İLERI CYPHER SORGULARI YAZARAK USE CASELER OLUŞTURMAK

Bu ödev için geçen hafta ödevimiz için yüklediğim cvs dosyamı tekrar import ediyorum. Öncelikle bu derste gördüğümüz temel başlıkları çıkardım.

1. Basic Cypher Queries
    - Filtering with WHERE(Testing equality,inequality,ranges,null property values,labels or patterns,list inclusion
    - Return->finding relationships type (exp:tyepe(r))),less than or greater than(T/F)
2. Controlling Results Returned
    - Ordering results(Is not null ile kullanımı),multiple ordering
    - Limiting results
    - Eliminating duplicate records(DISTINCT)
    - Map projections(CONTAINS->p-node, p{.*}-all prop.,p{.name} m{.*,fav:true}->döndürülen her m nesnesine yeni özellik ekler
    - Changing results returned->string or numeric operations|case-when/then-else-end
3. Working with Cypher Data
    - Agrregating data->count(n|*),collect(p),collect(a.name)[2..],size(collect(a.name)) Min(),sum(),avg(),stddev(),sum()
      Explain-> Query plan Profile->Query plan+Runtime (Zaman ve maliyetide görürüz)
      Comprehensions->list : [x IN m.lang WHERE x CONTAINS ‘EN’ OR ‘TR’]
      pattern:[<pattern>| value>
      Pattern comphression to create a list->Exp: [(a)-->(b:Movie) WHERE b.title CONTAINS "Toy" | b.title + ": " + b.year]
    - Dates and Times->date(), datetime(), time() , x.datetime.hour, x.date.day... duration.between(,) , duration.inDays(,).days, + duration({months: 6})... APOC lib
4. Graph Traversal
    - Filtreler mümkün oldukça erken yazılmalı(performans ıcın grafik üzerinde daha az dolasım yapmak adına), daha sonrada ilişkiler özgül olmalı
    - Daha iyi performans için etiketlerden kaçınma
    - Varying Length Traversal:Shortest path algo., [:r*2] ,[:r*1..4]
5. Pipelining Queries
    - Scoping variables: WITH:Sorgu içinde veri taşıma,bölme (Limit aldığımız değerleri başka sorguda kullanma vb gibi) Map projections yapabiliriz
    - Unwind: listedeki değerleri ayırır ve ayrı ayrı sonuç olarak görmek için kullanabiliriz
6. Reducing Memory
    - Subqueries:CALL->Sorguları alt soru şeklinde bölüp sonucu daha sonra kullanırız.
    - UNION:Subqueries sorgularını birleştirmek için kullanılır. Burada subquerilerin Return de alliasları aynı verilerek kullanılır. UNION ALL da aynı işlevde kullanılır tek farkı duplicate olma durumu olabilir.

7. Using Parameters
    - :param actorName:’Tom’ -> WHERE p.name=$actorName
    - :param number:10->default float
    - :param number=>10->integer olması için
    - Multiple parameters ->:params {actorName: 'Tom Cruise', movieName: 'Top Gun'}
    - :params -> Parametreleri görmek için

Genel olarak Cypher'da sorgular yazılırken filtreleme işlemlerinin mümkün olduğunca erken yapılması, ilişki türlerinin spesifik verilmesi ve gereksiz etiket kullanımından kaçınılması önerilir. Varying length traversals, comprehension yapıları ve aggregation fonksiyonları gibi özellikler de grafik veri üzerinde etkili sorgular yazmayı sağlar. EXPLAIN ve PROFILE komutları ise sorgunun maliyeti ve çalışma süreci hakkında fikir verir. Özellikle PROFILE, sorgunun çalışma zamanını da göstererek optimize edilmesi gereken noktaları belirlemede yardımcı olur.

Neo4j'de Cypher dilini kullanırken temel sorgulardan sonra daha verimli, okunabilir ve performanslı sorgular yazabilmek için bazı ileri seviye yapıları öğrendik. WITH, sorgular arasında veri taşımak, filtreleme yapmak ve sorguyu mantıksal parçalara bölmek için kullanılan güçlü bir yapıdır. Özellikle gruplama, sıralama, sınırlandırma (LIMIT) gibi işlemlerden sonra veriyi taşımak ve sadece ilgili değişkenleri bir sonraki adıma aktarmak için idealdir. Ayrıca, map projection yöntemiyle veriye yeni özellikler eklemek için de kullanılabilir. CALL ise alt sorgular (subqueries) yazmak için kullanılır; genellikle karmaşık işlemler, gruplanmış veri üzerinde işlem yapılacak durumlar ya da harici prosedürler (örneğin APOC kütüphanesi) çağrılırken tercih edilir. Bu yapı, sorgunun hem okunabilirliğini artırır hem de büyük sorguları modüler hale getirir. CALL { ... } yapısı içinde genellikle WITH ile veri taşınarak alt sorguda tekrar kullanılır. Bununla birlikte dışarıdan veri alıp sorguya dahil etmek gerektiğinde, örneğin kullanıcıdan gelen arama terimleri veya filtreler gibi, :param yapısı kullanılarak sorgular dinamik hale getirilir. Parametre kullanımı aynı zamanda SQL injection gibi güvenlik açıklarını önler ve sorgu önbelleği (query plan caching) sayesinde performansı artırır.

