Sosyal Çizgeler İçin Arama Motoru Geliştirilmesi

Erman Yafay ve Selma Tekir

İzmir Yüksek Teknoloji Enstitüsü Bilgisayar Mühendisliği Bölümü 35430 Urla, İzmir
ermanyafay@gmail.com selmatekir@iyte.edu.tr

Özet Sosyal ağlara giderek artan ilgi, beraberinde büyük ölçeklerde bağlantılı veri açığa çıkarmıştır. Bu büyük veriler üzerinde arama yapabilmek için özelleştirilmiş sistemlere gereksinim duyulmaktadır. Bu gereksinimi karşılamak üzere Facebook, 2013 yılında kendi arama motoru olan Unicorn'u[1] hizmete sunmuştur. Bu çalışmada, Unicorn'un asgari fakat temel özellikleri tasarlanıp gerçekleştirilmiştir. Yaklaşımımızda sosyal ağ bir çizge olarak modellenmiştir ve çizgedeki düğümler ve kenarlar farklı türlere sahip olabilecek şekilde genel olarak tanımlanmıştır. Düğümler, kişi veya sayfa gibi varlıkları ifade ederken; kenarlar, düğümler arasındaki arkadaşlık veya beğenme ilişkisini ortaya koyar. Verimlilik sorununu çözebilmek için tamamen bellek üzerinde çalışan bir indisleme sistemi geliştirilmiştir. Bu sistem geniş ölçekte veri işlenmesini sağlamak üzere geliştirilen dağıtık motor Spark[2] üzerinde gerçekleştirilmiştir. Son olarak, sosyal ağ yapısına uygun işleçler (ve, veya, zayıf- ve, güçlü-veya, uygula) tasarlanmıştır. Bu işleçler sayesinde kolayca kişilerin ortak arkadaşları veya arkadaşlarının arkadaşları gibi sorgular ifade edilip çalıştırılabilmektedir. Çalışmanın son bölümünde bu tip bir sistemin gerçekleştirilmesinde dikkate alınması gereken nitelikler, bu niteliklere ilişkin ödünleşimler ve karar mekanizmaları ele alınıp değerlendirilmiştir.

Anahtar Kelimeler: Sosyal ağ, Arama motoru, Bilgi Elde Etme, Dağıtık ve Paralel Sistemler

Giriş

Facebook 2013 yılında yeni çizge arama motoru Unicorn'u aktif hale getirmiştir. Unicorn, sosyal çizgedeki yapılandırılmış veri üzerinde hızlı ve ölçeklenebilir bir şekilde arama yapmayı ve arama sonuçları üzerinde gereksinimler doğrultusunda karmaşık işleçler çalıştırmayı sağlayan özel bir tasarıma sahiptir.

Unicorn temelde bir bilgi elde etme sistemidir, sosyal ağ verisini işlemek üzere özelleşmiştir. Sosyal çizge verisine erişmek ve veriyi sıralamak için etkin bir veri yapısı mevcuttur ve bu yapı üzerinde çalıştırılacak sorgu tasarımı ve gerçekleştirimi fonksiyonel programlama dillerindeki liste işleme altyapısı ve fonksiyonları ile yapılmıştır. Sistem bu sayede binlerce sunucu üzerinde tutulan düğümler (kullanıcılar ve diğer varlıklar) arasındaki trilyonlarca kenar üzerinde etkin bir şekilde arama yapabilmektedir.

Bu çalışmada, Unicorn'un çekirdek arama motoru geliştirilmiştir. Geliştirilen yazılım Unicorn'un asgari fakat temel özelliklerini kapsamaktadır. Unicorn'un karmaşık sistem mimarisine odaklanmak yerine bellek üzerinde indisleme, farklı kenar ve düğüm tiplerini destekleyen çizge veri yapısı oluşturma, veri sıralama stratejileri ve arama sonuçları üzerinde dağıtık bir şekilde çalıştırılabilen karmaşık işleçler gerçekleştirilmiştir. Bu temel özellikler büyük ölçekli sosyal çizge arama motorunun altyapısını oluşturmaktadır.

Geliştirilen sistemde dağıtık veri işleme motoru Spark kullanılmıştır. Çalışmada aynı zamanda sözkonusu temel özelliklerin gerçekleştirimi sırasında karşılaşılan ödünleşimler ve ilgili karar mekanizmaları üzerine bir değerlendirme yapılmıştır.

