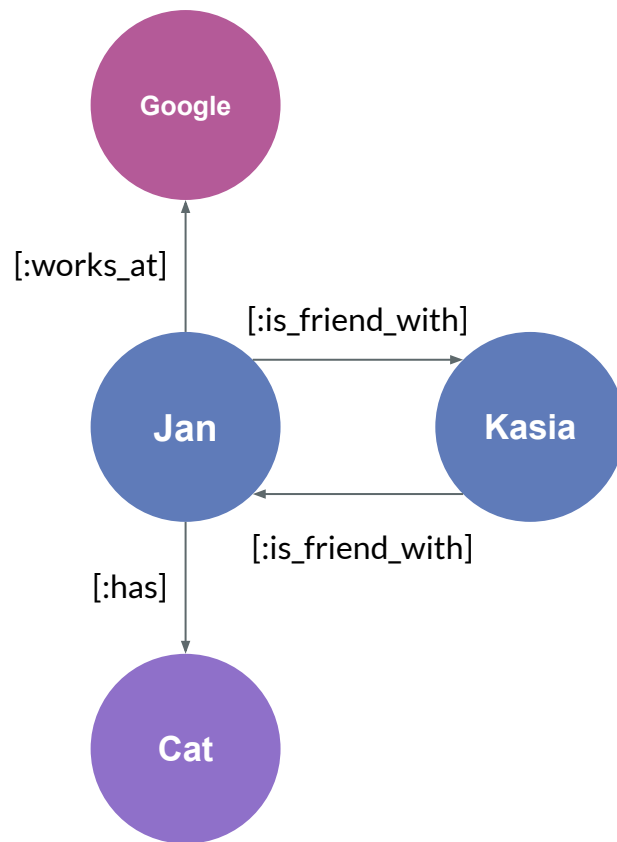
Jak działa Neo4j pod spodem

Istnieją dwa główne podejścia co do wewnętrznych struktur baz grafowych - RDF i "property graph". RDF opiera się na trójkach (wierzchołek, krawędź, wierzchołek), a property graph na obiektach w formacie JSON. Neo4j wykorzystuje property graph, bazą używającą RDF jest np, Dgraph.

```
<0x6bc818dc89e78754> <student> <0xc3bcc578868b719d> .  
<0x6bc818dc89e78754> <student> <0xb294fb8464357b0a> .  
<0x6bc818dc89e78754> <name> "awesome class" .  
<0x6bc818dc89e78754> <dgraph.type> "Class" .  
<0xc3bcc578868b719d> <name> "Alice" .  
<0xc3bcc578868b719d> <dgraph.type> "Person" .  
<0xc3bcc578868b719d> <dgraph.type> "Student" .  
<0xc3bcc578868b719d> <planet> "Mars" .  
<0xc3bcc578868b719d> <friend> <0xb294fb8464357b0a> .  
<0xb294fb8464357b0a> <name> "Bob" .  
<0xb294fb8464357b0a> <dgraph.type> "Person" .  
<0xb294fb8464357b0a> <dgraph.type> "Student" .
```

Źródło: <https://dgraph.io/docs/mutations/blank-nodes/>

Podobnie do baz dokumentowych w Neo4j nie ma potrzeby definiowania schematu danych, obiekty mogą posiadać różne atrybuty, przy czym atrybut o takiej samej nazwie może się różnić typem między obiektami. Bardzo często zamiast “typu” stosuje się pojęcie “label/metka”. Wstawianie obiektów danego modelu pod jedną metką jest obowiązkiem użytkownika, gdyż sama metka nie wymusza typowania, a jest jedynie pomocniczym oznaczeniem. Ważne są jednak “typy” relacji - nie są to typy w standardowym tego słowa znaczeniu, ale mówią o domenowym znaczeniu danej krawędzi (np “is_friend_with”, “works_at”, “has”).