Ben geçen haftaki sorgularımda WITH ile mapping yaptım , DISTINCT ,ORDER BY ile sonuçlarıma kontrol ekledim mümkün olukça erken node filtrelemeler ile performansıma dikkat ettim, aggregating kullandım, collect ile özelliklerden list olusturmayı yaptım. case-when/then-else-end yapısınıda kullandım.

Bu hafta ek olarak Explain ve Profile özelliğini datasetimde deneyeceğim.Unwind için örnek bir usecase oluşturcağım. Ve call ,wıth ,union ve param yapısını kullanrak daha ileri seviye bir sorgu denemesi yapmış olacağım.

Bu kısımı analiz etmek içinde son usecase'im üzerinden iyileştirme yaparak başlamayı düşündüm:

Aynı kalan üyeleri birlikte mi hareket ediyor?

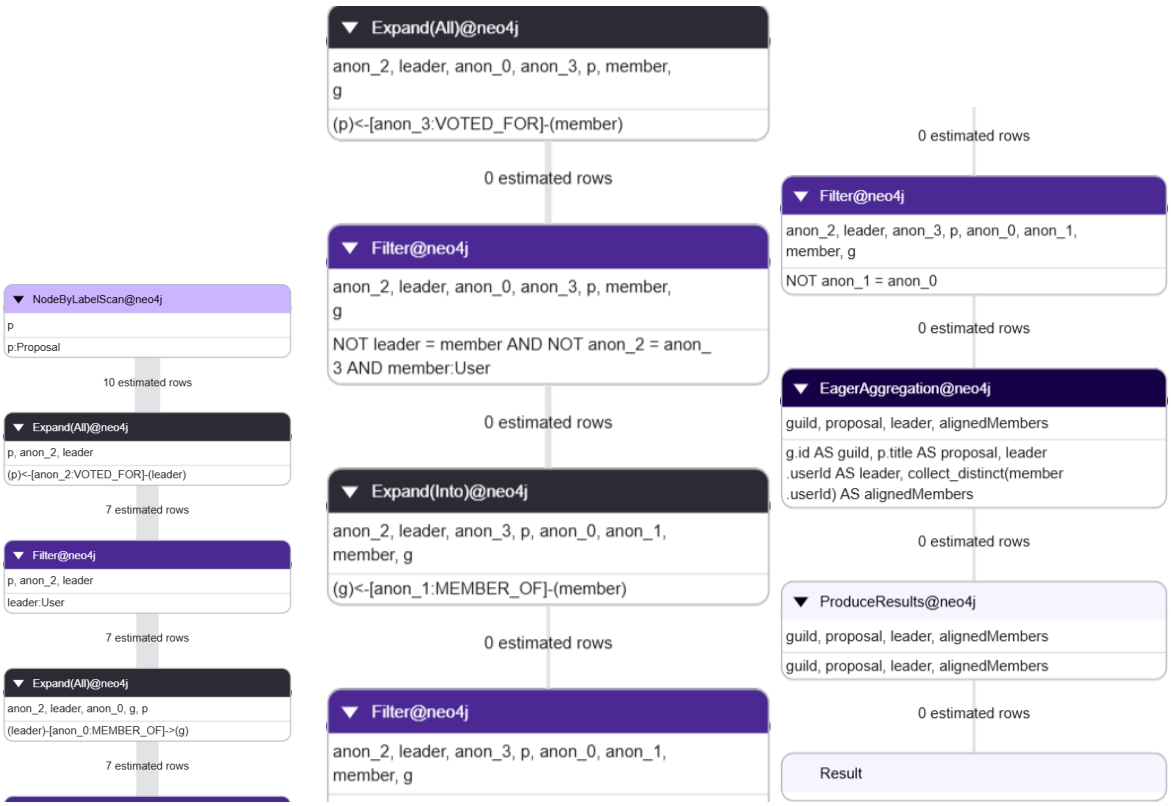
```
MATCH (leader:User)-[:MEMBER_OF {role: "Leader"}]->(g:Guild)<-[:MEMBER_OF]-(member:User),
      (leader)-[:VOTED_FOR]->(p:Proposal),
      (member)-[:VOTED_FOR]->(p)
WHERE leader <> member
RETURN g.id AS guild,
       p.title AS proposal,
       leader.userId AS leader,
       collect(DISTINCT member.userId) AS alignedMembers

```

No changes, no records

Burada görüldüğü üzere MATCH içinde birden fazla sorgu bulunuyor . Önce bu sorgu için Profile ve Explain yapıp performans hakkında gözlem yapıyorum . Daha sonrasında çoklu sorguyu CALL ,WITH ,UNION bırlıkte kullanarak denemeler yapacağım.