Metodoloji

Veri Toplama

Sosyal çizge verisini işlemek üzere özelleşmiş arama motoru gerçekleştiriminde ilk olarak üzerinde çalışılacak büyük sosyal çizge verisi belirlenmiştir. SNAP[3] verisetinden, farklı ve büyük çizge verileri sağlanmıştır. Bu sosyal çizgelerde düğümler arasındaki arkadaşlık ilişkisi incelendiğinde yapının seyrek (sparse) olduğu gözlemlenmiştir. Bu sebeple, çizge gerçekleştiriminde komşuluk matrisi yerine komşuluk listesi yaklaşımı kullanılmıştır. Komşuluk listeleri seyrek yapıdaki büyük verinin daha az bellek kaplayacak şekilde işlenmesini mümkün kılmaktadır.

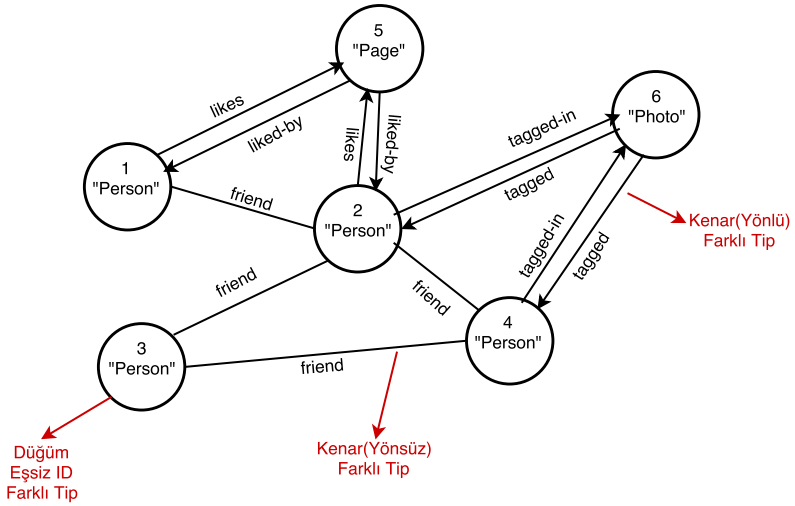
Arama motorlarında döndürülen sonuçların sıralanması (ranking) temel işlevlerden bir tanesidir. Sıralama yapmak üzere farklı ölçütler kullanılabilir. Gerçekleştirdiğimiz yapı içerisinde belli bir ölçüte göre sonuçları sıralı tutma gereksinimini de karşılamak üzere sıralama ölçütü olarak düğümlerin PageRank[4] değeri hesaplanıp komşuluk listesi veri yapısına dahil edilmiştir. Bu sayede PageRank değeri yüksek olan kullanıcılarla arkadaş olan kullanıcılar daha üstte sıralanmaktadır. Bu prestije dayalı sıralama ölçütü, sosyal ağların doğasına da uygundur.

Sosyal çizge arama motoru, anlambilimsel çizgeler üzerinde işlem gerçekleştirebilir. Anlambilimsel çizgelerde düğümler ve kenarlar anlamsal etiketlere sahiptir. Bir başka ifade ile farklı düğüm ve kenar tipleri bulunabilmektedir. SNAP verisetinden sağlanan çizgelerde sadece tek tip düğüm ve kenar bulunduğundan Yelp Academic Dataset Challenge'dan[5] ikinci bir çizge veriseti elde edilmiştir.

Sosyal çizge

Gerekli veri toplanıp analiz edildikten sonra sosyal çizge komşuluk listesi yaklaşımı kullanılarak Spark üzerinde farklı düğüm ve kenar tiplerini destekleyecek şekilde gerçekleştirilmiştir. Tanımlanan farklı tipler herhangi bir varlığı veya ilişkiyi temsil edebileceğinden genel (generic) bir çizge yapısı kurulmuştur. Komşuluk listeleri, kenar tipi-ID ikilileri kullanılarak indislenmiştir.

Bu sayede, bir kullanıcının arkadaşları ya da bir sayfanın beğenenleri gibi verilere kısa sürede ulaşılabilir. Sosyal çizge görseli şekil 1'de verilmektedir.



Şekil 1. Sosyal çizge.

İşleçler

Unicorn makalesinde anlatılan işleçler Spark üzerinde dağıtık ve paralel olacak şekilde gerçekleştirilmiştir. İşleçler sayesinde veri kümesine ait çizgeden pek çok farklı sorgu şekli üretilebilir.

Ters Dizin

Yelp'ten elde edilen veri kümesi içinde pek çok farklı varlık ve ilişki tipi bulunması ve bu varlık ve ilişkileri tanımlayıcı karakter dizilerinin (sözcüklerin) de mevcut olması bu karakter dizilerini temel alan ters dizinlerin yaratılmasını mümkün hale getirmiştir. Sonuç olarak, çizge üzerinde karakter dizisi ile de arama yapılabilmektedir. Örneğin, "Jon" ismine sahip kullanıcılar elde edilebilir.