456

123

Node properties ⓘ

AndroidPhone

Device

<id>	5	ⓘ
name	456	ⓘ

456

123

Node properties ⓘ

Device

PC

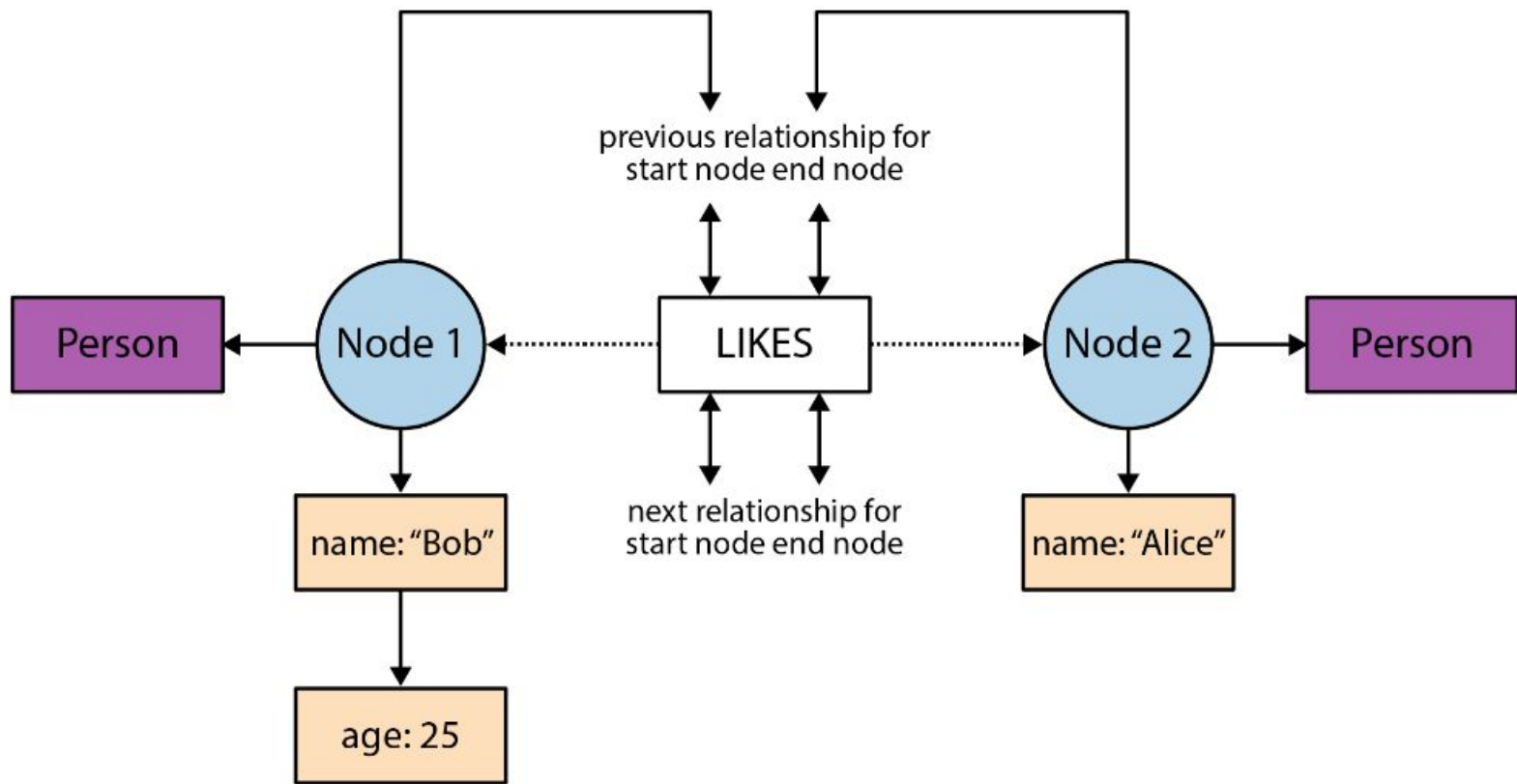
<id>	3	ⓘ
name	123	ⓘ

Wierzchołki grafu trzymane są w 15-bajtowych blokach zawierających wskaźnik do pierwszego bloku z relacją, wskaźnik do bloku z pierwszym atrybutem, wskaźnik do metek i identyfikator.

Krawędzie grafu są przechowywane w 32-bajtowych blokach zawierających wskaźniki do wierzchołka początkowego i końcowego, typu relacji oraz następnych relacji na listach krawędzi wierzchołków.

Ponieważ nie ma wymuszonej struktury danych (nie istnieją kolumny jak w bazach relacyjnych), to atrybuty wierzchołków/krawędzi trzymane są w listach wiązanych. Blok atrybutu zajmuje standardowo 41 bajtów, z wyjątkiem atrybutów tekstowych i tablicowych, których bloki mają 128 bajtów.

W przypadku atrybutów indeksowanych, tworzone są dodatkowe bloki potrzebne do zachowania indeksu.



Źródło: Graph Databases - Ian Robinson, Jim Webber, Emil Eifrem

Cypher Query Language

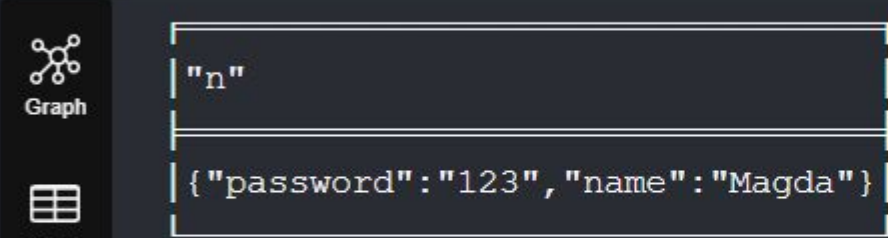
MATCH, RETURN, WITH, UNWIND, WHERE, ORDER BY, SKIP, LIMIT, CREATE,
DELETE, SET, REMOVE, FOREACH, MERGE, CALL, UNION, USE ...

Cypher był inspirowany SQL-em i jest bardzo do niego podobny, dzięki czemu mając doświadczenie z systemami relacyjnymi, możemy łatwo zacząć korzystać z Neo4j.

<code>select * from users</code>	<code>match (n:User) return n</code>
<code>select name from users</code>	<code>match (n:User) return n.name</code>
<code>select * from users where name = 'Tom'</code>	<code>match (n:User) where n.name = "Tom" return n</code>
<code>select * from users where name = 'Tom'</code>	<code>match (n:User {name: "Tom"}) return n</code>
<code>select users.name, phones.number from users join phones on users.phone_no = phones.number</code>	<code>match (u:User)-[:has_phone]->(p:Phone) return u.name, p.number</code>

Tworzenie wierzchołków

```
neo4j$ match (n:User {name: "Magda"}) return n
```



The interface shows a query result in a table format. The table has two rows. The first row contains the variable 'n'. The second row contains a JSON object representing the user Magda.