EXPLAIN MATCH (leader:User)-[:MEMBER_OF {role: "Leader"}]->(g:Guild)<-[:MEMBER_OF]-(member:User), (leader)-[:VOTED_FOR]->(p:Proposal), (member)-[:VOTED_FOR]->(p) WHERE leader <> member RETURN g.id AS guild, p.title AS proposal, leader.userId AS leader, collect(DISTINCT member.userId) AS alignedMembers ile sorgu çalıştırma planının detaylı analizini inceliyorum.

- Sorgu şu anda tüm Proposal düğümlerini tarayarak başlıyor
- Daha spesifik bir başlangıç noktası oluşturabilirim.



Profile ile de runtime sonuçlarını alıyorum.

Temel Performans Metrikleri

- Toplam Çalışma Süresi: 44 ms
- Toplam DB Hits: 73
- Bellek Kullanımı: ~1.5KB (1,088 bytes + 432 bytes)

Öncelikle with ve pram ile modüler ve dinamik bir sorgu oluşturmayı deniyorum .

```
:param minMemberCount => 1
MATCH (g:Guild)<-[:MEMBER_OF {role: "Leader"}]-(leader:User)
WITH g, leader
MATCH (leader)-[:VOTED_FOR]->(p:Proposal)
WITH g, leader, p

MATCH (member:User)-[:VOTED_FOR]->(p),
      (member)-[:MEMBER_OF]->(g)
WHERE leader <> member
WITH g, p, leader, collect(DISTINCT member.userId) AS alignedMembers
WHERE size(alignedMembers) >= $minMemberCount

RETURN g.id AS guild,
       p.title AS proposal,
       leader.userId AS leader,
       alignedMembers,
       size(alignedMembers) AS alignmentCount
ORDER BY alignmentCount DESC
```

Bu sorguya profile ile baktığımda 73 total db hits in 34 ms. çalışma süremde dataset boyuuma rağmen önemli miktarda düşüş olduğunu görüyorum.

Yeni Use Case:

Kullanıcıların birden fazla ilişkisini (oy verdikleri teklifler ve tuttukları tokenlar gibi) aynı anda sorgulayıp, sonra bu ilişkilerden belli bir kritere göre (örneğin oy verilen teklifler) filtreleme yaparak kullanıcı bazında özet bilgi nasıl elde edilir?

```
MATCH (u:User)
WITH u LIMIT 50

CALL {
  WITH u
  OPTIONAL MATCH (u)-[v:VOTED_FOR]->(p:Proposal)
  RETURN p.title + ": Voted" AS info
  UNION
  WITH u
  OPTIONAL MATCH (u)-[h:HOLDS_TOKEN]->(t:Token)
  RETURN t.name + ": Holds Token" AS info
}

WITH u, collect(info) AS details

// details listesini aç, içinde "Voted" geçenleri filtrele
UNWIND details AS detail
WITH u, detail
WHERE detail CONTAINS "Voted"

RETURN u.name, collect(detail) AS votedDetails
```

Table | RAW

	u.name	votedDetails
1	"NovaByte82"	["Predictive Analytics for Passenger Experience Enhancement: Voted"]
2	"PixelKing47"	["Smart Airport Experience with AI: Voted"]
3	"GhostFury8"	["Real-Time Flight Data with A

Bu sorguda öncelikle `User` düğümlerinden 50 kişiyi sınırlayıp `WITH` ifadesiyle sonraki işlemler için tutuyorum. Ardından her bir kullanıcı için `CALL` alt sorgusunu kullanarak iki farklı ilişkiden bilgi topluyorum: Birincisi, kullanıcının oy verdiği tüm `Proposal` tekliflerini `VOTED_FOR` ilişkisiyle bulup bunları `"<Teklif Başlığı>: Voted"` formatında alıyorum; ikincisi ise kullanıcının sahip olduğu `Token`ları `HOLDS_TOKEN` ilişkisi üzerinden `"<Token Adı>: Holds Token"` formatında topluyorum. Bu iki farklı alt sorgudan gelen sonuçları `UNION` ile birleştiriyorum ve her iki sorgudan dönen sonuçların alias'ını `info` olarak belirliyorum.

Daha sonra, her kullanıcı için bu `info` değerlerini `collect` fonksiyonuyla bir liste halinde topluyorum ve `details` olarak tutuyorum. `details` listesini `UNWIND` ile açarak, içindeki öğelerden sadece içinde `"Voted"` geçenleri filtreliyorum. Son olarak, her kullanıcı için sadece oy verdiği teklifler listesini toplayıp, `votedDetails` olarak döndürüyorum.

“OPTIONAL MATCH” kullanmamın sebebi, bazı kullanıcıların oy verdiği teklif veya tuttuğu token olmayabilir; yani bu ilişkiler her kullanıcı için zorunlu değil. Eğer sadece “MATCH” kullansaydım ve o ilişkiler olmayan kullanıcılar sorgudan tamamen çıkardı, sonuç eksik kalırdı.