Typeahead Arama

Son olarak, typeahead arama özelliği gerçekleştirilmiştir. Bu sayede, arama sorgusu sözcüğün ilk karakteri girildiği andan itibaren çalıştırılabilir ve bu karakter ile başlayan sonuçlara erişilebilir. Geliştirilen ters dizin şeması buna uygun hale getirilmiştir.

Sistem Analizi & Tasarımı

Komşuluk Listesi Veri Yapısı

Komşuluk listeleri hem kenar tipi-ID ikilileri hem de ters dizin üzerinden indislenmiştir. Kenar tipi-ID ikilisi temelli indisleme çizgede kenarlar üzerinde hareket

edilmesini sağlar, örneğin bir kullanıcının arkadaşlarının bulunması. Ters dizin yapısı ise belli bir karakter dizisini içeren düğümlerin döndürülmesini destekler.

Komşuluk listesindeki her eleman *Hit* olarak adlandırılır ve *Hit*'ler *DocId* ve seçmeli *HitData* bayt dizisinden oluşmaktadır. *DocId*'ler ondalıklı *sort-key* ve 32-bit tamsayı *id* ikilisinden meydana gelmektedir. *Sort-key* her düğümün sıralama ölçütüne göre aldığı değeri (sistemimizde PageRank değeri), *id* ise düğüm kimliği bilgisini karşılamaktadır. Komşuluk listesi örneği şekil 2'de verilmektedir:

$$List = \{Hit_1, Hit_2, Hit3, \cdots, Hit_n\}$$
$$Hit = (DocId, HitData)$$
$$DocId = (SortKey, Id)$$

sort-key	65	63	23	22	22	13
id	58	64	26	13	55	43
hit-data	10...	10...	10...	10...	10...	10...

Şekil 2. Komşuluk Listesi Veri Yapısı: Listeler önce büyükten küçüğe *sort-key*'e göre daha sonra küçükten büyüğe *id*'ye göre sıralanmıştır. Böylece, sırası daha yüksek olan düğümler kırpma olmadan işlenebilir ve ayrıca yüksek sıralı olanların önce gösterilmesi arama motoru sıralama özelliğine uygundur. Sosyal çizge, Spark üzerinde komşuluk listeleriyle dağıtılmıştır. Yani, şekildeki liste bir kullanıcının ilk parça üzerindeki arkadaşları ise, aynı kullanıcının ikinci ve diğer parçalarda da arkadaşlarının bir kısmı barınıyor olabilir. Bu şekilde bir dağıtım, ölü bir makine olduğu zaman hiçbir şey göstermemektense kullanıcının arkadaşlarının bir kısmını göstermeyi mümkün kılar.

İşleç Tasarımı

İşleçler sosyal çizgedeki bilgiye erişim sağlamak için tasarlanıp gerçekleştirilmiştir. Bu işleçler, *term*, *and (ve)*, *or (veya)*, *weak-and (zayıf-ve)*, *strong-or (güçlü-veya)* ve *apply (uygula)* işleçleridir. Önek simgelemi gösterimindeki işleçlerin sonsuz sayıda zincirlenebilmesi için işleçler *Composite Design Pattern* kullanılarak tasarlanmıştır.

term işleci bir düğüme komşu olan düğüm listesine veya bir karakter dizisini içinde barındıran düğümlere erişimi sağlar:

```
(term friend:5)
```

5 ID'li kullanıcının arkadaşları.

```
(term "michael")
```

Michael sözcüğünü barındıran düğümler.

and (ve) işleci komşuluk listelerinin kesişim kümesinin hesaplanmasını sağlar. Bu işlece ait kod algoritma 1'de verilmiştir.

```
(and (friend:5) (friend:6))
```

5 ve 6 ID'li kullanıcıların ortak arkadaşları.