"n"
{"password": "123", "name": "Magda"}

On the left side of the interface, there are three view options: Graph (with a network icon), Table (with a grid icon), and Text (with a large 'A' icon). The 'Text' option is currently selected and highlighted.

CREATE

```
neo4j$ create (:User {name: "Magda"})
```

```
neo4j$ match (n:User {name: "Magda"}) return n
```



Graph



Table



Text

"n"
{ "password": "123", "name": "Magda" }
{ "name": "Magda" }

MERGE

```
neo4j$ merge (:User {name: "Magda"})
```

```
neo4j$ match (n:User {name: "Magda"}) return n
```



Graph



Table



Text

"n"
{ "password": "123", "name": "Magda" }

Edytowanie atrybutów wierzchołków

```
neo4j$ match (u:User {name: "Magda"}) set u.password = "456" return u
```



Graph



Table



Text

"u"

{"password": "456", "name": "Magda"}



Graph



Table



Text

"u"

{"name": "Magda"}

Usuwanie wierzchołków

```
neo4j$ match (u:User {name: "Magda"}) delete u
```

```
neo4j$ match (u:User {name: "Magda"}) create (u)-[:has]→(f:Fruit {name: "apple"})
```

ERROR **Neo.ClientError.Schema.ConstraintValidationFailed**

Cannot delete node<6>, because it still has relationships. To delete this node, you must first delete its relationships.

```
neo4j$ match (u:User {name: "Magda"}) detach delete u
```



Table

Deleted 1 node, deleted 1 relationship, completed after 3 ms.

Tworzenie krawędzi

```
neo4j$ match (t:Person {name: "Tom"}) match (a:Person {name: "Amy"}) create (t)-[:likes]→(a)
```

```
neo4j$ match (p:Person) return p
```



Graph



Table



Text



```
neo4j$ match (t:Person {name: "Tom"}) match (a:Person {name: "Amy"}) merge (t)-[:likes]→(a)
```

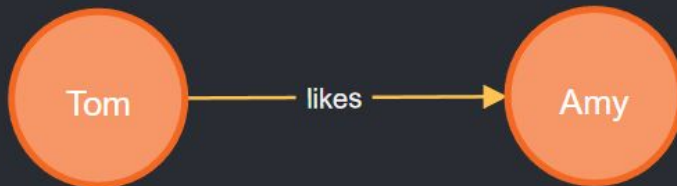
```
neo4j$ match (p:Person) return p
```



Graph



Table



Usuwanie krawędzi

```
neo4j$ match (t:Person {name: "Tom"})-[l:likes]-(a:Person {name: "Amy"}) delete l
```



Table

Deleted 1 relationship, completed after 2 ms.

```
neo4j$ match ()-[l:likes]-() delete l
```



Table

Deleted 1 relationship, completed after 6 ms.

Tworzenie indeksów

- lookup index
- range index
- text index
- point index
- full-text index

```
1 CREATE RANGE INDEX person_id_index
2 FOR (p:Person)
3 ON (p.name)
```

neo4j\$ show all indexes

	id	name	state	populationPercent	uniqueness	type	entityType	labelsOrTypes	properties	indexProvider
1	1	"index_343aff4e"	"ONLINE"	100.0	"NONUNIQUE"	"LOOKUP"	"NODE"	<i>null</i>	<i>null</i>	"token-lookup-1.0"
2	2	"index_f7700477"	"ONLINE"	100.0	"NONUNIQUE"	"LOOKUP"	"RELATIONSHIP"	<i>null</i>	<i>null</i>	"token-lookup-1.0"
3	3	"person_id_index"	"ONLINE"	100.0	"NONUNIQUE"	"RANGE"	"NODE"	["Person"]	["name"]	"range-1.0"

Constraints

```
neo4j$ create constraint for (p:Person) require p.pesel is not null
```

```
neo4j$ CREATE CONSTRAINT for (p:Person) require p.pesel IS UNIQUE
```

```
neo4j$ show all constraints
```



Table



Text



Code

	id	name	type	entityType	labelsOrTypes	properties	ownedIndexId
1	5	"constraint_32ccf3e"	"UNIQUENESS"	"NODE"	["Person"]	["pesel"]	4
2	6	"constraint_d0b7d448"	"NODE_PROPERTY_EXISTENCE"	"NODE"	["Person"]	["pesel"]	<i>null</i>

neo4j\$ show all indexes

	id	name	state	populationPercent	uniqueness	type	entityType	labelsOrTypes	properties	indexProvider
1	4	"constraint_32ccf3e"	"ONLINE"	100.0	"UNIQUE"	"BTREE"	"NODE"	["Person"]	["pesel"]	"native-btree-1.0"
2	1	"index_343aff4e"	"ONLINE"	100.0	"NONUNIQUE"	"LOOKUP"	"NODE"	<i>null</i>	<i>null</i>	"token-lookup-1.0"
3	2	"index_f7700477"	"ONLINE"	100.0	"NONUNIQUE"	"LOOKUP"	"RELATIONSHIP"	<i>null</i>	<i>null</i>	"token-lookup-1.0"
4	3	"person_id_index"	"ONLINE"	100.0	"NONUNIQUE"	"RANGE"	"NODE"	["Person"]	["name"]	"range-1.0"

Dodanie unique constraint powoduje utworzenie indeksu.