Algorithm 1 Liste kesişimi, özyinelemeli prosedür *(intersectRec)* kullanılarak gerçekleştirilmiştir. Algoritma, *intersect* prosedürünün, *intersectRec* prosedürünü son parametresi boş bir liste olacak şekilde çağırması ile başlar. Daha sonra, her iki listenin ilk elemanlarından başlanarak, *Hit*'lerin *docId* ikililerinin eşitlikleri karşılaştırılır ve eğer eşit iseler, her iki listenin geriye kalan elemanları karşılaştırılan eleman sonuç listesine sondan eklenerek, eğer biri diğerinden küçük ise, küçük olan listenin geri kalanı, diğer listenin tamamı ve sonuç listesi değiştirilmeden prosedür özyinelemeli olarak çağrılır.

```
1: procedure INTERSECT(l1, l2)
2:     return intersectRec(l1, l2, emptyList)
3: end procedure
4: procedure INTERSECTREC(l1, l2, r)
5:     if l1 == empty or l2 == empty then
6:         return r
7:     if l1.head.docId == l2.head.docId then
8:         return intersectRec(l1.tail, l2.tail, r.append(l1.head))
9:     if l1.head.docId < l2.head.docId then
10:        return intersectRec(l1.tail, l2, r)
11:    return intersectRec(l1, l2.tail, r)
12: end procedure
```

---

or (veya) işleci komşuluk listelerinin birleşim kümesinin hesaplanmasını sağlar.

```
(or (friend:5) (friend:6))
```

5 ve 6 ID'li kullanıcıların arkadaşları.

weak-and (zayıf-ve) işlecinin *ve* işlecinden tek farkı seçmeli sayı ve ağırlık parametrelerini kullanıyor olmasıdır. Bu parametreler sayesinde, *zayıf-ve* işlecinin sonuç listesinde olabilmek için parametre eklenmiş olan işlecin sonuç listesinin bu işlecin karşılaştırıldığı işlecin sonuç listesinin tamamı ile kesişmesi gerekmemektedir. Yani sıra değeri yüksek olan *Hit*'lerin belli bir kısmında kesişme koşulu aranmayabilir.

```
(weak-and (friend:5 :opt-weight 0.1) (term "michael"))
```

5 ID'li kullanıcının Michael isimli arkadaşları. (term "michael") sonuç listesindeki Hit'lerin sonuca yansıması için, (friend:5) sonuç listesindeki Hit'lerin yüzde 10'u ile kesişmesi gerekmez.

strong-or (güçlü-veya) işleci kırpma sonucu tamamen kaybolabilecek düşük sıra değerlerine sahip bir işleç sonucunun, sonuç listesinin bir kısmını kesinlikle oluşturmasını garanti eder.

```
(strong-or (live-in:100) (live-in:101 :opt-weight 0.1))
```

101 ID'li şehirde yaşayan kullanıcıların, sonuç listesinin yüzde 10'unda bulunması garanti edilmiştir.

apply (uygula) işleci sosyal çizgede bir kenardan öteye ulaşılmasını sağlar. apply işleci gerçekleştirimi algoritma 2'de verilmiştir.

```
(apply friend: (friend:5))
```

5 ID'li kullanıcının arkadaşlarının arkadaşları. (friend:5) işlecinden gelen listenin her bir elemanının arkadaşlarının bileşim kümesi sonuç olur.

Algorithm 2 Apply algoritması bir kenar-tipi ve işleci parametre olarak alır. Parametre olan işleç çalıştırılır ve sonuç listesindeki bütün *Hit*'lerin ID'leri ve parametre olan kenar-tipi kullanılarak yeni *term* işleçleri, belirlenmiş olan bir limit sayısı kadar oluşturulup bir listeye atılır. Daha sonra bu liste bir *or* işlecine parametre olacak şekilde verilir ve çalıştırılıp sonuç olarak döndürülür.

```
1: procedure APPLY(eType, operator)
2:     applyLimit ← 50
3:     rAdjList ← operator.execute()
4:     termList ← emptyList
5:     for i ← 0; i < applyLimit do
6:         id ← rAdjList[i].docId.id
7:         termList.append(Term(eType, id))
8:     end for
9:     return Or(termList).execute()
10: end procedure
```

Ters Dizin Tasarımı

Ters dizinler belli bir karakter dizisini içeren düğümlere erişilmesini sağlar. Bu indisler yolu ile ismi "Carl" olan kullanıcılar veya kategorisi "fast-food" olan restoranlar hızlı bir şekilde bulunabilir. Ters dizinlerin arkasındaki düşünce sosyal çizgeyi indislerken kullanılanla aynıdır. Tek fark indisin kenar tipi-ID ikilisi değil, bir karakter dizisi olmasıdır. İndislenmiş olan liste yine *Hit*'lerden oluşur ve *Hit*'lerin indisleyen karakter dizisini içerdiği kesindir. Standart bilgi elde etme sistemlerinde sözcüklerin indislediği listelere *posting list* denir. Yani bu araştırma kapsamında, sosyal ağ alanındaki *posting list*'lerin komşuluk listesinden veri yapısı olarak bir farkı yoktur. Böylece, bir ters dizinin veya çizge dizininin sonuç listesi ele alındığında bu listeler karşılaştırılabilir. Başka bir

deyişle, gerçekleştirilmiş olan işleçler iki dizinin sonuç listesini fark gözetmeksizin işleyebilir.

Ters dizinlerin işleçlerde kullanılabilmesi için *term* işleci karakter dizilerini de parametre olarak alabilecek şekilde güncellenmiştir.

```
(term "michael")
```

"Michael" karakter dizisi ile eşleşen düğümler.

Typeahead arama yapılabilmesini sağlamak için, indisleme şeması ilk karakterden başlayarak ilerlemektedir. Örneğin, ismi "Carl" olan bir kullanıcı için, "C", "Ca", "Car" ve "Carl" şeklinde farklı indisler üretilmiştir. Bu sayede, ilk karakterden başlanarak "C" veya "Ca" dizgilerinin indisinde "Carl" ismindeki düğümlere erişilebilir. *Term* işlecinin karakter dizisi parametresinin sonundaki "*" karakteri ise yapılacak aramanın typeahead araması olduğunu belirtmektedir.

Çözüm Yaklaşımına Dair Önemli Hususlar

Gerçekleştirilen sosyal çizge arama motorunda sistemin etkili ve etkin işleyişi için bazı hususların dikkate alınması gerekmektedir. Bu hususlar aşağıda verilmektedir:

- Veriyi Sıralı Tutma.
- İşleçlerin Yerellik Gereksinimi.
- İşleçlerin Birleşme Özelliği.
- Typeahead ve Standart Aramanın Sonuçlarını Ayırma.

Her birinin ele alınıp gerçekleştiriminde bazı ödünleşimler görülmüş ve bu ödünleşimlere ilişkin karar verilmiştir.

Veriyi Sıralı Tutma

Sistemde komşuluk listesinde tutulan veriler sıralıdır. Tüm sosyal çizge verisine ait komşuluk listesi dağıtılarak işlenmektedir. Bu dağıtım esnasında liste belli bölümlerinden kırpılmaktadır ve bu kırpma sonucunda dağıtılan parçaların yine sıralı tutulması önem arz etmektedir. Sistemimizdeki temel *term* işleci işletildiğinde komşuluk listesi dönmektedir. Gereksinim duyulan diğer sorguları karşılamak üzere Spark'ın *intersect* ve *union* gibi hazır fonksiyonları kullanılmak istendiğinde komşuluk listelerinin eleman seviyesinde dağıtılması gerekmektedir ancak eleman seviyesinde yapılan dağıtımda her işlem sonrası Spark'ta elemanlar karıştırılabileceğinden mevcut sıranın kaybedilmesi sorunu ortaya çıkmaktadır. Her işlemden sonra sıralama yapılması bütün dağıtık sistem üzerinde veri transferini yoğunlaştırır ve bu sıralama işlemi dağıtık sistemlerde en çok karşılaşılan darboğazlardan biridir. Dolayısıyla komşuluk listeleri seviyesinde dağıtım yapılmalıdır ve bu sayede kendi içlerinde sıralı olan listelerin yine sıralı bir şekilde birleştirilmesi etkin olarak tamamlanabilmektedir. Bu tasarıma uygun olarak Spark'ın *intersect* ve *union* işleçlerinin kullanımı yerine bu işleçler bizzat kodlanmıştır.

İşleçlerin Yerellik Gereksinimi

İşleçlerin çoğu hesaplamanın doğru yapılabilmesi için girdi listelerinin aynı makine üzerinde tutulmasına gereksinim duymaktadır. Örneğin aşağıdaki sorgu incelendiğinde;

```
(and (friend:5) (friend:6))
```

5 ve 6 ID'li kullanıcıların arkadaş listelerinin kesişim kümesinin bulunabilmesi için sistem üzerinde dağıtılmış olan bu iki listenin aynı makine üzerinde bulunması gerekmektedir. Aksi halde sonuç listesi doğru olmayacaktır. Bu nedenle *and*, *or*, *weak-and* ve *strong-or* işleçleri çalıştırılmadan önce bütün komşuluk listelerinin aynı makinede birleştirilmesi bir çözüm olabilir ancak böyle bir yaklaşım tek ve belki de uzun bir listenin tek bir makine üzerinde toplanması durumunu ortaya çıkarır ve bu durum bundan sonra liste üzerinde yapılacak olan işlemlerin paralellik özelliğini kaybetmesine neden olur. Bu problemin aşılması için yerellik ihtiyacı olan işleçler aşağıdaki adımlar izlenerek çalıştırılmaktadır:

- Parametre işleçlerinin sonuçları aynı makine üzerinde toplanır, sonuç hesaplanır ve eğer gerekli ise kırpma işlemi uygulanır. Bu aşama ile sonucun ne olması gerektiği öğrenilmiş olur.
- Parametre işleçlerinin sonuçları makine bazlı olarak birleştirilir ve ilk aşamadan öğrenilmiş olan sonuç listesi bunlara katılır. Bu aşamanın amacı dağıtıklığı sağlamaktır.
- Son olarak, elde edilen birleştirilmiş liste ve öğrenilmiş sonuç listesi kullanılarak birleştirilmiş listedeki öğrenilmiş sonuç listesinde bulunmayan elemanlar elimine edilir. Bu sayede hem hesaplama doğru yapılmış hem de dağıtık bir sonuç elde edilmiş olur.

Aşağıda verilen örnekte 5 ve 6 ID'li kullanıcıların arkadaşları P_1, P_2, P_3 parçalarına dağıtılmıştır ve bu kullanıcıların ortak arkadaşlarının sistematik bir şekilde nasıl hesaplandığı gösterilmiştir.

Verilen örnekte kırpma limiti 3 olarak kabul edilmiştir ve sonuç listesinin $0, 35, 57$ olması gerekmektedir.

Başlangıç Aşaması

(term friend:5)	**(term friend:6)**
$P_1 \rightarrow [0, 35, 86, 96]$	$P1 \rightarrow [4, 22, 57]$
$P_2 \rightarrow [10, 57, 66, 94]$	$P2 \rightarrow [0, 23, 82, 94, 97]$
$P_3 \rightarrow [75, 76, 97]$	$P3 \rightarrow [2, 35, 49]$

Her iki *term* sonuç listesi parçalar üzerinde birleştirilir ve aynı parça üzerinde toplanır.

(term friend:5)	**(term friend:6)**
$P_1 \rightarrow [0, 10, 35, 57, 66, 75, 76, 86, 94, 96, 97]$	$P_1 \rightarrow [0, 2, 4, 22, 23, 35, 49, 57, 82, 94, 97]$

Daha sonra oluşan iki listeye algoritma 1 uygulanır ve gerekli kırpma yapılır.

$$P_1 \rightarrow [0, 35, 57]$$

Elde edilmiş olan liste sonuç olması gereken fakat dağıtık olmayan öğrenilmiş listedir. Daha sonra **başlangıç aşaması**'ndaki *term*'ler parça seviyesinde birleştirilir yani aynı parçada olan listeler birleşirler.

$$\begin{aligned} P_1 &\rightarrow [0, 4, 22, 35, 57, 86, 96] \\ P_2 &\rightarrow [0, 23, 10, 57, 66, 82, 94, 97] \\ P_3 &\rightarrow [2, 35, 49, 75, 76, 97] \end{aligned}$$

Öğrenilmiş liste bütün parçalara katılır.

$$\begin{aligned} P_1 &\rightarrow [0, 4, 22, 35, 57, 86, 96] & P_1 &\rightarrow [0, 35, 57] \\ P_2 &\rightarrow [0, 23, 10, 57, 66, 82, 94, 97] & P_2 &\rightarrow [0, 35, 57] \\ P_3 &\rightarrow [2, 35, 49, 75, 76, 97] & P_3 &\rightarrow [0, 35, 57] \end{aligned}$$

Son olarak birleştirilmiş olan listelerde öğrenilmiş listede olmayan sonuçlar elimine edilir.

Sonuç Aşaması

$$\begin{aligned} P_1 &\rightarrow [0, 35, 57] \\ P_2 &\rightarrow [0, 57] \\ P_3 &\rightarrow [35] \end{aligned}$$

Sonuç olarak doğru hesaplanmış dağıtık bir sonuç elde edilmiş olur. Sonuç listesi tekrarlı elemanlar içerebilir fakat eğer bu **sonuç aşaması** daha sonraki başka bir işlece parametre ise o işlecin hesaplanmasında bir soruna yol açmaz, eğer değil ise sonuç döndürülürken tekrarlı elemanlar elimine edilir.