Cypher w praktyce

Nasza aplikacja

Zaprojektujemy platformę społecznościową z systemem rekomendacji. “Użytkownicy” będą mogli posiadać znajomych, brać udział w “wydarzeniach” oraz odwiedzać “miejsca”.

Utworzenie wierzchołków

Na początek utworzymy kilka przykładowych wierzchołków.

Person - wierzchołek z danymi użytkownika

Place - wierzchołek z danymi danego miejsca, w tym jego położenie - specjalny typ danych Point.

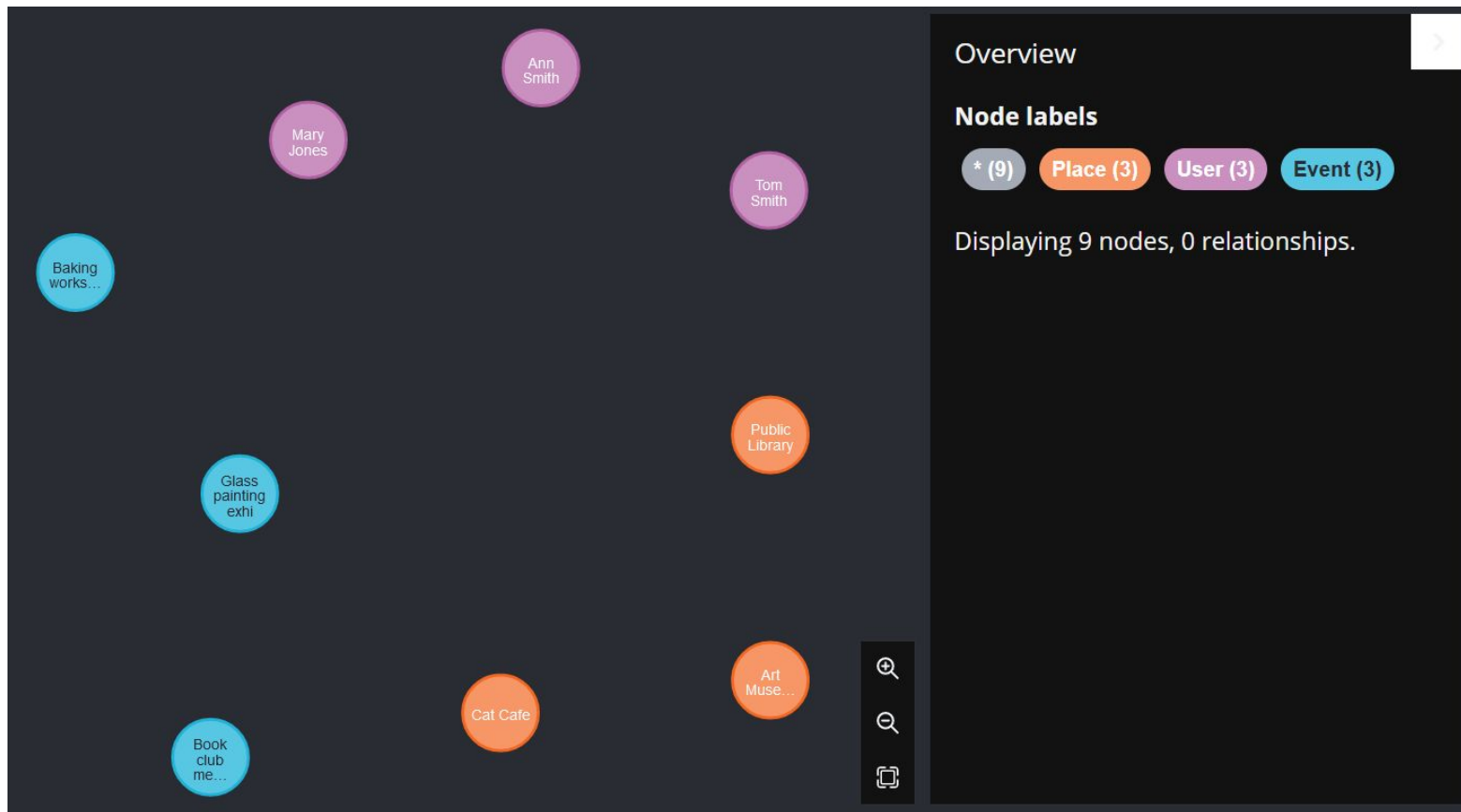
Event - wierzchołek z danymi wydarzenia, w tym czas odbywania - specjalny typ danych DateTime.

```
create (:Person {id: "1", name: "Tom Smith"}), (:Person {id: "2",  
name: "Ann Smith"}), (:Person {id: "3", name: "Mary Jones"})
```

```
create (:Place {id: "1", name: "Cat Cafe", location: point({latitude:  
50, longitude: 20}}}), (:Place {id: "2", name: "Art Museum", location:  
point({latitude: 51, longitude: 20}}}), (:Place {id: "3", name: "Public  
Library", location: point({latitude: 51, longitude: 21}}})
```

```
create (:Event {id: "1", name: "Baking workshop", startTime:  
datetime({year: 2023, month: 1, day: 10, hour: 12, minute: 30,  
timezone: "Europe/Warsaw"}), endTime: datetime({year: 2023,  
month: 1, day: 10, hour: 17, minute: 30, timezone:  
"Europe/Warsaw"}}}), (:Event {id: "2", name: "Glass painting  
exhibition", startTime: datetime({year: 2023, month: 1, day: 11,  
hour: 18, timezone: "Europe/Warsaw"}), endTime: datetime({year:  
2023, month: 1, day: 11, hour: 23, timezone: "Europe/Warsaw"}}}),  
(:Event {id: "3", name: "Book club meeting", startTime:  
datetime({year: 2023, month: 1, day: 10, hour: 16, minute: 30,  
timezone: "Europe/Warsaw"}), endTime: datetime({year: 2023,  
month: 1, day: 10, hour: 18, minute: 30, timezone:  
"Europe/Warsaw"}}})
```

```
match (n) return n
```



Utworzenie indeksów

Pierwsze sześć poleceń tworzy constraints tak, aby identyfikatory były unikalne w obrębie typu wierzchołka.

Przedostatnie polecenie tworzy index na położeniu geograficznym Place.

Ostatnie tworzy indeks do przeszukiwania Event-ów na podstawie czasu ich rozpoczęcia.

```
create constraint for (u:User) require u.id is not null
```

```
create constraint for (u:User) require u.id is unique
```

```
create constraint for (u:Event) require u.id is not null
```

```
create constraint for (u:Event) require u.id is unique
```

```
create constraint for (u:Place) require u.id is not null
```

```
create constraint for (u:Place) require u.id is unique
```

```
create point index for (p:Place) on p.location
```

```
create range index for (e:Event) on e.startTime
```


Dodanie krawędzi

Pierwszy zestaw poleceń dodaje połączenia między wydarzeniem a miejscem jego odbywania się.

Drugi zestaw poleceń dodaje połączenia między użytkownikami a wydarzeniami, w których uczestniczyli.

Polecenie w trzecim bloku tworzy połączenie mówiące o miejscu odwiedzionym przez użytkownika bezpośrednio.

Ostatni zestaw poleceń tworzy połączenia oznaczające znajomość dwóch użytkowników z datą dodania się do znajomych.

```
match (e:Event {id: "1"}), (p:Place {id: "1"}) create (e)-[:happens_at]->(p);
match (e:Event {id: "2"}), (p:Place {id: "2"}) create (e)-[:happens_at]->(p);
match (e:Event {id: "3"}), (p:Place {id: "3"}) create (e)-[:happens_at]->(p);
```

```
match (e:Event {id: "1"}), (p:User {id: "1"}) create (e)<-[:attends]-(p);
match (e:Event {id: "2"}), (p:User {id: "2"}) create (e)<-[:attends]-(p);
match (e:Event {id: "3"}), (p:User {id: "3"}) create (e)<-[:attends]-(p);
match (e:Event {id: "3"}), (p:User {id: "1"}) create (e)<-[:attends]-(p);
match (e:Event {id: "2"}), (p:User {id: "3"}) create (e)<-[:attends]-(p);
```

```
match (p:Place {id: "1"}), (u:User {id: "2"}) create (p)<-[:visits]-(u)
```

```
match (u1:User {id: "1"}), (u2:User {id: "2"}) create (u1)-[:knows {since:
datetime({year: 2020, month: 1, day: 11})}]->(u2)
```

1 Rejestracja użytkownika

2 Dodanie do znajomych

3 Utworzenie wydarzenia

4 Dołączenie do dwóch
wydarzeń jednocześnie*

*(mało realistyczna sytuacja, ale pokazuje możliwość
tworzenia większej ilości krawędzi)