İşleçlerin Birleşme özelliği

And ve *or* işleçleri birleşme özelliği gösterir. Yani aşağıda verilen iki sorgu birbirleri ile eşdeğerdir:

- `(and (friend:5) (friend:6) (friend:12))`
- `(and (friend:5) (and (friend:6) (friend:12)))`

Bu özellik kullanılarak *and* ve *or* işleçleri n tane işlenenle çalışacak şekilde gerçekleştirilebilir. Buna karşın *weak-and* ve *strong-or* işleçleri birleşme özelliği göstermez. Dolayısıyla aşağıda verilen iki sorgu birbirine eşit değildir:

```
- (weak-and (friend:5 :opt-count 1) (friend:6 :opt-weight
    0.1) (friend:12))

- (weak-and (friend:5 :opt-count 1) (weak-and (friend:6 :
    opt-weight 0.1) (friend:12)))
```

Bu sebeple, çalışmada *weak-and* ve *strong-or* işleçleri, sadece 2 işlenen ile çalışabilecek şekilde gerçekleştirilmiştir. Bunun yanı sıra, bu işleçler Spark'ın *cogroup* fonksiyonu kullanılarak kodlanmıştır ve bu fonksiyon 5 işlenene kadar destek verir. Gerekli değişiklikler ile işleçlerin 5 işlenene kadar çalışması mümkündür.

Typeahead ve Standart Aramanın Sonuçlarını Ayırma

Sistem typeahead ve standart olmak üzere iki tip aramayı desteklemektedir. *term* işlecine parametre olarak verilen karakter dizisinin * (yıldız işareti) ile bitmesi durumunda typeahead arama sorgusu ifade edilmektedir:

```
(term "Carl")

(term "Carl*")
```

Mevcut problemi aktarmak için örnek bir karakter dizisi ele alınabilir. Örneğin içinde "Carl" karakter dizisini barındıran bir düğüm için ters dizin yapılandırılırken düğüm "C", "Ca", "Car" ve "Carl" karakter dizileri ile indislendiği için standart ve typeahead bir aramanın farkı ayırt edilemez. Örneğin, "Carl" karakter dizisinin aranması sonucu elde edilen *Hit* listesindeki *Hit*'lerin carl sözcüğüne birebir eşit mi yoksa "Carl" karakter dizisi ile başlayan bir sözcük mü ("Carlos") olduğu bilinemez:

$$Carl \rightarrow (Hit_1, Hit_2, Hit_3, Hit_4)$$

Probleme çözüm olarak, seçmeli *HitData* bayt dizini kullanılarak, eğer listedeki herhangi bir *Hit* sözcük ile tamamen eşleşiyorsa, dizinin ilk elemanına 1, ya da sadece, ilgili *Hit* sözcük ile başlıyorsa 0 atanır. Böylece, eğer sorgu typeahead ise indisteki bütün liste sonuç olarak döndürülür. Aksi takdirde, standart bir arama, listenin bayt dizini 1 ile başlayan sonuçlarını getirir.

Benzer çalışmalar

Unicorn diğer arama motorlarından farklı olarak doküman tabanlı arama yapmak yerine sosyal ağda yer alan varlıklar, bu varlıklara ilişkin bilgiler ve bu varlıkların ilişkili oldukları diğer varlıklar altyapısı üzerinde arama gerçekleştirmektedir.

Bu kapsamda, benzer bir çalışma olarak Aardvark[6] arama motoru örnek gösterilebilir. Aardvark, verilen soruyu cevaplayabilecek en uygun kişiyi bulmayı amaçlamaktadır. Bu sebeple, uygulama alanı sosyal ağlar olup sonuçlar insanlar hakkındaki bilgileri içerir.

Bu sosyal çizge veri yapısı ele alındığında, bir arama motoru olmamasına rağmen SPARQL sorgu dili benzer bir çalışma olarak değerlendirilebilir. SPARQL sorgu dili, RDF çizge veri kümeleri için tasarlanmıstır. Unicorn'a benzer olarak anlambilimsel farklılıkları gözeterek, RDF veri kümelerindeki veriye erişmeyi sağlar. Fakat iki sistem arasındaki fark RDF verisindeki ilişkilerin *object-predicate-subject* üçlü'leri (*triplet*) kullanılarak, Unicorn altyapısında ise daha önce bahsedildiği gibi komşuluk listeleri aracılığıyla, sıralı bir şekilde gösterilmesidir.

Gerçekleştirilen çalışma, sosyal ağlar üzerinde arama yapmak üzere özelleşmiştir. Bu özelleşmeyi, klasik bilgi elde etme sistemlerinde veya arama motorlarında kullanılan ters dizin yapısı ile anlambilimsel çizgeler üzerinde tanımlanan sorgulama dili ve motorlarını entegre etmesi ile sağlamıştır. Bu entegrasyonda veri yapısı olarak komşuluk listelerinin kullanımı ve liste veri yapısına çok uygun fonksiyonel programlama işleçlerinin uyarlanması genel ve etkin bir yapı kurulmasını sağlamıştır.

Sonuç

Bu çalışma kapsamında Unicorn çizge arama motorunun temel bileşenleri analiz edilip gerçekleştirilmiştir.

Çalışmanın temel katkıları şu şekilde özetlenebilir:

- Standart bilgi elde etme kavramlarının sosyal ağlar alanına uygulanması konusunda bilgi birikimi oluşturulması.
- Karmaşık çizge tabanlı bir arama motorunun tasarımındaki hususlar ve bu hususlar kapsamındaki ödünleşimleri de dikkate alarak bir çözüm üretilmesi.
- Sistemdeki işleçlerin Spark üzerinde dağıtık ve paralel bir şekilde hesaplanabilmesinin sağlanması.

Komşuluk listesinde sıralama ölçütü olarak PageRank değerinin kullanılması, işleçler için Composite Design Pattern'in kullanılması, işleç algoritmalarının Spark ortamına uyarlanması, Typeahead arama özelliğinin eklenmesi yukarıda listelenen katkılara eklenti olarak değerlendirilebilecek çözüme dair özel niteliklerdir.

Gelecek Çalışmalar

Bu çalışmada Unicorn'un çekirdek yapısı Spark üzerinde yaklaşık olarak 800 kod satırı ile gerçekleştirilmiştir. Geliştirilen yazılımın etkinliğinin işleçler üzerinde yapılacak kapsamlı performans ölçümleri ile test edilmesi uygun olacaktır. Bu sayede paralel ve dağıtık bir yapıda sistemin performansı değerlendirilmiş olacaktır.

Unicorn gibi büyük ölçekli çizge tabanlı bir arama motorunun gerçek anlamda yapılandırılması için çözüme aşağıdaki eklentilerin dahil edilmesi de faydalı olacaktır:

– Bellek kullanımını verimli hale getirmek için, ters dizinlere ve komşuluk listelerine sıkıştırma algoritmaları uygulanması.
– Arama yapan kullanıcıya göre sıralama (ranking) yapılması. Arama yapan kullanıcıyla ortak veriye sahip olan *Hit*'lerin sıra değerleri orantılı olarak arttırılabilir. Örneğin, arama yapan kullanıcı "X" üniversitesinden mezun ise komşuluk listelerinde bu üniversiteden mezun olmuş *Hit*'lerin sıra değerleri arttırılabilir. Zira sosyal ağlarda ortak altgruplara aidiyetin kişilerin ilişkilendirmesinde faydalı olduğu görülmüştür [7]. Sıralamayı etkileyecek olan veriler seçmeli *HitData* bayt dizininde saklanabilir. Bu sayede, hem *PageRank* algoritmasına dayalı prestije bağlı sıralama hem de içeriğe bağlı sıralama melez bir yaklaşım olarak uygulanabilir.

Etki

İnternet ve sosyal ağlar hala kullanıcı sayılarını arttırmakta ve büyük veriler oluşmaktadır. Yakın zamanda, bağlantılı büyük veri üzerinde arama yapma probleminin sadece Facebook'un değil, diğer orta ölçekli pek çok kuruluşun ortak sorunu haline gelmesi beklenmektedir. Dolayısıyla bu çalışmanın yaklaşımı ve tartıştığı kavramlar, yeni yapılandırılması planlanan sistemlere uygulanabilir. Ayrıca, çizgenin gerçekleştirimi genel (generic) olduğu için, gerekli düğüm ve kenar tipleri tanımlanıp gereksinim duyulan alternatif sıralama stratejileri geliştirilirse sosyal çizge arama motoru herhangi bir alana (DNA veritabanları) uygulanabilir.

Kaynaklar

1. Michael Curtiss, Iain Becker, Tudor Bosman, Sergey Doroshenko, Lucian Grijincu, Tom Jackson, Sandhya Kunnatur, Soren Lassen, Philip Pronin, Sriram Sankar, Guanghao Shen, Gintaras Woss, Chao Yang, and Ning Zhang. Unicorn: A system for searching the social graph. *The 39th International Conference on Very Large Data Bases*, 2013.
2. Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. *HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010.
3. Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.
4. Sergey Brin and Larry Page. The pagerank citation ranking : Bringing order to the web. 1998.
5. Yelp. `https://www.yelp.com/dataset_challenge`.
6. Damon Horowitz and Sepandar D. Kamvar. The anatomy of a large-scale social search engine. *WWW '10 Proceedings of the 19th international conference on World wide web*, pages 431–440, 2010.
7. Jaewon Yang and Jure Leskovec. Overlapping community detection at scale: A nonnegative matrix factorization approach. In *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*, WSDM '13, pages 587–596, New York, NY, USA, 2013. ACM.