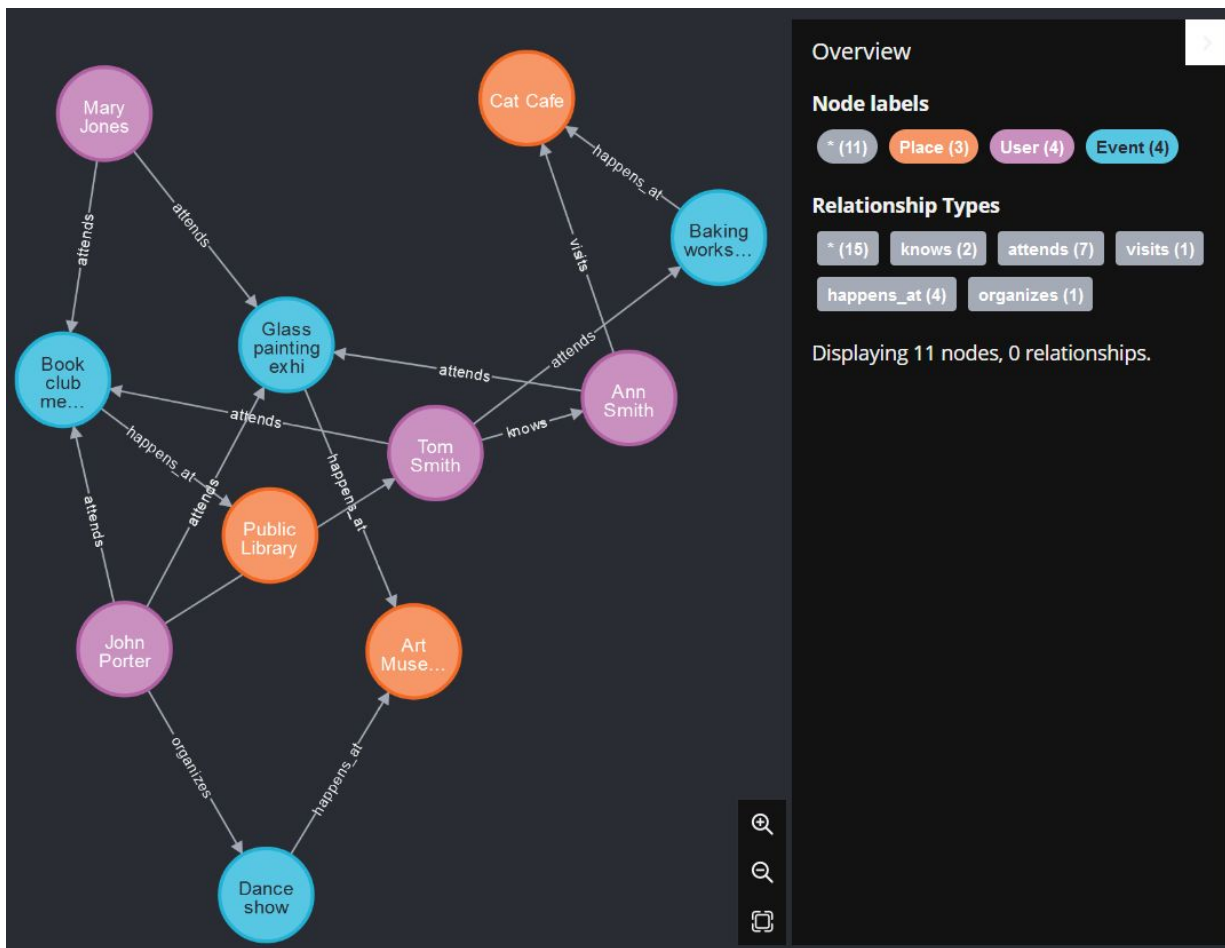
```
create (u:User {id: "4", name: "John Porter"})
```

```
match (u1:User {id: "4"}), (u2:User {id: "1"}) create (u1)-[:knows {since:  
datetime({year: 2023, month: 1, day: 11})}]->(u2)
```

```
match (u:User {id: "4"}), (p:Place {id: "2"}) create (e:Event {id: "4", name:  
"Dance show", startTime: datetime({year: 2023, month: 1, day: 20, hour:  
19, minute: 30, timezone: "Europe/Warsaw"}), endTime: datetime({year:  
2023, month: 1, day: 20, hour: 21, minute: 30, timezone:  
"Europe/Warsaw"}) })<-[:organizes]-(u)-[:happens_at]->(p)
```

```
match (u:User {id: "4"}), (e1:Event {id: "2"}), (e2:Event {id: "3"})  
create (e1)<-[:attends]-(u)-[:attends]->(e2)
```

match (n) return n



Proponowani znajomi

Znalezienie osób z którymi ma się wspólnych znajomych.

Jednak pierwsze zapytanie może zwracać osoby, które są już naszymi znajomymi, np. kiedy A zna B, B zna C a C zna A, to A i C zostaną sobie zaproponowani ze względu na ścieżkę A-B-C.

Aby temu zapobiec można użyć wbudowanej procedury do znalezienia ścieżek podanej długości, lub zastosować własną implementację.

```
match (u:User {id: "2"})-[:knows*2..3]-(f:User) return distinct f
```

```
match (p:User {id: "4"}) call apoc.neighbors.athop(p, "knows", 2)  
yield node return node
```

```
match (u:User {id: "2"})-[:knows]-(f:User)-[:knows]-(p:User)  
with collect(distinct f.id) as direct_friends, p  
where not p.id in direct_friends  
return p
```

Proponowani znajomi cd.

Posiadanie wspólnych znajomych to jeden ze sposobów na znajdowanie nowych. Innym pomysłem jest proponowanie na podstawie wydarzeń w których się razem uczestniczyło.

Pierwsze zapytanie zwróci dwa wierzchołki, pomimo tego, że użytkownik o id="4" jest zwracany dwa razy, ponieważ baza automatycznie usuwa duplikaty w przypadku zwracania pojedynczego wierzchołka. Jednak lepiej użyć drugiego zapytania, ponieważ w przypadku dalszego przetwarzania rezultatu (przed zwróceniem) pierwsze zapytanie może prowadzić do redundancji i powielonych danych.

```
match (u:User {id: "2"})-[:attends]->(e:Event)<-[:attends]-(p:User)
return p
```

```
match (u:User {id: "2"})-[:attends]->(e:Event)<-[:attends]-(p:User)
return distinct p
```

Można podejrzeć listę ścieżek zwracaną przez bazę przez przypisanie ścieżki do zmiennej.

```
match path = ((u:User {id:
"3"})-[:attends]->(e:Event)<-[:attends]-(p:User)) return path
```

```
neo4j$ match path = ((u:User {id: "3"})-[:attends]→(e:Event)←[:attends]-(p:User)) return path
```



Graph



Table



Text



Code

```
"path"
```

```
[{"name":"Mary Jones","id":"3"},{},{"name":"Glass painting exhibition",
"startTime":"2023-01-11T18:00:00[Europe/Warsaw]","endTime":"2023-01-11T23:00:00[Europe/Warsaw]","id":"2"},{"name":"Glass painting exhibitio
n","startTime":"2023-01-11T18:00:00[Europe/Warsaw]","endTime":"2023-01-11T23:00:00[Europe/Warsaw]","id":"2"},{},{"name":"John Porter","id":"4"}]
```

```
[{"name":"Mary Jones","id":"3"},{},{"name":"Glass painting exhibition",
"startTime":"2023-01-11T18:00:00[Europe/Warsaw]","endTime":"2023-01-11T23:00:00[Europe/Warsaw]","id":"2"},{"name":"Glass painting exhibitio
n","startTime":"2023-01-11T18:00:00[Europe/Warsaw]","endTime":"2023-01-11T23:00:00[Europe/Warsaw]","id":"2"},{},{"name":"Ann Smith","id":"2"}]
```

```
[{"name":"Mary Jones","id":"3"},{},{"name":"Book club meeting","startT
ime":"2023-01-10T16:30:00[Europe/Warsaw]","endTime":"2023-01-10T18:30:00[Europe/Warsaw]","id":"3"},{"name":"Book club meeting","startTime":"2023-01-10T16:30:00[Europe/Warsaw]","endTime":"2023-01-10T18:30:00[Europe/Warsaw]","id":"3"},{},{"name":"John Porter","id":"4"}]
```

```
[{"name":"Mary Jones","id":"3"},{},{"name":"Book club meeting","startT
ime":"2023-01-10T16:30:00[Europe/Warsaw]","endTime":"2023-01-10T18:30:00[Europe/Warsaw]","id":"3"},{"name":"Book club meeting","startTime":"2023-01-10T16:30:00[Europe/Warsaw]","endTime":"2023-01-10T18:30:00[Europe/Warsaw]","id":"3"},{},{"name":"Tom Smith","id":"1"}]
```

Proponowani znajomi cd.

Jak wybrać najlepsze propozycje w przypadku, gdy wejściowy użytkownik wziął udział w dużej ilości wydarzeń i/lub z dużą ilością osób?

Jednym z rozwiązań jest posortowanie osób według ilości wspólnych wydarzeń i wybranie kilku początkowych.

```
match (u:User {id: "2"})-[:attends]->(e:Event)<-[:attends]-(p:User)
return p
```

Powyższe zapytanie zwróci dwa wierzchołki, pomimo tego, że użytkownik o id="4" jest zwracany dwa razy, ponieważ baza automatycznie usuwa duplikaty w przypadku zwracania pojedynczego wierzchołka. Jednak lepiej użyć poniższego zapytania, ponieważ w przypadku dalszego przetwarzania rezultatu (przed zwróceniem) pierwsze zapytanie może prowadzić do redundancji i powielonych danych.

```
match (u:User {id: "2"})-[:attends]->(e:Event)<-[:attends]-(p:User)
return distinct p
```

```
match (u:User {id: "3"})
match (u)-[:attends]->(e:Event)<-[:attends]-(p:User)
return distinct p, count(distinct e) as event_count order by
event_count desc
limit 10
```

```
1 match (u:User {id: "3"})
2 match (u)-[:attends]->(e:Event)<-[:attends]-(p:User)
3 return distinct p, count(distinct e) as event_count order by event_count desc
```



"p"	"event_count"
{ "name": "John Porter", "id": "4" }	2
{ "name": "Ann Smith", "id": "2" }	1
{ "name": "Tom Smith", "id": "1" }	1

Proponowane wydarzenia

Będziemy szukać nadchodzących wydarzeń, w których udział lub organizację zadeklarowali znajomi.

Ponieważ Cypher nie ma wbudowanego mechanizmu do wygenerowania aktualnego timestamp-u, podajemy my obecną datę.

```
match (u:User {id: "1"})
match (u)-[:knows]-(f:User)-[:attends|organizes]->(e:Event)
where e.startTime > datetime({year: 2023, month: 1, day: 11})
with e, count(f) as friends_count
return e, friends_count order by friends_count desc
limit 10
```


Zadanie: proponowanie wydarzeń

point.distance(p1, p2) to funkcja zwracająca odległość między dwoma punktami.

Dla współrzędnych kartezjańskich zwraca wynik w jednostkach używając wzoru Pitagorasa.

Dla współrzędnych geograficzny wynik zwracany jest w metrach.

Mając podane aktualne współrzędne użytkownika jak znaleźć wydarzenia w odległości do pięciu kilometrów?

Rozwiązanie 1

Zapytanie przedstawione obok zwróci wszystkie nadchodzące wydarzenia w pobliżu.

```
match (p:Place)
where point.distance(p.location, point({latitude: 50, longitude: 20}))
< 5000
match (p)<-[:happens_at]-(e:Event)
where e.startTime > datetime({year: 2023, month: 1, day: 10})
return e
```

Widzimy jednak, że pierwszą wykonywaną operacją jest *NodeByLabelScan*, co może być problematyczne przy większej ilości danych.

Jak w inny sposób znajdować polecane wydarzenia, aby uniknąć operacji *Scan*?

```
1 explain match (p:Place)
2 where point.distance(p.location, point({latitude: 50, longitude: 20})) < 5000
3 match (p)←[:happens_at]-(e:Event)
4 where e.startTime > datetime({year: 2023, month: 1, day: 10})
5 return e
```

The diagram illustrates the execution plan for the provided Cypher query. It shows a sequence of operators connected by arrows, representing the flow of data. The operators are:

- NodeByLabelScan@neo4j**: The first operator, which scans for nodes of type `Place`. It has 10 estimated rows.
- Filter@neo4j**: The second operator, which filters the results based on the distance condition. It has 3 estimated rows.
- Expand(Ali)@neo4j**: The third operator, which expands the results to include the `happens_at` relationship. It has 1 estimated row.
- Filter@neo4j**: The fourth operator, which filters the results based on the `startTime` condition. It has 0 estimated rows.

The diagram also shows the variables `p` and `e` being used in the operators. The `Expand(Ali)@neo4j` operator shows the relationship `(p)←[:happens_at]-(e)`. The `Filter@neo4j` operators show the conditions `point.distance(p.location, point({latitude: $autoint_0, longitude: $autoint_1})) < $autoint_2` and `e.startTime > datetime({year: $autoint_3, month: $autoint_4, day: $autoint_5}) AND e:Event`.

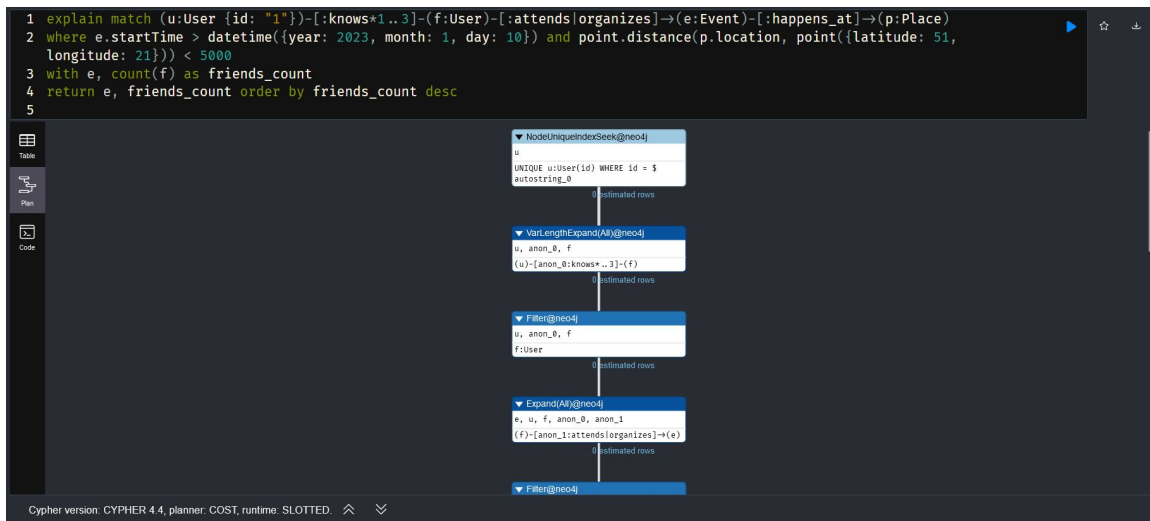
Cypher version: CYPHER 4.4, planner: COST, runtime: SLOTTED. ⚙️ ⚙️

Rozwiązanie 2

Takie zapytanie wyszukuje wydarzeń, w których uczestnictwo zadeklarowali znajomi użytkownika, ich znajomi itd. do trzeciego stopnia znajomości, a dopiero później sprawdza warunek odległości.

W planie wykonania widzimy, że pierwszą wykonaną operacją będzie teraz *NodeUniqueIndexSeek* a następnie *Expand*.

```
match (u:User {id:
"1"})-[:knows*1..3]-(f:User)-[:attends|organizes]->(e:Event)-[:happens_at]-(p:Place)
where e.startTime > datetime({year: 2023, month: 1, day: 10}) and
point.distance(p.location, point({latitude: 51, longitude: 21})) < 5000
with e, count(f) as friends_count
return e, friends_count order by friends_count desc
```



Źródła

[Blank Nodes and UID - Mutations](#)

Graph Databases - Ian Robinson, Jim Webber, Emil Eifrem

[Neo4j storage internals](#)

[Will It Graph? Identifying a Good Fit for Graph Databases – Part 1](#)

[Understanding Neo4j's data on disk - Knowledge Base](#)

[An Introduction to Graph Databases | by John Clarke | Towards Data Science](#)

[17 Use Cases for Graph Databases and Graph Analytics](#)

Dgraph: Synchronously Replicated, Transactional and Distributed Graph Database - Manish Jain

[Labels, Constraints and Indexes - graphgists](#)